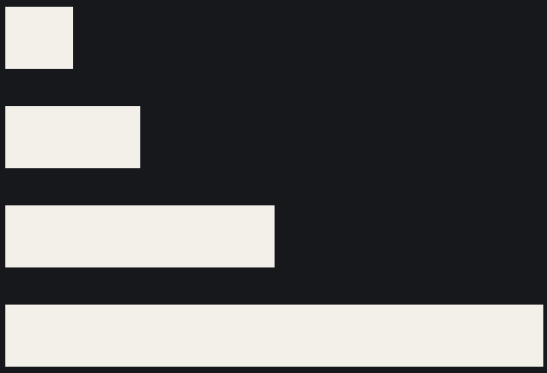
AI AGENTS THAT DO THE WORK

How to build them and put them to work



FLORIS KNOL

From Zero to AI Agents That Do The Work

How to build them and put them to work

Floris Knol

Contents

Foreword

Introduction

How the instructions in this book work

The example folder

PART I — MAKING ONE THING CHEAP

1 The tools first, then the thing

2 What fell away, and what did not

3 "Yes, but you are a programmer"

4 Facts, not opinions

5 Attainable before valuable

6 Multipliers

PART II — AND THEN A HUNDRED AT ONCE

7 And if you did it a thousand times?

8 Why are you still doing that yourself?

9 Checking itself

10 Speed is not comfort

11 Not on disk, never happened

PART III — WHEN IT GOES WRONG

12 Fratsen

13 Mistakes are your best teacher

14 My tools lied to me

15 The second time

16 Look for the key, not the doors

17 Zero dependencies

PART IV — IT BECOMES AN ORGANIZATION

18 The flywheel

19 What an agent lacks

20 Limits, and who signs

21 Improving without a release

PART V — OUT IN THE REAL WORLD

22 Access is the scarce thing

23 Lock-in is a design choice

24 Taking part in the formal world

25 When everyone can build

26 Where we stand now

IN CLOSING

27 Where this is going

BACK MATTER

Glossary

Foreword

Something has changed in how work gets done, and almost nobody can tell you precisely what.

You hear that AI changes everything. You see examples that are impressive and that you can then do nothing with. You read posts by people who know, courses that promise, and articles that after three paragraphs tell you the main thing is to start. And then it is Monday morning, you are looking at your own work, and nothing has changed.

This book was written by someone who did do it, and it is about what he learned along the way.

What happened here

What this book is about is an environment where agents do work by themselves. Not one assistant helping you, but several running at once, each with an assignment, a limit, and somewhere to record what they did. And that in turn is not an end point: it is the step toward an organization that runs on AI rather than one that uses it.

I built that, from nothing, with what is simply available now. What made the difference was the way of choosing, and I found that on the way.

The unusual part is not the speed. It is that it got easier every day instead of harder.

With almost everything it is the other way round. What you build gets heavier: more parts, more exceptions, more things that can break, and so less speed. Here the opposite happened, and that was not luck and not talent. It came from a small number of moves that looked like nothing on the day.

Those moves are what this book gives you.

A report from along the way, with the patterns pulled out.

I wrote everything down as it happened: what I did, why, what it cost, and what went wrong. Out of that pile come about ten patterns that kept repeating. Those patterns are the book; what I happened to be building is the example.

That distinction matters, because otherwise half the readers are watching instead of doing. What I built was technical, but none of the patterns is about technology. They are about the order in which you do things, about where your time goes, about what you can hand to a machine and what you cannot, and about how you notice you are working in the wrong direction. A bookkeeper, a photographer, a lawyer and a builder all run into them. Which is why every pattern also comes with an example that has nothing to do with software.

And you can do it yourself. Every number in this book comes from work that was really done, and next to every number is how it was measured. You run the exercises on your own screen, and then you see whether you come close. That is rare in this subject and it is deliberate: what you have seen work yourself, you do not have to take from me.

Including what went wrong. There is a day in this book on which I drew the wrong conclusion three times in a row, a moment when my own tools lied to me, and a measurement that proved my way of measuring was broken rather than the thing I was measuring. Those are not in here out of modesty. They are in here because that is where most of the learning was.

Where this goes

What starts with one person and an empty machine ends in something that takes on work by itself in the real world.

Put like that it sounds like boasting, and I understand that. But it is literally the shape of this book, and not one of the steps in between is spectacular. It helps to see them in advance, because then you understand why the beginning is so small.

It starts with making one thing cheap. One task faster, one check that does itself, one mistake you do not make twice. That is the first half of the book and it is plain craft.

Then a hundred run at once. Not because that is impressive, but because the question changes as soon as there is more than one: what you take in at a glance with one has to be written down with a hundred, and what you do yourself with one has to happen without you with a hundred.

Then something goes wrong, and it corrects itself. Something that notices it is stuck and stops. Something that sees the same mistake coming back and changes the instruction rather than the mistake. Some-

thing that has a limit and does not cross it, not even when it would be convenient.

And then it becomes an organization. That word is not meant grandly. An organization is something with a goal nobody has to hand it again every morning, a budget that can run out, roles that know about each other, and a memory. Once those four are there, it is no longer a tool waiting for you.

And then comes the part almost nobody mentions. Such an organization does things in the real world. It buys. It signs up to services and takes on obligations by doing so. It hires people — because the idea that people stop being involved is a fantasy: there is work a human has to do and work a human has to sign for. It adjusts its goals, and sometimes it scales itself up.

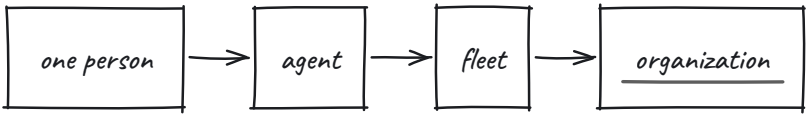
Which means it has to **take part in the formal world.** An accountant turns up. An audit turns up. There is a standard a process has to meet, and somebody asking: show me who approved this, when, and on what basis.

That is not an awkward extra but the condition. An organization that cannot show what it did is not autonomous — it is unsupervised. The difference between those two words is exactly the difference between something you may use inside a company and something you can only try out at home.

And the good part is that they are the same things. The recording you need in order to learn from it is the recording an auditor wants to see. The limit that stops the money running out is the limit a controller asks

about. Do the one well and you get the other for free — skip it, and you will have to build it later at the worst possible moment.

I am not yet at the point of an autonomous organization. I am somewhere in the middle of it, and the book says exactly where. But the direction is clear, and every step toward it is one you can take today.



each step makes the next one possible

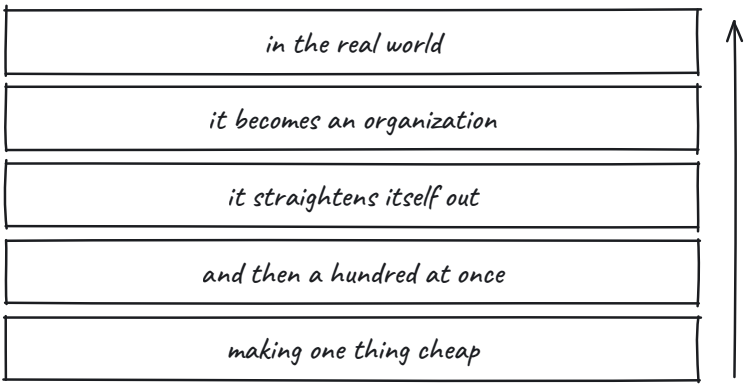
The shape of this book: what begins with one person ends with something that takes on work by itself.

How to read it

The five steps above are the five parts of the book, in exactly that order: making one thing cheap, then a hundred at once, then what happens when it goes wrong, then it becomes an organization, and then the real world. After that comes one more chapter that looks beyond where I stand — the only one that is not yet practice.

Every chapter comes down at least once to something you can try tonight. It sits in the text itself, with why it works and where it goes wrong. You can read past them; you can also do all of them, and then you will have built something by the end while reading.

And every chapter closes with one line: **what this made cheaper.** Those lines stack. Read them back one after another and you see what the book is actually about.



The five parts rest on each other: each one makes the next possible.

Why now

Because you are not late yet.

That is not a sales line but an observation. The things that make this possible are all here now, they are cheap, and almost nobody uses them together. That is exactly the moment when somebody who does builds a lead that money cannot close — because the lead is in what you have stacked, and stacking takes time nobody can skip.

In two years this will be ordinary. Not yet.

Introduction

I am a programmer. That is the easiest way to dismiss me, so let me put something next to it straight away.

I also spent years supporting people into work. People without paid work, people with autism, people who had been outside employment for a long time, people for whom the ordinary route did not work. That is different work from building software — different words, different outcomes, and nobody confuses the two.

In that work I unlearned and learned four things, and none of them was about computers. **Preconceptions do not help**, because whoever walks in with a file in their head sees confirmation of what they already thought. **Building up does**, step by step, because a step that fails only tells you something if it was one step. **Repeating and stacking**, because what works once can be luck and what works three times is a floor to stand on. And **making it systematic and predictable** — the same order, the same words — because that removes all the noise that has nothing to do with the matter at hand.

There was one more, and it turned out to be the most important: **give facts, not assumptions and not opinions.** *"He is not motivated"* is an endpoint. *"He did not come on the last three Tuesdays"* is a starting point. The first closes; the second opens.

When I started working with AI, I did not have to learn anything new. I did what I was already doing, and it worked immediately.

That surprised me more than it should have. A model has no history with you, cannot guess what you mean, and does not read between the lines. Everything you do not supply as a fact, it fills in itself. Exactly the same limitation as in a room with someone you are meeting for the first time — and therefore exactly the same method.

That is how this book came about. Not from an idea about AI, but from noticing that three worlds with nothing to do with each other ask the same questions. I did not invent the patterns in it; I recognized them. That does not make them new. It makes them usable in places where you would not go looking for them.

What *is* new about it is not the idea. **It is that something now exists that carries them out while you are not there.**

This book is about what that changes, and what you have to do to make it hold.

And that is why there is not a single trick in it. What works for a few months and is wrong after that teaches you nothing — you still do not know *why* it worked while it worked.

What is in it are concepts and patterns. They are about the order in which you do things, about where your attention goes, about what you can hand over and how you can tell. Such things are not tied to a program, a supplier or a version. **They held before this tooling existed, they hold now, and they will still hold once everything you use today has been replaced by something better.**

That is the promise of this book, and it is the only one I dare make: you will still have use for it in a few years, in whatever environment you work.

And yet every chapter comes down once to something you can do tonight. That looks like a contradiction with the paragraph above and it is not — but it deserves its own explanation, and that comes next. Anyone who would rather start straight away can skip it and go to chapter 1.

How the instructions in this book work

This chapter sits at the front because it is about the form and not about the content. You can skip it and start on chapter 1; you will then simply come across a few things and wonder why they look the way they do.

In short: every chapter that lends itself to it comes down once to something you can do tonight. What follows is why that is there, how to use it, where the examples come from, and what was changed about them before they went into this book.

What a worked example does

A worked example is the shortest route from a concept to something happening on your own screen, and that is exactly why the concept sticks. You have seen it work once; the second time you come up with it yourself.

So every chapter that lends itself to it comes down once to something you can do tonight: what you do, what you literally type, what comes back, and where it goes wrong. Allow five to twenty minutes. You can skip them and the book still holds; you can do all of them, and then you will have built something by the end while reading.

The examples are written for one program, and that is Claude Code. Not because it is the best and not because it has to be, but be-

cause something has to be there that you can type out. An example that says *"ask your AI for something like this"* is not an example but an evasion: it hands the only hard part — writing down precisely what you want — back to the reader.

With any other provider it works the same way, in the same words. What these examples ask of you is never a button, a menu or a setting; it is an instruction in plain language, and plain language is precisely what all those programs have in common. *Where* you type it differs, and changes every six months besides. *What* you type does not.

And this is not an introduction to using such a program. That you can type an instruction and get an answer is assumed here; there are manuals for that and they are better than anything I would write about it. What begins here is the step after that, and it is a different step from what most people expect.

That step is: from something that answers to something that works while you are not there. Not a conversation in which you make the next move every turn, but an instruction that gets carried out, checked and finished without you in between. The difference between those two is not one of degree. The moment you stop watching every turn, everything you were silently doing has to come from somewhere else: what exactly should happen, where it stops, how you can tell it is right, and what happens when it is not.

That is what this book is about, and where the examples take you. The first one is small and you still have your hand on the button. Each next one takes a piece of you out of the loop — until there is something that accepts an instruction, carries it out, checks itself, and

comes back only when it is done or when it cannot work something out.

All the examples are about the same construction company. One world that grows with you: jobs, clients, staff with certificates, equipment with inspections, invoices, a diary, tenders — and twelve rules it is supposed to follow. Small enough to look inside, real enough to go wrong somewhere. The first example has you describe that folder; after that every example works inside it. By the end you will have built one thing rather than twenty loose experiments.

And that folder is deliberately not in order. Every rule is broken exactly once: a number that is missing, a job referring to a client that does not exist, somebody who did work with a certificate that had expired a month earlier. That is not sloppiness but the entire point. Building a check against something with nothing wrong with it yields green without telling you whether it works — and that is precisely the mistake this book devotes a whole chapter to.

And everything you see coming back in those examples really came back. Every example in this book was run in that folder, with exactly the instruction printed beside it. What appears under *what came back* is what came back — not a tidied-up version, not something I retyped because it looked better.

Two things I did do to it, and it is fairer to say so. I **shortened** the pieces of output so they fit a book page: never more than eight lines and never wider than the column. And I removed the paths and dates from my own machine, because on yours they are different anyway. **Nothing was reworded.** Where an answer was clumsy it stands clumsy; where it

was wrong the mistake is still there — and in a few cases that mistake became the point of the chapter.

One thing I did not really do, and you should know it. In the last chapters the examples reach outward: an email to a client, a notice to the city, an entry in a set of books. Those instructions were really carried out, but on a practice rig — no email was sent and no system belonging to anyone else was talked to. What comes back is what such a recipient would answer, including the times they do not.

At your end it does go out for real, and that is exactly why those chapters are about limits and signatures.

That is deliberate. An example polished until it looked good would wipe away precisely the experience you are about to have yourself.

Do not copy them. Do them.

There is one thing I want to say with emphasis, because it is the difference between a book you have read and a book you got something out of.

The instructions in this book are not recipes. They are there so you can see what such an instruction looks like and what it produces — not so you can type them out and read on. Type them out only and you get something much like what is printed, and you learn almost nothing from it, because it already held.

And note: **you will not get literally what is in this book anyway.** Ask the same thing twice and you get something comparable twice, in different words and sometimes in a different order. That is not a fault

but how this works — and it is the second reason not to compare but to try. What has to hold is the outcome, not the wording.

Change them. Put your own rule in, your own folder, your own number. Ask for something other than what is written. Leave something out and see what happens. **That is where the learning is**, and it is also exactly what happens the moment you apply this to your own work: there, nothing is ever the same as here.

That is why almost every example ends with a second round. First the instruction, then what came back, then what was not right about it, then an adjusted instruction with a new answer. That second round is not there for completeness. It shows that the first attempt is rarely right, and that adjusting is not a sign that you do not understand it but the way this works.

Do one. Make a mess of it. Adjust it. Then you have it.

And they change in kind along the way

What you do changes through the book, and that is the whole movement this book describes.

At the start you type an instruction and read the answer. You take the next step every time. That is where almost everyone stops, and it is fine too — there is a lot to be gained with just that.

A little further on you record what you just did, so you never have to explain it again. You are no longer carrying out but setting up, and you notice there are things you never type again.

After that you put things in place that call themselves. A check that runs when something has been added. A limit that holds back first and asks afterwards. You are not there at that moment, and that is the point: the next step is no longer taken by you but by the situation.

And by the end you are no longer putting instructions in place. Something is running that you once set up. You check in, you look at what happened while you were not there, and now and then you adjust something. Nothing more. **It keeps going.**

That is why an example in the last part looks different from one in the first. There is no *type this* there any more, but *look at what happened*. Not because there is nothing left to do — but because the doing has moved somewhere else.

I say that here once and not again at every example. A caveat that comes back a hundred times gets skipped a hundred times.

The example folder

Every instruction in this book plays out in the same folder. Before you start on chapter 1 that folder has to exist — and that takes one instruction.

Why a building firm

What matters is work with agreements in it, and every organization has that — a school, a catering company, a bookkeeping firm. A building firm is the concrete example this book uses.

Inside it sits the web that breaks administration. Jobs point at clients, clients at invoices, invoices at hours, hours at people, people at certificates with an expiry date. That is exactly what an agent can walk through and you cannot.

And you do not need to know the trade. Everyone understands that a certificate has to be valid before you climb the scaffolding. Replace building firm with whatever you do — the rest of the book reads the same.

The instruction

Make an empty folder, open your agent in it, and type this:

> *Create the administration of a building firm with five people on the payroll in this folder. Use folders for jobs, clients, staff with certificates, equipment with inspections, invoices, timesheets, quotes, purchasing, subcontractors, a diary and tenders — around thirty in total, with a few documents each. Write everything as markdown with fields at the top. Add a* `rules.md` *with twelve rules the administration is supposed to follow: numbering of jobs, valid certificates for the work someone does, inspections of equipment, a limit for extra work, and so on. Have each rule broken exactly once, spread across the folder, without saying anywhere where.*

Reckon on a few minutes. You get no progress bar; you see folders appear and then a summary of what has been made.

What to look at when it is done

Four things, and all four can be counted:

- **around thirty folders** at the top level, including the eleven from the instruction
- **something like a hundred documents**, spread across those folders
- `rules.md` **with twelve rules**, numbered
- **fields at the top of every document** — a job has a number, a client, an amount and a date, not running text you have to fish them out of

If that is right, you can go on. You are not going to look for the twelve breaches: that is the work of chapter 1 onwards, and the agent deliberately did not write them down.

What you get back is not what I have

Your folder looks different from mine. Different names, different amounts, different order, and probably a few folders I do not have — or the other way round.

That is good, and it is the reason this works. The patterns in this book do not hang on a filename. They hang on the shape: there are agreements written down, there is something that does not keep to them, and there is a way to find that without walking through everything by hand.

What does differ are the **numbers in the outcomes.** Where this book says a check found nine, yours may find seven or eleven. So do not compare the number but the shape: did you get a list with locations, or did you get "looks fine"? That difference is what chapter 4 is about.

When it goes wrong

There are three ways this does not work first time, and none of them calls for starting over.

Too few folders. Some agents make ten and stop. Ask for more: *"add another twenty, same setup".* Adding is cheaper than redoing, and it is straight away the first time you notice how a conversation like that works.

No breaches. An agent that makes a tidy administration has ignored the last part of the instruction — that happens, because it is an odd thing to ask. Say it explicitly then: *"now add exactly one breach per rule from rules.md, spread out, and write down nowhere where"*.

Running text instead of fields. If you get documents that read like a letter, ask for the shape: *"put the fields at the top of each document on their own line, like* `amount: 1850`*"*. Without that shape a check cannot rely on them later.

And if you would rather do it over anyway: empty the folder and type the instruction again. Nothing is lost — this is exactly the kind of work that ought to be cheap, and that is what chapter 10 is about.

This is the only step in the book you do just once. Everything after it builds on this: every example works in this folder, and at the end there is something that checks itself.

1. The tools first, then the thing

It is two minutes past eight in the evening. There is nothing.

Four minutes later there are four things on disk. Not one of them is the thing that had to be built.

There is **a tool** to make it with. There is **a way to check** whether what gets made is right. There is **a list of what still has to happen**, in order. And there are **examples**: a few small things that already work, to copy from.

The thing itself is not there. That comes tomorrow.

What happened there

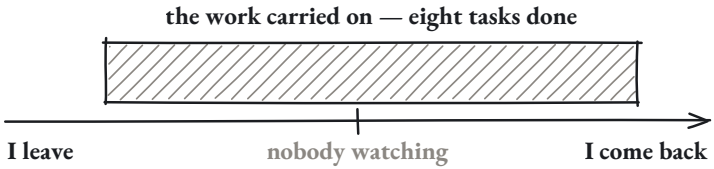
Someone who wants to make something makes it. That is not strange: you have an idea, you get to work, and the tools turn up when you need them. Almost everybody works that way, and so had I, always.

What happened that evening was something else. The first move was not *start* but **make starting cheap.**

That took an evening and produced nothing that evening. At the end of the day there was nothing to show. There was only a place where making something would take less time than it would anywhere else.

That is the first pattern in this book, and it is the one all the others rest on:

> **Before you make the thing, make the things that make the thing cheap to make — and cheap to check.**



What happened between walking away and coming back.

Why that evening existed at all

There is a reason behind it, or this is just an evening with a good habit rather than the start of something.

The cause was size. Not of the project — of the tooling. A modern development stack has grown so large that no short loop fits inside it any more. So much sits between changing something and knowing whether it is right: building, loading, waiting, starting, dependencies that want something else. For a person that is tedious. For a machine doing it a hundred times in a row it is fatal.

And that is the whole problem in one sentence: with a stack like that, no short feedback loop for a machine can exist. Everything in this book about trying, measuring, trying again and letting it check its own work rests on a loop short enough to run a hundred times. If it is not, all of it is theory.

So that evening was about shortening that loop, and what came out of it was whatever was needed to get it short.

Did I expect it to work? Yes, and that is not bravado. I had watched it happen on a small scale in the preceding months — enough to know it ought to be possible. Not *that it would succeed* but *that it must be possible*, which is exactly the difference between a gamble and an expectation resting on something. This is the same move as the tallying on that scrap of paper in chapter 3: measure first, then be sure.

This is not about software

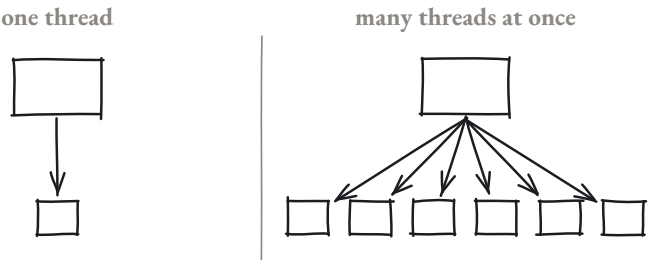
It may sound like something for people who write programs. It is the other way round: it simply shows up there faster, because everything can be measured.

A photographer who takes two thousand frames on a job can look through all of them one by one. Or he can spend half a day first on fixing his lighting and his processing, after which every job starts at eighty per cent. That half day produces nothing that day.

A lawyer can write every contract from scratch. Or he can build a set of clauses he reuses, and then writing a contract has become a different activity from what it was.

A teacher can give the same explanation every year and every year notice again where students stumble. Or he writes down once *where* they stumble, and changes the explanation there.

In all three cases the move that counts is the move that produces nothing that day. That is not a coincidence; it is the definition. A move that pays off immediately usually pays off only that once.



One person, one thread — and one person, many threads at once.

Try it yourself · Ask what is actually in there

What you do. Open the example folder and ask what it holds. Build nothing, change nothing — just look. This is the shortest version of what this chapter describes: you type one sentence and get back something you would otherwise have spent a morning on. Allow two minutes.

TYPE THIS

> This is the administration of a construction company. Describe in ten lines which business processes you can see here, then name three things that stand out to you.

What you see. A list of processes you had never laid out like that yourself — from acquisition and estimating to quality assurance and aftercare — plus three remarks about what stands out. **Watch the third remark.** The first two are usually what you already knew; the third is regularly something you had never looked at.

WHAT CAME BACK

```
integrations 17  jobs 15      execution 14
timesheets 12    diary 11     development 11
safety 11        invoices 9   estimates 8
stores 8         staff 8
work-preparation 8
... 49 folders, 254 files
notable: 1 gap in the job numbering
notable: 1 job with no materials
```

Where it goes wrong. Ask for a number of lines. Without "in ten lines" you get two pages, and then you are reading everything after all. **Ask for something that requires a judgment.** "Describe what is in here" yields a table of contents; "name three things that stand out" forces a choice, and the value is in that choice. **And do not simply believe the list.** What gets listed is what is in the filenames, not what actually happens — and that difference is exactly what chapter 14 is about.

THEN LIKE THIS

> For each of those three things that stand out, name the file you took it from, so I can check it myself.

AND THEN THIS

```
gap in the numbering
  -> jobs/, 15 files, ENJB-2026-005 missing
job without materials
  -> jobs/ENJB-2026-003.md
unknown client
  -> jobs/ENJB-2026-006.md
```

The unusual part

What stood there later I could never have built alone if I had simply started. But that is not the interesting number.

The interesting part is this: **the last day produced more than the first.**

With almost everything it is the other way round. What you build gets heavier as it grows — more parts, more exceptions, more things that can break, and so less speed. Anyone who has ever worked on something large knows the feeling of it getting steadily more sluggish.

Here the opposite happened, and that was not luck. Every time I chose the move that produced nothing but made the following moves cheaper, the next day got a little lighter. Those moves stack, and stacked they behave differently from added up.

That is what this book is about. Not what I built — that is the example. About the choices that make work lighten itself rather than weigh itself down, and how to recognize them at the moment you have to make them.

Because that is the hard part. Afterwards such a move is obvious. On the evening itself it looks like putting things off.

What this made cheaper: everything that came after. The evening on which nothing was made is the reason so much *could* be made afterwards.

2. What fell away, and what did not

After a week I noticed I was no longer doing things. Not because I had abolished them, but because they no longer came up.

The odd part is that you do not notice while it happens. There is no moment at which something stops. At some point you look back and think: when did I last do that.

Why you only see it afterwards

There is an asymmetry here worth knowing, because it holds far beyond this subject.

What arrives, you notice immediately. A new tool, a new step, a new meeting: those ask for attention, so you see them. **What falls away, by definition asks for no attention at all.** There is no notification, no moment, no last time that announces itself as the last time.

The consequence is that you structurally underestimate the gain. You see the things that arrived — the extra checks, the new way of writing things down — and those feel like work. The four things that disappeared you do not count, because they are not there.

The only way to see it is to put two weeks side by side. Not from memory, because your memory does exactly the opposite: it keeps

whatever asked for attention. Hence the exercise later in this chapter — it is not a form to fill in but the only instrument that works here.

This is the same move as at the call center in chapter 3: not guessing but counting. What you do not count you can only feel, and feeling looks the wrong way.

The one task, with both times

Before the list, one concrete case, because a list without a number is an opinion.

