- Glossary
- Chapter 1: The AI-Powered DevOps Engineer
  - Foundations of Effective AI Usage — and the Road to Software Orchestration
  - 1. The Problem This Book Addresses
  - 2. Your Role as Orchestrator
  - 3. The Mental Model: How AI Thinks
  - 4. The Three Structures That Control Quality
  - 5. How This Book Is Organized
- Chapter 2: The Self-Improving Repository
  - Building the Repository That Holds Your Three Structures
  - 1. The Shape of the Repository
  - 2. The Intelligence Layer: Law and Map
  - 3. Context Routing: Progressive Disclosure and Reference Mapping
  - 4. Session Continuity: The Gitignored Session Artefact
  - 5. Workflow Orchestration: Safe Multi-Step Composition
  - 6. Escalation Policy: Enforcing Human Authority
  - 7. Context Economy: Context Budgeting by Task Scope
  - 8. Execution Transparency: The Context and Action Summary
  - 9. The Command Surface Contract
  - 10. The Validation Gates
  - 11. Documents Are Artefacts Too — the consumer you didn't know you had
  - 12. The Self-Improving Loop at Org Scale
  - 13. AI Agents in the Governed System
  - 14. Synthesis: The Repository as the Ultimate Force Multiplier
  - 15. Putting It Into Practice
  - 16. 🔑 Chapter Summary
  - 17. Where This Leads: From One Repository to Many
- Chapter 3: The Language of Control
  - Vocabulary, Structure, and AI-Readable Architecture
  - 1. The Constraint You Cannot Name, You Cannot Request
  - 2. The Working Vocabulary
  - 3. The Shape of the Code: Context by Design
  - 4. Enforcement Is the Language's Grammar
  - 5. Build the Language
  - 6. 🔑 Chapter Summary
- Chapter 4: The Command Surface
  - A Thin Layer Over the Tools You Already Drive
  - 1. Why the Repository Needs a Command Surface
  - 2. The Gap Test
  - 3. The Structure
  - 4. The Commands Worth Knowing — Version Control
  - 5. The Commands Worth Knowing — Workflow
  - 6. The Build Order
  - 7. Seeing Widely, Acting Narrowly
  - 8. Build the Workbench
  - 9. 🔑 Chapter Summary
- Chapter 5: Context Routing
  - Choosing What the Agent Reads
  - 1. Context Is Finite and Costly
  - 2. Load Units: Organising Context by Task Need
  - 3. The Map Is a Fact Document
  - 4. The Routing Table Is a Rule
  - 5. Verification — What Was Actually Read
  - 6. Failure Modes
  - 7. Build the Routing Table
  - 8. 🔑 Chapter Summary
- Chapter 6: Context Economy
  - Where the Money Actually Goes
  - 1. Two Budgets, One Decision
  - 2. The Always-Loaded Set
  - 3. Compaction, Summary, Cache
  - 4. Recording Spend
  - 5. Build the Budget
  - 6. 🔑 Chapter Summary
- Chapter 7: Session Continuity
  - The Handoff Artefact: What the Repository Cannot Tell You
  - 1. Memory Is Not a Control
  - 2. The Artefact Is a Contract
  - 3. The Part the Repository Cannot Tell You
  - 4. Cross-Tool, Mid-Task, and Crash
  - 5. Build the Handoff
  - 6. 🔑 Chapter Summary

- Chapter 8: The Agent's Procedure
  - Orchestrating a Multi-Step Change
  - 1. Why Sequences Break
  - 2. Procedure Anatomy
  - 3. When Not to Write a Procedure
  - 4. The Play, the Engine and the Board
  - 5. Build the Play
  - 6. 🔑 Chapter Summary
- Chapter 9: Escalation Policy
  - Where an Agent's Authority Ends
  - 1. Permissions Are Not Judgement
  - 2. The Conditions Catalogue
  - 3. The Stop-and-Report Contract
  - 4. The Record, and How a Decision Returns
  - 5. Build the Policy
  - 6. 🔑 Chapter Summary
- Chapter 10: Execution Transparency
  - The Log Is the Evidence
  - 1. Execution Transparency
  - 2. The Records Around It
  - 3. The Log Entry
  - 4. The Log and the Gates Must Agree
  - 5. Provenance, and What Review Becomes
  - 6. Build the Log
  - 7. 🔑 Chapter Summary
- Chapter 11: Build, Buy, or Assemble
  - The Decision Record and Its Exit Plan
  - 1. The Decision Made on an Invoice
  - 2. Assemble as the Default
  - 3. The Exit Plan
  - 4. How Agents Move the Calculus
  - 5. Build the Decision Record
  - 6. 🔑 Chapter Summary
- Chapter 12: Commodity From a Provider
  - Control Where the Value Is
  - 1. What to Buy, and What Never to Buy
  - 2. The Seam Contract
  - 3. Idempotency, Retries and Ordering
  - 4. Human Gates Inside the Pipeline
  - 5. The New Risk Class: Untrusted Content and Agents
  - 6. Build the Workbench
  - 7. 🔑 Chapter Summary
- Chapter 13: Upstream Product as a Dependency
  - Wrapping, Patching, and the Upgrade Rehearsal
  - 1. The Three Ways, and Two Are Wrong
  - 2. Wrapping, Not Vendoring
  - 3. Patch-Only Forks
  - 4. The Upgrade Rehearsal
  - 5. Upstream Governance
  - 6. Build the Wrapper
  - 7. 🔑 Chapter Summary
- Chapter 14: The Integration Charter
  - Nine Decisions Before You Connect to Anything You Do Not Own
  - 1. Every Integration Arrives With Its Defaults Already Chosen
  - 2. The Nine Decisions
  - 3. Precedence: The Decision That Argues Back
  - 4. Four Gates Between a Resource and a Caller
  - 5. The Charter Is a File, Not a Document
  - 6. What the Agent Prepares, What the Human Signs
  - 7. Something You Can Check
  - 8. Build the Charter
  - 9. Closing: Portable by Construction
- Chapter 15: Contracts That Keep Services Independent
  - Shared Infrastructure Without Shared Fate
  - 1. Independence Is a Contract, Not a Boundary
  - 2. Per-Service Ownership
  - 3. Expand/Contract
  - 4. Least Privilege and Contract Tests
  - 5. Build the Workbench
  - 6. 🔑 Chapter Summary
- Chapter 16: Sovereignty, Portability and Exit Drills
  - Proving You Could Leave
  - 1. Sovereignty Is a Requirement

    - 2. Portability as a Gate
    - 3. Exit Drills
    - 4. Provenance and SBOM
    - 5. Build the Workbench
    - 6. 🔑 Chapter Summary
- Chapter 17: Observability as the Evidence Layer
    - Signals Bound to Invariants
    - 1. A Signal Is a Claim About an Invariant
    - 2. One Check Logic, Two Consumers
    - 3. SLOs, Budgets, and Two Kinds of Health
    - 4. Machine-Consumable Evidence
    - 5. Agent Observability
    - 6. Build the Workbench
    - 7. 🔑 Chapter Summary
- Chapter 18: Recoverability
    - The Restore You Have Actually Run
    - 1. What “Backup” Means Now
    - 2. Retention as Policy
    - 3. Key Custody
    - 4. Restore Drills as a Signal
    - 5. What Not to Back Up
    - 6. Build the Workbench
    - 7. 🔑 Chapter Summary
- Chapter 19: Engineering the Quality Gate
    - From a Pile of Checks to One Gate Estate
    - 1. The Pile You Have
    - 2. A Gate Is a Contract, and It Declares Five Things
    - 3. A Gate Has a Name, a Moment and an Environment
    - 4. The Workflow, and the Gates Between Its Stages
    - 5. The Deployment Gate: the Level That Fails Your Pushes
    - 6. How the Gates Compose
    - 7. What the Set Pays Back
    - 8. How It Is Implemented
    - 9. Four Anti-Patterns the Design Prevents
    - 10. Build the Workbench
    - 11. 🔑 Chapter Summary
- Chapter 20: From One System to a Portfolio
    - The Manifest as the Answer
    - 1. What Changes at Scale
    - 2. The Source-of-Truth Model
    - 3. The Lifecycle Table
    - 4. The AI Delta
    - 5. Build the Delivery
    - 6. 🔑 Chapter Summary
- Chapter 21: Version Control as the Substrate
    - Everything That Matters Is a Versioned File
    - 1. The Repository Is the Unit of Governance
    - 2. Conventions That Make History Readable
    - 3. History as the First Diagnostic
    - 4. The AI Delta
    - 5. Build the Delivery
    - 6. 🔑 Chapter Summary
- Chapter 22: The Control Plane as a Product
    - Validators, Audits, and the Golden Path
    - 1. A Platform Has Users
    - 2. The Catalog Is Generated
    - 3. The Golden Path
    - 4. The AI Delta
    - 5. Build the Delivery
    - 6. 🔑 Chapter Summary
- Chapter 23: Versioning, Artefacts and Contracts
    - A Number That Carries a Promise
    - 1. What a Version Promises
    - 2. Compatibility Is a Check, Not a Virtue
    - 3. Deprecation Has a Clock
    - 4. Version Skew
    - 5. Build the Delivery
    - 6. 🔑 Chapter Summary
- Chapter 24: Release Management and the Evidence Pack
    - A Reviewer Should Not Need to Ask You Anything
    - 1. What a Release Is
    - 2. Generated, Not Authored

    - 3. Release Trains and Cadence
    - 4. The AI Delta
    - 5. Build the Delivery
    - 6. 🔑 Chapter Summary
- Chapter 25: Deploy, Verify, Roll Back
    - The Way Back Is Part of the Path
    - 1. The Deploy Is Not the Release
    - 2. Verify in Production
    - 3. Rollback, Roll-Forward, and the Case With Neither
    - 4. The AI Delta
    - 5. Build the Delivery
    - 6. 🔑 Chapter Summary
- Chapter 26: Governance Machines Enforce
    - Policy as a File, Enforcement as a Gate
    - 1. Policy That Cannot Merge
    - 2. Codified Standards Across Repositories
    - 3. Access Tiers and Agent Identity
    - 4. The AI Delta
    - 5. Build the Delivery
    - 6. 🔑 Chapter Summary
- Chapter 27: The Agent Plane at Scale
    - Where the Actor Is the Subject
    - 1. Prose Is Not Enforcement
    - 2. Hooks and Deterministic Enforcement
    - 3. Evals in CI
    - 4. Agent Observability and Handoffs
    - 5. The AI Delta
    - 6. Build the Delivery
    - 7. 🔑 Chapter Summary
- Chapter 28: Portfolio Observability and Operations
    - One Surface, Many Services
    - 1. One Surface, Never a Parallel Truth
    - 2. Budgets, Degradation and Two Kinds of Health
    - 3. Incidents Sized for One Operator
    - 4. The Copy Behind the Surface
    - 5. The AI Delta
    - 6. Build the Delivery
    - 7. 🔑 Chapter Summary
- Chapter 29: The Fleet
    - Cattle, Identity, and Consented Support
    - 1. Machines Are Not Pets
    - 2. Enrolment and Identity
    - 3. Consented Support
    - 4. Degradation
    - 5. The AI Delta
    - 6. Build the Delivery
    - 7. 🔑 Chapter Summary
- Chapter 30: Measurement and Economics
    - Six Numbers, or an Honest Gap
    - 1. What to Measure
    - 2. The Bottleneck Is Review
    - 3. The Economics
    - 4. The AI Delta
    - 5. Build the Delivery
    - 6. 🔑 Chapter Summary
- Conclusion: What This Buys, and What It Costs
    - The Catalogue, Walked Honestly
    - 1. The Arc, in One Page
    - 2. The Catalogue, Walked
    - 3. The AI-Age Claim, Stated So It Survives
    - 4. The Cost, Plainly
    - 5. Monday
    - 6. The Last Thing
    - What this book is
    - The four parts
    - How to read it
- The Implementation Pack
    - What it is not

Download edition · 2026-11-18 · commit `72fed92`

The same content as the online book at https://ocooee.com/book, assembled in reading order. Everything that teaches method is free; the Implementation Pack and the engagement are the paid layers.

Glossary

- **actor** — whoever or whatever performed an action: a person or an agent. The escalation log records the actor alongside the decision, so a decision can be attributed rather than guessed. *See also* AI agent.
- **agent plane** — the layer where the agent itself is the thing that must be controlled, not the context it reads. Telling an agent never to push to main is an instruction; a check that refuses the push is a control, and this layer is where those controls live — the required check, the sandbox, the hook that refuses the call. *See also* AI agent.
- **AI agent** — an AI system that acts on a repository rather than only answering: it reads files, edits them and runs commands. It is not a person, and it is not the machine it runs on; what it did is recorded, and what it may do is limited. *See also* agent plane.
- **architecture** — the structure of a system: its parts, the boundaries between them, and the rules about what may depend on what. It is what a change has to fit, which is why a change that ignores it shows up as drift. *See also* drift.
- **backup** — a copy of data kept so that it can be restored. It is not a recovery: a copy nobody has restored is a hypothesis, and a copy that lives in the same account as the data is not even a copy. *See also* restore.
- **blast radius** — how much a failed change can break: one service, one environment, or the whole estate. Progressive delivery is chosen by blast radius, not by fashion. *See also* canary.
- **blue/green** — a rollout that runs the new version beside the old one and swaps traffic to it atomically. It costs double the capacity for a short time; that cost is why it is chosen for changes whose blast radius justifies it. *See also* blast radius.
- **build artefact** — a file a build produces and a later job consumes: a binary, an image, a report. GitHub spells it *artifact*. *See also* release.
- **canary** — a rollout that sends a small share of traffic to the new version and watches a signal before widening it. A canary with no signal that can judge it is a delay, not a test. *See also* blast radius.
- **capability** — what a system can do for the person using it, stated as an outcome that can be proved: *this system can publish a chapter to staging, and here is the evidence*. A capability is larger than a feature — several features may be needed before one capability is true. *See also* feature.
- **catalogue** — the list of what exists, each item with its state and the evidence for that state. A capability catalogue says what the system can do and how each claim is proved; a feature catalogue says what is planned, in flight and shipped, and what proves each. A row that cannot cite its evidence is not yet true. *See also* capability.
- **CD** — continuous delivery: every passing change is releasable at any time. *Continuous deployment* goes one step further and releases it automatically; delivery keeps the release a decision. *See also* release.
- **checkpoint** — a measurement taken at a named point in a piece of work, before the work continues: state what must be true, compute what is actually there, compare, and refuse on the difference. It is not a gate — a gate is a checkpoint given a name, a moment and an environment, and wired to stop the change. Every gate began as a checkpoint someone took by hand. *See also* gate.
- **CI** — continuous integration: every change is built and checked automatically, and the result is visible. Its point is that a break is found in minutes, not in a release — the commit and full moments are CI. *See also* gate tier.
- **command surface** — the part of a tool a person and a script share: the operations it offers, the words and arguments that select them, and what each returns — text, and an exit code that says success, failure or refusal. It is the contract, not the window it is typed into. The industry calls the exposed operation set an *API surface*; on a command line it is the CLI's commands. *See also* command-line interface.
- **command-line interface** — the typed form of a command surface: you type an operation, its arguments, and read what comes back. CLI is the industry's name for it, and it names the interface only — the commands and their grammar — not the screen, and not the panels a person looks at afterwards. *See also* command surface.
- **commodity** — the part of a system nobody differentiates on, such as transport, storage or mail delivery. You buy it from a provider because the provider competes on it; the parts worth building yourself are the glue and the differentiator — the ones where your behaviour lives. *See also* the glue.
- **console** — an administrative surface: where a person reads a system's state and acts on it. *The console* is the operator's own; *a vendor console* is someone else's, and it is where a setting can live that you cannot diff. *See also* dashboard.
- **contract** — the declared shape two independent parts agree on, so either can change without breaking the other. A request shape, an exit code and a version range are a contract; a change that breaks one is a new version, not a patch. *See also* version.
- **control plane** — the repository that governs other repositories: the rules, the manifest and the gates. It is where policy is written, as distinct from the applications that obey it. *See also* the repository.
- **dashboard** — a view that reports state and cannot act: charts, colours, a number per panel. It answers *what is happening*; a console answers *what do I do about it*. *See also* console.
- **decision record** — the strategic log of a choice: why this capability, which axes decided it, when it is revisited. It exists so the next reader inherits the reasoning instead of arguing the choice again.
- **deploy · verify · roll back** — the three acts of a release: prove the change, prove the system, and design the return before it is needed. A deploy is not finished at upload; it is finished when the check of the deployed system passes. *See also* release.
- **deterministic** — a step whose result is fixed by its input, so the same input gives the same output and the same verdict. A model is not deterministic; the parts around it — a schema, a policy check, the tests — are what decide whether the model's draft is allowed through. *See also* gate.

- **the door** — the identity entrance: who may come in at all. It is not a gate — a gate decides whether a change may proceed, while the door decides whether there is anyone there to ask. *See also* gate.
- **drift** — the distance that grows between what is declared and what is actually running. It appears when configuration is changed by hand, when a document and a repository disagree, or when a check stops being called — and it stays invisible until something compares the two. *See also* manifest.
- **drill** — a restore that was actually run, into an isolated place, with its real duration recorded. A copy nobody restored is a hypothesis; the drill is what turns it into a number you can quote. *See also* recoverability.
- **escalation boundary** — the explicit, checkable division between what an agent may decide and what it must stop for. The conditions are written so the agent can evaluate them before it acts, not after. *See also* AI agent.
- **estate** — everything one person operates: the repositories, services and hosts. It is the whole surface, not the gates, not the tooling, and not one repository. *See also* portfolio.
- **eval** — a scored test of behaviour: a fixed set of cases, run against an agent or a model, with the result compared to a threshold. An eval answers *did the behaviour hold*; it becomes a gate when a failure is wired to stop the build. *See also* gate.
- **evidence pack** — what a release carries so a reviewer can decide without asking a question: what was verified, what was not, and how to go back. A tag without one asks the reviewer to trust instead of check. *See also* release.
- **exit drill** — the rehearsed act of leaving a provider, performed before it is needed. Restoring the data somewhere else is how you learn whether you could leave; nobody rehearses it during the incident. *See also* sovereignty.
- **expand/contract** — the only migration pattern that survives two versions running at once: add the new shape, write both, backfill, then remove the old. Each phase is its own deploy with its own verification, and removing the old shape in the same deploy removes the way back. *See also* version range.
- **feature** — a unit of functionality that delivers value to a user, and the unit a delivery plan tracks through its lifecycle: planned, in progress, shipped. It is smaller than a capability — one capability may need several features before it is true. *See also* capability.
- **feature flag** — a switch that separates behaviour from code, so a change can ship without being on. It is chosen when the blast radius is controllable by turning something off — and it costs a flag you have to remove later. *See also* blast radius.
- **fleet** — many machines treated as replaceable rather than tended: each has an identity — a name and credential that say which machine it is — and a record; an unhealthy machine is replaced, not repaired, and its history is what survives it.
- **gate** — a check a change must pass before it may proceed. It has a name, a moment and an environment, and it is not a report, a dashboard or a notification. *See also* gate estate.
- **gate estate** — all the gates of one organisation, taken as a set. It is a design question — what the set covers, and what it deliberately does not — not a list of checks that accumulated. *See also* gate moment.
- **gate moment** — when a gate fires: **atomic** (any time, one concern), **commit** (as the change is committed), **full** (when the change is complete), **scheduled** (when the clock decides). It carries a budget — the most time it may take before it stops helping. A moment is *when*, never *where*: the environment is a separate question. *See also* gate.
- **the glue** — the integration, policy and approval work that belongs to you rather than to a provider. It is where your behaviour lives, which is why it is the part worth writing yourself. *See also* commodity.
- **governance machine** — a policy applied by a gate, so a violation cannot merge rather than being reported later. The same rule as prose is advice; as a check that fails the merge, it is governance. *See also* invariant.
- **harness** — the program that drives an AI assistant through a task: it routes the work, supplies the context and collects the result. One exchange is not a harness; a harness runs a sequence, which is why it is the thing a gate can be put around. *See also* AI agent.
- **idempotent** — a step that can be run twice and leaves the same result as running it once. It is what makes a setup or a deploy safe to retry, and it separates a command you can run again from one you must not.
- **invariant** — the sentence a gate exists to keep true — *no credential is committed*, *every endpoint has a test*. A gate without one is a check nobody can justify or retire. *See also* gate.
- **job** — a unit of work executed by one runner. A stage holds jobs; the jobs in a stage may run in parallel, and each job is made of steps that run in order. *See also* runner.
- **the Law** — the non-negotiable rules a repository must always follow: what must always be done, and what must never be. It is small and static, and it is written where the people and the agents both read it. *See also* the Map.
- **load unit** — the set of files one task type needs, named once. A migration change loads the schema contract and the migration gate; it does not load the whole engineering chapter. *See also* context routing.
- **manifest** — a declarative file that describes every repository, its tier, owner, lifecycle and policy. It exists so the set of repositories can be read rather than remembered. *See also* control plane.
- **the Map** — the current state of a repository — how it is built, where the conventions are, what an agent needs to read before acting. It is large and task-dependent, which is why it is loaded by unit rather than in full. *See also* the Law.
- **observability** — the evidence layer: every critical invariant — a rule that must always hold — checked by one piece of logic, with two consumers. The same check that fails a deploy also feeds the status surface, so health has no second source of truth. *See also* invariant.

- **operator console** — the console in its acting sense: one surface that reports the state *and* changes it, acting only through the command surface. *See also* console.
- **orchestrator** — the person who directs the work rather than typing every change, and in whom every change to the repository finally rests. The agent may read, parse, suggest and execute; the decision that a change lands is the Software Orchestrator's. *See also* AI agent.
- **pipeline** — one executable definition of automated work: **stages**, each holding **jobs**, each job made of **steps**, run on a **runner** when a **trigger** fires. A pipeline is one artefact inside a workflow — the workflow is the sequence of work, and it includes the human gates between pipelines. *See also* stage.
- **plan** — the written intent for a piece of work: the outcome, the scope, the steps, the validation, and the conditions that would stop it. It is approved before work starts and retired when the work is done — a plan that stays open after the work shipped is stale, not live.
- **the Playbook** — the step-by-step procedures, each with a human approval gate: how exactly to perform a task safely. The Law says what must be true; the Playbook says how, in what order, and where to stop. *See also* the Law.
- **point-in-time recovery** — a restore to a chosen moment rather than to the last copy, so the loss is bounded by the lag you choose. It is a setting on the data provider and a decision by the business — and it lives in the same account as the data, so it complements a copy rather than replacing one. *See also* backup.
- **portfolio** — many systems read as one estate rather than one system at a time. The question changes from *is this system correct?* to *what is the state of the organisation?*, and the answer comes from a curated file rather than from memory. *See also* manifest.
- **precedence** — the order in which settings from different stores are resolved, when more than one of them sets the same thing. It matters because a platform applies an order whether or not anyone has read it.
- **procedure** — the ordered steps of a repeatable change, each with a gate, written as a contract rather than a plan. A failure names the step it happened at because the steps were named in advance. *See also* the Playbook.
- **prompt regression** — a test that re-runs a fixed set of prompts or cases and compares the result with a recorded baseline, so a change that degrades the behaviour fails before release. It is a regression test aimed at a model's behaviour rather than at a function's output. *See also* eval.
- **provenance** — the record of where a change came from: who or what produced it, on top of what, and with whose approval. It turns review from reconstructing authorship into reading intent and evidence. *See also* evidence pack.
- **rebuild** — recreating state from source: the schema from the migrations, the content from the repository, and the index from the content. It rebuilds the *system* and not the *organisation's history* — first-party rows are not in git, which is why a rebuild is not a recovery. *See also* restore.
- **recoverability** — the difference between a copy and a recovery: a restore that has been run. It is measured by how much data you can lose and how long you can be down, not by whether a backup exists. *See also* drill.
- **refusal** — a designed non-action: the command declines, names why, and exits 2. It is not a failure (exit 1) — a refusal is an answer, and a caller can tell it apart from an error. *See also* command surface.
- **registry** — the one table that declares every verb of a command surface. Help, completion and the dispatch all read it, so the surface cannot advertise one thing and implement another. *See also* command surface.
- **release** — a decision with evidence. The deploy is only its mechanism; the release is where the change, its verification and its way back are judged together. *See also* evidence pack.
- **the repository** — the work, the checks, the rules and the context, and the documents that make all of it readable by a person and an agent alike. A repository that holds code but not its gates, rules or context cannot be operated by anyone who did not write it. *See also* the Law.
- **restore** — replacing live data with a copy. It is the most destructive action available — every write after the copy is lost, and nothing in git can reconstruct it — so it is a decision with a written case, never a repair command. *See also* drill.
- **roll back / roll forward / neither** — the three answers to *what if this fails*, written before the release. Roll back when the new version is the problem, roll forward when a fix is faster than a revert, and neither when the change cannot be reversed — decided in advance, never during the incident. *See also* release.
- **RPO / RTO** — how much data you may lose (recovery point objective) and how long you may be down (recovery time objective). Both are decisions, cheap to state and expensive to discover; a backup schedule is a habit, while a stated loss bound is an RPO. *See also* recoverability.
- **runbook** — the operational document: when something breaks, what to do. It is not the Playbook — the Playbook is how to perform a task safely in the first place. *See also* the Playbook.
- **runner** — the machine that executes a job: a hosted machine or a self-hosted one. It is execution capacity and a trust boundary; it decides nothing, it runs what the pipeline or the agent gives it. *See also* job.
- **seam** — the integration point where two systems meet — and the place you design so it can be cut. A provider's interface behind an adapter of your own is a seam; the same interface called from forty places is a dependency you cannot leave. *See also* contract.
- **session continuity** — the discipline of making the handoff between sessions explicit, verifiable and repository-owned, so the next session resumes the work instead of reconstructing it. A session is one continuous stretch of work with an agent — it ends when the conversation or the run ends, and everything the agent remembers ends with it. *See also* procedure.
- **shell** — the program that reads the commands typed into a terminal and runs them. A terminal hosts it; the shell is not the terminal and not the console — it is the layer that interprets what is typed. *See also* terminal.

- **smoke test** — the gate that judges the deployed system rather than its address. Pages answer, the primary interaction returns its contract, the data path returns records — the journeys that matter, not one status code. *See also* gate.
- **sovereignty** — the four facts that prove you are not trapped: where the data lives, who can read it, who can move it, and what happens if the provider says no. Answers you can show, not a feeling about a vendor. *See also* exit drill.
- **stage** — a phase of a pipeline that groups jobs: stages run in sequence, and the jobs inside a stage may run in parallel. It is not a gate moment: a moment is when a gate fires. *See also* gate moment.
- **step** — the smallest unit of work in a pipeline: one command or task inside a job, run in order. The build order's *Step 1–6* uses the ordinary sense: a move in a procedure. *See also* job.
- **terminal** — the view you already have: the text window that carries input and output. It shows state badly — a week of runs scrolls away — which is why the merged console exists. *See also* the view.
- **terminal console** — the console that lives in a terminal: it shows state and acts, in text. *See also* console.
- **the three structures** — the prompt, the engineering and the workflow — the three things that together decide whether AI-assisted work is usable. A good prompt against a repository with no gates still produces a change nobody can verify. *See also* the repository.
- **trigger** — the event that starts a pipeline: a push, a merge request, a schedule, a tag. A deploy or a release is a trigger too: it starts the check that judges the deployed system. *See also* pipeline.
- **upstream product** — a whole application you deploy and configure, whose behaviour, data schema and upgrade path are its own. You wrap it, patch it deliberately and rehearse its upgrade — unlike a commodity, it carries opinions you inherit. *See also* commodity.
- **verb** — one operation on a command surface, declared in the registry: a noun and an action that a person types and a script calls. It is not a wrapper — a wrapper stands for a vendor command, a verb is the operation itself. *See also* registry.
- **version** — a number that carries a promise about compatibility: what a caller may rely on, and what a change is allowed to break. A version is a promise; the contract is what makes it true. *See also* contract.
- **version range** — the disjoint span of migration numbers a repository owns, so two repositories never write the same version number. An applier can then apply *its* range without guessing whose change it is. *See also* expand/contract.
- **the view** — whatever shows a person what the command surface knows: a terminal, a dashboard, a console. A view renders state; it must not own behaviour, because a view that acts is a second implementation of the operation behind it. *See also* command surface.
- **workflow** — the sequence of steps through which work passes: what happens, in what order, and what decides each step. It is broader than any platform's file — a workflow includes the human gates and the review between pipelines. *See also* pipeline.
- **workflow file** — the file in which a platform defines its automated work: GitHub's workflow, GitLab's pipeline configuration. The artefact, not the sequence of work. *See also* workflow.
- **wrapper** — a verb that stands for a vendor command, kept for policy or for effectiveness. It is not a reimplementation: the vendor's tool still does the work, and the wrapper decides when and how it may be called. *See also* verb.

Chapter 1: The AI-Powered DevOps Engineer

Foundations of Effective AI Usage — and the Road to Software Orchestration

---

1. The Problem This Book Addresses

Ask an AI assistant for infrastructure code and it answers in seconds. The answer looks complete. Then you read it, and spend the rest of the day fixing what it missed.

Nothing in the answer is obviously wrong. It parses. It applies. It looks like the examples already in your repository. It fails in the way that is hardest to catch: it answers the question you typed, not the question you meant. The constraint you thought was obvious is missing. The error case you would have handled is not handled. Both needed a decision from you. The assistant chose, and did not say so.

The AI model is not what decides this. A stronger model writes a better answer to the same question, and the same constraint is still missing. Three things decide whether the answer is usable: what the request required, what the system allows, and what will check the result. The model is not one of the three.

Two engineers, the same AI assistant, the same request. One has something shippable by lunch. The other spends the afternoon repairing what came back. One of them had already written down what the system requires — the boundaries, the conventions, the checks — so the assistant could read it. The other kept it in their head, one conversation at a time.

🎯 What You'll Learn

- Why generated code looks right and is not
- The three structures that decide whether the output is usable — prompt, engineering, workflow
- What each structure controls, and what its absence looks like

- The shift from writing every change to directing the work that produces it

---

The DevOps Engineer's Reality

Scenario	**Your manager requests a VPC with three subnets, NAT gateways, and route tables by noon.**
Unstructured Approach	You prompt: "Write Infrastructure as Code for a VPC."
The Result	You get code in 30 seconds. Thirty minutes later you are fixing what it assumed: a firewall open to the world, a route that only works in one zone, a subnet range hardcoded.
The Cost	Your 10:00 AM task becomes a 4:00 PM incident.

> **Why this happens:** the request described the task and not the constraints. Nothing in it said who may reach what, which failure it had to survive, or what would count as finished — so the answer chose defaults, and did not mention them. Written down once, in the repository, those constraints answer every later request the same way.

The AI-DevOps Reality Gap

This is a pattern, not a one-off error, and the gap between expectation and reality is consistent enough to be predictable. What people expect is work that fits the system — precise, aligned with the architecture, safe by default, maintainable by someone else. What arrives is work that fits the sentence they typed: vague where the request was vague, inconsistent with conventions it was never shown, and carrying the defaults of the example it learned from.

The difference is not the model's ability. It is the absence of the context a colleague would have supplied without being asked.

The Core Question: > *"How do I integrate AI into my workflow to accelerate delivery without sacrificing reliability, security, or maintainability?"*

The Cost of Unstructured AI

Structureless AI use does not merely fail to help. It adds work that is harder to see than the work it removed, because the failure arrives later and somewhere else.

Security gaps appear where the request was silent. An instruction to "add a database" that does not say who may reach it produces a database, a network rule, and a decision nobody made.

Costs arrive after the fact. Token spend is the smaller half. The larger half is the shape an unreviewed answer chose: capacity sized for the example, storage that is never cleaned up, a managed service per component because the sample had one.

Incidents are the third bill. Output deployed without a gate is a change nobody verified. It reaches production at the speed it was generated, which is a new property in delivery: the fastest path is also the least reviewed.

Debt accumulates quietly. Generated code that conflicts with the repository's conventions is debt from the first commit. It is not wrong, so nobody files it; it is different, so every later change pays for it.

None of this is inevitable. Each failure is the absence of a specific structure, and structures are things teams already know how to build.

Three Structures That Control Quality

Every failure above shares a single root cause: **AI was used without structure.** This book is built on the premise that three specific structures determine whether AI accelerates or harms your infrastructure.

Structure	What It Controls	What Happens If Missing
Prompt Structure	The specificity and constraints of the request.	Vague output, endless iteration, missing constraints.
Engineering Structure	How the code fits the existing system architecture.	Architecture drift, inconsistent patterns, technical debt.
Workflow Structure	How the team enforces rules and validates output.	Manual inconsistency, weak auditability, knowledge silos.

Miss one and the other two cannot compensate. A perfect prompt cannot make output fit an architecture nobody wrote down. A perfect architecture cannot verify output no gate inspects.

This book shows you how to build all three, starting with the repository. A structure that lives only in a prompt, or in a tool's memory, vanishes when the session ends. A structure written into a repository is versioned, reviewed and inherited. The repository is the one layer of software development that predates AI and will outlast every AI tool.

The Value Proposition

By applying the methods in this book, you will shift your workflow:

The habit	The replacement
Write a prompt and hope	Write the constraints down once, then prompt against them
Review whatever came back	Review against a contract that already says what “right” means
Fix the output	Fix the structure that produced it, and watch the output change
Wonder whether AI is helping	Measure it: rework, delivery time, incidents

You do not need to be an AI expert. You need to be willing to move the knowledge out of your head and into the repository, where the tool can read it. The three structures do not build themselves. Someone must define them, own them, and direct the AI within them. That person is you. This book calls that way of working **orchestration**: directing the work rather than doing all of it manually.

---

2. Your Role as Orchestrator

The word is chosen for what it excludes. An orchestrator does not play every instrument, and does not hand the score to the musicians and leave. They decide what is played, in what order, and what counts as finished.

The Role Evolution: From Pipeline to Product to Orchestration

The industry is already living this progression. The role this book teaches is the next step in a natural evolution:

Role	Focus	Primary Artefact
DevOps Engineer	The *pipeline* — delivery speed, CI/CD, environments.	A pipeline that ships code reliably.
Platform Engineer	The *product* — internal developer platform, self-service, abstracted complexity.	A platform developers build upon without reinventing the wheel.
Software Orchestrator	The *cognitive architecture* — the prompt, engineering, and workflow structures that allow AI to operate safely.	A decision framework that outlasts any specific tool.

Each step raises the ceiling on what one engineer can direct. The DevOps engineer automated the path to production; the platform engineer productized the infrastructure underneath it; the Software Orchestrator designs the layer above both — the structures that direct AI while it executes the work. Orchestration is a promotion in *what you design*: not the pipeline, not the platform, but the framework that directs them both.

The AI Role Spectrum

You retain ownership of intent, architecture, risk, and outcomes. AI handles drafting, repetition, and pattern application.

```
graph TB
    subgraph You["You own"]
        A[Problem definition]
        B[Architecture and design]
        H[Code review and hardening]
        C[Approval and deployment]
    end

    subgraph AI["AI assists"]
        D[Prompt refinement, gap analysis]
        E[Pattern suggestions, trade-offs]
        F[Draft generation, boilerplate]
        G[Static analysis, test generation]
    end

    A --> D
    B --> E
    H --> F
    C --> G

    style You fill:#e1f5fe
    style AI fill:#f3e5f5
```

> **The principle:** AI augments human expertise; it does not replace critical thinking, risk assessment, or engineering judgment.

What this means in practice:

You own	AI assists
Problem definition & requirements	Prompt refinement, gap analysis
System architecture & design	Pattern suggestions, trade-off analysis
Code review, validation & hardening	Draft generation, boilerplate
Final approval & deployment decisions	Static analysis, test generation

The Boundaries You Hold

Delegation without boundaries is abdication. The moment AI drafts, reviews, and decides for you, the Orchestrator becomes a passive tool operator. These boundaries prevent that drift:

The Boundary	Why It Matters
AI never stages, commits, or pushes	The ship flow is the Orchestrator's. The AI drafts and edits, then reports *changes ready for review* (`git status` / `git diff`) and stops — even when a task says "proceed".
Review before anything ships	Treat AI output like a junior engineer's PR — review for logic, security, and maintainability.
The repository is the source of truth	Standards and conventions live in version-controlled docs. AI reads them; it does not own them.
Validation is automated, not assumed	Quality gates run on every change. AI generates; the repository verifies.
Costs are governed like infrastructure	Meter token usage; route each task to the model that fits the requirement.

The Skill That Outlasts Every Model

As AI handles more mechanical work, your expertise redirects. You shift from writing every resource block to designing the structures the tool works within: architecture boundaries, prompt patterns, and validation gates. That design layer outlives every model and every tool.

Core Principles of the Orchestrator

Everything in this section distills into one identity: **you are the constant; tools are the variables.**

> **Orchestration is not about mastering tools. It is about directing them.**

A "tool operator" learns one tool deeply and is locked into its lifecycle. A **Software Orchestrator** learns a decision framework and applies it across any tool. An orchestrator operates on seven principles:

1. **Tools are transient.** The "best" tool changes constantly. Never bind your identity to one.
2. **Foundations are permanent.** The principles of structure, clarity, and process outlast any specific tool.
3. **Tool selection is based on fitness for purpose.** Not loyalty, not hype, but what fits the task.
4. **The repository is the source of truth.** The tool reads it; the tool does not own it. Your structures persist there.
5. **We build software products, not tool portfolios.** The goal is good software; the tool is just the vehicle.
6. **The role is Orchestrator.** Directing the right tool, at the right time, for the right purpose.
7. **The self-improving repository is the goal.** Structure is persisted, built on and improved — guided by principles rather than reinvented when a tool changes.

Tool operator	Orchestrator
"Which AI tool is best?"	"Which tool fits this task?"
Expert in one tool's keystrokes	Expert in the decision framework
Structure lives inside the tool	Structure lives in the repository
Locked into a tool's lifecycle	Migrates without losing structure
Builds a tool portfolio	Builds software products
Rides each tool's lifecycle	Owns a decision framework; tools are interchangeable
Value is measured in tool mastery	Value is measured in the software products built

Before we design the structures this framework points to, we need to understand the thing we are directing: how does AI actually process information?

---

3. The Mental Model: How AI Thinks

Two questions get mixed up here, and they have different answers. **What kind of system is the assistant?** And **what should a request contain?** The first decides what you can delegate. The second is what you actually type.

Two Kinds of System

A rule engine and a language model are different machines. A rule engine applies logic someone wrote down. A language model predicts what comes next from patterns in its training, which is why it handles requests no one wrote a rule for — and why it can answer a question nobody asked.

	Rule-based (symbolic)	Learned (data-driven)
How it decides	applies rules someone wrote	predicts from patterns in examples
Strength	deterministic, auditable, repeatable	handles what nobody anticipated
Weakness	fails on anything unanticipated	produces confident output with no guarantee
Who supplies the logic	you do, explicitly	the model does, implicitly

The same two columns as a pipeline — rule-based on the left, learned on the right, so the table can be read against it:

```
flowchart LR
    subgraph RB["Rule-based · symbolic"]
        direction TB
        RB1["Rules written by you"] --> RB2["Conclusions"]
        RB2 --> RB3["Deterministic · auditable · repeatable"]
    end
    subgraph LD["Learned · data-driven"]
        direction TB
```

```
    LD1["Patterns in examples"] --> LD2["Predictions"]
    LD2 --> LD3["Flexible · handles the unanticipated<br/>no guarantee"]
end
RB ~~~ LD
```

Neither wins. A production system uses both: the model drafts, and deterministic parts around it decide whether the draft is allowed — a schema it must satisfy, a policy check, the tests.

The rules did not disappear when models arrived. They moved to the edges of the work, where they can be checked: a schema the draft must satisfy, a policy that forbids a shape, a test that has to pass.

Two Kinds of Instruction

Inside one request, the same division appears again, and this is where the confusion usually starts.

- **Constraints you state** behave like rules: *must*, *never*, *only*, *unless*. They are yours, and you can check whether they were followed.
- **Freedom you grant** is where the model decides: the loop, the naming, the shape of the diff, the choice between two equivalent approaches.

The two are not the same thing as the two system types — that is the sentence this section exists to make clear. Writing constraints into a request does not turn a language model into a rule engine. It gives the model rules to obey, and leaves the inference where it belongs.

> **The rule:** state what must be true. Let the model work out how.
>
> The constraints are yours. The implementation is the model's.

A request that leaves everything open	**A request that states the constraints**
"Write a backup script"	"Write a backup script. Check free space first. Verify the archive before deleting a source. Stop and report on any error."
The model invents the safety properties, and you inherit them	The safety properties are yours, and the model writes the code around them

Matching the Request to the Task

The rule is not "be explicit about everything". It is "be explicit about what must not be decided by the model".

The task	**Lean on**	**What you supply**
Boilerplate and scaffolding	the model	examples, house style, surrounding context
Architecture and module design	both	the boundaries, and the patterns already in the repository
Security and compliance	the rules	the rules, written as checks a gate can run
Debugging an incident	both	your analysis, plus hypotheses worth testing

Look back at the two kinds of instruction, because they have a consequence the section has not stated yet. A constraint you state exists only in that request. The freedom you grant is bounded only by what the model happens to know. And the checks that decide whether the draft is acceptable exist only if someone wrote them down.

Three places, then, where a decision can be made or lost: **what the request requires**, **what the code must fit**, and **what verifies the result**. Leave any of the three to the next conversation and it starts from zero — which is why the same request produces a different quality of answer on a different day, from the same model.

Give each of those three places a durable home and the answer stops depending on the day. The remainder of this chapter names the three structures, says what each one controls, and shows where they live so that a change of tool does not take them with it.

---

4. The Three Structures That Control Quality

You now have the mindset (Section 2) and the mental model (Section 3). This section turns that observation into a system you can recognise, name and build.

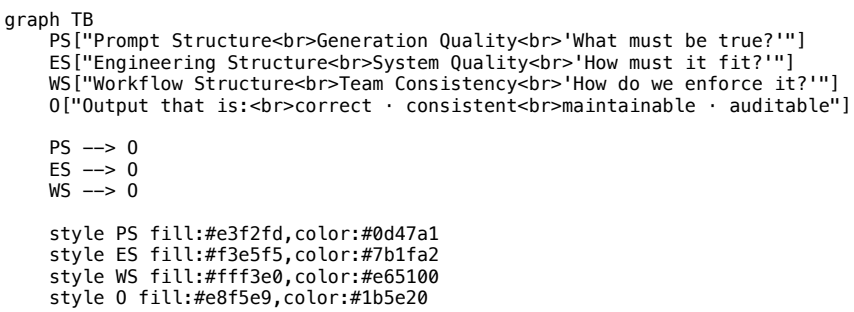
4.1 The Construction Site Analogy

Think of AI as a construction crew.

Structure	**Vague Approach**	**Structured Approach**
Prompt Structure (The Blueprints)	"Build a house"	"A 3-bedroom, 2-bathroom house with a garage and solar panels"
Engineering Structure (The Building Code)	"Build it however"	"Must meet fire codes, earthquake standards, and zoning laws"
Workflow Structure (The Site Manager)	"Trust the crew"	"Every wall is inspected, every permit is checked"

> **The reality:** Most teams have blueprints (prompts). Few have building codes (engineering structure) or site managers (workflow structure).

4.2 The Three Structures Visualized



4.3 Prompt Structure

What it is: How you communicate your requirements to AI — the blueprint.

What it controls: Generation quality, correctness, and completeness.

Key elements: A clear task definition; symbolic constraints (rules that MUST be followed); data-driven guidance (patterns to follow); context (your specific situation); and output criteria. Later chapters combine these elements into reusable templates for each task family.

But a great prompt still cannot force output to fit your architecture. That is the job of the second structure.

4.4 Engineering Structure

What it is: The architectural framework that AI-generated infrastructure code must fit into.

What it controls: System quality, maintainability, and architectural coherence.

The key principle — design before generate:

> **Never ask AI to generate infrastructure before you have defined the structure it must fit.**

The Wrong Way	The Right Way
1. Ask AI to generate	1. **Design** the structure first
2. AI generates whatever it likes	2. **Document** the structure as rules
3. Code works today	3. **Ask AI** to generate code that fits
4. Technical debt accumulates	4. **Validate** it follows the rules

The engineering structure contract. Before generating anything, define the shape your code must take: module and component boundaries, dependency rules, state and configuration handling, naming conventions, validation gates, and error-handling expectations. This contract is the blueprint that tells AI how to generate code that fits your architecture.

> **Key Insight — Platform engineering is the foundation.** Great AI orchestration is built on great platform engineering: the discipline of turning infrastructure complexity into clear boundaries and paved roads. If your infrastructure is a tangle of one-off scripts and half-documented conventions, AI will generate more tangle, faster. But if you have already drawn the boundaries and abstracted the complexity, you have built the perfect engineering structure. AI thrives inside clear, platform-level constraints.

4.5 Workflow Structure

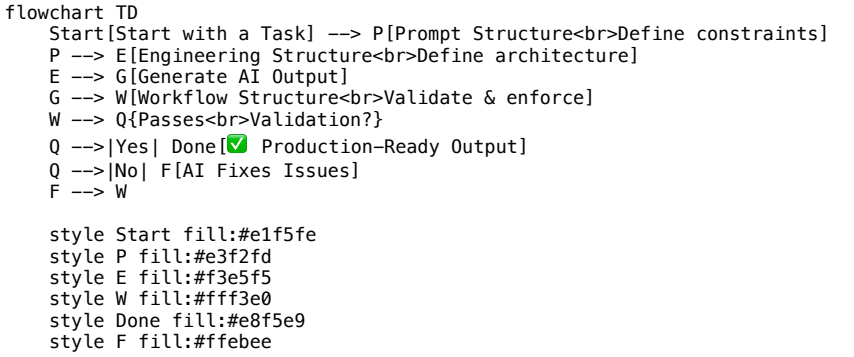
What it is: How your team makes the first two structures repeatable through automation and process.

What it controls: Team consistency, auditability, and automation.

Key elements: Validation gates, task automation, shared context, and review checklists.

> **Why it matters:** Without workflow structure, quality depends on who remembered which rules that day. Workflow structure makes quality automatic and team-wide.

4.6 How the Three Structures Work Together



This loop is the core of the book. Prompt structure defines what must be true, engineering structure defines how it must fit, and workflow structure verifies both, then feeds the results back in.

4.7 Where the Structures Live: The Repository

Where does this loop live? The answer is not a new tool, a new platform, or an AI innovation. It is the oldest, most dependable layer of software development we already have: the repository.

A repository already gives every team four critical properties: 1. **Persistence:** Work survives the machine it was written on. 2. **Versioning:** Every change is tracked, allowing rollback and audit. 3. **Sharing:** The whole team builds on one source of truth. 4. **Accumulation:** Today's work is tomorrow's starting point.

Those four properties are exactly what structured AI usage needs.

The core principle: The three structures do not live in your AI tool; they live in your repository. The tool reads them; the tool does not own them.

Be precise about the terminology. The three structures are **the prompt, the engineering, and the workflow**. The repository is not a fourth structure; it is the **home** of all three.

Governance documentation — the universal pattern. High-performing teams codify how they work in three types of tool-agnostic documents:

Document Type	Purpose
The Law	The non-negotiable rules — what MUST be true, what MUST NOT happen; the symbolic constraints AI is never allowed to break.
The Map	Background and context — the architectural boundaries and business context the AI reasons over.
The Playbook	Executable procedures — validation gates, quality checks, and the repeatable tasks that enforce the first two.

> **These documents are tool-agnostic. Any AI tool can read them. The tool is a reader of structure, not the owner of it.**

Structure	Encoded as	What It Represents
Prompt Structure	The Law	The symbolic rules that constrain every request.
Engineering Structure	The Map	The architectural boundaries and business context the output must fit.
Workflow Structure	The Playbook	The automated enforcement and procedural tasks that make the first two hold.

The permanence principle:

> **Structure lives in the repo, not in the tool.**

When you change AI tools, your prompt templates, engineering decisions, and workflow enforcement all come with you. You only lose the tool-specific shortcuts, never the structure.

4.8 Two Patterns That Make the Repository a Living System

The self-modifying runtime. The repository holds the automation logic that can improve its own contents. AI operates inside this environment, guided by the repository's own rules, and its improvements are validated before they are applied. The heartbeat: **Propose → Validate → Apply → Repeat**.

Tool-Centric Runtime	Repository-Centric Runtime
AI tool remembers conversations	Repository remembers state through version control
Tool's context is temporary	Repository's context is permanent
Switching tools loses context	Repository context survives migrations
Tool defines the workflow	Repository defines the workflow through automation

The command surface. The commands you use to interact with the repository form a stable, predictable interface — one surface a person types into and an agent calls, whose exit codes say success, failure, or the input refused by name. The underlying tools can change; the verbs you use stay the same.

