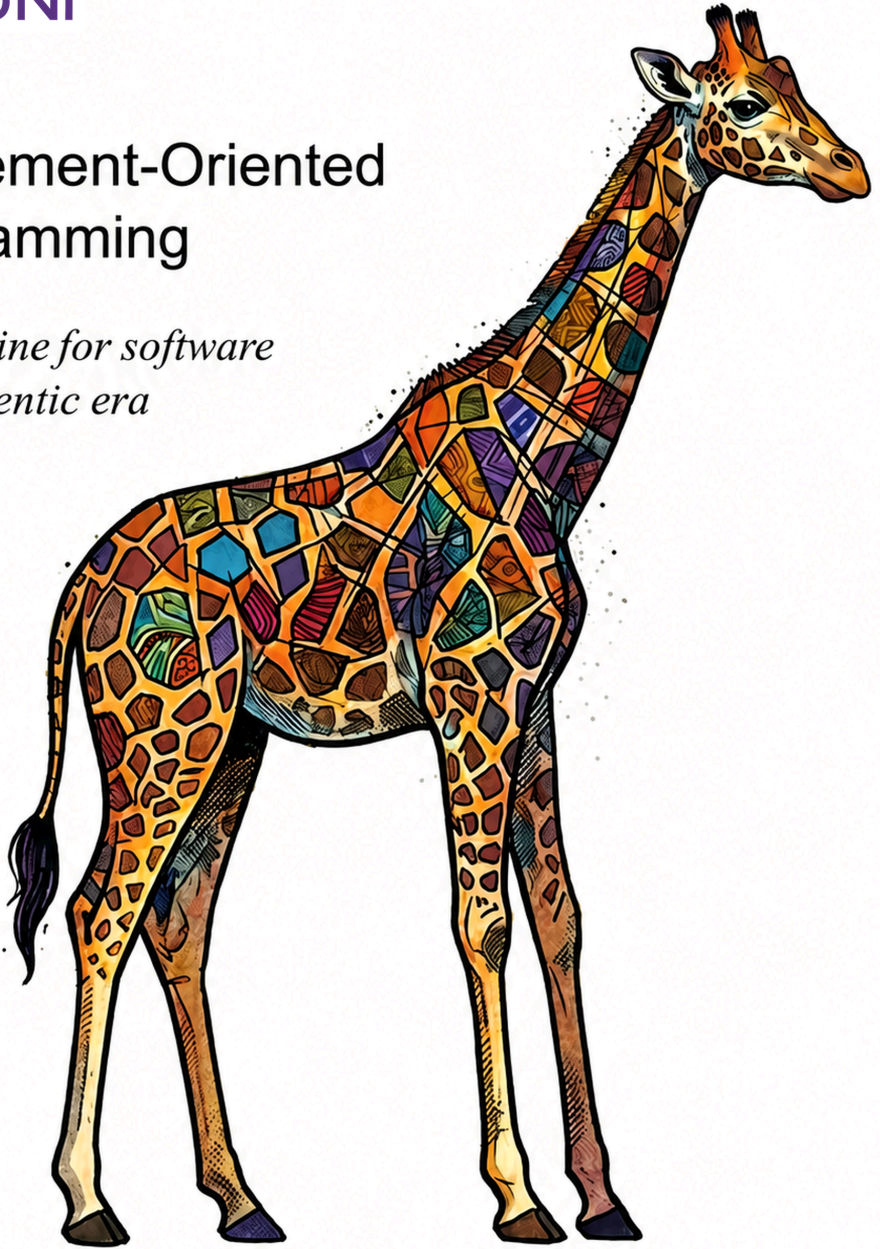


AYOUNI

# Refinement-Oriented Programming

*A discipline for software  
in the agentic era*



MANSOUR AYOUNI

TUNIS · NIAMEY · PARIS · 2026



# **Refinement-Oriented Programming**



# Refinement-Oriented Programming

*A discipline for software in the agentic era*

Mansour Ayouni

AYOUNI

TUNIS · NIAMEY · PARIS · 2026

© 2026 Mansour Ayouni

Published by AYOUNI, the independent imprint of Mansour Ayouni / Softanza Labs. Tunis · Niamey · Paris.

The text of this book is released under Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0). Illustrations are released under CC BY-NC-SA 4.0 unless otherwise noted. Terminal sessions and code fragments are released under the MIT License.

The free digital edition is at <https://mayouni.github.io/ayouni/books/refinement/en/>. Paper copies in quantity are supplied to organisations on request: <https://mayouni.github.io/ayouni/offer/>.

First edition: 2026 — the Teaching Edition. Typeset in EB Garamond, Inter, and Archivo Black. Cover illustration: a giraffe in patchwork coat, for Niger. [Artist credit to be added.]

ISBN: pending.



*For the compliance officers, programmers, and auditors of regulated banks across the WAEMU region, who knew the discipline this book argues for before they had a name for it.*

—MA · Niamey · 2026



*An artisan refines until the work is finished. A regulator  
asks how the work was refined. A programmer answers.*

— traditional adage, attribution lost



# Contents

|                                              |     |
|----------------------------------------------|-----|
| Start here .....                             | 15  |
| <b>PART I — THE JUNGLE, AND THE WAY OUT</b>  |     |
| 1 · The jungle .....                         | 19  |
| 2 · What broke .....                         | 27  |
| 3 · The unit changed .....                   | 33  |
| <b>PART II — THE DISCIPLINE</b>              |     |
| 4 · The graph .....                          | 43  |
| 5 · The gate .....                           | 49  |
| 6 · The cascade .....                        | 55  |
| 7 · Reversibility and the audit chain .....  | 61  |
| <b>PART III — AGENTS IN A GOVERNED WORLD</b> |     |
| 8 · Agents, demystified .....                | 69  |
| 9 · The safest write surface .....           | 75  |
| 10 · The jungle, revisited .....             | 81  |
| <b>PART IV — THE BIGGER PICTURE</b>          |     |
| 11 · One gate, many worlds .....             | 89  |
| 12 · In the boardroom .....                  | 93  |
| 13 · Honest boundaries .....                 | 97  |
| The next letter .....                        | 101 |
| The words, pinned .....                      | 103 |



## FRONT MATTER

# Start here

---

*The whole book in two minutes, and the map of the road ahead.*

---

This book makes you one promise. By its last page, the most confusing vocabulary in modern software — agents, memory, RAG, skills, harnesses, governance — will have settled into one simple picture. And you will know a discipline that lets you accept software changes at machine speed while keeping the right to ask, of any change: *who made you, were you checked, and how do I undo you?*

The discipline is called **refinement-oriented programming**. The name will make sense by Chapter 3, where you will run your first governed change end to end — propose it, preview everything it touches, watch four checks pass, land it, and hold its undo in your hand. Everything after that is unpacking what you will already have held.

You do not need to be a programmer to read this book. The argument runs on one story — a small commercial bank in Niamey, a regulator’s letter, and a methodologist named Amina Bello — and every technical idea is shown before it is named. If you *are* a programmer, you will find the terminal sessions and the mechanics precise enough to act on. Both readers get the same book; they just linger on different pages.

**A note on the terminal sessions.** The sessions in this book are set in an `stz` command idiom and show the machinery of the open-source **Softanza** foundation — the engine and library this discipline runs on, free, local, with nothing to buy. The machinery is real: the typed refinement points, the four-stage gate, the computed cascade, the reversible history you will meet are all defined by that foundation. The command names and flags are an idealized rendering, and may differ in detail as the tooling ships. Nothing shown goes beyond what the foundation defines. Commercial products are built on the same foundation —

**Refine**, an IDE for this discipline, is one — and when a picture teaches better than a terminal, this book shows an illustration of its screens. The discipline itself belongs to the open foundation, and to you.

**The map.** The book has four parts. *Part I* names the confusion honestly and finds the single question underneath it — then puts your hands on the answer. *Part II* takes the discipline apart, one mechanism per chapter: the graph, the gate, the cascade, the audit chain. *Part III* brings the agents in — and shows why a well-governed change does not care whether a human or a machine proposed it. *Part IV* widens the lens: beyond code, beyond one language, into the boardroom, and finally into the discipline's own honest limits.

**If you have ninety minutes**, read Chapters 1, 3, 5, 6, 10, and 13. That path carries the whole argument: the jungle, the new unit of work, the gate, the cascade, the jungle made legible, and what none of this can do.

**A word about honesty.** A book about trust earns its subject or it does not. Every chapter here ends by saying what it has shown and what it has not; the last chapter is nothing but limits. When the book does not know something, it says so.

...

## PART I

# The jungle, and the way out

---

If you have felt lost in the vocabulary of modern AI — agents, harnesses, memory, RAG, skills, MCP, governance — you are not behind. The vocabulary itself is the problem, and this part shows you the single question hiding underneath all of it. By the end of Chapter 3 you will have run your first governed change, end to end, and the rest of the book will feel like unpacking something you have already held.



## CHAPTER 1

# The jungle

---

*Every word in modern AI means three things. Underneath all of them is one question.*

---

In the spring of 2026, in a meeting room at Softibank Niger in Niamey, a vendor presented the future. The slides were handsome. They spoke of *agentic workflows* and *orchestration layers*, of *RAG pipelines* grounding *LLM agents*, of *guardrails* and *evals* and a *skills marketplace*, all connected over *MCP*. The bank's leadership nodded along. The future sounded expensive and near.

Amina Bello, the bank's methodologist, asked one question. "When your agent changes the penalty rate in our loan system — who checked that change, against which rule, and how do I undo it on Thursday when the regulator calls?"

There was a pause. The presenter talked about guardrails, then about evals, then about human-in-the-loop. All three answers were interesting. None of them was an answer. He had told her how the agent would be *filtered*, how it would be *scored*, and where a human would *watch* — but not what would happen to the change itself: the new number in the loan system, sitting there, unattributed, unproven, and hard to take back.

Hold on to that pause. This whole book lives inside it.

## §1 — TEN WORDS, THIRTY MEANINGS

Walk through the vocabulary the presenter used, slowly, and a pattern appears: every important word in modern AI means at least three things, depending on who is speaking.

An **agent**, in a research paper, is a program that perceives and acts toward a goal. In a product demo, it is a language model running in a loop with tools. In a boardroom, it is a synthetic coworker who will apparently do the work of four people. These are three different claims with three different consequences, wearing one word.

A **harness** is what an agent runs inside — except when it means the framework the agent was *built* with, except when a vendor uses it to mean the whole platform, governance included. **Memory** might be a vector database of embeddings, or a knowledge graph, or just the transcript of the current chat, and those three things behave nothing alike when you ask the only question that matters about memory: *what did the system know when it acted, and how do you know?*

**RAG** — retrieval-augmented generation — began as a precise technique: fetch relevant documents, let the model read them before answering. It now names almost any arrangement where a model consults anything. **Skills** are sometimes prompt files, sometimes plugins, sometimes formally verified capabilities; the word alone will not tell you whether a skill can be *trusted* or merely *invoked*. **MCP** is a protocol for connecting models to tools — and also, in looser speech, the tools themselves, and the ecosystem around them.

**Guardrails** are content filters, unless they are policy engines, unless they are a second model watching the first. **Evals** are rigorous benchmark suites, unless they are three test prompts and a good feeling. **Orchestration** is pipelines, or fleets, or simply a bigger loop around the smaller loops.

And **governance** — the word this book cares most about — is the worst of all. It can mean governing the *model* (what it may say), governing the *actor* (what an agent may reach and do), or governing the *change* (whether the thing the agent just did to your system should stand). Three different objects. Three different disciplines. One word, used interchangeably, in contracts worth millions.

## §2 — WHY THE JUNGLE GREW

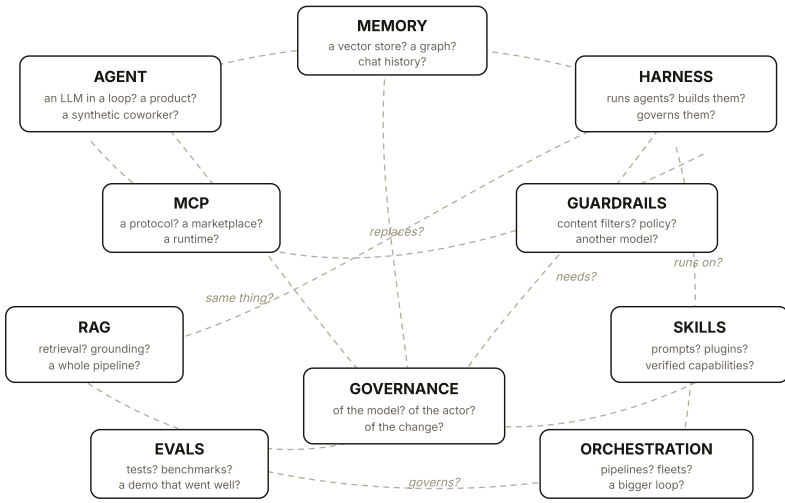
It is tempting to blame marketing, and marketing has not helped. But the deeper cause is more forgiving: the field moved faster than its language could settle.

Each of these words was minted in a real context, for a real idea. Then the field doubled, and doubled again, and every new tool needed to describe itself using words buyers already recognized. So the words stretched. Researchers, practitioners, and vendors now speak three dialects of the same vocabulary, and each hears the others through its own dictionary. Nobody is exactly lying. Everybody is approximately misunderstood.

This has a practical cost, and it is not confusion in the abstract. When two people agree that the bank should "deploy an agent with memory and guardrails under proper governance," they may be agreeing about *nothing at all* — each phrase resolves differently in each head. The project that follows inherits every unresolved ambiguity, and the invoice arrives regardless.

## §3 — THE MAP, AS IT STANDS

Here is the vocabulary drawn honestly — as a map of the jungle rather than a brochure of it.



*The question none of the boxes answers: can you trust what just changed?*

**Fig.** *The AI vocabulary in its natural state. Every box is a real idea; every box also carries the two or three other things people mean by the same word. Note the arrows: nobody can say with confidence which box contains, replaces, or requires which. And note the question at the bottom — the one no box answers.*

Spend a moment with the arrows. Does the harness contain the agent, or run beside it? Is RAG part of memory, or is memory part of RAG? Do guardrails govern the agent, or the model, or the output? The honest answer to each is *it depends who you ask* — which is the map's real finding. The boxes are all real ideas. The **relations between them** are folklore.