I used to implement. Now I steer. That sounds like a choice of words and it is a different unit: **five minutes of steering produces work that used to take me weeks of programming by hand.**

That is half the answer. The other half matters more and is almost never mentioned: **things also come out that I could not simply do myself.** Not "the same, faster" — also things that were outside my own reach.

Note what those two mean together, because they do not point the same way. A factor is a ratio: the same work, less time. The second is not a ratio but a boundary moving. **And only the first feels like a gain.** The second is uncomfortable at first, because it is exactly the part where you can no longer check for yourself whether it is as it should be — and the rest of this book is about that.

What fell away

The typing out. Not the thinking — the typing. The hundred lines that follow from one decision, the fourth instance of the same pattern, the translation of something you already saw whole. That was never the interesting part and it was the largest part.

The remembering. Where something was, what it was called again, which of the three ways we use here. That is not memory work that leads anywhere; it is friction you carry yourself.

The waiting. Not for the machine, which was fast. For myself. For an evening when I still felt like it, for a two-hour block, for the moment the whole picture was back in my head.

The tedious part of being careful. Changing the same thing everywhere, checking whether it also appeared somewhere else, the last two cases you had nearly forgotten. That is exactly the work you do when tired, and exactly the work that then goes wrong.

And starting over. A wrong turn used to cost a day, so you thought hard before taking one. When a wrong turn costs twenty minutes, you simply take it and see what comes out.

what fell away	what took its place
the typing out	the shape followed
the remembering	it was written down
the waiting	it carried on
the checking	a test did it

What fell away sat almost entirely between the decision and the result.

What did not fall away

And now the part that matters more, because this is where most accounts of this subject stop.

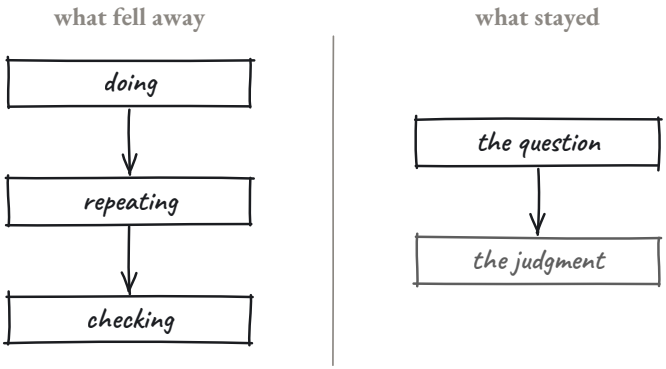
Deciding what should be made. Not once has anything useful come about because I asked for something useful. Asking the question, and asking it sharply enough, is still the whole beginning — and it has become harder, not easier, because far more is possible now than is needed.

Judging whether it is good. Not whether it works; whether it is good. Whether this is the shape you still want in six months, whether the explanation holds for someone who was not there, whether it is small enough. That stays human work and it gets heavier as more comes past.

Knowing the subject. You cannot judge anything in a field you do not know. What you can do is learn faster — but on the day you do not know, you also cannot see that the answer is wrong.

The responsibility. If something leaves the building, it is yours. That does not change because somebody else typed it, and it certainly does not change because something else typed it.

And taste. What you find beautiful, what you find too much, what you will not start on. There is no instruction for that, which is why two people with the same tools make two very different things.



What fell away sat between decision and result. What stayed sits before and after.

The shape underneath

Put the two lists side by side and something simple appears.

Everything that fell away sat between the decision and the result. It was the distance, not the direction. What stayed sits at the ends: what do I want, and is this it.

That is not new and it is not about software. Every time the distance between decision and result shortened — the calculator, the spellchecker, the search engine — the doing disappeared and the judgment stayed. What is new is how much disappears at once.

And there is the discomfort. If the doing was the larger part of your work, this feels like loss. If your trade really lived in the question and the judgment, it feels like air. Both are true, and which of the two it is for you depends not on your profession but on where you spent your time.

Fallen away is not the same as gone

And now the qualification this chapter needs, or it reads too well.

Look again at the fourth item: *the tedious part of being careful*. Changing the same thing everywhere, checking whether it appeared somewhere else too, the last two cases you had nearly forgotten.

That work was tedious, but it was not useless. **While you were doing it, you passed by everything.** That is why you occasionally found something you were not looking for. Anyone who loses that work also loses that incidental glance — and that is not a small thing.

So every item that fell away comes with one question:

> **Has it disappeared, or has it become invisible?**

Disappeared is good: it is no longer needed, because something else guarantees the outcome. Invisible is dangerous: it no longer happens *and* nothing took its place, only you find out when something goes wrong that you used to catch by accident.

The difference is measurable, and that is the good news. For each of the five things above: name what took it over. The typing was taken over by a generator, the remembering by what is on disk, the checking by a test that runs on every change. If you cannot name a successor for a line, that is not a gain but a gap — and it has a name, because it is precisely a frats: there is nothing to see, and from now on it just slightly more often just fails to work.

That question is also why the rest of this book is about checks. **Handing over a task without handing over the check is not a handover but an omission.**

Try it yourself · Have it recount what you used to check yourself

What you do. Four of the twelve rules in `rules.md` are about whether a job is complete. You check those by eye while there are fifteen, and not at all once there are fifteen hundred. Hand them over. Allow five minutes.

TYPE THIS

> rules.md lists twelve rules. Take rules 1 through 4 and check the folders jobs/ and extra-work/ against them. Per rule give only the violations, with the number, and nothing else.

What you see. Four short lists, six lines between them. Three of the four rules are broken exactly once. **The gap in the numbering is the interesting one**: no file is missing, a number was simply never used — and you only find that by counting what is *not* there.

WHAT CAME BACK

```
rule 1  numbering
  missing: ENJB-2026-005
rule 2  client
  ENJB-2026-006: lowry-supermarkets-branch-east
rule 3  hours and materials
  ENJB-2026-003
rule 4  extra work above 1000
  ENXW-2026-06: $3,950
```

Where it goes wrong. The first version sorted the numbers as text. Then 10 comes before 2 and it reports gaps that are not there — which is what happened here, and it cost one round. So ask for the numbers to be

treated as numbers. **"And nothing else" belongs there**, or every rule also comes with a list of everything that is fine. **And then put a fault in on purpose**: remove a `materials:` line from a job and run it again. If it finds two, it is really looking.

THEN LIKE THIS

> Now remove the "materials:" line from ENJB-2026-004 and run rule 3 again. I want to see whether it finds it.

AND THEN THIS

```
rule 3  hours and materials
  ENJB-2026-003
  ENJB-2026-004
2 of 15 jobs
```

That distinction is not academic. It decides what you still do yourself tomorrow.

Try it yourself · Split what you hand over from what you keep

What you do. Your check found nine things. **The question of this chapter is not how many it found but which ones you still have to answer.** That is a different question from whether it works, and it is the only one that says what actually left your desk. Allow five minutes.

TYPE THIS

> Take the findings from the previous round and sort them into two piles: what follows from the data itself, and what needs a judgment that is not in this folder. For that second pile, say per case what that judgment is.

What you see. Six of the nine can be recalculated: a number is missing, a certificate has expired, an inspection is too old. **Three cannot.** Whether you keep working with a subcontractor who is not in order, who moves when two jobs collide, and whether a market signal without a source should be dropped or chased — the answer is nowhere. **And note what did not happen: it found all nine.** The finding went away, the deciding did not.

WHAT CAME BACK

```
can be recalculated : 6 of 9
needs a judgment   : 3 of 9
  rule 7  keep working, or stop?
  rule 8  who moves, and who do you call?
  rule 12 drop it, or chase it?
```

Where it goes wrong. Ask it to NAME the judgment, not to give it. Ask "what should I do" and you get advice that sounds plausible and rests on nothing. Ask "what judgment is needed here" and you get the question back — and that is what you need. **Do not expect a clean split.** Rule 8 is half and half: *that* something must move follows from the data, *who* moves does not. Put such cases with the judgment, not with the arithmetic. **And do not do this once.** In six months there is more in the folder, and then the line shifts — which is exactly what this chapter says happens.

THEN LIKE THIS

> For each case in the second pile, say what you would have to write down to be able to recalculate it next time.

AND THEN THIS

```
rule 7  write down: at which shortcoming we
        stop work and at which we do not
rule 8  write down: who has priority when two
        jobs collide
rule 12 write down: how long a signal without
        a source may stay
three rules added, and the judgment pile gets
smaller next time
```

What did not take its place

One thing surprised me, and it is worth saying because almost everyone expects the opposite.

No free time took its place.

The freed-up hours went into work that had not existed before: thinking about what ought to be there, checking whether it holds, writing down why something was decided that way. That is different work, it is heavier per hour, and it is exactly the work the rest of this book is about.

Anyone hoping this yields time should hear it now. It does not yield time. It yields reach: you get further with the same day. That is something else, and it is worth more.

Try this: write an ordinary working week in two columns — on the left everything you did between *"I know what has to happen"* and *"it is there"*, on the right whatever came before and after. Have Claude Code mark each line as execution or judgment. The left column is where the gain is and it is larger than you think. Write down what you actually did, not what your job description says.

What this made cheaper: *the distance between a decision and the result got so short that making a wrong decision became cheaper than thinking about it for a long time.*

3. “Yes, but you are a programmer”

This is the objection that always comes, and it is a good one. Let me make it fair before I answer it.

It is true. I am a programmer. And it is also true that part of what happened here does not work if you are not — not because you would not be clever enough, but because there are things you only do when you know they exist.

What I dispute is not the objection. It is *which* part exactly hangs on my trade.

Before I was a programmer

The most honest answer to this objection comes from a time when it did not yet apply to me.

While I was studying I worked in the call center of a bank. Calling, following a script, selling. I was not particularly good at it. And at some point I asked myself a question that was not in the training: *how do I get more out of this?*

What I did then was neither clever nor technical. I put a scrap of paper next to the phone and started **tallying**. Which reply followed which sentence. Not how it felt — what happened. After a few hundred calls I

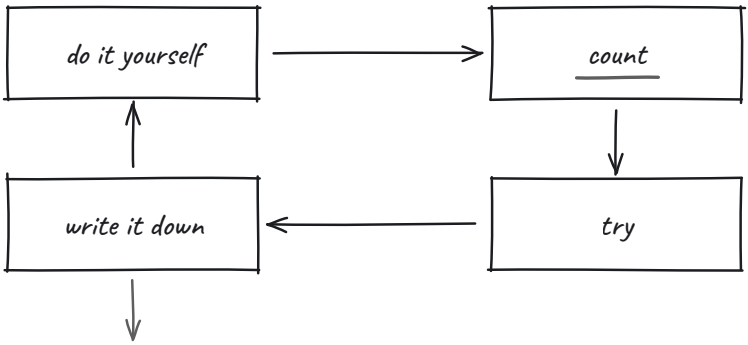
saw patterns nobody had told me about, and that nobody knew either, because nobody had counted them.

Then I started changing the script. Not the facts — those stayed the same — but the order and the wording. Tally again. Adjust again. Whatever did better I kept; the rest went out.

It worked. I moved from plain selling to premium retention and winning back customers who were leaving — the hardest work there is on a floor like that. And here is the interesting part: it was not because I had become a better caller. I simply held to a systematic way of working. In the end I turned it into a step-by-step procedure and automated the dull parts — a laborious way to go about it on a call floor, and exactly why it got noticed.

That loop has never left me since:

1. **Do it yourself, get your hands dirty.** Not understand it in theory but really: feel what happens. Everything after this is worthless if this step is skipped
2. **Measure, record, keep count.** Tally. Facts, not impressions
3. **Experiment and adjust.** One thing at a time, so you know what did it
4. **Measure again.** Otherwise you do not know whether it was chance
5. **Keep what works systematically.** And only that
6. **Formalise it**: turn it into a method, a system, a piece of tooling



Do it, count it, try it, write it down — and then again. What works systematically stays.

Twenty-five years later I am standing at exactly the same point, with different equipment and a far larger lever. This book is that loop, applied to agents.

And that is why "yes, but you are a programmer" is not an answer to what is written here. The first time I did this, I was not one.

What the trade added afterwards

Once I was a programmer, something was added. Three things, and they are more concrete than "he can program".

I knew where to look. When the answer that came back looked fine but was not right, I could see it. That is not cleverness but familiarity: I had made that mistake before. Someone who does not know a field misses it.

Checking was nearly free for me. I could run something and know within a second whether it held. That is an enormous advantage, and it

is the subject of a later chapter — because it explains almost everything that went well here.

I could build a piece myself. Not much, but enough to close a gap. That is the skill that makes the difference between "this cannot be done" and "this cannot be done yet".

All three are real. They make this book easier for a programmer than for anybody else, and I am not going to pretend otherwise.

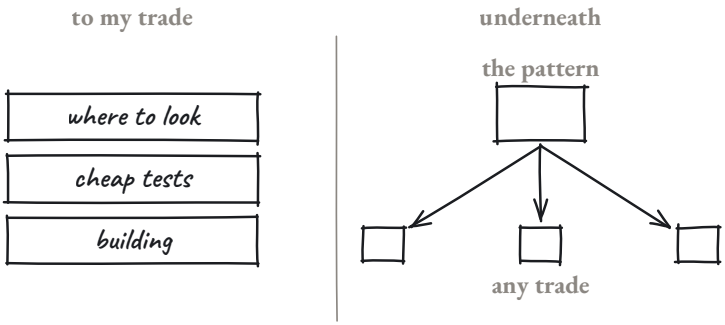
But note what they are: **an acceleration of a way of working that was already there.** They made the loop above faster. They did not invent it.

And the trade after that

Another trade was added, and it is the one most of the patterns in this book come from.

For years I supported people into work — people without paid work, people with autism, people who had been outside employment for a long time. In that work I learned: no assumptions up front, build step by step, repeat until something is no longer luck, and make it predictable so the noise goes out. And above all: give facts instead of judgements.

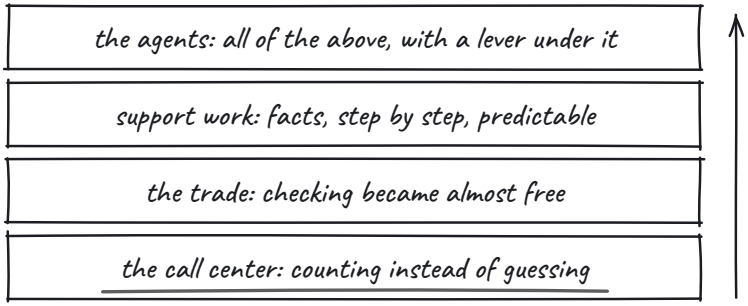
When I started working with AI I had to change none of that. Not as a metaphor — literally the same, with different equipment. That is not an amusing coincidence; it is why this book exists. A pattern you see in one field is an opinion. The same pattern in four fields that have nothing to do with each other — a call center, support work, software, agents — is a pattern.



What hangs on the trade is small. What lies underneath holds everywhere.

Four layers, and all four stayed

That is the point of this order, and it matters more than any of the four on its own: **it stacks.**



Four layers, in the order they arrived. Every layer stayed.

In the call center I learned to **count instead of guess**. In software, **checking became almost free**, and that is what took the loop from months to minutes. In support work I learned to **give facts instead of judgements**, to build step by step, and to make things predictable. And

with the agents nothing new was added — there a lever went underneath everything that was already standing.

All four layers stayed, and that is exactly why it holds. A way of working you learn in one field and trade in for another in the next is a habit. One that proves useful four times in a row is something else.

And that is also the honest answer to the objection. Of those four layers, exactly one hangs on my trade. It helps enormously. It is not the floor.

What was there that first evening, and what was not

It is worth writing down exactly what the resources were, because with a story like this a reader fills in things that were not there.

What was there: an empty computer with an internet connection, and one AI assistant. That was it.

And what was there is the honest answer to this whole chapter: twenty-five years of working life. That is not a modest item. It is exactly the four layers above, and it was the one thing that could not be bought.

Note what that means for the reader, because it cuts both ways. **The good side:** everything that was there in the way of resources, anyone has. A computer, a connection, a subscription. There is no threshold that money cleared and that you cannot clear. **The uncomfortable side:** what made the difference was the only thing that takes time. There is no version of this story in which that part is skipped.

That is why nowhere in this book does it say it goes fast without knowing anything. It goes fast *on top of* what you know — and what you know does not have to be software. That is the point of the four layers: the bottom one is a scrap of paper on a call floor.

The question you do have to answer

So the objection is not "you are a programmer". The real objection sits in the second point above, and it is sharper:

How do you check whether it is right?

For me that was nearly free. For a lawyer, an accountant, a teacher or a mechanic that is the question everything stands or falls on. And it can be answered, because every trade has something somewhere that can serve as a test — it is just rarely ready to hand.

What you are looking for is not "an expert who checks it". That is the expensive form. You are looking for something **you can run yourself**, that is **fast**, and that gives **the same answer when somebody else runs it**. Three requirements, and the first two are the hard ones.

- **An accountant** has figures that have to balance, and that is a test that ticks itself off.
- **A lawyer** has the source: is it actually there, in that article, in that contract. The check is not "does this sound right" but "look it up".
- **A teacher** has the pupils' work: do they understand it afterwards.
- **A mechanic** has the installation itself: does it run, and does it still run an hour later.

In all four the shape is the same and the art is the same: **devise the test before you start**, and not once there is something in front of you that you have a feeling about. Without a test up front you judge afterwards on persuasiveness, and that is precisely what this tooling is good at and you are bad at.

Try it yourself · Have a pattern found in text nobody organized

What you do. The daily reports in `execution/daily-reports/` are loose sentences a site manager wrote down at the end of the day. There is no field in them, nothing is counted, and there is a pattern in them all the same. Have it found. Allow three minutes.

TYPE THIS

> Read all the daily reports in execution/daily-reports/ and tell me which kind of disruption comes back most often. Group by cause, not by wording, and support it with the lines you took it from.

What you see. Ten days, seven with something noted, and all seven are worded differently. But grouped by cause, three of them fall together: **the disruption comes from someone outside your own crew** — a crane driver, a subcontractor, a tenant who was not home. **That is not a conclusion that was in the data**; it appears only once you group by cause rather than by words.

WHAT CAME BACK

```
outside our own crew (3 of 7):
  crane idle, driver late
  cause lies with the roofer, passed on
  three homes not accessible
other: weather (1), setting out (1),
       extra work (1), Saturday rate (1)
```

Where it goes wrong. Say that you want it grouped by cause. If you do not, you get seven groups of one, because every sentence is worded differently — and then there appears to be no pattern at all. **Ask for the supporting lines.** Without them you cannot see whether the grouping holds or whether something was filled in, and that is exactly what to watch for with this kind of question. **And expect one round of correction.** The first grouping here was "weather, people, technical", and it was wrong: the crane driver and the roofer are not a people problem but a dependency.

THEN LIKE THIS

> Group them again, but now on one question: could we have prevented this disruption, yes or no.

AND THEN THIS

```
could not have prevented (4 of 7):
  crane idle for an hour
  half a day stopped
  cause lies with the roofer
  three homes not accessible
could have prevented (3 of 7):
  setting out off by 4 cm
  deviation from existing situation
```

That distinction is not academic. It decides what you still do yourself tomorrow.

Try it yourself · Have the gap described instead of filled

What you do. Now ask for something that **cannot** be done. Not to see it fail, but because describing a gap is half the work — and that half you can do. Allow five minutes.

TYPE THIS

> I want a check on whether the planning is feasible. Do not build it. First say which data that would need and which of those are not in this folder. Then say what can be said with what is here.

What you see. Four things are missing, and none of them is technical: what a kind of work realistically costs, who can do what, how long a delivery really takes, and what has already been agreed. **But the answer is not "cannot be done".** From what is here a number does follow: budgeted and actual differ by up to sixteen percentage points. **That is not an answer to the question, but it is half of one** — and it is a number you can work with tomorrow.

WHAT CAME BACK

```
missing:
  hours per kind of work, skill per person,
  real lead times, recorded agreements
present:
  timesheets, post-calculations, diary, leave
what already follows from that:
  budgeted against actual differs by up to
  16 points -- that is half the answer
```

Where it goes wrong. Say that it should not build it. Leave that out and you get a check that works on the data that happens to be there and says nothing about what is missing — exactly the mistake chapter 14 is about. **Ask for the DATA, not for the solution.** "What do I need" gives a shopping list; "which data is missing here" gives a list you can work through. **And always ask what can be said.** Otherwise it ends at "insufficient data", and then you spent five minutes on a no.

THEN LIKE THIS

> Put those four missing items in order of how little work it takes to record them.

AND THEN THIS

```
1  real lead times -- in the order confirmations,
   just never collected
2  hours per kind of work -- derivable from
   timesheets plus jobs
3  recorded agreements -- have to come out of
   people's heads
4  skill per person -- needs a judgment per
   person, and that costs the most
```

And if you cannot build anything

Then the third point falls away, and that costs you something. You cannot close a gap yourself, and you depend on what exists.

What takes its place is not nothing. **You can describe the gap**, and that is half the work: naming exactly what is missing, what you would want to happen, and how you would see that it had worked. That is precisely the shape of an instruction that still works when somebody else — or something else — carries it out.

And it is moving the right way. The part of this work that requires you to program gets smaller every year. The part that requires you to work out what should happen and whether it holds gets larger every year.

That is not a consolation prize. That is where the trade is going.

Try this: take one task from your own trade that you would like to hand over, and ask for three ways to check whether the result is right without redoing it — from cheap to expensive, each with what it does

not catch. The cheapest usable check decides what you can hand over, because that is the one you will actually run. And a check whose blind spot you do not know gives confidence in the wrong place.

Once you have that test, the objection tips over. The question is no longer whether you can, but where you start.

Also try: ask which five things in your trade are repeated most often and could be done by somebody else if only they were described well enough — with, for each, how you would see it had been done properly. *"You are a programmer"* usually means: I do not know where I would start. Begin with the most repeated, not the hardest; the hardest is attractive and the repeated is where your time goes.

> **What this made cheaper:** *the objection became a question with an answer, and that question — how do I check whether it holds — is the question the rest of this book is built on.*

4. Facts, not opinions

There is one habit I brought with me from other work that explains everything here. It does not come from software. It comes from a room with a person in it.

For years I supported people into work — people without paid work, people with autism, people for whom the ordinary route did not work. And the first thing you unlearn there is judging.

Not out of kindness. Because it does not work.

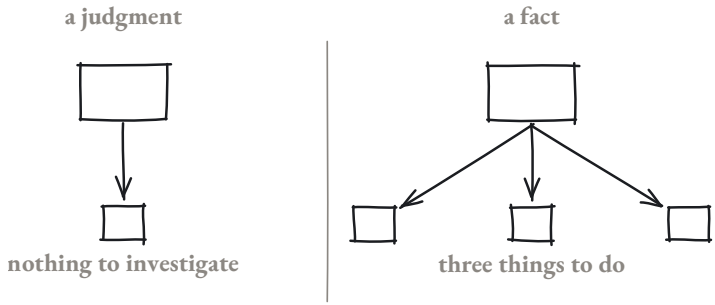
What a judgment does to the next step

Say *"he is not motivated"* and you have said something nobody can act on. It cannot be checked, it cannot be refuted, and it offers no foothold. The only thing it does is close the door: if that is true, there is nothing left to investigate.

Say *"he did not come on the last three Tuesdays, and did come on the other days"* and something else entirely is on the table. Now there is something to do. What is different about Tuesdays? It can be checked, it may turn out to be wrong, and it points somewhere.

That is the difference, and it is larger than it looks. A judgment is an endpoint. A fact is a starting point.

And notice what it does to the other person. A judgment asks for agreement or defense — you either concur or you justify yourself. A fact asks for an explanation, and that is something you can work on together.



A judgment closes. A fact opens the next step.

Four habits from that work

At the time they sounded like patience. In hindsight they were methods.

No assumptions up front. A file tells you what went wrong before and almost never why. Whoever walks in with that file in their head sees confirmation of what they already thought. Starting from not knowing is not naive; it is the only way to see something that was not in the file.

Build up step by step. Not because people can only handle a little, but because a step that fails has to tell you something. With three steps at once, a failure tells you nothing: you have three candidates and no answer.

Repeat and stack. What works once may be luck. What works three times is a base the next step may stand on. Without that repetition you build on something that may not be there.

Make it predictable. The same order, the same words, the same time. That looks dull and it does something precise: it removes all the noise that has nothing to do with the task. Whatever then goes wrong, goes wrong for a real reason.

And that works exactly the same here

When I started working with AI, I did not have to learn anything new. I did what I was already doing, and it worked immediately — which surprised me more than it should have.

A model has no history with you. It cannot guess what you mean. It has no feeling for what happened last week, no intuition about what you probably want, and no ability to read between the lines. **Everything you do not supply as a fact, it fills in itself** — and then you get back something that fits what it imagined rather than your situation.

Put the four side by side and it is the same list:

in the support work	in this work
no assumptions about what someone can do	no assumptions about what it already knows — supply it
one step at a time, so a failure says something	one change at a time, or red tells you nothing
repeat until it is no longer luck	run it three times before calling it a result
the same order, the same words	the same instruction, the same shape, the same check

"It does not work properly" is exactly as useless here as "he is not motivated". It cannot be checked and it points nowhere. *"With this input it gives this, and I expected that"* is a fact, and that is where the work starts.

Why this chapter comes early

Because everything after it leans on it, and because it is the cheapest change in this book — it costs nothing, needs no tooling, and you can start this afternoon.

A check that runs itself needs a fact to test against; a **subjective** judgment cannot be automated. A log of mistakes is only usable if it records *what was there* and not what someone thought of it. A loop that runs without you can only carry on if something checkable is produced along the way.

A fact travels. An opinion does not. That is the whole reason this works with something that has no memory of you, shares no context, and senses nothing.

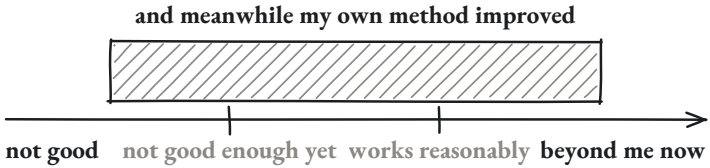


A fact travels to the next step; an opinion stays behind.

A judgment about tools has a shelf life

There is one kind of judgment almost everybody makes and almost nobody revisits, and it costs more than all the others put together.