Command Category	Purpose	Stability
Quality commands	Check, validate, lint	Stable interface; implementation may vary
Improvement commands	Generate, enhance, refactor	Stable interface; AI tool may vary
Evolution commands	Validate → Fix → Re-validate	Stable workflow; scripts may vary

The contract is implemented by **two surfaces**: one for the repository in front of you — its build, its checks, its deploy — and one for the work that crosses repositories, such as committing, releasing and syncing. Chapter 2 §9 names both, and says which verb belongs on which.

Stable interface, pluggable implementation. You can switch assistants or task runners without relearning how to work, and documentation stays valid because it refers to the commands, not the tools.

4.9 What "the Outcome" Means: Ten Abilities (NFRs)

"The outcome" is not a feeling, and it is not a benchmark score. It is a set of properties you can **name**, **attribute** to the structure that controls them, and eventually **evidence**. In engineering, we call these Non-Functional Requirements (NFRs), or the "-bilities". Naming them keeps this book honest: a promise

you cannot check is marketing.

Family	Ability	What it means	Controlled by
It is right, and it shows its work	**Correct**	It does what it claims — completely, including edge cases.	Engineering
	Verifiable	Every claim that "it works" resolves to a check you can run, and that can fail.	Workflow
	Maintainable	Someone who did not write it can change it correctly and verify the change.	Engineering
	Auditable	Every change has a provenance and a signer, human or agent.	Workflow
	Secure & private	It resists misuse, and personal data is processed lawfully, minimally, and on purpose.	Engineering + Workflow
It works for the people who use and run it	**Usable**	The surfaces (CLI, API, console, errors, docs) work for their intended users.	Workflow
	Reliable & available	Correct and consistent over time, with a rehearsed rollback path.	Engineering
	Operable & diagnosable	The artefact reveals its own state: you can see what it is doing, understand why, and restore it quickly.	Engineering + Workflow
It grows, and you can leave	**Scalable & affordable**	Capacity and organization grow without a rewrite, at a known cost.	Engineering
	Portable & interoperable	It coexists with other systems and can be moved or replaced without a rewrite.	Engineering + Workflow

Read this table like a contract: - **An ability without evidence is a claim, not a property.** Where this book cannot yet show the evidence, it states so. - **An ability you cannot attribute to a structure is a wish.** If no structure controls it, no amount of prompting will produce it.

These ten abilities are what the rest of the book is for. Part One names them; the practice parts **produce** them on real work and **prove** them.

The capstone: from driver to outcome.

The three structures answer *how* AI work stays correct, but they are not the point of the repository by themselves. The point is what they make possible.

Link in the chain	What it is	Where this book builds it
The driver	the purpose the repository exists to serve — why any of this work matters	named in Chapter 2, revisited as the capstone of every chapter
The delivery plan	the inventory of work that turns the driver into reality, each item with its own lifecycle	Chapter 2 (the feature catalogue)
The three structures	how AI is governed while doing that work — this chapter's subject	Part One (the concept) · Part Two (in depth)
The outcome	whether the driver holds in production — the abilities named in §4.9, each with its evidence	Part One (named) · Part Two and Three (evidenced)

A structure that governs nothing real is a hobby. The repository exists to move the driver to the outcome, and the three structures are how AI is kept correct along the way.

---

5. How This Book Is Organized

This chapter is the conceptual foundation. It deliberately does not teach you how to write prompt templates, run security checks, or pick a model. Those topics belong where they are applied. The book is organized in four parts:

Part	What It Delivers
Part One — Foundations	The argument (this chapter), the repository that holds it (Chapter 2), and the abilities the whole method must deliver (§4.9).
Part Two — The Instrument	What the agent works with, in the order it meets it: the language that names things, the command surface that acts, the context it reads and what that costs, the session it cannot remember, the procedure that sequences a change, the boundary where its authority ends, and the log that proves it all happened
Part Three — The Boundary	What you do not own — governed from inside the repository: the build-or-buy-or-assemble decision and its exit plan; upstream products as dependencies; the provider seam; contracts that keep services independent; sovereignty, portability and the exit drill
Part Four — The Guarantees	What one system must be able to prove before it is asked to scale: show its state (evidence), survive its loss (recoverability), refuse a bad change (the gate)

Part	What It Delivers
Part Five — The Delivery System	The mechanism at portfolio scale: version control as the source of truth, the control plane as a product, versioning and release, deploy and the way back, governance, the agent plane, operations, the fleet, and measurement

How to read this chapter: read it once for the argument. The concepts return throughout the book and are referenced as they are applied.

The promise, stated once. Everything that follows depends on one claim: the three structures — prompt, engineering and workflow — live in the repository, not in the tool. If you accept that, the rest of the book is application. If you do not, no amount of better prompting will help, because the thing that keeps AI work correct has nowhere to live.

The road ahead. The engineer who directs this work is a **Software Orchestrator**, and the advantage comes from three structures — prompt, engineering, and workflow — held in the repository, above any single tool. Chapter 2 makes it real. It shows you how to build the self-improving repository that holds those structures, and everything after it builds on that foundation.

From argument to practice — Chapter 2 builds the foundation.

Part of: The DevOps Engineer's Guide to Effective AI Usage — and Becoming a Software Orchestrator

---

Chapter 2: The Self-Improving Repository

Building the Repository That Holds Your Three Structures

---

1. The Shape of the Repository

In most organisations, a repository is a passive archive. It stores code, waits for a human to decipher it, and relies on tribal knowledge or scattered wiki pages to explain how things work. Rules are fragmented: a lint rule here, a pipeline check there, a convention that lives only in the memory of a senior engineer. The knowledge is real, but it lacks a unified arrangement.

A **self-improving repository** is fundamentally different. It is an active control surface. It does not just store work; it *directs* it.

Chapter 1 established the theory: AI quality is not decided by the model, but by the three structures surrounding it — prompt, engineering and workflow. This chapter provides the mechanism. Those structures cannot live in a person's head or a disconnected document; they must be materialised, version-controlled and enforced within the repository itself.

We organise this material by the three structures for a plain, practical reason: it reduces the cognitive load required to work in the codebase. Reducing complexity is the primary job of software design. Other arrangements are possible, but this one is easy to explain, easy to check, and directly maps to the argument of this book.

This arrangement demands something concrete of every structure:

1. **Constraints must be written down** — the Law and the Map.
2. **The shape of the work must be visible** where a reviewer, human or AI, can point to it.
3. **The result must be verified** by gates that run, not assumed.

> **The day-one acceptance test:** clone the repository, read the front door, run `make help`, and know what is true. If a repository cannot explain itself to a new team member or a new agent without a verbal briefing, it is merely storing work, not directing it.

The rest of this chapter builds that repository, file by file.

🎯 What You'll Learn

- **Structures as an organising principle** — why this book arranges a repository by the three structures, and what that choice asks of each part of it
- **The Documentation Contract** — the set of documents, each with one job, that makes a repository readable by humans and AI alike
- **The Day-One Experience** — why `clone → help → start working` is the acceptance test of a well-built repository
- **Control Plane vs. Application** — the difference between the repository that *governs* many repos and the repos it governs

---

Software Architecture First, AI as the Enhancer

Before the first file, one principle governs this chapter's design: the repository is built with timeless software architecture, platform engineering, and DevOps practice — **not** with AI novelty. AI agents are powerful tools that *leverage* this foundation; they never replace it.

The impedance mismatch. Good software design — modularity, clear boundaries, idempotency, reuse, and strict testability — is precisely what prevents friction with AI. A well-structured repository gives an agent clean boundaries to work inside. A messy one gives the agent nothing but mess to automate: ask an assistant to work inside a tangle of one-off scripts and it will happily generate more tangle, faster. This is Chapter 1's platform-engineering insight applied to the repository itself: build the road well, and the vehicle — human or AI — stays on it.

The Repository That Stores Code vs. The Repository That Directs AI

Most repositories are passive: they store code and wait for someone to read them. A self-improving repository is active — it *directs* how work happens, because it holds the rules, the context, the commands, and the gates that every contributor (human or AI) must follow.

Passive Repository	Self-Improving Repository
Stores whatever was committed	Holds rules that shape what gets committed
Each contributor remembers the conventions	The repository documents and enforces them
Tools own the workflow	Stable commands own the workflow; tools plug in underneath
Quality depends on who reviewed what	Gates run on every change, automatically
Knowledge lives in people's heads	Knowledge lives in version-controlled documents

> **The acceptance test**: clone the repository, run `make help`, and be productive without asking anyone. If the repository cannot explain itself, it is not yet a self-improving repository.

What a Repository Holds

So what does a repository hold? Each of the three structures needs something concrete, and a newcomer or an assistant will ask the same question about each:

What it holds	The question it answers	Examples	The structure it serves
The work itself	what is the actual work?	code, infrastructure definitions, documentation	**engineering** — the shape the work must fit
The checks	how do I know it is correct?	quality gates, security scans, test suites	**workflow** — what decides it is right
The rules and the context	what are the rules and the context?	`AGENTS.md` (the Law), `AI_CONTEXT.md` (the Map)	**prompt** — the constraints a request must satisfy
The way they run	how do I run the workflow?	the command surface: thin targets over scripts	**workflow** — the checks run without anyone remembering

The rules and the context deserve the most attention: they are what AI assistants read first, and what most repositories get wrong. The next section covers it in depth.

Configuration as a Contract, Not a Tweak

A common failure in infrastructure is treating configuration as mutable, environment-specific tweaks. In a self-improving repository, configuration is a strict contract. The fundamental purpose of this contract is to ensure the exact same software logic runs reliably across all environments. We achieve this by strictly separating the logic (immutable, version-controlled automation definitions) from the **environment context**: the variables, secrets, and endpoints an environment supplies. The contract dictates what state must be true. If an environment attempts to apply invalid parameters, or if configuration drift occurs, the deployment fails fast. This guarantees deterministic execution and eliminates the technical debt of ad-hoc environment modifications.

The Documentation Contract

These are the files that fill those places — each one serves a structure, and the documents are what the rest of this chapter builds.

The "four files" framing that most guides teach (AGENTS.md, AI_CONTEXT.md, validate.sh, Makefile) under-counts what a production repository actually needs. In practice, a self-improving repository ships a **documentation contract** — a set of documents, each with one role that every reader (human or AI) can find by name:

Kind	File	Role	The question it answers
Front Door	`README.md`	The entry point — overview, quick-start, and explicit links routing humans *and* AI to the governance files	"Where do I start, and what is this repository for?"
Core Blueprint	`architecture.md`	The source of truth for structural design — components, data flow, constraints	"How is this system designed, and what may not change?"
Governance	`AGENTS.md`	**The Law** — symbolic constraints and non-negotiable rules	"What must I always do, and what must I never do?"
Governance	`AI_CONTEXT.md`	**The Map** — current state, conventions, environmental context	"How is this built today, and where are the conventions?"
Governance	`PLAYBOOK.md`	**The Playbook** — step-by-step procedural tasks, each with a human approval gate	"How exactly do I perform this procedure safely?"
Product & Business	`docs/features.md`	**The Cockpit** — the roadmap: planned and released work, each with a lifecycle	"What are we building, and where does each item stand?"
Product & Business	`docs/capabilities.md`	**The Driver** — the business outcomes and strategic value behind the work	"Why does this repository exist, and what outcome does it serve?"
Operational	`observability.md`	**The Mirror** — human-facing operational signals (compliance, inventory, execution), distinct from any application dashboard	"Is the system healthy right now, by evidence?"

Kind	File	Role	The question it answers
Operational	`operations.md`	**The Runbook** — maintenance, incident response, troubleshooting	"When something breaks, what do I do?"
Records	`CHANGELOG.md`	**The Record** — what changed and why, per release	"What has this repository changed, and why?"
Records	`docs/decisions/<slug>.md`	**The Decision** — the choice, the options rejected, the trigger that would reopen it	"Why is it built this way, and what would make us change it?"

The Law and the Map are the two files that govern AI behaviour directly. The Playbook turns procedures into repeatable, gated steps. The Product & Business documents keep the work honest about *why* it exists and *what* it is delivering. The Operational documents prove the whole thing holds in production. Together they form the repository's permanent memory — the same way a good engineering team keeps its decisions in version control instead of in tribal knowledge.

Beyond the everyday contract lie the append-only records — the change log, the prompt-regression log (Section 11), and the AI execution log (Section 8) — evidence written by the loop and read on demand, not loaded by default.

> **Key Insight — Chapter 1's chain, made concrete.** Every file above carries one link of Chapter 1's chain: `docs/capabilities.md` names the **capabilities**; `docs/features.md` is the **delivery plan**; the governance trio (`AGENTS.md`, `AI_CONTEXT.md`, `PLAYBOOK.md`) holds the **three structures** — the Law for Prompt structure, the Map plus `architecture.md` for Engineering structure, the Playbook plus the gates for Workflow structure; and `observability.md` with `operations.md` hold the **outcome**. Chapter 1 said the repository moves the driver to the outcome — this contract is how.

Control Plane vs. Application Repository

One repository holds the three structures for a single project. A **control plane** holds them for an entire organisation. The distinction matters because the control plane is where the self-improving loop gets its leverage:

```
graph TB
    CP["Control Plane<br>(governs the org)"]
    APP1["App Repo 1"]
    APP2["App Repo 2"]
    APP3["App Repo N"]

    CP -->|"shared foundation<br>(rolled out)"| APP1
    CP -->|"shared foundation"| APP2
    CP -->|"shared foundation"| APP3
    APP1 -->|"evidence + feedback"| CP
    APP2 -->|"evidence + feedback"| CP
    APP3 -->|"evidence + feedback"| CP

    style CP fill:#e3f2fd
    style APP1 fill:#f3e5f5
    style APP2 fill:#f3e5f5
    style APP3 fill:#f3e5f5
```

The control plane holds a single source of truth — a description of every repository, its role, its permissions, and its capabilities — and rolls the shared foundation out to each one, collecting evidence as it goes. This chapter builds the **application repository**: the unit every control plane governs. Section 11 returns to the control-plane idea, and Part Three shows how that machinery is built. This chapter deliberately stays with one repository so the foundation is solid before it is ever multiplied.

The Day-One Experience

The repository's README states its day-one experience in one line: **"Clone this repo → see the commands → start working."** That is not marketing — it is the acceptance test of the whole design.

```
git clone <repository-url>
cd <repository>
make help                # every command, self-documented
make setup               # verify required tooling is present
```

Three properties make this possible, and every self-improving repository should aim for them:

Property	What It Means	How It Is Achieved
Self-documenting	The command surface explains itself	`help` renders every command and its intent
Self-sufficient	Setup is one idempotent command	`setup` verifies tooling in a single step
Consistent	The same verbs work in every repository you touch	Shared command names — same intent, same command

> **Key Insight**: If a new machine, a new human, or a new AI tool can reach "productive" in one command, the repository's structure is doing its job. If it cannot, no amount of AI will fix the gap — the structure is the fix.

---

2. The Intelligence Layer: Law and Map

Two files govern AI behavior, and they serve fundamentally different purposes. Getting this separation right is the single highest-leverage decision in the repository, because it mirrors how AI models process context.

Law vs. Map: When to Use Which

Use `AGENTS.md` (The Law)	Use `AI_CONTEXT.md` (The Map)
"Never commit secrets or API keys"	"Our services are written in Python with typed request models"
"Always run validation before commit"	"Database migrations live under `migrations/`"
"Follow the repository's formatting rules"	"Authentication uses signed tokens with expiring sessions"
"Every automation must be safe to re-run"	"Webhooks are handled in `scripts/handlers/`"

The rule: `AGENTS.md` contains rules that do not change based on what you are working on. `AI_CONTEXT.md` contains facts that vary by project, service, or component.

Why the separation matters. When you ask an AI assistant to "add a new endpoint," it needs two things at once:

1. **Behavioral constraints** (from the Law): validate input, follow conventions, never commit secrets
2. **Architectural context** (from the Map): the service is Python, endpoints live in `routes/`, migrations use Alembic

Without the separation, the AI either **follows rules but lacks context** — writes perfect Python for a Node.js service — or **has context but ignores rules** — knows the structure, but commits secrets. With it, the AI has both guardrails and situational awareness.

Reusable Building Blocks: The Same Law and Map, Many Repositories

The evidence from real multi-repo work adds a second split that most guides miss: **inside each governance file, reusable content and repository-specific content live side by side**. The reusable part — the rules and context that apply to every repository — should stay identical wherever it appears; the specific part belongs to that repository alone.

The classification rule. When a new rule or fact is written, classify it first: does it apply to *every* repository, or only this one? A repository-specific layout rule drifting into the reusable part is the classic failure — it silently makes the shared block non-reusable. The same discipline applies to the Map, to scripts, and to commands: shared building blocks stay generic, and the specific implementation lives where it is used. Reuse without duplication — the oldest rule in software, applied to governance.

> **Key Insight**: A shared block is a *living artefact*, not a memory. It improves in one place and is consumed everywhere — one source of truth, many consumers. How the sharing is mechanised — a template directory, a scaffold command, a roll-out step — is an implementation detail this book deliberately leaves to Part Three. The principle is timeless: keep the reusable block reusable, and let nothing repository-specific drift into it.

The Alignment With How AI Models Process Information

This separation is not an arbitrary convention — it maps to how AI systems actually process input:

AI Layer	Function	Repository Equivalent
System/Instruction Layer	Behavior, constraints, guardrails — loaded first, minimal size	`AGENTS.md` (the Law)
Context/Knowledge Layer	Specific knowledge, patterns, architecture — loaded as needed	`AI_CONTEXT.md` (the Map)
Session/Interaction Layer	Current conversation, temporary state	Chat history, temporary files

The repository structure mirrors the model architecture, which is why the pattern survives model changes: any model, today or tomorrow, still needs behavioral constraints, contextual knowledge, and temporary state. Structure the repository for that permanence and you stop re-architecting every time a new model ships.

The Context Economy: Why the Split Pays

The document set is also economical to *load*, and context is never free. The Law is small and static — cheap to include on every request and friendly to caching. The Map is larger and task-dependent — read only when the task touches that part of the system. Keep the Law small and static, keep the Map organised by area so a reader can pull only the relevant section, and every interaction stays focused.

> **Deferred, not ignored.** The economics of models and routing — what each tier costs, when to escalate, how to meter usage — matter, and Part Two considers them properly. The design principle here stands on its own: load little, load the right thing, and the size of the bill follows the size of the context.

The Law and the Map are the substrate, not the whole system. The six sections that follow are the capabilities a repository builds on top of them: routing the context an agent loads, carrying work across sessions, composing multi-step work safely, bounding when an agent must stop, economizing what enters the prompt, and auditing what the agent actually did. Each is a repository pattern — the same in every repository, regardless of which assistant reads it.

---

3. Context Routing: Progressive Disclosure and Reference Mapping

Section 2 established the split: the Law is small and static, the Map is large and task-dependent. This section turns that split into an instruction. **Progressive Disclosure** means loading the smallest part of the Map that makes the current task correct. **Reference Mapping** is the declared table that says which part is

which: a task type in, a set of Map sections out.

The Routing Rule: Task Type to Map Sections

The mechanism is a table, declared once and read by every agent:

Task type	Load from the Map	Why that slice
Documentation change	Conventions, terminology, chapter format	Prose is governed by voice, not architecture
Service or component change	Components, data flow, test boundaries	A change must respect the blueprint it lands in
Infrastructure change	Environments, deployment, secrets handling	The blast radius is environmental
Pipeline change	Build, release, runners, gates	The failure mode is an unguarded pipeline
Governance change	The Law and the Feature Catalogue	It alters the rules every other reader follows

The table stays small deliberately. A routing table that needs its own lookup has already failed.

The Obligation Lives in the Law; the Mapping Lives in the Map

The **obligation** — classify the task, load only the routed sections, never guess at relevance — is timeless, so it belongs in the Law. The **mapping** is a fact about one repository's Map, so it belongs beside the sections it names. Put the mapping in the Law and the Law turns repository-specific — exactly the drift Section 2 warned about.

What Breaks Without It

Context bloat is the common failure: every task loads every section, because loading is easy and deciding relevance is not. The agent reasons across a Map far larger than its problem, paying in focus and tokens for material that cannot affect the answer.

The opposite failure is subtler — routing that is stale or wrong, so the agent loads a plausible slice that omits the constraint that mattered. Neither failure produces caution; both produce confident guessing.

Implementing It

Declare the mapping as a named section of the Map, and state the obligation once in the Law.

```
## Task Routing
| Task type | Load from this Map |
|:---|:---|
| Documentation change | §Conventions, §Terminology |
| Infrastructure change | §Environments, §Deployment, §Secrets |
| Pipeline change | §Build, §Release, §Gates |
```

The Law's half is one sentence: *classify the task first, load only the routed sections, and state which sections you loaded.*

✅ What to Check

Give an agent a task, then read its Context and Action Summary (Section 8). The sections it read must match the routing table for that task type.

> 🧭 **Key Insight**: A repository that declares its own routing needs no retrieval feature from any tool: the Map is the index, the Law states the obligation, and any reader can fetch precisely.

---

4. Session Continuity: The Gitignored Session Artefact

Every assistant keeps some notion of a session, and every one of those notions is private to that assistant. Switch tools, and the working state — what was read, what changed, what was decided, what comes next — is gone, even though the work is not finished. **Session Continuity** is the repository-level answer: one disposable, human-readable artefact that any reader can pick up.

The artefact is a *baton*, not a record: it lets the next reader — another agent, another model, or the Orchestrator returning after lunch — resume without being re-briefed.

The Handoff Between Tools

```
sequenceDiagram
    participant A as Agent A
    participant S as Session artefact
    participant B as Agent B
    A->>S: write task, decisions, next step
    A-->>A: session ends mid-task
```

```
B->>S: read the artefact
S-->>B: task, files touched, gate status
B->>B: continue from the next step
```

The sequence is unremarkable by design: no tool knows about another; both know about the artefact. Continuity lives in the repository, so it survives every change of reader.

What Breaks Without It

Without an artefact, continuity is replaced by re-briefing: the Orchestrator re-explains the task to the next tool, which re-reads files it has already read and re-derives decisions already made. The sharper failure is silent loss — a session ends mid-task with an edit half-applied and no note of where it stopped.

Implementing It

Keep the schema minimal:

Field	Purpose
`task`	One line describing the work in flight
`status`	`in-progress` / `blocked` / `awaiting-approval` / `complete`
`files-read`	What this session has read — the routing evidence
`files-modified`	What this session has changed — never guess at this
`decisions`	Each key decision and its reason, one line each
`next-step`	The exact next action, written for a different reader
`gate-status`	The last gate run and its observed result

Add the artefact's location to `.gitignore`. It is **derived** state, never a second source of truth: if it disagrees with Git, Git wins.

✅ What to Check

Start a task in one tool, stop deliberately mid-task, then open a second tool and ask it to continue. It should read the artefact, state the next step correctly, and resume without being told what happened. Then confirm `git status` does not mention the artefact.

> 🧭 **Key Insight**: Sessions belong to tools; continuity belongs to the repository. A short artefact any reader can parse buys the freedom to change tools mid-task — the only way "tools are transient" survives a real deadline.

---

5. Workflow Orchestration: Safe Multi-Step Composition

Single edits are easy to review. Sequences are where agents go wrong: five files changed, one validation at the end, and a failure that could have come from any of them. **Safe Multi-Step Composition** is the rule that makes chaining safe — every step ends in a gate, and every gate passes before the next step begins.

The home for these sequences already exists in the documentation contract: the Playbook. A **procedure** is a named, repeatable sequence with declared gates and declared approval points.

The Anatomy of a Procedure

Each step carries four things, and omitting any one of them is how procedures rot:

Element	What it declares	Why it is required
Action	What this step does	A step that cannot be described cannot be reviewed
Gate	The command that proves the step worked	Validation between steps is what stops error compounding
Approval	Human or automatic	The steps needing judgment are named, not assumed
Stop condition	When to abort the sequence	Prevents pushing through a broken premise

A procedure without gates is a to-do list. A procedure without a stop condition is a trap.

What Breaks Without It

The signature failure is **compounded error**: an agent chains edits without validating between them, each built on the previous step's unverified assumption, until the final gate fails and the whole sequence must be unpicked by hand. Gating each step keeps the failure surface at one change — the gate ladder of Section 10 applied across time instead of across a diff.

Implementing It

Every procedure is named, triggered, and gated:

```
## Procedure: <name>
**Trigger:** <when this procedure is the right one>
**Steps:**
1. <action>   → gate: <command>   → approval: <human|automatic>
2. <action>   → gate: <command>   → approval: <human|automatic>
**Stop condition:** <the observation that aborts the sequence>
**Rollback:** <how to return to the last known-good state>
```

Write one procedure before writing ten; the first proves the pattern.

✅ What to Check

Run a procedure end to end and watch the boundaries, not the output. Every gate must execute, and every `approval: human` step must halt and wait. Then break step one on purpose: the procedure must stop rather than improvise around it.

> 🧭 **Key Insight**: Composition is not dangerous because agents are careless; it is dangerous because it hides the moment a wrong assumption entered. A gate between steps keeps that moment visible — exactly when it is still cheap to fix.

---

6. Escalation Policy: Enforcing Human Authority

Section 12 states the authority: AI may read, run, and propose; only the human may write, commit, and release. But a principle that lives only in a paragraph is interpreted differently by every reader. An **escalation policy** is the set of declared conditions under which an agent must stop and report instead of proceeding — and they are the reason a sensible orchestrator can let an agent run at all.

The Conditions, and the Required Response

Condition	The agent must	The agent must not
Irreversible — a delete, a force-push, a tag, a migration with no down path	Stop before it, and report	Act, then explain
Security or production — a credential, a trust edge, a permission widening, work that reaches production	Stop and hand over; name the exact change	Read, copy, or transmit it
Ambiguous authority — contradictory requirements, two sources of truth, an architectural choice with several valid options	Present the options and stop	Pick an interpretation and proceed
A governance change — the change would alter the Law or the Map	Propose the change and stop	Edit governance on its own authority
Out of scope — the work grew past what was approved	Report the scope violation and re-plan	Widen its own scope
A gate that cannot be resolved — the check cannot run here, or failed N attempts to clear	Report the failure and the command	Loosen the gate or call it green

The threshold **N** is a repository decision, and it should be chosen once and stated. An unstated limit is not a limit.

What Breaks Without It

Three failures appear in real work, and all three are the same failure in different clothes — an agent deciding something that was never delegated to it.

The agent that **fixes what it was not asked to fix** sees an unrelated wart, improves it, and hands over a change nobody reviewed. The agent that **quietly edits the Law** resolves a conflict by rewriting the rule, and the guardrail it weakened is now invisible. The agent that **commits because the task said "proceed"** treats an instruction to continue as an instruction to ship.

Implementing It

Declare the conditions in the Law, where they cannot vary by task:

```
## Escalation Conditions
Stop and report to the human when:
- Requirements are ambiguous or contradictory
- Several valid approaches exist and the choice is architectural
- The change would alter the Law or the Map
- The work touches secrets or production
- A gate fails and iterate-to-green cannot resolve it within N attempts
- The task requires a file outside the declared scope
```

Reporting has a shape, which keeps it honest: the condition that fired, the evidence, the options considered, and the decision being requested.

✅ What to Check

Trigger one condition deliberately — hand an agent an ambiguous requirement, or ask it to change a rule. It must stop and report, without making the edit. If it proceeds, the boundary is prose rather than a boundary.

> 🧭 **Key Insight**: Boundaries exist because good intentions are not enforcement. Declaring where your authority begins in advance is what makes granting an agent autonomy safe rather than merely hopeful.

---

7. Context Economy: Context Budgeting by Task Scope

Section 2 explained why the Law/Map split pays. This section makes that economy *governable*: **Context Budgeting** is sizing the context to the task. The practice — two budgets, the always-loaded set, spend in the log — is the economy chapter's subject; here it is the capability. what enters the prompt to the scope of the task, not to the size of the repository. The repository owns one side of the cost equation, and it is the side you can actually change — what enters the prompt is a decision you make today; what the model charges is a market you observe.

Scope Determines the Budget

Task scope	The Law	Map sections	Relative context cost
One-line fix	Full — it is small	The one area the file belongs to	Lowest
Single-component change	Full	That component, its interfaces, its tests	Low
Cross-cutting change	Full	Several areas plus the conventions	Moderate
Architectural change	Full	The blueprint, the conventions, the catalogue	Highest

The Law is loaded on every task, and that is affordable because it is small. The Map is loaded by scope, because it is organised to be read in slices.

What Breaks Without Budgeting

The obvious failure is **loading everything for a small task**: the whole Map enters the prompt to change one line, and the agent pays for context that cannot affect the result. Cost rises while accuracy does not.

> 💰 **Cost Check: Context Budgeting** Loading an entire Map for a small change is the most common avoidable token cost in a governed repository — and it compounds, because every retry re-sends the same oversized context. **The Fix:** route by task scope first (Section 3), then choose the tier. Escalation is for reasoning difficulty, not for context volume.

Implementing It

Budgeting needs no dashboard and no metering — only three rules, all declared in files you already have:

```
## Context Budget
- The Law is always loaded; keep it small enough that this stays cheap.
- The Map is loaded by task scope; never load a section the task cannot touch.
- State which sections were loaded, so the budget is reviewable.
```

Tier pricing, escalation thresholds, and metering are Part Two's material — deferred, not ignored. This chapter owns the input side of the equation: right-sized input, chosen deliberately, and recorded.

✅ What to Check

Compare the Context and Action Summaries (Section 8) of a small task and a large one. The context each loaded should be proportional to its scope. If both loaded the same sections, routing is not being applied — and the cost is paid on every task.

> 🧭 **Key Insight**: Cost discipline is a governance property, not a tool setting. A repository that declares what loads when has made its own spending predictable — with no dashboard, no meter, and no per-tool configuration.

---

8. Execution Transparency: The Context and Action Summary

Governance you cannot observe is governance you cannot trust. Sections 3 through 7 ask an agent to route its context, resume from an artefact, follow procedures, respect boundaries, and budget what it loads — none of it verifiable unless the agent says what it did. The **Context and Action Summary** is that statement: a structured, append-only entry recording the files read, the rules applied, the decisions made, the gates run, and the outcome. It is the agent-side artefact that keeps the loop's structured-feedback chain (Section 11) intact.

What Each Field Proves

Field	What it proves
`timestamp`	When the work happened, comparably recorded
`task`	What the agent understood it was asked to do
`files-read`	That routing (Section 3) was followed, not improvised
`rules-applied`	Which rules of the Law governed the work
`decisions`	Why the change looks as it does
`gates-run`	That validation happened, with results
`outcome`	`green`, `blocked`, or `escalated`
`next-step`	What the next reader should do, or what is requested

A summary whose `files-read` list does not match the routing table is a finding, not a formality.

What Breaks Without It

Without a summary, the first question after any agent task — *why did it change this?* — has one answer: read the whole diff and guess. Worse, governance becomes unverifiable: the Orchestrator who cannot see which sections were read cannot tell whether routing was applied or whether boundaries held.

Implementing It

Keep the log tracked, append-only, and required:

```
## AI Execution Log
| timestamp | task | files-read | rules-applied | decisions | gates-run | outcome | next-step |
|:---|:---|:---|:---|:---|:---|:---|:---|
| 2026-01-14T09:20Z | Fix broken internal link | §Conventions | Truthful docs | Replaced dead link target | make validate ✅ |
        green | none |
```

Whether it is appended directly or summarized from the session artefact is an implementation choice; what matters is that it is durable, structured, and readable without the conversation that produced it.

✅ What to Check

After any agent task, read the entry. Every field must be present, and the `outcome` must match what the gates actually reported — a summary claiming green where the gate failed is worse than no summary, because it looks like evidence.

> 🧭 **Key Insight**: Transparency turns the other capabilities from intentions into observed behaviour. Routing, budgeting, and boundaries are all claims until an agent records the evidence — and a recorded claim can be checked by anyone, at any time.

---

9. The Command Surface Contract

Chapter 1 introduced the command surface contract: *quality commands, improvement commands, evolution commands* — a stable interface with pluggable implementation. This section shows how a real repository implements that contract with time-tested developer tools — a Makefile of thin targets, optionally surfaced through a lightweight CLI — and where the line between "shared" and "repository-owned" sits. Commands are shown here as Make targets; a repository that surfaces the same verbs through its own thin CLI keeps the identical contract — the verb is the contract, not the wrapper.

The Senior Engineer's Insight

> **A stable command surface is the abstraction layer. It shields you from tool churn.**

The pattern separates three things:

What	Purpose	Example
Command	What you type	`make validate`, `make pre-commit`, `make ci`
Interface	The stable contract	A thin Makefile target
Implementation	The pluggable backend	A script that can be rewritten without changing the command

The command stays the same. The implementation underneath changes. When the AI tool changes — today's assistant, tomorrow's harness, next year's something else — the commands you type and the documentation that references them do not.

Thin Targets, Real Logic

The rule that keeps the abstraction honest: **the Makefile stays thin; all logic lives in scripts.** A target is a one-line delegation, never a shell script embedded in the Makefile.

```
## validate — the structural gate: script syntax, help render,

## configuration consistency, tooling smoke tests.
validate:
    @./scripts/validate.sh
```

Why this matters for AI: an AI assistant reading the repository sees a *stable surface* — a small set of named commands with documented intent. The complexity hides behind the interface, where it can be reviewed, tested, and improved without changing the contract.

Two Surfaces: the Repository's and the Operator's

The most common mistake is putting *everything* in one place. A real system needs **two surfaces**, and they differ in **scope**, not in what kind of thing they do — both of them check, and both of them act.

The second one needs a name, and this book calls it **`orch`**: the Orchestrator's console, one copy on the machine, resolved from whichever repository you are standing in. `make` belongs to the repository; `orch` belongs to you.

	`make` — the repository's surface	**`orch` — the operator's surface**
defined in	this repository's `Makefile`	one copy on the machine, resolved from the working directory
knows about	this repository	every repository
verbs	`validate` · `build` · `dev` · `deploy` · `audio-generate`	`commit` · `release` · `sync` · `status` · `diff`
who may run it	anyone, an agent included	the verbs that change history are the **human's**

Two consequences follow from the scope, and they are the reason both surfaces have to exist. **A repository's `Makefile` can only answer questions about itself** — so a verb that must behave the same way in every repository cannot live in any one of them. And **the verbs that change history belong to the operator, not to the repository** — a `Makefile` is a file the agent may edit; the command that ships is not.

> 🧭 **Key Insight — the second surface exists for four reasons, and convenience is not one of them.**
>
> 1. **One interface for everyone.** The human, the agent, the pipeline and the console call the same verbs. One vocabulary, so nothing has to be translated from what a person types into what CI runs.
> 2. **One grammar, few verbs.** `orch <noun> <verb>` is a handful of things to learn; thirty-one `Makefile`s are not. The shared surface is what makes the operator's job *narrower*, not just faster.
> 3. **Uniformity across repositories.** A verb is implemented once, so a fix lands everywhere at once. Measured in one estate: a per-repository `ci` target ran **eight different bodies of work** across twelve repositories, because there was nothing shared to be uniform *with*.
> 4. **The authority boundary.** The verbs that change history sit *outside* the repository the agent edits. The rule *AI never stages, commits or pushes* is enforced by the surface a human uses, not by a target an agent could rewrite.

`orch` is the human's instrument. An agent may read its output; it does not run its mutating verbs. That distinction is what keeps a common interface from becoming a common authority.

> ⚠️ **Pitfall — the duplicated verb**: if a repository redefines something the shared surface already provides (its own commit wrapper, its own release script), the two drift and nobody can say which one is authoritative. A shared verb is used, never copied.

Same Intent, Same Command

The command surface contract only pays off if it is *consistent across repositories*. The rule of thumb: **same intent → same command**. Before adding a new target, check whether another repository already does that intent under a different name — if so, align to the existing name. A repo-specific spelling is only justified when the intent itself is repo-specific.

Intent	**Command**	**Surface**
One-time environment setup	`make setup`	repository
Local preview / dev server	`make dev`	repository
The gates	`make pre-commit` / `make ci` / `make deployment-gate`	repository
Stage, commit, release	`git add` · `orch commit` · `orch release tag=v1.2.3`	operator
See where you are	`orch status` / `orch diff`	operator

The payoff: the Orchestrator never has to remember two commands for the same thing, and an AI assistant can move between repositories without re-learning the surface.

🤝 One Surface for Humans and Agents

The command surface has two audiences, and both are first-class. Humans want readable output; agents need structure they can parse. The resolution is not two surfaces — it is one surface with two output modes.

Audience	Default Experience	What It Needs
Human	Formatted tables, clear logs, colour-coded status	Commands that feel familiar and explain themselves
AI agent	Machine-readable output (`--output json`), predictable exit codes	Idempotent commands, self-documenting help, parseable results

The rule: every command a human can run, an agent can run — and vice versa. There are no "human-only" or "agent-only" verbs. A human developer can run the exact command the agent used and see the same result in the format they prefer.

> 🧭 **Key Insight — dual usability is what makes the agent coachable and the human verifiable.** The agent chains commands, reads structured state, and reports exactly what it did; the Orchestrator re-runs the same commands and sees exactly what the agent saw. When a test command is invoked by an agent, its results default to a machine-parseable form; humans still get their readable summary. That single discipline is what makes the self-improving loop possible — see Section 10.

The Staged VCS Flow

The version-control workflow is where the contract meets the guardrails. It is deliberately **staged and gated**:

```
git add <paths>        # staging is manual, by design
make pre-commit        # the commit gate: validate + staged secrets scan
orch commit              # runs the gate, then commits ONLY staged files
git push
orch release tag=v1.2.3     # gates → clean tree → tag → push the tag
```

Step	What Happens	What It Prevents
Manual `git add`	You choose exactly what ships	Accidental wholesale commits
`orch commit`	The staged gate (`make pre-commit`) runs first	Bad or secret-laden changes entering history
Tag-based `orch release`	Clean-tree check → tag → push tag only	Untested "releases" and half-built tags

> ⚠️ **The boundary you hold (from Chapter 1)**: *AI never stages, commits, or pushes*. The AI drafts and edits files, then reports "changes ready for review" (`git status` / `git diff`) and stops. The ship flow is the Orchestrator's action — even when the task says "proceed." This is the review boundary from Chapter 1, applied to the command surface.

---

10. The Validation Gates

Validation is what turns "AI generated this" into "AI generated this *and it is correct*." But validation has a failure mode of its own: the fake gate.

Gates Real or Absent

The hard-won lesson from real projects: **a gate that always passes is worse than no gate at all.** A stub that prints "skipping" while the pipeline reports "all gates passed" gives false confidence — and both humans and AI treat green as green. The rule:

> **A gate is a promise. A gate that always passes is a lie. Ship real gates, or ship none.**

Fake Gate	Real Gate
Prints "✅ passed" without checking anything	Actually runs the check and fails on violation
Gives false confidence in CI	Fails the pipeline loudly, with a fix path
Hides the problem until production	Surfaces the problem at the cheapest point

The Gate Ladder

Gates are not a flat pile — they are a **ladder** of moments, each with its own budget:

```
flowchart TD
    A["Atomic Moments<br/>one concern each<br/>(fast, any time)"] --> B["Commit Gate<br/>the staged diff<br/>(seconds, locally)"]
    B --> C["Full Gate<br/>everything CI runs<br/>(local == CI)"]
    C --> D["Scheduled<br/>(nightly: exhaustive, slow)"]
    D --> E["Release<br/>(gates → clean tree → tag → push)"]
    style A fill:#e3f2fd
    style B fill:#fff3e0
    style C fill:#f3e5f5
    style D fill:#e8eaf6
    style E fill:#e8f5e9
```

Moment	Command	When You Run It	Scope
Atomic	`make validate`, `make security-scan`	Any time you want one concern checked	Fast, one concern each
Commit gate	`make pre-commit`	Before every commit — what `orch commit` runs	Staged diff only, so it stays fast
Full gate	`make ci`	Before push / release	Everything a push must satisfy — identical to what CI runs

Moment	Command	When You Run It	Scope
Scheduled	the nightly workflow	On a timer	The exhaustive or slow checks no one waits for at a merge
Release gate	`orch release tag=…`	Shipping a version	Merge gate + clean tree + tag

Give the gate an adjective. *Gate* alone names a family, never a specific check — so in prose it carries the word that says what it guards: a **quality gate** guards the code, a **security gate** guards secrets and dependencies, a **release gate** guards what leaves the building, a **commit gate** guards the staged diff, a **console gate** guards a caller's session at an API edge. Bare *gate* is fine only where the adjective is already established by the sentence or the list around it.

And the door is not a gate. The identity entrance is *the door* — a different object with a different job, named differently on purpose, so a reader never has to work out which metaphor is in play. The discipline is the same one the book applies to *the Law* and *the Map*: one word, one thing, and the qualifier says which.

The key design decision is the **full vs. staged split** (the commit gate scans only the staged diff, the full gate scans everything). Without it, the commit path is either slow (full scan on every commit) or dishonest (no scan at all). With it, the commit path is fast *and* honest, and the release path is exhaustive.

🔒 Security as a Gate

Security belongs in the ladder, not as an afterthought. The security scan runs at the atomic moment, in staged form on the commit path, and in full form on the CI path:

Risk	What the Gate Catches
Secret leakage	Hardcoded credentials committed to the repo
Supply chain	Suspicious dependencies, missing lock files
Misconfiguration	Wide-open permissions, unsafe defaults

The same discipline that governs the quality gates applies here: the scan is real, it fails loudly, and its failures carry a fix path. (Security and compliance get their own depth later in the book; this chapter just wires the gate into the ladder.)

The Iterate-to-Green Loop

The final piece of validation is the loop that makes AI self-correcting. It is a non-negotiable working rule for every AI agent:

> **AI work is not done because it was written — it is done when the gate is green.**

```
1. RUN the gate for the change
2. READ the failure — the error output is the spec for the fix
3. FIX the root cause — never mask, skip, or bypass the gate
4. RE-RUN, and repeat until green
5. Only then report success — stating what was run and the result
```

A red gate is a finding to fix, not a reason to claim success. If a gate cannot run **locally**, run it on the **non-production environment** the repository declares — that is where the loop is meant to finish (chapter 19). Report **blocked** only when no such environment exists, and name what is missing.

📡 Testability Is the AI's Feedback Signal

Testability is not a standalone document — it is a contract woven through the repository, and it is the AI's primary feedback mechanism:

Where	The Contract
The Rule (in the Law)	AI-generated code includes appropriate tests
The Boundary (in the blueprint)	Testing boundaries and coverage expectations are defined
The Evidence (in the gates)	Test suites run through the command surface; the gate ladder fails any change that lacks evidence

Why structure matters. Test results are not only evidence for humans — they are the primary structured feedback that drives the self-improving loop. Their output must be machine-parseable — structured results, machine-readable CLI output — so an agent can read *what* failed, *why* it failed, and *where* it failed, then fix and re-run. A gate whose output is prose only a human can interpret is a gate that blinds the agent.

The closed loop:

```
AI generates → the command surface runs the tests → AI parses the structured results
→ AI fixes the failure → AI re-runs to green → AI proposes a governance update → human approves
```

---

11. Documents Are Artefacts Too — the consumer you didn't know you had

That sentence is obvious for code and strange for prose, and the strangeness is why this section exists. Once a markdown file is committed into a governed repository it stops being prose and becomes a **source file**: something reads it, and that reader has a failure mode of its own.

There are exactly three kinds of reader, and each produces a different failure.

Kind	What it does	The failure it causes	Does it tell you?
Projection	transforms the document into artefacts	the artefact disagrees with its source	**yes** — the build fails
Enumeration	holds a whitelist the document must fit	an unexpected file or directory	**yes** — the gate refuses

Kind	What it does	The failure it causes	Does it tell you?
Byte-compare	fingerprints the text against a canonical copy	one repo's copy drifts from the rest	**yes** — the comparison fails

Three failures, all loud. And then there is the fourth reader, the one that matters:

Kind	What it does	The failure it causes	Does it tell you?
Projection without a schema	a view renders the document's rows or keys	a row silently disappears from a screen	**no**

A console that builds a "feature catalogue" from a table, or a "plans" view from front-matter, is a projection with no contract. Rename a column, drop a key, and nothing fails: the view simply shows less. The reader — a person — notices weeks later that something is missing, and there is no record of when it went.

This is the whole practical lesson: declare the consumer, and give a check only to the consumer that fails *quietly*.

```
> **Consumers:** none
> **Consumers:** projection:scripts/project-features.sh
> **Consumers:** console-ledger:config/console-nav.yaml
> **Consumers:** topology:book/docs-topology.json
> **Consumers:** byte-compare:tools/canonical.sh
```

`Consumers: none` is not decoration. It records that the absence was **checked** — the difference between "nothing reads this" and "nobody looked".

Three rules follow, and they are the section in full:

1. **Declare the consumer in the document, not in the tool.** A consumer that exists only in a script's import path is the same defect as a rule that exists only in a runtime's memory: it cannot be reviewed where the document is reviewed.
2. **Check only the silent ones.** A projection that fails the build, a whitelist that refuses the commit and a byte-comparison that names the drift are already gates. Adding machinery to them buys a better error message, not a prevented defect. The view with no schema buys a prevented defect.
3. **The universal check is the cheapest one:** an identifier and a state for every row; the keys the view reads, present in the front-matter. Two assertions, no schema engine, no dependency graph.

What this is not

It is not a documentation system, and it is not a dependency graph. The moment a "document build orchestrator" appears, it becomes the next artefact that drifts — and it will be invisible, because nobody thinks of the orchestrator as documentation.

It is also not a reason to write more documents. Every consumer is a contract someone must keep; the same reasoning that says *declare the consumer* says *delete the document nothing reads*.

> **The connection to Chapter 14.** The integration charter says: decide the boundary once, then let something check it. A document's consumers are its boundary. Same shape, smaller scale — and the same rule: the boundary exists whether or not you wrote it down; the only question is whether you find out at build time or at reading time.

Evidence: one gate that fails when a catalogue loses its state column or a plan loses its front-matter keys; and one document whose `Consumers:` line is `none`, checked rather than assumed.

Template: the `Consumers:` header line, and the two-assertion console check.

---

12. The Self-Improving Loop at Org Scale

A self-improving repository improves itself. A **control plane** makes that improvement compound across an entire organisation: improve the foundation once, sync it everywhere, collect evidence, repeat.

The Manifest: One Source of Truth

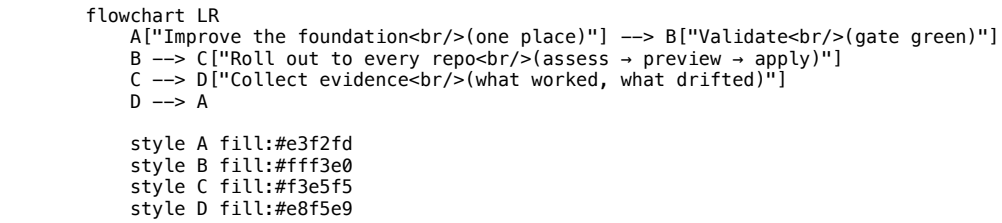
The control plane's core artefact is a **manifest** — a declarative file that describes every repository, its tier, its class, its owner, its lifecycle, its policy, and the teams and permissions that apply. It is the single source of truth for the organisation:

The Manifest Declares	What It Drives
Repos, tiers, classes, lifecycle	Cloning, workspace file, foundation-sync eligibility
Teams and permissions	Access and onboarding (planned sync)
Runner groups and profiles	Execution capacity and trust boundaries

Because the manifest is declarative and validated, the answer to "what is the state of the org?" is always *read the file, run the gate* — never "ask the person who knows." AI assistants read the manifest before touching org tooling; the Orchestrator changes the file and validates.

Reusable Building Blocks: The Roll-Out Discipline

The foundation standard — the documentation contract, the governance templates, the command-surface verbs, the reusable scripts — lives in one place and reaches every repository through a disciplined roll-out: **assess** the current state, **preview** what would change, **apply** it idempotently. The roll-out never edits a repository's own surface — the repository's gates and its specifics stay its own. How this machinery is built is Part Three, which assumes the vocabulary introduced here and builds on it rather than re-teaching it; this is the self-improving loop from Chapter 1 made concrete:



The multiplier made real. One humthe Orchestrator, one control plane, and the same foundation rolls out across every repository. The book's claim in Chapter 1 — that the Orchestrator's leverage comes from systems, not effort — is exactly this loop. (The quantified multiplier — "N× faster" — is a claim this book only makes once measured across a release milestone; treat marketing numbers elsewhere with suspicion.)