A map like this cannot be fixed by better definitions alone. Standards bodies have tried; glossaries have tried; this book will offer its own pinned vocabulary in the appendix, and that helps. But the jungle has a root, and until the root is named, every cleared path grows over again.

## §4 — THE QUESTION UNDERNEATH

Go back to Amina's question in the meeting room, because it was not really a question about agents.

*Who checked this change, against which rule, and how do I undo it?*

Now hold each jungle word up against that question, and watch what happens.

**Memory** answers a fragment: *what did the system know when it acted?* **Evals**

answer a fragment: *how good is this model, usually, on average, elsewhere?*

**Guardrails** answer a fragment: *what will we refuse to let it say or do?* **Governance**,

in its usual senses, answers fragments: *what may this actor reach? what has the*

*organization declared about AI use?* **Orchestration, harness, MCP** — these

answer *where and how the work runs*.

Every box on the map is a partial answer to a trust question. And not one of them answers the trust question that actually lands on a desk — because that question is never about the agent in general. It is about *one specific change*: the penalty rate that is different this morning. The report column that moved. The pricing rule that an agent, or a contractor, or a colleague at two a.m., altered — competently or not, authorized or not, reversibly or not.

*Can you trust what just changed?* Everything in the jungle circles this question. Nothing in the jungle answers it, because each box governs something *around* the change — the model's mouth, the actor's reach, the pipeline's shape, the organization's posture — while the change itself, the actual difference in the actual system, passes through ungoverned.

This is not a small gap at the edge of the picture. It is the center of the picture, unoccupied.

## §5 — THE WAY OUT, PREVIEWED

The way out of the jungle is not an eleventh box. It is a change of object: stop trying to govern the *intelligence*, and govern the *change*.

That is the whole move, and the rest of this book is that move taken seriously. A discipline — refinement-oriented programming — in which every consequential change to a system, whoever authors it, becomes a **refinement**: a typed, bounded proposal that must pass a gate of declared checks before it lands; that carries its author, its reason, and its blast radius; and that can be undone, atomically, because its undo was computed before it ever landed.

Three things about this discipline, up front.

First, it is **not another AI tool**. It governs changes from an AI agent exactly as it governs changes from a careful human, an inexperienced intern, or a script written in 2011. That indifference is the point. The gate does not care who is knocking.

Second, it is **free to practice**. The machinery this book teaches — the graph, the refinement points, the gate, the cascade, the audit chain — belongs to an open-source foundation, the Softanza engine and library. You will drive it from a terminal in Chapter 3. Commercial products build finer rooms on this ground; the ground itself is common.

Third, it is **small enough to learn**. The jungle is confusing because it is wide. This discipline is narrow: four mechanisms, which you will meet one at a time in Part II, each in one chapter, each shown before it is named.

In Chapter 10, this same map will be drawn again — every box in its place, every arrow labeled, every word carrying one meaning. The book will have earned that map by then. The jungle does not get simpler by being described more cleverly. It gets simpler when you stand somewhere solid and look back at it.

## WHAT YOU NOW KNOW

- Every load-bearing word in modern AI — agent, harness, memory, RAG, skills, guardrails, evals, governance — carries two or three incompatible meanings, and projects inherit the ambiguity.
- The jungle grew honestly: the field outran its language, and every tool stretched familiar words to describe itself.

- Each jungle word answers a *fragment* of a trust question — about the model, the actor, or the pipeline — while the concrete question, *can you trust what just changed?*, goes unanswered, because the change itself is the one object nothing governs.
- The way out is a change of object: govern the change. That discipline is this book's subject.

*What this chapter has not shown: that governing the change is actually possible, at machine speed, without drowning the people responsible. That demonstration begins in the next chapter — with the morning it became necessary.*



## CHAPTER 2

# What broke

---

*Writing code stopped being the bottleneck. Trusting it became the bottleneck.*

---

On a Tuesday in September, a letter arrived at Softibank Niger — paper, stamped, from the *Banque Centrale des États de l'Afrique de l'Ouest* in Dakar. BCEAO is the central bank of the eight countries of the West African Economic and Monetary Union; when it issues a circular, every commercial bank in the union adjusts. This one was routine, as regulation goes: under a revised directive, the cap on late-payment penalties was to drop from 18 percent to 16 percent. Compliance within ninety days.

Amina Bello read it twice, then opened a notebook and started listing what the number touched. The penalty cap was not one line in one file. It fed the pricing of fourteen loan products. It appeared in the risk-assessment module, in the quarterly compliance report, in the statement generator that printed customer letters, in the SMS templates that warned late payers, and in a scattering of tests that asserted, in effect, *the world is 18 percent*. Some of these she knew. Others she would discover, she wrote dryly, "when they break."

Ninety days was comfortable. That is worth saying plainly, because it is about to stop being true: in the world where Softibank's systems were built, a change like this was a project, and ninety days was a reasonable size for it. Someone would write the change, someone else would review it, the test suite would run, a manager would sign, and the bank would file its compliance report with time to spare.

The letter was routine. What had quietly stopped being routine was everything after the letter.

## §1 — WHAT WAS NEW

By the time that letter arrived, a competent AI coding agent could draft the entire change — all fourteen products, the report, the templates, the tests — in an afternoon.

Sit with the sentence, because it is the hinge of this book. The drafting was not sloppy work. The current generation of agents writes plausible, idiomatic, often correct code, at a speed no team can match: not one change at a time but dozens in parallel, around the clock, at close to zero marginal cost. Between roughly 2022 and today, the production of code went from the most expensive step in software to nearly the cheapest.

Every earlier era of programming had the same bottleneck: writing was slow. Tools, languages, methodologies — decades of them — were shaped around that scarcity. Then the scarcity ended, in about two years, and an uncomfortable question surfaced in every serious engineering organization at once.

If the change can be written in an afternoon — *by something that is not accountable for it* — who makes sure it is right?

## §2 — THE SYMMETRY BROKE

The old workflow had a hidden symmetry: code was written at human speed and reviewed at human speed. One slow process fed another slow process, and the two kept rough pace. Nobody designed this; it fell out of the fact that both sides of the desk held humans.

Machine-speed writing broke the symmetry in one direction only. A reviewer in 2026 reads exactly as fast as a reviewer in 2019 — carefully, at prose pace, with judgment that cannot be parallelized. But the queue in front of that reviewer no longer arrives at human speed.

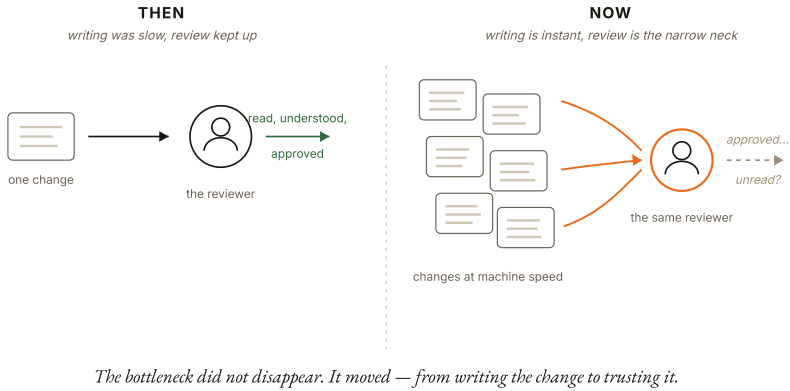


**Fig.** *The symmetry breaks. On the left, one hand-written change per week meets a reviewer with time to understand it. On the right, machine-speed changes pile into the same single reviewer — and approval quietly decouples from understanding.*

What happens next is well documented in every organization that has lived it, and it is not that review stops. It is worse: review continues, *formally*. The pull requests are opened; the approvals are clicked; the process is observed. But the reviewer facing forty machine-written changes cannot give each one the twenty minutes of comprehension the old symmetry assumed, so approval drifts away from understanding. The signature still lands on the page. It just certifies less and less.

At a bank, this quiet drift has a name and a consequence. When BCEAO's auditors arrive — and they do arrive — the question is not "was this change approved?" It is "*show me who approved it, what they checked, and what it affected.*" An approval that certified nothing answers none of that.

### §3 — THE INSTRUMENTS ASSUME A VANISHED WORLD

It is tempting to answer "review harder" — add reviewers, add checklists, slow the queue. Organizations try. It fails for an instructive reason: every instrument in the standard toolkit was designed for the world whose scarcity just ended.

The **diff** — the line-by-line comparison every code review stares at — shows which bytes moved. It was a brilliant fit for a world where a human wrote every line and a human could hold every line in mind. Show a reviewer fourteen products' worth of machine-written diff, and the diff faithfully displays everything except what she actually needs: *what does this change mean, what does it touch, and is it allowed?*

The **pull request** assumes changes are scarce and self-explaining — an author who can be questioned, a discussion thread with time to unfold. The **sign-off meeting** assumes changes are rare enough to schedule humans around. **Version control** remembers every change perfectly and evaluates none of them; it is a flawless historian with no opinions. Each instrument does exactly its job. The job description changed.

Here is the precise statement of what broke, and it is worth keeping: *the industry automated the writing of changes without automating the trusting of changes, and the instruments of trust were all designed for hand-written volume.*

## §4 — WHAT TRUST HAS TO MEAN NOW

If trust can no longer be produced by a human reading everything, what is it made of instead?

Not faith in the author — the author may be a model. Not vibes about the vendor. Not a green checkmark that certifies only that a button was clicked. For trust in a change to survive machine speed, it has to become a **property of the change itself** — something the change *carries*, that can be checked without re-reading the author's mind.

Concretely, five things, and this list is the skeleton key to the whole book. A trustworthy change is:

- **validated** — checked, before it lands, against rules that were declared in advance: bounds, policies, tests, invariants;

- **scoped** — its blast radius computed and visible, not discovered in production;
- **attributed** — carrying who proposed it, who or what authored it, and why;
- **reversible** — carrying its own undo, computed before landing, so that taking it back is one safe act rather than an archaeology project;
- **explainable** — able to answer, later, the auditor's question: *why was this allowed?*

Read the list once more, and notice what it never mentions: the author. Whether the change came from Amina, from a junior developer, from a script, or from an agent running at three a.m. — the five properties are checkable either way. That is what makes the list compatible with machine speed. Judgment of authors does not scale. Properties of changes do.

The old workflow produced these properties as a *side effect* of slow human review, unreliably, and lost them entirely when the queue sped up. The discipline in this book produces them *by construction* — every change, every author, every time — because they are built into the unit of work itself.

Which means the unit of work has to change. That is the next chapter, and you will not just read it. You will run it.

## WHAT YOU NOW KNOW

- Producing code stopped being the bottleneck within roughly two years; a regulatory change that was a ninety-day project can now be drafted in an afternoon.
- Review broke silently: it continues formally while approval decouples from understanding, because reviewing was never automated and cannot be rushed.
- The standard instruments — diff, pull request, sign-off, version control — each assume hand-written, human-scale change volume. They do their jobs; the jobs changed.

- Trust must become a checkable property of the change itself: validated, scoped, attributed, reversible, explainable — a list that never needs to mention who the author was.

*What this chapter has not shown: how a change could actually carry those five properties. That machinery is the rest of the book — starting now, in a terminal.*

## CHAPTER 3

# The unit changed

---

*From the line, to the diff, to the refinement. You will make one before this chapter ends.*

---

Every era of programming has a unit of work — the thing you make, review, and talk about. For decades the unit was the **line of code**: programming was typing, and a program was thousands of decisions frozen into text. Version control raised the unit to the **diff**: the set of lines that changed, which could be stored, compared, shared, and argued about. The diff is one of software's great inventions, and Chapter 2 showed its limit: it records which bytes moved and stays silent about everything a trusting person needs — meaning, permission, blast radius, undo.

This chapter introduces the third unit, the one this book is named for. Then, without further ceremony, you will make one.

## §1 — WHAT EACH UNIT CARRIES

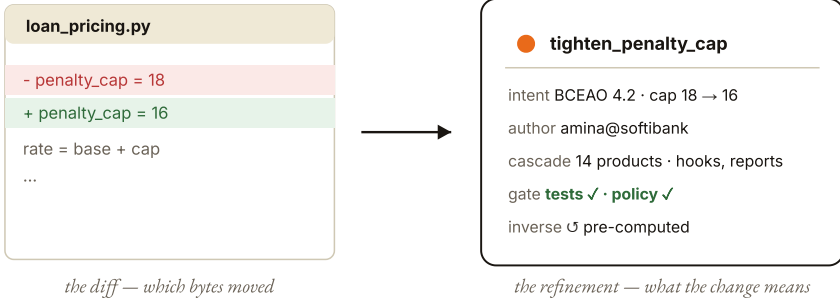


**Fig.** The same change as two different objects. The diff (left) records that bytes moved in a file. The refinement (right) is the change as a first-class object: its intent, its author, its computed blast radius, its checks, and its pre-computed undo — all in one governable unit.

Look at the two sides of the figure the way a bank auditor would. The left side proves that `penalty_cap` was 18 and is now 16. It proves nothing else — not who decided, not why, not what else in the system silently depends on that number, not how to get back. Every one of those answers lives *somewhere* — in a ticket, a memory, a meeting, a colleague on leave — which is to say, nowhere the system can vouch for.

The right side is the same change packaged as what it actually is: a decision with consequences. Intent, author, blast radius, checks, undo — attached to the change itself, not scattered around it. That package is a **refinement**, and the exact definition is worth stating once, cleanly:

*A **refinement** is a typed, bounded change to a declared point in a system, validated against declared rules before it lands, attributed to its author, aware of everything it affects, and reversible by construction.*

Two words in that definition do the heavy lifting. **Typed** means the change has a kind and a shape — “set this parameter to this value,” not “here are some new bytes.” **Declared point** means the system knows, in advance, *where* it can be changed: named, bounded knobs — a rate, a threshold, a strategy, an algorithm choice — that its authors marked as the intended surface of change. In the Softanza foundation these are called **refinement points**, and they come in eight kinds, from a single parameter up to a whole pipeline.

A system with declared refinement points is not a system you can only change in approved ways — files remain files; emergencies remain emergencies. It is a system that *knows its own knobs*, and can therefore govern the turning of them. How code comes to declare its points — new code by writing them, existing code by a process called the lift — is Chapter 4’s subject. Right now, something better is available: a system that already has them.