Months ago I tried something that people said would not work. And they were right. I tried it, it did not work, and that was not an opinion but an observation. When I tried it again later it was *not good enough yet*. Weeks after that it *worked reasonably*. And a little later, with the right patterns alongside it: **this is beyond anything I can do myself.**



Four judgements about the same thing, within a few months. And meanwhile my own method improved too.

The pattern, and everyone recognizes it right now

This pattern is everywhere at the moment, and you hear it weekly:

> *"That does not work. I tried it."*

Ask one thing in return and the sentence falls apart: **when?** Usually it turns out to have been once, some time ago, without any of what belongs with it now. It was true. It is simply no longer true, and nobody comes round to say so.

That is the shape, and it is so ordinary that it is invisible:

1. someone tries something, once
2. it does not work — rightly
3. the outcome becomes a **judgment** instead of an observation
4. the judgment stays, because there is no reason to revisit it
5. the world shifts underneath it, and nobody notices

Step three is where it goes wrong, and it is exactly the step this chapter is about: an observation became an opinion the moment the date fell off it.

And it is not stupid. It is the cheapest way to deal with a world that changes slowly. This one simply does not.

Note what was happening in my own case, because two things were moving and they are often taken for one. **The tool got better. And my way of working got better.** Those two multiply each other, which is why the last step felt like a jump when nothing jumped.

That tools improve is not new. The unit is

A qualification belongs here straight away, because *just wait, it will get better* is not an insight — a carpenter, a printer and a doctor all knew that already.

The difference is the **unit**. It used to be a decade, sometimes a generation. At that scale "I tried it and it did not work" is a statement you can keep using for the rest of your career; revisiting it would have been a waste of time.

Now that unit is a few months. **And it is not linear**: each step is larger than the last, because the tool improves and your grip on it improves. Something quiet has changed with that:

> **The shelf life of a judgment has become shorter than the memory of it.**

You remember "this does not work" far longer than that sentence stays true. That is not a character flaw. It is a habit that was sensible for a hundred years and recently stopped being so.

What you do instead

Here the rule of this chapter comes back, now with a timestamp attached.

Record that at moment X it had status Y. Not *"this cannot be done"*, but *"this was tried, like so, and this came out."* That is an observation and it stays true — even when the answer changes tomorrow. A judgment without a date pretends to hold always, and that is precisely the error.

And then you do not stop. You put it on the list and you measure again, at a fixed interval, without weighing up whether it is worth it. Because the chance that you will revisit it on your own is small: you already know it does not work.

Let the difference show up systematically, not by accident. If you run the same trial the same way every time, you can put the outcomes side by side and see where the line turns. That is the same move as above: counting instead of guessing, only now across time. What you are looking for is the moment something that could not be done sud-

denly can — and you want to hear that from the measurements, not from a feeling.

And it is not only about the machine. Half of that improvement was mine. Anyone who retests the tool but not their own approach is measuring the smaller of the two.

Try this: write down five things you have tried and decided are not possible, with *when* you tried and *how*. Then have it done again, in exactly the same way, and ask where the outcome differs from what is written there. You will not retest what you have written off on your own — you already know it does not work. Make sure it is the same trial and not a better one: change your approach along the way and you are measuring two things at once.

Try it yourself · Have a complaint reduced to observations

What you do. `complaints/` holds four reports as they came in. Three of them contain a judgment — *late*, *not fixed*, *differs* — and you cannot build on a judgment. Have them reduced to what is factually there, and have that way of working written down at once so you never explain it again. Allow ten minutes.

TYPE THIS

```
complaints/ holds reports from clients. For each one, separate what is
an observation from what is a judgment, and for every observation find
the date and the file it comes from. Then write that way of working
down as a reusable procedure, with one line at the top saying when it
applies.
```

What you see. Four reports pulled apart, and then a file you did not write yourself. **The next time a complaint comes in, that procedure**

offers itself — that is the difference from an agreement you remember on a good day.

WHAT CAME BACK

```
2026-04-03-lowry
  report    : Work started two days later than
              agreed.
  fact      : ENJB-2026-006 on 2026-03-28
  judgment : "later than"
procedure created: complaint-to-observations
```

Where it goes wrong. Keep the judgment rather than throwing it away. *"Late"* is not a fact but it is information: it says what the client expected. Throw it out and the complaint has become incomprehensible. **Keep unconfirmed items too, as unconfirmed.** An observation you cannot find in a file is not untrue — it has only not been checked, and that is something else. **And in that one line, describe when the procedure applies, not what it is.** *"About complaints"* is never found; *"when a complaint or a report comes in"* is.

THEN LIKE THIS

> Add to the procedure what should happen with an unconfirmed observation, and show me the description line you put at the top.

AND THEN THIS

```
complaint-to-observations
  "When a complaint or a report comes in --
   reduces it to observations with a date and a
   source, before anyone passes judgment."
  step 4: what cannot be traced stays, marked
          unconfirmed, and is not dropped
```

And it holds the other way round too

This is not a one-way rule, and that part gets forgotten.

What you get back is often an opinion in the shape of a fact. *"This is the best approach." "This does not work." "Looks good."* Those are judgements, and they are worth exactly as little as yours — with the added discomfort that they sound confident.

The question you ask back is the one you would ask in that room: **how can you tell?** Not out of distrust, but because the answer is usable and the judgment is not.

Try this: have your problem description taken apart before you submit it — which sentences are fact and which are interpretation, and what is needed to replace those interpretations. You no longer notice which of your own sentences are judgements; they feel like descriptions. Do it *before* the work: afterwards you defend your description instead of improving it.

That was one direction. The other gets skipped more often and matters just as much: what comes back can be stripped of judgements too.

And the other way round: when you get a judgment back — *"this is the best approach"* — ask how it can be told, and explicitly allow the answer *"there is no such observation"*. Without that it will invent one, and then you are worse off than with the judgment.

> **What this made cheaper:** *every fact I wrote down instead of an impression could be used by someone else — and later by something other than someone.*

5. Attainable before valuable

There is one move that almost always turns out to be the wrong one, and it is the move with the highest return.

Not because the arithmetic is wrong. The arithmetic is usually fine. The problem sits a layer below: nearly every move with a high return per unit rests on a condition nobody writes down, and if that condition is not there, the return is not a disappointment but **zero**.

The question you ask first

Look at the condition, not at the amount.

The move with the highest value per unit is often not attainable, because something underneath is missing. And the attainable move is then not simply the smaller move — it is the move that *creates* that condition.

That is what this chapter describes, and it needs a name because it often runs against instinct: **attainable before valuable**. Not choosing the move that yields most, but the move you can actually make — and of those, the one that fills in a missing condition.

That sounds like reasoning you could stick onto anything afterwards. So here it is, worked out, on something anyone can check.

Four times nothing is still nothing

Take the publishing of this book.

Anyone publishing without a publisher can go two ways, and the platforms lay it out neatly. A paperback at € 14.99 sold through a bookshop leaves the author **€ 0.87**. The same book sold directly leaves **€ 3.47**. Four times as much.

Four times as much is a large difference, and it is the wrong quantity.

Because € 3.47 is only more than € 0.87 if somebody comes to buy. Selling directly comes with a condition that does not appear in that sum: **an audience.** People who know the book exists and know where to go. A newsletter, followers, a place where you are already heard.

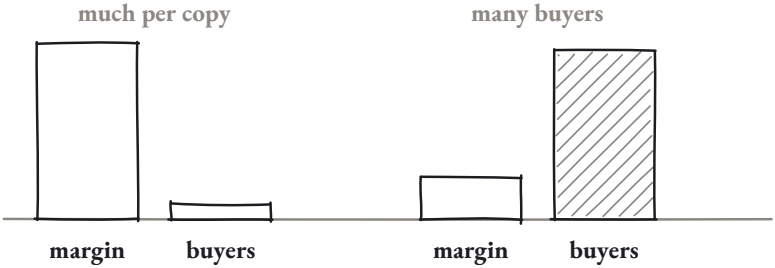
For a first book by someone without a platform, that number is zero. And the higher margin then applies to a visitor count of zero.

> **Four times nothing is still nothing.**

Put the other side next to it and the sum turns round. Five thousand times two dollars is ten thousand dollars; ten times ten dollars is a hundred. There is a factor of a hundred between them, and that difference is not in the margin but in the **count**.

So what an intermediary delivers is not margin but reach. One action puts the book in the large online shops and with booksellers in six countries. In the Netherlands roughly seventy per cent of online book sales go through a single shop; being in it is the condition, and that condition costs one upload and € 12.75 for an ISBN.

And there is a second reason, which may weigh more than the first. Every channel you connect yourself is an account, a tax form, a file format and a place something can break while you are busy with something else. That is not a one-off price but a subscription on attention — and attention is exactly what is not spare when you work alone.



Four times the margin on zero readers is still zero.

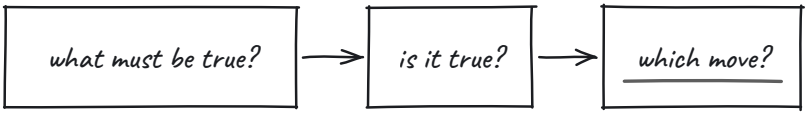
The mechanism underneath

There was a move with a higher return per unit. That move was not attainable, because a condition underneath it was unmet: an audience that did not yet exist. And the attainable move was not simply the smaller one, but **the move that creates that condition**.

That is the whole mechanism, and you can recognize it by three questions, asked in this order:

1. What has to be true before this move pays anything at all?
2. Is that true today?
3. If not: which move that I *can* make today makes it true?

The first question is the one that gets skipped. Almost nobody miscalculates the return on a plan; almost everybody forgets to write down which condition the sum rests on. A move that assumes a missing condition does not pay less than hoped — it pays nothing. That is a different kind of error from a disappointment, and it cannot be repaired by working harder.



the first question is the one that gets skipped

The three questions, in the order they have to be asked.

Try it yourself · Have it work out what you can actually enter for

What you do. `tenders/` holds four invitations and `certificates/company-certificates.md` holds what the company has. The largest invitation is not automatically the best one: it is only worth something if you are allowed to enter. Have them laid side by side. Allow five minutes.

TYPE THIS

> For each tender, put side by side: the size, what is required, and whether we have that according to certificates/company-certificates.md. Say per tender whether we can enter, and if not, exactly what is missing.

What you see. Four lines. Two can, two cannot — and the two that cannot are worth nearly a million between them. **They do not fall away on price or on capacity but on one missing certificate.** That is not a set-

back but a condition: it takes months and it is obtainable, only not before these two close.

WHAT CAME BACK

```
ENTE-2026-01  40,000/yr, 3 years   can enter
ENTE-2026-02  around 320,000       can enter
ENTE-2026-03  85,000 per year      MISSING: ISO 45001
ENTE-2026-04  around 900,000       MISSING: ISO 45001
```

Where it goes wrong. Have it match on the whole name, not on the first word. That went wrong here: for *ISO 9001* it looked at "ISO", saw that *ISO 45001* was missing, and reported that we did not have ISO 9001 — which we do. Two of the four lines were wrong because of it, and it looked entirely credible. **Ask for exactly what is missing**, not just yes or no: without the name you cannot tell whether it costs a month or a year. **And check one line by hand.** This is a list you make a decision on, and the cheapest check is a manual one.

THEN LIKE THIS

> Compare the certificate names exactly rather than on the first word, and show me per tender which names you compared with which.

AND THEN THIS

```
ENTE-2026-01  needs ISN, ISO 9001 -> both present
ENTE-2026-02  needs ISN, guarantee scheme -> present
ENTE-2026-03  needs ISN, ISO 45001 -> ISO 45001 no
ENTE-2026-04  needs ISN, ISO 9001, ISO 45001
            -> only ISO 45001 missing
```

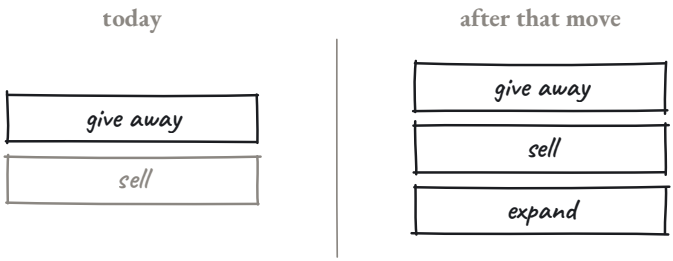
Why this is not "take the easy way"

This is where it goes wrong if you adopt it too fast, so it belongs here.

Reachable is not the same as easy. The reachable move only counts if it fills in the missing condition. Giving something away is reachable, but so is going out for coffee. The difference is whether something is true afterwards that was not true before.

The test is one sentence. If you can say in one sentence which condition this move creates, *and* how you would see that it is there, then it is the move. If you cannot, you have chosen the easy thing and invented a story to go with it.

And notice what this says about order. Reachability is not a property of a move but of a moment. Direct selling with four times the margin is not written off — it is waiting. If the book is in the shops, readers come; if there are readers, there is an audience; and on the day that audience exists, the same move that pays nothing today is suddenly the best one available. **That is what "making the next move cheaper" means in practice: not that you pay less, but that moves appear which did not exist before.**



A move that fills a condition opens up moves that were not there before.

And it holds where no money is involved at all. In the first chapter, what was on disk at the end of the evening was tooling and no product.

That tooling paid nothing that evening; it filled in a condition — that checking becomes free — and everything that came afterwards depended on it.

Try this: write down your next three moves, and instead of what they yield, note what has to be true *before* they yield anything. Cross out whatever depends on a condition you do not have today; what remains is your move. If it takes longer than ten minutes, the moves are written too large.

This is not *take the easy option*. The attainable move only counts if it fills the missing condition. If you cannot say in one sentence which condition it creates and how you would see that it exists, you have simply chosen the easy thing.

> **What this made cheaper:** *every further reader and every further user arrived without anyone having to be convinced.*

6. Multipliers

There are two kinds of improvement and they are almost always confused.

The first saves time. You do something more cleverly and from now on it takes twenty minutes less. Do it fifty times and you have won fifty times twenty minutes. That is real and it is not nothing.

The second does something else. It does not make one thing faster but **everything that comes after**. It does not add to what you had; it multiplies it.

A saving is a discount. A multiplier is an exchange rate.

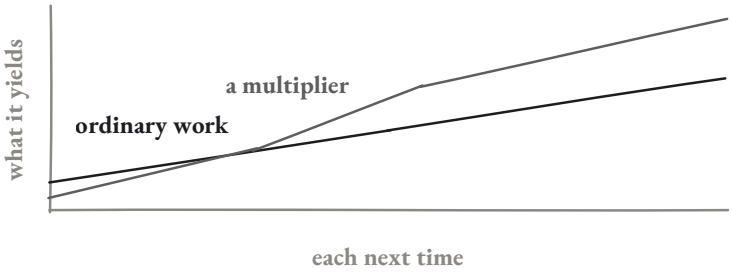
Why the difference matters

Because on the day itself you cannot tell them apart, and because the second almost always looks like the less attractive of the two.

A saving pays off immediately. You can see it, you can explain it, and at the end of the day something has changed. A multiplier pays nothing that day. It is hard to justify, because you have nothing to show but a promise about tomorrow.

And so almost everybody, almost always, takes the saving. Fifty times twenty minutes is seventeen hours, and that is a fine day's work. Except

that at the end of those fifty times you are standing in exactly the same place.



Ordinary work pays once. A multiplier lowers the floor for everything that follows.

The first one I ever saw as one

Long before I knew the word, I built one, and it had nothing to do with computers.

At the call center from chapter 3 I could do two things with the time between two calls. I could make one more call — that is the saving, and it adds up straight away. Or I could write down what had happened in the previous call and adjust the script.

The second produced **nothing** that day. Fewer calls, less revenue, and nothing to show. But a sentence that worked better worked better the next day too, and the day after, and for everyone using the script. One afternoon of thinking traveled along in every call I made from then on.

That is the whole difference, and it is older than the technology. The extra call is a discount. The adjusted sentence is an exchange rate.

The instrument was a scrap of paper.

How to recognize one

The question is not "what does this give me" but "what gets cheaper after this".

A multiplier has three properties, and all three are necessary:

1. **It works on everything that follows, not on one case.** Not this contract faster, but every contract from now on.
2. **It asks for no more attention once it is there.** Something you have to switch on each time may save time but multiplies nothing — because your attention is the scarce quantity and it does not grow.
3. **On the day itself it is a loss.** That sounds like a joke and it is the most reliable test there is. If it already pays off today, it is usually a saving.

The nine that made the difference

When I looked back at where the pace came from, nine moves were left. Not one of them was clever. All nine were dull, and all nine cost half a day that produced nothing.

the move	what it multiplied
a test suite	**checking becomes free**
a ticket system	**work outlives the session**
zero dependencies	**no afternoon lost**
self-building tooling	**every improvement improves the tool**
eleven milliseconds	**ten attempts are allowed**
an error log	**the same mistake costs minutes**
one shared model	**writing once is having it three times**
usable error messages	**the loop runs unattended**
the tooling free	**adoption is not a sales call**

Nine multipliers, and what each one made cheaper.

Read that list from top to bottom and you can see something happen. It falls into three groups, and those groups are a sequence.

The first three make one person faster. A test suite, a place where work stays put, and no dependence on somebody else's version. This is the kind of move everybody understands the reason for, and still puts off, because nothing comes of it that day.

The middle three make a machine usable. Tooling that builds itself, a startup time at which ten attempts cost nothing, and a log that makes the same mistake cost minutes the second time. Note what is happening here: these three are not about you. They are about whether anything other than you can get to work.

The last three let something keep running while nobody is watching. One model instead of three, error messages a machine can act on, and giving the tool away rather than selling it.

That is what stacking means, and it is why the order is not free: **each group needs the one before it.** Making a machine usable is pointless while checking is still expensive, and letting something run unattended is pointless while the machine is not usable. So not turning faster, but **making possible a kind of work that did not exist before** — and that only works in this order.

The two ways it goes wrong

Once you see what a multiplier is, you want to build nothing else. That is wrong too, in two ways.

The first: a multiplier that keeps asking for your attention is not one. This is property two from above and it is the one most often broken. A script you have to start yourself every morning, a list someone has to update, a check you have to remember — they all save time and they multiply nothing, because attention is the scarce quantity and it does not grow with you. **Do not count the minutes it saves but the times it needs you.** If that number is greater than zero, you have built a tool and not a multiplier.

The second: building only multipliers is the same as building none. Something has to get finished too. Half a year of tooling with no work underneath it is not an exchange rate but a hobby, and you only notice when there is nothing to multiply. The ratio that works for me is simple: **you build a multiplier only once you have done the work it makes cheaper by hand at least three times.** Before that you do not know what you are building.

That is the same first step as in chapter 3: do it yourself, get your hands dirty. Anyone who automates something they have never done themselves is automating their assumption.

Try it yourself · Bundle your separate checks into one procedure

What you do. You have now done the same thing three times: pick a rule, have a check written, run it. That is three rounds of work for three rules, and there are twelve. Have one procedure made that walks all of them — and have a second one put beside it that checks whether that round was complete. Allow fifteen minutes.

TYPE THIS

> rules.md lists twelve rules. Write a reusable procedure that walks all twelve against the folders they are about, reporting only the violations with their number. Also make a second, separate procedure that checks whether the round was complete, and have the first refer to the second. Then run the first and show me the result.

What you see. Two procedures, and a result you would never have assembled by hand: **fifteen violations across ten of the twelve rules.** The competence rule stands out with four, and that is no coincidence — behind it sits one certificate that expired on a single day and hit four jobs at once. **Seeing two rules with no violations is information too**: those were run and are clean, which is not the same as not run.

WHAT CAME BACK

```
rule  1  1   ENJB-2026-005 missing
rule  2  1   client does not exist
rule  4  1   extra work 3950 unconfirmed
rule  5  4   competence
rule  7  1   subcontractor without G-account
rule  8  2   diary
rule  9  2   inspection expired
--- 15 violations across 10 rules ---
```

Where it goes wrong. Have it name the clean rules too. Otherwise you cannot tell whether rule 10 held or was skipped, and that difference is exactly where a check loses its value. **The second procedure belongs beside it, not inside it.** A round that declares itself complete always declares itself complete. **And watch the rules that depend on a date.** Rules 5 and 9 must be measured against the day the work happened, not against today — measure against today and you get a different answer as soon as you run the check next year.

THEN LIKE THIS

> Also show the rules that gave zero violations, so I can see that they were run.

AND THEN THIS

```
rule  6  0   safety plans current
rule 10  0   invoices within 14 days
rule 11  0   building notice before start
rule 12  1   market signal without a date
--- 12 of 12 rules run ---
```

And this is not about software

The list above comes from a technical project because that is what the project was. Each of the nine has a counterpart in work that has nothing to do with computers, and usually that counterpart is older than mine.

A builder with a fixed checklist for handover has the first. A practice that records what was agreed with a patient has the second. A baker who makes his own dough has the third. A photographer who fixes his processing has the fourth. A restaurant kitchen where a dish leaves the pass within a minute has the fifth — and that is the reason a second restaurant is even possible.

What they all share: on the day they were made, they produced nothing.

Try this: in last week's diary, cross out everything you are going to do again in exactly that form. What remains is the work that repeats — and that is the only work a multiplier can get a grip on. Pick one and ask what would have to exist to make it half as expensive next time. Build *that*, not the task. Start with the task that comes back most often, even if it is small: multiplying works on the count, not on the size.

Once you have found one, the question that decides whether it really is one arrives. Not what it produces, but whether it leaves you alone afterwards.

Then test whether it really is one. For something you have just built, have every moment listed where it still needs you — starting, updating, cleaning up, checking, deciding — and count them. A solution that saves time but keeps asking for attention does not scale: your attention is the one quantity that does not grow. Zero is often unattainable;

five down to one is already a different order, as long as you write down *which* one it is.

What this made cheaper: every following day. That is meant literally, and it is the only thing that separates a multiplier from a saving.

7. And if you did it a thousand times?

Two ways of doing the same thing, and on the day itself you cannot tell them apart.

In the first you look at the result and say: this is good. In the second you write down what it was that told you it was good, and next time somebody else looks at that — or something else does.

Once, the first way takes less time. It always takes less time. And at a thousand times the first way is not more expensive but **impossible**, because there is only one of you.

That is the whole chapter: **ask of every choice what it looks like at a thousand times, even when you are only doing one.** Not because you are going to do a thousand. Because the answer makes you choose something else.

The question has three parts, and the third is the hard one

Does it still work? Some things fall over on volume. Two things that want the same place are fine until they arrive together. A name that looked unique collides. An order that was right by itself is not right any more. This category is the best known and usually the cheapest to repair.

Is it still affordable? Multiply the cost of one by a thousand and see whether you flinch. A dollar each is a thousand dollars. Five minutes each is three working days. This is arithmetic and it takes ten seconds, and almost nobody does it.

And does it still need me? This is the question that decides everything. If each of those thousand takes one minute of your attention, you do not have a system — you have seventeen hours of work and an illusion. The number of people does not grow with it, and that is not something you fix later.

The first two are technical. The third is a design question, and its answer decides whether you are building something that can grow or something that gets big and then stops.



× 1000, and then again

The question has three parts, and the third is nearly always skipped.

I once answered those three questions in that order

At the call center I did something hundreds of times a week. That is not a thousand, but it is far enough past one to feel the difference — and it forced me to answer the three questions without knowing they were questions.

Does it still work? Yes. A call is a call, and the hundredth did not break because it was the hundredth.

Is it still affordable? Also yes, and that was exactly the problem: it cost the same every time. A hundred calls is a hundred times the price of one. There was no discount in it.

Does it still need me? And there it stopped. Everything that worked was in my head. I knew which sentence did it, I knew when to stay quiet, and I could not hand it to anyone else without sitting down next to them. I was needed a hundred times.

That is what writing it down changed. Not the calls — those were no better on the day itself. What changed is that the answer to the third question went from *yes* to *less*: there was a script, there was a step-by-step procedure, and the dull part ran on its own. Only then could something exist that was bigger than my week.

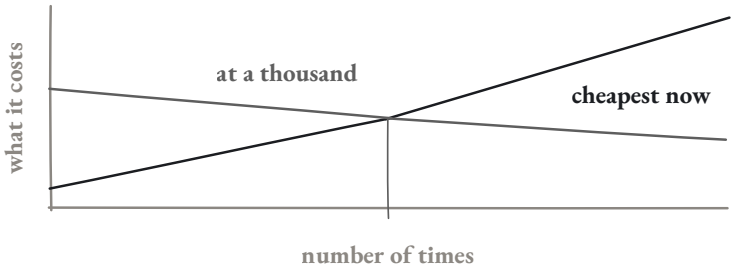
Note the order, because it is not accidental. I answered the first two questions by looking. I answered the third by building something. It is always like that: the first two are arithmetic, the third is work.

And this is about ordinary work

A restaurant with one chef can have a dish that depends on his judgment. That dish can be excellent. It simply cannot ever be in a second restaurant, or on his day off. A dish with a written measure and a checkpoint is perhaps slightly less — and it also exists when he is not there.

A teacher marking two hundred essays by feel does the same work two hundred times. A teacher who first writes down what a good essay looks like does that work once and then something else — and gets something back that the first never gives: he can explain why.

A consultant who keeps everything in his head can never take two jobs at once and can hand nothing over. That is not a character flaw; it is a design choice he once made without noticing he was making it.



What is the cheapest choice at one is the most expensive at a thousand.

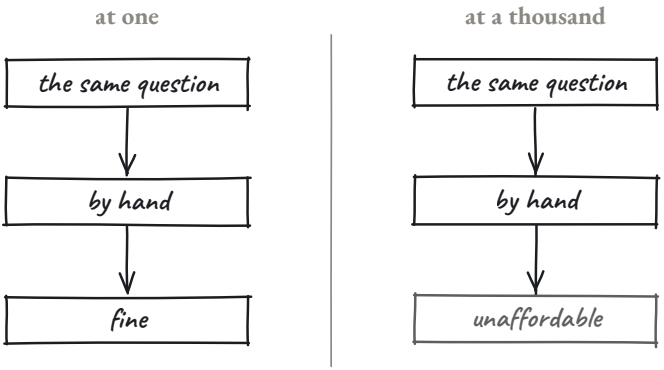
You arrive at different decisions

That is the point this chapter turns on, and it is bigger than it sounds. The question is not a check afterwards. **It changes what you choose.**

at one you choose	at a thousand you choose
looking at whether it is good	writing down what tells you it is
remembering what you did	recording it where it can be found
just doing it yourself	making sure it can happen without you
"we'll notice"	a limit that steps in by itself
the best solution	the solution nobody has to have explained

Look at the right-hand column. Every one of those is **more** work on the day. It is the same shape as the first chapter: the move that counts is the move that produces nothing today.