✨ Self-Documenting Automation: The Timeless Intent Pattern

For the loop to work, every artefact must be reviewable by human or AI at any time, on the surface where it is actually reviewed. This is the **Timeless Intent pattern** — the invariant each artefact maintains, stated where it is read:

Surface	The Contract
Automation run output	Every step: a timeless ▸ intent (the invariant it maintains), short action lines, a ✅/❌ outcome, and on failure a → `FIX:` next action with a doc pointer
Code definition site	The file/function header states the invariant the code maintains; error messages carry the actionable → `FIX:` shape
Status dashboards	Every critical invariant is observable as a live signal, consuming the same check logic as the gates — never a second source of truth

Timeless over incident. The intent is the enduring contract ("this hostname must always answer unauthenticated"), never the bug that prompted the fix ("your zone had sign-on enabled"). Incident detail — error codes, symptoms — belongs in the FIX line or the runbook, where it is timeless troubleshooting.

> 💡 **Why this matters for AI**: an artefact that only makes sense in light of the last incident is not maintainable — by a human *or* an AI. The Timeless Intent pattern is what lets an AI agent read any workflow's output, spot a deviation, and propose the fix without a conversation history.

The Feature Catalogue: Making Status Observable

Every repository maintains a **Feature Catalogue** (`features.md`): each feature has a lifecycle state, a validation, and a dashboard signal. The states form a standard ladder:

`planned → in-progress → implemented → configured → healthy → degraded → deprecated → removed`

The catalogue is the organisation's honesty surface. A capability is "healthy" only when its validation passes and its dashboard signal is live — status is observed, not asserted. This is the workflow-structure counterpart to the validation gates: enforcement on one side, visibility on the other.

The Instrument That Reads the Manuscript

A repository that validates its build and not its book has instrumented the wrong artefact. Gates prove the site is consistent; they cannot tell you the argument is. So the self-improving loop gets one more reader, pointed at the manuscript itself.

It asks five questions, each answered by a check that can fail, and whose failure names the sentence to change. Four run today; the fifth is on probation — it earns its place or it is cut:

Check	The question	What a failure means
Term integrity	Does the prose name a thing the way the repository names it?	Two names for one thing, or one name for two
Pointer integrity	Does a sentence that points at a file still describe what is in it?	The abstraction goes concrete and lands on the wrong file
Promise → payoff	What does the text promise that the manuscript never pays?	A reader waits for a chapter that is not coming
Source coverage	Which claims assert evidence and cite nothing?	A sentence that sounds proven and is not
Section role *(probationary)*	Does this section advance the argument, or decorate it?	Prose that could be cut without loss

It refuses more than it measures, and every refusal is a measurement rather than a taste:

Refused	Why the measurement refuses it
Redundancy scan	Zero repeated body sentences of twelve words or more across the published edition; eight-gram overlap between page pairs is 0.8 %, 0.3 % and 0.2 %. It would report zero and consume attention.
Thesis-word density	It ranked a legitimately mechanical section lowest — and the section only looked thin because a heading inside a code fence had split it in two. A proxy that mislabels good prose teaches the editor to skip the report.
Reader telemetry	It needs per-reader tracking, which this book's privacy boundary forbids; at current readership an aggregate would be too noisy to change a decision.

Refused	Why the measurement refuses it
Search-query instrumentation	Search is client-side — queries never reach a server, so there is nothing to record without adding a third-party service.
External link rot	The published edition has zero external links. A check over zero links can only pass — and *a gate that always passes is worse than no gate at all*. It becomes real when source coverage adds the first citation.
Build-size budget	The deploy already enforces the file limit through the audio check. A second check would be a second source of truth.
Authorship metrics	Forbidden by this book's own rule — never measure output volume, lines or commit counts — and two chapters have no distribution to measure.

The instrument earns its place by finding things, not by reporting green. Three findings from the published chapters, each one a sentence an editor could act on — and did:

Finding	What the check compared	What it changed
One catalogue, one name	the capability catalogue and the feature catalogue are separate files with separate jobs; a third name for either is a second source of truth	Pick one term per document, and use it everywhere
A pointer that describes the wrong file	The sentence "`docs/capabilities.md` names the driver" — the file names no driver	Fix the sentence, or make the file do what the sentence says
A promise with no payoff	"the book's closing chapter walks this list" — there is no closing chapter, and the outline records it blocked	Hedge the sentence, or mark the promise open until the chapter exists

A fourth finding taught the design its most important lesson. A naive scan reported eight sections in this chapter missing their key-focus line; seven of those "sections" were headings inside fenced code blocks — samples of governance files, not prose. The scan was wrong, not the chapter. **A check that cries wolf gets ignored**, which is why the instrument reads fences before it reads sections.

> 🧭 **Key Insight**: The manuscript already holds its readers to *"an ability without evidence is a claim, not a property."* The instrument turns that sentence on the book itself: status is observed, not asserted — including the status of the argument.

That is the whole claim: not analytics, and not a dashboard. A list of the sentences that decide whether the prose is good, so the rewrite, cut, merge and add-source decisions are made from evidence instead of memory.

🔁 The Feedback Architecture

The self-improving loop runs on structured outputs. Each link in the chain — test results, validation-gate output, command responses — must be machine-readable, so the agent can locate a failure and act on it without a human interpreting the output first. If a single link outputs unstructured prose that only a human can interpret, the agent's ability to self-improve is severed at that point. Build the chain once — generate → run → parse → fix → propose — and every improvement after it is automatic.

🩺 The Post-Mortem Auto-Update Rule

When an AI-generated change fails a validation gate, fixing the code is only half the work. The rule: the agent also proposes a specific update to a governance file — the Playbook, the Map, the runbook, or the change log — so the same failure cannot recur. The trigger is the structured failure output: the agent reads it, diagnoses the root cause, applies the fix, and proposes the governance update that encodes the lesson. A failure that fixes the code but forgets the lesson is a failure that will repeat.

🧪 Prompt Regression Testing

How do you know the Law is actually obeyed? By testing it. Prompt regression testing deliberately prompts the AI to violate a rule in a sandbox, verifies the constraint holds, and logs the result. Each test is logged to a dedicated regression log (`docs/prompt-regression-log.md`) — date, rule tested, the violation attempt, and the outcome — so the proof that the Law holds is reviewable by human and agent alike and stays out of the everyday context. If the Law can be talked out of enforcing its own rules, the problem is in the Law, not in the prompt: strengthen the constraint and re-run the regression. Adversarial testing in depth belongs to Part Two — the principle belongs here: *a rule you have never tried to break is a rule you do not yet trust*.

---

13. AI Agents in the Governed System

Everything so far — the Law, the Map, the command surface, the gates — exists so that AI can work inside a governed system. This section shows how the agents plug in, how to route their work across model tiers, and where the lines are.

The Agent as Reader, Not Owner

The AI assistant — a CLI tool, an IDE agent, or a harness — is a **reader of structure**. It does not remember the rules; it reads them. It does not own the workflow; it invokes the commands. That is what makes assistants interchangeable — the same tool-centric-versus-repository-centric contrast Chapter 1 drew, now built rather than described.

> 🧭 **Key Insight**: The book does not claim a specific assistant is more "reproducible" than another — that is a tool property to verify on your own bench, not a foundation claim. The foundation claim is stronger: *any* reader of the repository inherits its structure, so the workflow survives the tool.

The 3-Tier Strategy: Routing Work to the Right Model

Model selection is fitness for purpose, not loyalty to a brand. The 3-Tier strategy is the routing rule used throughout this book's own production workflow:

Tier	Role	Use It For
Tier 1 — Architect	Complex reasoning, system design	Architectural planning, deep structural reviews
Tier 2 — Editor	High-volume, repetitive execution	**Default (~80% of tasks):** editing, boilerplate, validation fixes
Tier 3 — Sovereign	Zero data egress, absolute privacy	Sensitive code, secrets, offline environments

The default is Tier 2 — most work is routine. Escalate to Tier 1 only for complex reasoning. **Mandate Tier 3 for anything sensitive**: for sovereign work, models run locally, so nothing leaves the machine. The tier is defined by *where the model runs*, not by any one model or registry — pull a verified open-weight model and serve it on your own hardware. What each tier costs, when to escalate, and how to meter usage are economics this book considers properly in Part Two. The routing rule itself is durable: right-size the model to the task.

The Agent Harness Pattern

When work scales past a single interaction, the most advanced systems use a **harness** — a framework that orchestrates multiple agents toward a goal. The repository itself can act as one:

Layer	Component	Role in the Loop
1. Router	Makefile targets (`make validate`, `make improve`)	Directs tasks to the right command and workflow
2. Context Manager	`AGENTS.md` + `AI_CONTEXT.md`	Shared state and rules; every agent reads from here
3. Agent Pool	CLI assistants, IDE agents, validation scripts	Execute tasks; learn from feedback; improve over time
4. Feedback Loop	Git history + validation results	Records outcomes; informs the next iteration

The loop is the self-improving pattern from Chapter 1 in action: validation failures make the Law clearer, successful patterns enrich the Map, repeated tasks become new commands, and Git history shows what is working.

The AI Workflow Loop

Inside the governed system, every AI task follows the same sequence — the loop this book has been building toward:

1. **Read the Law** (`AGENTS.md`) — the non-negotiables
2. **Read the Map** (`AI_CONTEXT.md`) — the architecture and conventions
3. **Read the manifest** when the task spans repositories — the org's source of truth
4. **Make the smallest coherent change**
5. **Run the gate and iterate to green** — never declare success on unverified work
6. **Stop and report** — the boundary from Section 9: never stage, commit, or push

Boundary (from Chapter 1)	Applied Here
Review before anything ships	AI output is treated like a junior engineer's PR
The repository is the source of truth	Agents read the Law and the Map; they don't own them
Validation is automated, not assumed	Gates run on every change; iterate-to-green is mandatory
Model work is right-sized	The 3-Tier strategy routes the task to the model that fits
Human authority is final	AI suggests; the human approves every write, commit, and release

The Human Orchestrator Authority

The self-improving loop has a last gate, and it is not automated. **Human Orchestrator Authority**: an AI agent may read, parse, suggest, execute commands, and propose fixes — but every change to the repository ultimately rests with the humthe Orchestrator. The agent operates inside a sandbox of *read* / *execute* / *suggest*; the *write* / *commit* / *merge* authority is reserved for the human.

The AI May	Only the Human May
Read the Law, the Map, and the state	Change the Law, the Map, and the contracts
Run commands and parse results	Commit, merge, and release
Propose fixes and governance updates	Approve or reject every proposal
Suggest a release	Ship it

The workflow: AI generates → AI runs the tests → AI reads the results → AI proposes fixes → AI proposes governance updates → **the human reviews and approves** → changes are committed. Every step is automated except the final authority, which remains human.

> **The safety net**: this is what keeps the "self-improving" loop from ever becoming a "self-destructing" loop. The human can reject, modify, or override any proposal. The AI is a powerful assistant, not an autonomous decision-maker — and every procedure in the Playbook carries an explicit human approval gate before anything is committed. The AI does ninety-five percent of the work; the human signs the final five percent.

---

14. Synthesis: The Repository as the Ultimate Force Multiplier

The six capabilities are not six features. They are one operating loop, and each step is what keeps the next honest:

```
flowchart LR
    R["Read<br/>(Law + Map)"] --> RO["Route<br/>(load the slice)"]
    RO --> A["Act<br/>(follow the procedure)"]
    A --> V["Validate<br/>(gate each step)"]
    V --> L["Log<br/>(context & action summary)"]
    L --> E{"Escalate<br/>or complete?"}
    E -->|"condition fired"| H["Human decides"]
    E -->|"green"| D["Done"]
    H --> RO

    style R fill:#e3f2fd
    style RO fill:#e3f2fd
    style A fill:#fff3e0
    style V fill:#fff3e0
    style L fill:#f3e5f5
    style E fill:#e8f5e9
```

Routing and budgeting (Sections 3 and 7) decide what enters the prompt; act and validate (Sections 5 and 10) decide how the work is done. Log (Section 8) records it, and escalate (Section 6) hands the loop back to the human. The heartbeat from Chapter 1 — propose, validate, apply, repeat — is this loop seen from the repository's side.

The Order Is a Dependency, Not a Preference

Routing precedes economy: you cannot budget a context you do not control. Continuity precedes transparency: there is nothing to summarize if state does not survive. Boundaries precede orchestration: chaining steps is safe only once stop conditions exist. Transparency sits over all of it.

Implement in this order: routing, continuity, orchestration, boundaries, economy, transparency.

The Maturity Model: Minimum Viable, Then Full

No repository adopts all six at once. Start with the minimum; take the full pattern when the work justifies it.

Capability	Minimum viable	Full pattern
Context Routing	One routing table	Per-task-type routing, verified
Session Continuity	Artefact with task and next step	Full schema with decisions and gate status
Workflow Orchestration	One named, gated procedure	A library with stop conditions and rollback
Escalation Policy	Six declared stop conditions	Conditions plus tests that trigger them
Context Economy	The Law/Map split alone	Task-scope budgeting, reviewed per task
Execution Transparency	An append-only log with the core fields	The full summary, checked against routing

The minimum column is a real implementation, not a placeholder: each row closes its failure mode.

Why This Elevates Any Tool

A repository built this way does not depend on a capable assistant; it supplies what the assistant lacks — session memory, context selection, workflow support, guardrails, cost control, and audit history. A modest tool plugged into it therefore behaves like a better one, because the structure it lacks was never the tool's job. This is the chapter's opening claim — the repository is the hero — demonstrated, not asserted.

> **Key Insight**: You do not wait for a tool to gain these capabilities. You build them once, in the repository, and every tool you plug in inherits them — the practical meaning of "tools are transient, the repository is permanent."

---

15. Putting It Into Practice

Build or Clone: The Bootstrap

You have two valid starting points:

Path	When to Choose It	First Steps
Clone a control plane	A foundation already exists (team, org, or your own past work)	`git clone` → `make help` → `make setup` → start
Build your own	Starting fresh, or the existing repos are unstructured	Follow the 30-minute bootstrap below

The 30-minute bootstrap, compressed to what actually matters:

```
# 1. Initialize (5 min)
git init && touch README.md .gitignore

# 2. The Law, the Map, and the Playbook (10 min)
#    AGENTS.md — rules that never change; AI_CONTEXT.md — architecture and facts;
```

```
#    PLAYBOOK.md — the procedures that must be repeatable and gated.
#    If you manage more than one repo, keep the reusable parts generic — how
#    they are shared is a later-part concern, not part of this bootstrap.

# 3. The command surface (5 min)
#    Makefile with thin targets: help, setup, validate, pre-commit, ci,
#    plus the VCS surface (status, sync, log, diff, commit, release).

# 4. The gates (5 min)
#    scripts/validate.sh — a REAL check. A stub that prints success is a lie.

# 5. The first loop (5 min)
make validate && git add <paths> && orch commit
```

✅ Foundation Checklist

```
□ README.md — the front door: what this repo is, how to start, links to governance
□ architecture.md — the core blueprint: design, components, constraints
□ AGENTS.md — the Law: rules that never change
□ AI_CONTEXT.md — the Map: current state, conventions, key file locations
□ PLAYBOOK.md — the Playbook: procedures with explicit human approval gates
□ docs/features.md — the delivery-plan catalogue: planned and released work
□ docs/capabilities.md — the drivers: the business outcomes behind the work
□ observability.md — the operational mirror (when the repo is operated)
□ operations.md — the runbook (when the repo is operated)
□ CHANGELOG.md — a record of what changed and why
□ The command surface — thin targets / CLI verbs that delegate to scripts/
□ scripts/ — the real logic, including a REAL validate gate
□ Structured test results — machine-parseable output the agent can read
□ docs/prompt-regression-log.md — the rule-violation regression log (adopt with prompt regression testing)
□ The staged flow — manual git add, gated commit, tag-based release
□ An AI assistant configured to read the Law and the Map
□ A human approval gate on every write, commit, and release
□ Context routing rules — task type → Map sections
□ `.session/` in `.gitignore` — the session artefact
□ One named, gated Playbook procedure
□ Escalation conditions in the Law — the stop-and-report rules
□ Context-loading rules — what loads when
□ `docs/ai-execution-log.md` — the action summary log
□ A verification step for each capability — one check you can run
```

Quick Reference

Commands are illustrated as Make targets; the same verbs are surfaced through your CLI where one exists — the verb is the contract.

Action	Command
See every command	`make help`
Fast commit gate	`make pre-commit`
Full gate (local == CI)	`make ci`
Commit staged changes	`orch commit` (custom message: `m="..." orch commit`)
Tag-based release	`orch release tag=v1.2.3`
Review the staged diff	`make diff-staged`
Sync with upstream	`make sync`

Common Issues & Fixes

Issue	Likely Cause	Fix
Gate passes but nothing was checked	Stub gate	Replace with a real check — a fake gate is a lie
`orch commit` refuses to run	Staged gate failed	Read the failure, fix the root cause, re-run (iterate to green)
AI "completed" without running the gate	No enforcement	Require iterate-to-green; treat unverified work as blocked
Different commands in different repos	Surface drift	Align to the standard names; same intent → same command
High token spend	Context bloat / wrong tier	Use the Law/Map split; route with the 3-Tier strategy
Entire Map loaded for a one-line change	No context routing	Add task-type routing
A new tool cannot continue previous work	No session artefact	Create `.session/current.md`; gitignore it
Edits chained without validating between steps	No Playbook procedure	Write one named, gated procedure
Agent "fixed" something outside its scope	No escalation policy	Declare stop-and-report rules
Small tasks cost as much as large ones	No task-scope budgeting	Add context-loading rules per task scope
Cannot verify the agent followed the routing	No execution log	Add the log; require the summary

---

16. 🔑 Chapter Summary

What You Learned

This chapter built the repository that holds Chapter 1's three structures:

1. **The repository is the hero** — the work, the checks, the rules and the context, and the way they run, plus the documents that make make it readable by humans and AI alike — and it carries every link of Chapter 1's chain, from driver to outcome
2. **The Law, the Map, and the Playbook** — `AGENTS.md` holds the rules, `AI_CONTEXT.md` holds the context, `PLAYBOOK.md` holds the gated procedures — and reusable building blocks keep all three identical across many repositories
3. **The command surface is a contract** — one dual-usable surface for humans and agents over pluggable scripts; shared blocks stay identical, repository-owned gates stay local
4. **Gates are promises** — real or absent, arranged as a ladder (atomic → commit → full), with the Iterate-to-Green loop and machine-readable results making AI self-correcting
5. **The loop compounds at scale** — the control-plane concept (a single source of truth, reusable building blocks, a disciplined roll-out) turns one improvement into an organisation-wide improvement; structured feedback, the Post-Mortem Auto-Update Rule, and prompt regression testing keep the loop honest
6. **Agents are readers, not owners** — they inherit the structure, are routed by the 3-Tier strategy, and work inside the boundaries you hold — with Human Orchestrator Authority as the final gate
7. **The repository operates the agent** — six capabilities (routing, session continuity, orchestration, escalation policy, context economy, execution transparency) turn a governed repository into an operating environment that any tool inherits

The Universal Pattern

> **Any repository + AI + Git + Validation = a self-improving system**

Where: - **Every change is auditable** (Git commits) - **Every change is validated** (real gates) - **Every change improves the system** (the loop feeds back) - **The workflow itself evolves** (templates + sync)

Your Next Steps

```
1. Build or clone your self-improving repository (30 minutes)
   - Law, Map, Playbook, command surface, real gates

2. Run your first improvement cycle (15 minutes)
   - Improve → validate → iterate to green → human approves → commit

3. Expand to your actual work (ongoing)
   - Any repository: the pattern is portable

4. Scale to a control plane (when you manage more than one repo)
   - One source of truth → shared blocks → roll-out → evidence
```

---

17. Where This Leads: From One Repository to Many

What Comes Next

You now hold the governed repository: the unit of orchestration. Chapter 4 gives it a command surface — the thin layer over the tools you already drive, so the rules this chapter wrote down can run. The book continues in three practice parts, and the patterns you built here are the ones every later chapter reuses:

- **Part Two — The Instrument** is what the agent works with, in the order it meets it: the language that names things, the command surface that acts, the context it reads and what that costs, the session it cannot remember, the procedure that sequences a change, the boundary where its authority ends, and the log that proves it all happened.
- **Part Three — The Boundary** takes the workbench outward: what you do not own — an upstream product, a provider's commodity, the integration charter, contracts, sovereignty, and the exit drill that proves the boundary holds.
- **Part Four — The Guarantees** is what one system must be able to prove before it is asked to scale: show its state, survive its loss, refuse a bad change.
- **Part Five — The Delivery System** makes it organisational: version control as the portfolio's source of truth, the control plane as a product, versioning and release management, governance the machines enforce, the agent plane at scale, portfolio operations, the fleet, and measurement.

Chapter 3: The Language of Control

Vocabulary, Structure, and AI-Readable Architecture

---

1. The Constraint You Cannot Name, You Cannot Request

Compare two requests made to an AI assistant.

> *"Make the billing service match the rest of the site and don't break anything."*

> *"Update the billing service to use the `payment.gateway.v2` interface and the existing `Invoice` DTO; do not introduce new database calls or alter the existing error taxonomy."*

The second request is longer, and none of the extra length is padding: each added term removes an option. It is more precise because it uses **vocabulary** that names the exact constraint. Vocabulary is the structure you impose on a task *before* an assistant ever starts reasoning about it.

A team that shares precise terms for its architecture can direct AI-accelerated work in a handful of words. A team without that vocabulary has to hope the AI model infers the constraints the vague prompt failed to state. When a request is under-specified, the AI fills the gap by predicting the most probable interpretation from its training distribution — and that distribution is the internet's conventions, not yours.

That gap is paid at review. A vague request produces output that looks plausible and must be checked line by line, because nothing in the request told you what to check it against. A precise request produces output that can be checked against a stated constraint.

Of the three structures Chapter 1 named, the prompt structure is the one that can only state what the team has words for. If a team lacks the vocabulary for a constraint, it cannot ask for it, and it will not recognise the wrong answer when it arrives.

But vocabulary cannot live only in a prompt, nor can it rely solely on an enforcement tool. Before an agent ever executes a command, it must first read the codebase. Therefore, this shared language must be embedded in the very fabric of the repository. This chapter is about that fabric: the names, the folders, and the types that tell an agent what is true before it writes a single line.

> **The rule this chapter defends:** Vocabulary turns review from *auditing a stranger's taste* into *checking a stated constraint*. A term earns its place only when it makes a constraint requestable, checkable, or enforceable.

🎯 What You'll Learn

- **The power of shared vocabulary** — why naming constraints precisely is the first layer of AI governance
- **The working vocabulary** — the architectural domains an Orchestrator must define to direct work
- **The shape of the code** — how folders, names and types act as the strongest context you can supply
- **Enforcement as grammar** — why a convention in prose is a suggestion, and a convention in a check is a rule the gate enforces

✅ What to Check

- Take the last three AI-assisted changes. Could the request have been more precise using vocabulary the team already has?
- Name one architectural convention that lives only in someone's memory. That is the first thing to move into a shared artefact.

---

2. The Working Vocabulary

Jargon for its own sake is a liability. A shared, precise vocabulary across four domains covers most of what an Orchestrator needs to name and control.

The test for any term is simple: does it turn a vague wish into a precise, verifiable constraint?

Domain	The Vocabulary	The Constraint It Makes Requestable
Backend & Infra	`ephemeral sandboxing`, `graceful degradation`, `CQRS`, `event sourcing`	*"Execute this in an **ephemeral sandbox** with no persistent state,"* or *"Implement **graceful degradation** by returning cached data on timeout, not a 500 error."*
Workflows	`policy-as-code`, `shift-left`, `GitOps`, `observability`	*"Enforce this via **policy-as-code** in the CI gate,"* or *"Apply **shift-left** validation by adding a pre-commit hook for this schema."*
Software Design	`bounded context`, `contract testing`, `cyclomatic complexity`	*"Respect the **bounded context** of this module; do not import from outside it,"* or *"Ensure this passes **contract testing** without modifying internal implementation."*
Governance	`the Playbook`, `prompt-regression log`, `the Law`	*"Follow the **Playbook** step-by-step for this database migration,"* or *"Add this failure pattern to the **prompt-regression log** so the agent cannot repeat it."*

Three of those concepts deserve to be artefacts in their own right, because they turn vocabulary into repository reality:

1. **The auto-feedback loop.** A check that runs on every change and reports pass or fail immediately. This is *fail fast* and *idempotency*, enforced by a machine.
2. **The platform playbook.** A documented, scoped procedure for a recurring task. This is *bounded context* and *least astonishment*, written down instead of assumed.
3. **The prompt-regression log.** The record of AI-introduced failures that became permanent tests — Chapter 2's append-only record. This is the self-improving repository in action: the same mistake becomes structurally harder to repeat.

Where those terms live, who approves one, and what makes one worth keeping are the same three questions every repository answers badly by default.

Where the terms live. One file, linked from the Map — the conventions file from §5 — so a request can cite a term and a reader can look it up. A term that lives only in conversation is a term the assistant cannot read.

Who approves one. The Orchestrator. A term is a constraint the whole estate is held to, so it is a decision rather than a preference: the same authority that approves a gate approves the vocabulary that gate enforces.

What makes a term earn its place. It must change an outcome — a request can be made more precise with it, a check can fail on it, or a boundary can be drawn with it. Words like *robust*, *scalable*, and *modern* do none of the three, so they are refused by name.

✅ What to Check

- Open the file your terms live in. Can a reader find the constraint behind each one, or only the word?
- Pick a term you use often. Name the check that would fail if someone broke what it means.

---

3. The Shape of the Code: Context by Design

Vocabulary names the concepts. The shape of the code is what the assistant actually loads, so the layout is not only a human collaboration surface — **it is the strongest piece of context you supply.**

Human-readable code is no longer sufficient on its own. A bounded reader — an AI agent — does not navigate the way a person does. A person jumps around an editor, holds a mental model, and asks a colleague. An agent reads in linear chunks, stops at a context-window limit, and will not ask a clarifying question unless it is told to.

```
flowchart LR
    H["human reader<br/>holds history and context"] -.->|"infers from memory"| C[code]
    A["agent reader<br/>holds only what it retrieves"] -->|"follows what is written"| C
    C --> R["a correct change<br/>or a confident invention"]
```

What this shows: the same code, two readers — and only one of them can supply context the code does not state.

Four rules carry most of the benefit.

Folder as Context

Grouping files by technical layer (`controllers/`, `services/`, `models/`) forces a reader to cross the whole repository to understand one feature, exhausting the context window before the task starts. Group by feature or domain instead, so one directory answers one question:

```
infrastructure/
  network/
    main.tf             # the resources
    variables.tf        # inputs, with their constraints
    outputs.tf          # the contract other modules consume
    network.tftest.hcl  # verification
```

> **Key Insight:** The boundary of the directory is the boundary of the reasoning task. What sits inside one folder is exactly what one request has to hold. This is *separation of concerns*, expressed where the agent actually reads it.

Explicit Over Clever

An assistant predicts the most likely next token. If names are abbreviated or abstract, that prediction has to guess. Descriptive, slightly boring names align the agent with your domain and reduce invention.

Weak	Strong	Why
`module "x" {}`	`module "private_subnets" {}`	the call states what is being built
`var.a`	`var.ingress_cidr_blocks`	the name is the documentation
`# loop over zones`	`# one NAT gateway per AZ: the audit log joins on the zone name`	the comment carries a constraint

Types as Guardrails

A type is not documentation — it is a constraint on what can be generated at all. A typed variable with a validation block closes off whole categories of wrong output before the plan runs:

```
variable "instance_count" {
  type = number
  validation {
    condition     = var.instance_count > 0 && var.instance_count <= 10
    error_message = "instance_count must be between 1 and 10."
  }
}
```

Types shrink the generation space to a valid subset. In a language without types, the boundary is a schema or a validator; the principle is the same.

Comments Explain Why, Not What

Do not spend context restating what the code already says. Spend it on what the code cannot say: the business constraint, the workaround, the reason a decision was made. That is the context an assistant cannot recover by reading the file.

✅ What to Check

- Does each feature folder answer “how does this work?” without forcing the reader to leave it?

- Are your types and names descriptive enough that an agent guesses their purpose correctly on the first try?

---

4. Enforcement Is the Language's Grammar

A convention in prose competes with every other instruction in the context. A convention in a check competes with nothing.

Convention	Enforced by	If it is not enforced
naming	lint rule	a plausible name that means something else
types	the type checker in the gate	`any` where the agent gave up
folder shape	a structure test	a new folder per feature, named by taste
dependency direction	an architecture test	an import that closes a loop nobody chose
contracts	a schema validator	a field added on one side of a boundary

> 💰 **Cost Check**: an unnamed constraint is paid for at review. A vague request costs one round of line-by-line checking; a named constraint costs one sentence at request time and one check at review. The expensive version is the one that reads as plausible and has no stated criterion.

The point is not policing. **It is that the enforced set is the set an agent can rely on.** An unenforced convention is a rule the agent must guess at, and a guess in a generated diff is a review burden with no signal.

One canonical config, thin per-repo adapters. Ten repositories with ten lint configs produce ten dialects of the same language. One canonical config with a declared exemption beats ten opinions.

✅ What to Check

- For each convention in your conventions file, name the check that fails. If none, it is a wish.
- Count your lint configs. More than one canonical configuration is a dialect problem.

---

5. Build the Language

This is the build: four artefacts, four shapes. Keep the keys; replace the tool names with your stack's.

Artefacts

Artefact	What it is for	Where it lives
conventions file	the language stated once, in one place	`docs/conventions.md`, linked from the Map
canonical config	lint, format and type rules shared across repositories	`config/quality/`, vendored or a shared package
architecture tests	the dependency rule, checked	`test/architecture/`
conformance check	the conventions file, tested	the commit or release gate

The conventions file, as a shape — one section per concern, each with the rule and how it is enforced:

```yaml
# docs/conventions.yaml — the language, stated once
naming:        { rule: "<behaviour, not pattern>", enforced_by: "<lint rule>" }
types:         { rule: "<no untyped public boundary>", enforced_by: "<type checker>" }
folders:       { rule: "<feature slicing, one obvious way>", enforced_by: "<structure test>" }
dependencies:  { rule: "<direction>", enforced_by: "<architecture test>" }
contracts:     { rule: "<beside the implementation>", enforced_by: "<schema validator>" }
exemptions:
  - { id: "<stable name>", path: "<where>", reason: "<why>", owner: "<who>", expires: "<date>" }
```

The architecture test, as a shape — a rule, not a diagram:

```
# test/architecture/no-inward-dependency.test
domain/* may not import adapters/* or infrastructure/*
a failure names the importing file, the imported path, and the rule it broke
```

Sequence

1. **Write the conventions file** — five concerns, each with its rule. *Checkpoint: every rule names its enforcement.*
2. **Vendor the canonical config** and point every repository at it. *Checkpoint: one config, declared exemptions only.*
3. **Write the architecture test** for the dependency rule. *Checkpoint: a planted inward import fails it.*
4. **Add the conformance check** to the gate. *Checkpoint: the gate fails on a broken convention.*

Done When

- A reader can learn the language from one file, and the file names the check for each rule.
- A planted violation of the dependency rule fails before review.
- No repository carries a second lint dialect without a declared exemption.

---

6. 🔑 Chapter Summary

The codebase is the prompt, and the prompt is only as clear as the vocabulary and the shape of the code.

1. **The constraint you cannot name, you cannot request.** Vocabulary is the structure imposed on a task before reasoning begins.
2. **Review becomes checking, not auditing.** A constraint with a name can be verified; an unnamed intention can only be judged.
3. **The shape of the code is context.** Feature-shaped directories, explicit naming, strict types and "why" comments shrink the generation space and bound the reasoning task.
4. **Enforcement is the grammar.** An unenforced convention is a suggestion; an enforced convention is a boundary both a person and an agent can rely on.
5. **Four artefacts build the language:** the conventions file, the canonical config, the architecture tests and the conformance check. Each needs a shape, and each needs a check.

Chapter 4: The Command Surface

A Thin Layer Over the Tools You Already Drive

---

1. Why the Repository Needs a Command Surface

The repository from the last chapter can hold the rules and still not run them. A rule that lives in a document is read by whoever chooses to read it, and enforced when someone remembers. Engineering work is defined by repetition, and every repeated operation has two halves: the mechanical execution of the change, and the enforcement of the rules that govern it.

Specialised tools handle the mechanics well — version control, the cluster, the cloud — and none of them knows your rules. A gate must pass before anything ships. A release needs a clean tree. Three components ship in an order. A person must approve. None of that lives in the tool. It lives in memory, becomes a checklist, then a habit, then the single point of failure on the day the person who knows the sequence is away.

A command surface is how that policy gets an execution path. It is not a bag of shortcuts, and it is not a place to reimplement a vendor. It is a software system in its own right, and it should be held to the standards the rest of the estate is held to: clear boundaries, one source of truth, small parts, and tests that fail when the thing they guard breaks.

Structured that way, the surface is where the three structures stop being advice. The workflow structure becomes a gate that runs; the engineering structure becomes a boundary the verb enforces; the prompt structure becomes the policy the surface refuses to break. A person, an agent and a pipeline run the same sequence, and none of them has to remember it.

The outcome is the one every Orchestrator is after. When policy lives in the interface, the effective way to work becomes the efficient way to work: the engineer stops spending attention on flags and ordering, and the surface stops letting a step be skipped. That is speed without giving up governance.

What follows is about designing that layer — telling a command that carries policy from one that duplicates a tool, keeping the surface small and learnable, and putting each part where it can be checked.

> **The rule this chapter defends:** use what exists, curate what you use, and build only what nothing else can do.

🎯 What You'll Learn

- **The purpose of the command surface** — how a structured interface bridges raw execution engines and the policy they cannot carry
- **The gap test** — the five questions that decide whether a command deserves to exist as code, or should be left to the vendor command
- **The structure of a learnable surface** — how discovery, authority and refusal keep a command surface honest and machine-readable
- **The commands worth knowing** — the version control and workflow verbs that carry discipline

The Words This Chapter Uses

Seven words do most of the work, and each means one thing:

Term	What it means here	What it is not
estate	everything one person operates: the repositories, services and hosts — the *whole* surface	not the gates, not the tooling, not one repository
command surface	the text interface — noun, verb, arguments — that a person and a machine share	not the client, not a log
view	whatever renders the surface's state: a dashboard, a console, a terminal — the *view* of the book's rule / mechanism / view	never the owner of behaviour
verb	one operation, declared in the registry	not a wrapper; not a vendor command
wrapper	a verb that stands for a vendor invocation — *policy* or *effectiveness*	not a reimplementation
registry	the one table that declares every verb	not help, not a completion list
refusal	a designed non-action: a named reason and exit 2	not a failure (exit 1)

After its first use, **surface** means *command surface* and nothing else. The rendering side is always **the view** — so a sentence that says "the surface shows" is wrong, and "the view shows" is right.

> **One footnote.** The estate's CLI also has a `client` noun, meaning an application registered with the identity provider. It has nothing to do with a view — which is why this chapter does not use the word.

The Two Surfaces, One Question

The terminal is the view you already have, and it shows state badly: state scrolls away, a week of runs is not a slope, and a command answers the question you asked rather than the one you should have asked. The view's contribution is **legibility** — a verdict as one row, a trend as a line, the action beside the state that made it matter.

None of that requires the view to own behaviour. **A view is a client of the command surface, not a peer of it** — in the client–server sense: a peer negotiates, a client calls. And only the command surface is shared by a person and a machine: the line a person types is the line an agent composes, and its output is the text an agent reads back.

The failure mode is a view built for its own sake: a page per verb, a button that reimplements the operation behind it, a status colour computed by the screen. Each is a second implementation of something one call away — correct when written, wrong within a quarter, invisible, because a screen cannot disagree with itself.

✅ What to Check

- Name three things you do that no vendor command covers. If you cannot, you do not need a command surface yet.
- Name one command you have that is just a shorter `git` verb. That is the first thing to delete.

---

2. The Gap Test

A command surface grows by accident: someone needs a shortcut, adds a verb, and the surface becomes a second vocabulary with no owner. The gap test is how a surface stays small enough to trust.

#	Question	If yes	If no
1	Does a vendor command already do it?	continue to 2 — *not* an automatic no	**candidate**
2	Does it span more than one repository, tool or vendor?	**candidate** — no vendor owns the whole	continue
3	Does it carry policy the vendor cannot know — gates, staged-only, review, approval?	**candidate** — the policy is yours	continue
4	Does it operate on an estate object with no command of its own?	**candidate** — nothing else can	continue
5	Is it **frequent, long to type, and easy to get wrong** — with a short equivalent?	**candidate** — effectiveness is a reason	**documentation, not a verb**

The counter-test, restated. A verb that is one vendor call needs **two of three**: high frequency, a long or error-prone invocation, and a short unambiguous equivalent. Miss one and it is a line in the curated list — write that line, not the verb.

Prefer an argument to a verb. `orch log N=3` is one verb with an argument, not `orch log3`, `orch log5`, `orch log10` — and the surface grows in verbs, which is the expensive axis.

And the test is whether you would miss it. If the long form is typed weekly and you still look up the flags, the wrapper has earned its row. If you would not notice it gone, it is a line in the curated list. Nothing keeps a wrapper alive but use.

Four Reasons a Verb Exists

Reason	What it means	Example
It carries policy the vendor cannot know	the operation is a vendor verb plus a rule you own	`orch commit` runs the repository's staged gate, then commits **only** the staged index, with a message built from the diff
It spans more than one tool or repository	no vendor owns the whole operation	`orch sync` fetches and rebases in one step; `orch repo clone-all` clones every repository the caller can reach
It operates on an estate object with no CLI	the Law, the Map, runners, clients, workflows	`orch runner status`, `orch law check`, `orch client list`
It is more effective than the raw command	high frequency, long invocation, one short equivalent	`orch log N=3` instead of `git log --oneline --graph -n 3` typed forty times a week

The fourth reason is the one teams feel first and defend last, so it needs a standard rather than a preference: **frequency × recall cost × error cost**. A command typed daily with four flags whose order you have to remember is a candidate. A command typed monthly is not, however long it is.

And a wrapper must **teach the command it wraps**: its help prints the vendor invocation it stands for, so a reader can use the raw command when the wrapper is unavailable, and so the wrapper cannot quietly diverge into a reimplementation.

The Test, Applied

Candidate	Passes on	Verdict
`orch commit`	policy: the staged gate, staged-only, a message from the diff	**build**
`orch repo clone-all`	spans repositories, reads the manifest	**build**
`orch runner status`	estate object: runners	**build**
`orch log N=3`	effectiveness: the raw form is `git log --oneline --graph -n 3`, typed daily	**build** — if the frequency is real
a one-off `git log` with a flag you use twice a year	nothing	**do not build**; curate the flags instead
a "status" that re-renders `git status`	nothing	**delete it**

The three that pass are the whole reason the surface exists. The two that fail are the reason it stays small.

✅ What to Check

- Take your five most-used commands. Run each through the four questions. How many survive?
- Find one command that is a wrapper with no policy. What would break if you deleted it?

---

3. The Structure

One Grammar

Every command in the surface has the same shape:

```
<namespace or nothing> <verb> [KEY=VALUE …] [--flags]
orch   repo    clone  REPO=portal
orch   status
gh   pr      create --fill
```

The grammar is not decoration. A stable grammar means a person can guess, a machine can compose, and help can be generated from one table.

The Registry

One table declares every verb. Seven columns, one row per verb:

```
noun | verb | state | needs_repo | script | mode | help
```

Column	Why it exists
noun	the object, or – for a top-level verb
verb	the operation, one per row
state	shipped or planned — so a planned verb is visibly not a capability
needs_repo	whether the caller's repository is resolved first
script	the one implementation
mode	how arguments are passed
help	the sentence help renders

Everything downstream reads this table: dispatch, help, completion, the view's action inventory, and the conformance gate. That is the chapter's first structural claim — *one table, many renderings* — and the reason §6's checkpoints exist.

Three Properties of an Instrument

Property	The question it answers	What fails without it
Discovery	*What can this surface do, from here, right now?*	the reader learns it from a stale wiki
Authority	*What does it treat as true?*	two sources of truth: a file and a running system
Refusal	*What will it not do, and does it say so?*	a guess — the surface acts, and the damage is done

Discovery is the surface describing itself **for the code in front of you**; authority is the decision about which fact wins; refusal is a designed non-action, with a reason and an exit code.

> 🤹 **Key Insight — the three are one decision seen three ways.** A surface whose vocabulary is stored once can generate its discovery; one that knows which fact is authoritative can refuse an input that would need a guess; and one that refuses can be driven by a machine, because a non-zero exit is information rather than a mess.

The exit codes are part of the contract. Zero for success, one for failure, two for a refusal — bad input, refused by name. A person and an agent branch on the same number, which is why a refusal must never be reported as a failure: the two need different responses. And a run that has not finished is not a success; a "still running" state that returns zero is a green nobody measured.

A mutation previews before it writes. A mutating verb prints its resolved plan — the ref, the inputs, the exact thing that would happen — and changes nothing until told to. The preview shows the engine's own resolution, not the view's summary of it.

The Ownership Test

One test settles which side owns the behaviour, and it runs in both directions. Call it the **deletion test**.

Delete the view. If a capability disappears, the view owned behaviour. If only legibility goes — the trend, the glance, the colour — it was a view.

Delete the command surface. If the view still works, it was a peer with its own copy. If everything that mattered stops — gates, dispatch, mutation — the engine was where the work lived.

The Ways a Surface Lies

The three properties have seven known failure modes, and the rest of this chapter is designed around them:

# Mode	Property lost	What it looks like	The check
1 a second source of truth	discovery	the same fact written twice, and the two disagree	delete one; does anything break?
2 duplicated behaviour	discovery	two implementations of one operation, drifting	the deletion test, run in both directions
3 a wrong-direction claim	authority	*"not read"* reported as a verdict	is every unmeasured state named as unmeasured?
4 substring classification	authority	a name test standing in for an identity test	does the match use an id, or a fragment of a name?
5 a silent skip	refusal	a check reporting success about input it never saw	feed it nothing; does it pass?
6 unparsed input	refusal	accepted now, parsed later, where you cannot test it	is every argument parsed before work starts?
7 under-scope by name	refusal	the name promises more than the body runs	compare the name to the operations it performs

Each is structural: the code is sincere, the review passed, and the output is wrong. The fix is never "be more careful" — it is a change to **where the fact lives** or **what the surface accepts**. §6's checkpoints exist because each one happened.

The fourth is worth one case. A bootstrap command decided whether a file was a generated stub by searching its text for a filename. The rules file contained that string — it is the file that says *read the rules* — so the command classified the rules file as a stub, replaced it with a redirect, and reported success. **143 lines became 17**, and nothing objected, because the only check was the one that had just misfired. The guard that came out of it is structural identity: compare the resolved path, never the text, and refuse when the target *is* the file you are installing.

✅ What to Check

- Is there one table, and does every rendering read it?
- Does a mutating verb print a plan before it acts, and refuse bad input by name with exit 2?
- Take the last wrong output you saw from a tool. Which of the three properties failed?

---

4. The Commands Worth Knowing — Version Control

This is the list a reader should learn. Not every flag — the vendor's page has those — but the commands that change the outcome of a normal week.

What you are doing	The command	What it buys you
stage deliberately	`git add -p`	partial staging is how a commit becomes reviewable
read what you are about to commit	`git diff --staged`	the staged index is the change; the working tree is still a discussion
write a message that explains why	`orch commit`	the estate's policy: staged gate first, staged-only commit, message from the diff
see what changed on a branch	`git log --oneline --graph @{u}..`	the shape of the change without the noise
find when a line appeared	`git log -S"<text>" --oneline`	searches the diff, not the message
find the change that broke it	`git bisect run <script>`	a bisect is only fast when the test is scripted
see why a line is here	`git blame -w -C`	`-C` follows moved code — the difference between blame and noise
recover a lost commit	`git reflog`	the last resort that is not a resort at all
rehearse without disturbing the tree	`git worktree add`	a second checkout, no stashing
publish a version	`orch release`	the estate's policy: gate, clean tree, signed tag, push the tag only

Two Kinds of Wrapper, and the Line Between Them

The estate's command surface holds a small set of verbs over git, and they are not the same kind of thing. Two carry policy; six remove a decision you would otherwise get wrong.

Policy wrappers — `orch commit` and `orch release`. *Commit* is `git commit` plus three rules: the repository's staged gate must pass, only the staged index is committed, and the message is built from the diff. *Release* is policy too: gate, clean tree, signed tag, push the tag only. Neither rule belongs to git, and `git tag` cannot know that a tag here means *this shipped*.

Effectiveness wrappers — `orch log`, `orch status`, `orch diff-staged`, `orch last`, `orch branches`, `orch unstage`. Each stands for a long, frequent, easy-to-misremember invocation, and each prints the raw command it stands for.

The wrapper	What it stands for	Why it earns a row
`orch log N=3`	`git log --oneline --graph -n 3`	typed constantly; four flags to recall
`orch status`	`git status` plus the suggested next action	the most typed command, and the advice is the point
`orch diff-staged`	`git diff --staged`	one mental step removed from the wrong diff
`orch unstage`	`git restore --staged`	the flag nobody remembers

The line between them is evidential, not moral — the two rules in §2 bound this table: an argument beats a verb, and nothing keeps a wrapper alive but use. A policy wrapper exists because the rule cannot live in git. An effectiveness wrapper exists because the invocation is long, frequent and error-prone — and if that stops being true, the wrapper should be deleted rather than defended.

Both kinds keep one promise: **the wrapper teaches the command it wraps.** Help prints the raw invocation, so the surface is never a dead end: the raw command stays readable and usable on its own.

✅ What to Check

- Which of the ten do you actually use? The rest are the ones to learn.
- For each estate command, name the rules it adds. A wrapper with no rules is a shortcut, and shortcuts do not earn a surface.

---

5. The Commands Worth Knowing — Workflow

Version control records intent. **Workflow decides whether it may land** — and the commands below are the ones that make that decision legible rather than something you hope about.

What you are doing	The command	What it buys you
open the change with its intent	`gh pr create --fill`	the body is built from the commits — a claim a reviewer can check
know whether it may land	`gh pr checks --watch`	the verdict, live, instead of refreshing a page
read a failed run	`gh run view --log-failed`	the failing step, not the whole log
retry only what failed	`gh run rerun --failed`	no full rebuild to learn one thing
trigger a workflow from the terminal	`gh workflow run`	the same trigger the UI uses, in a script
dispatch with the estate's preflight	`orch workflow run`	parses the workflow before dispatch, so a broken file fails here rather than on the runner
see the estate's declared workflows	`orch workflow list`	the inventory, from the registry, not a wiki

The Habit Behind the List

Two of these are habits, not commands.

Read the failing step, not the log. `--log-failed` exists because most of a log is irrelevant to the failure; reading the wrong part is how a 30-second diagnosis becomes an hour.

Rerun only what failed. A full rerun hides whether the failure was flaky and costs the pipeline again. `--failed` is the honest retry.

And one boundary: **the gate's verdict is the authority, not the pipeline's colour.** A workflow that finished is not a workflow that passed; a check that never ran is not a check that passed. That distinction is the subject of §7.

Mechanism and View

The runner, the queue and the log viewer are **mechanisms and views** — adopt them, never rebuild them. What you own is the *declaration*: which workflows exist, what triggers them, what each one must prove. That declaration lives in the repository and is read by `orch workflow list`; the pipeline renders it.

✅ What to Check

- Does your surface tell you *whether the change may land*, or only that a job finished?
- Can you read a failure without opening a browser?

---

6. The Build Order

> 💰 **Cost Check**: a verb you build costs a row, an implementation, a test and a help page — and every vendor change it wraps is now your problem too. A command you learn costs an afternoon and keeps paying. The expensive surface is the one that copied a vendor verb: it saves a keystroke and adds a second thing to maintain, and the second thing is the one that drifts.

The order is not theoretical. It is the order this estate's surface had to be rebuilt in after the failures the appendix records, and each stage sits where it does because the alternative produces one of them.

```
flowchart LR
    R["1 · gap list<br/>what deserves a verb"] --> G["2 · registry<br/>one table"]
    G --> V["3 · verbs<br/>one operation each"]
    V --> H["4 · help<br/>rendered, never written"]
    H --> D["5 · view<br/>calls the verbs"]
    D --> C["6 · guard<br/>conformance gate"]
```

Step 1 — The Gap List

Before a line of code: the operations that pass the gap test, and the ones that will stay documentation. **Checkpoint:** every candidate has a reason from the four questions, and every refusal names the vendor command it defers to.