## §2 — YOUR FIRST SESSION

What follows is a terminal session with `stz`, the open foundation’s command line, on a small pricing module. Read it slowly — every line is about to be walked through.

```

stz — your first refinement

> stz lift pricing.py
lifted 214 nodes into the graph · parser: python (its own ast)

> stz points
vat_rate    param    0.18    bounds 0.00..0.25
rounding    algo     banker's  2 alternatives declared
penalty_cap param    18      bounds 10..20

> stz refine vat_rate --to 0.20
proposal rfn-0114 recorded · a typed proposal, not a diff

> stz cascade
3 nodes revalidate: pricing.quote · invoice.total · tests/pricing

> stz apply
structural  pass    python re-validated the file
constraint  pass    0.20 within declared bounds
derivation  pass    cascade verdicts unchanged elsewhere
governance  pass    author authorized for this point
landed. reversible · stz revert undoes it atomically

```

**Fig.** *Your first governed change, end to end: lift the code into the graph, list its declared points, propose a typed change, preview its blast radius, and watch it pass the four-stage gate — landing reversible.*

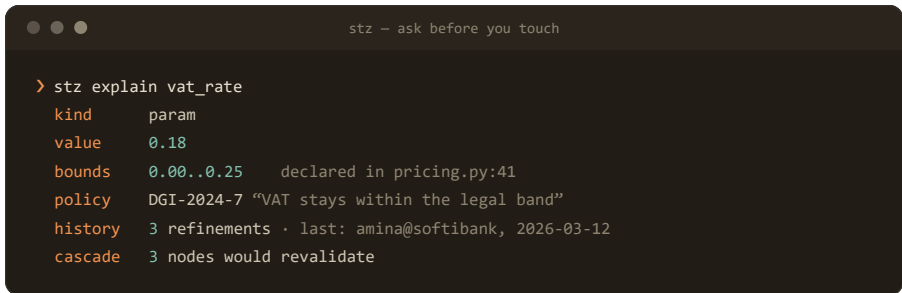
`stz lift pricing.py` is the entry move. The **lift** reads existing source code and represents it as a **graph** — functions, parameters, dependencies, tests, each a node, each connection an edge. Notice the note in its output: the file was parsed by *Python's own parser*. The foundation does not impose its private reading of your language; each language stays the authority on its own structure. The number that matters is the effect: 214 nodes now describe this module in a form the machinery can reason about.

`stz points` answers *where may this system be changed?* Three declared points come back, each with its kind and bounds: `vat_rate`, a parameter constrained to the band 0.00 to 0.25; `rounding`, an algorithm choice with two declared alternatives; `penalty_cap`, a parameter bounded 10 to 20. This short table is

something the diff era never had — a system's **declared change surface**, printed on demand. You are not spelunking through files wondering what is safe to touch. The system is telling you what it was built to have changed.

### §3 — ASK BEFORE YOU TOUCH

Before changing anything, ask the system what it knows about the knob.



```

> stz explain vat_rate
kind      param
value     0.18
bounds    0.00..0.25   declared in pricing.py:41
policy    DGI-2024-7 "VAT stays within the legal band"
history   3 refinements · last: amina@softibank, 2026-03-12
cascade   3 nodes would revalidate
  
```

**Fig.** Everything the system knows about one refinement point: its kind, current value, declared bounds, the policy that constrains it, its history — and what a change here would touch. All of this exists before any change is proposed.

`stz explain vat_rate` returns the point's whole biography. Its kind and current value. Its bounds — and *where* they were declared, file and line, because bounds are code, not folklore. The policy the bounds serve, with its human-readable reason: *"VAT stays within the legal band."* Its history — three refinements, most recently by `amina@softibank` in March. And a preview: a change here would send three nodes back for revalidation.

Pause on how much of Chapter 2's broken world this one screen quietly repairs. The auditor's questions — *who last changed this? under what rule? what does it affect?* — are answered before anything has even happened. In the diff era, assembling this screen took an afternoon of `git log`, ticket archaeology, and asking around. Here it is a query, because the change history is data, not sediment.

## §4 — PROPOSE, PREVIEW, LAND

Now the change itself — the last three commands of the first session, which contain the whole discipline in miniature.

`stz refine vat_rate --to 0.20` **proposes**. Note what the output calls it: a *typed proposal*, not a *diff*. Nothing has changed yet. You have not edited a file; you have stated an intention in the system's own terms — *set this declared point to this value* — and received a named proposal, `rfn-0114`, that the system can now reason about.

`stz cascade` **previews**. Before anything lands, the system computes the proposal's blast radius from the graph: three nodes would need revalidation — the quote function, the invoice total, the pricing tests. This is the **cascade**, and Chapter 6 will argue it deserves to replace the *diff* as the thing reviewers actually read. For now, notice only its tense: *would*. You are seeing consequences before committing to them, the way a chess player sees a move before making it.

`stz apply` **submits the proposal to the gate** — and the gate is the discipline's beating heart. Four checks run, in order, each printing its verdict:

**Structural** — is the change well-formed? Not by the foundation's opinion: Python itself re-validated the resulting file. **Constraint** — is 0.20 within the declared bounds? It is; the band ran to 0.25. This is where a fat-fingered 2.0 instead of 0.20 dies, instantly, structurally, no vigilant reviewer required. **Derivation** — do the downstream consequences hold? The three cascade nodes were revalidated; their verdicts came back unchanged. **Governance** — is this author allowed to change this point? Bounds are not the only law; *who may turn which knob* is declared too, and checked last.

Four passes, and the output ends with two words that summarize the era this book is arguing for: **landed**, **reversible**. The change is in. Its undo — the typed inverse, back to 0.18 — was computed *before* landing and stored with it. `stz revert`

would take it back as one atomic act. Not a hurried counter-edit under pressure; a stored, tested return path.

## §5 — WHAT JUST HAPPENED

Take stock, against Chapter 2's five properties, because you just produced all of them without doing anything that felt like paperwork.

The change was **validated** — four declared checks, passed in sequence, before landing. **Scoped** — its blast radius computed and shown to you in advance. **Attributed** — proposed under your identity, recorded with the point's history. **Reversible** — its inverse pre-computed and held. **Explainable** — every verdict above can be asked for later, with its reasons; `stz why rfn-0114` will narrate the whole decision to anyone who asks, including an auditor, including in eighteen months.

And now the sentence this chapter was built to deliver. Nothing in that session depended on *you* being the one typing. If a colleague had proposed the same change, the gate would have run the same four checks. If a script had proposed it — same checks. And if an AI agent, drafting at three a.m. at machine speed, had proposed it — *the same four checks*, because the gate examines proposals, not authors. Chapter 2 ended by observing that judgment of authors does not scale but properties of changes do. You have now watched a change acquire those properties in about ninety seconds of terminal time.

That is the discipline in miniature: **expression is free, admission is governed**. Anyone — anything — may propose. What lands is what passes.

The rest of this book is this session, taken seriously. The graph that made the cascade computable: Chapter 4. The gate's four stages and its way of refusing: Chapter 5. The cascade as the review surface: Chapter 6. The history and its two clocks: Chapter 7. Then the agents arrive in Part III — and you already know the punchline, because you have already seen the gate not care who is knocking.

## WHAT YOU NOW KNOW

- The unit of work has evolved twice before: the line, then the diff. Each carried more than its predecessor; the diff still carries only *which bytes moved*.
- A **refinement** is the third unit: a typed, bounded change to a *declared point*, validated before landing, attributed, cascade-aware, and reversible by construction.
- Systems declare their change surface as **refinement points** — named, bounded knobs of eight kinds — so change stops being file surgery and becomes knob-turning under law.
- You ran the full loop: lift → points → explain → propose → cascade → apply — four gate stages (structural, constraint, derivation, governance) — landed, reversible, explainable.
- None of it referenced the author. The gate examines proposals, not authors — which is exactly what makes the discipline hold at machine speed.

*What this chapter has not shown: how real, messy, existing codebases acquire a graph and declared points in the first place — that is Chapter 4 — and what a refusal looks like, which is Chapter 5's opening scene.*

...

## PART II

# The discipline

---

Four mechanisms carry the whole discipline: the graph, the gate, the cascade, and the audit chain. One chapter each. Every one is shown before it is named, in the terminal first and in a product's IDE second — because the mechanism belongs to the open foundation, and the pixels are just one way to see it.



## CHAPTER 4

# The graph

---

*Your codebase is already a graph. The discipline just refuses to forget it.*

---

Go back to the notebook Amina opened in Chapter 2 — the page where she listed everything the penalty cap touched. Fourteen products. The risk module. The report, the letters, the SMS templates, the tests. That page was her doing, by hand and from memory, what this chapter is about: tracing the web of connections that every codebase has and almost no codebase *keeps*.

Because here is the quiet scandal of software as it is normally stored. The connections are real — this function calls that one, this test asserts that rule, this report depends on that rate. The system runs on them every day. But they are written down nowhere. They live in senior engineers' heads, in tickets, in comments that were true two years ago, and in the sediment of experience that walks out the door with every resignation. The most valuable map in the company is a folklore document, and everyone maintains it by getting burned.

## §1 — WHERE THE MEANING LIVES

Look at how a codebase presents itself, and then at what it actually is.

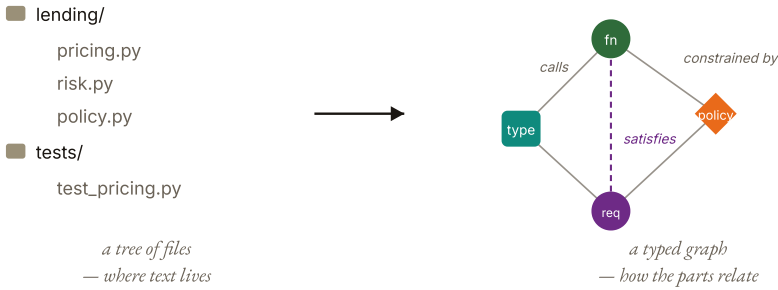


**Fig.** The same codebase, two ways. The file tree answers "where is the text?" The typed graph answers "how do the parts relate?" — including the purple *satisfies* edge, which ties code to the rule it exists to honor. Only one of these two can compute a blast radius.

The file tree on the left is how every mainstream tool sees your system: folders and files, a hierarchy of *storage*. It is a fine answer to the question "where is the text?" — and it answers nothing else. Nothing in the tree says that `pricing.py` depends on the number in `policy.py`, or that a test three folders away silently asserts today's penalty cap.

The graph on the right holds the same system by its *meaning*: every function, type, parameter, policy, test, and requirement a **node**; every real relationship a typed **edge** — *calls*, *depends on*, *constrained by*. Nothing exotic is being added. These relations already existed; the graph is simply the refusal to forget them.

One edge deserves its own paragraph, because it does work no ordinary tool does. The **satisfies** edge connects code to the *reason the code exists* — this pricing function *satisfies* policy BCEAO-4.2; that validation routine *satisfies* a clause of the data-protection rule. The moment those edges exist, questions that were unanswerable become queries. *Which code implements this regulation?* Follow the

edges backward. *If this regulation changes, what must change with it?* Follow them forward. Amina's notebook page, computed in milliseconds — and never out of date, because it is derived from the system rather than remembered about it.

## §2 — THE LIFT

The obvious objection arrives immediately: nobody is going to rebuild a working bank around a graph. Correct — and nobody has to. Existing code *enters* the graph through the *lift*, and the lift moves nothing.

```

stz - the lift: code enters the graph

> stz lift lending/
pricing.py    lifted · 214 nodes    parser: python
risk.py       lifted · 178 nodes    parser: python
policy.py     lifted · 61 nodes     parser: python
tests/        lifted · 93 nodes     parser: python

graph 546 nodes · 1,312 edges · 9 refinement points declared
note    2 annotations reference policies not yet in the graph
        → stz points --missing to list them
  
```

**Fig.** *The lift, module by module. Files stay exactly where they are — the graph is a second reading of the same code, not a migration. Note the parser column: each language is the authority on its own structure.*

Two details in that panel carry the whole design. First, the files are not converted, relocated, or wrapped. `pricing.py` remains `pricing.py`; your editors, your build, your habits all still work. The graph is a *reading* of the code, kept alongside it — which means adopting the discipline is not a rewrite. It is closer to indexing a library than to moving one.

Second, look at the parser column: *python*. The foundation does not bring its own opinion of what a Python file means — it asks Python. Each language's own parser is the authority on that language's structure, and the graph records what the authority reported. This will matter again in Chapter 11, where the same move

makes the whole discipline work across languages; for now it is enough to notice the humility. The graph adds relations *between* meanings. It does not overrule anyone's meaning.

And the refinement points of Chapter 3? They ride in the same way — declared as structured annotations in ordinary comments, lifted into the graph as typed nodes. The panel's last lines show the honest edge of this: two annotations referenced policies nobody has declared yet. The lift does not invent what it cannot find. It tells you, and gives you the command to see the gap.

### §3 — WHAT THE GRAPH MAKES POSSIBLE

Everything in the rest of this book is a graph operation wearing a friendlier name. It is worth seeing that plainly once, because it demystifies all four mechanisms at a stroke.

The **cascade** you previewed in Chapter 3 is reachability: start at the changed node, walk the edges, collect what you touch. The **gate**'s constraint stage reads bounds and policies straight off the nodes they constrain. The **audit chain** is a growing subgraph of proposals, verdicts, and landings, linked to everything they concerned. Even **stz explain** is just a node reciting its own edges.

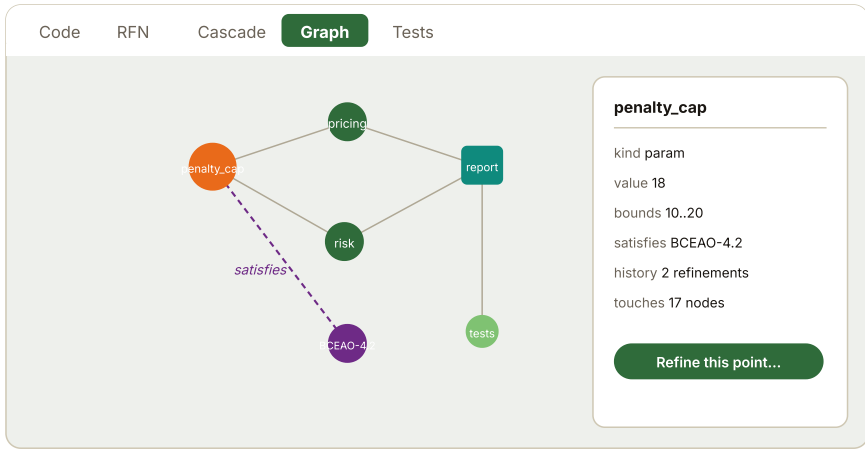


**Fig.** An illustration of the Refine IDE's Graph facet: the lifted graph on canvas, the satisfies edge in purple, and an inspector reading one node's biography — the same information `stz explain` prints, drawn instead of typed. (An illustration, not a screenshot.)