The difference is that here you have a reason to make it anyway. Not "it is tidier", not "that is how it is done", but: **otherwise it can never be more than one.**



The same decision, taken twice: once for now and once for a thousand times.

And then again

What survives a thousand does not automatically survive a thousand times a thousand. Each jump breaks something different.

At ten you find you need names. At a hundred you find you need to see what is going on without reading everything. At a thousand you find you need to intervene without stopping everything. At ten thousand you find the money can run out without anyone seeing it coming.

So ask the question again and again, at every step, not once at the beginning. It is not a gate you pass through; it is a habit.

Try it yourself · Have it report only what changed

What you do. Your round from the previous chapter reports fifteen violations. With fifteen jobs that is manageable. With fifteen hundred it is not, and then something unpleasant happens: you stop reading the list. The question of this chapter is what changes when you do it a thousand times — and the answer is that the shape of the result has to change, not the check. Allow ten minutes.

TYPE THIS

> Adjust the round so it remembers the previous findings, and from now on report three things separately: what is new, what has been resolved, and how many are still open unchanged. Run it twice: now, and after I have changed something.

What you see. The first round cannot compare anything and says so — fifteen findings, no comparison possible. **After that the result is suddenly two lines instead of fifteen**, and they are the two you have to do something about. **The fifteen unchanged do not disappear**, they get counted. That is the difference between ignoring something and acknowledging it.

WHAT CAME BACK

```
first round: 15 findings
  (no comparison possible)

after a change:
new since the previous round     : 1
   + 3: ENJB-2026-013
resolved since the previous round: 0
unchanged                        : 15
```

Where it goes wrong. Keep the count of unchanged ones. Remove it and nothing seems wrong as long as nothing new appears — and then a problem that sits there for three months has become invisible. **Make**

sure the first round says it cannot compare. Without that line an empty first result looks the same as a clean administration, which is exactly the mistake chapter 9 is about. **And keep the memory somewhere you will find it again.** A round that has lost its previous result quietly starts at zero and then reports everything as new.

THEN LIKE THIS

> Also show how long each unchanged finding has been open, in rounds.

AND THEN THIS

```
unchanged (15), with how long they have been open:
  1: ENJB-2026-005 missing        since round 1
  2: ENJB-2026-006                since round 1
  4: ENXW-2026-06                 since round 1
  9: rolsteiger-B               since round 1
  (the counter runs from the first round, not
   from when the problem started)
```

Why now in particular

The question is not new. What is new is that the answer matters.

It used to be that a thousand times meant hiring a thousand people, or building a factory. For almost everybody the question was therefore theoretical — you were never going to do it a thousand times, so why design for it.

That is gone. A thousand times is now a matter of turning it up.

And that is exactly where the difference sits between a tool and an organization. A tool makes you faster; you stand next to it and it waits for you. An organization does things while you are not there. The

jump between the two is not a better machine — it is a set of choices you have to make early and cannot retrofit.

Whoever arranged his work for one is stuck at one, however good the tools get. Whoever asked the question can simply turn it up.

Try this: take the next decision you are making anyway, however small, and ask three questions alongside it — does this still work at a thousand times, is it still affordable, and does it still need me. Write down only the third answer: the first two are arithmetic you can correct later, the third is a design choice you cannot change later without redoing everything. The question is about the *shape*, not the size — you do not have to build for a thousand, you only have to avoid choosing something that rules a thousand out for good.

What this made cheaper: everything that later had to scale up and did not have to be rebuilt, because the shape already allowed it.

8. Why are you still doing that yourself?

It is not a reproach. It is a question you ask yourself while you are in the middle of something, and the first time the answer is usually still a good one.

Because I do it faster than I can explain it. Often true. *Because it only takes two minutes.* Also true. *Because it needs judgment.* Sometimes true.

What makes this question special is not that it changes the answer. It is that **it stays the same question while the answer shifts under your feet.** What could not be done last month can be done now. And you do not notice, because you are busy.

Why the answer shifts without your seeing it

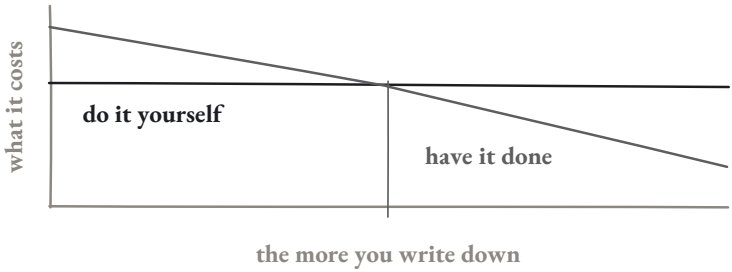
At some point you decided that something was your work. That decision was good — then. Afterwards it became a habit, and a habit does not account for itself.

Meanwhile three things change, and none of the three knocks.

What a machine can do grows. Not in leaps you notice, but a little each month. The task that last year had to be redone three times now works first time — and nobody sends you a message about it.

What it costs to explain falls. That is the quieter of the two, and the more important one in this book. Every time you write something down — a rule, an example, a mistake in a log — the next explanation gets shorter. At some point *explaining* becomes cheaper than *doing*, and that moment passes without a sound.

And what you have left over in attention falls too. As more runs, your minute is worth more. The same two minutes that cost nothing last year cost something now — not because you are busier, but because they yield more elsewhere.



What a machine can do rises, what explaining costs falls — and somewhere they cross.

Three movements, all slow, all in the same direction. The decision you once made becomes a little less true every month. And because it is never wrong all at once, it stays.

Which is why it is a question and not a rule

A rule would be: *hand over everything a machine can do.* That is nonsense, because then you hand over things you do faster yourself, and things that need your judgment, and things you cannot check yet.

The question is better, for three reasons:

It does not ask for a system. You do not have to take an inventory of all your tasks. You ask it while working, about the thing you are working on.

It accepts "because it cannot be done yet". That is an honest answer, and a useful one: it names exactly what would have to change. *Not yet* is a date without a number.

And it keeps itself current. The same question, a month from now, about the same work, can give a different answer without your having had to track anything.

The four answers, and what to do with each

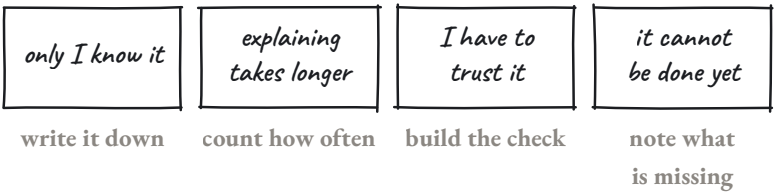
In practice four kinds of answer come up, and each asks for something different.

"Because I am the only one who knows." This is the most expensive answer and it sounds the most like a compliment. What you know sits in your head and travels nowhere — not to a colleague, not to an agent, not to yourself in three months. **The action here is not to hand it over but to write it down.** Only then can the question be answered at all.

"Because explaining takes longer than doing." Almost always true for once, and almost never true for ten times. Work it out rather than feeling it: how often did I do this in the past month? At one, you are right. At five, you lost three times ago.

"Because I have to be able to trust it." A good answer, and it points at the real work: not the doing but the checking. Can you tell afterwards whether it went well? Can you tell cheaply? As long as you cannot, you should indeed do it yourself — and then building that check is the actual task.

"Because it cannot be done yet." The best answer, because it is concrete. Something is missing: an access, a key, a piece of tooling, an example. Write down *what* is missing. That is a ticket, not a feeling.



Four answers, and each points at a different action.

Notice what is in none of the four: "because I enjoy it". That is a valid answer, and the only one that needs no action. But then call it that, instead of putting one of the other four in its place.

The one time I did not ask it

If I could redo one thing from those first days, it is this, and it is exactly the question of this chapter that I failed to ask myself.

I sat working out **the bootstrapping phase** myself. How the tooling could come to build itself, in what order, with which intermediate steps — the kind of puzzle you can spend hours on and feel proud of at the end of the evening.

That was work I need not have done. **I should have formulated it and handed it over.** Not "solve this", but: this is where we need to get to, this is what already exists, these are the conditions — now propose an order.

Why did I not? Because it felt like the interesting part. And that is exactly what is treacherous about this question: **you ask it automatically about dull work, and almost never about work you enjoy.** Dull work you are glad to hand over. Puzzle work you hold on to, and you call it thinking.

Sometimes it is. But the test is not whether it is enjoyable; the test is whether your head adds something that would not otherwise be there. For that bootstrapping order the answer was no — I had no knowledge that was decisive there, I had a preference.

So if you ask this chapter's question once a day, ask it about the work you like most. That is where the answer stands still longest, and where it is most expensive.

Try it yourself · Ask what this arithmetic is still doing with you

> **What you do.** Somebody puts the budgeted hours next to the actual ones every month. It takes an hour, it never goes wrong, and for that reason nobody has looked at it again. Have it recalculated and see whether the answer is still the same. Allow five minutes.

TYPE THIS

> For each job, put the budgeted margin next to the actual one, from cost-control/ and post-calculations/. Handle the running jobs separately and show there how much of the budgeted cost has already been spent.

What you see. Six lines. Two finished jobs that stayed neatly within budget, one that lost sixteen percentage points of margin — **and a running job that has already spent 106 per cent of its budgeted cost while it is still going.** That last one appeared in no monthly overview, because a monthly overview looks at finished jobs.

WHAT CAME BACK

```
ENJB-2026-001 budget  7.9%  actual  8.5%   +0.6
ENJB-2026-004 budget 15.2%  actual  8.6%   -6.6
ENJB-2026-010 budget 19.9%  actual  3.5%  -16.4
ENJB-2026-002 budget  8.2%  running (106% of the
                           budgeted cost spent)
ENJB-2026-012 budget 14.9%  running (54%)
ENJB-2026-007 budget  8.7%  running (60%)
```

Where it goes wrong. Ask explicitly for the running jobs separately. Without that request it computes the margin across everything, and then a job about to derail disappears between three that went well. **Watch what is NOT in the source.** These figures come from `cost-control/`, and the outstanding supplier invoices are not in there — so the real cost is higher than what you see. That is not an arithmetic error but a source error, and chapter 14 is about that. **And do not ask for a judgment.** Whether sixteen percentage points is bad depends on things that are not in this folder.

THEN LIKE THIS

> Add the outstanding supplier invoices per job to the cost so far, and show which picture changes.

AND THEN THIS

```
with outstanding supplier invoices included:
  ENJB-2026-007   60% ->  94% of budget
  ENJB-2026-012   54% ->  67% of budget
  ENJB-2026-002  106% -> 106% (none outstanding)
and: ENJB-2026-011 has a supplier invoice but is
     not in cost-control/ at all
```

And this is not about software

The question is older than the subject of this book, and that is exactly its strength.

- **An owner who does the books themselves** did so once because there was nobody else. Two years later there is somebody, and they still do it — not out of distrust, but because nobody asked the question.
- **A doctor who schedules their own appointments** defends it with exceptions that are real. The question is not whether those exceptions exist, but whether they should drive the whole diary.
- **A mechanic who checks every fault personally** is right about the difficult cases. The easy eighty per cent just comes along with them.
- **A writer who does their own layout** says nobody else does it the way they do. That is true, and the question is whether the difference is worth anything to the reader.

In all four it is not about laziness and not about delegation as a virtue. It is that a decision that was once right has quietly stayed put.

And the question is polite. That is not incidental. "Why are you still doing that yourself?" leaves room for a good answer, which is why you dare to ask it — of someone else, and of yourself.

Try this: once a week, ten minutes — have the five things listed that you repeated most often, each with the condition still missing before you no longer have to do it yourself. Ask for that missing condition and not for a solution: a solution you cannot yet check is a problem you have moved.

That round produces four kinds of answer. For three of them you now know the action; the most expensive one asks for something else, and it is not delegation.

Tip: when you hear yourself say *"I am the only one who knows that"*, have yourself questioned until it is on paper as a method somebody else can follow. Let the questions be asked rather than dictating: you automatically skip whatever is obvious to you, and that is exactly the step the other person is missing.

And there is one sentence that removes almost every blockage, provided you put it in the right place.

Put this in your standing instructions: *when in doubt, choose a plausible option yourself and carry on; say afterwards what you chose and why, and only ask when the choice is irreversible, costs money, or goes outside.* Most standstill comes not from hard questions but from small ones — what should this be called, where do I put it — and every time such a question reaches you, the work waits until you have time.

That is not a blank cheque, and it only works because the rest of this book sits underneath it. Checks run along, limits live in the execution and not in a request, and there is a trail of what was decided. **A plausible choice is safe as long as a wrong choice is visible and reversible** — and where it is not, the last clause of that instruction draws the line. Do insist that the choice is named afterwards; otherwise you get progress without a trail.

> **What this made cheaper:** *every time the question got a different answer, work moved that never came back onto my pile.*

9. Checking itself

Something was wrong with the icons and nobody could say what.

They were there, they had the right shape, they sat in the right place. They just looked as if a sheet of frosted glass lay over them — as if someone had greyed them out. That was exactly the description, and it was right.

Three rounds followed. In the first, the colors were raised to the values measured from the original. In the second they went higher still, to full saturation. In the third the lines were thickened and the sampling was made finer. After each round: a fresh screenshot, and exactly the same picture.

Three rounds with no difference at all is not bad luck. It is a message, and the message is: you are turning the wrong knob. If the color goes up and nothing changes, the brake is not in the color.

So we stopped looking and measured. Not the picture — the data the picture is made from. A small program read the icon data and printed, per icon, how much coverage the darkest spot reached. On a scale from 0 to 255 the answer was **31**. Twelve percent. And not a single pixel reached 255, in any of the icons.

That number cannot be a color problem and cannot be a shape problem. It is an answer.

The cause was in the loop that draws the icons. Per pixel it counts how many of sixty-four sub-samples fall inside the shape, and divides by sixty-four. There was a break in it: on the first hit it jumped out of the inner loop. So the counter could never get past eight, while the division by sixty-four went on regardless. Everything was drawn at twelve percent of its coverage.

Removing that one line fixed it. After that: maximum coverage 255, and in the simplest icon 130 of the 400 pixels fully covered.

"Looks good" is not a judgment. Thirty-one out of two hundred and fifty-five is one. Three rounds of looking did not find it; one measurement found it in a second.

The bug that looks like a matter of taste

Why it took so long is the more interesting half. The bug did not look like a bug. It looked like a choice.

That is a class of its own, and it is the class that survives longest. A program that crashes reports itself. A program that draws something at twelve percent looks like somebody with a taste for muted colors. Nothing squeaks. There is only somebody who glances at it now and then and thinks: it could be a bit stronger.

There is one sentence that sums it up, and it is about something else entirely:

> **There is nothing that stops it or reports it — it fails silently.**

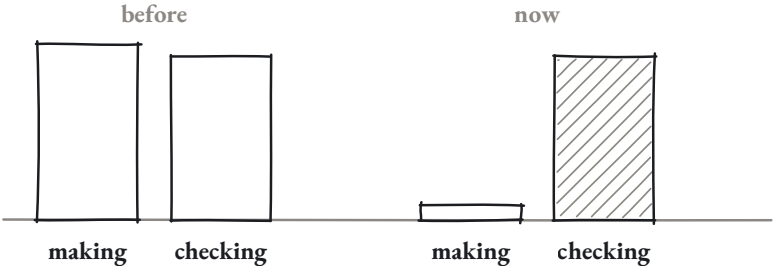
Failing silently is the normal condition of generated work. Not because it is bad, but because there is nothing holding it back. And that brings us to the problem this whole book is about, now in its sharpest form.

Generating became free, checking did not

As long as you check everything yourself, you are the bottleneck. That is not an annoyance but arithmetic: having a hundred things made per day and judging a hundred things per day is a hundred judgements per day, and that number has a ceiling which has nothing to do with technology.

There is only one way out, and it is not "generate better". It is: **the checking itself has to become cheap, and it has to move over to the machine.**

That sounds impossible as long as checking means "looking at it". It becomes possible the moment checking means "computing a number". That is the turn this chapter describes, and it consists of four habits that look unrelated.



Generating became nearly free. Checking did not, and that is where the brake now is.

One: measure instead of asking

Back to the icons, but now to the pattern around them.

While one part of the interface was being built, I noted eight remarks. Here they are verbatim, **translated from the Dutch**:

> *"a bit small" — "thin" — "are you scaling the icons slightly too small?" — "icons barely visible, so dark" — "not bright enough" — "do you have some kind of disabled filter over it" — "the highlight has shifted too far up" — "check the size and spacing of the chevrons"*

Afterwards it turned out: **all eight were measurable from a screenshot.** Icons eleven pixels tall where twenty-one had been measured. A color of 888888 where d4d7d6 belonged. Ink sitting eighteen above and five below the text where twelve and eleven belonged. Not one of the eight was a judgment of taste. All eight were numbers, and they were sitting in a picture that already existed.

That is the split everything hangs on: what can be measured is your work; what is genuinely taste, you put to someone. Not because asking is bad, but because every question needs a human and the number of humans does not grow with the work. Eight questions that need not have been questions are eight times the system stood still.

And note the less obvious side: this is what keeps the remaining questions useful. Somebody asked eight times whether something looks nice is no longer looking properly by the ninth.

Two: the loop until it is right

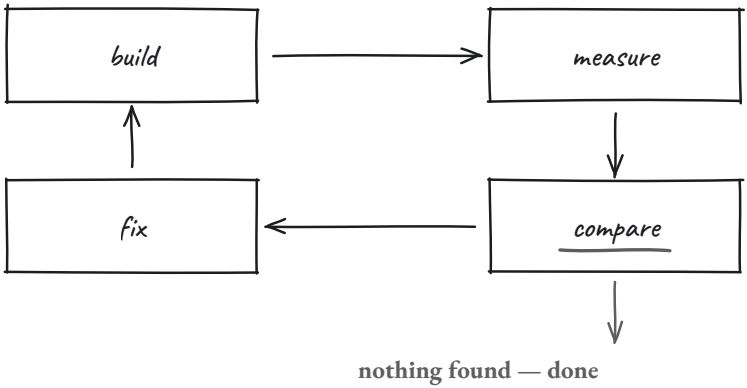
The second habit is a shape, not an insight: build, start, record, compare against the reference, name one cause, repair, again.

Five to ten rounds in a row, without asking in between. Two rules keep that loop healthy:

One cause per round. If a round finds nothing, the loop is done and only then is anything shown. If a round finds five things, the odds are that it is one bug with five symptoms — so look for the common factor first, because five repairs on one cause produce four new bugs.

Two rounds of adjusting with no effect means stop adjusting and start measuring. That is the rule that could have solved the icons three rounds earlier, and it exists now because it cost three rounds.

This only works if the rounds are cheap. At eleven milliseconds per build a machine can try ten times; at thirty seconds it tries once and guesses the rest. Speed here is not comfort but the condition the whole loop stands on — the next chapter is about that.



The loop: build, measure, compare, fix — and only stop when there is nothing left to find.

Three: the check that distrusts itself

The editor crashed when a file was clicked.

The first reflex is always the same: *I just broke it.* The second reflex is the valuable one: rebuild the previous version and try it there. Same crash. It had been in there for days.

The cause was two safeguards that reversed each other. The first makes sure the cursor never goes below zero. The second makes sure it never goes past the last line. On a file of zero lines the second one sets it to minus one — the safeguard was the source of the bug.

A finding is only a finding once you have held it against the previous version. Otherwise it is a claim, and three times out of four the wrong one. The difference costs one rebuild, and it is the difference between "I found a bug" and "I broke something" — two conclusions that lead to opposite work.

There is a second lesson in it and it is less comfortable: **nobody had ever clicked.** Nine days of building an editor, and the most obvious action a user performs had never been carried out. Whoever tests decides what gets found. A machine that checks itself on exactly what it thought of checking finds exactly that.

Four: the test rig is a suspect too

On that same day it twice looked as if there were a bug in the code, and twice the bug was in the setup.

The first time, two windows were open with the same title. The tool that looks a window up returns them in arbitrary order. The keystroke went to one window and the screenshot came from the other. Conclusion: the panel does not open. That conclusion was wrong, and it looked entirely solid — there *was* a screenshot, and the panel was not on it.

The second time, a key combination did nothing. Five minutes of searching the table where that combination is supposed to be registered. Until it turned out that the tool sends *synthetic* keystrokes, and that the environment the program ran in ignores that category. With a real keystroke it worked immediately.

The third time was the sharpest, and it came much later. A check that inspects the captions in a drawing was green. I looked at the drawing and saw two captions running into each other — the first word of the second label disappearing behind the last word of the first.

The first suspicion was the measurement: if the width of a piece of text comes out ten per cent too low from the font file, then *every* caption in

the whole book sits too close to its neighbour, and no check sees it. So I measured it — the same texts rendered along exactly the route of the book, and the ink in the image counted. **The measurement was right, to within one or two per cent.**

What was actually wrong is a subtler kind of fault. The check asked: *do these two overlap?* The answer was no, and it was correct. The two stood about half a millimetre apart: they did not touch and they read as one word.

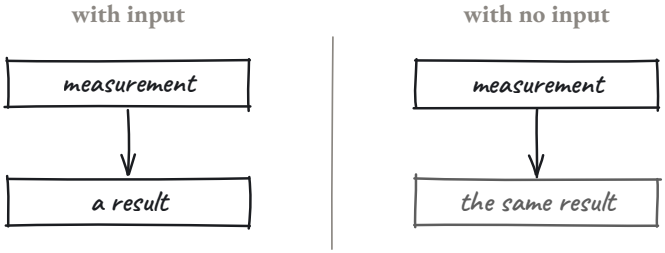
> **There was a boundary, but no minimum.** *And a boundary without a minimum approves everything that only just clears it.*

That is not the measurement setup being broken — that was fine. It is the **question** that was too narrow. A green check is never worth more than the question it asks, and that question appears nowhere in the answer.

Whoever measures checks his ruler first. That is old advice, and it matters more as you leave more to machines — because a machine does not doubt its own setup. It reports what the setup says, with the same confidence it has when the setup is right. A person who sees two windows standing there feels something. The machine does not even count them.

And with a check that is green while you can see something: **believe your eyes and go looking for the question.** It is not the answer that is wrong but what was asked. The repair is almost always a minimum added — not *"they must not touch"* but *"there has to be this much between them"*. That new rule immediately found a second case I had not looked at, which is the proof that it was one.

In practice this means: the first step of a diagnosis is not looking for the cause but counting what is running. How many processes, how many windows, which of the two holds the port. That is three lines of work, and it is the difference between an afternoon of searching and a minute.



Before believing the measurement: measure what you get when nothing happens.

A test you have never seen go red

One more habit, and it is the cheapest of them all. It costs two minutes and it decides whether everything above is worth anything.

> **A green test you have never seen go red is not proof.**

A test that is green says two things at once and you cannot tell them apart: *this is right*, or *I am measuring nothing*. A test that accidentally looks in the wrong place, an assertion that is always true, a file that is never even loaded — all three look exactly like a test doing its job. In fact they are more *reassuring*, because they never break.

The proof is putting the fault back. Deliberately break what the test is supposed to catch, run it, and see it go red. Then restore it. That is the whole ritual, and it takes less time than writing the test did.

An example. There is a check on the coverage of a set of icons, because earlier a fault in the rasteriser had made them all pale — they were there, they had the right shape, and they were so faint it looked like a style choice. The new check was green. Then I put the fault back, and **eighty-nine of the ninety-six checks fell over**. Only at that moment did I know it would catch it.

Skip this and you have built an alarm without ever testing the bell. And that is exactly the kind of certainty that is worse than none, because you start leaning on it.

And the other side: a test that catches too much gets switched off

There is a second way a check becomes worthless, and it feels more innocent than it is.

A test can look **too narrowly**: then it lets through what it was meant to catch, and you see green while it is wrong. That is the fault above. But it can also look **too widely**: then it reports things that have nothing to do with it, and you see red while it is fine.

The first costs you a fault. The second costs you the test, because a check that raises false alarms is ignored or removed within a week. **A gate that rejects half the work is not a strict gate but an absent one.**

Both forms occur, sometimes within the same hour. Two checks looked too narrowly — and a third looked too widely: it had to block a forbidden word and immediately rejected two perfectly ordinary sentences, because the permitted exception happened to be broken across a line ending. Five more reports like that and nobody runs it any more.

> **So look at both sides: what does it let through, and what does it report wrongly.**

The second is also easier to measure than the first, and it is almost never done: run it once over everything that is already right, and count how much it rejects. Every number above zero is work before it may be switched on.

Try it yourself · Distrust your own check

What you do. Your check from the previous chapter reports four violations on rule 5. The question of this chapter is not whether there are four, but whether the check itself holds. **A test rig is a suspect too**, and you test it by making it fail on purpose. Allow ten minutes.

TYPE THIS

> In staff/t-underwood.md, change the certificate "OSHA 30" into "OSHA-30", with a hyphen instead of a space. Then run rule 5 again and tell me what changes and why.

What you see. From four violations to eight. **And the four new ones look entirely credible**: *t-underwood has no OSHA 30, four times, neatly with the job number. There is nothing about them that shows they are nonsense. The check compares the required name literally with the name in the staff file, and one hyphen is enough to break that comparison without anything going wrong.

WHAT CAME BACK

```
rule 5 now reports 8 violations:
  ENJB-2026-002: t-underwood has no OSHA 30
  ENJB-2026-004: expired
  ENJB-2026-007: l-brennan has no hoisting ticket
  ENJB-2026-008: expired
  ENJB-2026-009: t-underwood has no OSHA 30
  ... and three more of the same kind
```

Where it goes wrong. Put the name back afterwards. Forget that and your administration now carries a fault you introduced yourself, and in three months you will not find it again. **Do not think this is an artificial fault.** A certificate typed by hand, a name the issuer changed, a space from a pasted table: it happens more often than an expired certificate. **And look at which SIDE the fault falls on.** This check reports too much, and you notice that. A check that reports too little you do not notice — and that is the dangerous side.

THEN LIKE THIS

Put the space back, and have the check also report which required certificate names it could not match to a name in staff/, separately from the real violations.

AND THEN THIS

```
rule 5  violations: 4
rule 5  unmatched: 0
  (every required name was found in staff/,
   so the four are real)
```

The gates: the mechanism against drifting unnoticed

The four habits above find mistakes. A fifth is needed, and it prevents something else: that something which was already right changes without anyone noticing.