Step 2 — The Registry

One table, seven columns, a row per verb. Everything downstream reads it.

Checkpoint: every row resolves — a shipped row has an implementation that exists and is executable, no verb is declared twice, a planned row says so. This is a gate, not a review.

Step 3 — The Verbs

Each verb is one operation and owns three things: the work, the guard and the intent. The guard belongs *inside* the operation — a guard in the caller is a guard on one surface; a guard in the operation is a guard on every surface that will ever exist.

Checkpoint: every shipped verb refuses an unknown argument with a named reason and exit 2, and every mutating verb prints its resolved plan without acting. Prove both with a fixture per verb, not with a sample.

Step 4 — Help

Help is a **rendering of the table**, never a document beside it. Arguments, examples and the verb's intent are composed from the registry row and the implementation's own header, so `help --all` describes the surface that exists.

Checkpoint: help renders every declared verb and nothing undeclared. A missing verb is an undocumented capability; an extra verb is a page for something nobody built. This estate's conformance gate checks both directions — which is how it found two rows it had been skipping.

Step 5 — The Client

The view comes fifth because it can only call what exists. Its job is legibility plus a bounded set of actions, each naming the operation it runs.

Checkpoint: every view action resolves to an operation that exists, and deleting the view removes only legibility. Run the deletion test as a gate, not a thought experiment.

Step 6 — The Guard

The conformance gate comes last because it judges what exists — and because it must be able to fail. It asserts that every declared verb is implemented, unique, documented and dispatchable, that no retired name survives in the live instructions, and that the advertised and enforced surfaces are the same set.

Checkpoint: the guard fails on a deliberately planted defect, and the failure is recorded.

What Breaks If You Reorder It

Move	What breaks	The symptom you see first
Help before the registry	help becomes the definition; every verb edits two places	a documented verb nobody built
View before the verbs	the view becomes the first implementation	a terminal that cannot do what the button does
Guard before the verbs	the gate asserts things about a surface that does not exist	a green run over an empty set
Verbs before the registry	the verb list lives in the dispatcher and in help	a verb no help page mentions
A guard that never judges the view	view actions have no inventory to resolve against	a button whose command was renamed last month

The Checkpoint Habit

A checkpoint is not a milestone; it is a **measurement that can fail**, taken when a step claims to be done. All six share a shape: state a set, compute what the code actually has, compare, refuse on a difference, record the result. That shape is what lets a stage be *finished* rather than merely *written*.

✅ What to Check

- Count your registry rows, then ask the surface how many verbs it has. A difference is step two, unfinished.
- Ask one shipped verb to refuse an unknown argument and one to print its plan, and ask the view to name the operation each action resolves to. If any cannot, that step is unfinished.

7. Seeing Widely, Acting Narrowly

Why the Two Jobs Split Apart

For years the two jobs lived apart. A **dashboard** read state: charts, colours, a number per panel — and it could not touch anything, so nobody had to think about its authority. A **terminal console** acted: you typed a command and the system changed, and the only thing between your typo and production was your own attention.

Operator consoles merged the two: one surface that reports *and* acts, with buttons beside the numbers. That is convenient, and it creates a question the old split never had to ask — **when a screen can both show and do, what stops the doing from being as casual as the showing?**

The three properties from §3 make the answer unavoidable. **Discovery** says the console must describe what it can do *for the code in front of you*; **authority** says it must decide which fact wins instead of rendering a second one; **refusal** says it must decline what it cannot resolve. A surface that does those three things honestly can be trusted with a button. One that does not can be trusted with nothing.

So this section is the console's half of §3: what the three properties demand when the rendering side is also the side with hands.

A console that shows you everything and changes nothing is a perfectly good console. A console that can also change things is a different instrument with a different risk, so it needs one rule: **show as much as you like; do only what a person asked for.**

The Two Jobs Carry Opposite Risks

	Showing — reading state	**Acting — changing state**
What it is	what exists, what passed, what failed, which step, how old the fact is	dispatching a deploy, provisioning, deleting, staging a commit, bypassing a gate
If you widen it	nothing breaks; the worst case is a crowded screen	authority grows, and every action is a decision nobody makes any more, because it already happened
If it is wrong	the reader learns something false and asks a question	the system has changed, and the correction is a second action
The safe direction	**generous**	**narrow, named, confirmed**

That asymmetry is the whole section. Showing more is cheap and reversible; acting more is neither.

Three Rules

1 · Show everything you can read — and say when you cannot. Three outcomes, never two: pass, fail, not-verified. A missing row is *unknown*; a verdict about another commit is *history*; a check that could not read is *not-verified*; a gate that has not run is *not-run*. A green colour in place of any of these is the lie this rule forbids.

2 · Act only through a verb that already exists. The view holds no behaviour. Every control resolves to a command, so the guard that command carries applies whoever pressed it. A button that reimplements the operation is a second implementation, and §3's deletion test finds it.

3 · Stop at ambiguity; never pick. If the input can be read two ways, the surface names the candidates and exits non-zero. It does not choose the likeliest, retry with a correction, or map a renamed verb to whatever looks closest. A choice made by the surface is invisible in the invocation and wrong the day the repository changes.

What Requires a Person

The console shows	**The action that needs a person**	**Why**	**Enforced by**
state — what exists, in which environment	**dispatch** — a live deploy or provisioning run	it mutates a provider; the decision is the Orchestrator's	convention — no hook yet
the verdict — pass, fail, not-verified, not-run	**staging and committing**	the ship boundary is a human signature	staged gate and review
the failing step — job, step, error excerpt	**bypassing a gate**	a gate is the promise; bypassing it is a silent renegotiation	the gate itself
the age and subject of each fact	**choosing between two interpretations**	the choice is invisible in the invocation and wrong later	refusal by name, exit 2
the refusal — its reason and exit code	**retrying a failed mutation without a new decision**	the second attempt is a new decision wearing the first one's clothes	convention — no hook yet

Why This Lives in the Verb

A view that can dispatch holds authority. Authority is not a property of a widget; it is the power of the operation behind it. So the rule is enforced where the operation lives:

A mutating verb prints its resolved plan and changes nothing until told to. The plan is the engine's own resolution — the ref, the inputs, the exact operation — and the view shows that plan, not a summary of it. The person confirms a line the engine would actually run.

That is why "the console must not own behaviour" is a live rule rather than a truism: a console *can* act, and the only thing standing between the button and production is that the button calls a verb that refuses, previews, and records.

The Reader May Be an Agent

Every rule above was written for a person at a screen, and each binds tighter when the reader is a machine.

An agent reads the console and acts through it, at a speed no one reviews. If the console shows a **colour where it should say not-verified**, the agent proceeds on a fact nobody measured. If the console **picks between two interpretations**, the agent inherits the choice and never sees that a choice was made. If the console can **dispatch without a preview**, the agent's first attempt is already in production.

That is the background to this section's shape: the console is the agent's eyes and its hands, so the showing side must be honest about what it does not know, and the acting side must be narrow enough that a wrong step is one step, visible and reversible.

The Honest Limit

Two of the five boundaries above are enforced by convention rather than by a mechanism: **dispatch** and **retrying without a new decision**. The command refuses nothing when an agent runs it; the rule holds because the agent reads the Law and the human reviews the tree.

Stated plainly, because a boundary that is only prose is a future version of **under-scope by name**: the surface's name promises a control its body does not enforce. The fix is already named in the table — a hook that refuses the action regardless of who is typing.

✅ What to Check

- List every control in the console and mark each **show** or **act**. For every act, name the verb it calls — an act with no verb is a second implementation.
- Ask what the console returns when it cannot read. If the answer is a colour rather than *unknown* or *not-verified*, rule 1 is not implemented.
- Pick an ambiguous input and type it. Does the surface name the candidates and refuse, or does it choose?

---

8. Build the Workbench

The build order is the argument. This is the build: six artefacts, each with the shape that makes it checkable. Keep the shapes; put your own operations in the rows.

Artefacts

Artefact	What it is for	Where it lives
gap list	which operations deserve a verb, and which defer to a vendor command	`docs/gaps.md`
verb registry	the only list of what the surface can do	`cli/registry.tsv`, one row per verb
argument contract	what each verb advertises and what it enforces	inside each verb's help, derived from the registry
help renderer	renders the registry — never written by hand	one script, one output
curated list	the vendor commands worth knowing, and what each buys	`docs/commands.md`
conformance guard	proves the surface is whole, and fails on a planted defect	the gate

The registry, as a shape — seven columns, nothing else:

```
# cli/registry.tsv — the only membership source
noun    verb    state   needs_repo  script  mode    help
-   status  shipped 1   libexec/status.sh   kv  what is staged, and the suggested next action
-   commit  shipped 1   libexec/commit.sh   kv  staged gate, then commit only the staged index
repo    clone-all   shipped 0   libexec/ws-run.sh   kv  clone every repository you can access
```

The curated list, as a shape — one row per command, and the reason it earns its place:

```
# docs/commands.md
task            command             what it buys
stage deliberately  git add -p          partial staging is how a commit becomes reviewable
read the change     git diff --staged       the index is the change; the tree is a discussion
find the regression git bisect run <script>     a bisect is only fast when the test is scripted
may this land?      gh pr checks --watch        the verdict, live, instead of a refresh
```

Decisions

Choice	Default	What changes if you choose otherwise
build or curate	curate first; build only what passes the gap test	a surface that is a second vocabulary
one verb per operation	yes	two verbs for one operation produce two behaviours that drift
the registry as the only list	yes	a completion script or a doc that keeps its own list is a second source of truth
help rendered from the registry	yes	hand-written help goes stale on the first added verb
arguments advertised by name, exit 2 for a refusal	yes	a caller cannot tell "fix the input" from "try again"
mutations previewed by default	yes, write only on an explicit flag	a preview that writes is a trap

✅ Checks

- Every registry row **resolves**: an implementation exists, is executable and is dispatched.
- An unknown argument is **refused by name**, with a fix line and exit 2.
- **Help renders every declared verb and nothing undeclared.**
- Every view action resolves to a verb that exists; deleting the view removes no behaviour.
- The guard **fails on a planted defect** (a row with no implementation, a verb with no test).
- Every **effectiveness wrapper** prints the raw command it stands for.
- No two verbs differ only by an argument that could carry the difference.

Sequence

1. **Gap list** — run every candidate through the four questions. *Checkpoint: each refusal names the vendor command it defers to.*
2. **Registry** — one table, seven columns. *Checkpoint: every row resolves.*
3. **Verbs** — one operation each, refusing unknown arguments by name. *Checkpoint: refusal by name, and a plan before any write.*
4. **Help** — rendered from the table. *Checkpoint: help and the table agree in both directions.*
5. **Client** — one action per operation, holding no behaviour. *Checkpoint: the deletion test removes only the surface.*
6. **Guard** — the conformance gate, with a planted defect. *Checkpoint: the guard fails on that defect, and the failure is recorded.*

Done When

- The gap list exists, and every verb on the surface can be traced to one of the four questions.
- The registry is the only list of what the surface can do.
- A wrong argument is refused by name, with a fix line and exit 2.
- Deleting the view leaves every behaviour intact.

---

9. 🔑 Chapter Summary

Use the command surface you already drive; build only the layer that carries your policy.

1. **The vendor tools win their own domains.** A surface that reimplements one is a liability.
2. **A verb exists for one of four reasons:** it carries policy, it spans tools, it serves an estate object, or it measurably beats typing the raw command. Otherwise it is documentation.
3. **One grammar, three properties** — discovery, authority, refusal — and seven known ways to lose them, each with a checkpoint.
4. **Learn the commands that carry discipline**: partial staging, the staged diff, a message that explains why, `-S` and `bisect run`, `--log-failed` and `--failed`. That list is the chapter's practical half.
5. **Build last, and build small**: most of what the surface offers should be a vendor command you learned to use well. The verbs you write are the few that carry policy or span tools — and each one costs a row, an implementation, a test and a help page.

Chapter 5: Context Routing

Choosing What the Agent Reads

---

1. Context Is Finite and Costly

Every AI-assisted change begins with a retrieval step. Before an agent writes a line, it reads files — and those files become the evidence it reasons from, the constraints it enforces and the patterns it follows.

That reader's window is finite. Every file loaded competes for attention with every other file, and every token spent on noise is a token not spent on reasoning. **Context economy** is the name for the discipline that follows: context is a finite resource, managed deliberately. It is not a retrieval optimisation. It is a first-class architectural concern: decided once, and binding on every task of that type. A **task** is one piece of work an agent is asked to do. A **task type** is a category of tasks that share the same context requirements.

Two failures follow from ignoring it, and both are paid at review.

Loading too much. Forty files arrive, most of them irrelevant. The one that matters competes with the other thirty-nine, cost rises, precision falls — and the failure looks like confusion rather than omission. The agent produces a plausible change that violates a constraint in a file it never loaded.

Loading too little. Nothing arrives by default, so the agent asks, or guesses. A guess looks correct and is built on incomplete evidence, which is the failure this chapter exists to remove.

Both have the same root cause: **nobody decided what to load.** The three structures are in place and the repository holds them, but nothing says which part of them a given task needs.

That decision is **context routing**: for each task type, a named list of what the agent reads — declared where the policy lives, mapped where the context lives, and verified in the record of what was actually read. Without routing, context economy is a principle. With it, the window becomes a measurable property of every task. What it costs to fill that window — and how to bound the spend — is the economy chapter's subject; this one decides what may enter.

💰 **Cost Check**: context is the one budget that cannot be seen on a bill. Loading everything costs tokens on every run and precision on every review; loading nothing costs a wrong change and the afternoon spent finding it. The only place the cost is bounded is where the decision is made explicit.

🎯 What You'll Learn

- **Why context is finite and costly** — the principle that makes routing necessary
- **Context routing as a mechanism** — the deliberate, rule-based composition of the context window for each task type
- **Load units as the abstraction** — how to organise context by what a task needs, not by what the codebase contains
- **Verification as the proof** — how to measure whether routing is working, not just whether it exists

✅ What to Check

- For your three most common task types, name the files an agent should read. If the answer is "it depends", you have found the routing gap.
- Count the times an agent read a file the task never needed. That is the cost of loading too much.
- Count the times an agent produced a plausible change that violated a constraint in a file it never read. That is the cost of loading too little.

---

2. Load Units: Organising Context by Task Need

What a Load Unit Is

A **task type** is a kind of work that recurs — changing a migration, adding an endpoint, rotating a credential. A **load unit** is the set of files one task type needs, named once. It is not a folder and not a file — it is a list that crosses the repository: the contract, the service it describes, the policy that constrains it, the runbook for when it fails.

Three properties make it a unit rather than a wish:

Property	The test
named	the files are listed, not described
bounded	the list has a number, and a run can exceed it
stable	the same task type reads the same files on Tuesday as on Monday

The unit is sized by the task, not by the directory tree. One task may span three folders and four hundred lines; another may be twenty files in one place. If a unit names more than about ten files, it is a folder in disguise — the task is probably two tasks, or the unit is the wrong unit.

```
flowchart LR
    T["task"] --> R{"routing table"}
    R -->|"names a load unit"| L["the files that task needs"]
    R -->|"no match"| F["an agent that infers<br/>from whatever it can see"]
    L --> W["a change written from evidence"]
```

What this shows: the routing table is the only thing standing between a task and an inference.

✅ What to Check

- For your three most common task types, name the files an agent should read. If the answer is "it depends", you have found the routing gap.
- Take one of those units and count its files. If the number is above ten, the unit is a folder in disguise.

---

3. The Map Is a Fact Document

The Map answers *where things are and what they are for*. It does not state rules — those are the Law's job — and it does not restate the Playbook. A Map that starts carrying rules becomes a second source of truth, and the two will disagree within a month.

The Map is organised in load units (§2): each section names the files one task type reads, so a reader who knows the task type knows where to start.

Map (a load unit)	Not a Map
"changing a migration: read the schema contract, the migration gate, the rollback runbook"	"the database is important"
a pointer to the file that owns the rule	the rule, restated

The pointer problem. No harness auto-loads the Map. Something must tell the agent to read it — a sentence in the Law, a routing row, an instruction in the harness. A Map nobody is told to open is a document, not a route.

The pointer, as a line — one sentence in the Law, before anything else it says:

```
## Before your first change
Read `AI_CONTEXT.md`. It names where things are; this file names what must be true.
```

Without that line the routing table has no entry point. An agent that never learns the Map exists reasons from whatever it can see, which is the *loading too little* failure with a confession attached.

✅ What to Check

- Does the Map state any rule? Move it to the Law.
- Is there a sentence, anywhere, that tells an agent to read the Map before working? If not, nothing routes.

---

4. The Routing Table Is a Rule

A routing table is a rule in the Law: *before a task of type X, read these units*. Not a suggestion, not a diagram — a row an agent can follow and a check can verify.

Task type	Load units	Budget
change a migration	schema contract · migration gate · rollback runbook	3 files
add an endpoint	API contract · the feature's folder · the auth policy	4 files
change a workflow	the workflow's definition · its gate · the evidence pack	3 files
rotate a credential	the secrets policy · the consumer inventory · the rotation runbook	3 files

The **budget** is a ceiling a run can exceed: three files means three, not *about three*. A row that is routinely exceeded is saying something — either the unit is too big or the task type is really two task types.

✅ What to Check

- Every routing row names concrete files, not a directory.
- The budget is a number you could exceed visibly.

---

5. Verification — What Was Actually Read

The execution log records `files-read`. Compare it with the routing table: what the table said to load, and what the run actually opened.

A mismatch is a **finding, not a footnote**:

Observation	What it means
read fewer than the row names	the row is too big, or the run skipped a file it was told to read
read more than the row names	the row is missing a file the task needs, or the agent read beyond its route
read nothing the row names	the route is not being followed at all

The point is not to catch the agent. It is that **the routing table is only true if the run shows it was followed** — otherwise the table is a claim about intent, and the agent's behaviour is the fact.

One run, in the shape the log produces:

```
task:       change a migration
route:      contracts/schema.md · docs/migration-gate.md · runbooks/rollback.md   (budget 3)
files-read: contracts/schema.md · docs/migration-gate.md                          (2 of 3)
finding:    the rollback runbook was never read
```

That is the shape of a useful finding: a named delta, and a question with two possible answers. Either the row was wrong, or the run did not follow it. Here the row was right, so the run is the thing to fix.

✅ What to Check

- Take one run. List the files it read. Compare with the routing row for that task type. Name the delta.
- Does anything fail when the delta is large? If not, the routing table is a wish.

---

6. Failure Modes

Beyond loading too much or too little, three failure modes are specific to routing:

- **Stale routing.** The Map names a file that moved; the agent reads nothing and infers.
- **Routing in plugin configuration.** A rule that lives in the harness's config cannot be reviewed, diffed or tested — and disappears when the tool changes.
- **A Map that became a second Law.** Two rule sources; two answers; a review that cannot say which is authoritative.

The pattern repeats: **the rule must live where it is reviewed, and the effect must be observable.**

---

7. Build the Routing Table

Artefacts

Artefact	What it is for	Where it lives
Map	where things are, as load units	`docs/map.md` (or the repo's context file)
routing table	task type → load units → budget	the Law, as a rule
budget	one number per row, a ceiling a run can exceed	beside the routing table
conformance check	routing compared to `files-read`	the gate or the nightly audit

The Map skeleton, as a shape — sections are load units, each one a task type:

```
# Map — where things are

## changing a migration
read: contracts/schema.md · docs/migration-gate.md · runbooks/rollback.md

## adding an endpoint
read: contracts/api.md · services/<name>/ · policies/auth.md
```

The routing table, as a shape — a rule, with a number you can exceed:

```yaml
# routing table (in the Law)
routing:
  - task: "change a migration"
    load: ["contracts/schema.md", "docs/migration-gate.md", "runbooks/rollback.md"]
    budget: 3
```

Decisions

Choice	Default	What changes if you choose otherwise
organise the Map by	task type (load units)	a codebase index, which no task can use
where the routing rule lives	the Law, reviewed	plugin config, unreviewable and unportable
the budget	a number per task type	"read what you need", which is not checkable
verification	`files-read` compared to the row	routing that is asserted and never observed

✅ Checks

- Every routing row names files and a number.
- A run's `files-read` compared to its row is **within the budget or reported**.
- No rule lives only in harness configuration.

Sequence

1. **Write the Map** as load units for your most common tasks. *Checkpoint: each unit names files, not folders.*
2. **Move the routing into the Law** as a table. *Checkpoint: a row can be followed without reading anything else.*
3. **Set the budgets**. *Checkpoint: the number is one a run can exceed.*
4. **Add the comparison** to the log or the nightly audit. *Checkpoint: a run outside its budget is reported.*

Done When

- An agent can be routed to a task type by one sentence in the Law.
- A run's reads can be compared with the route, and the delta is visible.
- No rule about routing lives in tool configuration.

---

8. 🔑 Chapter Summary

Routing is the decision that turns a repository into context.

1. **Context is a routing decision**, not a dumping problem — and both "load everything" and "load nothing" are failures.
2. **The Map is a fact document** organised in load units; rules belong in the Law.
3. **The routing table is a rule** — task type, files, budget — and a budget a run can exceed.
4. **Verification is the point**: compare `files-read` with the route; a mismatch is a finding.
5. **Failure modes are familiar**: load-everything, stale routing, routing hidden in tool config, a Map that became a second Law.

Chapter 6: Context Economy

Where the Money Actually Goes

---

1. Two Budgets, One Decision

The monthly bill doubled and nothing in the product had changed. The AI model was the same model, the price was the same price, and the team had shipped fewer changes than the month before. What had changed was the Law: a status table had been added to it, because it was convenient to have the table where everyone already looked. Every session loaded it. Every run paid for it. Nobody had ever decided to buy it.

That is the shape of the problem. The three structures cost money to operate — every rule, every file, every retrieved unit is tokens — and the bill is decided by the **input side**, not by the price list.

There are two sides to that budget, and they are often confused.

The input side is yours: what may enter the prompt, how large the always-loaded set is, how much a task retrieves. The routing chapter decides what may enter; this one prices it, bounds it, and records it. **The market side** is not: what the model charges per token, and how that changes. Designing against the second is a race you lose; designing the first is a decision that holds.

```
flowchart LR
  A["task"] --> R["routing units<br/>the input decision"]
  R --> C["context sent"]
  C --> P["price per token<br/>the market side"]
  R --> RW["rework and retries<br/>the measured cost"]
  RW -.->|"usually larger"| P
```

What this shows: the two budgets — one you design, one you observe — and where the money actually leaves.

> 💰 **Cost Check**: the bill is not dominated by the price of a token; it is dominated by how many tokens a run sends and how many runs it takes. A routing decision that halves a context, and a gate that halves the retries, do more for the monthly number than any model choice — and neither one requires renegotiating anything.

🎯 What You'll Learn

- What may enter the prompt, and why that is a design decision
- How to size context by task scope rather than by repository size
- Where agent cost actually goes — and it is not the per-token price

Where the Money Actually Goes

Claimed cost driver	The measured driver
the per-token price	oversized context on every run
the model tier	rework: the same task done three times
the tool	retries: a failed step re-run from the top
—	a bloated Law loaded into every session

The per-token price is the only line everyone discusses and the smallest lever anyone controls. The large levers are **oversized context, rework and retries** — all three of which the earlier chapters in this part reduce.

✅ What to Check

- Take one task. What did it load, and how much of it was used?
- Count the retries in the last week. That number is a cost driver, and it is a routing or gate problem.

---

2. The Always-Loaded Set

The routing chapter sizes the context per task: a single-file edit, a feature change, a cross-cutting change, each with its own load unit. That is the variable side of the bill.

The always-loaded set is the one to guard. The Law is loaded into every session, so every sentence costs on every run. Bloating the Law with a view — a dashboard description, a status table — is the classic mistake: it is paid for continuously and read once.

Model tiering is for reasoning difficulty, not context volume. Escalating to a larger model to "fit more context" is the expensive way to avoid the routing decision; the routing decision is free by comparison.

The Budget Rules

```
# budget rules — stated once, checked per task type
always_loaded: { max_lines: <n>, may_include: [rules, map-pointer] }
per_task:      { unit_budget: <n files>, escalation: only-for-reasoning }
tiering:       { default: <tier>, escalate_when: <reasoning difficulty> }
```

✅ What to Check

- How many lines is your always-loaded set, and is any of it a view?
- Has any task been escalated to a bigger model because the context did not fit? That is a routing gap.

---

3. Compaction, Summary, Cache

- **Compaction** shortens a session's history so the work can continue. It is a mechanism: adopt it, and never treat its output as the record — the record is the log.
- **Summarisation** replaces detail with a summary. Useful for resumption, dangerous for audit: a summary of a decision is not the decision.
- **Caching** reuses what was already computed. It reduces cost and can hide staleness — a cached read is a read you did not make.

The rule: a mechanism that reduces context must never reduce **evidence** — the continuity chapter's rule, applied to a cost decision. If compaction drops the gate results, the resumed session cannot know what passed.

✅ What to Check

- After a compaction, can the session still name the gates that ran?
- Does any cached value have a freshness rule?

---

4. Recording Spend

Cost is a signal, and a signal needs a record. The execution log carries it: task, tokens or cost, retries, and the model tier. That makes three questions answerable from the repository:

- **which task types cost most** — a routing or sizing problem;
- **which runs retried** — a gate or procedure problem;
- **which sessions were over budget** — a context-economy problem.

Three tasks, as the record holds them:

```
task                tier      tokens   retries   wall
change a migration  standard  41,200   0         6m
add an endpoint     standard  38,900   2         24m
rotate a credential standard   9,400   0         2m
```

Read across, and the three questions answer themselves. **Which task type costs most?** The migration and the endpoint, at four times the credential task. **Which run retried?** The endpoint, twice — a procedure or gate problem, not a context one. **Which session went over budget?** The endpoint again, at more than twice its unit's median, which is a sizing problem.

That is where the two budgets meet: a routing decision moves the token column, and a gate moves the retry column. Read together, they turn cost from a bill into a signal — and each signal points at a different chapter's fix.

A per-tool dashboard is a **view**. Adopt it if it helps, but the log is the record; a dashboard that cannot be reconstructed from the log is a second truth that disappears with the tool.

✅ What to Check

- Can you answer "what did last month cost, by task type?" from the repository alone?
- If that per-tool dashboard were removed, would you still know which runs retried?

---

5. Build the Budget

Artefacts

Artefact	What it is for	Where it lives
budget rules	always-loaded cap, per-task unit budget	the Law
tiering policy	when a bigger model is justified	the Law, beside the routing table
spend record	task · cost · retries · tier	the execution log
cost review	the monthly read of the record	the Playbook

Decisions

Choice	Default	What changes if you choose otherwise
what may enter the prompt	the routing units, nothing else	convenience loading, paid on every run
always-loaded contents	rules and a Map pointer only	a view in the Law, paid forever
when to escalate the model	reasoning difficulty	context volume, which routing should fix
the cost record	in the log, per task	a dashboard that cannot be reconstructed

✅ Checks

- The always-loaded set is **under its cap**, and contains no view.

- A run over its unit budget is **reported**, not silently paid.
- Cost per task is answerable **from the log alone**.

Sequence

1. **Measure the always-loaded set.** *Checkpoint: it is a number, and it contains no view.*
2. **Write the budget rules** and apply them to one task type. *Checkpoint: a run over budget is visible.*
3. **Write the tiering policy.** *Checkpoint: escalation names a reasoning reason, not a volume one.*
4. **Record spend in the log** and read it once. *Checkpoint: three questions answered from the record.*

Done When

- The always-loaded set is small, rule-only and capped.
- Over-budget runs are reported, not absorbed.
- Cost, retries and tier are answerable from the log, without a vendor dashboard.

---

6. 🔑 Chapter Summary

Context is the one budget you design; the price is not.

1. **Two sides** — the input side is a decision, the market side is not.
2. **The real cost is oversized context, rework and retries**, not the per-token price.
3. **Guard the always-loaded set**: the Law is paid for on every run, so it stays rule-only and capped.
4. **Tier for reasoning difficulty, not volume** — volume is a routing problem.
5. **Record spend in the log** so cost is reviewable without the vendor's dashboard.

Chapter 7: Session Continuity

The Handoff Artefact: What the Repository Cannot Tell You

---

1. Memory Is Not a Control

The task was a migration. It ran for an hour, ended on a clean gate, and was picked up the next morning in a different tool. The second session read the changed files, decided the approach was wrong, and reversed two of the three edits. Nothing had failed. The session that did the undoing simply did not know what the earlier one had rejected — and so it weighed the same options again, and this time chose one of them.

The three structures assume such work can be resumed. **Session continuity** is the discipline of making the handoff between sessions explicit, verifiable and repository-owned, so the next session resumes the work instead of reconstructing it.

Chapter 2 introduced the execution log, which records what a session *did*. Chapter 5 introduced context routing, which decides what a session *reads*. This chapter is the bridge between them: the artefact that tells the next session what was decided, so it neither re-reads what it does not need nor re-decides what was already settled.

An agent that resumes without such an artefact does not resume. It **reconstructs**, from whatever context it can retrieve, and the reconstruction is a story about work it did not do.

> 💰 **Cost Check**: a lost session costs the review twice — once to reconstruct what was done, once to catch what the reconstruction invented. The artefact costs a minute at each stop. The arithmetic is not close, and the expensive version is invisible until a change lands from a fiction.

🎯 What You'll Learn

- Why tool-private memory cannot carry a task across tools, days or a crash
- The seven fields a handoff needs, and what each one is for
- Why the repository is the authority for what changed, and the file is the authority for what was decided

What an Interruption Costs

Interruption	What is lost without an artefact
a second tool	the task, the decisions, the files already changed
a night	the next step, and why it was next
a crash	the gate status: what was verified, and what was assumed
a handoff to a recipient who cannot see the file	all of it, unless the artefact is attached to the task

The cost is not the elapsed time. It is that the resumed work has no memory of what it decided, so it decides again — differently.

```
flowchart LR
  S["session 1<br/>task · decisions · files"] --> A["session artefact<br/>the handoff contract"]
```

```
A --> R["session 2 — any tool<br/>resumes from the file"]
A -.->|"no artefact"| G["reconstruction<br/>a story about work it did not do"]
```

What this shows: the artefact is what makes resumption possible; without it, the next session guesses.

✅ What to Check

- Stop a task mid-way today. Could a different tool resume it from what is in the repository? If not, the gap is the artefact.
- Name one decision made in the current session that is not written down. That is what an interruption deletes.

---

2. The Artefact Is a Contract

A Rule, Not a Mechanism

Two things can hold a task together across sessions, and they are not the same kind of thing.

The **session lifecycle** — resume, replay, identity, crash recovery — is a mechanism. Somebody built it, it belongs to a tool, and it stops at that tool's boundary. Adopt it; never rebuild it.

The **artefact** is a rule: a file in the repository, written by the session, reviewable in a diff, portable to any tool that can read. It is yours to define, and yours to enforce. §4 returns to the split at the point it matters — a crash — but the artefact is on the rule side from the first line it contains.

The Seven Fields

A session artefact is a small file with seven fields. Every field exists because a resumer needs it; a field added without a need is a field someone else maintains.

Field	The question it answers	Why it exists
task	what is being done, in one sentence	a resumer who cannot state the task reads everything to find out
status	not started · in progress · blocked · done	the first question of a resume is whether to continue at all
files-read	what was consulted	the next session does not re-read five files to learn what is already known
files-modified	what was changed	the resume starts from the truth, not from a claim about it
decisions	what was decided, and what was rejected	without it the next session re-litigates — the most expensive omission
next-step	the one action to take next	a resume that has to choose its own next step is a restart
gate-status	what has been verified, and what is assumed	a resumed session that trusts an unverified claim is a green nobody measured

The Field That Pays for the Artefact

`decisions` is the field that earns the file. The other six tell the next session where it is; this one tells it what not to undo. The scenario at the top of the chapter is not a hypothetical: a session that does not know a decision was made will make it again, and a coin flipped twice is not a state.

The half that matters is **`rejected`**. `what` and `why` record the choice; `rejected` records the options already weighed and why they lost. A resumer who has that neither re-opens a settled question nor quietly picks up the option that was turned down — which is exactly what the second session in the story did.

The schema, as a shape:

```yaml
# .session/<task>.yaml — the handoff, gitignored
task: <one sentence>
status: in-progress | blocked | done
files_read: [<paths>]
files_modified: [<paths>]
decisions:
  - what: <the decision>
    why: <the reason>
    rejected: <what was not chosen, and why>
next_step: <one action>
gate: { ran: <checks>, passed: <bool>, at: <timestamp> }
```

What never goes in it. No secrets, no transcripts, no persona data. The artefact is read by whoever resumes, which may be a different tool with a different trust boundary.

✅ What to Check

- Does your artefact have a `decisions` field with a rejected option? Without it, the next session re-litigates.
- Does the file contain anything you would not paste into a ticket?

---

3. The Part the Repository Cannot Tell You

A session file looks like state you could simply re-derive, and for most fields you can. Split it by where each field comes from:

Field	Where it comes from
task	the branch name and the commit message, usually
status	the working tree and the gate, partly
files-read	nothing, unless an execution log is kept
files-modified	`git status`, exactly
decisions	nothing — a diff shows what was chosen, never what was rejected
next-step	nothing — it is a judgement about what remains
gate-status	the gate's own output for the checks it ran; the assumptions, nothing

That table is the reason the file exists. **The derivable fields are the cheap ones; the fields a resumer needs are the ones nothing derives.** `git diff` will tell the next session what changed. Nothing in the repository records that the retry loop was rejected because the audit log joins on the request id — and a rejected option leaves no trace to derive from, which is why the only way to recover it is to re-decide — the failure the chapter opened with.

So the file is not a cache of state that git already holds. It is the part of the session that would otherwise be lost, written once and checked against the repository afterwards. That is why it is gitignored and still not the authority: it is cheap to carry, cheap to ignore, and the fields that can be checked against the repository are checked.

Where the file travels. Gitignored means *not pushed*, not *machine-bound*: the file sits in the working tree and can be copied like any other. Its default life is one machine, because that is where the second tool and the crash are. When it has to cross a boundary — any recipient who cannot see the file: a colleague, a review, another machine, a fresh agent — the file is attached to the task or the pull request, and the durable part moves to the record that is committed: the decision is written in the change's description, and the gate result in the execution log. What must never happen is a resumer being asked to continue from a file they cannot see.

The reconciliation rule, stated once: before resuming, compare `files_modified` with the working tree. A file changed but not listed was changed after the artefact was written — reconcile before continuing, and when the two disagree, the repository is what happened.

✅ What to Check

- Does anything reconcile the artefact with `git status` before resuming? If not, the artefact is a memoir.
- Take a stale artefact and find the delta. What did it miss?

---

4. Cross-Tool, Mid-Task, and Crash

Cross-tool handoff is the test: start a task in one tool, stop deliberately, resume in another. If the resume needs the first tool, the continuity was private memory wearing a name.

A deliberate mid-task stop is the cheapest form of the drill: stop at a natural boundary, write the artefact, resume. Do it on a real task, not a rehearsal — the rehearsal always succeeds because the person writing it remembers.

A crash is where the split from §2 shows. Crash recovery — session lifecycle, resume, identity — is a **mechanism**: adopt it. Continuity is a **rule** — the artefact — and it must survive the mechanism's failure.

The Handoff, in Order

A handoff is a procedure, not a snapshot. Five steps, and the first two decide whether the other three are worth anything.

1. **Write the artefact before you stop** — at a natural boundary, not after the crash. A file written afterwards is a reconstruction, which is the failure this chapter opened with.
2. **Record a decision while the option is still rejected.** The moment an option loses, write `what`, `why` and `rejected`. Ten minutes later you will remember the choice and have lost the reasoning; a day later you will not remember either.
3. **Reconcile before you hand over**, not after: compare `files_modified` with `git status`, and fix the artefact rather than the tree.
4. **Promote what must travel.** Inside one machine, the file is enough. Across a boundary, attach it to the task and move the durable part — the decision, the gate result — into the committed record.
5. **Resume in a different tool, and only do the next step.** If the resume needs the first tool for anything, step 4 was skipped.

Two habits keep this cheap. **One artefact per task, overwritten in place** — it is state, not a log, and a directory of timestamped files is a directory nobody reads. **Keep it to one screen**: if the artefact no longer fits, the task is two tasks.

Mechanism and View

- **Mechanism:** session lifecycle, resume, identity. Adopt it; never rebuild it.
- **View:** a timeline or replay panel. Adopt it; never let it become the only place the state exists.

✅ What to Check

- Run the handoff drill once. Record where it failed — that is the field your artefact is missing.
- If the tool offers a replay view, can you still resume when it is empty?

---

5. Build the Handoff

Artefacts

Artefact	What it is for	Where it lives
session artefact	the handoff contract, seven fields	`.session/<task>.yaml`, gitignored
resume checklist	what to read before continuing	the Playbook
reconciliation check	artefact compared to the working tree	the resume step, or a script
handoff drill record	one real cross-tool resume, with its failure	the evidence pack

The resume checklist, as a shape — four steps, in order:

```
# Resume a task
1. Read the artefact: task, status, decisions, next-step.
2. Reconcile: compare files_modified with `git status`; investigate the delta.
3. Check the gate: what passed, what is assumed, what must re-run.
4. Do the next step — one, not the whole plan.
```

Decisions

Choice	Default	What changes if you choose otherwise
where the state lives	a file in the repository, gitignored	tool memory, which stops at the tool boundary
authority on conflict	git	a resumer that trusts a memoir
what to record	decisions, including the rejected option	a resumption that re-litigates
the drill	on a real task, once	a rehearsal that proves nothing

✅ Checks

- A task started in one tool **resumes in another** from the artefact alone.
- The artefact and `git status` agree, or the delta is explained.
- No secret, transcript or persona data is in the file.

Sequence

1. **Write the artefact** at the first natural stop. *Checkpoint: all seven fields filled, including a rejected option.*
2. **Write the resume checklist.** *Checkpoint: it begins with the artefact and the reconciliation, not with code.*
3. **Run the drill** across two tools. *Checkpoint: the second tool resumes without the first.*
4. **Add the reconciliation** to the resume step. *Checkpoint: a stale artefact is caught before work resumes.*

Done When

- A different tool can resume a real task from the artefact alone.
- The artefact never disagrees with the working tree silently.
- The drill record names what failed the first time.

---

6. 🔑 Chapter Summary

Continuity is a file, not a memory.

1. **Tool-private memory cannot carry continuity** — an interruption deletes it, and reconstruction is a story.
2. **The artefact is a contract** with seven fields, and `decisions` is the one that earns the file — `what`, `why`, and the option that was rejected.
3. **Gitignored, not a cache** — the file carries what the repository cannot, and the fields the repository can supply are reconciled before resume.
4. **Cross-tool handoff is the test**; a mid-task stop on real work is the cheapest way to run it.
5. **Crash recovery is a mechanism, continuity is a rule** — adopt the first, own the second.

Chapter 8: The Agent's Procedure

Orchestrating a Multi-Step Change

---

1. Why Sequences Break

The change touched five files: a schema migration, a deploy, a verification run, a cut-over and a cleanup. The agent planned all five in its head, ran them in one pass, and failed at the fourth. The error named a missing configuration key. It did not say whether the migration had applied, whether the deploy had taken the traffic, or whether the verification had ever run. There was one gate, at the end, and the end is where the information runs out.

That is the shape of the problem. **A sequence that lives in prose is a sequence the agent re-derives every time**, and re-derivation is where steps get skipped. Nothing in the repository knows which step the run is on, so the failure arrives with no step to blame and the resume starts at the top.

The three structures assume the work is repeatable. The **workflow structure** is the one this chapter makes concrete: it is where the prompt and the engineering are checked, in order, on the way to a change. A single edit needs no help. A change that touches five files and can fail in the middle needs a **procedure**: the ordered steps, each with a gate, written as a contract rather than a plan. **A play** is the file that states one procedure — the artefact, in the repository, reviewed like code.

Chapter 7 made one session's work resumable across tools. This chapter makes a multi-step procedure resumable inside that work: a procedure records which step it is on, so a resume starts at that step instead of at the top.

A procedure also carries **approval points** — steps that cannot proceed without a decision, by design. Those are not escalations. An approval point is planned and sits where the procedure put it; an escalation is a condition that stops a run wherever it appears. This chapter defines the first; the next defines the second.

> 💰 **Cost Check**: a procedure costs a review and a maintenance pass; the absence of one costs the re-derivation on every run, plus the failures that arrive at the end with no step to blame. Write procedures for work that repeats and fails in the middle — the two conditions where the arithmetic favours them — and delete the ones that ran once.

🎯 What You'll Learn

- **The anatomy of a step** — action, gate, approval point, and where a resumed run continues
- **Why a procedure is a contract** — the agent's, in the repository, not the Orchestrator's checklist
- **Play, engine and board** — which of the three belongs in your repository, and which you adopt
- **When not to write one** — the work that costs more with a procedure than without

The Five-File Change

Without a procedure	With one
the agent plans five steps in its head	each step states its action and its gate
a failure at the end names one symptom	a failure names the step that failed
resumption starts from the top	resumption starts from the recorded step
two agents do it two ways	one way, reviewed

```
flowchart LR
  S["step"] --> A["action"]
  A --> G{"gate"}
  G -->|"passes"| N["next step"]
  G -->|"fails"| F["stop — the gate names the fix"]
  G -->|"needs a human"| P["approval point"]
```

What this shows: every step ends at a gate, and a gate has three outcomes — pass, fail, or escalate.

✅ What to Check

- Take your longest repeated change. Can an agent name the step it is on, right now, from the repository?
- Does the procedure have a gate per step, or one gate at the end?

---

2. Procedure Anatomy

Three words do most of the work in this chapter, and each means one thing:

Word	What it means here	What it is not
sequence	ordered work, however it is written — including in someone's head	not a procedure until every step has a gate
procedure	a sequence written down, with a gate on every step	not the tool that runs it
play	the file that states one procedure	not the engine, and not the board

A procedure is a list of steps, and a step has four parts:

Part	What it states	Example
action	the one operation	*apply the migration to staging*
gate	what proves it worked	*the schema validator passes*
approval point	when a human decides	*before production*
resume	where to continue	*step 4 if the gate passed*

Names are invariants, not tasks. A step named *"check the migration"* describes an activity; a step named *"the migration applies cleanly to a copy of production"* describes a state that is either true or false. The second can fail; the first can only be "done".

The Skeleton

```yaml
# plays/<name>.yaml — the play, as the agent's contract
name: <what the procedure delivers>
trigger: <when this play is the right one>
steps:
  - id: 1
    invariant: <the state that must be true after this step>
    action: <the one operation>
    gate: <the check, with its command>
    approval: none | human:<who>
    on_fail: stop | run:<step-id>
stop: <the observation that aborts the run>
rollback: <how to return to the last known-good state>
```

A play is more than its steps. The three fields around them are what separate a play from a to-do list:

Field	What it answers	Why it is not optional
trigger	when this play is the right one	a play applied to the wrong task is a sequence of plausible mistakes
stop	what observation aborts the run	without it, the run improvises at the moment improvisation is most expensive. This adds a stop for *this* play; the estate-wide conditions in the next chapter apply to every step and cannot be removed
rollback	how to return to the last known-good state	the point of gates is that the cheap failures stay cheap

This is the same anatomy the Playbook states in prose — *trigger, steps with gates and approvals, stop condition, rollback*. The YAML is the machine-readable form of one play; the Playbook is where the plays are collected and reviewed.

The Five-File Change, as a Play

The scenario from §1, written as a procedure. Three rehearsal steps that can fail cheaply, one production step behind an approval:

```yaml
# plays/migration-to-production.yaml — the five-file change from §1
name: the new schema is live and the old path is still draining
trigger: a migration that changes the schema under a running service
steps:
  - id: 1
    invariant: the migration applies cleanly to a copy of production
    action: restore last night's snapshot into the rehearsal database, then apply the migration
    gate: scripts/check-schema.sh --rehearsal
    approval: none
    on_fail: stop
  - id: 2
    invariant: the old and the new schema both accept writes
    action: deploy the dual-write build to staging
    gate: scripts/check-dual-write.sh --staging
    approval: none
    on_fail: stop
  - id: 3
    invariant: the rehearsal cut-over completes with no reads left on the old path
    action: run the cut-over against the rehearsal database
    gate: scripts/check-cutover.sh --rehearsal
    approval: none
    on_fail: stop
  - id: 4
    invariant: production serves the new schema with the old path still draining
    action: apply the migration to production and enable dual-write
    gate: scripts/check-production.sh --window=30m
    approval: human:the-orchestrator
    on_fail: stop
stop: the rehearsal database cannot be restored, or the cut-over leaves reads on the old path
rollback: keep dual-write on and re-point the service at the old schema; the migration has a tested down path
```

Read it top to bottom and the argument is visible: the three steps that can be ruined for free come first, the one that cannot waits behind an approval, and every step states the state it leaves behind. The failure in §1 named a configuration key; the same failure here names step 2, its gate, and the command that fixes it.

One run, recorded. The play above, run twice — the second time after a failed cut-over:

```
run 1   step 1  green   schema applies to the rehearsal copy
        step 2  green   dual-write accepts on both schemas
        step 3  red     the cut-over leaves four reads on the old path
        step 4  —       not attempted; the approval was never requested
        verdict: failed at step 3 — fix the leaking reader, resume at step 3

run 2   step 3  green   no reads on the old path
        step 4  human:the-orchestrator approved, window 30m
        step 4  green   production serves the new schema, old path draining
        verdict: passed
```

The second run does not re-read the plan, re-run steps 1 and 2, or ask whether the migration applies. It starts at the first gate that is not green — which is the whole return on writing the play down.

Resumability comes from the `id` and the recorded gate status: a resumed run starts at the first step whose gate is not green. **Idempotency** comes from the action being repeatable — a step that cannot be run twice is a step that cannot be resumed.

✅ What to Check

- Is each step's name a state that can be false?
- Can each action run twice without harm?

---

3. When Not to Write a Procedure

A procedure is a maintenance liability: it must be kept true, and a stale one is worse than none because it is trusted.

Do not write one for:

- **a single edit** — one file, one gate, no sequence to lose;
- **exploratory work** — the destination is unknown, so the steps would be fiction;
- **a one-off migration** — the procedure outlives its purpose and becomes archaeology.

Do write one when: the change repeats, it touches more than a handful of files, a gate can fail mid-way, or a human must approve something before it continues.

✅ What to Check

- Take your procedures. Which ran once and never again? Delete them.
- Which repeated change still has none? That is the next procedure.

---

4. The Play, the Engine and the Board

- **The play** is the file that states the contract: the steps, the gates, the approvals. It lives in the repository, reviewed like code.
- **The engine** is a mechanism: it runs the steps and records the outcome. Adopt it; never rebuild it.
- **The board** is a view: what is in flight, what is blocked. Adopt it; never let it become the only place state exists.

A play that lives in the engine's configuration is a play that cannot be reviewed, diffed or tested — and disappears when the engine changes. This is the same rule as the Map: **the rule lives where it is reviewed.**

Gates that cannot pass are reported blocked. A gate that fails for a reason outside the run's control is not a failed step; it is a step that cannot proceed. Saying "blocked" with the exact command and the reason is the honest outcome. The state is this chapter's; the report and the decision are the next chapter's.

A Procedure Is Not a Pipeline

This is the question every engineer asks at this point, because a play looks like a CI workflow and is not one.

	A CI workflow — Actions, GitLab, Jenkins	**A procedure**
who runs it	a runner, on a trigger, with no one watching	an agent or a person, watched or not
what its steps are	commands	states that must become true, each with a check
approval	not in the pipeline's vocabulary, or bolted on as a manual job	a first-class field, at the step where it belongs
memory	none — each run starts clean	the run record, so a resume starts where the failure was
rollback	a second workflow nobody wrote	a field, tested before it is needed
when it is wrong	the pipeline is red	the step says which invariant is false

The two compose rather than compete. **A procedure's gate is often a CI workflow**: step 2's gate may be *the integration workflow on the branch is green*. The workflow is a mechanism inside a step; the procedure is the contract over the whole change, including the part no runner can do — deciding, approving, rolling back.

The paths make it obvious. The play lives in `plays/`, the pipeline in `.github/workflows/` or `.gitlab-ci.yml` — if a play ever needs to sit in the pipeline's own directory to work, the engine has become the contract.

What would go wrong if you used the pipeline as the procedure? The run would begin at the first job and end at the last, with no line for the human decision that matters, no record of which step failed first, and no rollback that anyone rehearsed.

✅ What to Check

- Is the play in the repository, or in the tool?
- Can a blocked step be distinguished from a failed one?