Commercial tooling renders this graph visually — the panel above illustrates how the Refine IDE draws it, with the same node biography you met in the terminal. The pixels are a convenience. The point is the object underneath: one graph, readable by the terminal, by an IDE, by an auditor's export, and — Part III will make much of this — by an agent, which finds in it the exact map of what it may touch.

A last word about honesty. The graph knows only what was declared or derived: lifted structure, declared points, declared policies, satisfies edges someone asserted. It is not omniscient, and this discipline never claims it is; an undeclared dependency is exactly as invisible here as it was in the file tree. What the graph changes is what happens *next*: every gap found becomes a declaration added, and the map improves monotonically — instead of decaying in someone's head.

## WHAT YOU NOW KNOW

- Every codebase already runs on a web of real relations; standard tooling stores the text and forgets the web, so the map lives in folklore and decays.

- The graph keeps the system by its meaning: typed nodes and edges — including **satisfies**, which ties code to the rule it honors and turns compliance questions into queries.
- Existing code enters by the **lift**: files stay put, each language's own parser is the authority, declared points and policies ride in as annotations, and gaps are reported rather than papered over.
- The discipline's mechanisms are graph operations: cascade = reachability, constraint checks = reading the nodes, the audit chain = a subgraph. Terminal, IDE, and agent all read the same object.

*What this chapter has not shown: what actually happens at the moment of judgment — the four checks, and what a refusal looks like. That is the gate, next.*

## CHAPTER 5

# The gate

---

*Four checks stand between a proposal and your codebase. Nothing lands without passing them.*

---

In week three of the penalty-cap work, a developer named Kofi proposed a tidy little refactor — renaming a scoring function to match the team's conventions. The kind of change that sails through any code review on a nod. The gate refused it.

It refused it because a test, three folders away, still called the function by its old name — a stale reference nobody in the room remembered. In the diff era this lands in production on a Friday, breaks the Monday batch run, and costs an afternoon of archaeology plus a week of "how did that get through review?" Here it cost Kofi ninety seconds, and the refusal *told him what to fix*.

Hold that story while this chapter takes the gate apart, because it reframes everything about what a refusal is. A gate that refuses well is not an obstacle. It is the fastest honest answer you can get.

## §1 — FOUR QUESTIONS, IN ORDER

Every proposal — Chapter 3's `stz apply`, an IDE button, an agent's submission over a protocol; the entry door does not matter — faces the same four stages in the same order.

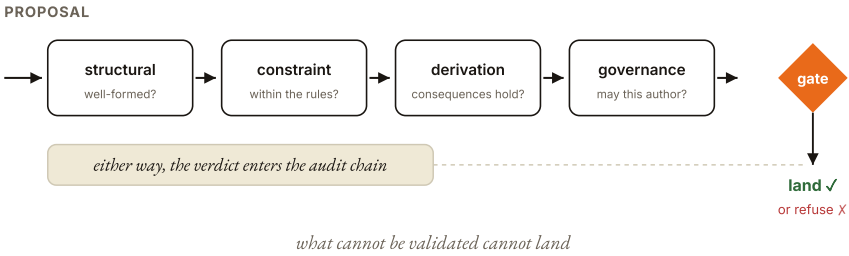


**Fig.** *The gate. Four stages, one order, no exceptions: is it well-formed, is it within the rules, do the consequences hold, may this author do this. Pass all four and the change lands; fail any and it is refused with reasons. Either way, the verdict is recorded.*

**Structural** asks: *is the change well-formed?* And it asks the only authority that counts — the language itself. If the proposal touches Python, Python's own parser re-validates the result; the foundation does not substitute its opinion for the compiler's. Malformed changes die here, instantly.

**Constraint** asks: *is the change within the declared rules?* Bounds on the point, policies that constrain it, invariants that must hold. This is where a typo'd `2.0` for `0.20` dies — not because a reviewer was sharp-eyed at 4 p.m., but structurally, the way a cash machine refuses a withdrawal that exceeds a balance.

**Derivation** asks: *do the consequences hold?* The cascade is walked, and every node it reaches is revalidated — do the tests still pass, do dependent verdicts still stand? This is the stage that caught Kofi's stale test: the rename was internally perfect and *consequentially* broken, and only this stage looks there.

**Governance** asks, last: *may this author make this change?* Bounds on values are not the only law; *who may turn which knob* is declared too. A junior developer may refine display text but not `policy/*`; an agent may propose anywhere but land only where its authority reaches. Chapter 3 promised the gate does not care who is knocking — this stage is where that promise is kept precisely. The gate cares *what authority* the knocker holds, and nothing else about them.

## §2 — A VERDICT THAT LANDS

Here is the canon change of this book — Amina's penalty cap, the Q3 letter's answer — passing through.

```

stz — a verdict that lands

> stz refine penalty_cap --to 16 --reason "BCEAO circular 2026-Q3"
proposal rfn-0207 recorded · author amina@softibank

> stz apply
structural    pass    python re-validated 14 files
constraint    pass    16 within declared bounds 10..20
derivation    pass    17 nodes revalidated · verdicts hold
governance    pass    amina@softibank may refine policy/*
landed. reversible · audit entry sealed
  
```

**Fig.** *The Q3 letter, answered. Note what each stage contributes and how the verdict ends: landed, reversible, sealed. The reason travels with the change — an auditor reading this in two years needs no meeting notes.*

Read the right-hand column of that verdict and notice it is *prose about reasons*, not status codes. Fourteen files re-validated by Python. Sixteen inside the declared band. Seventeen downstream nodes revalidated and holding. An author whose authority covers policy. The verdict is not "green"; it is an *argument*, four sentences long, machine-checked and permanently recorded. That is what Chapter 2 meant by trust becoming a checkable property.

## §3 — A REFUSAL WORTH READING

Now the other verdict — the one most systems make painful and this discipline makes useful. A colleague proposes 22.

```

stz - a refusal worth reading

> stz refine penalty_cap --to 22
proposal rfn-0208 recorded · author kofi@softibank

> stz apply
  structural    pass
  constraint    FAIL
refused — nothing changed.
what happened  22 exceeds the declared bound 10..20
why            policy BCEAO-4.2 caps late-payment penalties
what to do     propose within 10..20, or refine the policy
               itself (requires compliance authority)
learn more     stz why rfn-0208 · stz explain penalty_cap

```

*Fig. A refusal in four parts: what happened, why, what to do next, where to learn more. Nothing changed, nothing broke, nobody's afternoon was lost — and the proposer leaves knowing more than they arrived with.*

Look at the structure of the refusal, because it is a designed grammar, not an error message. **What happened:** 22 exceeds the declared bound. **Why:** the bound serves BCEAO-4.2, named. **What to do next:** two honest paths — propose within the band, or refine the policy itself, which is possible but requires compliance authority. **Where to learn more:** the commands that narrate the whole decision.

Every refusal in the discipline speaks this grammar, and the cumulative effect on a team is hard to overstate. A refused proposal costs seconds, not days. It cannot half-land, so there is nothing to clean up. It names the rule, so the proposer learns the system instead of learning fear. And — the deep one — it makes *daring cheap*. When a wrong proposal is guaranteed to bounce with a map back, trying things stops being dangerous. The gate is strict so that people can be bold.

## §4 — DETERMINISTIC, AND HONEST ABOUT IT

One property runs under all four stages and gives the gate its authority: it is **deterministic**. The same proposal against the same declared rules yields the same verdict — today, tomorrow, replayed in front of an auditor. There is no model in

the loop deciding differently after lunch. Whatever else in your stack is probabilistic (and in Part III, plenty will be), the moment of admission is not.

That determinism is also the gate's honest limit, stated here and again in Chapter 13: the gate proves a change against *what you declared*. It does not know the rules you never wrote down, and it does not judge taste, elegance, or wisdom — a change can pass all four stages and still be a bad idea, beautifully validated. The gate was never meant to replace judgment. It is meant to *spend* judgment where judgment is needed, by disposing of everything checkable before a human ever looks.

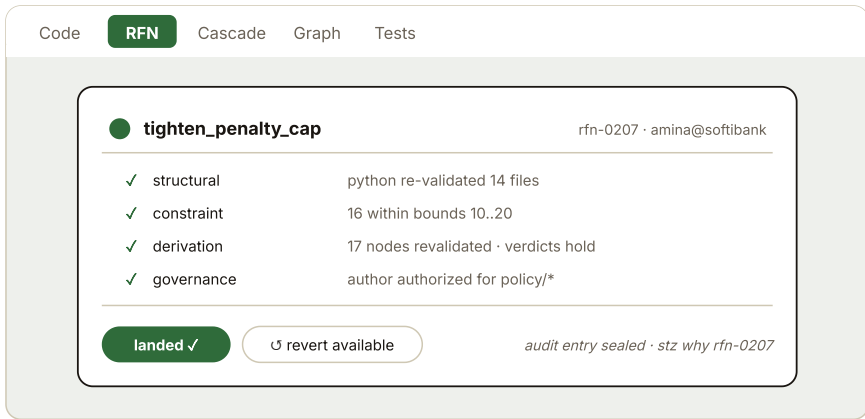


**Fig.** *The same verdict as a surface: the Refine IDE's verdict card for rfn-0207 — four stages, their reasons, the landed state, and the revert within reach. (An illustration, not a screenshot.)*

## WHAT YOU NOW KNOW

- Every proposal, from every author, faces four stages in order: **structural** (the language itself validates), **constraint** (declared bounds and policies), **derivation** (the cascade revalidated), **governance** (declared authority).
- A landing verdict is an argument with reasons, recorded permanently — not a green light.
- Refusals follow a four-part grammar — what happened, why, what next, where to learn — and a good refusal costs seconds, teaches the system, and

makes daring cheap.

- The gate is deterministic: same proposal, same rules, same verdict, replayable. And it is honest: it proves what you declared, and judges neither the undeclared nor the unwise.

*What this chapter has not shown: the thing the derivation stage walks — the cascade itself, which is about to replace the diff as the surface a reviewer actually reads.*

## CHAPTER 6

# The cascade

---

*Before a change lands, you see everything it touches. That view replaces the diff.*

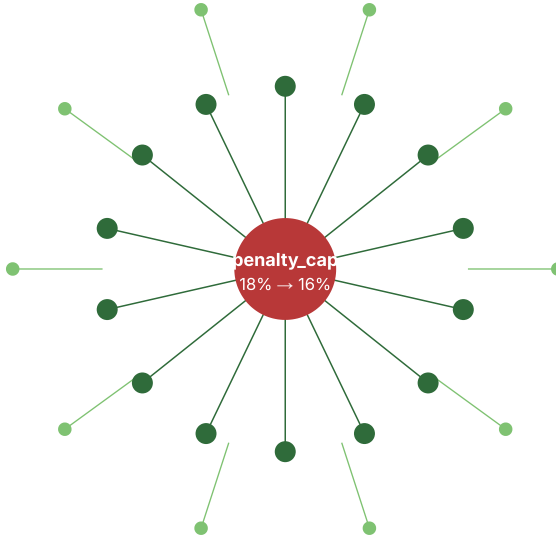
---

Here is Amina's review problem, stated with the numbers from her own notebook. The penalty-cap change touches fourteen loan products, a risk module, a report, letters, templates, tests. As diffs, that is hundreds of changed lines across dozens of files — an afternoon of reading that would still answer the wrong question, because what she needs to know is not *which bytes moved*. It is *what does this change do to the bank?*

The discipline's answer is the artifact you glimpsed in Chapter 3 and the gate walked in Chapter 5. It deserves its own chapter, because it changes what "reviewing" means.

## §1 — THE BLAST RADIUS, COMPUTED

revalidated:



*A single penalty-cap change forces fourteen products to revalidate, then ripples outward — until a revalidated node's verdict comes back unchanged, where the cascade stops.*

The **cascade** is the answer to one question, computed from the graph before anything lands: *if this node changes, what must be re-examined?* Start at the change, walk the edges, collect every node reached — products that reprice, reports that restate, tests that re-run, policies that re-check. The figure above is the penalty-cap change's cascade drawn as ripples; on the web edition it animates, and the animation is worth watching once, because it shows the cascade's most important property in motion: it *ends*.

## §2 — READING ONE

In the terminal, the same cascade reads as a structured brief.

```

stz - the blast radius, before anything lands

> stz cascade rfn-0207
depth 1 14 loan products reprice      gates: tests/pricing/*
depth 2 risk_assessment · statement_gen gates: tests/risk/*
depth 3 q3_compliance_report          gate: policy/BCEAO-4.2
stopped 4 further nodes unchanged — early cutoff

17 nodes revalidate · nothing lands until stz apply

```

**Fig.** *A cascade as a reviewer reads it: what re-validates, at what depth, under which gates — and where propagation stopped because verdicts came back unchanged. Seventeen nodes, grouped and bounded, instead of hundreds of diff lines.*

Read it the way Amina does. **Depth 1:** the fourteen products that consume the cap directly — every one will reprice, and the pricing tests gate them. **Depth 2:** what depends on the products — risk assessment, statement generation. **Depth 3:** the compliance report, gated by the policy itself. Then the line that keeps cascades honest: **stopped** — four further nodes were revalidated and their verdicts did not change, so propagation ends there.

That last line is called **early cutoff**, and it is the difference between a useful blast radius and a terrifying one. A naive dependency walk says "everything eventually touches everything" and drowns you. The cascade re-validates as it walks, and where a node's verdict comes back *unchanged*, the wave stops — proven harmless, not assumed harmless. The result is a blast radius that is honest in both directions: nothing reached is hidden, and nothing unreachable is included to frighten you.

### §3 — WHAT REVIEW BECOMES

Now compare the two review sessions side by side, because this is the chapter's cash value.