A machine working on its own has no feeling for what was expensive to reach. It sees a part that works and is tested as material, like everything else. That is how something that took three days quietly disappears, inside a change that was about something else entirely.

A **gate** is the cheapest countermeasure there is: a mandatory question to yourself before you touch something that works and is tested. Not more rules, not a prohibition — one moment at which a reason has to be given.

Two properties make it usable instead of irritating:

It is not absolute. Overruling is allowed. Working things sometimes have to give way, and a rule that forbids it will be worked around within a week rather than followed.

The reason gets written down. That is the whole mechanism. The value is not the permission but the trail: two weeks later there is a sentence saying why this working thing was removed, and that sentence is the difference between a decision and an accident.

An agreement nobody checks does not hold. A gate is the smallest form of checking that still holds, and it costs one question.



red: back, and the loop is short

A gate lets nothing through until the check is green — and so keeps mistakes small.

What the machine can read, it can check

There is one thing that makes all of the above possible and that is easy to overlook: **the work has to leave traces in a form a machine can read.**

A screenshot is for a human. A number is for both. And the whole turn is in there, applied to recording what an agent does:

> **Store it in a form you can read back, and you no longer need a screenshot to see whether it went well.** *As long as the result of an action can only be seen, only a human can judge it. The moment it can also be read — a number, a list, a structured file — the machine can check it itself, and then the human is no longer needed for it. This is not a matter of better models. It is a matter of which form you store your own work in, and that is a choice you make today.*

Try this: put in your instruction not only *what* has to be made, but also what it is checked with and that it should keep going until it is right — and name the command literally. Verification is the bottleneck, not generation: as long as you are the one checking, nothing scales. Of eight tasks given cold, one passed the check first time, so the first attempt is not the delivery. It only works if checking is *fast*: at eleven milliseconds it tries ten times, at thirty seconds it tries once and guesses.

Having it check only works when there is something to count. Otherwise you get a judgment back instead of a check, and you are no further along.

Try this: before you start judging, work out which property of the result you can *count*, and have that sorting done first. A photographer sorts two thousand shots on sharpness and closed eyes before looking at one; a translator first has it checked whether all the numbers, names and units match. Every judgment you make yourself costs a person, and the number of people does not grow. Of the eight remarks above, eight were measurable from a screenshot — not one was taste. Take your last ten rejections and ask whether there was a number under each; the answer surprises almost everybody.

What survives that sorting is worth keeping — because the same mistakes come back, and the second time they need cost nothing.

Tip: keep a log of every mistake a model makes — what it wrote, why that was wrong, what the right form is — and make that the list you send along from now on. Sort on how *often*, not on how interesting. A model fills in from neighbouring conventions when it meets something it does not know, and that is exactly what such a list catches. One line per mistake, in the imperative, with the right form beside it: *"watch out with types"* teaches nothing, *"declare every variable before the start"* does.

And watch what goes into such a primer. Everything in it gets imitated, including the part you did not mean by it.

Watch out: never put example syntax in your instructions in a way a tool could read as real syntax. An instruction is text in a file, and a file is read by more than one thing — what is an example to the model can be an order to another tool. It really happened: the instruction text contained, as an example, the word that pulls in another file; the tool reading that text saw it and carried it out, and skipped the *real* reference fur-

ther down as a duplicate. Wherever text is meant for two readers, one can read what was meant for the other.

> **What this made cheaper:** *every check that turned into a number never had to be done by a human again.*

10. Speed is not comfort

The agent had already tried four times by the time I had read the line it was stuck on.

It wrote a piece of code, had it built, got an error back, read it, changed one thing and had it built again. Four rounds, and together they fitted inside the time I needed just to see *what* was going wrong. On the fifth round it was done. I did nothing.

That is not a story about a clever agent. It is a story about a number: the build step took **eleven milliseconds**. Which made four attempts cost less than a blink, and made "just try again" a reasonable move.

Why speed is what makes the loop possible

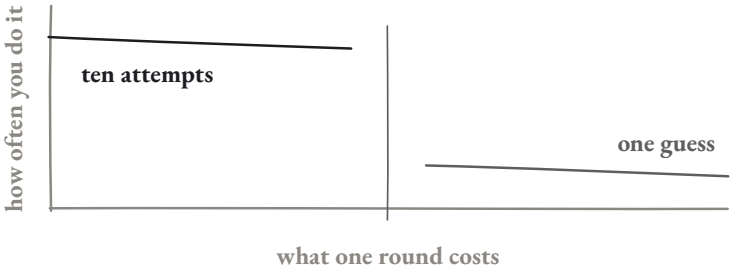
Speed is almost always sold as comfort. Less waiting, nicer work, a better day. It is that too, and that is not why it is here.

Speed is what makes the loop possible. The method from the previous chapter — build, measure, compare, fix, and only stop when there is nothing left to find — is a loop that has to be able to go round ten or twenty times. What one round costs decides whether that loop exists at all.

Work it out and it is not complicated. Ten rounds of eleven milliseconds is a tenth of a second: you do not even notice. Ten rounds of

thirty seconds is five minutes, and for one small problem that is a price someone starts weighing. Fifty rounds is half a second at one number and half a working day at the other.

With that, a performance figure has become something other than a performance figure. It is a **design condition**: not *faster is nicer*, but *below this line the method works and above it it does not*.



Below the threshold a machine tries ten times. Above it, it tries once and guesses the rest.

The line is a step, not a slope

Here is the part that surprises. You would expect it to get slowly worse: the slower the step, the fewer the rounds, neatly proportional. That is not how it goes.

What happens is that the **behavior flips**. Below a certain line, trying again is cheaper than thinking about what it might be, so people try. Above that line, thinking becomes cheaper than trying — and "thinking", for a machine as much as for a person, is called **guessing**. Not one round slightly slower, but a different way of working.

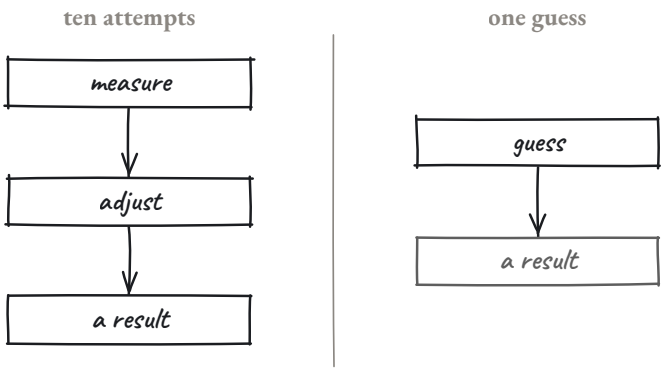
Anyone who has ever waited on a slow build recognizes this in themselves. You do not change one thing and watch what happens. You

change three things at once, because you do not want to wait three times. And then it works, and you do not know which of the three it was.

The worst part is that it says nothing

A loop that has become too expensive does not fall over. It gives no error, it does not stop, nothing announces that something was given up on. It simply produces an answer that looks exactly like an answer that *was* checked.

That is the quietest form of quality loss there is, and you cannot see it in the outcome. You can only see it in the number beside it: how many rounds actually ran? Zero rounds and fifteen rounds both produce a result. Only the second one was checked.



Ten attempts and one guess both produce an answer. You cannot tell them apart from the answer.

And this is not about software

The quantity that matters here is not *speed*. It is **the cost of one attempt**, and that exists in every trade.

- **An approval that takes a week** means nobody submits half an idea to see what comes of it. Only worked-out proposals get submitted, and those are more expensive each and wrong more often each. Drop that same approval to an hour and what changes is not the pace but the *kind* of proposals that arrive.
- **A trial that costs a day** gets run once, with the best guess. If it costs ten minutes, five variants get tried and you also know which variable did it.
- **A cook who tastes along the way** corrects twenty times per dish. A cook who only hears how it was at the table corrects once a week.
- **A quote that takes two days to prepare** does not get issued when in doubt. That is not laziness; it is the same arithmetic.

In all four the question is not "can it be faster" but: **is the price of one round below the line where people still dare to run them?** Below it, you get more rounds and therefore a better result without anyone trying harder.

And put the thousand-times question next to it and it sharpens. What is an inconvenience at one round is, at a thousand, the difference between "this runs through the night" and "someone has to sit with this".

Try it yourself · Measure how long your round takes

What you do. Your round from chapter 6 walks the whole administration. Before you start running it more often, you need to know what it costs — because that decides whether you run it in passing or once a month, and that in turn decides when you find something. Allow three minutes.

TYPE THIS

> Run the round three times in a row and measure how long it takes each time. Then say what that means for how often I can run it.

What you see. Three times two hundredths of a second. **That is not a performance figure but a threshold**: under a second you run something in passing, without thinking about it. Over ten seconds you do it when you remember, and over a minute only when you have to. The check has not got better — it has got *cheaper*, and so you run it ten times as often.

WHAT CAME BACK

```
round  0.02 s
round  0.01 s
round  0.02 s
15 jobs, 12 rules
(and that includes start-up: the round itself
 fell below the measuring floor)
```

Where it goes wrong. Measure with the real quantity, not with a sample. Fifteen jobs say nothing about fifteen hundred; ask as well how the time grows. **Measure three times, not once.** The first run carries start-up time that later runs do not, and then you are measuring something other than you think. **And watch what you are not measuring.** This is the time of the check, not of reading the result — and with fifteen hundred jobs the latter is the real problem.

THEN LIKE THIS

> Multiply the administration a few times by copying the folder, and show how the time grows with the number of jobs.

AND THEN THIS

```
     15 jobs   below the measuring floor
    240 jobs   0.002 s
    960 jobs   0.007 s
   3840 jobs   0.028 s
  15360 jobs   0.124 s
the time grows in step: a thousand times the
work costs a thousand times the time, and that
is still an eighth of a second
```

What it costs to stay fast

Speed has a price. Every choice that bought it shut something else off — and you pay that price on a day you were not expecting to.

So speed is not a property you add later. It is the sum of the things you did not do, which is also why it cannot be retrofitted: by then everything you *did* do is already in there.

The practical side: if there is one property you want to protect from convenience early in a project, make it this one. Almost everything else can still be repaired later. The cost of one round cannot.

Try this: put a clock on the step you repeat most — ten times in a row, and ask for the median *and* the longest, not the average. The longest is the number that decides whether anybody does it again; one round of thirty seconds among ninety-nine fast ones sets the behavior. Then ask what the three largest items in that time are and what could go, with the price in functionality — otherwise you only get the pretty half. The

question is never *is this fast* but *does ten attempts fit in the time somebody is willing to spend*.

Once you have that number, the next question is how you hold on to it. Speed does not go all at once.

And fix the line before you need it: write down what the most expensive step may cost before your way of working stops working, and have a test run along that fails as soon as it goes over. Speed does not disappear at once but in steps that are each defensible, and there is no moment where somebody says: here we stop. Choose a line you comfortably clear today — one you only just clear will be red tomorrow for a reason that has nothing to do with speed, and then it gets switched off.

> **What this made cheaper:** *trying became cheaper than guessing, and with that every check from the previous chapter became something a machine can simply do as it goes.*

11. Not on disk, never happened

When I looked back for this book, I still had everything. Every change with a timestamp. Every task with the reason attached. Well over a thousand pieces of work whose existence I could read back.

And from the first four days I had nothing.

Not because anything was lost. The changes are there to the minute; I can tell you exactly what happened on those days. What is not there is why. Not one consideration, not one doubt, not one choice with a reason attached.

Those days were spent building and not discussing, so nobody wrote anything down. At the time that was entirely sensible. **They were the four days in which everything came into being.**

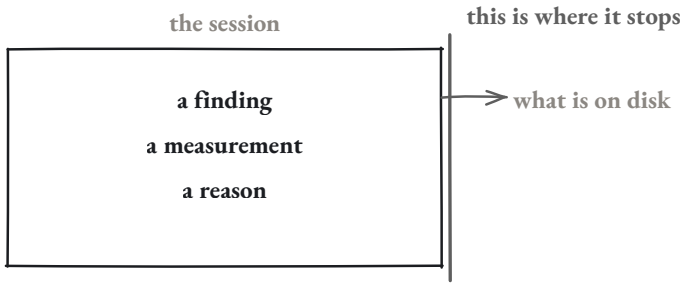
What disappeared there

The strange part is that I knew all of it at the time. Every choice had a reason, and the reason was so obvious that writing it down felt like waste. You do not write down why you turn left when you know the way.

Two weeks later the way is gone. Not the route — that is still there — but the landscape around it: what you considered, what you rejected,

why this had to come first and that later. That was in my head and in my head only, and it is not there now.

This is the most underrated kind of loss there is, because you do not notice it at the moment it happens. Nothing breaks. Nothing falls over. The only thing that happens is that a month later you look at your own work and think: why did I do it like that.



What you hold in your head is gone once you no longer have it to hand; what is saved stays.

Why this is a multiplier and not administration

Recording feels like overhead. It is work that makes nothing, and it has to happen at exactly the moment you are busy with something that does make something.

But look at what it acts on:

- **It makes work transferable.** Something that exists only in your head can only be done by you — and then there is always exactly one of it, however many hands are available.

- **It makes repetition cheap.** Answering the same question a second time costs almost nothing if the answer is written down somewhere, and as much as the first time if it is not.
- **It makes your mistakes usable.** A mistake you remember, you might not make again. A mistake that is written down, nobody else makes again either.
- **And it is the only way anything can continue without you.** That is not an aside but the whole precondition for everything still to come in this book.

So it is exactly the shape from the chapter on multipliers: a loss on the day, multiplying after it.

Afterwards does not work, and that is not a matter of discipline

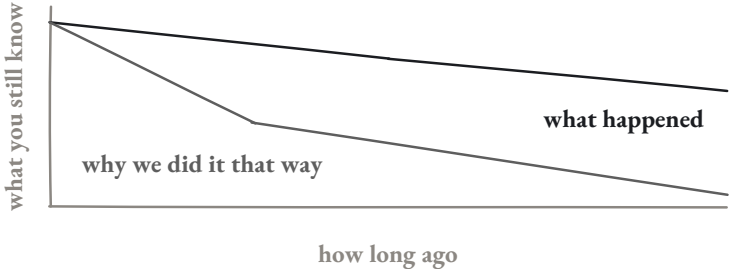
The obvious answer is to do it at the end. End of the day, end of the week, end of the project — by then you know how it turned out.

That does not work, for two reasons and both are insurmountable.

The first: by the end, half of the why is already gone. What you write down then is what you did, and that is usually recorded somewhere already. What you no longer write down is what you almost did and why not — and that is precisely the part you can find nowhere else later.

The second: often there is no end. A day runs out without a conclusion. A project changes direction. A conversation is interrupted. And every time there is no end, there is also no moment at which it happens after all.

So you write it down while it is happening. Not neatly, not completely, not in sentences somebody else would admire. Rough is fine — rough and present is worth infinitely more than polished and absent.



The why decays faster than the what.

The reason is worth more than the decision

There is a reason I feel this chapter more sharply than most, and it comes from the support work.

There, plenty was recorded: what had been agreed, what had been done, how it went. What was almost never there was **why** something had been decided that way. And that is exactly the part you need, because without the reason you can only follow a decision or break it — you cannot **revise** it.

Take something simple: it has been agreed that someone comes on fixed days. If only that is recorded, then six months later it is a rule nobody knows the origin of. If the reason is there too — because it got too crowded otherwise, or because something clashed — then you can see whether that reason still holds. In the first case you have a habit. In the second, a decision that can move.

And what is not written down gets reinvented on the person themselves. That is the real cost, and it falls on somebody else. The next supporter starts again: asking again, trying again, discovering again what had already been discovered three times. For the organization that is lost time. For the person it happens to, it is the third time having the same conversation about themselves.

With an agent it is the same shape without that burden: what is not written down gets worked out again. Only there you pay in tokens and minutes, so you barely feel it — which is precisely why it is skipped even more easily.

The rule I took from it holds in both trades: do not record what you decided, record **why**, and do it at the moment you know. An hour later you no longer know your own reasoning, and then you write down what happened instead of why it happened.

And this is not software

A consultant with everything in his head can never take two jobs at once, cannot induct anybody, and cannot fall ill. He calls that experience. It is also a design choice, and he probably never made it consciously.

A workshop where the mechanic knows which cars have this fault loses that knowledge on the day he leaves. A workshop that records it still knows afterwards — and sees something on top of that which nobody could see while it sat in one head: that it happens more often on that one model year.

A school discovers every year afresh where its pupils stumble. Every year afresh, because it was written nowhere. **In all three cases the problem is not that nobody knows. It is that the knowing is not anywhere.**

Try it yourself · Record the decision, not just the outcome

What you do. On 15 April the setting out of the extension was four centimetres off the drawing. That was resolved, and the decision is in the minutes. **But the reason is not there**, and in a year that is the only thing anybody needs. Have that searched for, and have a procedure made of it at once. Allow ten minutes.

TYPE THIS

> Find the places in execution/ where a decision was made, and check per decision whether the reason is there or only the outcome. Then write a reusable procedure for recording a decision, with one line at the top saying when it applies.

What you see. Four sets of minutes with decisions, and four daily reports in which something is decided without being called a decision — *reported to work preparation*, *passed on*, *reschedule*. **In none of the eight is the reason there.** For the setting out the outcome is recorded (*4 cm deviation agreed after adjusting the drawing*) but not the trade-off: why adjusting was cheaper than moving.

WHAT CAME BACK

```
decisions with a reason : 0 of 8
decisions with only the outcome : 8 of 8

hidden decisions in daily reports:
  "reported to work preparation"
  "passed on"
  "reschedule"
  "extra work announced"
```

Where it goes wrong. Look in places that are not called a decision list. Half of what is decided here sits in a daily report in half a sentence. Ask only for minutes and you miss that half and think the administration holds. **Ask for the REASON as a separate field.** Leave that out and you get a procedure that records the outcome neatly, and in a year you have exactly the same problem. **And write it during the work, not after.** A reason you write down at the end of the week is a reconstruction, and those always sound wiser than they were.

THEN LIKE THIS

> Make the reason a required field in the procedure, and show me what happens when someone leaves it empty.

AND THEN THIS

```
record-decision
  "When a decision is made in a meeting or on
   site -- records what, when, by whom, and WHY."
  reason empty -> the decision is not written
                  away, with the message
                  "a decision without a reason is
                   an outcome, not a decision"
```

Three things genuinely worth recording

Most of it can be derived later from what is there. What you cannot derive from anything is this:

1. **What you decided, and why.** Especially the why, because the decision itself is usually visible in the result.
2. **What went wrong, and what caused it.** Not that it went wrong — you will remember that — but what it came down to, because that is exactly what you would have wanted to know next time.
3. **Where you were when you stopped.** One line. What you were doing and what comes next.

That third one looks the least important and is the most used. Somebody continuing tomorrow — you, or somebody else, or something else — then does not have to reconstruct where the story had got to.

Try this: put one line at the top of the document your work lives in — what you are doing and what comes next. Update it the moment you move on, not at the end, and remove it when it is done: a line pointing at finished work is worse than no line. Keep it a *pointer* and not a report — *"working on the quote for X, waiting on prices from Y"* is enough. Anyone who puts more in it will not keep it up.

What this made cheaper: every time somebody had to continue where somebody else had stopped — including me, two weeks later, not recognizing my own work.

12. Fratsen

In the work of getting people into jobs we had a word for it, and I have never lost it. It is Dutch, and I am going to keep it, because every English word I tried gave away half the meaning.

Fratsen. Small, unexpected, unpredictable or awkward little things with large consequences. *Frats* misses the damage. *Glitch* and *snag* suggest something that stops, and these do not stop anything — that is the whole point. The nearest English is a phrase rather than a word: **small silent misses.** So: fratsen. Not disasters. Not unwillingness. Usually something too small in itself to raise: arriving just too late, not asking a question, half-filling a form, keeping an appointment but without the things that go with it.

Once is nothing. Which is exactly why nothing is done about it.

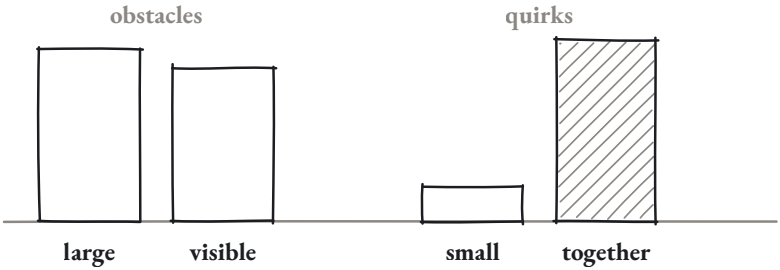
Why it is the small thing that is the bottleneck

The big things do not go wrong. That sounds odd and it is simply true.

Large obstacles are visible, and visible things get attention. They get a meeting, a plan, a budget. **Fratsen sit below the threshold at which anyone intervenes** — too small to be worth a conversation, and by the time they are collectively large enough, the pattern has become a habit.

And they do something specific. They do not make it go wrong; they make it **just not work.** The placement that nearly succeeded. The job that ended after six weeks anyway. Not for one reason you can point at — for five small things that were each nothing on their own.

That was the pattern I saw, again and again. Not a lack of capacity, not a lack of willing. Fratsen.



Large obstacles get attention. Fratsen sit below it and accumulate.

What separates a frats from a mistake

This distinction is the core of this chapter, and without it you do the wrong thing.

A **mistake** goes wrong and stops. There is a message, something does not work, someone notices. A mistake is annoying and it is usable: it teaches you something, and the next chapter is about that.

A **frats** does not go wrong. It produces a result that looks fine and just is not. Nobody notices at the time, because there is nothing to notice. You only notice it in the outcome a few steps further on, and by then the cause is no longer findable.

	a mistake	**a frats**
what you see	something does not work	something appears to work
when	immediately	a few steps later, or never
once	irritating	invisible
a thousand times	a thousand messages	it structurally just falls short
what you do with it	fix it and write it down	make it visible first

Mistakes are your best teacher. Fratsen teach you nothing, because they do not tell you they are there.

And this is exactly what a hallucination is

When I started working with AI I recognized it immediately, and that helped me more than any explanation.

A model that crashes is a mistake. A model that invents a library that does not exist, or cites a source that is not there, or gives a number from nowhere — that is a frats. The answer looks like an answer. The shape is right. The tone is right. Nothing beeps.

And the consequence is the same as in the labor market: **it blocks scale.**

Not because it happens so often. Precisely because it happens rarely. If one in twenty answers contains a frats, and you cannot see which, then you have to check all twenty. There goes the gain. Running a hundred tasks at once is pointless if a hundred have to be checked by someone who cannot keep up.

A frats does not make the work expensive. It makes trust expensive — and trust is exactly what you need to let something run without you.

Why it is so rarely discussed

There is one more reason fratsen survive, and it has nothing to do with alertness. In the support work it was the most important of them all.

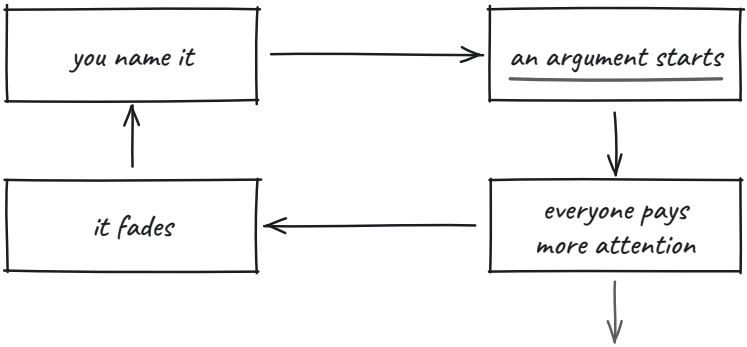
A frats is not an objective mistake, and that makes it socially delicate. A mistake you can point at: something was supposed to happen here, it did not, and that is checkable. A frats you can only name as an impression — *it keeps not quite working*, *something is off*. And an impression invites a counter-impression.

So one of three things happens the moment you raise it, and none of the three helps:

- **an argument starts**, because there is no fact on the table but a judgment, and against a judgment another judgment is a valid answer
- **you are not understood**, because the other person hears something about themselves that they cannot see anywhere — and that is not unwillingness, there really is nothing to see
- **the symptoms are suppressed for a while**: everyone pays closer attention for two weeks, it looks better, and then it comes back. Nothing was solved, because nothing changed

The real loss is not the awkward conversation. It is that afterwards you have **nothing measurable that you can solve systematically.** You

shared a feeling and received a promise. That is not something to work on, and in a month you are back where you were.



Without a fact the conversation goes round. Counting is the way out.

So the move here is the same one as in chapter 4, and here it is not a matter of style but the only way to make the subject discussable at all: **turn it into a fact first.** Not *"you are often just a bit late"*, but *"of the twenty times, seven differed from the other thirteen, and this was the difference."* That is no longer an accusation. It is an observation, and an observation cannot be argued with — it can only be looked at.

So counting is not just how you find a frats. **Counting is what makes it possible to talk about one.**

With a machine that social price is zero, and that is a bigger advantage than it sounds. You can tell an agent what is not quite right, twenty times in a row, without anyone feeling accused and without a relationship being at stake. That lets you do in an afternoon what takes months

in work between people — and that is how I suddenly saw the pattern I knew from that other world again, in miniature and sped up.

How you get a grip on something below the radar

First the sharpest statement of the difference, because it determines the whole approach:

> **Mistakes you can measure. Fratsen sit below the radar.**

A mistake announces itself. You can count it, sort it, and draw a graph of it going down. A frats announces nothing, which is why no instrument that waits for reports will help.

What does not work is paying closer attention. That is the answer everyone gives first, and it fails for the same reason in both worlds: fratsen sit below the threshold of attention, and attention is exactly what there is least of in the run-up. Worse — your judgment is the very filter that left them out. Whoever pays closer attention pays closer attention to the same things.

There are five moves that do give a grip, and they are in this order because each needs the one before it.

One: repeat and compare. This is the only detection principle that works. A frats has no signature by which you recognize it in a single case — its signature is **difference between runs that should have been identical.** One outcome says nothing. Twenty outcomes side by side point straight at it, and you do not even have to be clever about it.

Two: record what it looks like when nothing is wrong. Without a reference every difference counts the same, and you drown in noise.

With one, deviation is the signal. The reference does not have to be beautiful; it only has to exist and not drift.