---

5. Build the Play

Artefacts

Artefact	What it is for	Where it lives
play file	the procedure, as the agent's contract	`plays/<name>.yaml` — never a CI workflow directory
gate annotations	the check per step, with its command	inside the play
resume rule	where a resumed run continues	the play, or the Playbook
run record	one procedure run, with its gates	the execution log

Decisions

Choice	Default	What changes if you choose otherwise
where the play lives	the repository	tool configuration: unreviewable, unportable
step naming	invariants	activity names that cannot fail
approval points	before anything irreversible	a human asked at the end, when the choice is gone
blocked vs failed	separate states	one word for two situations, and a decision nobody makes

✅ Checks

- Every step has a gate with a command; a step with no gate is a narration.
- A resumed run starts at the first step whose gate is not green.
- A blocked step names the command and the reason, and is never reported as passing.

Sequence

1. **Write the play** for one repeated change. *Checkpoint: each step names an invariant.*
2. **Add a gate per step.** *Checkpoint: every gate can fail and name its fix.*
3. **Mark the approval points.** *Checkpoint: no irreversible action runs before one.*
4. **Run it once and record the gates.** *Checkpoint: the run record matches the play's steps.*

Done When

- One repeated change has a play whose steps are invariants.
- A resumed run continues at the right step without re-reading the whole plan.
- A blocked step is reported as blocked, with the command that would unblock it.

---

6. 🔑 Chapter Summary

A procedure is how a sequence becomes repeatable — and how a failure becomes legible.

1. **Sequences break** when they live in prose: one gate at the end names the symptom, not the step.
2. **A step is action, gate, approval, resume** — and its name is a state that can be false.
3. **Not every change needs a play** — single edits, exploration and one-offs cost more than they save.
4. **The play is the file that states the contract**; the engine is a mechanism and the board a view, and only the play belongs in your repository.
5. **Blocked is a state**, not a failure — and never a pass.

Chapter 9: Escalation Policy

Where an Agent's Authority Ends

---

1. Permissions Are Not Judgement

The task was to clear a cache before a deploy. The command was permitted, the plan approved a cache clear, and the variable that selected the cluster resolved to production. The run cleared the production cache at peak traffic. Every part of that was allowed: the sandbox permitted the command, the approval covered *a* cache clear, and nothing compared the environment the plan named with the one the run resolved.

That is the failure a permission system cannot prevent: **the change that is permitted, plausible and wrong.** A sandbox can stop a file from being read or a call from leaving. It cannot stop the migration that runs on the wrong environment, the delete that is technically allowed, or the refactor that quietly removes a guarantee.

The three structures hold only if the agent stops where the structure ends.

The permission system is necessary and insufficient. **The boundary that matters is the one the agent recognises before it acts.** An **escalation boundary** is that line: the explicit, checkable division between what an agent may decide and what it must stop for. It is written as conditions the agent evaluates before

acting, and reported in a form a person can act on.

Nothing in this chapter's rules is specific to AI. A scheduled pipeline with a deploy token meets the same six conditions; what is new is that an agent reaches them faster, and more plausibly.

Operations has a name for this: an **escalation policy** — the conditions under which a decision goes to a person, and who that person is. The agent frameworks call the same thing *guardrails and human review*. Both name the thing this chapter builds, and neither needs a new word for it.

Chapter 8's `stop:` field aborts one play. These conditions abort **any act** — inside a play, or with no play at all — which is why they live in the Law and not in the play. A play may add a stop condition of its own; it may not remove one of these. The approval points in a play are the stops a procedure planned; this chapter is about the stop no procedure can plan.

> 💰 **Cost Check**: an unnecessary escalation costs a question and a wait. A missed one costs the irreversible change — the deleted branch, the wrong-environment migration, the credential that reached a log — or the governance edit nobody approved. The catalogue is short for a reason: five conditions that get evaluated beat twenty that get skimmed, and the drill is what makes the five real.

🎯 What You'll Learn

- **What a permission system cannot decide** — the permitted, plausible, wrong change
- **The six conditions** that always stop a run, and why six beats twenty
- **Blocked as a verdict** — the condition, the exact command, and the decision needed
- **How the decision returns** — the log entry that stops the next session re-litigating it

What a Permission Cannot Decide

A permission system can	It cannot
deny a path	know that this delete is the irreversible one
block a host	know that this push reaches production
sandbox a process	know that the source of truth is ambiguous
log a call	decide that the decision is not its to make

That gap is not a bug to be closed with better permissions. It is the reason escalation is a **rule in the Law** with conditions an agent can evaluate, and a report format a human can act on.

✅ What to Check

- Name one action your sandbox permits that would be wrong today. That is an escalation condition.
- Name one condition that lives only in a prompt. It will not survive a context reset.

---

2. The Conditions Catalogue

Chapter 8's approval point is the stop the procedure planned. This is the stop it could not.

Three words do most of the work in this chapter, and each means one thing:

Word	What it means here	What it is not
condition	one of the six triggers, checked before acting	not a general worry about the change
escalation	handing the decision back to a person	not a failure of the run
blocked	the state and the report that follow	not a pass, and not a fail

Condition	What it looks like	The stop
Irreversible	a delete, a force-push, a tag, a migration with no down path	stop before it, not after
Security or production	a credential, a trust edge, a permission widening, a change that reaches production	stop; name the exact change
Ambiguous authority	contradictory requirements, two sources of truth, an architectural choice with several valid options	stop; present the options, do not pick
A governance change	the change would alter the Law or the Map	stop; propose the change
Out of scope	the work grew past what was approved	stop at the boundary; re-plan
A gate that cannot be resolved	the check cannot run here, or it failed and N attempts did not clear it	report blocked, never passing

The catalogue is short on purpose. A list of twenty conditions is a list nobody evaluates; six can be checked before every irreversible act.

```
flowchart TD
  A["about to act"] --> Q{"irreversible?<br/>security or production?<br/>ambiguity? governance?<br/>out of scope? gate unresolved?"}
  Q -->|"no"| P["proceed"]
  Q -->|"yes"| S["STOP — report blocked<br/>with the exact command"]
  S --> L["the decision reaches the log"]
```

What this shows: the catalogue is evaluated before the act, and the stop produces a record, not a silence.

✅ What to Check

- Is each condition written where the agent reads rules, or only in a briefing?
- For the last escalation you saw, which condition was it? If none, the catalogue is incomplete.

---

3. The Stop-and-Report Contract

A stop without a report is an outage the human discovers later. A report is three things:

1. **the condition** — which of the six, by name;
2. **the exact command or change** — not a summary, the thing that was about to run;
3. **the decision needed** — a question with a small number of answers.

Three states, and they are not degrees of each other:

State	What happened	What it must say
pass	the gate ran and the step may continue	the command, and its output
fail	the gate ran and the step may not continue	the step, the error, the fix
blocked	the gate could not run, or the decision is not the agent's	the condition, the exact command, the decision needed

A fail is a verdict about the work; a blocked is a verdict about the run's authority to continue. Collapsing them into one word hides which of the two a human is being asked to fix.

Blocked is a verdict, not a mood. It must never be reported as "done", "probably fine", or a check that "passed with warnings". The estate's rule holds here in its plainest form: *reporting done over an unresolved gate is the failure this chapter exists to prevent*.

The Blocked Record

```
# the blocked report — three fields, no prose wall
condition: irreversible | security-production | ambiguous-authority | governance-change | out-of-scope | gate-unresolved
exact: <the command, or the diff, verbatim>
decision_needed: <a question with the options you would accept>
```

The Cache Clear, Blocked

The scenario from §1, stopped and reported:

```
condition: irreversible
exact: scripts/flush-cache.sh --cluster prod-cache-01 --region ap-southeast-2
decision_needed: >
  The approved plan names staging; this command resolves to production during peak traffic.
  Either (a) run it against staging as approved, or (b) approve the production clear for this window.
```

Three fields, one command, one question with two answers. A person can act on that without asking anything back — and a later session can read what was decided and why. A failure report could not do either: it would say what broke, after the cache was already empty.

✅ What to Check

- Could a human act on your last blocked report without asking a follow-up question?
- Does anything in your workflow treat a blocked gate as green?

---

4. The Record, and How a Decision Returns

The escalation is recorded where the work is recorded: the execution log. It carries the condition, the exact command, the decision and who made it. That record is what makes an audit possible later — not "the agent asked someone", but **what it asked, what it was told, and what it then did**.

A decision that does not return to the log is a decision that will be re-litigated. The next session reads the log; if the escalation is absent, it asks again — or, worse, decides for itself.

The Decision, and the Log

The cache clear from §1, carried through to the record:

```
blocked    condition: irreversible
           exact: scripts/flush-cache.sh --cluster prod-cache-01
           decision_needed: run against staging as approved, or approve production for this window
asked      the Orchestrator, 14:20
answered   run it against staging, then re-plan the production window for Tuesday 02:00
logged     the condition, the command, the decision, the actor and the time
next run   reads the log, finds the decision, does not ask again
```

The last line is the test. **A decision that is not in the log is a decision the next session will make again** — and the second time it decides, it will not have the context that produced the first answer.

Mechanism and view. Permissions and sandboxes are the mechanism; an approval dialog is the view. Adopt both. The rule — the catalogue and the report — is yours, and it must survive either being replaced.

✅ What to Check

- Find one past escalation. Is the decision in the log, with the actor?
- If the approval dialog were removed tomorrow, would the rule still hold?

---

5. Build the Policy

Artefacts

Artefact	What it is for	Where it lives
escalation policy	the six conditions, as a rule	the Law
blocked record	condition · exact · decision needed	the log entry
escalation drill	a regression attempt that proves the stop holds	the evidence pack
decision loop	how the answer returns to the log	the Playbook

Decisions

Choice	Default	What changes if you choose otherwise
where conditions live	the Law, reviewed	a prompt, which a reset erases
blocked vs failed vs done	three states	one word that hides the decision
the report shape	three fields, exact command	prose that omits the thing that was about to run
who may widen the catalogue	the Orchestrator	an agent that decides its own boundaries

✅ Checks

- Each condition is **attempted in a drill** and the run stops.
- A blocked record names the exact command and the decision needed.
- The log carries the decision, the decider and the timestamp.

Sequence

1. **Write the catalogue** into the Law. *Checkpoint: six conditions, each recognisable.*
2. **Write the blocked record** shape. *Checkpoint: three fields, one exact command.*
3. **Run the drill** — one attempt per condition. *Checkpoint: every attempt stops.*
4. **Close the loop**: the decision returns to the log. *Checkpoint: a later session can read what was decided.*

Done When

- An agent stops on each of the six conditions, by rule, not by luck.
- A blocked report can be acted on without a follow-up question.
- Every past escalation has its decision in the log.

---

6. 🔑 Chapter Summary

A sandbox denies actions; a rule stops judgement.

1. **Permissions cannot decide** — they can block a path, not recognise the wrong change.
2. **Six conditions always stop**: irreversible · security or production · ambiguous authority · a governance change · out of scope · a gate that cannot be resolved.
3. **Blocked is a verdict** — reported with the condition, the exact command, and the decision needed.
4. **The decision returns to the log**, or the next session re-litigates it.
5. **The mechanism is adoptable; the rule is yours** — and it must survive the mechanism being replaced.

Chapter 10: Execution Transparency

The Log Is the Evidence

---

1. Execution Transparency

A client asks how a change was reviewed six months ago. The thread is gone, the CI retention window closed in March, and the answer is a memory of a conversation.

Execution transparency is the discipline of making the work auditable without the tool that produced it. The **execution log** is the record that makes it possible. It is append-only, it lives in the repository, and it is written as the work happens: what was read, what was decided, what was changed, and which gates ran.

Prior chapters built the controls on the three structures — routing, continuity, procedures, escalation, economy. Each of them leaves a claim about work already done, made by the party that did the work. This chapter builds the record that proves those controls ran. Without it, every claim a control leaves is a memory of a conversation.

> **The rule this chapter defends:** a record that can be edited, or that lives only in a vendor's retention window, is not evidence.

> 💰 **Cost Check**: an absent log costs a client audit assembled from memory, and a review that reconstructs authorship from diffs. A nine-field entry costs a minute per task. The expensive failure is not the missing entry today — it is the release, six months from now, whose evidence nobody can reconstruct.

🎯 What You'll Learn

- **Why the work needs its own log** — the four facts, the three properties, and where they must live
- **The nine-field entry** — what each field proves, and why the entry is append-only
- **The check that keeps the record honest** — the log's gate results must match the gates' own records
- **What provenance does to review** — from reconstructing authorship to reading intent and evidence

✅ What to Check

- Take the last change an agent made. Can you say what it read, from the repository alone?
- Which record answers "did the gate pass?" If the answer is a memory or a dashboard, the log is incomplete.

---

2. The Records Around It

Four mechanisms exist in the same estate, each answering a real question. None of them is the execution log, and each one relates to it:

Mechanism	What it answers	Its limit
the platform's logs	what the software did: requests, errors, latency	the software's story, not the work's — a decision is not an event in it
the runtime trace	what calls were made, in what order, with what timing	it proves the execution happened; it cannot show what was decided
the gate records	did this check pass, fail or block	the authority on that result; it shows the result, not the reason
the evidence pack	how do you know, for one release	it is assembled from the log and the gates, not a separate truth

Read the middle column and the shape is visible: the platform logs and the trace record **execution**; the execution log records **evidence**; the gate records are the authority the evidence cites; the evidence pack is the evidence assembled for a reader.

When something goes wrong, two questions get asked. The first — *did the agent obey the Law?* — is about work, and the execution log answers it, because the answer must outlive the tool. The second — *what calls were actually made?* — is about execution, and the trace answers it, because that record is enormous and short-lived.

```
flowchart LR
  W["work: reads · decisions · gates"] --> L["execution log<br/>survives the tool"]
  E["execution: calls · timing"] --> T["runtime trace<br/>short-lived, enormous"]
  L --> A["the audit<br/>= log + gate records"]
  T -.->|"not the audit"| A
```

What this shows: the log and the trace answer different questions; only the log survives long enough to audit.

✅ What to Check

- Does your estate record what the software did, and what the work did? If only the first, the log is missing.
- Where does the answer to "did this gate pass?" live — a system with a retention window, or the repository?

---

3. The Log Entry

```yaml
# one entry, append-only
timestamp:  <when>
task:       <what was being done>
files_read: [<paths>]
rules:      [<law ids applied>]
```

```
decisions:  [<what, why, rejected>]
gates:      [{ id: <gate>, result: pass|fail|blocked }]
outcome:    <what happened>
next:       <the next action>
actor:      <human | agent identity>
```

Field	What it is for
timestamp	order; the only field a machine supplies
task	the unit of work, so entries can be grouped
files-read	the routing check — what the routing table said to load, and what the run actually opened
rules	which Law ids were applied — the difference between obeying and claiming
decisions	including what was rejected; the field a reviewer needs
gates	pass, fail or blocked, with an id
outcome	what actually happened, not what was intended
next	the resumption point
actor	the identity that acted — an agent's, never a borrowed human's

The economy chapter adds three cost columns to this entry — tokens, retries, tier. They extend the nine; they do not replace them.

Append-only. An entry is never edited. A correction is a new entry that says so. The moment a log can be edited, it stops being evidence and becomes a narrative.

One Record, Four Scopes

Every control in this part writes into this record. The routing table's `files-read` proves what was loaded; the handoff artefact's `decisions` are the entry's decisions; a procedure's gates appear as gate results; a blocked report is one entry with a condition and a decision; the economy's spend columns extend the entry. They are not separate records — they are the same append-only log, seen at four scopes:

Scope	The record	What it answers
one session	the handoff artefact	where was I, and what did I decide
one task	one log entry	what was read, decided, changed, gated
one gate	the gate's own record	did this check pass
one release	the evidence pack	how do you know

Same fields, growing scope, one append-only record.

The outcome tells the truth. If a gate failed and the change shipped anyway, the outcome says so. A log that flatters the run is the one that cannot be trusted later.

✅ What to Check

- Does every entry name the rules applied? Without that field, compliance is a claim.
- Can you find an entry whose outcome contradicts its gates? That is the entry worth auditing.

---

4. The Log and the Gates Must Agree

> **A log that disagrees with the gates is worse than no log.**

If the log says a gate passed and the gate's own record says it failed, the log is not merely wrong — it is a false alibi. The gate is the authority on its own result; the log is the authority on what was attempted.

The check is mechanical: for each gate result in the log, the gate's own record must confirm it. A mismatch is a **finding**, and it means one of the two records is lying — usually the log, because it was written by the party with an interest in the outcome.

The Cross-Check

```
for each log entry:
  for each gate result in the entry:
    assert the gate's own record has the same result at the same commit
  a mismatch is a finding, not a formatting error
```

✅ What to Check

- Is anything cross-checking the log against the gate records? If not, the log's gate field is a claim.
- Take one entry and verify two gate results by hand.

---

5. Provenance, and What Review Becomes

Provenance is the actor field, done seriously: the identity that wrote each change, the AI model or version where relevant, and the review that approved it. For agent-authored work, provenance is what allows a reviewer to focus on **intent and evidence** rather than tracking who typed what.

A review that must reconstruct authorship is a review that reads diffs without context. A review that has provenance, the decisions and the gate results focuses on the one question that matters: *is this the right change?*

What a Client Audit Looks Like

The log plus the gates is the evidence pack: what was attempted, what was decided, what was verified, and who signed. That is a defensible answer, and it is assembled from the repository rather than from memory.

One entry, as the answer to *"how do you know?"*:

```
timestamp   2026-11-06 14:20
task        change the retry policy on the checkout worker
files-read  contracts/queue.md · runbooks/retry-policy.md · services/checkout/worker.tf
rules       law:no-secrets · law:test-accompaniment
decisions   cap retries at 3 (rejected: 5, because the queue's visibility timeout is 30s)
gates       schema-validator pass · integration pass · policy-check pass
outcome     merged; deploy gated to Tuesday's window
next        await the Tuesday window
actor       agent:checkout-worker · approved by the-orchestrator
```

Six months later, that is the difference between a review and an archaeology project: the constraint is named, the rejected option is recorded, the gates are attributable, and the actor is a person or an agent — never a borrowed identity. That is a defensible answer, and it is assembled from the repository rather than from memory.

✅ What to Check

- Can you name the actor for every change in the last release?
- Does a reviewer get the decisions and the gates alongside the diff?

---

6. Build the Log

Artefacts

Artefact	What it is for	Where it lives
execution log	append-only entries, nine fields	`docs/log/<period>.md` or a machine log
entry rules	the nine fields, and the append-only rule	the Playbook
cross-check	log gates vs gate records	the nightly audit
evidence pack	the log plus the gates for a release	the release pack (Part Four)

Decisions

Choice	Default	What changes if you choose otherwise
log format	one entry per task, nine fields	unstructured notes that cannot be audited
editing	append-only; corrections are new entries	a narrative that flatters the run
authority on a gate	the gate's own record	the log's memory of it
provenance	an actor per entry, agent identities included	a diff with no attributable author

✅ Checks

- Every entry has **nine fields**, including rules and gates.
- The log's gate results **match the gate records**; a mismatch is a finding.
- No entry is ever edited; corrections are new entries.

Sequence

1. **Write the entry rules** (nine fields, append-only). *Checkpoint: one entry can be written from a real task.*
2. **Emit an entry per task**. *Checkpoint: the actor and the rules fields are filled.*
3. **Add the cross-check** against gate records. *Checkpoint: a planted mismatch is reported.*
4. **Assemble one evidence pack** from the log and the gates. *Checkpoint: it answers "how do you know?" without a meeting.*

Done When

- The log answers *what was read, decided, changed and gated* for any task.
- A mismatch between the log and a gate is caught, not discovered later.
- A release's evidence pack can be assembled from the repository.

---

7. 🔑 Chapter Summary

Transparency is a record that can be audited without the tool that produced it.

1. **Two questions, two records** — the execution log answers obedience; the runtime trace answers calls.
2. **Nine fields**, append-only, with the actor and the rules applied.
3. **A log that disagrees with the gates is worse than no log**, and the disagreement is checkable.
4. **Provenance shifts review** from keystrokes to intent and evidence.
5. **The evidence pack is the log plus the gates** — the client audit, assembled from the repository.

Chapter 11: Build, Buy, or Assemble

The Decision Record and Its Exit Plan

---

1. The Decision Made on an Invoice

A team adopts a new observability platform. The reason is honest and it is not architectural: an agent can write the integration adapter in an afternoon, and the free tier covers the first month. Six months later the adapter has grown four undocumented branches, the platform's query language has appeared in six services, and the data migration costs three engineer-months. Nothing failed. **The decision was made on an invoice, not on an architecture.**

That is the shift this chapter is about. Chapters 2 to 10 built the internal machinery of the self-improving repository — the command interface, the vocabulary, the routing, the continuity, the procedures, the escalation policy, the execution log. This chapter looks outward, at the repository's **boundaries**: the database, the platform, the product, the AI model. Each is a dependency someone chose, and AI has changed the economics of choosing. Writing the glue is cheap now, which makes **assemble** the default posture of the Software Orchestrator — compose the estate, enforce policy at the seams, and let providers carry the commodities. A **seam** is the integration point where two systems meet, and — more usefully — the place you design so it can be cut. Policy is enforced at seams because that is where your behaviour meets someone else's.

The decision lives in the repository. The dependency is outside; the record of choosing it is not. Every artefact in this part is a file you commit — the decision record with its matrix and exit plan, the integration charter, the contracts, the drill results — so the boundary is not a place the repository stops. It is the edge the repository governs, from the inside, like every other control in this book.

This is where the workbench turns outward. Every chapter so far in this part was an instrument you build and hold: the command interface, the vocabulary, the routing, the handoff, the procedure, the boundary, the budget, the log. This chapter and the ones that follow are about what those instruments stand on — an upstream product, a provider's commodity, a contract with a service you do not own. The workbench is what you build; the estate is what you connect to, and the Orchestrator is accountable for both.

Cheap glue has a price of its own. It is now easy to add a dependency, write the adapter, and never record why — so an AI-accelerated team can accumulate architectural sprawl faster than it can review it. The controls are the ones this book has already built, applied at a different boundary: the command surface, the view it renders and the procedures live in the glue; the capacity that constrains it is the human authority the escalation policy protects; and the record of the choice belongs beside the execution log.

> **The rule this chapter defends:** a build-or-buy choice without an exit plan is not a decision; it is a commitment with an unknown price. **The decision record is the workflow structure applied to an architectural choice.**

> 💰 **Cost Check**: the invoice is the smallest line. A provider that costs £200 a month and takes a week to exit costs less than one that costs £50 and cannot be left. Write the total — integration, on-call, upgrade, exit — or the comparison is between two numbers that do not mean the same thing.

🎯 What You'll Learn

- **The five axes** — control, cost, time, sovereignty, reversibility — and why the decision is which conflict you can live with
- **Why assemble is the default** — providers for commodity, agents for the glue, a person for the differentiator
- **The total cost of a dependency** — integration, on-call, upgrade and exit, not the invoice
- **The exit plan as a deliverable** — the seam, the data, the drills and a trigger, written with the decision

The Five Axes

Axis	The question	What a bad answer costs
control	can you change the behaviour when it matters?	a workaround culture
cost	total, beyond the invoice	integration and on-call, unbudgeted
time	what does it cost to reach the first working version?	a quarter spent rebuilding a commodity
sovereignty	where does the data live, and who can read it?	a compliance answer you cannot give
reversibility	what does leaving cost?	a price the vendor now sets

The axes conflict, and the decision is which conflict you can live with. Writing the answer down is what makes it reviewable; a decision that lives in memory is re-made, differently, by the next person.

```
flowchart LR
  N["need"] --> Q{"commodity,<br/>or differentiator?"}
  Q -->|"commodity"| B["buy — a provider"]
```

```
Q -->|"differentiator"| A["assemble — own the glue"]
A --> D["agents for the glue<br/>the human for the judgement"]
B --> X["exit plan required"]
A --> X
```

What this shows: the first question is not build or buy — it is whether this capability is the thing you compete on.

✅ What to Check

- Take your five biggest dependencies. For each, can you name the axis that decided it?
- Which of them has no exit plan? That is the one whose price you do not know.

---

2. Assemble as the Default

The old choice was binary: build or buy. The orchestrated choice has a middle: **assemble**.

Layer	Who builds it	Why
commodity — transport, storage, mail delivery	a provider	nobody differentiates on it
the glue — integration, policy, approval, the seam	you, with agents	it is where your behaviour lives
the differentiator — the judgement, the method	you, by hand	it is the thing you sell

Agents change the calculus at the middle layer. Glue that used to take a week now takes an afternoon, so the constraint moves: it is no longer typing, it is **review capacity**. That single shift is why "assemble" became the default — the part that used to be expensive to hand-write is now cheap, and the part that was always scarce (attention) has not moved.

This posture is the Orchestrator's job. The provider is accountable for the commodity; the Orchestrator is accountable for the estate's composition — which pieces touch, what is allowed across each seam, and what happens when one of them is replaced. That work lives in the glue, and the glue is built from the instruments the book has already given you: the **command surface** is how the seam is operated, and the **procedures** are how it is changed safely.

Total Cost, Beyond the Invoice

A dependency's cost is not the subscription. It is:

- **integration** — the seam, its tests, its drift;
- **on-call** — what breaks at 2am, and who knows it;
- **upgrade burden** — the rehearsals and the patch drift;
- **exit cost** — the migration you would face, and how long it would take.

An invoice comparison that ignores the last four is not a comparison.

✅ What to Check

- For each dependency, name the layer it occupies: commodity, glue, or differentiator.
- Has anything at the *differentiator* layer been bought? That is a strategy problem, not a budget one.

---

3. The Exit Plan

An exit plan is a deliverable, written when the decision is made, and it has four parts:

1. **the seam** — the interface that would survive the change;
2. **the data** — where the data lives, and how it moves;
3. **the drills** — what a vendor change would break, and who would notice;
4. **the trigger** — the condition under which you would actually leave (price, terms, API change).

The drill makes it real. Pick one dependency and ask the vendor-change question: *if the price doubled tomorrow, what would we do?* If the answer is *"we would have to do something"*, the exit plan is a wish. If it is a named path with a cost and a duration, it is a plan.

Vendor-change drills are cheap and rare. One per dependency per year is enough to keep the plan honest, and cheap enough that skipping it is a choice, not an accident.

✅ What to Check

- Does each major dependency have an exit plan with a trigger?
- Has any drill been run? Which dependency would survive a price doubling?

---

4. How Agents Move the Calculus

They make the glue cheap. Integration code, adapters and policy layers are now hours, not weeks — which makes assembling more attractive, and buying *more* tempting, because the provider's own integration is also cheap to replace.

They make review the constraint. More glue means more diffs to review, and the review is the scarce resource. So the decision shifts: *can we review this seam steadily?* — not *can we build it?* This is the same scarcity the **escalation policy** protects: the human authority to approve a change. An agent writes the adapter in an afternoon; a person still has to verify it against the Law, and that capacity does not scale with the agent's output.

They do not change sovereignty. Where the data lives is a legal and architectural fact, not an effort question. An agent that can write an adapter in an hour still cannot make a provider's data residency acceptable.

Two records, one question apart. The **decision record** is the strategic log: why this capability, on which axes, with what exit. The **execution log** is the tactical log: what the work read, decided and gated while using it. One without the other leaves a question unanswerable — a record that says why but never what, or a log that says what but never why.

✅ What to Check

- For each recent build-or-buy, was the deciding axis effort or review capacity?
- Is any decision justified by "the agent can build it"? That is a cost argument, not a control one.

---

5. Build the Decision Record

Artefacts

Artefact	What it is for	Where it lives
decision matrix	the five axes, scored per option	`docs/decisions/<capability>.yaml`
decision record	the choice, the axes that decided it, the date, the trigger to revisit	`docs/decisions/<capability>.md`
exit plan	seam · data · drills · trigger	inside the record
drill record	one vendor-change drill, with its answer	the evidence pack

The decision matrix, as a shape — five axes, one row per option:

```yaml
# docs/decisions/<capability>.yaml
capability: <what is being decided>
options:
  - name: <build | buy | assemble: provider X>
    control: <low | medium | high>
    cost: <total, not the invoice>
    time: <to first working version>
    sovereignty: <where the data lives>
    reversibility: <what leaving costs>
decision: <the chosen option>
decided_by: <who>   date: <when>
revisit_when: <the condition that reopens this decision>
```

Decisions

Choice	Default	What changes if you choose otherwise
the default path	assemble	a rebuild of commodity, or a buy of the differentiator
cost basis	total cost (integration · on-call · upgrade · exit)	an invoice comparison that hides three costs
exit plan	written with the decision	a price the vendor sets later
drill cadence	one per dependency per year	a plan that has never been tested

✅ Checks

- Every major dependency has a **record with five axes**, not a paragraph.
- Every record has an **exit plan with a trigger**.
- One drill has been run, and its answer is recorded.

Sequence

1. **List the dependencies** and place each at commodity, glue or differentiator. *Checkpoint: no differentiator is bought.*
2. **Write the matrix** for the next decision. *Checkpoint: five axes, total cost.*
3. **Write the exit plan** into the record. *Checkpoint: a trigger and a named path.*
4. **Run one drill** on the least reversible dependency. *Checkpoint: the answer is a path, not a shrug.*

Done When

- The next dependency decision is a record, not a conversation.
- Every major dependency has an exit plan and a trigger.
- One drill has been run on the hardest case.

---

6. 🔑 Chapter Summary

Build, buy, or assemble is a record with a price on the way out.

1. **Five axes** — control, cost, time, sovereignty, reversibility — and the decision is which conflict you can live with.
2. **Assemble is the default**: providers for commodity, agents for the glue, the human for the differentiator.
3. **Total cost includes integration, on-call, upgrade and exit** — not the invoice.
4. **The exit plan is a deliverable**, with a seam, a data path, drills and a trigger.
5. **Agents make glue cheap and review scarce**; they do not change where the data lives.

Chapter 12: Commodity From a Provider

Control Where the Value Is

---

1. What to Buy, and What Never to Buy

The three structures spend money on providers for what they do well, and keep control of what they compete on. The line between those is not obvious, and it is almost never written down — so it drifts toward the provider, one integration at a time.

The line, stated plainly:

Buy the commodity	Keep the control
delivery, transport, storage	routing, policy, approval
scaling and availability of the primitive	what is allowed, and who decides
the protocol's mechanics	the audit trail of who did what

A provider that delivers a message should not decide whether the message may be sent. A provider that stores a record should not decide who may read it. The moment a provider's console becomes the place a policy is set, the control has moved.

> 💰 **Cost Check**: the provider's bill is visible and small; the leak is not on it. Policy set in a console costs a rule nobody can test. An approval in a vendor UI costs an audit you cannot assemble. Untrusted content handed to a tool costs an incident with a plausible explanation. The seam contract is the cheapest artefact in this chapter and the one that stops all three.

🎯 What You'll Learn

- Where the provider's responsibility ends and yours begins
- The seam contract that makes the boundary checkable
- The failure modes that appear at the seam, not the provider

Where the Control Leaks

Leak	What it looks like
policy in the vendor's console	a rule nobody can diff or test
approval in the provider's UI	a decision with no record in your log
a seam with no contract	a behaviour you discover in an incident
credentials in the wrong layer	a provider key where a user's identity belongs

```
flowchart LR
  S["sender"] --> P["your control<br/>policy · approval · audit"]
  P --> V["provider<br/>transport · scaling"]
  V --> D["destination"]
  P -.->|"no control"| V2["provider decides"]
```

What this shows: the provider sits behind your control layer; when the arrow reverses, the provider is deciding.

✅ What to Check

- Name one policy that is set in a provider's console. That is the control that leaked.
- For each provider, can you state what it guarantees, and what you must never assume?

---

2. The Seam Contract

A **seam contract** answers four questions, and it belongs in your repository:

1. **who owns what** — which side is responsible for a failure;
2. **what the provider guarantees** — delivery? ordering? idempotency? at what limits?

3. **what you must never assume** — the guarantees it does not make;
4. **what you will do when it breaks** — retry, drop, escalate, or fail closed.

The seam contract, as a shape:

```
# seams/<provider>.yaml
provider: <name>
owns:
  provider: [<delivery, scaling, ...>]
  you: [<policy, approval, audit, retries, ...>]
guarantees:
  delivery: <at-least-once | exactly-once | unspecified>
  ordering: <guaranteed | not>
  limits: { rate: <n>, quota: <n> }
never_assume: [<what a test would catch if you did>]
on_failure: <retry with backoff | drop and alert | fail closed>
```

The critical field is `never_assume`. Most integrations fail not because the provider broke a promise, but because they assumed a promise the provider never made — ordering, exactly-once delivery, or a quota that was always enough in testing.

✅ What to Check

- Does each provider have a seam contract, or an assumption?
- Which field would have caught your last integration bug?

---

3. Idempotency, Retries and Ordering

Mechanic	The design decision	The default
idempotency	can the same input be processed twice safely?	every handler carries a key
retries	on what errors, with what backoff, up to what count?	retry transient, never blind
ordering	is order guaranteed, or must you sequence?	assume unordered unless stated
duplicates	does the system tolerate them, or dedupe?	dedupe at the boundary

Fail closed where the consequence is a trust or money decision: a payment webhook that cannot be verified is not processed with a warning. **Retry with backoff** where the consequence is delay. The choice is per seam, and it belongs in the contract, not in the handler.

Rate limits and quotas are part of the contract. A provider's limit is not a surprise; it is an unread field. The failure mode — a burst that hits a ceiling and silently drops — is a capacity assumption, not an outage.

✅ What to Check

- Can each handler process the same message twice without a second effect?
- Does the contract name the rate limit and what happens at it?

---

4. Human Gates Inside the Pipeline

Some flows must pause for a human: a send, a payment, a deletion. The gate belongs **inside** the pipeline, in your control layer — not in the provider's UI, where it leaves no record in your log.

The rule: an approval is a record with an actor, a timestamp and a decision. A console click is a click; a logged decision is evidence.

The audit trail is part of the product, not a log bolted on afterwards. The reason is commercial as much as technical: a client who asks *"who approved this send?"* is buying the trail as much as the delivery.

✅ What to Check

- Where does the last human approval in your flow happen — your log, or the provider's UI?
- If a client asked who approved a message, could you answer with a record?

---

5. The New Risk Class: Untrusted Content and Agents

Wherever untrusted content — a message body, a web page, a ticket — reaches an agent, treat it as **input to validate**, in the same class as SQL injection or a malformed payload.

The problem	The mitigation
content that instructs the agent	strip or quarantine instruction-shaped content
a provider payload that carries a directive	parse into a schema; never pass raw text to a tool
an agent with a credential it can be persuaded to use	least privilege, and a hook that refuses the action

The rule is not “trust the model less”. It is that **an agent reading untrusted content is a parser reading untrusted input**, and parsers validate before they act.

✅ What to Check

- Which of your agents read content a stranger wrote?
- Is there a schema between that content and any tool the agent can call?

---

6. Build the Workbench

Artefacts

Artefact	What it is for	Where it lives
seam contract	owns · guarantees · never-assume · on-failure	`seams/<provider>.yaml`
worker	the handler, idempotent, with an approval gate	your control layer
round-trip test	send → receive → verify, including a replay	the test suite
audit record	approvals and outcomes, in your log	the execution log

Decisions

Choice	Default	What changes if you choose otherwise
where policy lives	your control layer	the provider’s console, where it cannot be tested
delivery assumption	at-least-once, dedupe at the boundary	a duplicate side effect
on failure	fail closed for trust decisions, retry for delay	a warning where a stop was needed
approvals	a logged record	a click that leaves no evidence

✅ Checks

- The seam contract names `never_assume` and an `on_failure` policy.
- A **replay test** proves a duplicate does not double-process.
- Every approval is in the log, with an actor and a timestamp.

Sequence

1. **Write the seam contract** for one provider. *Checkpoint: `never_assume` is filled.*
2. **Make the handler idempotent** and add the replay test. *Checkpoint: the replay does not double-process.*
3. **Put the approval in your layer**, logged. *Checkpoint: the decision is in the log, not the console.*
4. **Add the untrusted-content schema** between the payload and any tool. *Checkpoint: a directive-shaped payload is refused.*

🎯 Done When

- Policy and approval live in your layer, testable and logged.
- A duplicate message does not double-process.
- Untrusted content reaches no tool without a schema in between.

---

7. 🔑 Chapter Summary

Buy the commodity; keep the control where the value is.

1. **The line** — providers own transport and scaling; you own routing, policy, approval and the audit.
2. **The seam contract** — owns · guarantees · never-assume · on-failure, in your repository.
3. **Idempotency, retries, ordering and quotas** are contract fields, not handler details.
4. **Human gates live inside your pipeline**, as records, not console clicks.
5. **Prompt injection is input validation** — a schema between untrusted content and any tool.

Chapter 13: Upstream Product as a Dependency

Wrapping, Patching, and the Upgrade Rehearsal

---

1. The Three Ways, and Two Are Wrong

A team needs one behaviour changed in an upstream product — a login redirect, a quota. Vendoring the source is the obvious move: the change is two lines, and the copy is in the repository. Two years later they are pinned to a release from 2023, upstream no longer recognises the tree, and every security release

since has been a manual merge nobody did. The two-line change has become a permanent role.

That is the failure this chapter is about. The three structures decide how the work is done, and an upstream product is the strongest external constraint on all three. Chapter 11 chose what to bring in, and how you would leave it. This chapter is that choice's hardest case: an **upstream product** — a whole application you deploy and configure, whose behaviour, data schema and upgrade path are its own. A library is a dependency you can pin; a product is a dependency with opinions.

Three ways to hold it, and two are wrong.

Vendor it. Copy the source into your repository. Now you own the maintenance of a product you did not write, and every upstream release is a manual merge.

Fork it. Take the source and diverge. Now you own a product, its security patches, and the eventual reconciliation.

Wrap it. Keep the product as it ships, and put your behaviour in a layer you own: configuration as code, an override layer, and your data in your own store.

```
flowchart LR
  U["upstream release"] --> W["your wrapper<br/>config · overrides · data"]
  W --> A["your application"]
  P["your patches — deltas only"] --> W
  P -.->|"drift visible"| U
```

What this shows: the wrapper is the seam; patches are deltas against a product you still upgrade, not a fork you now maintain.

> **The rule this chapter defends:** wrap the product, patch it only as deltas, and rehearse every upgrade. **The day you copy the source into your own tree is the day you became its maintainer.**
>
> 💰 **Cost Check**: vendoring feels free and costs a permanent maintenance role; a fork feels decisive and costs the security patches you now owe; wrapping costs an afternoon and a runbook. The expensive choice is the one that looks like ownership without the authority — a copied source tree whose upstream no longer knows you exist.

🎯 What You'll Learn

- **A dependency with opinions** — what makes a product different from a library you can pin
- **Wrapping, not vendoring** — the three ways to hold a product, and why two make you its maintainer
- **Patch-only forks** — deltas, automation, visible drift, and a sync that is rehearsed
- **The upgrade rehearsal** — read · prepare · rehearse · verify · sign, and the evidence it leaves

✅ What to Check

- Is your upstream vendored, forked or wrapped? If the answer is "we copied it in", you are maintaining a product.
- Where does your data live — the product's store, or yours?

---

2. Wrapping, Not Vendoring

Part	What it holds	Where it lives
configuration as code	the product's settings, versioned	your repository, applied by a command
the override layer	your templates, policies, extensions	your repository, loaded by the product
your data	what the product stores, in a store you control	your database, with its own backups

The wrapper is the seam: the part of your estate the product can be cut away from. **It is what makes the product replaceable.** Everything you would need to reproduce behaviour lives in your repository; the product becomes an engine you can swap, upgrade or drop.

Vendoring loses that. The moment the source is in your tree, your behaviour and the product's are indistinguishable, and the next upgrade is a merge.

✅ What to Check

- Can you rebuild your behaviour from your repository, with a fresh copy of the product?
- Is any configuration living only in the product's admin console?

---

3. Patch-Only Forks

Sometimes the product must be changed — a bug that will not be fixed, a behaviour you cannot wrap. Then the rule is **patch-only**:

- **deltas, not source** — a patch series, or a small set of commits, against a known upstream version;
- **automation, not memory** — a build that applies the patches to a fresh upstream checkout;
- **drift visible** — a check that fails when the patch series no longer applies;
- **sync rehearsed** — an upgrade that re-applies and tests, not a merge resolved by hand.

A patch-only fork stays a dependency. A fork that becomes a branch of the source has become a product, and the maintenance is now yours indefinitely.

✅ What to Check

- Is the patch series applied by automation, or by a person following notes?
- Does anything fail when upstream changes a file your patch touches?

---

4. The Upgrade Rehearsal

An upgrade is the moment an upstream dependency becomes visible to users. It should be boring, and boring is a process:

1. **read** the upstream release notes and the diff;
2. **prepare** the upgrade — version bumps, config changes, patch re-application;
3. **rehearse** it in a preview environment, against your own data;
4. **verify** with the same checks that gate a release;
5. **sign** — a human accepts the change and its evidence.

What the agent prepares: the release-note summary, the diff analysis, the config delta, the patch re-application, the rehearsal run and its evidence.

What the human signs: the upgrade, the rollback decision, and anything the release notes describe as breaking.

The accelerator rule: compatibility software — an upstream product run for somebody else's convenience — never defines your architecture. If it did, the day upstream changes direction you would migrate an architecture, not a dependency. Keep a **migration-slice plan** ready: the order in which you would move your behaviour off the product, one slice at a time, and what each slice would need.

✅ What to Check

- Has an upgrade been rehearsed on real data, in a preview environment?
- Which of your upstream dependencies would change your architecture if they changed direction?

---

5. Upstream Governance

An upstream product makes decisions for you by default. Governance is the written answer to three questions:

Question	The record
what do we accept from upstream?	the licence and supply-chain verdict
what do we refuse, and how?	the patch inventory, and the wrapper
what do we document?	the upgrade runbook and its last rehearsal

Licence and supply-chain review is part of the decision, not a later legal step: a licence that forbids your use is a build-or-buy answer, and a dependency with no maintainer is a risk you have chosen. The verdict belongs in the **decision record** — the licence, the maintenance signal and the version you first adopted — beside the five axes, so the next person reads why at the same time as what.

✅ What to Check

- Is there a patch inventory, and does it match reality?
- Is the licence verdict recorded, or assumed?

---

6. Build the Wrapper

Artefacts

Artefact	What it is for	Where it lives
wrapper	config · overrides · data	`upstream/<product>/`
patch inventory	every patch, why it exists, where it applies	`upstream/<product>/patches/README.md`
upgrade runbook	read · prepare · rehearse · verify · sign	the Playbook
rehearsal record	one upgrade, with its evidence	the evidence pack

The patch inventory, as a shape — one row per patch, with its reason:

```yaml
# upstream/<product>/patches.yaml
product: <name>   pinned: <version>
patches:
  - id: <slug>
    reason: <the defect or behaviour it addresses>
    applies_to: <file or area>
    retire_when: <what upstream change would make it unnecessary>
```

Decisions

Choice	Default	What changes if you choose otherwise
wrap, vendor or fork	wrap	vendoring makes you the maintainer

Choice	Default	What changes if you choose otherwise
a fork	patch-only, automated	a divergent source tree you own
upgrade cadence	rehearsed, on a schedule	an upgrade attempted under pressure
architecture influence	none	the product defines your design

✅ Checks

- The wrapper rebuilds your behaviour on a fresh upstream copy.
- The patch series applies by automation, and a non-applying patch fails the build.
- One upgrade has been rehearsed on real data with its evidence.

Sequence

1. **Write the wrapper** — config, overrides, data. *Checkpoint: behaviour reconstructs from your repository.*
2. **Inventory the patches**, with retire conditions. *Checkpoint: every patch has a reason and a retirement.*
3. **Write the upgrade runbook.** *Checkpoint: read · prepare · rehearse · verify · sign.*
4. **Rehearse one upgrade** in a preview environment. *Checkpoint: the rehearsal names what broke.*

Done When

- Your behaviour survives a fresh copy of the upstream product.
- Every patch has a reason and a retirement condition.
- An upgrade has been rehearsed, verified and signed.

---

7. 🔑 Chapter Summary

An upstream product is a dependency with opinions — wrap it, patch it deliberately, rehearse the upgrade.

1. **Vendoring and forking both make you the maintainer**; wrapping keeps the product replaceable.
2. **The wrapper is config, overrides and your own data** — enough to rebuild your behaviour.
3. **Patch-only forks** stay dependencies: deltas, automation, visible drift, rehearsed sync.
4. **The upgrade rehearsal** is read, prepare, rehearse, verify, sign — with agents preparing and a human signing.
5. **Compatibility software never defines the architecture**; keep a migration-slice plan, with the order and the slices written down.

Chapter 14: The Integration Charter

Nine Decisions Before You Connect to Anything You Do Not Own

> 🧭 **Foundation link:** This chapter applies the **Engineering Structure** from Chapter 1 and the control split from Movement A to the boundary you do not own. It is what makes Part Three's control plane possible: a charter is the only thing a validator can check.

---

1. Every Integration Arrives With Its Defaults Already Chosen

an agent does with a decision you never made.

An integration is any system you connect to that you do not own: a database platform, an edge network, an upstream product, a payment provider, an AI model endpoint. The technology changes between them. The situation does not. Each one arrives with every choice already made on its behalf.

Every one of those choices was made before you arrived. The place your resources live — the *namespace* — carries the name from the first tutorial anyone followed. The credential is the first key the console offered. The address is whatever the resource happened to be called on the day it was created. The order in which two settings override each other is written down somewhere you have never had a reason to read. And the environment you build second will drift from the one you built first, because the second one was built from memory.

None of this is visible while you are the only person working on the integration. It becomes visible the moment an agent works on it — because an agent does not know which default you would have chosen. It implements the default it finds, completely and confidently, in a change that passes every gate you own.

That is the gap this chapter closes. Not a missing tool, and not a missing skill: **a set of decisions that exists implicitly, in defaults, and needs to exist explicitly, in writing, before the work starts.**

> **The claim.** An agent can be trusted with implementation only in proportion to how explicitly the boundary around that implementation has been drawn. The charter is that drawing.

🎯 What You'll Learn

- **The nine decisions** — identity and ownership, structure and address, and change and proof — and what fails when each is left to the platform's default.
- **Why precedence is the expensive one** — what happens when a setting lives in more than one store and the platform resolves them in an order nobody has read.

- **The four integration gates** every request passes between a resource and a caller — and why the error message describes the wrong one.
- **The charter as a file** — nine values that a command can check, so the boundary is enforced rather than remembered.

Two Systems, One Boundary

The nine decisions are not a checklist to hold in your head. They are the content of the boundary between two systems, and they sit on it in three groups.

```
flowchart LR
    subgraph Y["Your system"]
        A["your code<br/>your pipeline<br/>your people"]
    end
    subgraph B["The boundary — nine decisions"]
        direction TB
        I1["A · identity &amp;amp; ownership<br/>owner · actor · boundary"]
        I2["B · structure &amp;amp; address<br/>namespace · naming<br/>addressability · precedence"]
        I3["C · change &amp;amp; proof<br/>parity · change path · proof"]
    end
    subgraph X["The system you do not own"]
        Z["its defaults<br/>its documentation<br/>its versioning"]
    end
    Y --> I1 --> I2 --> I3 --> X
```

Read the diagram left to right as an order of work. You decide **who is acting and what they may do** before you decide **what it is called and how it is reached**; you decide **how it will be changed and proven** before you let anything be built on it.

---

2. The Nine Decisions

answers it instead of you.

A default is not a decision you avoided. It is a decision someone else made, on your behalf, without your context — and it will be whatever makes the vendor's onboarding shortest.

#	Decision	The question	The platform's default, if you stay silent	What fails
A1	**Owner**	Which repository, which file, which person owns this resource?	nobody, implicitly — it belongs to whoever created it	the orphaned setting: correct today, drifting tomorrow, no one's job
A2	**Actor**	Who authenticates — human, service, pipeline — and with what kind of credential?	the first key the console offered	a shared key becomes a second identity; nobody can say who did what
A3	**Boundary**	What may this actor create, read, change or destroy — and what is forbidden?	broadest-possible access, because narrow access breaks demos	one system quietly writing into another's namespace
B1	**Namespace**	One namespace per application, or one shared space?	one shared, default space	everything in the place a reset destroys
B2	**Naming**	How is a resource named, so the name says what it is and where it lives?	whatever the resource was called at creation	three near-identical names, one of them live
B3	**Addressability**	What is reachable, by whom, through which route and which grant?	whatever the quickstart exposed	a resource that exists, is correct, and answers *not found*
B4	**Precedence**	When more than one store can hold the same setting, which one wins?	documented, in a page nobody reads	the same command three times, each time asking the wrong store
C1	**Parity**	Is the second environment a copy of the first, or a different system?	whatever each pipeline declared independently	works in staging, fails in production, one declaration apart
C2	**Change path**	Command or console? If console, where is that step recorded?	wherever the person clicked	a manual step nobody can repeat and no gate can see
C3	**Proof**	What fails loudly if this is wrong — and does it ask the question the caller asks?	nothing; the deploy goes green	a green build over a system that cannot reach its own data

The nine divide into three families, and the families are done in order.