The diff session: hundreds of lines, file by file, the reviewer reconstructing in her head — from bytes — what the change means, what it touches, whether the touched things still hold. All of Chapter 2's arithmetic applies: at machine speed,

that reconstruction stops happening, and approval quietly decouples from understanding.

The cascade session: *the reconstruction is already done, by the machine that is good at it*. What reaches the reviewer is the meaning layer — seventeen nodes, grouped by depth, each already revalidated, gates named. Amina’s attention goes where attention is actually needed: is depth 3’s compliance restatement *supposed* to change this quarter? That is a judgment call, and she is the right instrument for it. The other 95 percent of the reading — the mechanical *did-anything-break* sweep — was never a good use of her, and the derivation stage already did it.

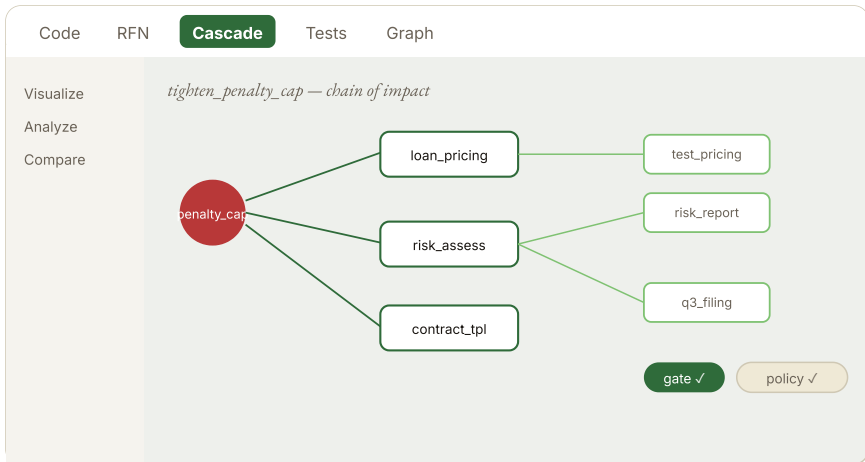


**Fig.** *The Refine IDE’s Cascade facet — the chain-of-impact view a reviewer reads in place of the diff, with the target change at left and its consequences flowing rightward through gates. (An illustration of the working prototype, not a screenshot.)*

This is what re-couples approval to understanding at machine speed. The reviewer reads less and knows more, because what she reads is consequences, not bytes. When the industry’s own code-review tools began building change-impact graphs, they rediscovered this surface — as commentary beside the diff. The discipline’s move is to make it the *review object itself*, computed by the same machinery that gates the change.

**WHAT YOU NOW KNOW**

- The cascade answers *what must be re-examined if this lands*, computed from the graph before commit — products, reports, tests, policies, grouped by depth with their gates named.
- **Early cutoff** keeps it honest: propagation re-validates as it walks and stops where verdicts come back unchanged — proven, not assumed.
- Review changes object: instead of reconstructing meaning from bytes, the reviewer reads consequences directly, and spends her judgment only where judgment is needed.
- The same cascade is the derivation stage's checklist and the reviewer's brief — one computation, two consumers.

*What this chapter has not shown: what happens after landing — the undo you were promised, and the chain of record that answers for everything, years later. That closes Part II next.*



## CHAPTER 7

# Reversibility and the audit chain

---

*Every change carries its own undo, and its own answer to "who did this, and why?"*

---

Return, one last time, to the question Amina asked the vendor in Chapter 1 — the second half of it, the half that stumps every demo: *"...and how do I undo it on Thursday when the regulator calls?"*

Every experienced engineer knows why that question lands. In the diff era, undoing a landed change is not an operation; it is a small emergency project. Someone writes a *counter-edit* — a new change meant to neutralize the old one — under time pressure, against a system that has already moved on, hoping nothing else grew roots into the thing being removed. "Rollback" is the industry's most optimistic word.

## §1 — THE RETURN PATH, PACKED IN ADVANCE

The discipline's answer was computed before the change ever landed. Chapter 3's session ended with two words — *landed, reversible* — and the second word was not a mood.



*Undo is not an emergency counter-edit. It is a return path, packed before departure.*

**Fig.** *The timeline the system actually keeps. Every landed refinement stores its typed inverse at landing time. Revert is not a counter-edit written under pressure — it is the stored return path, applied as one atomic act, and itself recorded.*

Because a refinement is typed — *set this point from 18 to 16* — its inverse is computable at proposal time: *set it from 16 back to 18*, plus the precise undoing of everything the cascade re-derived. The gate stores that inverse as part of landing. `stz revert rfn-0207` applies it as **one atomic act**: no half-states, no archaeology, no hoping. And the revert is itself a governed event — attributed, recorded, visible in history — because taking a change back is also a change, and the discipline does not look away while it happens.

Feel what this does to the texture of operating a system. The cost of being wrong drops from "emergency project" to "one command." Combined with Chapter 5's cheap refusals, the discipline's posture toward its humans becomes legible: *make trying safe, make landing strict, make retreat instant*.

## §2 — THE CHAIN THAT ANSWERS BACK

Reversibility is the audit chain's most dramatic use, but its daily use is quieter: it *answers questions*. Every proposal, verdict, landing, and revert became a record the moment it happened — typed, linked, and queryable.

```

stz - the history that answers back

> stz history penalty_cap
rfn-0207  18 → 16      amina@softibank  2026-09-30  landed
rfn-0163  20 → 18      amina@softibank  2025-11-02  landed
rfn-0090  set bounds   methodologist    2025-03-17  landed

> stz why rfn-0207
proposed  2026-09-30 14:02 by amina@softibank
reason    "BCEAO circular 2026-Q3"
checked   4 stages passed · 17 nodes revalidated
landed    2026-09-30 14:06 · inverse stored (16 → 18)
as of     in force from 2026-10-01 · recorded 2026-09-30

```

**Fig.** *The history that answers back: a point's whole biography, then one refinement narrating itself — who, why, what was checked, what landed, the stored inverse, and the two dates that close this chapter.*

`stz history penalty_cap` is the point's biography: three refinements, three authors, three dates, three outcomes — including the methodologist's original bounds-setting, which is itself on record. `stz why rfn-0207` is one decision narrating itself: proposed by whom, for what stated reason, checked how, landed when, reversible to what.

Notice what is *absent* from those screens: any act of documentation. Nobody wrote this record; it is the exhaust of the workflow itself, which is the only reason it can be trusted to exist. Documentation that must be remembered is documentation that is sometimes missing — and always missing the week the auditor visits. Here, the workflow *is* the documentation. The audit is not something the team prepares. It is something the system already contains.

### §3 — THE TWO CLOCKS

One subtlety separates a good audit record from a bank-grade one, and it is worth the two minutes it takes. Look again at the last line of `stz why`: *in force from 2026-10-01 · recorded 2026-09-30*. Two different dates, on purpose.

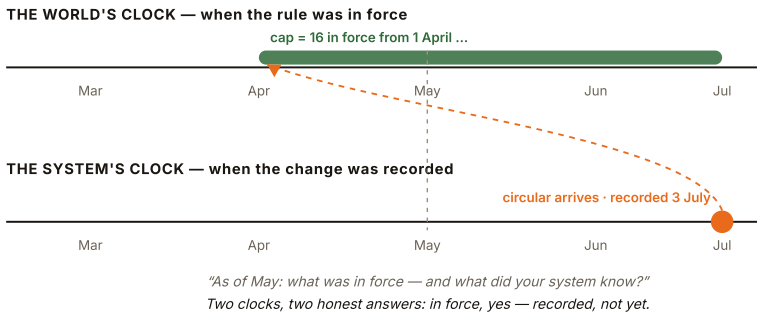


**Fig.** Regulation lives on the world's clock; systems live on their own. A circular recorded in July can be in force from April — and an auditor's "as of May" question needs both clocks to be answered honestly. The chain keeps both.

Every fact in the chain carries two timestamps. **The world's clock:** when the rule was actually in force. **The system's clock:** when the system learned of it and recorded the change. Usually they nearly coincide. But regulation has a habit that breaks naive record-keeping: the retroactive rule. A circular issued in July, effective from the first of April, is entirely ordinary in a regulated industry — and a system with one clock can only lie about it, one way or the other.

With two clocks, the auditor's sharpest question — "as of May: what was in force, and what did your system believe?" — gets the honest double answer: in force, yes; recorded, not yet; and here is the refinement, dated on both clocks, that closed the gap when the circular arrived. That answer, given plainly, is what regulatory trust is made of.

## §4 — PART II, ASSEMBLED

Step back, because the four mechanisms you now own are one machine.

The **graph** keeps the system by its meaning, so consequences are computable. The **cascade** computes them, before commitment, honestly bounded by early cutoff. The **gate** judges each proposal against declared rules — structure, constraints, consequences, authority — and refuses in a grammar that teaches. And the **audit chain** remembers every verdict on two clocks, holds every return path, and answers every later question without anyone having remembered to write anything down.

Validated, scoped, attributed, reversible, explainable — Chapter 2's five properties, produced by construction. And still, in all of Part II, not one mechanism asked who the author was: the gate examined authority, never identity; the cascade computed consequences, never intentions. The discipline is finished learning to govern *changes*. What walks through the door next is the author everyone is arguing about.

## WHAT YOU NOW KNOW

- Reversibility is a data property, not a heroic act: every landed refinement stores its typed inverse at landing; revert is one atomic, governed, recorded act.
- The audit chain is workflow exhaust, not documentation: proposals, verdicts, landings, and reverts become queryable records the moment they happen — so the audit always exists.
- Every record carries **two clocks** — in force, and recorded — which is what lets a system answer retroactive-regulation questions honestly.
- The four mechanisms compose: graph → cascade → gate → chain. Five trust properties, by construction, author-blind throughout.

*What this chapter has not shown: agents. Part III opens the door — and you already know the one thing that matters about what happens when they walk through it.*

...



## PART III

# Agents in a governed world

---

The industry's agent is an LLM in a loop. This part takes the loop apart, shows the ladder it sits on, and puts it back together inside the discipline — where an agent's proposals face the same gate as everyone else's, and where the safest thing you can hand an agent is a typed knob, not your file system.



## CHAPTER 8

# Agents, demystified

---

*An agent is a mind with a body and stakes. The mind does not have to be a language model.*

---

Of all the jungle words, *agent* has done the most damage, because it is the one executives buy. So before Softibank lets anything called an agent near its loan system, this chapter does to the word what a watchmaker does to a watch: takes it apart on a clean table, names every piece, and puts it back together — running, and understood.

The industry's definition is easy to state, because for once the industry agrees: an agent is *a language model in a loop* — give it a goal, let it call tools, let it keep some memory, repeat until done. This is a real and useful design; nothing in this chapter mocks it. But notice what kind of definition it is. It names the parts, not the properties. It says what an agent is *made of*, and nothing about what an agent can be *held to* — which is the only question a bank, or anyone accountable, actually has.

## §1 — THE LADDER

To hold something to account, you need to know what it is. Here is the same territory as a ladder, where each rung adds exactly one thing.

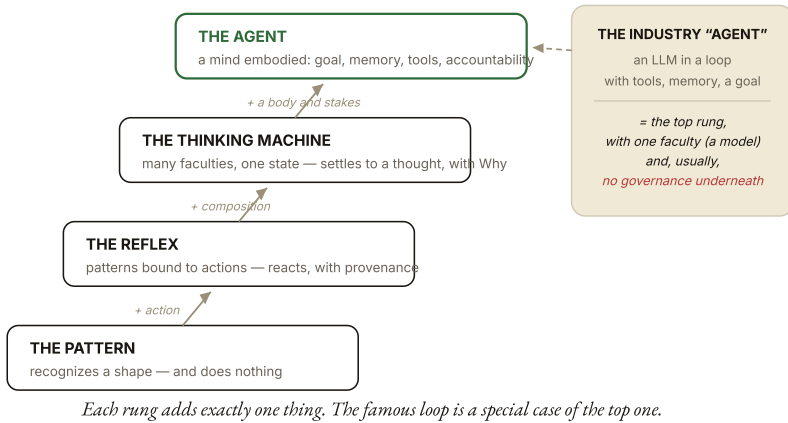


**Fig.** Four rungs, each adding one thing: a pattern recognizes; a reflex reacts; a thinking machine composes faculties and settles on a thought, with reasons; an agent embodies a mind with a goal and stakes. The industry's loop is the top rung — with a model for a mind and, usually, nothing underneath.

At the bottom, a **pattern**: something that recognizes a shape — in text, in a list, in a graph — and can say *this matches, and here is how*. It does nothing; it only sees. One rung up, a **reflex**: patterns bound to actions. When this shape appears, do that — and record why. A thermostat is a reflex; so is an alert that fires when a payment fails. Real, useful, mindless.

The third rung is where “thinking” stops being a metaphor. A **thinking machine** is many faculties — pattern-matchers over different media — sharing one working state, each able to react to what the others notice, until the whole thing *settles* on a result. A thought, in this machine, is not mystical: it is an entry in that state, carrying its provenance. You can ask the machine *why* it concluded what it concluded and get an answer, every time, the same answer for the same inputs.

And the top rung: an **agent** is a thinking machine *embodied* — given a goal to pursue, memory to draw on, tools to act with, and accountability for what it does. A mind with a body and stakes.

Now place the industry's loop on this ladder, and the fog around the word lifts. The LLM-in-a-loop *is* a top-rung agent — with two particular choices made silently: its entire mind is **one faculty** (a language model), and it usually stands on **nothing** — no settled state, no provenance, no answer to *why* beyond the model's own eloquence. The famous definition is not wrong. It is a special case, and naming it as one is what lets you engineer around it.

## §2 — TWO KINDS OF MIND, ONE GATE

Because once the mind is just a component, you can choose it per task — and there are two kinds on the shelf.