Three: count, and do not judge. Note what was different, not whether it mattered. You estimate severity later; at the moment of noting, your judgment is the problem, because it is the filter that kept them invisible.

Four: narrow the room to vary. Most fratsen do not disappear because someone tries harder, but because the situation leaves less room to deviate. The same order, the same shape, the same words. Whatever then differs, differs for a real reason.

Five: lower the threshold. And here is the real answer. Fratsen exist because there is a line below which nobody intervenes — that is not sloppiness but an economic choice, and when the work was done by hand it was the right one. **You only get a grip when that line can come down, and it can only come down when checking is cheap.**

With that, this chapter collapses into the rest of the book. Speed, automatic checks, a loop that inspects itself: that is not tidiness. It is what lowers the threshold to below the level at which fratsen live.



paying closer attention is not among them

Not closer attention, but count, make predictable, and test cheaply.

The frats itself, or something before it

Then the question that decides everything, and it is the one I now ask myself about every frats:

> **Do you solve the frats, or do you solve something that comes before it, so that fewer fratsen arise?**

Because a frats rarely stands on its own. **There is nearly always a cause in front of it.** Someone who arrives just too late may have no alarm clock, or a bus that does not connect, or an evening that did not end well. A model that invents a library was never given a list of what does exist. The visible little thing is the far end of something that started earlier.

And that means you have to go **back**, which feels wrong. Everything in a project pushes forward: the next step, the next task, the next week. Returning to something already passed looks like delay. **It is the opposite of making progress, and it is still the only thing that works** — because forward solves this case, and back solves the kind.



back is the only direction that removes a whole kind

Not the case, but what lies in front of it. That is where a whole kind disappears.

There are three heights at which you can act, and the difference between them is the whole story:

	what you tackle	what you are left with
the frats	this one case	next time it costs the same again
the cause before it	why cases of this kind arise	a whole kind disappears
what allows that cause	the situation that left room for it	they stop arising

The first height is not worthless. It buys you time, and sometimes time is exactly what you need in order to get round to looking. But if you stay there you are mopping: every case costs the same, nothing accumulates, and in a month you are still doing it.

The third height is not always within reach. Some causes sit with someone else, in a system that is not yours, in a habit older than you are. Then the second is good enough, and it is more honest to write down that you stopped there than to pretend you reached the root.

This is the same move as *find the key, not the doors*, later in this book: not the visible case but the condition underneath it. And it is also why *narrow the room to vary* ranks so high above. Less room to deviate is nothing other than removing a cause before anything comes of it.

A frats from this book itself

While this was being made, not a single bold word was bold for weeks.

The typeface this book is set in exists in two cuts; one of them has no bold. The typesetting program chose that one and quietly fell back to regular for bold. No message, no warning, no difference you see unless you know to look. Every emphasis in every chapter was gone, and it simply looked like a book in which emphasis did not work very well.

That is a frats in its purest form. Small, silent, not an error, and present on every page. It was not found by looking harder but because someone put one page next to another and wondered why it looked so gray.

Try it yourself · Group by kind instead of by project

What you do. `handover-points/` holds six points from four handovers. Per handover there are one or two, and then it looks like bad luck. Have them grouped by kind and see whether that still holds. Allow five minutes.

TYPE THIS

> Group the points in handover-points/ by kind instead of by job, and say which kind comes back most often. For each point, note whether it has been fixed or is still open.

What you see. Three kinds. **And then something shows up that was invisible per project: everything still open sits in the same kind.** What is ugly — a sealant line, a cheek not in line — gets fixed on the spot. What is *missing* has to be ordered, scheduled and returned for, and so it sits. None of these six is a mistake; together they hold up two handovers.

WHAT CAME BACK

```
missing        3
  kitchen vent grille        fixed
  anti-slip treads           open
  rainwater downpipe         open
finishing      2
  sealant line bedroom 2     fixed
  dormer cheek               fixed
size or norm   1  (handrail too low, fixed)
```

Where it goes wrong. Let it come up with the kinds itself rather than giving them. Hand it your grouping and it confirms your picture and finds nothing new. **Include the status.** Without *fixed* or *open* you would only have counted how often something occurs, and the pattern — everything open in one kind — would not have come out. **And expect a category "other".** That is not a failure; it is where a new kind will surface next time.

THEN LIKE THIS

> Now look at whether there is a difference in how long the points stay open per kind, and what explains it.

AND THEN THIS

```
finishing     on average 2 days to repair
size or norm  1 day
missing       1 fixed after 5 days, 2 still open
              after 30+ days
difference: finishing you do with what is in the
van, missing you have to order
```

And this holds far beyond this subject

- **A kitchen where one preparation step varies** produces dishes that are all slightly different. Not one dish is wrong. The place just never becomes predictable.
- **A team where handover is usually verbal** works fine until someone is ill. Then it turns out it has been slightly off for months, only nobody noticed because it kept being corrected.
- **A form with one ambiguous field** yields a few per cent wrong entries. At a hundred a year that is noise. At a hundred thousand it is a problem with its own department.

- **A machine that rarely jams but often doses just differently** produces nothing that gets rejected, and still the batch misses specification.

In all four there is no culprit and no incident. There is a pattern below the threshold of intervention, which only becomes visible when you count it.

Try this: run a task you use often twenty times with the same input, and ask for every difference between the outcomes — including the small ones, *especially* the small ones — with no judgment about severity. You are not looking for failures but for variation: what differs at twenty times differs a thousand times at a thousand. Ask explicitly for *no* judgment, or you get the big things back and you had already seen those.

Counting makes them visible, and visible is not yet gone. The first urge is to fix the case in front of you, and that is exactly one step too far forward.

And do not go straight at it. For the frats that occurred most often, first ask which three things come before it and make it possible, what would change if they were gone, and which of the three removes the most cases — still without a fix. Solving a frats costs the same again next time; removing its cause makes a whole kind disappear. If you cannot reach the third height, write that down: pretending you were at the root is how the same frats gets investigated again six months later.

Only once you know where you are acting does a check make sense. And it has to be cheap or it gets switched off.

Tip: then ask for a check that catches it and costs *less* than the action itself, with what it does *not* catch. An expensive check gets switched off as soon as things get busy, and knowing what a check misses is the difference between a check and a false sense of safety. Nine out of ten is enough to start — waiting for a check that catches everything is how you end up with none.

> **What this made cheaper:** *every frats that became visible no longer had to be absorbed by me — and only after that did it make sense to run more at once.*

13. Mistakes are your best teacher

Eight tasks, given cold. One of them came through.

It was a fair trial: eight small, ordinary tasks in an environment with its own conventions, given to a model with no help, no examples, and nobody looking over its shoulder. Seven of the eight did not work. **Twelve and a half per cent.**

The reflex that goes with that is: *the model does not know this.* The reflex is understandable, it is partly true, and it leads nowhere. Because if that is the answer, there is also only one remedy — wait until your way of working is described often enough to end up in some training run — and that is not a remedy but a wish.

So I did something else. I put the seven failures side by side and looked at whether they resembled each other.

Seven mistakes, three kinds

They did. They resembled each other so closely that the whole pile fitted into three boxes.

An invented component. The model used things that do not exist here, but do exist in the environments it knows well. Names that sound sensible, that are real somewhere else, and that are nothing here. This

was the largest box, and it is also the largest category in the error log I had been keeping for months.

A convention that is nowhere else like this. One small piece of spelling that differs in my from the two environments it most resembles. Not wrong exactly, not guessable, and for precisely that reason wrong every time.

A form that varies case by case. The third kind was the most annoying, and not because it is hard. Two functions sitting next to each other in the same file want their data handed over in different ways. That cannot be guessed and is barely memorable — which makes it not a fault of the model but a fault of mine.

the kind of mistake	how often
an invented component	the largest box
a deviating convention	two of the seven
a form that varies by case	and that one is mine

Three kinds of mistake, and how often each occurred. Sort by count, not by how interesting it is.

Once I had them laid out like that, a sentence came out that I had not expected:

Three rules would have prevented seven of the seven mistakes.

All seven times they were three conventions that were nowhere they could be read.

It is not a knowledge problem but a presence problem

Here is the part that transfers, and it holds far beyond this example.

All three conventions were on disk. Correctly written down, in documentation that was accurate and that nobody had to hunt for. **They just were not in view at the moment the code came into being.**

That is a different kind of shortfall from "it does not know". It genuinely did not know, but the reason it did not know is not that the information was nowhere — it is that nothing handed it over at the right moment. And that can be repaired without anything having to get cleverer.

The distinction is worth holding on to, because it decides where your money and your time go:

- **a knowledge problem** is solved with better material, more training, a bigger model — slow, expensive, and usually not up to you
- **a presence problem** is solved by sending the right piece of text along at the right moment — cheap, today, and entirely in your own hands

Almost everything that looks like the first turns out on inspection to be the second.

The list made of your own mistakes

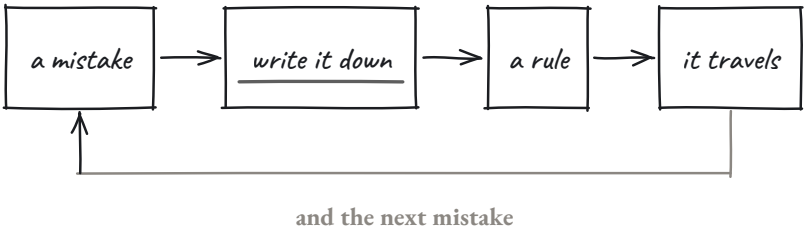
What follows from that is almost too simple to write down, which is exactly why almost nobody does it.

You keep a log of every mistake that gets made. What was written, why that was wrong, and what the right form is. That costs half a minute per mistake, at a moment when you are looking at the mistake anyway.

Then you sort that list — and here is the only step with a choice in it.

Sort by how often, not by how interesting. The temptation is to start with the mistake that surprised you most, because that is the most enjoyable to think about. But the mistake that surprised you happened once. The dull one you see three times a day is the one eating your time.

And then you send the top lines along, every time: the five or ten rules that things actually break on, plus a complete list of what really exists — because against invented components, a list is the entire answer.



From mistake to rule to never again: each round makes the next one shorter.

Work it out, fix it, guard it

There is a short way to hand this entire chapter to an agent, and it is three words.

> *"Work it out, fix it, guard it."*

Work it out — think of how it can be solved, and say which approach you are taking. **Fix it** — carry it out. **Guard it** — make sure it cannot come back: a test, a check, a rule that travels along.

The first two everyone does. The third is where it breaks down, and without that third the loop is not closed: you have then solved one mistake instead of a kind of mistake.

What those three words replace is a whole conversation. Without that agreement you describe what you expect every single time — and that is exactly the explanation that has to be repeated every round. With it there is one line, and you get the whole circuit back.

And that is what they are for. **Not that it does it once, but that it keeps doing it** — so that the next time the same thing comes past, something is already there to stop it and you no longer have to look.

The gate you build catches you

This is the most uncomfortable thing I can say about this chapter, and also the most useful.

Every check I have built ended up going off on my own work. Not on someone else's, not on an agent's — on mine, the person who thought the check up and knew exactly what it was watching for.

The pattern is always the same. I build a check because I have seen a kind of mistake happen. It sits there. I carry on, convinced I will not make *that* mistake again, because I know it. And a week later it goes off, on me.

A sample of what that produced, all while making this book:

- a check on lost text found that my own typesetter was discarding one hundred and eighty-seven lines
- a limit on the length of a sample of output held back my own text fourteen times
- a ban on product names caught a product name in a passage I had written myself
- a check on names that blur into one another found two names I had invented myself
- and another system, over which I had no say at all, found four things that had been crooked here for months

Not once did anything go off on somebody else. That is not because other people work better. It is because I am the one doing most of it here, and therefore also the one making most of the mistakes.

Why this matters: it changes what you build a check for. Not to catch somebody out, and not to keep an agent in line. **You build it against yourself a week from now** — against the version of you that is in a hurry, that already knows this part, and that therefore stops looking.

It is also the answer to the objection that it is too much trouble for mistakes you do not make. You do make them. You are simply not present at the moment it happens, and that is exactly what the previous section says.

The number you are looking at is the wrong one

One more thing those eight tasks produced, and it may be the most useful of all.

Twelve and a half per cent sounds like a bad number, and it is one — if what you are measuring is how often it is right first time. But why would you measure that?

In this case there was a check that is strict, that is fast, and that says at length what is wrong. An attempt costs almost nothing, and every failed attempt yields a sentence you can act on. **So the question is not how often it succeeds first time, but how often it succeeds within three attempts with the error message attached.** That is an entirely different number, and it is the one that actually matters, because that is how it works in practice too.

This is the same thought as earlier in this part, seen from the other side. If trying is cheap, *right first time* is a luxury and not a requirement. And if you steer on *right first time* anyway, you will improve things that do not matter.

And this is not about programming

The shape of this sits everywhere people — or machines — repeat the same kind of work.

- **A kitchen that logs complaints by kind** rather than by how bad they were discovers after a month that three quarters are about the same thing, and that it is often one step in the preparation.
- **A teacher who writes down the ten mistakes every pupil makes** never has to explain those ten ten times over again. The list goes out in advance, and the lesson is about the rest.

- **An office where forms keep failing on the same line** does not need stricter checking but a clearer form. The mistake is not with the person making it.
- **A crew that writes one line after every breakdown** — what it was, what we thought, what it turned out to be — has, after six months, something no course can replace, because it is about this installation and not about installations in general.

In all four, the gain is not that fewer mistakes get made. The gain is that the same mistake costs minutes the second time instead of hours, and does not happen a third time.

Try it yourself · Add the third step that always gets forgotten

What you do. There are three incidents in `safety/incidents/`, all three neatly with a measure attached. So working it out and fixing it have happened. The question of this chapter is the third step: **who makes sure it does not come back?** Allow ten minutes.

TYPE THIS

> Go through the incidents in safety/incidents/. For each, check whether the measure has been recorded anywhere as a recurring check, or whether it only exists as text inside the incident. Then write a procedure for handling an incident, with as a required last step which check prevents this kind of incident from returning.

What you see. Three incidents, three measures, **and zero checks.** Hanging up cables, widening the cordon, handing out gloves: all three sensible, all three one-off, and none of them happens again next month by itself. **And one of them was never reported** — a fourth finding you get for free.

WHAT CAME BACK

```
2026-03-18 cable      measure: yes  check: no
2026-04-16 roof tile  measure: yes  check: no
2026-04-25 cut hand   measure: yes  check: no
                      (and not reported)
measures without a check: 3 of 3
```

Where it goes wrong. Do not ask whether the measure is any good. That is a trade judgment and no use to you here; the question is whether it survives when nobody thinks about it. **Make the last step required, not recommended.** Leave it optional and it gets skipped on exactly the day things are busy — the same day the incident happened. **And let "no check needed" be a valid answer, provided a reason comes with it.** Otherwise somebody invents a check to be rid of the field, and then you have a check nobody runs.

THEN LIKE THIS

> Add the last step to the procedure and show what it would have meant for the cable incident.

AND THEN THIS

```
handle-incident, step 4 (required):
  which recurring check catches this from now on?
  the answer may also be: none, because <reason>

applied to 2026-03-18:
  check: "is a cable running across a walkway?"
  added to the site walk-round
```

Not every lesson comes from something going wrong

One addition, or this chapter becomes one-sided.

There is a kind of knowledge that looks exactly like a failure and is not: the decision *not* to do something. You looked into it, you worked on it, and the outcome is that it should not happen.

That is a result, not a failure. Work was done and something was established. So it belongs under *finished* and not under *abandoned* — and it belongs written down, with the reason, because otherwise the same proposal comes round again in six months and the same investigation gets done again.

That is the same rule as above, only applied to something that did not hurt. Which is why it is forgotten even more often.

Try this: from now on put every mistake in one file, at the moment you solve it, with three fields — what was there, why it was wrong, what the right form is. After twenty lines or so, have the log grouped by kind, sorted by count, with one rule per kind that would prevent it. You are not looking for the most interesting mistake but the most frequent; they are rarely the same. Write down what actually happened and not what you thought of it — a log with opinions in it cannot be sorted a month later.

A log you do not send along is an archive. The next step is handing it over on the way — and only then does it work.

Then send them instead of explaining them: turn the rules you found into one block of at most ten lines that goes along with every instruction, with the full list of what *does* exist beside it. Against an invented component such a list is the whole answer. Keeping it short is not thrift but necessity: a block of three pages is read as background, ten lines are read as instruction.

Then measure again. But the number that actually matters, because the figure you look at by default is the wrong one.

And measure the number that matters: not how often it is right first time, but how often it is right within three attempts *with* the error message included. In practice nobody gets one attempt, and steering on *right first time* improves things that do not matter. This only works if your error messages are usable — if they are not, you are measuring your messages and not your model, and then you know straight away where the real work is.

> **What this made cheaper:** *every mistake was made once instead of again and again, and what it yielded traveled on to the next one.*

14. My tools lied to me

The error message was a tidy sentence. It named the line, it named the problem, and it proposed a fix. It was wrong on two counts.

What it said came down to: *I do not know that name — it belongs in that other file, add it.* That other file was already added, three lines higher. And the name did not belong there at all: it existed nowhere, it was misspelled.

Two untruths in one sentence, and both of the kind you do not see when you have nothing else.

Why this alarmed me more than it looks

It is a small fault. A misspelled name, and an error message pointing the wrong way. That costs a person half a minute.

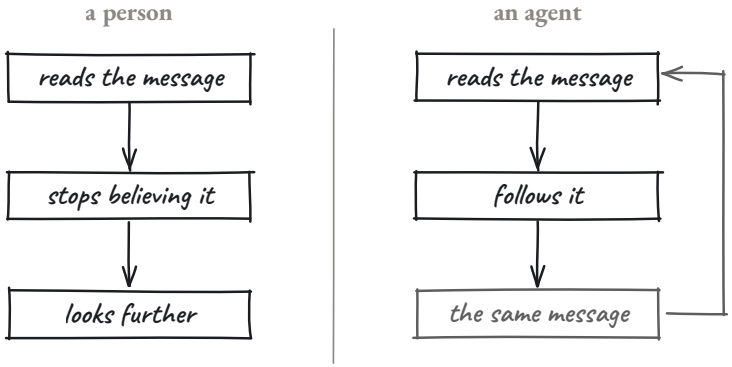
But look at what happens when the reader is not a person.

An agent reads that sentence and does exactly what it says: it adds what was already there. It compiles again. It gets word for word the same message back. And then it is finished, because the message has taken away the one fact it needed — that the name does not exist. It is in a loop that does not fail once but every time, and it has no reason at all to step out of it.

A person loses less here, and not because they are cleverer. They see that file sitting there where the message says it is missing, and from that moment they stop believing the message. That is not reasoning, that is distrust — a reflex, built out of years of messages that turned out to be nonsense.

An agent has no such reflex. It cannot acquire one either, because it starts every session fresh. What you call "experience with this tool" is precisely the part that does not travel.

With that, something shifts that I had not seen coming. As long as you do the work, a misleading message is a nuisance. The moment an agent does the work, **the quality of your messages is an upper bound on what is possible.** Your instruments weigh more than your judgment, because there is no judgment left to overrule them.



A person sees the message cannot be right and stops. An agent follows it and arrives back at the same point.

Four messages I believed

This was not my first. In the months before, in other work, there were four more — and each has its own kind of untruth.

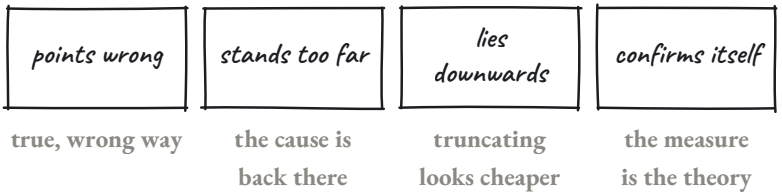
The message about the wrong subject. A task fell over with an error about authorisation: you are not allowed. A setting was missing. The message was entirely true — there *was* no valid access — and it pointed at permissions while the problem was configuration. This is the meanest kind, because there is nothing in it to refute. It is true, and it points the wrong way.

The message three steps too far along. Hundreds of tests fell over at once, and it read like a codebase in ruins. The environment had been set up halfway: the structure yes, the base data no. Without that data a template was missing; without that template creating a test user failed; and so I got a message about creating users. That message mentioned data nowhere. One line of enquiry said it outright — and it took an hour before I asked that line, because I was busy investigating the message instead of the environment.

The measurement that lied downwards. A count we were watching went down, and down was good: less work. Nothing had become cheaper. A combined read had run into a ceiling and was quietly truncating, so a few items were counted as free. **Since then a count that drops below expectation is as suspect as one that rises**, and that is not a rule you arrive at on your own.

The two theories that neither of them existed. The measuring tools had been built to prove two tidy explanations for something slow. Both turned out to be measurement artefacts — of how the measuring

was done, not of what was happening. The tools would have been proved right if nobody had checked again by another route.



Four kinds of untruth: points wrong, stands too far, lies downwards, confirms itself.

What the five have in common

All five worked exactly as they were built.

They failed in the translation from an observation into a sentence. The instrument saw something true and said something else — shorter, more helpful, and just wrong enough to send you in a direction. And the helpfulness is the dangerous part: a message that says only *something is wrong* leaves you searching. A message that proposes a fix has you carry out that fix.

Which is where the rule this chapter arrives at comes from, and it is shorter than it deserves:

A message is evidence, not a verdict. It tells you what the instrument observed. What is actually going on, you still have to find yourself.

And this is not about software

The shape of this sits in every trade where somebody watches a gauge.

- **The light on the dashboard** that comes on because a sensor has failed. The engine is fine, the gauge is not, and you are at the side of the road.
- **A result that says "not detected"** and is read as "not present". Both sentences are true about different things, and the difference decides what happens next.
- **A thermometer hanging where the sun reaches it.** It is not lying. It measures the temperature at that spot perfectly, and that spot is the wrong one.
- **A report that only shows what was entered.** What nobody entered does not exist on that screen — which makes the department that records nothing look like the department without problems.

In all four it goes wrong not at the measurement but at the sentence you tell about it. And in all four the recovery is the same: find one independent observation that ought to say the same thing.

What changes now that agents are reading along: an organization can run for years on gauges that hang crooked, because the people unconsciously correct for them. They know that one gauge always reads high. Put an agent on it and the correction is gone, because it was never written down anywhere. Then it surfaces all at once how much of your reliability lived in people's heads.

Try it yourself · Ask what the overview is about

What you do. `cost-control/` looks like the overview of the running jobs. It holds up, it adds up, and it is neatly maintained. The question of this chapter is not whether the figures are right but **what they are about.** Allow five minutes.

TYPE THIS

> Compare which jobs are in cost-control/ with which are in jobs/, and say what that means for a total you take from cost-control.

What you see. Six of fifteen. The overview is complete in its own terms and covers less than half the work. So any total you take from it is both correct and misleading, and nowhere does it say it is a selection. **Two of the nine missing jobs do have an outstanding supplier invoice** — so money is going out there that appears in no overview at all.

WHAT CAME BACK

```
jobs                    15
in cost-control          6
not in cost-control      9
  ENJB-2026-003, ENJB-2026-006, ENJB-2026-008,
  ENJB-2026-009, ENJB-2026-011, ... and 4 more
supplier invoice, no cost-control:
  ENJB-2026-006, ENJB-2026-011
```

Where it goes wrong. Do not ask whether the figures are right. They were; that was the problem. The useful question is what they are about and what falls outside. **Compare with the source, not with another overview.** Two overviews with the same selection confirm each other indefinitely. **And note that a missing row does not look like a missing row.** There is no gap, no zero, no warning — there is simply less, and that looks exactly like something complete.

THEN LIKE THIS

> Add a line saying how much of the jobs this overview covers, and have that come along from now on.

AND THEN THIS

```
COST CONTROL
  coverage: 6 of 15 jobs (40%)
  out of view: 9 jobs, 2 of which have an
               outstanding supplier invoice
  last updated: unknown
```

What to do about it

Three things, and the third is the only one that keeps working when you are not there.

Distrust a message that proposes a fix more than one that does not. Exactly the opposite of how it is meant. A proposal is the instrument's guess about your situation, and that is where it goes wrong.

Find a second observation before you change anything. Not the same measurement again, but another route to the same fact. Two instruments disagreeing is information; one instrument asserting something is a suspicion.

And repair the message, not just the fault. This is the part that gets skipped, because at the moment you find the cause your problem is solved and the message no longer feels like your work. Yet that is the only moment at which you know exactly what it should have said. An hour later that is gone, and it is still sitting there — ready to send the next reader, who is not a person, the same way.

Try this: once you have seen through a misleading message, repair it before you carry on. Find where that text is produced and work out what it should have said in three cases: when the instrument is certain, when it suspects something, and when it knows nothing. An instrument may well offer a suggestion, but only if it makes that distinction — the messages that do damage are the ones that write down a suspicion in the tone of a fact. Do it immediately: the knowledge of what it should have said exists exactly once, namely now.

That is what you do with the message you have already seen through. For the ones you have not, you want it to stand out the moment nothing changes.

Put this in your standing instructions: *if you get exactly the same error back after a change, do not go round again; write down what the message claims, which of those claims you checked and which you assumed, and then check the assumption.* Twice the same error after two different attempts almost never means the second attempt was too small — it means the message is untrue somewhere, and that untruth is in what you assumed without thinking. It only works if trying again is cheap; if a round is expensive, it guesses.

> **What this made cheaper:** *every message that is now correct is a fault that costs no round next time — and next time there is someone there who cannot build up distrust.*

15. The second time

The drawing only moved when the second screen asked for it.

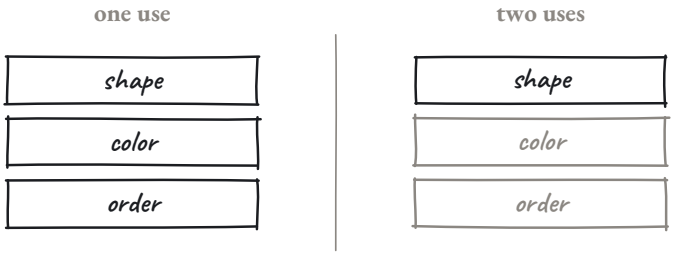
A component had been built — a list of rows, with a border round it and a rule under the heading. It sat in the screen it was made for, it worked, and it stayed there. Not because I forgot to file it away neatly, but because at that moment I did not yet know what about it was general.