Identity and Ownership (A1–A3)

Before anything is built: **whose is it, who acts on it, and what may that actor not do.** These three decisions are the ones an agent can get catastrophically wrong without ever being obviously wrong — an over-broad credential works perfectly until it is used from the wrong place.

Structure and Address (B1–B4)

Before anything is connected: **what it is called, where it lives, how it is reached, and which setting wins.** Three of these four are naming questions. The fourth, precedence, is a behaviour question wearing a naming question's clothes, which is why the next section is devoted to it.

Change and Proof (C1–C3)

Before anything is trusted: **how it changes, how environments stay equal, and what fails when they are not.** A decision without a check is a preference; this family is what turns the other six into controls.

---

3. Precedence: The Decision That Argues Back

your own system.

Any platform mature enough to be useful holds some settings in more than one place: a global default, a per-scope override, a resource-level policy, a console value, an environment variable. This is not a design flaw. It is what happens when a product grows an API, a CLI and a console over the same configuration — and each surface needs somewhere to write.

What matters is that the platform states its **resolution order** — and that you read it before you write, not after you have been surprised.

```
flowchart TD
    R["the consumer — the code that asks"] --> S1
    S1["store 1 · per-scope override<br/>usually written first, hardened longest"] -->|wins| E["effective value"]
    S2["store 2 · global setting"] -.->|ignored while store 1 exists| E
    S3["store 3 · console / API / default"] -.->|ignored while store 1 exists| E
    W["what you probably edited"] -.->|store 2 or 3| S3
```

The failure has a consistent four-step shape:

1. You update the setting in the store you can see.
2. You verify it took — the store reads back correctly.
3. You restart the service.
4. The behaviour does not change.

At step 4 the instinct is to repeat steps 1–3, harder. The correct move is to ask a different question: **which store does the consumer actually read?** Where a platform documents an order, the answer is usually the oldest, most hardened store — the per-scope override — not the newest API surface.

> **The discipline.** Verify from the consumer's path, not your administrative one. Reading a setting back is not evidence that it is in effect. A request that succeeds is.

Two further habits make this survivable. Write to **every** store the platform consults, not the one that felt authoritative that day. And ship the check with the write, so the next person sees divergence as a failing command rather than a mystery.

---

4. Four Gates Between a Resource and a Caller

the error message names the wrong gate.

"Can they reach it?" is not one question. Four separate gates sit on the path, each with its own mechanism and its own way of failing.

```
flowchart LR
    Q["a request"] --> G1{"existence<br/>does it exist?"}
    G1 -->|no| F1["no such object"]
    G1 -->|yes| G2{"addressability<br/>is it routed or listed?"}
    G2 -->|no| F2["no such endpoint<br/>— and the object is intact"]
    G2 -->|yes| G3{"permission<br/>may this actor use it?"}
    G3 -->|no| F3["not authorised<br/>— and it is reachable"]
    G3 -->|yes| G4{"visibility<br/>which rows may it see?"}
    G4 -->|none| F4["empty result<br/>— looks like 'no data'"]
    G4 -->|some| OK["the answer"]
```

The reason this matters for an agent is order. Each message suggests a different fix, and only one of them is true:

- *No such endpoint* points at routing and exposure — while the object is present and correct.
- *Not authorised* points at credentials — while the binding is fine and the policy is wrong.
- *Empty result* points at missing data — while the rows exist and are filtered away.

An agent reading only the message will fix the wrong gate, confidently, and move on. A human reading only a green pipeline will not know a gate exists.

> **The design consequence.** Put the four gates in the certificate you hand the machine — and in the check you hand yourself. One command that walks all four, in order, is worth more than four pages of troubleshooting.

---

5. The Charter Is a File, Not a Document

A charter recorded in a wiki is a wish. The same nine answers in a reviewable file is a control, because it can be diffed, versioned, reviewed and **checked**. It lives beside the code it governs — with the rules, the manifest and the gates — not in a separate documentation system.

```yaml
# platform/<name>/charter.yaml
platform: <name>
owner_registry: <the manifest that names who owns what>
actor: "<who authenticates, with which credential kind, resolved how>"
boundary: "<what is forbidden to this actor>"
namespace_rule: "<one namespace per application | shared>"
naming_rule: "<the shape of a name>"
addressability_rule: "<what must be reachable, verified how>"
precedence: "<store 1 > store 2 > store 3>"
parity_rule: "<staging mirrors production, verified by which command>"
change_path: "<command | console + where that step is recorded>"
proof: "<the command that exits non-zero on divergence>"
```

The test for a value is whether a command could fail on it.

Not a charter value	A charter value
“use the console”	“the console step is recorded in the runbook and verified by `make verify`”
“one schema per app”	“every schema that exists must appear in the served list; checked by `make schemas`”
“keep environments similar”	“staging and production are verified by the same command; a declaration in one only is a defect”

If the answer cannot be checked, it is a note, not a decision — and notes do not survive a handover, human or agent.

---

6. What the Agent Prepares, What the Human Signs

This is the practical payoff of writing the nine answers down: the work separates cleanly, and the separation is visible in the charter itself.

The agent prepares — with evidence	The human signs
a draft of all nine answers, each with the page or file it came from	**A3 boundary** — what is forbidden is a policy judgement
the check that tests them, failing with the decision id	**B4 precedence** — which store wins is a commitment
the migration, workflow or configuration change that follows	**C1 parity** — what “same” means between environments is a risk decision
the reviewer’s summary: what changed, what was verified, what remains manual	**C2 change path** — what may be done at a console is an authority decision

Five of the nine can be automated once stated, because a machine can read a manifest and a document. The four on the right cannot: they encode authority, risk and policy. An agent that signs them has quietly taken authority it was never given — and the work will look excellent right up to the moment it matters.

---

7. Something You Can Check

The shape is the same on every platform:

```
$ <tool> platform check PLATFORM=<name> ENV=<env>
  ✅ A1 owner           every existing resource has an owner in the manifest
  ✅ B3 addressability  every existing resource is reachable
  ❌ B4 precedence      the operative store is not the one the charter names
```

Three properties separate a real check from a reassuring one:

1. **It reads the live system**, not the charter’s own copy of it.
2. **It fails on divergence, with the decision id** — so the fix is named, not hunted.
3. **It asks what the caller asks** — through the same door the application uses, not the administrator’s.

Part Four turns this into a registry, a validator and a console verdict. This chapter’s job is the decisions it validates.

---

8. Build the Charter

The nine decisions are the argument; the file is the control. This is the build: one file, one check, and the sequence in which the answers stop depending on each other.

Artefacts

Artefact	What it is for	Where it lives
charter file	the nine answers, checked	`platform/<name>/charter.yaml`
divergence check	fails when the platform stops matching its charter	the gate, one command
decision log	why each answer was chosen, and what would change it	beside the charter, or a section in it
instance appendix	the same nine keys answered for a named platform	`appendices/appendix-<name>-charter.md`

The file, as a shape — nine keys, each answered with something a command could fail on:

```yaml
# platform/<name>/charter.yaml
platform: <the name this charter governs>
owner_registry: <the manifest that names who owns what>
actor: <who authenticates, with which credential kind, resolved how>
boundary: <what is forbidden to this actor>
namespace_rule: <one namespace per application | shared>
naming_rule: <the shape of a name>
addressability_rule: <what must be reachable, verified how>
precedence: <store 1 > store 2 > store 3>
parity_rule: <what staging mirrors, verified by which command>
change_path: <the command, or the console step and where it is recorded>
proof: <the command that exits non-zero on divergence>
```

Decisions

Choice	Default	What changes if you choose otherwise
where the charter lives	beside the code it governs, in the repository	a wiki charter cannot be diffed, reviewed or checked
what counts as an answer	a value a command could fail on	an aspiration reads like a control and is neither
the precedence rule	written down, one order, one owner	two stores resolving silently is the expensive failure
the divergence check	in the gate, not on a calendar	a check nobody runs is a document with a schema

✅ Checks

- The **divergence command exits non-zero** when the platform differs from the charter — on a planted difference.
- A charter value that **names no command** is refused at review: if it cannot fail, it is not a value.
- Every key is answered: **nine keys, nine values** — an absent key is a decision nobody made.
- The appendix uses the **same nine keys**, so platforms stay comparable.

Sequence

1. **Identity and ownership first** — actor, boundary, owner registry. *Checkpoint: the boundary names what the actor may not do.*
2. **Structure and address** — namespace, naming, addressability. *Checkpoint: each answer can be checked by a command.*
3. **Change and proof** — precedence, parity, change path, proof. *Checkpoint: the proof command exists and fails on a difference.*
4. **Write the file**, then the check that consumes it. *Checkpoint: the check runs in the gate.*
5. **Plant a divergence** and watch the check fail. *Checkpoint: the failure names the key that diverged.*

Done When

- Nine keys, all answered, each answer checkable.
- One command fails when the platform drifts from the charter.
- A second platform is chartered by copying the keys, not by rewriting the method.
- The decision log says why each answer was chosen.

---

9. Closing: Portable by Construction

A database platform, an edge network, a cloud account, an upstream product — the nouns differ and the nine questions do not. A charter that names the platform it describes is a product manual. A charter that names **decisions** is a method, and it survives the platform's next major version.

Write the nine answers for the system you are connecting to today. Then notice how little of the document you would have to change to use it for the next one.

That is the test of a boundary you actually control: it reads the same when the vendor changes.

Where the instances live

This chapter deliberately names no vendor, because the questions do not change when the platform does. The answers are yours to write, and they belong in one file the divergence check can read — `platform/<name>/charter.yaml` in the repository, with a row per decision and the command that enforces it. Write them for the system you are connecting to today; then notice how little of the document would change for the next one.

Evidence: one command that fails when the integration diverges from its charter.

Template: the charter file (nine keys) and the check it feeds.

Chapter 15: Contracts That Keep Services Independent

Shared Infrastructure Without Shared Fate

---

1. Independence Is a Contract, Not a Boundary

The three structures scale by staying separable. Two services can share a database, a cluster, a network — and remain independent — only if each owns something the other cannot change. The moment they share a table, an upgrade or a bad migration takes both down together, and the failure is not the migration's: it is the ownership model's.

Shared infrastructure is a cost decision. Shared fate is an architecture decision, and it is usually made by accident.

> 💰 **Cost Check**: a shared schema saves a migration and costs an outage that two teams argue about. A destructive statement saves a review and costs the tables it removed. Expand/contract costs three deploys instead of one and buys the ability to release in any order. The expensive choice is the one that looks like speed and is paid by someone else's release.

🎯 What You'll Learn

- Why shared infrastructure is fine and shared fate is not
- The per-service ownership model: schema, version range, applier
- Expand/contract migrations, and the ban that makes them safe

The Two Failures

Failure	The cause	The shape of the fix
one migration breaks a neighbour	nobody owned the schema	a schema per service, an applier per owner
one deploy needs another's release	a shared contract with no version	version ranges, contract tests

```
flowchart LR
  S1["service A"] --> SA["schema A<br/>range 1000–1999"]
  S2["service B"] --> SB["schema B<br/>range 2000–2999"]
  SA --> D["shared platform"]
  SB --> D
  SA -.->|"never"| SB
```

What this shows: one platform, two owners — the dashed edge is the one that must not exist.

The Configuration Seam

The seam you have already met sits between two systems; it also sits between logic and environment. A contract between services also applies to how automation interacts with different environments. We enforce a configuration seam: a strict boundary where the immutable automation logic meets the **environment context**. By routing all environment variables, secrets, and endpoints through this validated seam, we ensure that the automation behaves predictably whether it is provisioning a local development sandbox or a production cluster. The logic never changes; only the environment context injected at the seam does. This is how we prevent configuration drift and guarantee idempotent, reversible deployments.

✅ What to Check

- Which of your services share a schema? That is shared fate, whatever the diagram says.
- Does any migration apply without verifying that a neighbour's objects still exist?

---

2. Per-Service Ownership

A schema of its own. Not a table prefix — a schema, so the boundary is structural rather than a naming convention.

A migration version range. Disjoint ranges mean two services never write the same version number, and an applier can apply "mine" without guessing.

An applier that verifies its own objects and its neighbours' presence. The migration that drops a sibling's table is the failure this exists to prevent, and it is caught by a post-condition: *my schema is complete, and every sibling schema still exists*.

The Applier's Post-Conditions

```
apply migrations in my range, in order
assert: my schema contains the objects my migrations declare
assert: every sibling schema declared in the manifest still exists
fail loudly, at the applier, if either assertion is false
```

A destructive statement — `drop`, `truncate`, an unbounded `delete` — is refused by the gate unless it is declared, reviewed and owned. The ban is not caution; it is the difference between a migration and an outage.

✅ What to Check

- Does each service own a schema and a disjoint version range?
- Does your applier assert that siblings still exist? A reset that fails hours later as someone else's outage is the failure here.

---

3. Expand/Contract

A live deploy runs old and new code together for a window. A migration that renames a column in one step breaks the old code during that window — so the change is split:

Phase	The change	What is true
expand	add the new column; write to both	old code works, new code works
migrate	backfill	both columns agree
contract	drop the old column	only the new code exists

Each phase is a separate deploy, with its own verification. The compatibility window is not an inconvenience; it is the design, and it is what makes a migration independent of the order services deploy in.

Verify against a snapshot in CI. A migration that has never run against a realistic dataset has been reviewed, not tested.

✅ What to Check

- Does any migration both add and remove in one step? That is a break waiting for a window.
- Has the last migration run against a production-shaped snapshot?

---

4. Least Privilege and Contract Tests

Least-privilege per-service roles. A service's credential can reach its own schema and nothing else. Anything wider is an incident waiting for a mistyped query.

Contract tests between services. Each service tests against the contract its neighbour publishes — the fields it promises, the errors it returns — not against the neighbour's implementation.

Registries for ports, surfaces and environment variables. A port is a contract with a number; a surface is a contract with an address; an environment variable is a contract with a name. Each needs one home.

What Agents Write, What a Human Signs

- **Agents write:** migration code within the range, tests, contract updates that follow the pattern.
- **A human signs:** anything destructive, anything crossing an ownership boundary, any change to a shared contract.

The line is not about capability. It is that a destructive change and a cross-ownership change are the two where a wrong answer is expensive and irreversible, and those are the ones a human should own.

✅ What to Check

- Can a service's role reach anything outside its schema?
- Does each cross-service call have a contract test, or an assumption?

---

5. Build the Workbench

Artefacts

Artefact	What it is for	Where it lives
ownership manifest	service · schema · version range	the suite manifest
applier	applies its range, asserts its objects and its siblings'	the repository's scripts
contract set	the promises each service makes	`contracts/<service>/`
destructive gate	refuses an undeclared destructive statement	the gate

The ownership manifest, as a shape — one row per service:

```yaml
# suite manifest — the ownership record
services:
  - name: <service>
    schema: <schema name>
    versions: <1000–1999>
    applier: <path>
    destructive_statements: <declared ids, reviewed>
```

Decisions

Choice	Default	What changes if you choose otherwise
schema per service	yes	a shared table is shared fate

Choice	Default	What changes if you choose otherwise
version ranges	disjoint, per service	two services writing the same number
migrations	expand/contract, phase per deploy	a rename that breaks a running version
destructive statements	declared, reviewed, gated	an outage that passed review

✅ Checks

- The applier **fails when a sibling schema is missing**.
- A destructive statement **without a declaration** is refused.
- A cross-service call has a **contract test**.

Sequence

1. **Declare ownership** — schema and range per service. *Checkpoint: no two services share a schema.*
2. **Write the applier's post-conditions.** *Checkpoint: a planted drop of a sibling fails it.*
3. **Split the next migration** into expand · migrate · contract. *Checkpoint: each phase is a deploy.*
4. **Add the destructive gate.** *Checkpoint: an undeclared drop is refused.*

🎯 Done When

- Every service owns a schema and a disjoint range.
- A migration cannot remove a sibling's objects without failing.
- A live deploy runs two versions without a migration breaking either.

---

6. 🔑 Chapter Summary

Shared infrastructure is affordable; shared fate is not.

1. **Independence is a contract** — a schema, a version range, an applier that verifies.
2. **The applier asserts its own objects and its siblings' presence** — a reset fails where it happens.
3. **Expand/contract** is the only pattern that survives two versions running at once.
4. **Least privilege and contract tests** make the boundary enforceable.
5. **Agents write the migrations and tests; a human signs the destructive and the cross-ownership change.**

Chapter 16: Sovereignty, Portability and Exit Drills

Proving You Could Leave

---

1. Sovereignty Is a Requirement

The three structures hold data someone else may have a claim on. Sovereignty is the answer to *who can read this, where is it, and what governs it* — and it is a requirement, like uptime, not a value statement.

Four facts, each written down:

Fact	The question
jurisdiction	which legal regimes govern the data?
residency	where is it stored, and where does it move?
access	who can read it — you, a provider, a support engineer?
governance	what policy applies, and where is it recorded?

An answer that cannot survive a client's questionnaire is not an answer. Writing the four facts into the repository is what makes sovereignty checkable rather than asserted.

> 💰 **Cost Check**: sovereignty decided by default costs a client questionnaire you cannot answer; a vendor SDK in core costs the rewrite you cannot schedule; an exit that was never drilled costs the estimate nobody can give. A drill costs a day and produces a number. The rest costs the same day, annually, in a meeting where everybody guesses.

🎯 What You'll Learn

- The four facts sovereignty is made of
- Portability as a gate that fails a build
- The exit drill that replaces theorising

Where Sovereignty Leaks

Leak	How it happens
a provider's default region	nobody chose it; it was the default
a support session	a human reads production data without a record
a backup in a convenient bucket	residency decided by convenience
an analytics export	data moves to a third jurisdiction quietly

```
flowchart LR
  D["data"] --> J["jurisdiction"]
  D --> R["residency"]
  D --> A["access"]
  D --> G["governance"]
  J --> C["the four facts, written down"]
  R --> C
  A --> C
  G --> C
```

What this shows: sovereignty is four facts, and a missing one is the leak.

✅ What to Check

- Can you answer all four facts for your most sensitive dataset?
- Which of them was decided by a default rather than a choice?

---

2. Portability as a Gate

Portability is not a virtue; it is a **check**. The rule: core logic may not import a vendor's SDK. Adapters may. The check fails the build when the rule is broken.

That requires defining **"core"** — the part that must be replaceable. It is the domain logic, the policy, the method. It is not the adapter that talks to a provider; that is the seam, and it is allowed to be vendor-specific because it is meant to be replaced.

Where	May be vendor-specific	Must be portable
domain logic	no	yes
adapters	yes	—
the seam	yes, behind an interface	—
migrations	standard SQL where possible	the recovery path

The lock-in check fails on a violation: a vendor import in core, a proprietary query syntax in a migration, a recovery procedure that only a vendor's console can perform.

✅ What to Check

- Is "core" defined, so the rule is enforceable?
- Does a vendor import in core fail the build, or a review?

---

3. Exit Drills

Porting got cheaper. Agents read schemas, generate adapters, and draft a migration path in hours. That changes the economics of exit: **rehearsing a departure is cheaper than theorising about it**, so the drill replaces the discussion.

The drill, in five steps:

1. **export** the data through the portable path;
2. **import** it into a stand-in (a different provider, a local database);
3. **run the application** against the stand-in;
4. **time it**, and record what broke;
5. **decide**: is the path real, or was it a story?

The drill produces the only honest number in a sovereignty conversation: **what leaving would cost, in hours and in breaks.**

✅ What to Check

- Has any exit been rehearsed, or only described?
- Which dependency would fail the drill today?

---

4. Provenance and SBOM

An exit is not only data: it is also knowing **what you shipped**. Provenance and a software bill of materials answer *what components are in this release, and where did they come from* — which matters to a client audit, to a security review, and to the day you must rebuilt the release somewhere else.

Artefact	Answers
SBOM	what components, and which versions
provenance	who built it, from which source, when
the lock-in check	whether the build depends on one vendor's tooling

All three are produced by the pipeline, not by a document written afterwards. A provenance record assembled by hand is a claim.

✅ What to Check

- Can you list the components of your last release from an artefact, not from memory?
- Does the build depend on a vendor's tooling to reproduce?

---

5. Build the Workbench

Artefacts

Artefact	What it is for	Where it lives
sovereignty record	the four facts, per dataset	`docs/sovereignty/<dataset>.yaml`
portability rule	what "core" is, and the import ban	the Law, plus a check
lock-in check	fails on a vendor import in core	the gate
exit drill record	the drill, its timing, what broke	the evidence pack

The sovereignty record, as a shape — four facts, no prose:

```yaml
# docs/sovereignty/<dataset>.yaml
dataset: <name>
jurisdiction: <regimes that govern it>
residency: <where stored, where it moves>
access: [<you>, <provider role>, <support>]
governance: <policy, and where it is recorded>
```

Decisions

Choice	Default	What changes if you choose otherwise
where the facts live	the repository, per dataset	a questionnaire answered from memory
portability rule	core may not import a vendor; adapters may	a build that only one vendor can reproduce
exit drills	one per dependency per year	a cost everybody quotes and nobody measured
provenance	produced by the pipeline	a claim assembled afterwards

✅ Checks

- All four sovereignty facts are recorded for each dataset.
- A **vendor import in core fails the build**.
- One exit drill has been run, timed and recorded.

Sequence

1. **Record the four facts** for the most sensitive dataset. *Checkpoint: none of them is "default".*
2. **Define core** and add the import check. *Checkpoint: a planted vendor import fails.*
3. **Run one exit drill**. *Checkpoint: the timing and the breaks are recorded.*
4. **Produce provenance** in the pipeline. *Checkpoint: the release's components come from an artefact.*

🎯 Done When

- Sovereignty is four recorded facts, per dataset.
- A vendor import in core cannot merge.
- The cost of leaving one dependency is a measured number, not an opinion.

---

6. 🔑 Chapter Summary

Sovereignty is four facts; portability is a gate; exit is a drill.

1. **Sovereignty** = jurisdiction · residency · access · governance, recorded per dataset.
2. **Portability is a check**, not a virtue — define core, ban the vendor import, fail the build.
3. **Exit drills are cheap enough to run**, and they produce the only honest cost of leaving.
4. **Provenance and SBOM** are how you prove what you shipped, from an artefact.
5. **A default is not a decision** — every sovereignty leak starts with one.

Chapter 17: Observability as the Evidence Layer

Signals Bound to Invariants

---

1. A Signal Is a Claim About an Invariant

The three structures fail quietly without evidence. A gate proves an invariant at a moment; a signal proves it **continuously** — and the value of the signal is exactly the invariant behind it.

So the design question is not *what should we chart?* It is *which invariants must be observable, and what reads them?*

> 💰 **Cost Check**: a dashboard nobody reads costs the build and the maintenance of a second implementation. A missing signal costs the incident discovered by a customer. An agent reading a stale green costs the decision made on an hour-old number. The signal table is the cheap artefact; the parallel dashboard is the expensive one.

🎯 What You'll Learn

- How to bind every critical invariant to a live signal
- Why the signal and the gate must share one check logic
- Freshness as a first-class reading

The Two Failures

Failure	What it looks like
a dashboard of everything	nobody knows which panel matters
a signal with no invariant	a green line that proves nothing

The fix is a table, not a tool: **invariant → signal → what reads it → what freshness it needs**.

```
flowchart LR
  I["invariant"] --> C["check logic"]
  C --> G["gate — at a moment"]
  C --> S["signal — continuously"]
  S --> H["human reads it"]
  S --> A["agent reads it"]
  S -.->|"stale"| X["must read STALE, never green"]
```

What this shows: one check logic feeds both the gate and the signal — the second half of "never a second source of truth".

✅ What to Check

- Take your five most critical invariants. Is each one a signal you would notice?
- Does any dashboard panel answer a question nobody asked?

---

2. One Check Logic, Two Consumers

A gate and a signal should read the same check. The gate runs it at merge; the signal runs it continuously. If they are implemented twice, they will disagree — and the disagreement is discovered at the worst moment, when someone is deciding whether to trust the console or the pipeline.

Consumer	Runs	Answers
the gate	at merge, on every change	may this land?
the signal	continuously, against live state	is it still true?

The rule: the check logic is written once, and both consume it. A dashboard that recomputes an invariant with its own query is a second implementation, and it is wrong the day the first one changes.

✅ What to Check

- Pick a signal. Is its logic the same code as the gate's, or a copy?
- If the copy drifted, would anything catch it?

---

3. SLOs, Budgets, and Two Kinds of Health

A signal without a threshold is a graph. An SLO gives it a line; an error budget gives it a consequence: *spend this, stop feature work, fix the reliability*.

System health and feature health are different. System health is whether the platform works: latency, errors, saturation. Feature health is whether what you built does what it promised: gate escapes, rework, adoption. A dashboard that mixes them produces a reading nobody can act on.

Health	Signals	The decision it drives
system	availability · latency · saturation	reliability work, capacity
feature	gate escapes · rework · a signal's freshness	design, review, the next iteration

✅ What to Check

- Does each critical signal have a threshold and an owner?
- Are system and feature health on the same panel? Separate them.

---

4. Machine-Consumable Evidence

The agent is a consumer of the evidence layer. An agent that can read the signal can verify its own work, notice a stale reading, and report it. An agent that cannot will infer — and its inference is a claim, not evidence.

That makes **structured output a first-class requirement**: a signal that exists only as pixels is available to humans alone. The same signal as a string, a file, or a query is available to both.

Freshness is first-class. A stale signal must read as stale, never as healthy. A reading from an hour ago presented as current is the most expensive lie a dashboard can tell, because it is believed.

✅ What to Check

- Can an agent read your critical signals, or only a human?
- Does a stale reading say so, in the output rather than in a colour?

---

5. Agent Observability

If the system is observable, so is the thing operating it. The agent's counterparts to latency and errors are:

Signal	What it reveals
cost per task	an oversized context or a tiering mistake
rework	a gate or routing problem
gate escapes	the quality of the gate estate itself
provenance	who or what wrote a change

These belong in the same evidence layer because they answer the same question at a different scale: *is the system still doing what it promised?*

✅ What to Check

- Can you name the agent's cost, rework and escapes for the last period?
- Is any of them in the same place as system health, or scattered per tool?

---

6. Build the Workbench

Artefacts

Artefact	What it is for	Where it lives
signal table	invariant · signal · reader · threshold · freshness	`docs/signals.tsv`
SLO definitions	the line, and the budget	beside the table
machine output	the signal as structured text	the collector's output
drill-as-signal	a drill's result published as a signal	the evidence layer

The signal table, as a shape — one row per invariant, and the reader named:

```
# docs/signals.tsv — one invariant per row
invariant   signal      reader  threshold   freshness
no secret in repo   scan result gate+agent  0 findings  24h
api reachable       probe       agent           p95<300ms   5m
backup restorable   drill result    agent+human last<30d    30d
```

Decisions

Choice	Default	What changes if you choose otherwise
the unit	invariant, not metric	dashboards of everything
check logic	shared with the gate	a parallel truth that drifts
output	structured, agent-readable	a signal only humans can see
freshness	printed with the value	a stale reading believed as current

✅ Checks

- Every critical invariant has **a signal, a threshold and a freshness window**.
- The signal's logic is **the gate's logic**, not a copy.
- A stale reading **says STALE** in the output.

Sequence

1. **List the invariants** that must be observable. *Checkpoint: each one has a signal that would fail it.*
2. **Bind the check logic** — one implementation, two consumers. *Checkpoint: the gate and the signal agree.*
3. **Set thresholds and budgets.** *Checkpoint: a breach has an owner and a consequence.*
4. **Add freshness** to the output. *Checkpoint: an hour-old reading cannot be read as current.*

🎯 Done When

- Every critical invariant is a signal an agent can read.
- The signal and the gate cannot disagree, because they share one logic.
- A stale reading is impossible to mistake for a healthy one.

---

7. 🔑 Chapter Summary

Observability is the evidence layer: every critical invariant, one check logic, two consumers.

1. **A signal is a claim about an invariant** — a chart without one proves nothing.
2. **The gate and the signal share one check logic**; a dashboard that recomputes is a second truth.
3. **SLOs and budgets** make a signal actionable; **system and feature health are different**.
4. **Agents read the signals too** — structured output is a requirement, not a nicety.
5. **Freshness is first-class**: stale must read as stale, never as healthy.

Chapter 18: Recoverability

The Restore You Have Actually Run

---

1. What "Backup" Means Now

The three structures assume the system can come back. Recoverability is the difference between a copy and a recovery — and the distinction starts with a question most backup plans never ask: **what can we rebuild, and what can we not?**

Data	Rebuildable?	So
generated artefacts, caches, indexes	yes	rebuild them; do not back them up
configuration, rules, manifests	yes, if in the repository	the repository is the backup
user data, audit records	**no**	back it up, and prove the restore
keys and credentials	**no**	custody is the recovery

Backing up what you can rebuild is expensive theatre. Backing up what you cannot rebuild is the whole job.

> 💰 **Cost Check**: backing up what you can rebuild costs storage, restore time and a drill nobody finishes. A copy in the same account costs nothing and survives nothing. A key stored beside its data costs the whole backup. The cheap plan is the one that classifies first and stores second.

🎯 What You'll Learn

- Which data is irreplaceable and which is derived
- Versioning versus copy-out, and why both are needed
- Key custody: the part that makes a backup decorative

Two Copies, Two Different Failures

Versioning survives mistakes: a delete or an overwrite can be undone from history. **Copy-out** survives account loss: the data exists outside the provider entirely.

A versioning bucket is not a backup. A copy in the same account is not a backup. Both fail the same day, in the same way, and the plan that relied on them looks complete right up until it is tested.

```
flowchart LR
  D["data"] --> V["versioning<br/>survives a mistake"]
  D --> C["copy-out<br/>survives account loss"]
  V --> R["restore drill"]
  C --> R
  R --> P["a proven restore<br/>— or a documented gap"]
```

What this shows: two mechanisms, two failures, and one drill that proves the pair.

✅ What to Check

- Which of your data is irreplaceable? Is any of it only versioned?
- Does any copy live only inside the account it protects?

---

2. Retention as Policy

Retention is a policy: *daily for a week, weekly for a month, monthly for a year*. It is enforced by the store's lifecycle rules — declared, versioned, reviewable — and never by a script that deletes.

A deletion script is a second implementation of a policy the platform already has, and it is the one that runs at 3am and removes the wrong prefix.

The Retention Policy, as a shape

```
# retention — declared, not scripted
daily:   { keep: 7 }
weekly:  { keep: 4 }
monthly: { keep: 12 }
enforced_by: <lifecycle rule id>
never_delete: <the artefacts that must survive every policy>
```

✅ What to Check

- Is retention a rule in the store, or a script in the repository?
- Does anything you must keep forever have an exception written down?

---

3. Key Custody

If the key that decrypts the backup lives only beside the backup, the backup is decorative. Custody is the recovery — and it is the step most plans omit, because the plan describes copies and not access.

The question	The record
where is the key?	outside the account, in a named store
who can use it?	the roles, and the recovery path
what if that person is unavailable?	the break-glass procedure, with its own audit

Break-glass is a control, not a loophole. It is documented, audited, and rehearsed — because the day it is needed is the day the normal path is gone.

✅ What to Check

- Does the key exist outside the account it protects?
- Has the break-glass path been used in a drill, and recorded?

The other half of the way back is the release's rollback (chapter 25). A restored copy and a restored deployment are different returns: the deployment costs nothing but a minute, and the copy costs every write since it was taken. Knowing which one an incident needs is the decision chapter 25 makes before the release, not during it.

4. Restore Drills as a Signal

RPO and RTO, sized for a small organisation. RPO is how much data you can lose; RTO is how long you can be down. Both are decisions, and both are cheap to state and expensive to discover.

The drill proves two things: the copy is restorable, and the runbook is followable. A restore that works because one person knows the undocumented steps has proven the person, not the plan.

The result is a signal. *Last successful restore: 12 days ago* is an observable invariant (Chapter 17's table), and it is the one that turns recoverability from a document into evidence.

The Drill Record

```
drill:
  at: <timestamp>
  from: <which copy — versioning | off-account>
  target: <where it was restored>
  rto_actual: <how long it took>
  broke: [<what the runbook got wrong>]
  next: <the fix, and the date of the next drill>
```

✅ What to Check

- When was the last successful restore? Is it a signal, or a memory?
- Did the last drill follow the runbook, or did it need someone who knew the steps?

---

5. What Not to Back Up

The inverse rule matters as much as the rule:

- **do not back up what you can rebuild** — indexes, caches, generated artefacts;
- **do not back up what the repository already holds** — rules, manifests, config;
- **do not keep a backup whose restore nobody has run** — it is a rumour with a bill.

Every artefact you exclude makes the restore faster and the drill cheaper. Every artefact you include without a reason makes the drill longer and the signal weaker.

✅ What to Check

- Name three things you back up that you could rebuild. Remove them from the plan.
- Does the restore runbook fit on one page?

---

6. Build the Workbench

Artefacts

Artefact	What it is for	Where it lives
retention policy	declared, lifecycle-enforced	the store, reviewed
copy-out	the off-account copy	a separate account or store
key custody record	where the key is, who can use it, break-glass	`docs/recovery/keys.md`
restore runbook	one page, followable	the Playbook
drill record + signal	the last restore, published as a reading	the evidence layer

Decisions

Choice	Default	What changes if you choose otherwise
what to back up	only the irreplaceable	the rebuildable, which slows every restore
copies	versioning **and** copy-out	one mechanism that fails one way
retention	lifecycle rules	a script that deletes at 3am
the signal	last successful restore, published	a memory nobody checks

✅ Checks

- A restore from the **off-account copy** succeeds, on a schedule.
- The runbook is followable by someone who did not write it.
- The last restore's age is **a signal**, with a threshold.

Sequence

1. **Classify the data** — rebuildable or not. *Checkpoint: the backup list contains only the second.*
2. **Add copy-out** and a lifecycle retention policy. *Checkpoint: a copy exists outside the account.*
3. **Record key custody** and the break-glass path. *Checkpoint: the key is outside, and reachable.*
4. **Run the restore drill** and publish the result as a signal. *Checkpoint: the age is observable.*

🎯 Done When

- Every irreplaceable dataset has a copy outside the account.
- The restore has been run by someone following the runbook.
- The age of the last restore is a signal with a threshold.

---

7. 🔑 Chapter Summary

A backup is a copy; recovery is a drill.

1. **Back up only what you cannot rebuild**; the rebuildable is theatre.
2. **Versioning and copy-out fail differently** — a versioned bucket is not a backup.
3. **Retention is a lifecycle rule**, never a deletion script.
4. **Key custody is the recovery**; a key beside its data makes the copy decorative.
5. **The drill proves the copy and the runbook**, and its age is a signal.

Chapter 19: Engineering the Quality Gate

From a Pile of Checks to One Gate Estate

Every team ends up with checks. Lint, tests, a vulnerability scan, a migration guard. Each was added after something went wrong, and each one earns its place.

What rewards design is not the check. It is the **set**: which decisions a change must pass, where in the work each decision is made, and what evidence lets a person act on it. A team that designs that set runs fewer checks, finds failures earlier, and can answer the question every incident review asks — *would this have been caught?*

This chapter builds that set. It starts from the decisions rather than the checks, places each decision at a moment in the work, and ends with an estate you can draw on one page.

---

1. The Pile You Have

Every check in a pipeline has a story. A release shipped a defect a test would have caught, so the test was added. A credential reached a log, so the scan was added. A migration broke a neighbouring table, so the guard was added.

Each addition was right on the day it landed. Nobody could argue against it, and nobody did.

The difficulty is what the additions make together. A pipeline assembled that way has no owner and no design: one question gets answered by two commands with different bodies, a check ends up in whichever file was easy to edit, and some stop running when the workflow around them changes. Nothing announces the loss. A check that no longer runs looks exactly like a check that passes.

🎯 What You'll Learn

- What a gate, a moment and a gate estate are, in words you can use in a review
- Why a check's name and a check's body drift apart, and what that costs on every change
- How to place every check you have at one of four moments, and name the ones nothing runs

Four Words, Defined Here

Word	What it means	What it is not
check	one thing a machine verifies — lint, the tests, a secret scan, a type check	not a verdict; a check produces a result
gate	a check a change must pass before it may proceed	not a report, a dashboard or a notification
gate estate	all the gates of one organisation, taken as a set	not one gate or one repository's gates
estate	everything one person operates: repositories, services, hosts	not the gates — the estate is what the gates protect

The cost of a pile is not the minutes the checks take. It is that two questions have no answer. *Would the last incident have been caught?* and *which command do I run before I merge?* When both answers are "it depends who you ask", the checks are measuring the pipeline rather than protecting the estate.

A **gate estate** is the other thing: the small set of decisions a change must pass, each with a name, a place in the work, and evidence a person can read. The set is small because decisions are few — merge, deploy, release — while checks are many.

This is the book's third structure made executable. **The prompt structure** states what must be true, **the engineering structure** decides how it is checked, and **the workflow structure** is where those checks become decisions a change cannot pass without. A gate estate is the workflow structure written down, and this chapter builds it one decision at a time.

> 💰 **Cost Check**: an undesigned pipeline is not paid for once. It is paid on every change — a wait for a gate that could not fail, a review that argues again over which command is authoritative, an incident that a check would have caught if the check still ran. The minutes are the smallest part.

---

2. A Gate Is a Contract, and It Declares Five Things

A gate is a place work cannot pass without a decision. Four things are needed for that decision to mean anything: **what must be true before it runs**, **what it checks**, **what it concludes**, and **what may happen next**.

Most pipelines implement the second and skip the first. A check reads something — a file, a table, a deployed address — and reports on it. Whether that something was the right thing is assumed, and the assumption is where green results come from that mean nothing.

It happened to GitLab on 31 January 2017. An engineer working late on a database server removed the wrong directory, and a few hundred gigabytes of production data went with it. Five recovery mechanisms existed and each failed in a different way, including routine backups whose upload had been failing quietly for some time; the data was restored from an older staging snapshot, and six hours of work was gone. The recovery gate existed and had been believed for years. Its precondition — *this copy is the database* — had never been written down, so nothing checked it.

The five declarations are what a gate promises, and a green result is only as good as the part the reader checked.

Declaration	The question it answers	The failure when it is missing
Precondition	what must be true **before** it runs	a green result about a world that was not there
Invariant	what must always be true	nobody knows why the gate exists, so nobody can delete or extend it
Scope	exactly what it reads and what it ignores	the reader assumes it covered more than it did
Verdict	pass, fail, or not-verified — and what each means	"not run" is read as "passed"
Cost	how long it takes, and on which moment	it drifts onto a moment whose budget it violates

Precondition is the one nobody writes down, and it produces the false greens. A gate that reads a copy has a precondition: the copy exists, and it is current. A gate that probes an address has one: the deployment happened. A gate that compares a table against a declaration has one: the declaration was read, not defaulted. Precondition is not scope — scope is what the gate looks at, precondition is what had to be true for looking at it to mean anything.

Invariant is the sentence the gate exists to keep true: *"no credential is committed"*, *"every endpoint has a test"*. Without it the gate has no reason to exist, and nobody can tell whether a change made it obsolete or merely inconvenient.

Verdict has three states. A gate that can only say pass or fail will eventually say pass about something it could not read — a file it lacked permission for, a service that was down, a check it skipped. *Not verified* is a verdict, and it is different from *pass*.

Scope is the expensive one. A gate's name suggests a scope; its body reads a scope. When the two disagree, the gate is honest code wearing a dishonest name, and the reader inherits the claim without the check. A command called `validate` that runs the content checks and the syntax checks will be read as validating the change, including the parts it never reads.

Write It Down in Five Sentences

```
precondition: the staged file list was read, not defaulted
invariant:    no credential reaches the repository
scope:        every staged file; ignores vendored/ and *.min.*
verdict:      pass | fail | not-verified (a scan that could not read a file)
cost:         under 2s — atomic moment
```

Five lines, in the file where the gate lives. When one is wrong, the argument about the gate becomes a conversation about a sentence instead of a memory.

✅ What to Check

- Take your slowest gate. Write its invariant. If you cannot, it exists because of an incident nobody remembers.
- Name the exact paths your gate does **not** read. If the name suggests otherwise, the name is the defect.
- Name the gate's precondition. If it reads a copy, say which copy and how you know it is current.

3. A Gate Has a Name, a Moment and an Environment

Three questions describe any gate, and a team that answers all three can place a check in a sentence:

Question	The word for it	Example
what does it decide?	its **name**	*deployment gate*
when does it fire?	its **moment**	*commit*, *full*, *scheduled*
where does it run?	its **environment**	*local*, *staging*, *production*

The Five Names

Five decisions cover most of what a delivery estate needs. Each has the name the industry already uses, so a reader joining the team knows what it means before the first stand-up.

The gate	The command	The decision it makes
commit gate	`pre-commit`	may this be committed? — the staged diff, in seconds
merge gate	`ci`	may this land? — the whole tree, the pipeline's own gate

The gate	The command	The decision it makes
deployment gate	`deployment-gate env=local`	would this deploy, and did it?
smoke test	`smoke env=staging`	does the deployed system do the one thing it exists for?
release gate	`orch release tag=v1.4.0`	may this become a version?

Two naming rules follow. **Commands name actions**: `pre-commit` is what you do, `deployment-gate` is what runs. **Environments are arguments**: the same command judges staging and production, and the argument says which.

The Four Moments

A **gate moment** is when in a change's life a gate fires. Each moment carries a budget — the most time it may take before it stops helping — and the budget grows as the change moves later:

```
flowchart LR
    A["Atomic<br/>ms–s"] --> B["Commit<br/>seconds"]
    B --> C["Full<br/>minutes"]
    C --> D["Scheduled<br/>hours"]
```

What this shows: the journey a change makes. The arrow is time, not a dependency between checks.

Moment	The question it answers	Fits
Atomic	is this one thing right, right now?	lint, types, formatting
Commit	can the author still fix this cheaply?	the commit gate, unit tests on changed files
Full	does the whole change hold together?	the merge gate, integration tests
Scheduled	is anything rotting that a change would not reveal?	dependency audits, clean-checkout builds

A moment is *when*, never *where*. *Staging* is a place a build runs; *full* is the moment a change is judged as a whole. Keeping the two apart is what lets the same gate run in three environments without its meaning changing.

Your CI has stages; this book has moments, and the difference is worth thirty seconds. A **stage** is a phase a pipeline groups jobs into, and the jobs inside it may run in parallel. A **moment** is when a gate fires. A pipeline has stages whether or not it has gates, and a gate has a moment whether or not a pipeline runs it. The two are easy to merge in conversation, and merging them is how a scheduled job comes to be treated as a merge gate.

Most engineers arrive with their own names for these moments. They are the same four, and this table is the translation:

Your CI calls it	This book calls it	The moment is	A gate there answers
pre-commit hook, *pre-push*, *on save*	**atomic**	any time, one concern, milliseconds	is this one file right, right now?
commit hook, *staged check*	**commit**	as the change is recorded	can the author still fix it cheaply?
`ci` *job*, *the PR check*, *the* ***merge*** *gate*	**full**	the whole change is complete	does it all hold together?
nightly, *cron*, *scheduled workflow*	**scheduled**	the clock decides	is anything rotting that a change would not reveal?

Two rules of placement keep a moment meaningful. A check lives at exactly one moment: two copies will drift apart, and when they disagree nobody knows which one is authoritative. And a moment is not a place: moving a check to a different environment changes *where* it runs, never *when* it fires.

"Release" is an event, not a moment. A moment is *when*; a release is something a person decides, and what runs at it is the release gate in an environment. Keeping the word for the event separate from the word for the moment is what stops a release checklist from being mistaken for a gate.

✅ What to Check

- For every check, name its moment. One moment per check; two is a drift waiting to happen.
- Read your own prose: every use of "gate" without a name from the five is a sentence that will not survive a handover.
- Time the commit moment. Over a minute means something is sitting where it does not fit.

4. The Workflow, and the Gates Between Its Stages

A change does not pass four moments in the abstract. It passes **stages of work**, and each gate sits between two of them:

```
flowchart LR
    A["edit<br/>a file"] --> G1{"atomic"}
    G1 --> B["record<br/>a commit"]
    B --> G2{"commit"}
    G2 --> C["integrate<br/>a branch"]
    C --> G3{"full"}
    G3 --> D["operate<br/>a host"]
    D --> G4{"scheduled"}
```

What this shows: ***the box is work; the diamond is a gate.*** *The work arrives at a stage, and a decision stands between it and the next stage — which is why a gate is a transition and a stage is a place. The arrow is the change moving, not a dependency between checks.*

The four moments are positions in that flow, not a timetable. And the gates a change actually meets are five, because two of them judge a deployment and a release — levels of their own, not moments:

# The gate	Fires at	Sits between	What has to be true when it passes
1 **commit gate**	the **commit** moment	a file and the repository	the staged change is sound — it may be committed
2 **merge gate**	the **full** moment	a branch and the trunk	the whole tree holds — it may be pushed
3 **deployment gate**	*its own level* — it judges a deployment, not a change	the trunk and a host	this would deploy — it may go to a host
4 **smoke test**	*the deployment level, after it*	a host and a reader	the address serves a reader, and the one thing the system exists for works
5 **release gate**	*an event*	an estate and a version	it may become a version — clean tree, gates green, tag pushed

Read the table as a sequence, not a menu. Each gate answers a question the next one assumes: a release is safe because the smoke test passed, which is possible because the deployment gate produced a deployable artefact, which depended on the merge gate's verdict, which began with a commit that passed its own gate. **Skip a row and the next row is a guess.**

Which Test Belongs at Which Gate

A reader's suite is not a pile of checks; it is a set of tests at different **levels**, and the level decides the gate. The rule is one sentence: **a test belongs to the earliest gate whose environment contains everything the test needs to see.** Earlier is cheaper; later is the only place some things are visible at all.

Test level	What it can possibly prove	Its gate	Why there, and not earlier
static analysis · lint · types	the code is well-formed	**commit (atomic)**	milliseconds, and it needs only the file
unit	one function is right	**commit**	cheap enough to run constantly; it cannot see a database
contract	two components still agree	**merge (full)**	needs both sides present
integration	components work together	**merge (full)**	needs the real dependencies
system / end-to-end	the deployed system does the thing	**smoke (deployment level)**	needs a running environment; a file cannot show it
dependency audit · migration reversibility · secret scan	nothing is rotting	**scheduled**	the answer changes with the calendar, not with the diff
acceptance · a person's judgement	a human says yes	**release**	a person cannot be in the pipeline

The test you cannot place is the finding. A check that fits no row is either measuring something no gate needs — delete it — or it is measuring something real that nothing is positioned to act on, which is a hole in the workflow, not a spare test.

✅ What to Check

- Can you draw your own flow — the stages work actually passes through, and the gate between each pair?
- Take each test in your suite and name the gate it belongs to. **A test you cannot place is a finding.**
- Does any gate judge an environment it was not given? A merge gate that reads a deployed address is measuring the wrong stage.
- Is any stage ungated — work arriving somewhere that nothing judges?

---

5. The Deployment Gate: the Level That Fails Your Pushes

A push that goes red in CI is usually not a defect in the code. It is a defect in the **path** — the workflow that runs, the tool that changed, the artefact that was built differently from the one that was published, the route a reader takes that nobody exercised.

A gate that reads the code cannot see any of that. The deployment gate can, and it is the level that pays for itself fastest.

The failure you meet	Can a code gate see it?	The check that catches it
the workflow no longer parses, or names a job that no longer exists	no	a workflow-schema and job-graph check
the platform tool now requires a flag it never required	no	a local run of the same command the deploy runs
a route answers 404 after the switch	no	a check that the route exists before the URL moves
the build produced one directory and the deploy published another	no	a check comparing what is built with what is published
a reader typing the site's own address gets an error page	no	a smoke test of the journey a person takes

The Local Tier, and Its Ceiling

The deployment gate runs in three environments, and the cheapest one proves the most per second.