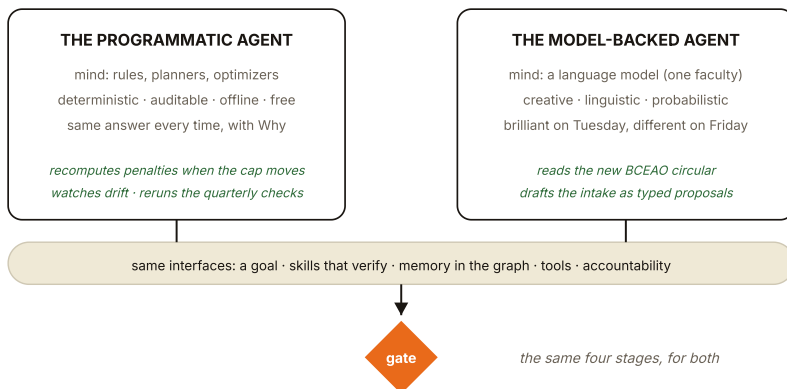


**Fig.** *Two minds, one contract. The programmatic agent is deterministic and free; the model-backed agent is creative and probabilistic. They share the same interfaces — goal, skills, memory, tools — and, crucially, the same gate. Choosing between them is engineering, not ideology.*

The **programmatic agent** has a mind made of rules, planners, and optimizers — the machinery of the third rung, pointed at a goal. It is deterministic: same situation, same conclusion, same *why*, at no cost, offline, forever. When the penalty cap moved from 18 to 16, the agent that recomputed every affected schedule and re-ran the quarterly checks did not need creativity. It needed to be *exactly right, provably, at three a.m.* — the programmatic mind's home ground.

The **model-backed agent** has a language model where its mind's creative faculty goes. It can read a new BCEAO circular — prose, French, ambiguous — and draft what the circular implies. No rule-based mind can do that; this is genuinely what models are for. The cost is the model's nature: probabilistic, brilliant on Tuesday, subtly different on Friday, eloquent even when wrong.

The discipline's position is neither "always the model" nor "never the model." It is: **choose the mind per task, and govern them identically.** Both kinds expose the same interfaces — a goal, skills, memory, tools. Both propose changes the same way. And both face the same four-stage gate, where — say it once more — the gate examines proposals, not authors. Softibank's directive-intake agent (model-backed, reads circulars) and its recompute agent (programmatic, moves numbers) are colleagues under one law.

### §3 — MEMORY AND SKILLS, HELD TO THE SAME STANDARD

Two of the loop's components deserve their own sentences, because Part II already built their governed versions.

**Memory.** In the loop, memory is whatever the agent kept — a transcript, a vector store, a scratchpad. Unauditable by construction: *what did the agent know when it acted?* has no reliable answer. In the discipline, an agent's memory is a **view of the graph** — the same governed object everything else lives in. What the agent knew is a query. When it knew it is a query too, because the two clocks of Chapter 7 apply to an agent's knowledge exactly as they apply to a regulation.

**Skills.** In the loop, a skill is often a prompt — instructions the model is trusted to follow. In the discipline, a skill is a **capability that verifies**: it declares what must be true before it runs, what it does, and how to check that it did it. The difference is not pedantry. A skill that cannot verify is a hope with a name; a skill that verifies is an engineering component. Registries, versions, and trust labels follow naturally — Chapter 10 files them on the map.

## WHAT YOU NOW KNOW

- The industry's agent — an LLM in a loop with tools, memory, and a goal — is a parts list, not a set of properties; it says nothing about what the agent can be held to.
- The ladder adds one thing per rung: pattern (recognizes) → reflex (reacts, with provenance) → thinking machine (composes faculties, settles, answers *why*) → agent (a mind embodied, with goal, memory, tools, accountability).
- The famous loop is the top rung with one faculty (a model) and usually no governed floor — a special case, not the definition.
- Two minds are on the shelf: programmatic (deterministic, free, auditable — for the work that must be exactly right) and model-backed (creative, linguistic, probabilistic — for the work that needs language). Same interfaces, same gate, chosen per task.
- Memory and skills have governed forms: memory as a queryable view of the graph on two clocks; skills as capabilities that verify.

*What this chapter has not shown: what you actually hand an agent when you let it act on your system. That is the next chapter, and it is the discipline's sharpest practical payoff.*



## CHAPTER 9

# The safest write surface

---

*Give an agent typed knobs, not your file system. Wise coding asks; vibe coding guesses.*

---

The month Softibank first let an agent propose changes to the loan system, nobody in the compliance department slept badly. This chapter is about why that sentence is possible — because it is not possible in most organizations, and the difference is not courage. It is surface area.

When people imagine "giving an AI write access," they imagine the file system: the agent, alone with the code, editing anything, the way a contractor with a master key is alone with the building. Every fear about agentic coding is downstream of that picture, and the fears are rational — *against that picture*. Unbounded surface, untyped edits, consequences discovered later, undo by archaeology. The discipline's answer is not a braver key policy. It is a different building.

## §1 — THE AFFORDANCE MAP

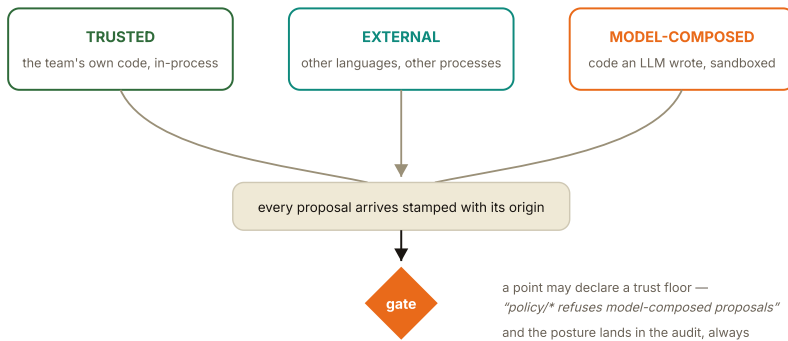
Recall what Chapter 3 established and Chapter 4 built: a system that declares its refinement points has a *published change surface* — named, typed, bounded knobs, each carrying its policy, its history, its blast radius. `stz points` prints it for you.

Here is the turn this chapter exists for: **that same listing is what the agent reads**. The points are the agent's *affordance map* — the machine-readable answer to "what may be changed around here, within what limits, under whose authority?" An agent facing declared points does not rummage through files wondering what is safe. It faces a panel of labeled controls, each one saying what it is, what band it accepts, and what it touches.

And its proposals are *typed* — "set `vat_rate` to 0.20", never "here are three hundred edited lines." A typed proposal against a declared point is the **safest write surface** you can hand an autonomous system: bounded (the bounds are law, not advice), discoverable (the map is generated, not documented), gate-validated (four stages, no exceptions), reversible (the inverse is stored before landing), attributed (the author is part of the record). The agent is not trusted less than a human, or more. It is trusted *identically*: to propose, and to have its proposals judged.

## §2 — WHERE THE CODE CAME FROM IS PART OF THE CHANGE

One honest complication, and the discipline meets it head-on. Agents do not only *propose* changes — sometimes they *write code* that runs: a report generator, a data-cleanup script, a scheduled job. The proposal is typed, but the thing proposed is executable. Does provenance vanish once it runs?



*Where code came from is part of what the change is.*

**Fig.** Every execution carries its origin — the team's own code, an external process, or model-composed code in a sandbox — stamped on the proposal and preserved in the audit. Sensitive points can declare a trust floor: some knobs simply refuse lower-trust origins, structurally.

No: **where code came from is part of what the change is**. Every execution in the discipline carries a *trust posture* — the team's own trusted code, an external process, or model-composed code running in a sandbox with only the capabilities it was explicitly handed. The posture is stamped on the proposal, checked by the gate, and preserved forever in the audit chain. And a sensitive point can declare a **trust floor**: `policy/*` at Softibank refuses model-composed proposals outright — not by convention, by construction. The auditor's question "did a model write this?" has a stored answer, for every change, in both directions.

### §3 — WISE CODING: THE SYSTEM ASKS

There is one more inversion, and it may be the deepest in the book. The industry's default interaction — call it what it is, *vibe coding* — has the human prompting and the machine guessing. Fluent, fast, and structurally silent about what the machine did not know. The gap between what you meant and what it assumed gets discovered later, in production, by someone else.

The discipline runs the conversation the other way. The system *knows what a complete answer requires* — the ontology, the bounds, the policies are all declared — so it can measure the gap between what it needs and what it has heard, and ask the next well-structured question itself.

```

stz - wise coding: the system asks

> stz refine --assist "student savings product"
stz: a savings product needs a rate profile. Which shape?
  1 fixed rate    2 tiered by balance    3 indexed (BCEAO base)
> 2
stz: tiers need boundaries. Answer with values, a rule - or examples.
> examples: 0 -> 2.5    100000 -> 3.0    500000 -> 3.5
stz: induced 3 tiers from your examples
      all within policy savings_rate_band ✓    nothing guessed, nothing landed
      proposal rfn-0233 drafted · stz cascade to preview
  
```

**Fig.** *Wise coding, in one exchange. The system knows what a rate profile requires, asks for the missing piece, accepts an answer in whatever register suits you — here, examples — induces the structure, checks it against policy, and drafts a typed proposal. Nothing was guessed; nothing has landed.*

Read the exchange for its manners. The system asked, because it knew what was missing. The human answered *with examples* — one of several accepted registers; a formula, a plain-English sentence, or a data structure would have done. The system induced the tiers, checked them against the declared rate band, and drafted — a *proposal*, not a change, with the cascade one command away. Where several interpretations survive, it enumerates them and asks; where none survives, it refuses with the reason. **Expression is free; admission is governed** — the gate's principle, extended backward into the conversation itself.

The name for this is **wise coding**, and it is what lets people who will never write code — a product owner, a compliance officer — build governed systems by answering good questions well.

#### §4 — LIVE REFINEMENT

One more door, because it shows how far "expression is free" reaches. In commercial tooling built on this foundation, you can point at the *running product* — click the actual sentence in the actual customer portal — and refine it from there.

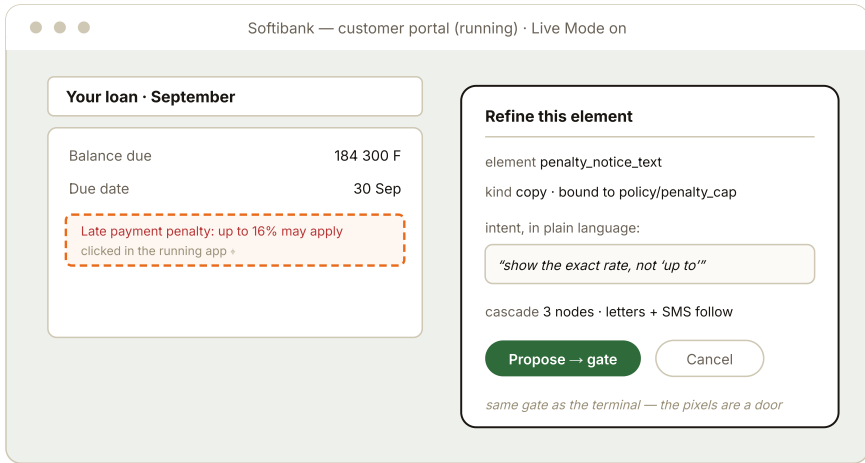


**Fig.** *An illustration of the Refine IDE's Live Mode: an element clicked in the running application opens a typed-refinement popover — the element's binding, an intent stated in plain language, the cascade previewed — and Propose leads to the same four-stage gate as everything else. The pixels are a door, not a bypass. (An illustration, not a screenshot.)*

The panel is doing exactly what `stz refine` did in Chapter 3 — same typed proposal, same cascade preview, same gate — reached through pixels instead of a prompt. That is the constant this whole Part has been circling: doors multiply — terminal, IDE, conversation, protocol, a click on a running screen, an agent at three a.m. — and the gate never does. Many doors, one law.

## WHAT YOU NOW KNOW

- The fear of "giving an AI write access" is a fear of the file system — unbounded, untyped, archaeologically reversible. The discipline changes the surface, not the courage.
- Declared refinement points are the agent's **affordance map**; typed proposals against them are the safest write surface an autonomous system can be handed — bounded, discoverable, gate-validated, reversible, attributed.

- **Trust postures** make code origin part of the change: stamped, gate-checked, audit-preserved — and sensitive points can refuse lower-trust origins by declared floor.
- **Wise coding** inverts vibe coding: the system measures the gap and asks; answers arrive in any register (including examples); candidates are checked against the declared rules; enumerate-or-refuse, never a silent guess.
- Doors multiply — terminal, conversation, running pixels, protocol — and the gate never does.

*What this chapter has not shown: the promised payoff — the jungle of Chapter 1, walked again with everything you now hold. Next.*

## CHAPTER 10

# The jungle, revisited

---

*The same map as Chapter 1 — but now every word has one governed meaning.*

---

Go back, one last time, to the meeting room where this book began — the handsome slides, the ten words, and Amina's question hanging in the air. Nine chapters later, you are better armed than the presenter was. Not because you memorized definitions, but because you now stand somewhere solid: you know what a governed change is, end to end. Stand there, look back at the jungle, and watch what happens to each word.

## §1 — THE WORDS, ONE BY ONE

**Agent.** A mind — programmatic or model-backed — embodied with a goal, memory, tools, and accountability. In a governed system it is, precisely, *an author whose proposals face the gate*. The word stops being a claim about magic and becomes a role in a process. (Chapter 8.)

**Harness.** Where agents are *built and run* — the loops, the scheduling, the plumbing. Real, necessary, and a different job from governing what agents change. A harness answers "how does the agent operate?"; the gate answers "does its work stand?" Confusing the two is how organizations buy execution and believe they bought governance. (Chapters 8–9.)

**Memory.** A governed view of the graph. *What the agent knew, and when* becomes a query on two clocks — not a rummage through a vector store. (Chapters 7–8.)

**RAG / context.** What a model reads before it acts — legitimate and useful, now with a governed form: sources carrying provenance, so that "what informed this draft?" has an answer. Retrieval feeds the *author*; it never touches the *verdict*. (Chapters 8–9.)

**Skills.** Capabilities that verify — precondition, action, check — registered and trust-labeled like anything else that acts on a system. A prompt in a costume does not qualify. (Chapter 8.)

**MCP.** A door. A protocol by which tools and agents reach a system — one of many doors (terminal, IDE, conversation, running pixels) through which *proposals arrive*. Doors multiply; the gate never does. (Chapter 9.)