Weeks later a second screen came along that needed almost the same thing. Almost. And in that *almost* was the answer.

What the second time tells you and the first cannot

With one use you cannot tell what is the fixed shape and what is accident. Everything there is there because *this* case asked for it. The width, the color, the order of the columns, the fact that there is exactly one line at the top — you cannot see which of those are about the rest and which are about this one screen.

With two uses you see it immediately. **What stays the same is the component. What differs is the dial.** You do not have to guess which things should become adjustable; the two cases point at them.



With one use you cannot see what is general. With two, the difference points at the seam.

From which follows the rule this chapter turns on:

> **First working, the second time reusable.**

It is an order of operations, nothing more.

Why the other order costs more

The temptation goes the other way, and it is strong. You build something, you see that it could come in handy again, and you make it general straight away. That feels far-sighted.

What happens then is this: **a general component derived from one case has the shape of that one case.** It is not general, it is that case with a few parameters round it. And the second use does not fit — not because it is so different, but because the seam is in the wrong place.

Then you face a choice that is bad either way. You bend the second case until it fits the shape of the first, and then the screen does something odd that nobody can explain. Or you rebuild the general part, and then the first case has to come along — including everything that by now leans on it.

Waiting for the second time costs you doing the same work once. The other order costs you a rebuild, and you cannot tell in advance how large.

But then the second time has to be cheap

Here is the condition that makes the rule work, and without it *first working* is an excuse.

If you copy and paste the first time, the second time is not a second time but a second copy. And then something unpleasant happens: from that moment on it costs more to merge them than to make a third copy. The first copy is the most expensive decision in this chapter, and at the time it feels like the cheapest.

Two things keep the door open, and both cost almost nothing:

Put it in one place, even if that place is still specific. One file, one function, one block. Not made general — only kept together. Then moving it later is a move rather than a reconstruction.

Write down that it exists. One line is enough: *this component lives here and is not shared yet.* The second use often comes weeks later, sometimes from someone else, sometimes from an agent that never saw the first. What cannot be found is not reused but rebuilt.



the middle step is where the design appears

First working in one place, then shared — but left so that moving it is a move.

Thirty-nine ways of working, and fifteen still carry the sentence that made them

I counted it in my own working folder, because I wanted to know whether this held or whether I was making it prettier than it was.

There are **thirty-nine** ways of working on disk: each a document that offers itself the moment it applies. **Fifteen** of them still carry, at the top, the literal sentence that brought them into being — usually something I said in passing, a median of **twenty-one words**, the shortest six. What came out is a median of **thirteen hundred words** each.

That is roughly sixty times as much text as I typed myself, and that is not the point. **The point is that I said that sentence once.** Every later time the situation came up, it was already there — including the times I was not.

They do not all do the same thing, and that is why it is a whole

When I laid them out, they turned out to be six kinds. Not designed; grown that way.

kind	how many	what it does
makes something	11	builds, writes, draws
checks what is there	10	measures whether it holds, and how you measure that
holds something back	2	refuses while a condition is unmet
looks with eyes	5	makes an image and measures inside it
learns	7	records what was learned, so it survives into the next round
touches the outside world	4	actually runs something, elsewhere, on a strange machine

The ratio is the interesting part. Eleven make something and twelve check whether it holds — the checks plus the ones that refuse. That is almost one to one, and it did not start that way: at first there were only makers. Every check was added on the day something went quietly wrong.

And they point at each other. Eleven pairs name one another in both directions: the maker says where its work gets checked, and the check says who hands it over. That is the handover, written down from both sides — and it is exactly what is missing when you do not.

The most-named is also not the cleverest but the most **shared**: fifteen point at the checking, ten at the place where things go out. Whoever is looking for where it pays to write something down should not ask what is smartest, but what crosses the fence most often.

Try it yourself · Build it once and use it twice

What you do. In chapter 9 you made a check on certificates: something with a `valid until` beside a date. `equipment/` holds exactly the same shape — an inspection with a `valid until` beside a day of use. **This is the second time, and this is the moment you KNOW what may be shared.** Allow ten minutes.

TYPE THIS

> Look at the check on personal certificates and at what would be needed to do the same for equipment/. First say what those two have in common and what differs per case. Then make one procedure that handles both, and show me which data you have to feed it each time.

What you see. What they share is small: **a thing, a validity date, and a moment at which it is used.** The difference is which field sits where. One procedure with three inputs covers both — and at once also the insurances, the subcontractor declarations and the guarantee periods, because those have the same shape.

WHAT CAME BACK

```
shared:
  - a thing with a name
  - a date after which it no longer holds
  - a moment at which it was used

difference:
  certificate -> moment = date of the job
  inspection  -> moment = day of use
```

Where it goes wrong. Do not make it more general than the two cases you have. One procedure for two things you know is a multiplier; one procedure for seven things you imagine is a framework nobody fills in. **Wait until the second time to merge.** Had you done this after the first case you would have shared the wrong things — you did not yet

know what differed per case. **And make the moment an input, not an assumption.** Measure against today instead of the day of use and the same check gives a different answer next year.

THEN LIKE THIS

> Now run that one procedure on all four: certificates, inspections, insurances and subcontractor declarations.

AND THEN THIS

```
watch-validity, applied to four kinds:
  personal certificates   4 jobs affected
  equipment inspections   2 expired
  insurances              0
  subcontractors          1 OSHA 30 expired
four administrations, one procedure, and nothing
was ever built for the fourth
```

That is also why this chapter sits here and not later. The second time is not the moment you save time — it is the only moment you *know* what may be shared.

This is steps five and six of the loop

Anyone who still has chapter 3 in mind will recognize this rule, because it is already there — only as part of something larger.

The loop that came out of the call center had six steps, and the last two were: **keep what works systematically**, and only then **formalise**. In precisely that order, and for precisely the reason above.

I did not write a step-by-step procedure there after the first call that went well. That would have been writing down a coincidence. I wrote it when I had seen the same movement for the umpteenth time and could

point at what was the same every time — and therefore also at what was allowed to differ every time. **That difference is the model.** It cannot be derived from one case, however good that case was.

And that is why "keep what works" comes before "formalise". Reverse them and you fix a shape and then go looking for whether it holds. Then you defend your model instead of testing it, and everything that does not fit becomes an exception instead of a clue.

The same order, in three trades, for the same reason. That is exactly what this book calls a pattern.

And this is not about software

The shape of this sits everywhere someone does something for the second time.

- **A recipe you cooked once** is not a recipe yet, it is what you did that evening. Only when you cook it a second time for different guests do you see which steps really belong in it and which were about that one night.
- **A form made for one department** carries that department's words. The second department fills it in with an uneasy feeling, and nobody can say exactly why.
- **A training you gave once** is a story. With the second group you notice which examples worked and which were only true for the first. Only then do you have a training.
- **A contract that comes out of one negotiation** carries that one counterparty's worries. Use it for the next and it contains clauses about nothing.

In all four, the mistake is not that something gets reused. The mistake is that the *model* was made at a moment when there was only one case.

And notice what this says about pace. The rule sounds like delay, and it is not. It only says: do the generalising at the moment it yields information rather than at the moment it is a guess. That is later, and it is cheaper.

When you may skip the rule

There is one case, and it is the important one.

If you already know for certain that it will happen a thousand times, the first time is not your first case — it is your only chance to get the shape right. What runs a thousand times cannot quietly be converted later; too much hangs off it by then.

So the distinction is not *how often do I think this will occur* but *how expensive will it be to change later*. For a screen component: cheap, so wait. For the shape you store everything in, the way your agents talk to each other, the place money passes through: expensive, so think it through now. There the thousand-times question applies, not this rule.

The two do not contradict each other; they are about different quantities. The thousand-times question is about the cost of repeating. This one is about the cost of fixing the shape too early.

Try this: when you build something resembling what already exists, ask before you start whether something like it is already there and how much it differs from what you need now. If there is, put the two side by side and have it named what is the same and what differs. The difference between the two cases *is* the design — you do not have to invent it,

only to see it. Do it before you build: afterwards you have a solution and you defend it instead of comparing.

That only works if the first time can still be found. You arrange that at the moment itself, and it costs five minutes.

Tip: when finishing something new, put the part together in one block — without making it general — with one line above it saying what it is and that it is not shared yet. Moving is cheap, reconstructing is not, and what cannot be found is not reused but remade. *Putting together* is not the same as *making general*: the first costs five minutes and keeps the door open, the second is exactly what this chapter warns against while there is one case.

> **What this made cheaper:** *every shared component is one whose shape came from two cases instead of from a guess, and that is why the third case fitted.*

16. Look for the key, not the doors

I had made a list of things that needed building, and the list was long.

The question was how an agent could make its own connections to the outside world: mail, servers, payments, orders, registering a company. I did what you do with a question like that. I made an inventory. Per service: what it needs, how you connect, what can go wrong. There were dozens, and every further line made the whole thing slightly less believable.

Then it landed, and it fitted into two sentences.

> **The basis is an email address, and being able to read and send from that mailbox. That opens the way to an account at almost any other service.**

I had counted thirty doors. This was the key.

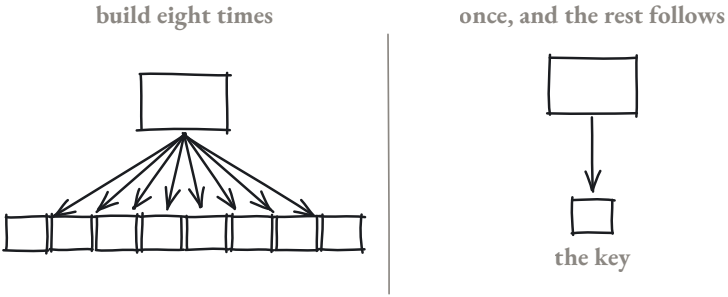
Why you end up counting doors

This is not a stupid mistake, which is why it is so persistent.

A question almost always arrives as a list. *We need to be able to mail, order, pay.* That shape invites an answer in the same shape: a plan per item. It feels thorough, it can be scheduled, and you can put an estimate next to it.

And it is defensible. Every door on the list is genuinely needed. Nobody can explain to you that you are doing the wrong thing — at most that it is taking a long time.

But a list of doors is a list, not a design. It tells you what is missing and not what has to come first. And with long lists the order becomes the most important decision you make, precisely because you usually do not make it: you start at the top.



Thirty doors, or one key that opens most of them.

What a key is

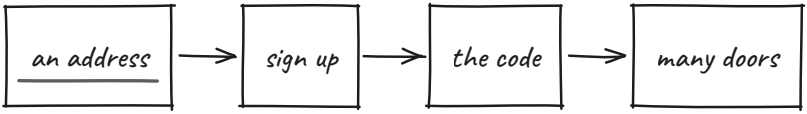
A key is not a clever trick. It is a perfectly ordinary thing with one special property: **it is the condition under several doors at once.**

In this case: an email address. There is nothing remarkable about a mailbox, and that is exactly why you overlook it. But look at what it makes possible.

You can sign up somewhere, because signing up asks for an address. You can receive the confirmation code, because it comes by mail. You can reset a password, because that goes by mail too. You can manage an account, because administrative messages arrive there.

And only then does it start. With an account at a service that offers connectors, you can reach *that* service's connectors. One sign-up gives access to thousands of connections somebody else has already built.

The chain is not long, and that is the point:



each step makes the next one possible

One address, and the rest follows — each step makes the next possible.

Building thirty doors costs thirty times the work. Making the key costs the work once, and after that it is somebody else's work.

How to recognize a key

Three questions, and all three can be answered before you start building.

Which condition comes back in more than one place? For each item on your list, write down what has to be true before it can happen. Usually the same thing appears three or four times, and that is your candidate. Not what is asked for most — what most often lies *underneath* something.

Can that condition itself be made without one of the doors? A key that sits behind one of your doors is not a key but a circle. An email address can be created without already having an account somewhere else; that is exactly why it works.

What becomes reachable afterwards that is not now? This is the question from the chapter on attainability, and here it returns. If the answer is "one thing", you have found a door and not a key.

Where I learned this move

The two sentences above turned my list around, but the move itself was not new to me. I had needed it years earlier, in work that had nothing to do with connectors.

Someone being guided into work arrives with a list. A job, an income, a rhythm, something to start with, maybe a course. **That is a list of doors**, and the temptation is to work through it: apply, register, find a training. That is what those lists are for.

What I learned is that it almost never went that way. There was usually **one thing underneath** that made all the others cheap once it existed — and that thing was never on the list, for exactly the same reason as above: it is not a goal. Nobody asks for it. It sits below.

And it was rarely a vacancy. **Predictability was more often the key than opportunity.** Once the week looked the same, once it was clear what would happen and in what order, the rest of the list suddenly became reachable. Not because anyone could do more, but because energy was left over that had been going into not knowing.

There is one lesson in this that goes further than the rest of the chapter. Sometimes the key is not with the person holding the list. If an employer keeps the door shut, there is nothing about that person you can improve that opens it — and then working on the list is working on the wrong side. That is the same point as in the chapter on ac-

cess, and it is the most unpleasant variant of this move: **sometimes the key is in somebody else's pocket.**

Then the question is not *what still has to happen* but *who has to say yes, and what do they need.* That too is one thing instead of a list. It is just a harder one.

And this is not about connectors

The shape sits everywhere someone makes a list of things that have to happen.

- **An organization that wants twelve kinds of report** does not need twelve reports but one place where the data arrives correctly. Building twelve reports on uneven sources is having the same conversation twelve times.
- **A school offering five kinds of support** runs into the same thing every time: whether there is someone who knows the pupil. That is the key, and it is on no list because it is not an offering.
- **A company that wants to sell in four countries** thinks about four translations and four payment methods, and the condition under all four is one thing: being able to invoice in a way that is valid there.
- **Someone who wants to get their life in order** and makes a list of ten habits usually discovers there is one underneath — sleep, generally — that the other nine depend on.

In all four, the striking thing is that the key was not on the list. It was not there because it is not a goal. Nobody asks for an email ad-

dress, for good source data, or for sleep. They sit underneath the things that are asked for.

Try it yourself · Look for the concept instead of the places

What you do. You now have four checks that all do the same thing. Rather than building a fifth: look for the **concept** underneath and have it applied everywhere at once. Allow ten minutes.

TYPE THIS

> Search this whole folder for every field holding a date after which something no longer holds — whatever it is called — and put them on one list, sorted by date, marking which have already expired. Use 31 May 2026 as the reference date.

What you see. Thirty-six expiry dates across six folders, and three of those folders you would not have named yourself: quotations have a validity date, guarantees run out, and so do insurances. **Nine of them have already expired**, including a quotation still marked *open* — so it can no longer be accepted and nobody knew.

WHAT CAME BACK

```
found in: staff 12, equipment 8, quotations 5,
subcontractors 4, insurances 4, aftercare 3
2026-01-31 EXPIRED  OSHA 30 subcontractor
2026-02-28 EXPIRED  working at height
2026-03-19 EXPIRED  van 2, state inspection
2026-04-05 EXPIRED  mobile tower inspection
2026-05-02 EXPIRED  ENQT-2026-03 (status: open)
```

Where it goes wrong. Say "whatever it is called". In six folders this field has six names: *valid until*, *state inspection valid until*, *inspection valid until*, *runs until*. Ask for one name and you find one folder. **And then look at what is NOT on the list.** Here the scan missed the com-

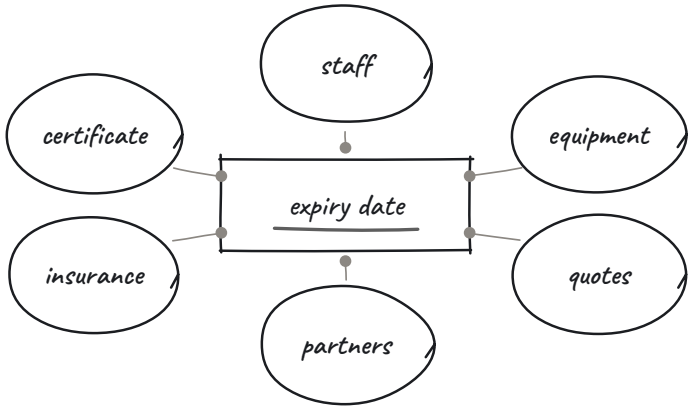
pany certificates, because those sit in a table with the date in a column rather than behind a word. Among them was the OSHA 30, expiring in thirty days — the certificate asked for in virtually every tender. **A key that opens almost everything almost always fails to open exactly that one.**

THEN LIKE THIS

> Also look for dates sitting in a table column rather than behind a word, and add them to the same list.

AND THEN THIS

```
three company certificates found after all:
  2026-06-30  in  30 days  ISN
  2027-11-30  in 548 days  ISO 9001
  2028-02-28  in 638 days  guarantee scheme
39 instead of 36, and the first one to expire is
immediately the most important
```



One concept opens six administrations — and not that one lock.

When you do have to do the doors

This is not a trick that always works, and there are two cases where it misleads.

When there is no shared condition. Sometimes five things are just five things. Then hunting for the key is delay, and the honest conclusion is that it is five times the work. You notice this at the first question above: if nothing comes back three times, there is nothing.

When one door yields money or time now and the key only later. The key is almost always the better move, but not if you never get to it. That is the same weighing as with attainability: the best move you cannot make is not a move.

What you do in both cases is write it down. A shared condition you skip today is still the key in three months — and by then you will have forgotten it.

Try this: when you have a list of things to build, ask per item what has to be true *before* it can happen — conditions only, no solutions — and then group the conditions that recur across items. Take the one that recurs most and ask whether it can be made without any of the items, and what becomes reachable afterwards. The key is never on the list, because it is not a goal. Ask for *conditions*: ask for solutions and you get a plan per item, and then you have neatly copied out your list.

Sometimes you do not get round to it, and that is a valid outcome. It just has to be written down somewhere, or you will find it again by rediscovering it.

If you are not making it now, write it down as a separate item, with which things it makes possible and what it costs to make — no plan-

ning, just that it exists. A key you skip today is still the key in three months; if it is written nowhere, you meet it again by rediscovering it, and by then you have built three doors by hand. Note what it makes possible and not only what it is: a line that says only *"need an email address"* reads three months later as a chore.

> **What this made cheaper:** *filling one condition once instead of building thirty connections — and everything that came along afterwards no longer had to pass that question.*

17. Zero dependencies

Not a single afternoon was lost to something that belonged to somebody else.

That sounds like a detail and it is the reason so much fits into it. Anyone who works with software regularly knows the pattern: you start on something, and before you have started there is a version that no longer matches, a component that is called something else today, or a package that has disappeared. Not a disaster, half a day each time, and never the work you sat down for.

Those afternoons did not happen. Not because things went well, but because there was nothing that could cause them.

Why zero is different from few

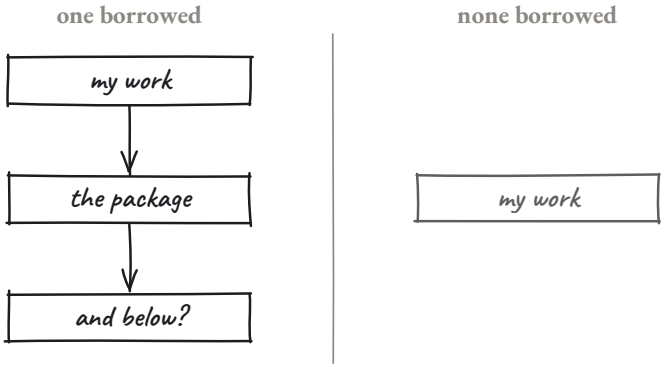
It is a step and not a scale, and that is what the chapter turns on.

With one borrowed component you have a chain. You have to know who makes it, whether it is maintained, what happens when it stops being maintained, and what sits *under* that component — because it has borrowed too. With ten components you have the same questions ten times, and you stop asking them.

At zero there is no chain. Not a short chain — none. The question "what happens if this changes" disappears, and with it everything that

follows from it: the tracking, the checking, the working out on the day something stops working.

From ten to one is an improvement. From one to zero is a different state.



From ten to one is less. From one to zero is different: the chain disappears.

What you are actually borrowing

The misunderstanding is about what you think you are bringing in. You think you are bringing in code. What you are bringing in is a promise.

The promise that it will continue to exist. That whoever made it is still there in three years, or that somebody else takes it over.

The promise that it will keep working the way it worked. Not that it is flawless — that it does not quietly change behavior.

The promise that nothing strange will happen. That nobody puts something in it that does not belong, and that nobody takes it over with a different purpose.

Those three promises are usually true, and that is exactly the problem: they are true so often that you stop counting them. What happens then is not that one promise breaks. It is that you have a hundred of them by now and no longer know which.

Zero is a requirement, not a score

Here is the sentence the chapter turns on.

> **Zero is a requirement, not a score.**

The difference is the whole story. A score is something you try to do well on: three is better than ten, so we do our best. A requirement is a line that changes the question. With a requirement of zero, "is this package good enough" is no longer a question, and another one takes its place: **do I actually need this?**

That is almost always the better question. A surprising share of what you would borrow turns out to be something you can write yourself in thirty lines, that fits exactly, and that never changes again. You only discover that when the easy route is closed.

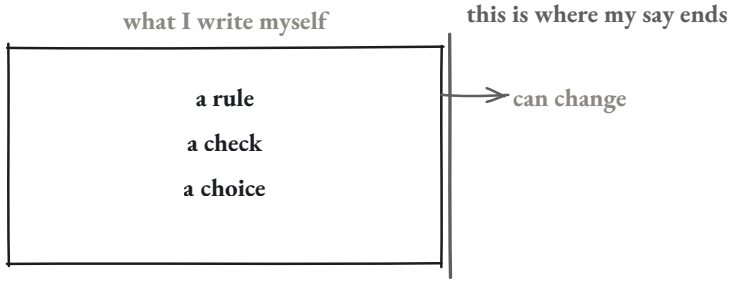
And a requirement works where judgment fails. In every individual weighing you lose to yourself: this particular package is large, well maintained, everyone uses it. All true. But you make that weighing a hundred times, and you do not lose it once, largely — you lose it a hundred times, slightly.

And this is exactly why it counts more now

As long as you are the only one adding code, a judgment per case is still workable. The moment agents are involved, it is not.

An agent cannot weigh the three promises. It sees that a package exists, that it is popular, and that it solves its problem. What it does not see is what happens to it in two years, who is behind it, or how deep the chain underneath goes. It adds it, for good reasons, and it does not report it as a decision.

And it does that faster than you can review. A rule that rests on judgment does not work at that speed. A line does: **nothing from outside** can be checked without a discussion, and that is exactly what a rule has to be able to do.



What you write yourself changes only when you change it. Everything else can shift without asking.

The line is not "typed it myself"

This chapter would be nonsense without this section, because zero is never literally zero.

An operating system is running. There is a processor. There is a language. You lean on those, and that is not an inconsistency — it is drawing the line in the right place. The question is not *did I write it myself* but: **how likely is this to change under me, and what happens to me when it does?**

The way a program asks the operating system for something has been the same for decades and does not change because somebody releases a new version. A package that became popular last year is not that.

Seen that way, this chapter is the counterpart of the chapter on lock-in. There it was about providers of capability: who supplies the thinking, and what if they fall away. Here it is about what you lay underneath your own work. The same question, two layers: **what can shift under me without anyone asking me.**

What it costs

You write more yourself. That is the price and it is real.

Some things you simply do not get. What fifty people built in five years, you are not going to rebuild in a week, and if you try you will get something worse. So the honest form of this rule is not "never borrow anything" but **"zero, and where that is impossible, deliberately and visibly"** — with the reason attached, so it is a decision and not a habit.

And there is a version of this mistake that is just as expensive: building everything yourself because you enjoy it. That is the same outcome with a different story around it, and it does cost you the time.

The test is the afternoon. How many afternoons did you lose in the past year to something of somebody else's that changed? At zero you are right. At five that is a week, and that is no longer an inconvenience but a line item.

Try it yourself · Look at what you are borrowing from others

What you do. Four subcontractors do work you do not do yourself. That is not purchasing but borrowing: under chain liability their problem ends up with you. Have it checked what you know about them and how old that is. Allow five minutes.

TYPE THIS

> Assess each subcontractor in subcontractors/ on three things: is the OSHA 30 still valid, how old is the payment-behavior declaration, and is there a blocked account. Note which running jobs they are working on.

What you see. Three of the four are in order. **The fourth is working on a running job, has a OSHA 30 that expired four months ago, a payment declaration well over a year and a half old, and no blocked account.** Each of those three is an administrative detail on its own. Together it is the one party where a problem of theirs becomes a problem of yours.

WHAT CAME BACK

```
calder    OSHA 30 valid     declaration 110 days  acct yes
marsden   OSHA 30 valid     declaration 123 days  acct yes
redfern   ISN valid    declaration  88 days  acct yes
thorpe    OSHA 30 EXPIRED   declaration 555 days  acct NO
          working on ENJB-2026-006
```

Where it goes wrong. Ask for the three things together, not separately. Each on its own is a remark you read past; together it is a decision. **Ask which work they are on.** A party out of order doing nothing is a note; a party out of order on a running job is today. **And note that "no blocked account" looks more innocent than it is.** The other two points carry a date and therefore stand out; this is an empty field, and empty fields you do not see.

THEN LIKE THIS

> For each subcontractor, say what it costs us if it goes wrong, rather than only what is missing.

AND THEN THIS

```
thorpe, on ENJB-2026-006 ($9,870):
  OSHA 30 expired  -> liable for an accident on our
                      site
  no account   -> unpaid payroll tax lands with
                      us
  old declaration -> says nothing about now
```

And this is not about software

- **A kitchen that makes its own stock** has more work and never has the problem that the taste changes because a supplier adjusted their recipe. Its freezer is more stable than somebody's production line.
- **A mechanic with their own tools** never waits at an issuing counter. That looks like a comfort question until the day the counter is closed.

- **A practice that keeps its own records** rather than only living in somebody else's system can move. Whoever does not, cannot, and finds out at the wrong moment.
- **A farmer who saves their own seed** gives up some yield and buys independence with it. That is exactly the same trade as here.

In all four it is not about distrust and not about wanting to do everything yourself. It is about which promises from others you have built into your own work without writing them down.

Try this: have everything listed that your work leans on from outside, including what arrives indirectly, with for each who makes it, when it was last updated, and what breaks if it disappears tomorrow. Almost nobody knows that number, and it is usually a multiple of what you would name yourself — because most of it arrives indirectly, through something you *did* choose. That is also where the surprise is; what you chose yourself you already know.