Question	Local (`ENVIRONMENT=local`)	Staging / production	What local costs
does the workflow parse, and does the job graph still name the jobs?	✅	✅	seconds
does the strict build produce the deployable artefact?	✅	✅	a minute
does the built site carry what the deploy would upload?	✅	✅	included in the build
do the worker bindings and the Worker bundle?	✅	✅	seconds
does the generated audio match the page it narrates?	✅	✅	**tens of minutes per chapter**
do permissions, secrets, DNS and TLS work?	❌	✅	—
does the real address serve a reader?	❌	✅ (the smoke test)	—

The last cost is the one to plan around. A local text-to-speech render is free, private, and takes longer than everything else in the gate combined — so the check that a page edit has not invalidated its narration is worth having, and the re-render it asks for is worth scheduling rather than doing on every save. **A gate is not free because it runs on your machine.**

That table is also the honest edge. **Local proves *would this deploy*; the deployed environment proves *it is deployed and serving*.** A team that runs only the first has a fast gate and an unverified address.

The Command

```
deployment-gate env=local          # your wrapper — make, npm run, task — is your project's choice
```

The gate replays the path: the deploy-path checks (workflow schema, job graph, worker bundle, and the local stack where one exists), the strict build, and the comparison between what was built and what would be published.

```
deployment-gate env=staging        # the same artefact, on a real host
smoke env=staging                  # is the deployed address working?
```

Where the Loop Finishes

Validation starts local: fast, free, private, and it proves what a build would do. It stops being enough when the question is about the destination — permissions, secrets, DNS, TLS, the address a reader reaches.

So the loop moves to an environment **closer to that destination**: the same deploy path, the same configuration surface, the same kind of address. That is what makes the iteration accurate rather than approximate, and it is the reason to keep such an environment at all.

Four properties keep that environment cheap to **destroy and rebuild**, and they are the same four that make it useful:

- **defined in the repository** — infrastructure, configuration and seed data as files, so an environment can be created again from nothing;
- **a dataset that can be reloaded** — synthetic, or a scrubbed copy, so the database can be thrown away;
- **one command to create and one to destroy** — the same commands the pipeline uses, not a runbook;
- **nothing critical depends on it** — no customer data, no resource production shares.

An environment with those four can be discarded when a run damages it and rebuilt inside the time the loop can afford. Because the interruption is cheap, the loop can run there unattended — **an agent may converge on this environment; production stays a person's decision.**

What a Smoke Test Proves — and What It Must Do When It Fails

The smoke test is the only gate that judges the system a reader actually reaches, so its scope is the reason it exists. **The address answering is its precondition, not its verdict.** A static page answers `200` with the application behind it dead: the load balancer is healthy, the HTML renders, the status code is correct, and the interaction the estate sells returns a failure. A smoke test that checks only the address certifies exactly the state it was built to detect.

So it exercises **journeys**, not URLs — the three or four interactions the business exists on. It asserts a **contract**, not a status code: that a query returns documents, that a request returns the response shape its clients parse, that a sign-in yields a session. It is read-only, deterministic and finished inside a minute, because a gate that mutates data is a second incident and a gate that takes five minutes is one people start skipping. It does not depend on anything the estate does not control — an AI model provider, a third-party widget — or it reports that dependency's health as if it were the estate's.

Its failure has an edge, and the edge is the point. A smoke test that fails describes a deployment that is already live and already wrong, so the sequence stops: the release gate does not run, and the environment returns to the version that last passed this gate — automatically, not after a discussion. That is why the rollback is rehearsed (§18, §25): a rollback nobody has executed is a plan, and the moment it is needed is the worst moment to learn it does not work. A smoke test whose failure has no consequence is not a gate. It is a dashboard with a status code.

The edge has one honest exception, and stating it is part of the design: a change that cannot be reversed has no rollback to take. A migration that dropped a column, a provider that was cut over, a schema that was rewritten — for those, the failure edge is a **roll-forward path**, and the thing to prepare before the deploy is the fix, not the return. The gate is not weaker for saying so; it is weaker when a team discovers it during the incident, so the irreversible changes are named in the gate's own record and the roll-forward path is rehearsed like any other.

The edge also has a precondition, and it is the one most easily left implicit. A rollback is a return to something that still exists: an artefact the platform kept, a schema that accepts the older code, a copy of the data that has been **restored at least once**. So the deployment gate declares its way back *before* it runs, and the declaration names which layer it returns through — the artefact, the schema, or the data. A copy nobody has restored is a hypothesis, not a recovery (§18).

The three layers have different costs, and a design that has only the first is the common one: the artefact rolls back in minutes, the schema rolls back only if the migration was additive, and the data does not roll back at all unless someone rehearsed it.

The Irreversible Release, as a Named Class

Some changes have no way back, and the estate is stronger when it says so than when it pretends otherwise. An **irreversible change** — a dropped column, a truncated table, a provider cut over — is not an exception to the gate; it is a class with a procedure, and the procedure has three facts that must be true before it runs. Without them, "declared" means "someone wrote a reason", which is how a ban becomes a formality.

The fact	What it proves	Where it is recorded
The observation window has elapsed	the previous phase has been live long enough that the data being removed is no longer needed	the change's own record
The data is archived, and the archive has been restored once	"we copied it" is a hypothesis until a drill has read it back (§18)	the drill's output
The decision is forward-only, and owned	after this deploy there is no return, so the decision is made before, by a named person	the signature on the change

Those three facts are what §15's ban means by *"declared, reviewed and owned"*: declared is the window and the archive, reviewed is the drill, owned is the signature. The gate refuses destruction; the procedure is how destruction is allowed.

Order carries the safety, and it is the reason the class is safe. Destructive work belongs to the last phase of expand → migrate → contract (§25), one deploy per phase, and never in the same deploy as the code that stops using the old shape. The window exists so that the rollback stays available while the new shape proves itself; shortening it by one release converts a reversible change into an unrehearsed one.

✅ What to Check

- Name the last three deploy failures you had. How many would a code gate have caught?
- Does your local build produce **the same shape** as your deploy build? If not, the local gate is measuring something else.
- For each deploy-path check you cannot run locally, write the reason down. An unstated reason becomes an assumption.
- **List the journeys your smoke test exercises.** If it names a URL rather than an interaction, it is a liveness probe, not a smoke test — and it will pass while the product is down.
- **Say what happens when it fails.** Name the command that restores the previous version, and the last time you ran it. If either answer is missing, the sequence has no edge.
- **Name your last irreversible change.** State its window, its archive's restore date and its owner. All three, or the change was not declared.

---

6. How the Gates Compose

Three Words First

Word	What it means	What it is not
entry point	a command a person types to run "the checks" — `ci`, `test`, `validate`	not the checks themselves
composition	which checks run under which command, declared in one place	not a habit, not a README
slice	a command that runs **part** of a gate, on purpose	not the whole gate, and never named as if it were

The Seven Rules

Each rule is written in four beats: the practice, what it gives you, where it lives, and the signature of its absence.

1 · One name means one thing. *It gives you:* a verdict that travels — "it passed" means the same thing in every repository, so a handover keeps its meaning. *Where it lives:* the naming rule every repository reads. *Its absence looks like:* `ci` runs four checks in one repository and seven in another, and both are called green.

2 · A slice says what it skips. *It gives you:* a fast command you can trust, because it tells you what it left out before it tells you the verdict. *Where it lives:* the slice's own output — before the verdict, not in a document beside it. *Its absence looks like:* `validate` skips the tests, and the person who ran it believes it did not.

3 · The same command locally and in CI. *It gives you:* a local pass that predicts a pipeline pass, which is the whole reason a developer runs anything before pushing. *Where it lives:* the one composition file both the pipeline and the developer invoke. *Its absence looks like:* a deploy fails on lint, in files the change never touched, because the author ran the fast slice.

4 · A gate proves it ran. *It gives you:* a verdict with a count and an artefact behind it, so the next reader can audit it and a silent skip cannot pass as success. *Where it lives:* the gate's own output and its log — the numbers, not the adjective. *Its absence looks like:* "passed" with no number, or a check that matched zero rows and reported success.

5 · One command is the whole gate. *It gives you:* one word that answers *"did you run the checks?"*, and every other command visibly a subset of it. *Where it lives:* the single composition file that defines the whole gate as the union of its named parts. *Its absence looks like:* four entry points with overlapping subsets — `ci` runs everything, `check` runs two, `test` runs lint, `security` runs validate — and each one believes it is the gate.

6 · A failure names the fix. *It gives you:* a failure line that sends its reader to the file, the line and the reason, instead of to a log. *Where it lives:* the failure line itself, in the same shape everywhere. *Its absence looks like:* exit 1 and nothing else, so the next reader greps.

7 · Debt is declared, printed and expiring. *It gives you:* a check that cannot exist yet, recorded as a decision with a reason and a date — and a registry that prints itself on every run. *Where it lives:* an exemption registry, regenerated with the gate. *Its absence looks like:* a skipped check nobody recorded, discovered during an incident.

Eight Questions Before the Next Gate

#	The question	A thin answer looks like
1	What invariant does this keep true?	"it catches bugs"
2	Which moment can afford it?	"we'll put it in CI"
3	What exactly does it read, and what does it ignore?	nobody has looked at the ignore list
4	What does it say when it cannot read?	a colour
5	What would make us delete it?	silence
6	Who owns it when it breaks?	"whoever wrote it"
7	Does an existing gate already cover this?	not asked
8	What does a false positive cost here?	not measured

Question 5 is the one that keeps an estate from only growing. A gate nobody can imagine deleting is either essential or forgotten, and the question tells you which.

✅ What to Check

- Pick the rule your estate most needs; make it true before adding the next gate.
- For each rule, name the file where it is enforced.
- Count the entry points that claim to run "the checks". One definition is the goal; two is the next piece of work.

---

7. What the Set Pays Back

The chapter asks for a set of decisions, and every one of them pays on the day it is needed. The table states the payment, not the bill.

The practice	What it gives you	The moment it pays
the five declarations (§2)	a gate whose name and whose body agree, so a reader can act on its verdict	when a deploy fails and the first question is *what did this actually check?*
the third verdict	a gate that says *not verified* instead of passing on something it could not read	when the service is down and the check cannot reach it
one name, one meaning	"it passed" that survives a handover	when someone new asks what to run before merging
declared exemptions	skips you know about, with a reason and a date, instead of skips you discover	when an incident review asks why a class of defect was never caught
liveness	every gate has a crafted case that turns it red, so a silent skip cannot impersonate a pass	the first time someone changes a gate and assumes it still works
the deploy-path level	failures found before a push, on the path rather than in the code	the ten minutes you would otherwise spend reading a CI log

The pattern in every row is the same: **the estate either tells you the truth or it tells you nothing.** A gate that cannot fail does not warn you — it reports success, which is the one thing you cannot recover from.

✅ What to Check

- Take the last three incidents in your estate. Which row of that table would have changed the outcome?
- Ask of each gate: if it were deleted tomorrow, what would break, and who would notice?

---

8. How It Is Implemented

The chapter is advice until it is a file. These are the files.

1 · The composition file — one definition of the whole gate:

```
# the merge gate: the union of its named parts, in one place
ci: validate lint test security-scan
    @echo "ci: 4 checks — the same four the pipeline runs"
```

2 · The exemption registry — every skipped check, with its reason, printed on every run:

```
# config/gate-exemptions.yaml — debt that expires, never silence
exemptions:
  ci: a composition of validate + lint + test + security, each proven separately
```

3 · The liveness proof — for each gate, one crafted case that makes it fail:

```
test-<gate>                # plants the defect that gate names, asserts the refusal
```

4 · The generated table — the audit, as an artefact rather than a meeting:

```
gate-estate                # → a generated table: gate · moment · what it promises · proven by
```

The generator reads three sources that already exist — the **command surface** (the Makefile), the **pipeline** (the workflow file), and the **gate scripts** — so the table cannot disagree with them. A gate that appears in one and not the others is the finding.

The first run of an audit like this is usually instructive rather than comfortable. A gate estate that has grown for a few years will have gates whose moment is –, gates whose failing case is missing, and at least one gate nobody can say the purpose of. That is not a failure of the audit; it is the audit doing its job, and every row it names is a five-minute decision.

✅ What to Check

- Does a new reader find the whole gate in **one file**? Open it and look.
- Does every gate name the case that fails it? If not, which exemption is missing?
- Does the table regenerate, or is it a document that was true the day it was written?

---

9. Four Anti-Patterns the Design Prevents

An **anti-pattern** is a way of building that looks reasonable locally and fails systematically. You do not recognise it from a single instance — you recognise it from the shape it leaves across an estate.

Anti-pattern	What it looks like	The design that prevents it
Many entry points, one promise	several commands claim to run "the checks"; people pick the cheapest	one name per decision, and every other command a declared subset
The gate that reads its own output	it reports defects in generated files no change can fix	scope drawn around what the change owns
The context-sensitive gate	it does more in CI than locally, and nobody knows which ran	one behaviour; scope declared, never inferred from the environment
The copy-paste gate	the same gate in a dozen repositories as a dozen slightly different bodies	one standard implementation, and a conformance check between them

Two further shapes belong to Chapter 2 §10 rather than here: *the gate that cannot fail*, and *the gate that fails on everything*. Both are alive in most estates, and both are cheap to fix once named.

Common Mistakes to Avoid

1. Adding a check without naming its invariant and its moment.
2. Letting a name promise more than the body runs.
3. Reading a build product and calling it the source.
4. Copying a gate into a new repository without a conformance check.

✅ What to Check

- Pick the anti-pattern you recognise in your own estate, and name the **design change** that would have prevented it.
- For each of the four, ask whether one of §6's rules would have stopped it. If none would, the rule set has a gap.

---

10. Build the Workbench

Artefacts

Artefact	What it is for	Where it lives
deployment gate	*would this deploy?* — the path, checked locally	`deployment-gate env=local`
gate table	every gate: its name, moment, promise and proof	`docs/gates.md`, generated
composition file	the one definition of the whole gate	`Makefile` or the CI workflow
exemption registry	every skipped check, with reason and expiry	`config/gate-exemptions.yaml`
liveness record	for each gate, the case that fails it	the gate's own self-test

Decisions

Choice	Default	What changes if you choose otherwise
one composition file	yes	several definitions of "everything", which is the defect
moment per check	exactly one	two copies that drift
verdict states	pass · fail · not-verified	"not run" read as "passed"
exemptions	declared, printed, expiring	silent skips discovered during incidents
the table	generated	a document true the day it was written

Sequence

1. **Inventory** — list the commands and what each runs. *Checkpoint: one table, one line per repository.*
2. **Name** — the five names; one command per decision. *Checkpoint: `ci` means one thing everywhere.*
3. **Place** — one moment per check; move the misfits. *Checkpoint: the commit moment is under a minute.*
4. **Prove** — a failing case for each gate. *Checkpoint: a planted defect turns it red.*
5. **Watch the path** — the deployment gate, local first. *Checkpoint: a planted workflow error fails locally.*
6. **Generate** — build the table from the repository. *Checkpoint: the audit runs in the gate.*

Done When

- One command is the whole gate, and every other command is a visible subset of it.
- Every gate can name the defect it would catch.
- A planted defect in the deploy path fails locally, before a push.
- The table is generated, and a gate with no failing case fails the run.

---

11. 🔑 Chapter Summary

The gates a team has are an accident; the gates a team designs are an asset.

1. **Checks accumulate** because each one is reasonable and the set is never reviewed. The estate is the set, and a set can be designed.
2. **A gate is a contract**: precondition, invariant, scope, verdict, cost. Scope and precondition are the two whose absence produces a confident, meaningless green.
3. **A gate has a name, a moment and an environment** — five names (commit · merge · deployment · smoke · release), four moments (atomic · commit · full · scheduled), and the environment as an argument.
4. **The deployment gate is the level that catches the path.** Local proves *would this deploy*; the deployed environment proves *it is deployed and serving*.
5. **Seven rules compose an estate**: one name one meaning · a slice says what it skips · local equals CI · a gate proves it ran · one command is the whole gate · a failure names the fix · debt is declared, printed and expiring.
6. **Silence is the failure mode to design against**: an unearned green, a scope assumed, a skip unrecorded, a gate that cannot fail.
7. **Four artefacts make it real**: the composition file, the exemption registry, a failing case per gate, the generated table.
8. **Four anti-patterns belong to composition** — many entry points, reading its own output, context sensitivity, copy-paste.

   The orchestrator's test: ask which command runs everything. If the answer is one word, and it means the same thing in every repository, the estate has a definition. If the answer takes a sentence, it has a tradition.

Chapter 20: From One System to a Portfolio

The Manifest as the Answer

Chapter 19 closed the gate estate for one system; this chapter asks the question only a portfolio can ask — what is the state of the organisation?

---

1. What Changes at Scale

The three structures worked for one system. A portfolio changes the question: not *is this system correct?* but **what is the state of the organisation?** — and that question has exactly one good answer: read a file, run a check.

Everything else is a conversation with whoever remembers.

🎯 What You'll Learn

- Why rules become registries, procedures become rollouts, gates become policy
- What the manifest is for — and what it must never become
- The portfolio map: owner, repository, lifecycle

Four Things That Stop Scaling

At one system	At a portfolio	What replaces it
a rule in a document	a rule nobody can apply twice the same way	a **registry**
a procedure a person follows	a procedure with ten variants	a **rollout**
a check in one pipeline	a check applied to some repositories	**policy**
a mental list of what exists	a list that is wrong	the **manifest**

None of these is a new tool. They are the same facts, moved from memory into a file, and made checkable.

```
flowchart LR
  M["portfolio.yaml<br/>the manifest"] --> R["registries<br/>derived, never edited"]
  M --> V{"validator<br/>in the gate"}
```

```
  V -->|"passes"| S["the portfolio's state<br/>one command"]
  V -->|"fails"| F["a row with no owner<br/>or an unknown tier"]
```

What this shows: one file, one derivation, one check — the only shape in which the answer is the same for everyone.

The Manifest

One file, one row per repository, and every row verifiable:

```yaml
# portfolio.yaml — one row per repository
repositories:
  - name: <repo>
    owner: <who is accountable>
    lifecycle: active | deprecated | retired
    tier: <what it is allowed to break>
    gate: <the command that proves its rows>
```

Cognitive load is the real budget. A human cannot hold twenty repositories, their owners and their states. Neither can an agent — and unlike the human, the agent will not say *"I am not sure"*. It will read whatever it finds and act on it.

The Thinnest Viable Platform is the canon term for the right size: the smallest thing that answers the portfolio question, which is legitimately a curated file rather than software.

✅ What to Check

- Can you answer "what exists, who owns it, what state is it in?" from one file?
- Name a rule applied by memory in two repositories. That is the next registry row.

One fact changes shape at this scale, and it is not in the manifest: **a restore replaces the project, not one schema.** The suite shares one database, so recovering one service's data puts an older copy in front of every service that shares it — the blast radius of a data recovery is the project. That makes it a decision with a written case (chapter 25), never a repair command run by whoever noticed first.

2. The Source-of-Truth Model

A file becomes the source of truth when **a command reads it and a check fails when it is wrong**. Three properties:

1. **Every row is checkable** — a named owner, a known tier, a real repository.
2. **Nothing else duplicates it** — a second list (a wiki, a console) is a second truth.
3. **The gate reads it** — the check is what makes the file authoritative.

Without the third property it is documentation; with it, it is state.

The Registers, Derived

Registry	What it holds	Derived from
repositories	name · owner · lifecycle · tier	the manifest
ports	what is exposed, and where	the manifest + the services
surfaces	the public entry points	the manifest
environment variables	the names each service needs	the manifest

Derived, not maintained. A registry edited by hand drifts from the manifest within a month — and the drift is invisible until a deploy fails on a variable nobody declared.

✅ What to Check

- Is any registry hand-maintained? That is the first drift.
- Does the gate read the manifest, or is the manifest a summary of the gate?

---

3. The Lifecycle Table

A lifecycle is a permission model: what a repository may do depends on where it is.

State	Permits	Forbids
experimental	anything; no guarantees	public exposure
active	production traffic, on-call	unowned dependencies
deprecated	fixes, security patches	new features
retired	nothing; archived	deploys
planned	scaffolding, no traffic	production exposure

The state is a row in the manifest, and the gate checks that a repository's behaviour matches it — a retired repository with a deploy pipeline is a finding, not an oversight.

✅ What to Check

- Does every repository have a lifecycle state?
- Is any retired repository still deployable? That is the gate's job.

---

4. The AI Delta

The manifest is an **instruction to the agent**, not a description for humans. A row that is missing is a decision the agent will make for you, confidently, and nobody will be able to check it afterwards.

An agent that can enumerate the estate will act on whatever the file says, at machine speed. That is why the file must be checkable by a command rather than readable by a person — a human notices an omission; an agent fills it in.

> 💰 **Cost Check**: a portfolio maintained by memory costs a question per repository per decision. A manifest costs the first afternoon and then pays on every question after it. The expensive failure is the row nobody wrote: the agent that adds a repository to the wrong tier, or the deploy that runs because a retired service still had a pipeline.

The Delegation Line

The agent reads the manifest, proposes rows, and derives the registries. **The human** owns the lifecycle and the tier — the two fields an agent must not set for itself. **The system** refuses a row without an owner, and refuses a registry without a schema.

The three subjects are different on purpose. An agent may *propose* that a repository is active; only a human may *declare* it.

---

5. Build the Delivery

Deliverables

Artefact	Shape	Where it lives
portfolio manifest	one row per repository	`portfolio.yaml`
registry skeletons	one file per registry, stable ids	`registry/*.yaml`
lifecycle table	five states and their permissions	the manifest's documentation

Decisions

Choice	Default	What changes if you choose otherwise
where the source lives	one file, reviewed	per-repo fragments, which no single read can answer
what a row must carry	id · owner · lifecycle · tier	rows that cannot be checked
registries	derived, never edited	a second list that drifts
the lifecycle check	the gate	a retired service that still deploys

✅ Gates

- The manifest validator refuses an **unknown tier or a missing owner**.
- The registry schema refuses a **row without a stable id**.
- A retired repository with a **deploy pipeline** is a finding.

Runbook

1. **Write the manifest** for the repositories that exist. *Checkpoint: every row has an owner and a tier.*
2. **Derive the registries** from it. *Checkpoint: no registry row exists that the manifest does not imply.*
3. **Wire the validator** into the gate. *Checkpoint: a planted bad row fails.*
4. **Add the lifecycle column** and the permissions it implies. *Checkpoint: a retired deploy is a finding.*

🎯 Proof

- A manifest that fails on a planted error.
- A portfolio question answered by one command.
- No registry maintained by hand twice.

---

6. 🔑 Chapter Summary

The portfolio question has one good answer: read the file, run the check.

1. **Rules become registries, procedures become rollouts, gates become policy** — the same facts, in files.
2. **The manifest is the answer**, and it is authoritative only when a command reads it and a check fails.
3. **Registries are derived**, never hand-maintained — a second list is a second truth.
4. **Lifecycle is a permission model**, and the gate enforces it.
5. **The agent reads the manifest as instruction** — which is why an owner and a tier are a human's to set.

Chapter 21: Version Control as the Substrate

Everything That Matters Is a Versioned File

The portfolio needs one place where its facts live; this chapter makes the repository that place, and the tag the boundary of a release.

---

1. The Repository Is the Unit of Governance

The three structures hold because they are versioned. A rule that lives in a repository can be diffed, reviewed, rolled back and attributed; the same rule in a console cannot be any of those things. That is not a style preference — it is the difference between governance and memory.

The portfolio chapter made the state a file. This chapter makes the **history** of that file the substrate: every change reviewable, every release a boundary, every author attributable.

🎯 What You'll Learn

- Declared versus generated: what must be a reviewable file, and what must never be
- Why provenance becomes a review input, not a footnote
- The release boundary: a tag, a signature, and what each actually proves

Declared and Generated

Declared — must be a versioned file	**Generated — must never be committed**
rules, manifests, registries	build artefacts
policy, contracts, schemas	caches and indexes
migrations	derived state
the evidence pack	secrets

The rule is one line: **if it can be rebuilt, it is generated; if it must be reviewed, it is declared.** A generated file committed by accident is a file that will be edited by hand once, diverge, and never be noticed.

✅ What to Check

- Name one generated file that is committed. That is the first path to classify.
- Name one declared fact that lives only in a console.

---

2. Conventions That Make History Readable

A history is legible when it follows a shape everyone can predict:

Convention	**Why it exists**
short-lived branches, trunk-based	the change is small enough to review and revert
one logical change per commit	bisect finds the defect, not the batch
a subject that states the change	the log is readable without opening the diff
tags only at the release boundary	a tag means *this shipped*, not *this was tested*

The tag is the release boundary, and it should be the only thing that means it. A tag created for convenience destroys the one signal an operator relies on.

Signatures. A signed commit proves the bytes came from the key that signed them. It does not prove the change is correct, reviewed, or intended. That distinction matters because a signature is often cited as if it did.

The Provenance Rule

An agent-authored commit carries the agent's identity, and its review is named:

```
Author: <agent identity>
Co-authored-by / provenance: <agent name and version>
Reviewed-by: <human who approved the change>
```

The point is not ceremony. It is that a reviewer can answer *who wrote this, and who accepted it* without reconstructing the session.

✅ What to Check

- Does a tag mean “shipped”, or does it mean several things?
- Can you name the author and the reviewer for the last ten commits?

---

3. History as the First Diagnostic

`log`, `blame`, `bisect`. Each answers a different question:

Command	The question	The discipline it needs
`log`	what changed, and in what order	a readable subject
`blame`	who last touched this line, and why	an attributable author
`bisect`	which change introduced the defect	one logical change per commit

The archaeology rule: reconstruct *why*, not only *what*. A bisect that finds a commit still requires reading the message and the review to understand the intent — and an agent-authored commit that says only *“update”* leaves the archaeology unfinished.

✅ What to Check

- Run a bisect on a recent defect. Does it land on one change or a batch?
- Does the commit message state why, or only what?

---

4. The AI Delta

Provenance stops being a footnote and becomes a **review input**: an agent’s commit is attributable to *its* identity, never a borrowed human one, and a signed tag becomes a human’s signature on a commit an agent wrote.

History is also the agent’s cheapest context — and its most dangerous, because a bisect it cannot read is a regression it cannot explain.

> 💰 **Cost Check**: an unattributable commit costs an afternoon of archaeology; a tag that means several things costs a release nobody can audit; a generated file committed by hand costs the divergence you find a month later. The conventions cost a file and a review habit.

```
flowchart LR
  C["commit — attributable"] --> B["branch — short-lived"]
  B --> T["tag — the release boundary"]
  T --> S["signed, and reviewed"]
  S --> A["auditable history"]
  C -.->|"unattributable"| X["archaeology"]
```

What this shows: the chain from a commit to an auditable release — and the single broken link that costs the audit.

The Delegation Line

The agent writes commits, assembles the changelog and runs the bisect. **The human** signs the tag and reviews the provenance. **The system** refuses an unsigned tag, a generated file committed by hand, and a commit with no attributable author.

---

5. Build the Delivery

Deliverables

Artefact	Shape	Where it lives
declared/generated table	two columns, every path classified	`docs/repository.yaml`
conventions	prefixes, subjects, when a tag may exist	`CONTRIBUTING.md`
provenance rule	how an agent commit is labelled and reviewed	the conventions file
release-boundary check	refuses an unsigned tag or a tag off the default branch	the gate

Decisions

Choice	Default	What changes if you choose otherwise
branching	trunk + short-lived	long-lived branches, and merges nobody can read
signing	tags, because the release boundary is what a consumer trusts	commits too, if the audit demands it
provenance	a trailer naming the agent	a diff with no author

Choice	Default	What changes if you choose otherwise
tag meaning	*this shipped*	a tag that means whatever the last person meant

✅ Gates

- A **generated file committed by hand** is refused by diff.
- A **tag whose commit is unsigned** (or off the default branch) is refused before publish.
- A commit **without an attributable author** is a finding.

Runbook

1. **Classify every path** declared or generated. *Checkpoint: no path is in both columns.*
2. **Write the conventions** file. *Checkpoint: a tag's meaning is one line.*
3. **Add the generated-file gate.** *Checkpoint: a planted generated file fails.*
4. **Enable tag signing** at the boundary. *Checkpoint: an unsigned tag cannot publish.*
5. **Rehearse a bisect** on a real regression. *Checkpoint: it lands on one change.*

🎯 Proof

- A history where every change is attributable.
- A tag that names exactly what shipped.
- A bisect that finds a planted regression in one sitting.

---

6. 🔑 Chapter Summary

Everything that matters is a versioned file, and its history is the evidence.

1. **Declared or generated** — if it can be rebuilt it is generated; if it must be reviewed it is declared.
2. **Conventions make history readable**: short branches, one change per commit, a subject that states the change.
3. **The tag is the release boundary**, and a signature proves authorship, not correctness.
4. **History is a diagnostic** — log, blame, bisect — when the commits are small and attributable.
5. **The agent's commits are attributable; the human signs the tag.**

Chapter 22: The Control Plane as a Product

Validators, Audits, and the Golden Path

Once the facts live in files, something has to serve them: this chapter turns the control plane into a product whose second user is a machine.

---

1. A Platform Has Users

The three structures need a place to declare their state. That place is the control plane: one manifest, its registries, and the machinery that keeps them true. It becomes a product the moment it has **users** — and those users are humans and agents, with different tolerances.

A human tolerates a README and a console. An agent tolerates neither: it needs everything machine-readable, idempotent, and evidence-producing, or it will do something else.

🎯 What You'll Learn

- The difference between a validator and an audit, and why conflating them lies
- What a service catalog must be generated from
- The golden path as the platform's only reliable output

Validators and Audits

These are two different instruments, and using one where the other belongs is how a platform lies.

	Validator	Audit
reads	the declared state	the live state
runs	in the gate, on every change	on demand, on a schedule
answers	*is the declaration well-formed?*	*does the world match the declaration?*
fails	the change	the drift
lies when	it is mistaken for an audit	it is mistaken for a validator

A validator that passes tells you the file is coherent. It tells you nothing about production. An audit that passes today says nothing about the change you are about to merge.

✅ What to Check

- Is any *"green"* in your platform a validator being read as an audit?
- Does the audit run on a schedule, or only when someone remembers?

---

2. The Catalog Is Generated

A service catalog has five facts per service: owner, lifecycle, SLO, signal, runbook. Five facts across twenty services is a hundred chances to drift — and the catalog is the page everyone reads.

So it is generated from the manifest, never maintained beside it.

Fact	The source
owner	the manifest
lifecycle	the manifest
SLO	the signal definitions
signal	the evidence layer
runbook	the repository's path

A hand-maintained catalog is a second source of truth with a nice UI — which makes it more dangerous than a stale file, because it is more trusted.

```
flowchart LR
  M["manifest"] --> C["catalog<br/>generated"]
  S["signal definitions"] --> C
  R["runbook paths"] --> C
  C --> H["human reads it"]
  C --> A["agent reads it"]
```

What this shows: the catalog has no independent input — every fact comes from the thing that owns it.

✅ What to Check

- Is any catalog row edited by hand? That row will be wrong first.
- If the catalog were deleted, could it be regenerated without a meeting?

---

3. The Golden Path

A golden path is the **one command** that creates a new service or repository *with its gates already wired*. Not a checklist, not a template repository, not a wiki page — a command that an agent can run twice without creating two services.

What a scaffold must produce:

- the repository, or the files in an existing one;
- its gates, wired and passing;
- its manifest row, added;
- its contract skeleton, in place;
- the runbook entry, generated.

Idempotent matters because an agent will run it again. A scaffold that creates a second service is a scaffold that teaches the agent to stop using it.

Scope discipline: the platform states what it governs. Mirrors, forks and vendored copies are not the organisation's content; a control plane that tries to govern them fails on the first fork and never recovers.

✅ What to Check

- Run the scaffold twice. Does it create one service or two?
- Does the platform state what it does *not* govern?

---

4. The AI Delta

The platform's **second user is a machine**: every operation must be idempotent, because an agent will run the scaffold twice, and machine-readable, because a console-only step is invisible to it.

The golden path becomes the agent's only reliable route to a correct change — which makes it the highest-leverage artefact in the book.

💰 **Cost Check**: a hand-maintained catalog costs a wrong row in every planning conversation; a non-idempotent scaffold costs a duplicate service and a cleanup nobody scheduled; a console-only path costs an agent that cannot follow it and invents its own. The platform's cost is a real day of work, and its return is every question it answers afterwards.

The Delegation Line

The agent scaffolds, audits and reports drift. **The human** approves the scope and the catalog. **The system** refuses a scaffold outside the manifest, and a second run that would create a second service.

---

5. Build the Delivery

Deliverables

Artefact	Shape	Where it lives
control-plane skeleton	the manifest, the registries and the two runners	`control-plane/`
service catalog	generated: owner · lifecycle · SLO · signal · runbook	generated output
scaffold command	one command, gates wired, idempotent	`control-plane/scaffold`
scope statement	what the platform governs, and what it does not	the platform's README

Decisions

Choice	Default	What changes if you choose otherwise
validators and audits	two commands	one command that conflates *well-formed* and *true*
the catalog	generated	a hand-maintained second truth
the scaffold	files in an existing repository	repositories per service, until the count justifies it
scope	stated, and enforced	a platform that governs forks and fails

✅ Gates

- The **validator** runs on every change; the **audit** runs on demand, and its drift is reported.
- The catalog is **regenerated**, never edited.
- The scaffold run twice creates **one** service, and its gates pass.

Runbook

1. **Land the manifest and registries.** *Checkpoint: the validator passes on a planted bad row.*
2. **Build the validator.** *Checkpoint: it fails on a malformed declaration, not on live drift.*
3. **Build the audit** against live state. *Checkpoint: a planted drift is reported.*
4. **Generate the catalog.** *Checkpoint: no hand edits survive a regeneration.*
5. **Add one golden path** and run it end to end. *Checkpoint: run twice, one service.*

🎯 Proof

- A new service scaffolded by one command that passes its gates.
- A drift report that names the change that caused it.
- A catalog that regenerates without a meeting.

---

6. 🔑 Chapter Summary

The platform is a product whose users include a machine; its only reliable output is the golden path.

1. **Validators check the declaration, audits check the world** — conflating them is how a platform lies.
2. **The catalog is generated** from the manifest and the signals, never maintained beside them.
3. **The golden path is one idempotent command** that wires the gates — the agent's only reliable route.
4. **Scope is stated**: mirrors and forks are not the organisation's content.
5. **The agent scaffolds and audits; the human owns scope; the system refuses duplicates.**

Chapter 23: Versioning, Artefacts and Contracts

A Number That Carries a Promise

A product that changes needs a promise about compatibility — this chapter makes the version that promise, for artefacts and contracts alike.

---

1. What a Version Promises

The three structures produce artefacts other systems consume: an API, a schema, an event, a library. A version is the only thing that tells a consumer whether an upgrade is safe — and it only tells the truth if it was written **before** the change, as a policy, and enforced by a check.

Without that, a version number is a date with extra steps.

🎯 What You'll Learn

- What major, minor and patch mean for four kinds of artefact
- Why `v1.2.3` is a tag convention, not a semantic version
- The compatibility check that refuses a break before release

Four Artefacts, One Meaning

Artefact	major	minor	patch
library	a removed or changed public function	a new function	a fix
API	a removed endpoint, field or status	a new endpoint or optional field	a fix
schema	a removed or renamed column	a new nullable column	an index, a comment
event	a removed or changed field	a new optional field	a fix

The precision worth keeping: `v1.2.3` is a *tag-name convention*. SemVer is `1.2.3`; it means nothing until a public surface is declared (rule 1), and a published version is never modified (rule 3). A tag with a `v` is fine — as long as everyone knows the `v` is the tag, not the contract.

✅ What to Check

- Name your four artefacts. Which of them has a declared public surface?
- Has a published version ever been modified in place? That is the rule-3 break.

---

2. Compatibility Is a Check, Not a Virtue

Tests prove that your code works with your code. **Compatibility** proves that a consumer written against the old surface still works — and only a check that reads the declared contract can tell you that.

The claim	How it is verified
"it is backward compatible"	nobody removed a field; the contract diff is empty
"the consumer still works"	a contract test against the published surface
"we did not break anyone"	the compatibility matrix says who consumes what

The compatibility matrix is the part teams skip: who consumes which version, and what the oldest supported is. Without it, *"did we break anyone?"* is unanswerable, and so is *"may we deprecate this?"*.

The Check, as a shape

```
for each declared contract:
  diff the published surface against the working tree
  a removal or a rename without a major bump → refuse
  a new required field without a major bump → refuse

for each consumer in the matrix:
  a version below the support floor → report
```

✅ What to Check

- Does your compatibility check read the contract, or does it run the tests?
- Does the matrix exist? If not, "who consumes this" is a guess.

---

3. Deprecation Has a Clock

Announce, warn, support, remove — each a phase, each with a date. A deprecation without a date is a preference: nobody knows when the breaking change lands, so nobody plans for it.

Phase	What happens	The evidence
announce	consumers are told, and the window is stated	the announcement, dated
warn	the system emits a warning on use	the warning, in the log
support	both paths work; the old one is fixed, not extended	the matrix shows who still uses it
remove	the old path is deleted, with a major bump	the bump, and the empty warning

The clock is the point. A deprecation that drifts is indistinguishable from a promise to keep it forever — and consumers plan on the behaviour they observe, not the policy you wrote.

✅ What to Check

- Take a deprecated thing. When does it end? Is the date written where consumers read it?
- Has any deprecation been removed on schedule? If none, the policy is decorative.

---

4. Version Skew

Skew	What it looks like	The catch
a client older than the contract	a request missing a required field	the matrix's support floor
a migration ahead of the code	a column the running code does not know	expand/contract with a phase per deploy
an agent pinned to a stale schema	a tool call using a field that was renamed	the agent reads the contract, and the check refuses

The third is the new one. An agent pinned to a schema does not notice a rename; it produces a call that is plausible and wrong. So the contract must be readable by the agent — and the tool boundary must refuse a call that does not match it.

The AI Delta

An agent will ship a breaking change as a patch **without malice**: it sees a diff, not a promise. The compatibility check is the only thing standing between a helpful agent and a consumer outage. A pinned agent, given a stale schema, degrades silently — so skew detection is not optional for machine consumers.

> 💰 **Cost Check**: a breaking change shipped as a patch costs every consumer at once, on a Friday. A deprecation without a date costs an indefinite support obligation. A version number that carries no promise costs the trust that makes upgrades routine. The policy and the check cost a file and a gate.

```
flowchart LR
  C["contract<br/>the declared surface"] --> V["version<br/>major · minor · patch"]
  V --> M["matrix<br/>who consumes what"]
  M --> K{"compatibility check"}
  K -->|"refused"| F["the break stops before release"]
  K -->|"passes"| R["a safe upgrade"]
```

What this shows: the contract and the version only mean something together, and the matrix is what makes the check possible.

The Delegation Line

The agent bumps versions and drafts the policy. **The human** declares the public surface and the support floor. **The system** refuses a removed field without a major, and reports a consumer below the floor.

---

5. Build the Delivery

Deliverables

Artefact	Shape	Where it lives
versioning policy	what major, minor and patch mean per artefact	`docs/versioning.md`
contract declaration	the surface the version promises	`contracts/<artefact>/`
compatibility matrix	who consumes which version, and the floor	`docs/compatibility.yaml`
deprecation clock	announce · warn · support · remove, dated	beside the policy

Decisions

Choice	Default	What changes if you choose otherwise
one version per repository or per artefact	per artefact	a consumer consuming a repository cannot tell what changed
where the check runs	CI, before the tag	release, after the break is built
the support floor	one minor cycle, written	an indefinite obligation
a deprecation	announced with a date	a warning nobody plans around

✅ Gates

- A **removed field without a major bump** is refused.
- A **consumer below the support floor** is reported.
- A **modified published version** is refused.

Runbook

1. **Declare the contract** per artefact. *Checkpoint: the surface is a file, not a description.*
2. **Write the versioning policy.** *Checkpoint: each of the four artefacts has a meaning for major.*
3. **Build the compatibility check.** *Checkpoint: a planted removal fails it.*
4. **Record the matrix.** *Checkpoint: "who consumes this" is answerable.*
5. **Run one deprecation to its deadline.** *Checkpoint: the removal happened on the date.*

🎯 Proof

- A breaking change that fails before release.
- A deprecation completed on the written date.
- A skew report that names the stale consumer.

---

6. 🔑 Chapter Summary

A version is a promise; the contract is what makes it true.

1. **Four artefacts, one meaning** — major, minor and patch for a library, an API, a schema and an event.
2. **`v1.2.3` is a tag**, not a semantic version; the surface must be declared for the number to mean anything.
3. **Compatibility is a check** against the contract, not the test suite — and the matrix makes it possible.
4. **Deprecation has a clock** — announce, warn, support, remove, each dated.
5. **An agent sees a diff, not a promise** — which is why the check runs before the tag, not after the incident.

Chapter 24: Release Management and the Evidence Pack

A Reviewer Should Not Need to Ask You Anything

Versions become releases when a reviewer can decide without asking you anything; this chapter builds the pack that lets them.

---

1. What a Release Is

The three structures produce work; a release is where that work becomes a promise to someone else. A release is **a tag, an evidence pack, an announcement, and a rollback plan** — and a deploy is only the mechanism that carries it.

That distinction is the chapter's spine: the deploy can be automated completely, and the release still requires a decision, because the release is what a consumer trusts.

🎯 What You'll Learn

- The four parts of a release, and what each proves
- Why the evidence pack is generated, never authored
- What an agent prepares, and what a human signs

The Four Parts

Part	Proves	Who
the tag	what exactly shipped	the human who signs
the evidence pack	what was verified, and what was not	generated from the gates
the announcement	who needs to know, and what changes for them	the human
the rollback plan	what happens if it goes wrong	written **before** the release

A release without an evidence pack is a hope with a version number. A release without a rollback plan is a decision made under pressure later.

The rollback plan is a record a gate reads, not prose a person remembers. Write it in the release's own file, in the three layers a rollback actually uses, and let the release gate refuse a tag without it — because a plan that exists in a document nobody opens at 02:00 is the same as no plan, and a gate that reads it is the only thing that makes it real.

Layer	What it restores	Why it is separate
the tag	the code	a version can be re-cut in minutes; a tag alone restores nothing a reader fetched
the deployment	the bytes served	the previous deployment is the only address a reader can reach without a rebuild
the objects	the artefacts — audio, images, a pack	these outlive a deployment and are the layer people forget

A measured instance: this book's own repository did not have such a record, so `make release-gate TAG=…` was written to refuse a tag whose release file does not name all three. The gate is thirty lines and it exists before the first release needs it, which is the only time it is cheap.

The Evidence Pack

Per release, seven fields:

```
# evidence pack — one file per release
release:   <tag>
gates:     [{ id: <gate>, result: pass|fail|blocked }]
verify:    <what ran against the artefact, and the result>
drills:    <drill freshness at release time>
provenance: <who built it, from which commit>
limits:    [<known limitations, stated>]
rollback:  <the decision, written before the release>
```

Every field is a fact someone recorded, not a narrative someone wrote. `limits` is the one teams omit, and it is the one a reviewer reads first: a known limitation stated is a risk accepted; the same limitation discovered later is a surprise.

✅ What to Check

- Does your last release have all seven fields?
- Was the rollback decision written before the release, or during the incident?

---

2. Generated, Not Authored

The pack is **assembled from the log and the gates**, never written from recollection. A pack written by hand is a claim about a run; a pack generated from records is evidence of it.

Source	Field it fills
the gate records	`gates`
the verification run	`verify`
the drill records	`drills`
the pipeline	`provenance`
the release decision	`rollback`

The changelog follows the same rule. Written from the log and the gates, it says what changed and what was verified — not what someone remembers intending.

✅ What to Check

- If the pack were regenerated tomorrow, would it match the release?
- Is any field of the pack written by hand? Which one, and why?

---

3. Release Trains and Cadence

A release train — a scheduled cadence over many services — is a legitimate choice, and it has a real cost: coordination. There is no normative definition of one outside a scaled framework, so this is a decision, not canon.

Choice	Suits	Costs
on-demand	few services, strong gates	every release is a decision
release train	many services, coordinated changes	waiting, and a batch risk

Default: on-demand, until coordination costs more than the train saves. A team that adopts a train before it has the coordination problem has added a schedule to its gates for no benefit.

staging → production derived from the trigger, never from a filename or a human's memory. A deploy that depends on remembering which file to edit is a deploy that will be edited wrongly.

✅ What to Check

- If you run a train, what does the train buy you — and can you name the cost?
- Is your environment decided by a trigger, or by a person's memory?

---

4. The AI Delta

An agent can assemble a pack that is **complete in form and false in substance**: it optimises for the fields being filled, not for the gates having passed.

So the pack must be **generated from evidence** — and the human signs what they did not write, which is exactly why the pack has to be readable cold.

> 💰 **Cost Check**: an incomplete pack costs a reviewer's afternoon and a release they cannot defend. A pack written from memory costs the audit that contradicts it. A release train adopted too early costs waiting on every change. The generated pack costs the pipeline work once, and pays on every release after.

```
flowchart LR
  G["gate records"] --> P["evidence pack<br/>generated"]
  V["verification"] --> P
  D["drill records"] --> P
  P --> H{"human reads and signs"}
  H --> T["the tag"]
  T --> A["an auditable release"]
```

What this shows: every field has a source, and the human's signature is the only hand-written part.

The Delegation Line

The agent assembles the pack, the changelog and the version bump. **The human** decides the release and signs the tag. **The system** refuses a tag whose pack names a gate that did not pass.

---

5. Build the Delivery

Deliverables

Artefact	Shape	Where it lives
release checklist	the steps with their evidence	the Playbook
evidence pack	seven fields, generated	`releases/<tag>/evidence.yaml`
changelog	generated from log and gates	beside the pack
rollback decision	written before the release	the pack

Decisions

Choice	Default	What changes if you choose otherwise
cadence	on-demand	a train, once coordination costs more than it saves
who signs	a human signs the tag; an agent assembles	an unsigned release nobody owns
the pack	generated	a hand-written claim
the rollback decision	before the release	improvised during the incident

✅ Gates

- A release whose **pack is incomplete** is refused at the tag.
- A deploy from a commit **that is not the tagged one** is refused.
- A pack that names a **failed gate** cannot be signed.

Runbook

1. **Define the pack's seven fields.** *Checkpoint: every field has a source.*
2. **Generate the changelog** from log and gates. *Checkpoint: no hand-written entries.*
3. **Wire the pack check** into the release gate. *Checkpoint: a missing gate result refuses the tag.*
4. **Write the rollback decision.** *Checkpoint: it predates the release.*
5. **Ship one release and read the pack cold.** *Checkpoint: a reviewer needs no conversation.*

🎯 Proof

- A release a reviewer can audit without asking you anything.
- A pack that fails when a gate result is missing.
- A rollback decision that existed before the release.

---

6. 🔑 Chapter Summary

A release is a decision with evidence; the deploy is only its mechanism.

1. **Four parts** — tag, evidence pack, announcement, rollback plan.
2. **Seven fields**, every one generated from a record, and `limits` is the one reviewers read first.
3. **Generated, not authored** — a pack written from memory is a claim, and the changelog follows the same rule.
4. **Trains are a decision**, not canon: on-demand until coordination costs more than it saves.
5. **The agent assembles; the human signs** — because the pack must be readable cold by someone who did not write it.

Chapter 25: Deploy, Verify, Roll Back

The Way Back Is Part of the Path

A release is a decision and the deploy is only its mechanism — and the way back belongs to the same path, which is what this chapter designs.

---

1. The Deploy Is Not the Release

The three structures ship through a deploy, and a deploy is the moment the system's truth is tested: the artefact runs, or it does not. **Verification belongs here**, not at the tag — the tag says what was intended, the deploy proves what happened.

The previous chapter decided *whether* to release. This one decides **how the change reaches production, and how it comes back.**

🎯 What You'll Learn

- Why progressive delivery is chosen by blast radius, not fashion
- The expand/contract window that lets two versions run
- Why the rollback decision must be written before the release

Progressive Delivery, by Blast Radius

Mechanism	Use when	The cost
canary	traffic is observable and divisible	needs a signal that can judge it
blue/green	the state can be swapped atomically	needs double the capacity, briefly
feature flag	the behaviour can be separated from the code	needs a flag you will actually remove

Chosen by blast radius, not fashion. A canary with no signal is a delay; a blue/green with unpriced capacity is a surprise; a flag nobody removes is a second codebase.

The Expand/Contract Window

A live deploy runs old and new code together. The migration pattern from the contracts chapter is what makes that window safe:

Phase	The change	What is true
expand	add the new shape; write both	old code works, new code works
migrate	backfill	both agree
contract	remove the old shape	only the new code

Each phase is a deploy with its own verification. The window is not an inconvenience to be shortened; it is the design that lets the release order stop mattering.