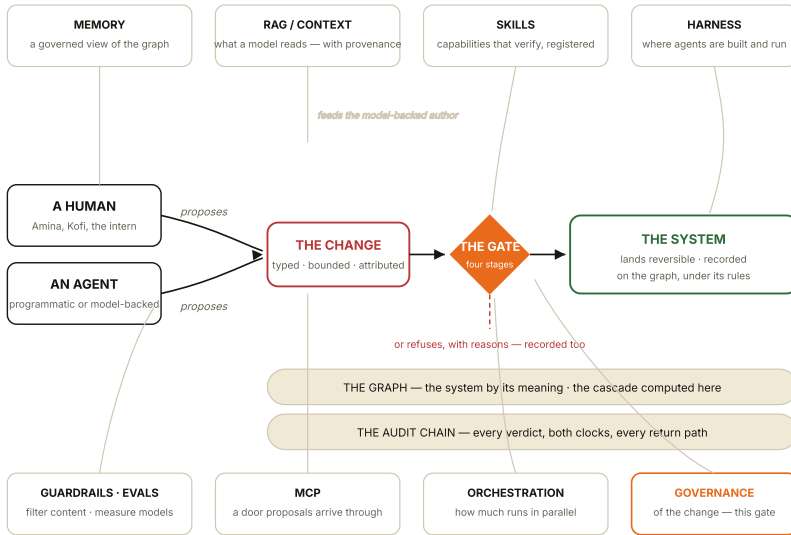
**Guardrails.** Filters on what a model says or is asked — content classification, at runtime, probabilistic. Useful upstream of the author. They *classify text*; they do not *decide changes*. The decision belongs to a deterministic gate reading declared rules. (Chapter 5.)

**Evals.** Measurements of how good a model is *on average, elsewhere, usually*. Indispensable for choosing a model; incapable of judging *this Tuesday's change to the penalty cap*. Averages inform; the gate decides each case. (Chapter 5.)

**Orchestration.** How much work runs in parallel — one agent or a fleet, one loop or a pipeline. The discipline's answer is almost shockingly calm: run what you like; the governance floor is flat. Every change from every lane faces the same four stages, so scaling the fleet scales throughput, not risk. (Chapters 5, 9.)

**Governance.** The word the book set out to rescue. It still legitimately means three things — governing the *model* (guardrails and policy), governing the *actor* (what an agent may reach and do at runtime), and governing the *change*. All three are real; they compose; and they are different objects. This book built the third, because it was the one nobody's box answered — and because it is the one the auditor asks about.

## §2 — THE MAP, REDRAWN



*The same words as Chapter 1 — each doing one job, around a governed center.*

**Fig.** *The same vocabulary as Chapter 1, redrawn from solid ground. The change is at the center, through the gate, onto the graph, into the chain — authors on the left, human and agent on equal arrows. Every other word holds an honest job at the edges. Nothing was banished; everything was placed.*

Put this figure next to Chapter 1's and notice what changed — and what did not. Not one box was deleted. Every word survived, because every word named something real all along. What changed is the *geometry*. The first map had no center, so every box competed to be one, and the arrows were question marks. This map has a center — **the change, through the gate, onto the graph, into the chain** — and around a fixed center, everything else relaxes into its actual job. Memory serves the authors. Retrieval feeds the model-backed ones. Guardrails and evals stand upstream. MCP and the terminal and the pixels are doors. The harness runs the fleet. And governance-of-the-change is no longer a gap in the middle of the picture. It *is* the middle of the picture.

### §3 — WHAT THE CONFUSION WAS, ALL ALONG

The jungle, it turns out, was a **category error at scale**. An industry facing a trust problem about *changes* reached for words that describe *actors*, *models*, and *pipelines* — and then stretched each word toward the gap, because the gap was real and something had to cover it. No box could, because the gap was not box-shaped. It was change-shaped.

This is why the jungle resisted better definitions. Glossaries tidy words; they cannot supply a missing object. Once the object exists — the refinement, with its graph, gate, cascade, and chain — the words stop stretching, because nothing is asking them to cover for it anymore.

And the practical test, the one from Chapter 1: two people can now agree to "deploy an agent with memory and guardrails under proper governance" and *mean something checkable* — a model-backed author whose knowledge is a query, whose inputs are filtered, and whose every change faces a deterministic gate it cannot talk its way past. That sentence has become an engineering specification. In Chapter 1 it was a mood.

Amina's question — *who checked this change, against which rule, and how do I undo it on Thursday?* — now has a three-part answer that fits on an index card: *the gate, by name; the declared rule, by name; one command, stored since the moment it landed*. The rest of this book asks how far that card travels: beyond code, beyond one language, into the boardroom — and where it stops.

### WHAT YOU NOW KNOW

- Every Chapter 1 word survived with one honest job: agent = a governed author; harness = where agents run; memory = a graph view on two clocks; RAG = provenance-carrying context for authors; skills = verifying capabilities; MCP = a door; guardrails = content filters upstream; evals = averages that inform, never verdicts; orchestration = parallelism over a flat governance floor.

- "Governance" splits into three real, composing disciplines — of the model, of the actor, of the change — and the third was the empty center of the map.
- The jungle was a category error: actor-words and model-words stretched to cover a change-shaped gap. Supply the missing object and the vocabulary relaxes; specifications replace moods.

*What this chapter has not shown: whether the governed center holds beyond code — for designs, data, other languages, and the people who answer to regulators. That is Part IV.*

...



## PART IV

# The bigger picture

---

The discipline is not only for code, and not only for programmers. This part widens the lens — to design and data, to any language, to the boardroom where regulators now ask what you can prove — and then closes the book the only honest way: by saying plainly what the discipline does not do.



## CHAPTER 11

# One gate, many worlds

---

*Anything that lives in a graph can be refined through the same gate — in any language.*

---

Look back at Amina's notebook list one more time, and notice something this book has been quietly stepping around since Chapter 2. The penalty cap was never just a code change. The customer letters and SMS templates that mention the rate — that is *interface*: wording, presentation, what a person sees. The compliance report and the column checks behind it — that is *data*: schemas, validations, derived figures. One regulatory sentence, and the ripple crossed three different crafts before lunch.

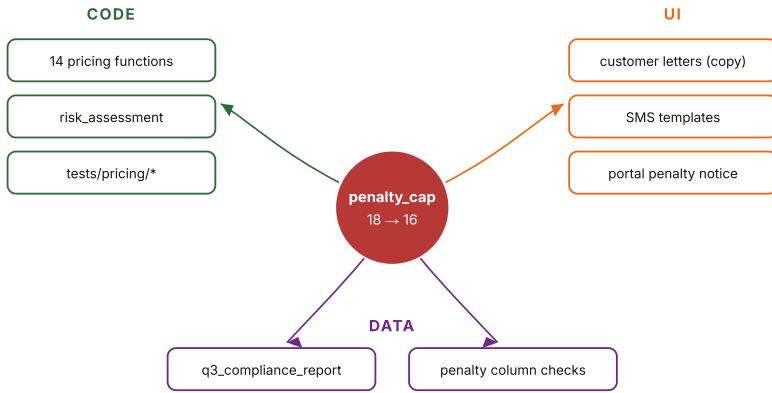
The industry treats those crafts as separate worlds with separate tools and separate change ceremonies — code review here, design review there, a database change-request form in a third place, and no instrument anywhere that sees the whole ripple. Which is exactly backwards, because the ripple does not respect the org chart.

## §1 — THREE DOMAINS, ONE CASCADE

The discipline's answer follows from what a refinement *is*. Nothing in Chapter 3's definition — a typed, bounded change to a declared point, validated, attributed, reversible — mentions code. It mentions a *graph*. And interfaces and data live in graphs as naturally as functions do.

A design system declares its points: colors, spacings, the wording of a notice — named, bounded, versioned. A data world declares its own: schema columns, validation rules, report derivations. The moment those are declared, everything in

Part II applies to them verbatim — proposals, bounds, the gate, the inverse. A copy change to the penalty notice is a refinement of kind *copy*; a report-column check is a refinement of kind *validation rule*. Same machinery, different craft.



*One change, three domains, one cascade — and one gate for all of it.*

**Fig.** *The Q3 change as it actually was: one declared point, and a cascade that crosses three crafts — fourteen pricing functions, the customer-facing wording, the compliance data — computed as one object, gated as one law.*

And because all three domains share one graph, the cascade is finally shaped like the truth. When the cap moved, the *one* computed cascade said: fourteen pricing functions, the letters, the SMS templates, the portal notice, the report, the column checks. No handoff meeting where code tells design what changed. No quarter-end surprise where the report disagrees with the product. The ripple is one object, reviewed once, landed once, reversible once.

## §2 — BEYOND ARTIFACTS

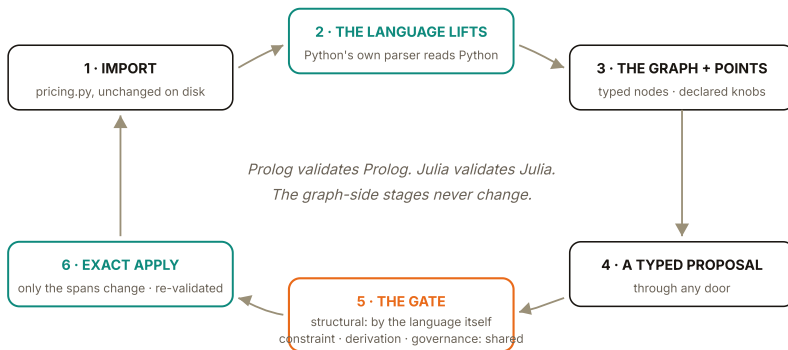
Push the definition once more and it keeps giving. Anything that lives in a graph under declared rules can be refined through the same gate — and by Chapter 4, *the system's own rules live in the graph too*.

So the rules themselves are refinable. When BCEAO's circular arrived, the deepest change was not to any pricing function — it was to the *policy node*: the declared bound moved from 20-max-18 to a new band. That change went through the gate like any other (a policy refinement, requiring compliance authority — the governance stage at work), and then the cascade re-governed everything downstream of the rule. The same holds for a threshold in an optimization model, a plan's parameters, even the elicitation policy of a wise-coding conversation. The discipline does not divide the world into "governed things" and "the configuration that governs them." It is graphs all the way down, and one gate over all of it.

### §3 — ANY LANGUAGE

One boundary remains, and it is the one a polyglot world cares about most. Softibank's estate is not one language — Python here, Java there, SQL everywhere, a vendor system speaking something older. Does the discipline require the one true language?

It requires the opposite, and Chapter 4 already showed the mechanism in miniature: *the language is the authority in its own gate*.



*One gate, many languages — each language the authority on its own structure.*

**Fig.** The loop that makes the discipline polyglot. The file never migrates; the language's own parser lifts it and re-validates it; apply rewrites only the exact spans a refinement touched. The structural stage belongs to each language — the other three stages never change.

Walk the loop. Python enters through Python's own parser — the foundation never writes its private opinion of what Python means. The graph and its declared points are language-neutral from that moment on. A proposal arrives through any door; at the gate, the *structural* stage is delegated back to the language itself — Python re-validates the resulting file, because on Python's structure, Python outranks everyone. The other three stages — constraint, derivation, governance — read the graph, and the graph does not care what language a node was lifted from. Apply is surgical: only the exact character spans of the refined point change on disk, so formatting and comments survive untouched. And the pattern generalizes for free, because every serious language ships its own parser: Prolog validates Prolog, Julia validates Julia, SQL validates SQL.

One gate, many worlds — many crafts, many artifacts, many languages. The discipline never conquered any of them. It only asked each to declare its knobs, and kept one law over the turning.

## WHAT YOU NOW KNOW

- A refinement's definition never mentioned code: interfaces and data declare points too, and Part II applies to them verbatim — one graph, one cascade shaped like the real ripple, one gate across the crafts.
- The rules themselves are refinable: a regulation change is a *policy refinement* (under compliance authority), after which the cascade re-governs everything downstream of the rule.
- The polyglot rule: each language's own parser lifts and re-validates its files (the structural stage is delegated); the graph-side stages are language-neutral; apply touches only exact spans. No migration, no one true language.

*What this chapter has not shown: what all of this is ultimately for — the day someone from outside walks in and asks the bank to prove itself.*

## CHAPTER 12

# In the boardroom

---

*Regulation no longer asks what you intend. It asks what you can prove.*

---

Eighteen months after the Q3 letter, on a Monday in March, the auditor came to Softibank. Not a crisis — the scheduled kind of visit, with a checklist refined over decades of asking banks to demonstrate that their systems do what their policies say. He had done a hundred of these. He opened with the question he always opens with, because it usually takes a week to answer.

"The penalty-cap revision, last September. Show me who approved it, what it affected, and how you know your customer communications matched the new rate."

In most banks, that question mobilizes a small task force: pull the tickets, find the emails, reconstruct the deploy logs, interview whoever still works there, assemble a binder, and hope the binder's story has no gaps. The industry name for this is *audit season*, and its cost is measured in weeks.

Amina typed for a few seconds and turned the screen around.

## §1 — WHAT THE QUESTION HAS BECOME

Something has shifted in what regulators ask, and every compliance officer has felt it. The old regime audited *intentions*: show me your policy documents, your procedures, your org chart of responsibility. The new regime — across financial supervision, data protection, and now the emerging rules for AI-assisted systems — audits *evidence*: show me that the control operated, on this change, on this date, and produce the record now.

The shift is rational. Regulators learned that documented intentions and operating reality drift apart, and that the drift is where the failures live. So the burden moved: from *what do you intend?* to *what can you prove?* — with the follow-up that breaks organizations built on binders: *and how quickly?*

Read Chapter 2's five properties again with a regulator's eyes and you can see this whole book from their side of the desk. Validated, scoped, attributed, reversible, explainable — that list *is* an examiner's checklist. The discipline was never only an engineering convenience. It is what being auditable looks like when it is designed in rather than reconstructed after.

## §2 — EVIDENCE ON DEMAND

Code RFN Cascade **Audit**

**The chain — Q3 period · policy/\***

|                                |                    |
|--------------------------------|--------------------|
| rfn-0207 penalty_cap 18 → 16   | amina · sealed ✓   |
| rfn-0211 letters + SMS wording | fatima · sealed ✓  |
| rfn-0219 report column checks  | agent · sealed ✓   |
| rfn-0208 penalty_cap → 22      | refused · recorded |

... 43 more in this period, every verdict on both clocks

**Evidence export**

scope BCEAO · policy/\*

period Q3 2026

entries 47 refinements

clocks in force + recorded

bundle signed, verifiable

**Export for the auditor**

minutes, not a war room

**Fig.** An illustration of the Refine IDE's Audit facet: the chain for the period — including the refusal, which is evidence too — and an evidence export scoped to the regulator's question: period, scope, both clocks, 47 entries, one signed bundle. Minutes, not a war room. (An illustration, not a screenshot.)

What Amina turned around was nothing she had prepared. It was Chapter 7's audit chain, filtered to the question: the policy refinement that moved the cap, under compliance authority; the fourteen products' revalidation, gates named; the wording changes to letters and SMS — the cross-domain cascade of Chapter 11, so the "communications matched the rate" question is answered by *the same object*

that made them match; the agent-authored column checks, trust posture stamped; and one refused proposal — Kofi's 22 — because refusals are evidence too, evidence that the gate was awake.

Every entry on both clocks. Every approval by a named authority distinct from the proposer. Every change carrying its stored inverse — because "could you have taken it back?" is also an examiner's question. And the whole thing exportable as a signed bundle scoped to exactly what was asked, in minutes.

The auditor's craft does not disappear in this picture — it *concentrates*. Freed from archaeology, his questions climb to where human judgment belongs: are the declared rules the *right* rules? Is the compliance authority assigned to the right people? He spent the afternoon on those. That is what an audit is supposed to be.

### §3 — THE LOOP, CLOSED

One more thing happened during that eighteen months, quieter than everything above and worth the chapter's last section. Regulation kept arriving — it always does. And the bank's *absorption* of it changed shape.

A directive used to enter Softibank as prose on paper, then diffuse — through meetings, memos, and individual judgment — into dozens of hand-made changes whose completeness nobody could prove. Now a directive enters as a **policy refinement**: the rule's node changes, under compliance authority, through the gate. The cascade then *computes* the compliance work — every function, wording, and report downstream of the rule — and the team works through a generated list instead of a remembered one. When the regulator later asks "how did you implement circular such-and-such?", the answer is the refinement's identifier, and everything it cascaded into, on the record, on both clocks.

Ninety days of project, become an afternoon of proposal and a week of governed review. Not because anyone worked faster — because the work stopped being *rediscovered* each time and started being *computed*.

The auditor's report, filed three weeks later, was clean. One sentence from it is worth quoting, because it is the whole book in a regulator's prose: *"The institution was able to produce, on request and without delay, complete records linking each system change to its authorizing policy, its approver, and its effects."*

## WHAT YOU NOW KNOW

- Regulation shifted from auditing intentions to auditing evidence — *what can you prove, and how quickly?* — and the five properties of Chapter 2 read as an examiner's checklist.
- The audit chain answers as queries what binders answer in weeks: approvals by distinct named authorities, cascades as proof of matched communications, trust postures, refusals-as-evidence, both clocks, signed scoped exports.
- Directives now enter as policy refinements: the cascade computes the compliance work, and "how did you implement it?" has an identifier for an answer.
- The auditor's judgment is not replaced but concentrated — on whether the declared rules are the right ones. Which is the question that was always supposed to get the time.

*What this chapter has not shown: the limits. The book owes you those before it lets you go — and they get the final chapter, undiluted.*

## CHAPTER 13

# Honest boundaries

---

*What this discipline does not do — said plainly, because trust is the whole subject.*

---

A book about trust ends honestly or it refutes itself. This chapter is the ledger of what the discipline cannot do, will not do, and has not yet done — written with the same care as the promises, because you will bet real systems on knowing the difference.

## §1 — THE GATE PROVES WHAT YOU DECLARED

Every verdict in this book checked a change against *declared* rules — declared bounds, declared policies, declared authorities. Nothing checks the rule you never wrote down. If nobody declared that penalty timing matters, the gate will happily land a change that breaks penalty timing, four green stages and all. The discipline converts unknown risks into declared rules *over time* — every incident becomes a declaration, and the map improves monotonically — but at any given moment, your blind spots are exactly as blind as before. Tests, scanners, monitoring, and experienced humans hunt the undeclared; the gate governs the declared. Both jobs are real. Neither does the other's.

## §2 — VALID IS NOT WISE

The gate answers *may this land?* It cannot answer *should it?* A change can be typed, bounded, within policy, fully cascaded, properly authorized — and still be a bad product decision, a clumsy wording, a strategically foolish move, beautifully validated. Wisdom — taste, judgment, knowing what the customer will feel — is not a stage in the pipeline, and this book has been careful never to pretend otherwise. The discipline's honest ambition is narrower and, lived with, more

valuable: it *spends judgment where judgment is needed*, by disposing of everything checkable before a human looks. What the human then decides is not the gate's business.

### §3 — DECLARING IS WORK, AND ADOPTION IS CHANGE

The whole machine runs on declarations — points, bounds, policies, satisfies edges, authorities — and declarations are authored by people, maintained by people, and neglected by people. The lift reports what it cannot find; someone must still close the gaps. A team that declares nothing gets a gate that governs nothing.

And adopting the discipline is organizational change, not a tool install. Review becomes reading cascades; authority becomes explicit (which some enjoy less than ambiguity); refusals land on humans with human feelings — the four-part grammar exists precisely because a gate that refuses rudely gets routed around. Budget for the culture, not just the tooling. The bank in this book took eighteen months, and the months were not spent typing.

### §4 — THE TOOLING IS YOUNG

Said once in the front matter and repeated here where it belongs: the **stz** sessions in this book are an idealized rendering of an open foundation whose machinery — the graph, the points, the gate, the cascade, the reversible history — is defined and real, and whose tooling ships in stages. Command names will drift; rough edges exist; the commercial products built on the foundation are further along in some surfaces and absent in others. This book taught the discipline, not a version number. The judgment of *when* the tooling under it is ready for your penalty cap is — fittingly — yours.

## §5 — THIS IS A BET

Finally, the widest honesty. This book has argued one side of a live question. There is another view of the agentic era — coherent, well-funded, seriously held — in which the artifact stops mattering: code becomes disposable, systems are regenerated from intent on demand, and governing changes to an artifact is nostalgia, because there is no artifact, only the agent and the ask.

The discipline bets the other way: that consequential systems — the ones holding money, health, and law — will keep needing a persistent, governed, explainable artifact, precisely *because* someone must answer for it on Thursday. The evidence of the regulated world leans this way; the bet is stated so you can judge it, not so you can inherit it unexamined.

## §6 — MONDAY MORNING

None of the limits above excuse waiting, because the discipline's first steps need no tooling at all. Five lines, one workweek each:

1. **List your knobs.** The ten most consequential adjustable things in your system — rates, thresholds, limits, wordings. That page is your first refinement-point inventory.
2. **Write their bounds, and who may change them.** One sentence each. You have just drafted your constraint and governance stages.
3. **Tie each to the rule it serves.** Which regulation, contract, or decision does each knob answer to? Those are your first satisfies edges.
4. **Make one undo cheap.** Pick the change you fear most and pre-compute its reversal, tested, before you next need it.
5. **Demand proposals, not edits.** From this week, anything autonomous — script or agent — *proposes*; a declared rule or a person disposes. That single sentence of policy is the gate, in embryo.

A team that does these five things has begun practicing refinement-oriented programming with a notebook. The tooling makes it fast. The discipline is what makes it work.

## WHAT THIS BOOK HAS ESTABLISHED — AND WHAT IT HAS NOT

**Established:** that the bottleneck moved from writing changes to trusting them; that trust at machine speed must be a property of the change, not a feeling about the author; that a refinement — typed, bounded, validated, attributed, cascade-aware, reversible — carries those properties by construction; that four mechanisms (graph, gate, cascade, chain) produce them; that the same law governs humans and agents, code and interfaces and data, one language or many; and that the result is auditable in minutes because the audit is the workflow's exhaust.

**Not established, by design:** that the gate catches what you never declared; that valid means wise; that adoption is easy; that the tooling is finished; or that the industry will choose governed artifacts over regenerated ones. Those are, respectively: your vigilance, your judgment, your leadership, the foundation's roadmap, and the era's open question.

The jungle, at least, is behind you. What you build on the clearing is yours.

...

## C O D A

## The next letter

---

On a Tuesday in June — it is always a Tuesday — a letter arrived at Softibank Niger from Dakar. Amina recognized the envelope before she opened it. A revision to provisioning ratios this time, effective in sixty days, with the usual annexes.

She read it twice, out of habit. Then she opened a terminal and typed the proposal — a policy refinement, compliance authority, reason field quoting the circular's number. The cascade came back: forty-some nodes, two of them in wording, one in a report she had forgotten existed. She scheduled the review for Thursday, put the letter in the drawer, and went to lunch.

It is not that the letters stopped mattering. It is that they stopped being emergencies. The distance between those two sentences is everything this book was about.

...



## APPENDIX

# The words, pinned

---

*Every contested term in this book, pinned to the one meaning it carries here.*

---

**Refinement.** A typed, bounded change to a declared point in a system — validated against declared rules before landing, attributed to its author, aware of its cascade, reversible by construction. The unit of work this book is named for. (*Chapter 3.*)

**Refinement point.** A declared, named, bounded knob — a parameter, a wording, a rule, an algorithm choice — marking where a system was built to be changed. The system's published change surface, and an agent's affordance map. (*Chapters 3, 9.*)

**The graph.** The system held by its meaning: functions, types, policies, tests, requirements as typed nodes; their real relations as typed edges. Not a diagram *of* the system — the system, read properly. (*Chapter 4.*)

**Satisfies.** The edge tying code to the rule it exists to honor. Makes compliance questions computable in both directions. (*Chapter 4.*)

**The lift.** How existing code enters the graph: read by its own language's parser, files unmoved, declared points and gaps reported. Indexing, not migration. (*Chapter 4.*)

**The gate.** The four checks every proposal faces, in order: structural (the language itself validates), constraint (declared bounds and policies), derivation (the cascade revalidated), governance (declared authority). Deterministic; refuses in a four-part grammar; examines proposals, never authors. (*Chapter 5.*)

**The cascade.** The computed blast radius of a proposal — everything that must be re-examined, grouped by depth, bounded honestly by early cutoff (propagation stops where revalidated verdicts come back unchanged). What a reviewer reads instead of a diff. (*Chapter 6.*)

**The audit chain.** The record produced by the workflow itself: every proposal, verdict, landing, and revert, queryable, on two clocks — *in force* (the world's) and *recorded* (the system's). The reason the audit always exists. (*Chapter 7.*)

**Inverse / revert.** Every landed refinement stores its typed undo at landing time; revert applies it as one atomic, governed, recorded act. (*Chapter 7.*)

**Authority.** Who may change what — declared, checked by the gate's last stage, recorded with every verdict. Humans, agents, and rules all hold authority the same way. (*Chapters 5, 8.*)

**Agent.** A mind embodied with a goal, memory, tools, and accountability; in a governed system, an author whose proposals face the gate. Two kinds of mind: **programmatic** (deterministic, auditable, free) and **model-backed** (creative, linguistic, probabilistic) — same interfaces, same gate. (*Chapter 8.*)

**Memory.** An agent's knowledge as a governed view of the graph — what it knew, and when, as a query on two clocks. Not a private store. (*Chapter 8.*)

**RAG / context.** What a model reads before acting — governed form: sources with provenance. Feeds the author; never touches the verdict. (*Chapters 8–10.*)

**Skill.** A capability that verifies: precondition, action, check. A prompt that cannot verify is not one. (*Chapter 8.*)

**Harness.** Where agents are built and run. A different job from governing what they change; the two compose. (*Chapter 10.*)

**MCP.** A protocol door through which tools and proposals reach a system. Doors multiply; the gate never does. (*Chapters 9–10.*)

**Guardrails.** Content classification around a model, at runtime, probabilistic. Upstream of authorship; never the decision on a change. (*Chapters 5, 10.*)

**Evals.** Measurements of a model's average quality. Inform the choice of author; decide no individual change. (*Chapters 5, 10.*)

**Governance.** Three real objects, one overloaded word: of the *model* (what it may say), of the *actor* (what it may reach and do), of the *change* (whether what it did should stand). This book built the third: **change-governance**. (*Chapters 1, 10.*)

**Trust posture.** The recorded origin of executed code — trusted, external, or model-composed-in-a-sandbox — stamped on proposals, checked against declared trust floors, preserved in the audit. (*Chapter 9.*)

**Wise coding.** The inversion of vibe coding: the system measures the gap between what a complete answer requires and what it has heard, and asks the next well-structured question; answers arrive in any register (including examples); candidates are checked against declared rules; enumerate-or-refuse, never a silent guess. (*Chapter 9.*)

**Valid / wise.** Valid: passed the gate. Wise: was a good idea. The gate produces the first and defers the second to humans — and no amount of tooling changes whose job the second one is. (*Chapters 5, 13.*)

**Softanza.** The open-source foundation this book's machinery belongs to — an engine and library holding the graph, the points, the gate, the cascade, and the reversible history. Free, local, nothing to buy. (*Front matter.*)

**Refine.** A commercial IDE built on that foundation, whose visual surfaces illustrate several chapters. One product among what the foundation makes possible — not the discipline itself. (*Front matter.*)

**stz.** The command idiom of this book's terminal sessions — an idealized rendering of the open foundation's machinery. The commands may drift; the machinery is defined. (*Front matter; Chapter 13.*)

*If this book is useful to your organisation, I work with organisations on its subjects: paper copies in quantity, workshops and training, talks, consulting and the building of solutions. Write to [kalidianow@gmail.com](mailto:kalidianow@gmail.com).*

How these books are offered: <https://mayouni.github.io/ayouni/offer/>

Free to read and share under CC BY-NC-ND 4.0; code under the library's licence.

<https://mayouni.github.io/ayouni/licence/>



# Refinement-Oriented Programming

*A discipline for software in the agentic era*

---

**Software is now written faster than anyone can trust it.** An AI agent can draft in an afternoon what a team once built in a quarter — and every organization shipping consequential software has felt the ground move: the bottleneck is no longer writing changes. It is trusting them.

## WHAT IT IS

This book teaches the discipline that answers. Told through a small bank in Niamey facing a regulator's deadline, it shows how every change — human or machine — becomes typed, checked before it lands, attributed, and reversible.

## WHO IT IS FOR

You do not need to be a programmer to read this book. If you are a programmer, you will find the terminal sessions and the mechanics precise enough to act on. Both readers get the same book; they just linger on different pages.

*Stop trying to govern the intelligence, and govern the change.*

---

**AYOUNI**

SECOND OF THE AYOUNI BOOKS

Free to read, in every form

[mayouni.github.io/ayouni/  
books/refinement/en/](https://mayouni.github.io/ayouni/books/refinement/en/)