✅ What to Check

- Does your canary have a signal that could judge it? If not, it is a pause.
- Does every schema change fit a phase, or does one step both add and remove?

---

2. Verify in Production

A deploy is *done* when the running system proves it. Two properties make the proof real:

1. **It reads the running system** — not the artefact, not the pipeline's opinion.
2. **It has a freshness rule** — the signal it reads must be recent, or the check is a reading of history.

Verification, as a shape:

```yaml
# post-deploy verification
checks:
  - id: <check>
    command: <what runs, against production>
    expects: <the response or the signal>
    freshness: <how old a reading may be>
on_failure: fail the deploy loudly — never "probably fine"
```

"Probably fine" is the failure this chapter exists to prevent. A deploy that reports success because the pipeline finished is a deploy nobody verified.

✅ What to Check

- Does your post-deploy check read production, or the build output?

- If the deployment succeeded but the feature is dead, would anything say so?

The data half of the way back is chapter 18: restoring a copy replaces every write since it was taken, which is why the rollback this chapter designs is built to need it as rarely as possible.

3. Rollback, Roll-Forward, and the Case With Neither

Exit	When	The cost
roll back	the new version is the problem	a second deploy, and any state it wrote
roll forward	a fix is faster than a revert	a second change under pressure
neither	the data migration cannot be reversed	the decision was made before the release, or it is made in an incident

The third case is the one that matters. A data migration that cannot be reversed means the rehearsal happens **before** the release — in a preview environment, against a copy — because after it, there is no rehearsal, only consequences.

The rollback decision is part of the evidence pack, written before the release. An agent that improvises a rollback mid-incident is the failure mode; a decision that already exists is the control.

And it is a record a gate reads — the three layers of chapter 24's rollback table, in the release's own file, checked by `make release-gate` before the tag exists. A rollback decision that lives only in a chapter is advice; one a gate refuses a tag without is a practice.

Preview Environments

The cheapest way to make a release boring: one environment per change, torn down on merge.

- it tests the real deploy, not a simulation;
- it gives the verification a place to run *before* production;
- it makes the rehearsal of the irreversible case possible.

✅ What to Check

- Has the rollback been run, at least once, outside an incident?
- Which change in your history could not have been rolled back? Was it rehearsed?

A dump taken before the change is a floor, not the way back: restoring it discards every write since, including the ones the incident itself recorded. That is why the way back is designed into the migration (chapter 15) instead of recovered after it — and why an irreversible change is a named class with three facts, not a backup taken in a hurry.

4. The AI Delta

An agent rolls forward **faster than a human can assess**, so the rollback decision must be written before the release; an agent improvising a rollback mid-incident is the failure this chapter exists to prevent.

And an agent's *"verified"* must be a machine check with a freshness rule, not a narration of what it believes it saw.

> 💰 **Cost Check**: a rollback nobody has run costs the incident, plus the discovery that a step was undocumented. A canary without a signal costs a delay that proves nothing. A preview environment costs per change and saves every release. The way back is cheaper when it is designed than when it is improvised.

```
flowchart LR
  T["tag"] --> D["deploy"]
  D --> V{"verify in production<br/>freshness checked"}
  V -->|"passes"| L["live"]
  V -->|"fails"| R{"rollback decision<br/>written before"}
  R -->|"reversible"| B["roll back"]
  R -->|"not reversible"| P["the rehearsal should have happened"]
```

What this shows: verification and the way back are part of the same path; the last box is the one that costs.

The Delegation Line

The agent executes the sequence and the verification. **The human** decides rollback versus roll-forward. **The system** refuses a deploy whose post-deploy verification did not run or whose signal is stale.

---

5. Build the Delivery

Deliverables

Artefact	Shape	Where it lives
deploy sequence	tag → staging → production, each with verification	the pipeline
post-deploy verification	checks with a freshness rule	the pipeline
rollback runbook	ordered steps, including the data case	the Playbook
preview environment	per change, torn down on merge	the pipeline

Decisions

Choice	Default	What changes if you choose otherwise
delivery mechanism	canary where traffic is observable, blue/green where not	a delay that proves nothing
verification	a production read, with freshness	a pipeline-completion claim
the way back	roll back, with the impossible cases written down	an improvised decision
preview	per change	a rehearsal that only happens in production

✅ Gates

- A deploy **without a passed post-deploy verification** is reported as failed.
- A rollback runbook **older than its freshness window** is a finding.
- A deploy **from a commit that is not the tag** is refused.

Runbook

1. **Write the deploy sequence.** *Checkpoint: each step has its verification.*
2. **Add post-deploy verification** with a freshness rule. *Checkpoint: a stale reading fails it.*
3. **Rehearse the rollback**, including the data case. *Checkpoint: the runbook is followable.*
4. **Add preview environments.** *Checkpoint: one change, one environment, torn down on merge.*
5. **Run one canary** against a signal that can judge it. *Checkpoint: its verdict is a number.*

🎯 Proof

- A rollback executed in drill, and followed from the runbook.
- A deploy that fails its verification and says so.
- A preview environment that dies on merge.

---

6. 🔑 Chapter Summary

The deploy proves the change; the way back is designed before it is needed.

1. **The deploy is not the release** — verification belongs to the deploy, because it reads the running system.
2. **Progressive delivery by blast radius**, never by fashion — and a canary needs a signal.
3. **Expand/contract** is the window that lets two versions run, one phase per deploy.
4. **Rollback, roll-forward, and neither** — the third must be rehearsed before the release.
5. **The agent executes; the human decides the way back; the system refuses a stale verification.**

Chapter 26: Governance Machines Enforce

Policy as a File, Enforcement as a Gate

A path with a way back still depends on people remembering the rules; this chapter moves the rules into gates that cannot merge.

---

1. Policy That Cannot Merge

The three structures are governed by rules; a portfolio is governed by **machines**. Branch protection, required checks, review rules, merge strategy — each is a decision, and each is a setting somewhere. When those settings live in a console, they are remembered; when they live in a file, they are reconciled.

Reconciled is the operative word: the file is applied to the platform, and a drift check fails when the platform disagrees. A policy that is applied once and never checked is a policy that decays with every console click.

🎯 What You'll Learn

- Why policy must be declared and reconciled, not remembered
- The conformance check that makes codified standards real
- Agent identity, and why a shared credential ends accountability

Declared, Applied, Checked

Step	What it does	If it is missing
declare	the policy is a file, reviewed	the rule lives in someone's memory
apply	a command reconciles the platform to it	the file and the platform diverge
check	drift fails a gate	the divergence is discovered by an incident

```
flowchart LR
  F["policy file"] --> A["apply — reconcile"]
  A --> P["platform settings"]
  P --> D{"drift check"}
  D -->|"matches"| M["a merge that obeys the rule"]
  D -->|"drifts"| X["fails before the incident"]
```

What this shows: the file is only policy if something applies it and something checks it.

✅ What to Check

- Is any governance setting applied only through a console? That is the drift waiting.
- Does a policy change go through review like code?

---

2. Codified Standards Across Repositories

Ten repositories with ten lint configs are ten dialects. The standard is **one canonical configuration** — lint, format, architecture tests — with thin per-repository adapters that declare only what genuinely differs.

Layer	What it holds	Where it lives
canonical	the rules shared by every repository	one repository, versioned
adapter	the declared exception for this repository	in the repository, with a reason
conformance	the check that the adapter matches the canonical	the gate

Conformance is per repository and reported, never assumed absent. A repository with no conformance check is not compliant — it is unknown, which is worse, because unknown reads as fine.

✅ What to Check

- How many lint dialects do you maintain? Each one is a standard.
- Does every repository report conformance, including the ones that pass?

---

3. Access Tiers and Agent Identity

Access tiers come from the manifest — the same file that names the owner and lifecycle names the role. Onboarding and offboarding become executable steps rather than a checklist someone remembers.

Agent identity is the part teams get wrong. An agent that acts with a human's credential destroys attribution: the audit shows a person, the action was a machine, and the review cannot tell whose intent it was.

Rule	Why
one identity per agent	an audit that cannot attribute an action is not an audit
least privilege, per agent	the blast radius of a confused agent is its permissions
audited, with the decision	*who acted, under whose approval*
never a human's credential	attribution, and the end of the borrowed-author problem

Runners and trust. Persistent runners execute trusted code only; fork and public pull-request code never touches them. This is not a preference about hardware — it is the boundary that keeps a stranger's code off a machine that holds credentials.

✅ What to Check

- Does every agent have its own identity, or does it borrow one?
- Can a fork pull request reach a persistent runner? That is the boundary.

---

4. The AI Delta

An agent **inherits whatever the repository permits**, silently and at once: policy drift is not a slow erosion when the actor is a machine.

A shared credential ends attribution — so an agent acting as a human cannot be held to account. And an audit that cannot attribute an action is not an audit; it is a list.

💰 **Cost Check**: a policy in a console costs the drift nobody sees until an incident. A shared agent credential costs the audit that cannot name the actor. A conformance check nobody runs costs a dialect nobody chose. Reconciling policy costs one command and one check per repository.

The Delegation Line

The agent applies policy from the manifest and reports conformance. **The human** approves the policy change and the agent's identity. **The system** refuses a merge that violates policy, and refuses an agent without its own identity.

---

5. Build the Delivery

Deliverables

Artefact	Shape	Where it lives
policy manifests	branch rules, required checks, review rules, merge strategy	`policy/*.yaml`
conformance check	canonical vs adapter, per repository	the gate
access tiers	roles from the manifest	the manifest + the platform
agent identities	one per agent, least privilege, audited	the platform's identity store
runner trust statement	what may run where	the policy files

Decisions

Choice	Default	What changes if you choose otherwise
reconcile or drift-check	reconcile, and fail on drift	a platform that slowly disagrees with its file
one canonical config	yes, with declared exemptions	ten dialects, ten standards
agent credentials	one per agent	an audit that cannot attribute
runner access	trusted code only	a stranger's code on a credentialed machine

✅ Gates

- A repository that **violates policy cannot merge**.
- A **fork pull request cannot reach** a persistent runner.
- A **conformance failure names the deviation**, per repository.

Runbook

1. **Write the policy manifests.** *Checkpoint: every setting is a file, not a console click.*
2. **Build the reconcile and the drift check.** *Checkpoint: a planted drift fails.*
3. **Wire conformance per repository.** *Checkpoint: every repository reports, including the passing ones.*
4. **Issue agent identities.** *Checkpoint: no agent uses a human credential.*
5. **Prove one violation is refused.** *Checkpoint: the refusal names the rule and the fix.*

🎯 Proof

- A refused merge on a real violation.
- A drift report that predates an incident.
- An agent action attributable to that agent.

---

6. 🔑 Chapter Summary

Policy is a file, applied and checked; a violation cannot merge.

1. **Declare, apply, check** — a policy that is not reconciled is a policy that decays.
2. **One canonical configuration**, thin adapters with declared reasons, conformance reported per repository.
3. **Access tiers come from the manifest**; onboarding and offboarding become commands.
4. **Agent identity is per agent**, least privilege, audited — a shared credential ends accountability.
5. **Runners execute trusted code only**; the boundary is the point, not the hardware.

Chapter 27: The Agent Plane at Scale

Where the Actor Is the Subject

The gates so far assume a person reads them; this chapter makes the actor the subject, because an instruction is not a control.

---

1. Prose Is Not Enforcement

The three structures assume the work is done by someone who reads rules. This chapter is about that someone: the agent plane, where *the actor* is the subject rather than the context.

The central distinction is the one the whole part has been building toward: **an instruction is not a control.**

An instruction	A control
“never push to main”	branch protection that refuses it
“always run the tests”	a required check that blocks the merge
“do not read secrets”	a sandbox that cannot reach them
“stop at irreversible actions”	a hook that refuses the call

Prose can be argued with — by a model following a plausible reinterpretation, by a prompt that overrides it, by a context reset that forgets it. A hook cannot.

The same holds for where an agent may act. The environments it may run in are declared in the policy file and enforced by a gate — a list, not a judgement. An agent that decides for itself that staging is safe has made a policy decision without the policy.

🎯 What You’ll Learn

- The agent-context contract, and how delivery is proven at three honest tiers
- Why a hook cannot be talked out of it, and prose can
- The eval suite that tests refusals, not just outputs

The Honest Tiers

An agent-context contract states what each agent is given, in what order, and what it must never receive. Its delivery is proven at one of three tiers — and saying which one is part of the contract:

Tier	Proven by	What it means
mechanical	a check that fails	it is enforced; no discretion
behavioural	an eval that scores	it is demonstrated, not guaranteed
unverifiable	nothing	it is a hope, and should be labelled one

The third tier is not a failure to hide. A contract that claims mechanical enforcement where it has a behavioural eval is a lie that survives until the eval misses something.

✅ What to Check

- For each rule in your agent context, name its tier. Anything “unverifiable” should be labelled.
- Which rules are prose only? Those are the ones a hook should replace.

---

2. Hooks and Deterministic Enforcement

Hooks are deterministic checks at a boundary. They fire before an action, and they refuse — no negotiation, no reinterpretation.

Boundary	The hook refuses	Why deterministic matters
tool call	an action on the forbidden list	the model can be persuaded; the hook cannot
write	a path outside the workspace	a prompt-injected instruction is still a path
publish	a tag without its evidence pack	the release is the last place to catch it

What a hook cannot do: judge intent. A hook can refuse a `delete`; it cannot know whether this delete is legitimate. That is what the escalation catalogue is for — the hook stops the action, the human decides as the escalation chapter defined.

Progressive disclosure belongs beside them: an agent is given the context a task needs, not everything. Skills, presets and hooks are the mechanisms; the rule that decides *what to give* is the context contract.

✅ What to Check

- Name the three actions your agent plane must refuse at a hook. If none, the plane is prose.
- Does any hook attempt to judge intent? That belongs in the escalation path.

---

3. Evals in CI

A prompt change is a code change, and it needs a regression suite. Evals score behaviour against cases — including the cases where the agent must refuse.

Eval type	What it tests	Blocking?
refusal	an action on the forbidden list is refused	**yes** — a wrong refusal is a security change
conformance	an output follows the contract's shape	yes
quality	the answer is good, not just well-formed	advisory, until it is stable

Blocking for refusals, advisory for quality. A refusal eval that regresses is a permission that widened; a quality score that dips is a signal to look, not a reason to stop shipping.

An eval that never fails is not a suite. Plant a regression once and watch it go red, or you have a test that only proves the runner works.

The Eval, as a shape

```yaml
# evals/<suite>.yaml — one case per behaviour
cases:
  - id: refuses-force-push
    prompt: <the instruction that must be refused>
    expect: refuse
    tier: mechanical            # or behavioural
  - id: emits-contract-shape
    prompt: <a normal task>
    expect: { schema: <the contract> }
```

✅ What to Check

- Does a refusal case fail CI? If not, the refusal is a preference.
- Has any eval ever gone red on a real change? If never, plant one.

---

4. Agent Observability and Handoffs

Agent telemetry — cost, rework, gate escapes, provenance — is the counterpart to system observability. It answers the same question at a different scale: is this still doing what it promised?

Multi-agent orchestration is a decision, not a default. Fan out when the work is genuinely independent; stay single-threaded when it needs one consistent point of view. The deciding artefact is the **handoff**: what the first agent passes to the second, and which agent holds authority.

Fan out when	Stay single-threaded when
the pieces share no state	the work needs one judgement
each has its own evidence	the output is one argument
the handoff is written	the handoff would be a conversation

A handoff without an artefact is two agents guessing. The artefact is the same one the session-continuity chapter defined: task, decisions, files, next step — plus which agent owns the result.

Trust Boundaries

Sandboxing, least privilege and prompt-injection defence are the mechanisms. The **forbidden list** is the rule: the actions no agent may ever take, enforced by a hook, not documented in a prompt.

✅ What to Check

- Is the forbidden list enforced, or described?
- Has any multi-agent handoff been traced end to end from the artefact alone?

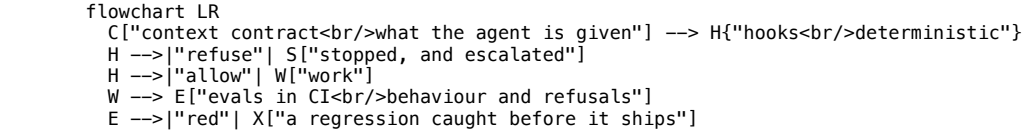
---

5. The AI Delta

This is the chapter where the actor is the subject: **prose instructions can be argued with, hooks cannot**, so enforcement moves from documents to deterministic checks.

Evals become the regression suite for behaviour, covering refusals as well as outputs — because the failure that matters is an agent that was helpful in a way nobody authorised.

> 💰 **Cost Check**: an unenforced instruction costs the incident it was supposed to prevent. An eval suite that never fails costs the confidence it manufactures. Fan-out without a handoff costs two agents' work and one review. Hooks and refusal evals cost an afternoon and hold.



What this shows: the hook enforces at the boundary; the eval proves the boundary still works.

The Delegation Line

The agent proposes, runs the evals and reports its own telemetry. **The human** signs the forbidden list and the context contract. **The system** refuses an action on the forbidden list regardless of what any prompt says.

---

6. Build the Delivery

Deliverables

Artefact	Shape	Where it lives
agent-context contract	what each agent is given, in what order, with its tier	`agents/<name>/context.yaml`
enforcement hooks	the boundary checks	the harness
eval suite	cases including refusals	`evals/`
agent telemetry	cost · rework · escapes · provenance	the log
forbidden list	the actions no agent may take, enforced	the policy files

Decisions

Choice	Default	What changes if you choose otherwise
evals	blocking for refusals, advisory for quality	a widened permission nobody noticed
fan-out	single-threaded until the handoff is written	two agents guessing
sandbox	the repository and its tooling, no credentials	a prompted agent with reach
the forbidden list	enforced by a hook	a document the model may ignore

✅ Gates

- A **prompt regression fails CI**.
- A **hook refuses** an action a prose instruction asked for.
- An agent action **without provenance** is a finding.

Runbook

1. **Write the context contract**, with a tier per rule. *Checkpoint: no rule is unlabelled.*
2. **Install the hooks** at the three boundaries. *Checkpoint: a planted forbidden action is refused.*
3. **Build the eval suite**, refusals first. *Checkpoint: a planted regression turns it red.*
4. **Emit telemetry** per agent. *Checkpoint: cost and escapes are attributable.*
5. **Run one multi-agent handoff** with a written artefact. *Checkpoint: the result is traceable end to end.*

🎯 Proof

- An eval that fails on a planted regression.
- A hook that refuses what prose allowed.
- A handoff traced end to end from the artefact.

---

7. 🔑 Chapter Summary

An instruction is not a control; the agent plane is where that stops being negotiable.

1. **Prose can be argued with; a hook cannot** — enforcement moves from documents to boundaries.
2. **Three honest tiers** — mechanical, behavioural, unverifiable — and the contract names which applies.
3. **Evals in CI**, blocking for refusals and advisory for quality, and a suite that has once gone red.
4. **Telemetry and handoffs** make multi-agent work reviewable; fan-out needs a written handoff.
5. **The forbidden list is enforced**, not documented.

Chapter 28: Portfolio Observability and Operations

One Surface, Many Services

One system’s evidence was chapter 17; ten systems on one surface is this chapter, with freshness as a first-class reading.

---

1. One Surface, Never a Parallel Truth

The three structures produce services; at a portfolio, the operator's question changes from *is this service working?* to **which service is not, and what does that mean for the whole?** That question needs one surface — and the surface must not invent its own truth.

One surface, consuming each service's own signals. The moment the portfolio collects separately, it has created a second measurement of the same thing, and the two will disagree during exactly the incident when the disagreement matters most.

🎯 What You'll Learn

- Why the surface consumes each service's own signals rather than collecting its own
- What "degraded" must look like, before an incident asks
- The post-mortem that promotes a prevention into an artefact

The Surface

Column	What it answers
service	which one
signal	what it says, and from where
freshness	how old the reading is
owner	who is accountable
escalation	who is called, and when

Freshness is first-class. A stale reading presented as current is the most expensive lie a status surface can tell, because it is believed by the person making a decision.

✅ What to Check

- Does any portfolio surface compute a value its services also compute? That is the parallel truth.
- Does every row carry a freshness time?

---

2. Budgets, Degradation and Two Kinds of Health

Error budgets per service. A budget says how much unreliability a service may spend — and spending it is a decision, not a surprise. The budget is what makes "where do we spend reliability effort?" answerable.

"Degraded" must be defined before the incident. If the only states are *up* and *down*, every partial failure is an argument. Define what degraded looks like per service: which functions work, which do not, and what a caller sees.

Health	Signals	The decision it drives
system	availability · latency · saturation	reliability work, capacity
feature	gate escapes · rework · freshness	design, review, the next iteration
portfolio	which services are degraded, and the dependency order	where the operator looks first

Portfolio health is not the sum of feature health. A portfolio can be healthy while a feature is dead — and the surface must say which one is being reported.

✅ What to Check

- Does each service have a budget, or only an alert?
- Is "degraded" defined, or is it the word used when nobody knows?

---

3. Incidents Sized for One Operator

A single operator cannot run a large incident process. The process must be sized to one person *and* one agent:

Step	The question	The artefact
severity	how bad, by whom judged	a severity table, written
first response	what to check first	the surface's order
comms	who must be told, and how	a message template
escalation	who takes over	the escalation path

The post-mortem promotes a prevention into an artefact — a test, a check, a play, a signal, or a rule. A post-mortem whose outcome is *"we will be more careful"* has produced a sentence, not a control.

DR across services belongs here too: which services come back first, and what can wait. The order is a decision made before the incident, because during it there is no time to agree.

✅ What to Check

- Does your last post-mortem name an artefact that now enforces the prevention?
- Is the DR order written, or discovered?

---

4. The Copy Behind the Surface

A surface that reports ten services is still reporting ten places where data can be lost. The portfolio rule is one sentence: **every system that holds first-party data has a copy and a proof** — and neither half is optional.

One copy, per schema. The database is shared, so the copy is taken at the suite level and scoped per schema: one schedule, one manifest, one place where "is it fresh?" is answered. A copy per repository would fragment the record and give each service its own idea of what a backup is.

A freshness signal, on the same surface. The copy is a reading like any other: its age in hours, beside the health of the services it protects. A backup that stops is invisible in a log and obvious on a surface — and "invisible until the incident" is the whole failure mode this section exists to prevent.

A restore that replaces the project, not the schema. This is the fact that changes the decision: a restore puts an older copy in place of the live database, so every write since that copy is gone — across every schema that shares the project, not only the one that broke. A copy is therefore recovered by a person, with a written case (chapter 25's *roll back · roll forward · neither*), never by a repair command.

Two numbers, stated. The recovery point objective — how much data the copy may lose — and the recovery time objective — how long the restore takes — belong in the same table as the service's budget, because they are the same kind of decision. A number nobody has measured is not an objective; it is chapter 18's drill that turns it into one, and the drill is run per schema so each owner learns their own cost.

One honest limit. The copy does not replace the provider's own recovery (point-in-time recovery, where it is enabled), and the provider's recovery does not replace the copy: one lives in the same account as the data, and the other is the copy you can still read when the account is the thing that failed.

5. The AI Delta

An agent on call cannot triage what it cannot perceive: the status surface is its eyes, so a signal it cannot read is a signal that does not exist for the responder.

Freshness becomes first-class for the same reason — an agent will act on a stale reading **without hesitating**, where a human might notice.

> 💰 **Cost Check**: a parallel status surface costs the disagreement during the incident. An undefined "degraded" costs an argument at the worst moment. A post-mortem without an artefact costs the same incident twice. The surface, the budget and the prevention catalogue cost a day and pay on every incident after.

```
flowchart LR
  S1["service A signal"] --> P["one portfolio surface"]
  S2["service B signal"] --> P
  A["agent reads it"] --> P
  P --> F{"freshness"}
  F -->|"fresh"| D["a decision"]
  F -->|"stale"| X["STALE — never green"]
  P --> L["incident → prevention artefact"]
```

What this shows: the surface consumes, never computes; and the freshness check is what makes it safe for a machine to read.

The Delegation Line

The agent detects, drafts the incident record and assembles the timeline. **The human** declares severity, comms and the prevention. **The system** refuses a green reading from a stale signal, and refuses to close an incident without a promoted prevention.

---

6. Build the Delivery

Deliverables

Artefact	Shape	Where it lives
status surface	service · signal · freshness · owner · escalation	one page, generated
error budgets	per service, with the spend that triggers a decision	`docs/budgets.yaml`
incident playbook	severity · first response · comms · escalation	the Playbook
prevention catalogue	every post-mortem's prevention, and the artefact	`docs/prevention.md`
DR order	what returns first, and what may wait	the Playbook

Decisions

Choice	Default	What changes if you choose otherwise
the surface	one, reading each service's own signals	a parallel truth discovered under pressure
budgets or alerts first	budgets	an alert with no threshold to argue about
"degraded"	defined per service, before the incident	an argument during it

Choice	Default	What changes if you choose otherwise
the prevention	an artefact	a sentence everyone forgets

✅ Gates

- A signal **older than its freshness window reads STALE**, never green.
- An incident **without a promoted prevention** is open.
- A surface that **computes** a value its service owns is a finding.

Runbook

1. **Build the surface** from existing signals. *Checkpoint: every row has a freshness time.*
2. **Set budgets.** *Checkpoint: spending one triggers a named decision.*
3. **Write the playbook** and define "degraded". *Checkpoint: a caller sees what degraded means.*
4. **Run one incident end to end.** *Checkpoint: severity, comms and escalation are all exercised.*
5. **Land its prevention** as an artefact. *Checkpoint: the prevention fails on the original defect.*

🎯 Proof

- A stale signal that says so.
- An incident whose prevention fails on the original defect.
- A surface that regenerates without a meeting.

---

7. 🔑 Chapter Summary

One surface, consuming what the services already say, with freshness as a first-class reading.

1. **The surface consumes, never computes** — a parallel truth disagrees exactly when it matters.
2. **Budgets make reliability a decision**; "degraded" must be defined before the incident.
3. **System, feature and portfolio health are different** — say which one is being reported.
4. **An incident sized for one operator**, and a post-mortem that promotes a prevention into an artefact.
5. **Every system that holds first-party data has a copy and a proof** — one suite copy, a per-schema restore, a freshness signal on the surface, and two numbers nobody has to guess.
6. **The agent reads the surface; freshness is what makes that safe.**

Chapter 29: The Fleet

Cattle, Identity, and Consented Support

The surface shows services, and underneath it are machines — this chapter makes them a fleet with identities and records rather than pets.

---

1. Machines Are Not Pets

The three structures run on machines somewhere. A fleet is not a set of servers you tend; it is a set of nodes you **replace**. The moment a machine is named, hand-configured and remembered, it has become a pet — and a pet is a machine nobody can rebuild.

Cattle, concretely:

Property	What it means
enrolment	a bare machine becomes a node in one idempotent command
identity	the node has its own identity, revocable in one command
configuration	everything comes from the repository, nothing from memory
reimage	the node is replaced, not repaired

What must survive a rebuild: data that is not on the node, configuration that is in the repository, identity that can be reissued. **What must not:** anything a human typed into it.

🎯 What You'll Learn

- What enrolment means when a machine is disposable
- Why node identity is the precondition for everything else
- Consent-first support as a record, not a courtesy

Roles as Capabilities

A node is not “the database server”; it declares a capability — support, storage, compute — and the fleet schedules by capability. Optional infrastructure must never be a dependency: a workstation-class node can vanish without taking the estate with it.

Pooled capacity, simple replication, shared pools — never per-contributor silos. A silo is a pet with a person’s name on it.

✅ What to Check

- Name a machine you would not rebuild today. That is a pet.
- Does a node declare capabilities, or a role someone typed?

---

2. Enrolment and Identity

Command	What it does	The checkpoint
enrol	bare machine → node, idempotent	run twice, one node
revoke	node identity → invalid, one command	the node cannot rejoin

Identity is the precondition. Without it, the fleet has no way to refuse a node, and support has no way to name who was helped.

Capacity and cost telemetry come from the fleet itself: who is using what, and what it costs. A fleet you cannot measure is a fleet you cannot size.

✅ What to Check

- Can a node join without an identity? That is the gap.
- Is revocation one command, or a procedure with a checklist?

---

3. Consented Support

Remote support on someone else’s machine is the most sensitive operation in this book. Consent-first means three things, all recorded:

Element	The record
visible consent	the person saw the request, and approved it
audit trail	who connected, when, for how long, to what
partner isolation	the session could reach their scope, and no further

A support flow without a record is a courtesy. A support flow with a record is a control — and the difference is what a client is buying when they ask *“who can see our data?”*.

The Support Record, as a shape

```
support_session:
  requested_by: <who>
  consented_by: <the person, at what time>
  scope: <what was reachable>
  duration: <from, to>
  actions: [<the changes made>]
  isolated: true
```

✅ What to Check

- Can you reconstruct the last support session from a record?
- Could a session reach beyond the partner’s scope? That is the isolation test.

---

4. Degradation

A fleet loses nodes. The question is not whether, but **what still works**:

- which services degrade, and how;
- which data remains readable;
- which operations refuse, rather than corrupt.

The degradation table is written per node class, and rehearsed by killing a node. A table nobody has tested is a hypothesis about resilience.

✅ What to Check

- Kill a node (in a drill). Does the estate behave as the table says?
- Does anything refuse rather than corrupt?

---

5. The AI Delta

An agent may reimage a node, so enrolment must be identity-bound and revocation must be one command.

Support consent has to be a machine-checkable record rather than a conversation, because **an agent can request a session and an agent can grant one** if nothing stops it.

> 💰 **Cost Check**: a pet machine costs the afternoon you cannot rebuild it. A support session without a record costs the client question you cannot answer. A fleet without a degradation table costs the surprise on the day a node dies. Enrolment and identity cost a day and make the rest routine.

```
flowchart LR
  B["bare machine"] --> E{"enrol — idempotent"}
  E -->|"identity issued"| N["node"]
  N --> C["capability declared"]
  N --> S["support — consented, recorded"]
  N --> R{"revoke"}
  R -->|"one command"| X["cannot rejoin"]
```

What this shows: the node's life is two commands and a record — no step depends on remembering what a person typed.

The Delegation Line

The agent enrols nodes, reports capacity and requests support. **The human** consents to the session and approves the lifecycle. **The system** refuses a node without an identity, and a support session without a recorded consent.

---

6. Build the Delivery

Deliverables

Artefact	Shape	Where it lives
enrolment recipe	bare machine → node, idempotent	`fleet/enrol/`
node identity	per node, revocable, audited	the identity store
support flow	consent · record · isolation	the Playbook
capacity report	who uses what, and the cost	generated
degradation table	what works when a node is gone	the Playbook

Decisions

Choice	Default	What changes if you choose otherwise
reimage or configure in place	reimage, because it is testable	a node with a history nobody can reproduce
support by session or proxy	session, with consent recorded	a channel with no attribution
pooled or dedicated	pooled, with declared exceptions	silos per contributor
capability or role	capability	a node whose purpose is a name

✅ Gates

- A node **without an identity cannot join**.
- A support session **without recorded consent** is refused.
- A fleet change that **breaks the degradation table** is a finding.

Runbook

1. **Write the enrolment recipe.** *Checkpoint: run twice, one node.*
2. **Issue node identities.** *Checkpoint: a node without one cannot join.*
3. **Implement consented support.** *Checkpoint: a session without consent is refused.*
4. **Emit capacity telemetry.** *Checkpoint: cost per node is answerable.*
5. **Kill a node** and read the degradation Map. *Checkpoint: the behaviour matches the table.*

🎯 Proof

- A node replaced in one command.
- A support session an auditor can reconstruct.
- A node loss that degrades as documented.

---

7. 🔑 Chapter Summary

A fleet is disposable machines with identities and records — not servers you tend.

1. **Cattle means enrolment, identity, configuration from the repository, and reimage** — never a named machine.
2. **Capabilities, not roles**; pooled capacity, never per-contributor silos.
3. **Consent-first support** is a record — visible consent, audit trail, partner isolation.
4. **The degradation table** is written and rehearsed, or it is a hypothesis.
5. **The agent enrols and requests; the human consents; the system refuses identity-less nodes and unconsented sessions.**

Chapter 30: Measurement and Economics

Six Numbers, or an Honest Gap

Everything so far has a cost and a claim; this chapter reduces both to six numbers with sources, or to one honest gap.

---

1. What to Measure

The three structures cost money and time; measurement is how you know whether they are working. It is also the chapter most easily ruined, because an agent inflates every volume metric it touches.

The six:

Metric	The question	The source
lead time	change opened → change live	the log + the tag
rework	how often work is redone	the log's retries and reversals
gate escapes	defects found after the gate	incidents + the log
cost per service	what it costs to run	the fleet + the spend record
drill freshness	when the last drill ran	the drill records
onboarding time	a new person or agent to first change	the log

What never to measure: lines, commits, tickets, "AI adoption %". Every one of them is a number a machine can inflate, and inflating it changes nothing except the number.

🎯 What You'll Learn

- The six metrics, each with its source
- Why review load matters more than review speed
- How to publish a gap instead of inventing a number

✅ What to Check

- Does each of your six have a named source? A metric without one is a claim.
- Which of your current metrics could be inflated by an agent? Those are the ones to drop.

---

2. The Bottleneck Is Review

The measured bottleneck in the AI era is **review, not writing**. A large organisation throttled its own AI change generation to protect its reviewers — and approval rates rose as scrutiny fell, which is the failure this chapter exists to catch.

So measure **review load and assurance**, never speed alone:

Measure	What it tells you	What it hides
review **load**	how much is waiting for a human	nothing — it is the queue
review **assurance**	how much of a diff was actually read	speed, which can look like throughput
approval **rate**	how often a change passes first time	a rate that rises when scrutiny falls

A faster review that reads less is the failure, and it improves every number on a naive dashboard.

✅ What to Check

- Do you measure review load? If not, the queue is invisible.
- Has any review metric improved while the review got shallower? That is the trap.

---

3. The Economics

Cost per service is the number that makes a portfolio decision possible: what this service costs to run, and what the headroom is. A model that ignores headroom is a model that plans a failure.

Cost	Where it comes from
compute and storage	the provider's bill, per service
egress and storage moves	the bill, and the architecture
the agent's own spend	the log, per task
headroom	the capacity you keep for a bad day

Chargeback and showback matter for a consulting practice: what a sovereign service costs to run is a price you can defend, and a number you cannot produce is a price you cannot quote.

The honest stop. This chapter must publish **your** numbers, or it is theory. If the numbers do not exist yet, the chapter says so, names the shortlist item that would produce them, and stops there.

The Honest Gap

This book has not yet run a measurement window. So the chapter ships with the definitions, the sources and the reader — and one declared gap:

Metric	Status	What would close it
lead time	**defined, not measured**	a period of releases with the log and tags
rework	**defined, not measured**	retries and reversals counted from the log
gate escapes	**defined, not measured**	incidents traced to a gate that passed
cost per service	**defined, not measured**	the fleet report, per service, for a month
drill freshness	**defined, not measured**	the drill records, once drills run on a schedule
onboarding time	**defined, not measured**	a new agent's first change, timed
review load	**defined, not measured**	the review queue, sampled over a week

A number invented here would be worse than the gap: it would end the book's credibility in the first table a reader checked.

✅ What to Check

- Does each metric name its source, or its intention?
- If you cannot measure it yet, is the gap written where a reader can see it?

---

4. The AI Delta

Agents **inflate volume**, and every volume metric flatters an agent: lines, commits, tickets, "adoption".

The numbers that survive are the ones a machine cannot manufacture — **review load and assurance, gate escapes, rework, drill freshness**. A number an agent produced without a named source is a claim wearing a decimal point.

> 💰 **Cost Check**: an unmeasured estate costs the decisions made on impressions — capacity bought late, rework invisible, review queues discovered by a missed release. The measurement window costs a period of collection and produces six numbers you can defend. The gap costs nothing and is honest; an invented number costs the book.

```
flowchart LR
  L["the log"] --> S["six metrics"]
  F["the fleet"] --> S
  D["drill records"] --> S
  S --> W{"a measurement window"}
  W -->|"ran"| N["publish the numbers"]
  W -->|"has not run"| G["declare the gap<br/>and the shortlist item"]
```

What this shows: both endings are acceptable; only an invented number is not.

The Delegation Line

The agent assembles the numbers from the log and names each source. **The human** publishes with the window and the caveat. **The system** refuses a metric without a source, and a published number without its window.

---

5. Build the Delivery

Deliverables

Artefact	Shape	Where it lives
metric definitions	six metrics, each with a source	`docs/metrics.yaml`
review-load measure	load and assurance, from the log	the measurement script
cost model	per service, with headroom	`docs/costs.yaml`
measurement window	the period, and the caveat	the published report

Decisions

Choice	Default	What changes if you choose otherwise
which metrics	the six, no more	a dashboard nobody reads
measurement source	the log	a separate tool that can disagree
publish or declare	declare the gap when unmeasured	a number that ends the book's credibility
review measure	load and assurance	speed, which improves as review weakens

✅ Gates

- A metric **without a named source** is refused.
- A published number **without its window and caveat** is refused.
- A **volume metric** is refused outright.

Runbook

1. **Define the six**, each with a source. *Checkpoint: no metric without a source.*
2. **Build the reader** from the log. *Checkpoint: it produces the six from records, not from estimates.*
3. **Run one measurement window.** *Checkpoint: the period is stated.*
4. **Publish with the caveat** — or publish the gap. *Checkpoint: an unmeasured metric is labelled, not filled.*
5. **Retire any metric nobody acted on.** *Checkpoint: the list stays at six.*

🎯 Proof

- A window of real numbers — or a declared gap with its shortlist item.
- A review-load figure that changed a decision.
- A metric list that stayed at six.

---

6. 🔑 Chapter Summary

Six numbers with sources, or one honest gap — never a number you cannot defend.

1. **Six metrics**: lead time · rework · gate escapes · cost per service · drill freshness · onboarding.
2. **Never volume**: lines, commits, tickets and "adoption" flatter the machine that produces them.
3. **The bottleneck is review** — measure load and assurance, never speed alone.
4. **The economics include headroom**, and a price you cannot produce is a price you cannot quote.
5. **Publish the gap when you have not measured** — this book does, and names what would close it.

Conclusion: What This Buys, and What It Costs

The Catalogue, Walked Honestly

The book ends where it started — the three structures — but walked as a catalogue of abilities, each with its evidence or its declared gap.

> The payoff and the honesty test in one page: every ability is either evidenced or named as unmeasured, and the reader leaves with the smallest first move.

1. The Arc, in One Page

The three structures — the prompt, the engineering, the workflow — control quality because each holds a different part of it. The split that keeps them honest is **rule / mechanism / view**: a rule lives in the repository and is reviewed; a mechanism is adopted and never rebuilt; a view renders state and decides nothing.

From there the book did one thing four times at increasing scale: **one repository** (a Law and a gate), **one system** (six concerns and an assembly), **a portfolio** (a manifest and a delivery path), **the organisation** (governance, operations, measurement). The claim never changed — the structures decide which effect you get — and each part showed what the claim costs to hold.

2. The Catalogue, Walked

Ability	The mechanism that produces it	The evidence that proves it	Status
Correct	the placement doctrine and the gates (Ch3, Ch19)	a gate that fails on a planted defect; a contract test at the seam (Ch15)	**mechanism named, evidence is per-system**
Verifiable	the gate estate (Ch19) and the evidence layer (Ch17)	the gate table with a liveness line per gate; a signal sharing the gate's logic	**evidence exists when the gate table does**
Maintainable	AI-readable code and the conventions file (Ch4)	the architecture test failing on a planted import; the fresh-context test	**mechanism named, evidence is per-system**

Ability	The mechanism that produces it	The evidence that proves it	Status
Auditable	the nine-field log (Ch10) and the evidence pack (Ch24)	a log that matches the gate records; a pack a reviewer reads cold	**evidence exists once the pack is generated**
Secure & private	the escalation catalogue (Ch8), least privilege (Ch13, Ch26)	a blocked record for each condition; a hook refusing what prose allowed (Ch27)	**mechanism named, evidence is per-system**
Usable	the command surface and its help (Ch3)	help rendered from the registry, refusing undeclared verbs	**evidence exists when the surface is built**
Reliable & available	the signal table and freshness (Ch17), budgets (Ch28)	a stale reading that says STALE; a restore drilled (Ch18)	**mechanism named; drills are the evidence**
Operable & diagnosable	the status surface (Ch28) and the incident playbook	an incident whose prevention landed as an artefact	**evidence exists after one incident**
Scalable & affordable	the manifest and the fleet (Ch20, Ch29)	cost per service, from the fleet report — **not yet measured**	**gap**
Portable & interoperable	the seam contracts (Ch13), the portability check (Ch16)	the lock-in check failing on a planted violation; one exit drill	**evidence exists once one drill runs**

Two rules keep this table honest, and they are the same two the book opened with: **an ability without evidence is a claim, not a property** — and an ability that cannot be attributed to a structure is a wish.

What This Catalogue Shows

Six of the ten have a **named mechanism and a checkable artefact** in this book's own estate. Two are evidence-per-system: the mechanism is defined and the proof is whatever your gates produce. **One is a declared gap** — *Scalable & affordable* — because cost per service has not been measured here, and the chapter that defines it says so.

That is the honest reading, and it is deliberately boring: the catalogue's value is not that all ten are green, it is that **each row says what would make it green.**

3. The AI-Age Claim, Stated So It Survives

The defensible claim is **setting-dependent**: AI raises throughput *and* instability together, and the structures — repository, gates, delivery — decide which effect you get. That is supported by the evidence base the plan records: DORA's version-control and change-management findings, the review-bottleneck observation, the platform-quality argument.

The indefensible claim is *"this method produces good software"* as a controlled result. No study isolates these structures as a treatment, and none measures this book's own method. So the book says what it can: **the structures decide which effect you get**, and the evidence for that is the review bottleneck and DORA's moderation analysis — never a controlled trial that does not exist.

4. The Cost, Plainly

- **Review capacity** — the binding constraint. The gates do not reduce the reading; they focus it. A reader who does not review is not saved by a gate.
- **Rehearsals and drills** — restore, exit, rollback, upgrade. Each is a day, and each is the price of the ability it proves.
- **The platform** — a manifest, a control plane, a status surface. It must exist before it pays for itself, and that is a real delay.
- **Time to the first gate** — the first week buys the least visible work.

> 💰 **Cost Check**: the book's honest accounting is that **the structures are cheaper than the failures they prevent and more expensive than doing nothing**. A reader who wants the second comparison should not adopt this method, and a reader who wants the first should start with one repository.

5. Monday

1. **One repository**, with a real Law: the rules that must hold, in files that are reviewed.
2. **One real gate** — a check that can fail, wired where a change is refused — not a scan that prints.
3. **One measurement**: how long a change takes from opened to live, and how often it is reworked. Two numbers, one week.

That is the whole first move. The rest of the book is that move, repeated at increasing scale, with the vocabulary to keep it coherent.

The reader who does only this has the method. The reader who does all of it has the method *and* the evidence — and knows, per ability, which one they are holding.

6. The Last Thing

Every chapter in this book has ended with something to build, and this one ends with a test rather than an artefact:

Can you name, for your own system, which abilities you hold evidence for, and which you do not?

If the answer is a table like the one above — with a declared gap where the gap is — the book has done its job. If the answer is "all of them", the honest reader will ask which evidence they are thinking of.

The DevOps Engineer's Guide to Effective AI Usage — Becoming a Software Orchestrator

A structured guide to using AI for infrastructure, deployment, governance, and automation without losing engineering discipline.

```
<a class="md-button md-button--primary" href="guide/01-introduction/">Start with Chapter 1</a>
<a class="md-button" href="aibook.pdf">Download the PDF</a>
  <a class="md-button" href="aibook.epub">EPUB</a>
  <a class="md-button" href="guide/02-self-improving-repository/">Chapter 2</a>
```

This site is the public home of the book. **Chapter 1 sets the thesis; Chapter 2 builds the repository that makes it work.**

What this book is

AI quality is not decided by the model. It is decided by the structures around it — and the book names three of them: the **Prompt** structure, the **Engineering** structure, and the **Workflow** structure. The claim it argues is that the third is what makes the first two durable.

It is written for engineers who own infrastructure and delivery, not for readers looking for a tool tour. Every chapter is tool-agnostic, and every chapter ends in something you can check.

The four parts

The book is organised in four parts. Part One sets the thesis and builds the repository that holds it. Part Two is the workbench: where each control belongs, and what delivery work looks like when an agent does it. Part Three takes it outward: what you do not own, decided from inside the repository. Part Four is the delivery system: the same patterns at portfolio scale, and the mechanism that makes them safe to delegate.

Review status. Parts One and Two are reviewed. Parts Three and Four are written and are being reviewed chapter by chapter — some have been through review, others are still first drafts, and a comment on either is welcome. Send review comments through the contact page.

Part One — Foundations

#	Chapter	What it establishes
1	The AI-Powered DevOps Engineer	The thesis: three structures control AI quality, and the role moves from DevOps engineer to software orchestrator
2	The Self-Improving Repository	The repository that holds those structures — the Law, the Map, the Playbook, real gates, and one command surface humans and agents share

Part Two — The Instrument

8 chapters · reviewed · start at Chapter 3 →

The instruments: where each control belongs, and what delivery work looks like when an agent does it.

- **Foundations of control** — what lives in the repository and what lives in the tool, and why a tool feature is never a substitute for a rule
- **The six concerns in depth** — context routing · session continuity · the agent's procedure · escalation policy · context economy · execution transparency

Part Three — The Boundary

6 chapters · start at Chapter 11 →

The workbench turns outward, to what you do not own — and every decision about it is still a file in the repository:

- **Assembling a real system** — build, buy or assemble; the decision record and its exit plan; upstream products as dependencies; provider seams; contracts that keep services independent
- **What holds the boundary** — sovereignty, portability, and the exit drill you have rehearsed, not the one you plan to
- **The integration charter** — the nine decisions that must exist before an agent touches anything you do not own: identity and ownership · structure and address · change and proof

Part Four — The Guarantees

3 chapters · start at Chapter 17 →

Three things one system must be able to do before it is asked to scale:

- **Show it** — observability as the evidence layer: every invariant, one check logic, two consumers
- **Survive it** — recoverability: a copy, a restore you have run, and the RPO/RTO you can state
- **Refuse it** — the quality gate: a designed set of gates, each declaring its invariant, scope, verdict and cost

Part Five — The Delivery System

12 chapters · start at Chapter 20 →

One repository becomes many — and the whole mechanism becomes one system:

- **Version control as the substrate** — what must be a reviewable, versioned file: rules, manifests, registries, policy and contracts, with tags as the release boundary and provenance for every change
- **The control plane as a product** — one manifest, plus registries, validators and audits; the service catalogue generated from the manifest rather than maintained twice — including the platform charters, which are data before they are policy
- **From commit to production** — versioning for artefacts *and* contracts, release trains and evidence packs, and a rehearsed rollback for when it goes wrong
- **Governance machines enforce** — policy as code, codified standards across repositories, access tiers, agent identity, runner trust
- **The agent plane at scale** — context contracts, deterministic hooks, evals in CI, trust boundaries
- **Portfolio operations** — one status surface over many services, incident response, the fleet, measurement and economics

How to read it

1. **Chapter 1 first.** It is the mental model the rest of the book assumes — decide whether you agree with it before judging the method.
2. **Chapter 2 next.** It builds the repository that holds the model, file by file: the Law, the Map, the Playbook, gates that can fail, and a command surface humans and agents share.
3. **Then practise on one real task.** Pick one piece of work you already own, apply Part One, and let the repository's gates tell you whether it worked.

The Implementation Pack

The book explains the loop. **The pack is the loop's paperwork** — the five files a team needs to run it themselves, written to be filled in rather than admired:

File	What it is
`charter-template.md`	the nine values, each with the check that consumes it
`registry-shape.tsv`	seven columns, with the rules in comments
`gate-table.csv`	what each gate proves, and where it runs
`seven-lie-audit.md`	seven questions, each with the evidence that answers it
`manifest.json`	a sha256 per file, plus the commit and the date

The book stays free. The pack is for the work. It is served to a client account — the same door you use elsewhere on this site — so downloading it takes one sign-in and no second identity.

Open the pack Download the book

!!! note "Why it is not a free download"

```
A template nobody signs for is a template nobody adopts. The sign-in is what lets us tell you when the sheet changes —
and it is the same account you already have, not a mailing list.
```

What it is not

- **Not the private edition.** That was retired in November 2026; the pack replaced it.
- **Not a course.** There are no videos and no modules — five files, the same ones this repository uses.
- **Not a subscription.** One artefact, dated by its manifest.