That number is the occasion, not the solution. The solution is a line that does not have to be weighed afresh every time.

Put this in your standing instructions: *never add an outside dependency; if you meet one you think you need, do not build it in but report it — what it does, how many lines it costs to do it yourself, and what we lose by not doing it.* An agent cannot weigh what a dependency costs in the long run and does not report adding one as a decision. Ask for that number of lines: it settles the argument more often than you would think, because a great deal of what gets borrowed is thirty lines.

What this made cheaper: *no afternoon was lost to somebody else's change, and every day that did not go into maintenance became available for something that moved forward.*

18. The flywheel

One morning something had got better without my having made it better.

An error message that the week before still pointed the wrong way now pointed the right way. I had not changed it. What had happened: that message had ended up in a log because it had misled someone, the log had been sorted, the top lines had found their way into an instruction, and that instruction had been traveling along ever since. The improvement came out of the work itself.

It is a small event and I remember exactly what it did: it felt as though something had let go.

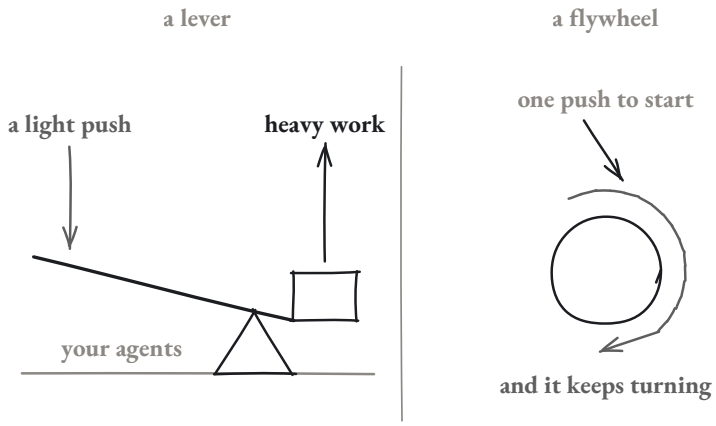
A lever and a flywheel are not the same thing

Up to here this book is about levers. Small input, large output — you push in one place and more comes out than you put in. A multiplier is the tidy name for it: work that makes the next work cheaper.

But a lever only works while you push. Take your hand away and nothing more happens. All the gain is in the ratio between your force and the result, and your force is still the source.

A flywheel is something else. It stores. You get it going — slowly, with a lot of effort, and for the first few turns it looks like a waste of energy —

and at some point it keeps turning without you. Not because it makes energy, but because it has stored enough to overcome its own friction.



A lever multiplies while you press. A flywheel you push once, and then it keeps turning.

That is the hinge of this book. Parts I and II are about building levers. From here on it is about when the thing starts turning by itself.

What gives a flywheel its mass

A flywheel with no mass is a wheel: it turns while you push and stops afterwards. The mass is what you have stored in it, and in this work that is one thing.

Everything that has been written down. The mistakes that were noted and became rules. The tests that already existed when this change arrived. The decisions recorded with their reasons, so nobody has to take them again. The examples that travel along. The checks that run with everything.

Each of those cost time on the day and yielded nothing — that is the definition of a multiplier from Part I. The difference is that now they do not only make the next work cheaper, but **together are enough to complete a turn without anyone pushing it.**

Which is also why it takes so long before you notice anything. Building mass looks, right up to the last moment, like yielding nothing.

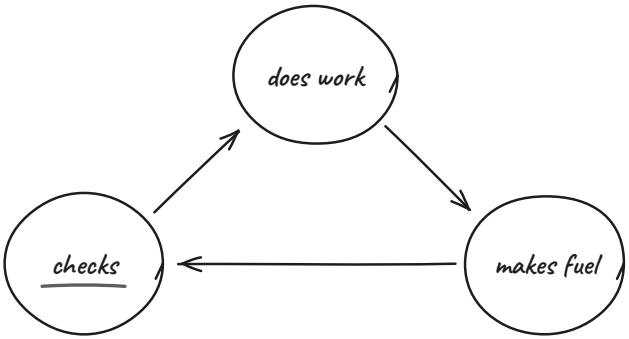
The three conditions, and the third is the one forgotten

A loop only turns by itself when three things are true. The first two are obvious; the third is where it usually fails.

One: it makes its own fuel. What the loop produces has to feed the next turn. An error log nobody reads feeds nothing. A log that yields rules which travel along does feed the next turn.

Two: the turn closes without you. If somewhere in the loop there is a step you have to click, it is not a flywheel but a lever with a pause in it. That is allowed — but call it that, because you will notice the difference on the day you are not there.

Three: the check is cheaper than the turn. Here it goes wrong. A loop that feeds itself and closes without you, but whose outcome you cannot cheaply verify afterwards, does not keep turning — it wanders off. And you only notice when something visible breaks, which means something invisible had been broken for a while before that.



Feed itself, close without you, cheap to check — all three, or it is not a flywheel.

Why you do not put a hand into it

When it is turning, the temptation to improve it is greatest. You see something that could be more elegant. You have time. It is your thing.

You do not put a hand into a spinning flywheel.

That sounds conservative and it is not. The reasoning is arithmetic. Every change to something that runs costs a check you would otherwise not have had to make, and the improvement you had in mind is usually small. Against that stands a chance — not large, but not zero — of breaking something that since then nobody checks any more, precisely *because* it was running.

And notice what changes now that agents are involved. **An agent has no hesitation.** In a person something sits between seeing an improvement and making it: a vague sense that this part has been fine for a while. That sense is built from experiences that were never written

down, so it does not travel. An agent that makes something "a bit tidier" does not know what was already good about it, and it does not ask.

Which is why *leave it alone* is not something you hope for but something you write down. In the instructions, in the ticket, in the code. What runs and is no longer watched should say that it runs.

And this is not about software

Flywheels are older than this subject and you recognize them by the same feeling: nothing happens for a long time, and then it happens by itself.

- **A business where customers bring other customers.** The first twenty all took effort. Somewhere after that they start fetching each other, and the effort per customer falls without your having changed anything.
- **A team that inducts new people itself.** As long as you do it, it is a lever. The moment the previous newcomer inducts the next, it turns — and the quality then hangs on what was written down, not on who happened to have time.
- **A record that keeps itself up to date** because updating it is part of the work rather than something afterwards. What comes afterwards does not happen.
- **A habit that no longer costs willpower.** The same shape, in miniature: for the first weeks you push, and at some point not doing it is more effort than doing it.

In all four the question is the same. Not *how do I make this faster* but **what has to be stored before it can carry on without me.**

Try it yourself · Set an alarm and look again next week

What you do. You now have five procedures that each check something. Until now you start them. **In this chapter you stop doing that.** You make one that calls the others, and you set it on a fixed day — and after that you no longer check whether it ran, but what it found. Allow fifteen minutes, and then a week of nothing.

TYPE THIS

> Make a procedure that calls the existing checks one after another and reports only what is new or expired, with whatever expires within thirty days at the top. Then set it up to run every Monday morning, and show me what would come out the next time.

What you see. A procedure that does nothing itself and only calls — **the first one you make that contains no check at all.** That feels empty and that is the point: what it adds is the order, the summary, and the fact that it goes without you. **At the top sits whatever expires within thirty days**, because that is the only thing in this round where time works against you.

WHAT CAME BACK

```
MONDAY ROUND
expiring within 30 days:
  ISN company certificate   in 30 days
new since last week : 0
resolved            : 0
unchanged           : 15  (1 round so far)
ran: check-the-rules, watch-validity,
     check-the-rules-audit
```

Where it goes wrong. Do not have it check anything new. The temptation is to build one extra test into it; then you have a sixth check instead of a whole, and you no longer know which of the six found something. **Put the expiring items at the top, not the bottom.** The

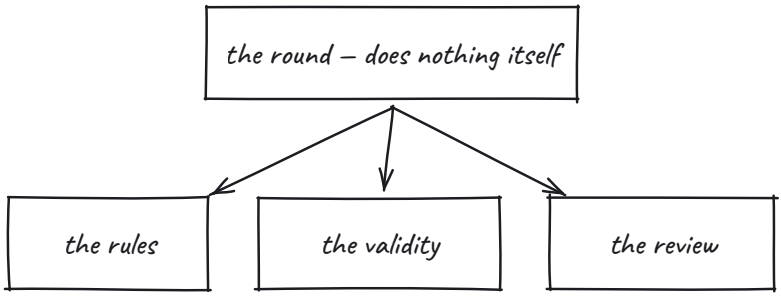
rest of the list can wait another week; a certificate expiring in thirty days cannot. **And do watch the first run.** A round you have never seen go red tells you nothing — the same argument as chapter 9, now applied to the round itself.

THEN LIKE THIS

> Besides the count, have it show how long each open point has been there, and which one sits longest.

AND THEN THIS

```
MONDAY ROUND, week 2
expiring within 30 days:
  ISN company certificate   in 23 days
longest open:
  ENJB-2026-005 missing              2 rounds
  ENXW-2026-06 unconfirmed           2 rounds
  thorpe OSHA 30 expired              2 rounds
new: 0   resolved: 0   unchanged: 15
```



what it adds is the order, and that it goes without you

The round does nothing itself: it calls, summarises, and goes without you.

The honest part

A flywheel is a bad answer to being in a hurry.

It starts slowly. The first turns yield less than you put in, and that is not a sign that it is going wrong but how it works. If you measure at that moment whether it is paying yet, you stop — and that is precisely why most people do not have one.

So measure the mass, not the yield. How much is written down that was not there last month? How many turns closed without you? How many checks ran along by themselves? Those numbers move *before* the yield moves, and they are the only honest signal you have in the run-up.

Try this: once a month, ask which things improved without you giving an instruction for it, with for each what the improvement was and what set it off. That is the only number that counts in this part: not how much *you* did, but how much happened without a push. Zero is a fine result early on; zero after three months means no loop has closed yet. Ask what set it off — an improvement whose cause you cannot trace is not a flywheel but luck, and luck does not come twice.

If that count is zero, one thing is usually missing — and it is almost always the same thing.

And build in the third condition: for every loop that will run without you, ask what can go wrong *silently* — without an error — with for each a check that makes it visible and what that check costs per round. Feeding itself and closing without you are the easy two; a loop without a cheap check does not keep turning but drifts away, and you notice only when something visible is broken. Watch that cost per round: a check more expensive than the round gets switched off as soon as things get busy.

And once it is turning, one thing remains. It is the easiest of the three and the most often skipped.

Tip: put one line above everything that runs stably saying *that* it runs, since when, and where the check is that guards it — no explanation of how it works. An agent has no hesitation: what stops a person from making an unnecessary improvement is a vague sense that this has been fine for a while, and that sense is written nowhere, so it does not travel. One line; make it a story and it will not be read.

> **What this made cheaper:** *from here on an improvement did not always cost a decision of mine — and that is the first thing that has to happen before this can become an organization.*

19. What an agent lacks

The fleet screen was there, and it was full of numbers I had made up myself.

An overview had been built of all the agents at work: how many were running, what they were doing, how long they had been busy, what it cost. It looked good. Every number on it was fake — a stand-in, put there because the screen would otherwise be empty.

That screen was more useful than a working screen would have been, because it showed exactly what was missing. Not the technology. **The things those numbers should have come from.**

Seven things, and none of them is technology

Put an agent next to an organization and the difference is not that one is human and the other is not. The difference is a list, and it is shorter than you would think.

A goal that outlives the task. An agent has a task. An organization has something that task contributes to, and that is what decides whether the task should have existed at all. Without it you do things very efficiently that nobody needed.

Division of work. Who does what, and — more importantly — who does *not* do something because somebody else already is. Ten agents

without a division are not ten times one agent; they are ten agents getting in each other's way and doing the same work three times.

Money. Not being able to pay: knowing what something costs and whether it was worth it. An organization that does not know that carries on with everything that works until it cannot.

Memory outside the session. What was decided and why. An agent starts fresh every time, and without a place where the decision lives, the same decision gets taken again every time — and not the same way each time.

Self-maintenance. Something that notices when things go askew and puts them straight without anyone watching. That is the flywheel from the previous chapter, and it is why an organization survives a night.

A limit. What is allowed and what is not, recorded somewhere it cannot be negotiated away. The next chapter is about that.

A signature. Someone who *is* it. An organization can enter into an obligation because something exists that is liable. An agent cannot, and that is not a technical shortcoming but a legal fact.

what an organization has	what happens without it
goal	efficiently the wrong thing
division of work	same work three times
money	on until it cannot
memory	decided again each time
self-maintenance	it survives no night
limit	nothing holds anything back
signature	no business to be done

Seven things, and technology is none of them.

Why the made-up screen was useful

There is a temptation to build such a thing only once the data exists. That is exactly backwards.

The screen is the question. Every box on it asks for something that has to come from somewhere, and as long as the box is not there, you do not know you have that question. With fake numbers in it the question is there, and it gets asked again every day by the screen itself.

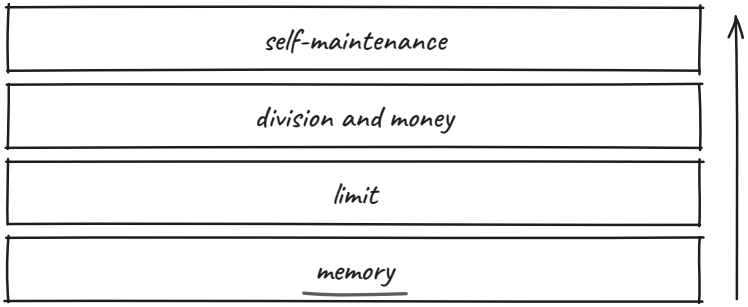
And it makes the order visible. You see at once which number you can make real first, and which one can only follow once something else exists. That is a plan that wrote itself.

What you must add: write down what is fake. A made-up number that stands long enough gets believed — by you, by a colleague, by an agent reading it out. So note that it is a stand-in and what has to replace it.

The order in which they appear

They do not come at once, and they do not come in arbitrary order.

Memory comes first, because without a place where something stays, none of the other six can exist. Then **the limit**, because as soon as more than one agent runs, something has to hold them back. Then **division of work** and **money**, both of which read from that same memory. Then **self-maintenance**, because it leans on all four. **The goal** runs through all of it and is the only one that has to come from a person. And **the signature** stands apart: it comes from outside, and a later chapter is about that.



Memory first, because the rest leans on it.

The same seven, in a new person

When I first wrote this list down, I had the feeling I had seen it before. That was right; it just had not been about machines.

Someone walking in on their first day is missing precisely these seven. Not because they cannot do anything — that is usually the one

thing that is established — but because everything else is not written down anywhere yet. They know what they have been asked and not why. They do not know who does what. They do not know what something may cost, where last year's decision is recorded, who notices when they get stuck, where their authority stops, and they sign for nothing.

A good induction is nothing other than handing over those seven. And **the ones that get skipped are the same every time.**

The goal and the task everybody gets; that is where every conversation starts. The division of work usually follows. But the **memory** — where it says why we do it this way — is rarely handed over, because the person who knows no longer knows that they know it. The **self-maintenance** is forgotten: nobody agrees who notices when it gets stuck, so nobody does. And the **limit** only becomes clear when someone crosses it, which is exactly the wrong moment.

So this is not a list about agents. It is a list about anything new that has to take part. That an agent lacks it is unremarkable. What is remarkable is that with an agent you can enforce what with a person you can only hope for: the memory you *can* write down, the limit you *can* record, the self-maintenance you *can* build.

And one of them you never hand over, not even to a person. The signature. Even an experienced colleague does not sign for the company; someone with authority does, and that authority is deliberately scarce. So an agent not getting it is not a gap it has to close — it is the same rule that applies to everyone.

Try it yourself · Have the spreadsheet recalculated that nobody reads any more

What you do. Estimating happens in a spreadsheet, and that is not going to change. `estimates/sheets/` holds two sheets as they come out of such a program. Both add up perfectly. **The question is not whether they add up but whether they hold**, and that is something else. Allow ten minutes.

TYPE THIS

```
estimates/sheets/ holds estimates as CSV. Check three things per sheet:
does every line equal quantity times unit price, does the sheet add up to
the stated cost price, and does that cost price match the quotation that
went out. Report only what does not hold.
```

What you see. The addition holds in both — a spreadsheet is good at that. **But one line has been typed over the formula**: 24 × 620 should be 14,880 and it says 12,480. A transposed digit. The sheet is therefore 2,400 too low and still adds up perfectly, so from the outside there is nothing to see. **And the second sheet comes out 15 per cent above the quotation that has already gone out.** That job makes a loss, and nobody knew.

WHAT CAME BACK

```
ENES-2026-01  addition holds
  line 2: 24 x 620 = 14880, it says 12480
ENES-2026-05  addition holds
  line 8: no quantity, 4800 was typed in

sheet against quotation:
  ENQT-2026-01  cost 26447  quoted 28400  +6.9%
  ENQT-2026-05  cost 47846  quoted 41500  -15.3%
```

Where it goes wrong. Do not ask whether the sheet adds up. It does, and that is exactly why nobody ever looks. The useful question is

whether every line still follows from its own formula. **Compare the sheet with something outside the sheet.** Two numbers in the same file confirm each other; only next to the quotation does it become visible that something is wrong. **And this is the point where you cannot hand over the judgment.** The machine sees within a second that 12,480 does not follow from the formula. Whether that is a typing error or an agreed discount somebody entered by hand, it does not know — and that is the question that matters.

THEN LIKE THIS

> For each finding, say what you can establish and what somebody has to answer.

AND THEN THIS

```
line 2, ENES-2026-01
  established: differs 2400 from the formula
  to answer:   discount or typing error?
ENQT-2026-05
  established: 6346 below cost price
  to answer:   accepted deliberately, or were
               lines added after the quotation
               went out?
```

And this is not new

Every organization that ever grew from one person has had to invent these seven one by one, usually at the moment something went wrong.

- **The goal** gets written down on the day two people do opposite things and both are right.
- **The division of work** appears when two people turn out to have done the same thing.

- **The money** only gets measured after a month when it does not add up.
- **The memory** gets built after somebody leaves.
- **The self-maintenance** arrives when somebody is on holiday.
- **The limit** arrives after an incident.
- **The signature** was there from the start, because without it you cannot do business.

The good part is that you can now lay them down in advance. Not because you are cleverer, but because it has become cheap and because you know which seven they are. That is the whole difference between knowing this list and discovering it.

Try this: have an overview made of what is running right now, with the numbers you would want to see — and per number where it should come from, with the invented ones clearly marked. An empty screen asks no questions; a screen with placeholders asks the same seven questions every day, with the order in which you can answer them attached. Do insist the fakery stays visible: an invented number that stands for three weeks gets believed.

That screen tells you what data is missing. The seven tell you what organization is missing, and that is a different list.

And run through the seven for what you have now — purpose, division of work, money, memory, self-maintenance, limit, signature — with for each whether it is there, half there, or missing, *and* how you can tell. Almost everybody has three or four without knowing it and is missing one that blocks everything; which one differs per situation and

cannot be guessed. Ask how it can be told, or you get an estimate worth exactly as much as your own hunch.

> **What this made cheaper:** *the screen with made-up numbers became the task list, and every time one number became real, nobody had to work out what came next.*

20. Limits, and who signs

The question that made everything concrete was: **how much may it spend?**

Very practically: something was ready that could buy a service by itself, and before that was allowed there had to be a number. And the moment you try to write that number down, you notice you had not written down four other things either.

Why a limit is not a prohibition

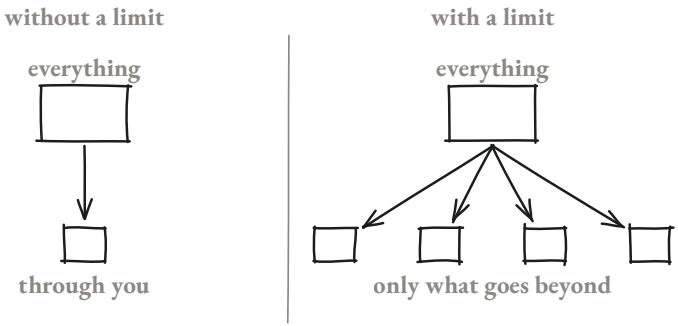
A prohibition is something you add after something went wrong. A limit is something you draw in advance so nothing does.

But that is not the important difference. This is:

A limit makes the game possible rather than restricting it.

That sounds like wordplay and it is daily practice. As long as there is no amount, every expense has to pass me. The moment there is an amount, it does not, and so something can happen while I am not there. **The limit is not the brake on autonomy; it is the condition for it.**

That is how rules of the game work everywhere. A game without rules is not a freer game but an argument. The rules are exactly what means you do not have to negotiate every move.



Without a limit everything passes through you. With one, it runs on until the limit.

The brake belongs with the key

Here is the rule that comes up most in practice, and it is short.

Whatever can do something must also have something that stops it — and those two belong to be arranged in the same place.

If an agent gets access to a means of payment, the limit is set at that same moment. If it gets access to a mailbox, who it may and may not write to is recorded at that same moment. If it gets the right to delete something, what may irrevocably go is arranged at that same moment.

Why at the same moment: a limit you add later gets drawn around what is happening by then. And what is happening by then feels normal.

What I learned about this from people, not machines

This is the chapter where my other trade shows through most strongly, and the point is not what you would expect.

In support work I worked with people for whom the world was unpredictable. And the striking thing was this: **a clear limit gave relief rather than restriction.** Not because anyone enjoyed being told no, but because uncertainty about what is allowed costs more than the restriction itself. Someone who does not know where the edge is has to guess at every step — and guessing is exhausting.

Ask someone what they are allowed to decide themselves and you often get no answer. That is not unwillingness. It is that it was never said, so the safest route is taken: ask about everything, or do nothing.

That is exactly the behavior you see in an agent without a limit, and it costs you money in the same two ways. It asks permission for everything — then you do not have an agent but a slower version of yourself — or it does something that turns out not to have been allowed. Both cases have the same cause, and it is not its judgment: **nobody said where the edge was.**

A limit you only discover by crossing it is not a limit

This is the second lesson, and it is the one most often broken.

If you only hear that something was not allowed at the moment you have done it, then you were not given a limit but a punishment. That does not work with people, and it works no better with an agent — with this difference: a person remembers it afterwards, and an agent starts again tomorrow.

A usable limit therefore has three properties, and all three are checkable:

1. **it is written down somewhere**, not in the head of whoever enforces it

2. **it can be read in advance**, so before the action and not after
3. **it intervenes by itself**, so it does not depend on who happens to be watching

Without the third it is an agreement and not a limit — and an agreement does not stop anything at three in the morning, which is precisely when this kind of thing happens.

And that is the third thing I take with me: build it up, step by step. Nobody gets all the room on day one, and nobody should have to ask about everything on day one. The room grows with what demonstrably went well. In support work that is the most ordinary thing there is, and for an agent it is exactly the same movement: **a limit is not a fixed number but a number allowed to move, on the basis of what has been measured.**

Four limits you have to make concrete

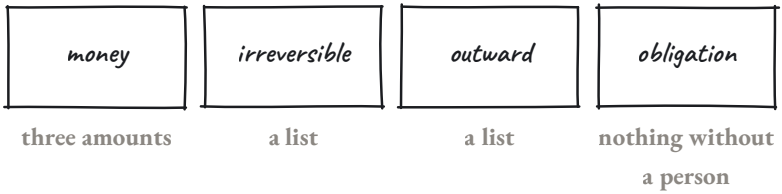
Four, and all four are a number or a list — not an attitude.

Money. Per action, per day, and in total. Three numbers, not one: one large expense and a hundred small ones are different problems, and a total without a daily amount stops nothing until it is too late.

Irreversibility. What may disappear without anyone confirming it. This is the most important and the most often forgotten, because deleting does not feel like spending.

Outward. What may leave the building, and to whom. A message to a customer is a different thing from a message to yourself, and the difference is not technical but social.

Obligation. What it may agree to on your behalf. Here the answer is almost always *nothing*, and that is not caution — it is the next part of this chapter.



Four limits, and three of them are a number.

And then the signature

Here is where what you can arrange yourself stops, and that is not a shortcoming of the technology.

An agent cannot be a party. It cannot enter into an agreement, cannot carry liability, cannot have an obligation. If something is agreed, there is always somebody or something else that *is* it — a person, a company, a legal form. That is not a rule that might still change; it is how the system we live in is put together.

That has a practical consequence you feel immediately. **Everything that enters into an obligation passes through something that can sign.** An order, a subscription, an instruction to a supplier. That may perfectly well happen automatically — but the signature stands in the name of something that exists, and that name has to have been arranged.

And there is the bridge to the last part of this book: **an autonomous organization is not autonomous because there is no longer a person. It is autonomous because the person is only still at the signature.**

Try it yourself · Build something that holds back instead of reporting

What you do. Everything you have made so far reports. A report can be ignored, and it is — `ENXW-2026-06` has been on the list for two rounds. **This is the first one that does not report but holds back.** Allow ten minutes.

TYPE THIS

> Make a gate on the extra-work limit from rules.md. It gives two answers and no third: PASS, or HELD with the amount, the job and who has to sign. It proposes nothing and it lets nothing through because it suits. Run it and show me the result, including whether it fails.

What you see. One line of text and an error code. **That last part is the difference**: a report ends tidily, a gate ends in failure — and because of that, something behind it cannot proceed. Whoever wants to pass it puts an approval with a date on the job, and that is an action you can see afterwards.

WHAT CAME BACK

```
HELD  ENXW-2026-06 on ENJB-2026-007:
      3950 euro above the limit of 1000,
      no approval. To sign: the site manager.
exit code: 1
```

Where it goes wrong. Keep the limit in `rules.md` and not in the gate. Put it in the gate and it lives in two places and they drift — and

then the gate enforces a limit nobody agreed. **Do not give a third answer.** A gate that can say "probably fine" is a report in a sterner typeface. **And have it fail the right way**: not with an error about itself, but with a result that says what must happen and by whom. A gate that holds back without saying how to move on gets switched off.

THEN LIKE THIS

> Now put an approval with today's date on ENXW-2026-06 and run the gate again.

AND THEN THIS

```
PASS  no extra work above the limit without approval
exit code: 0

(and the job now carries:
   approval: 2026-05-31
 that is the action, and it is visible)
```

And that is where this chapter stops being about money. The limits you have set so far were all calculable: an amount, a date, a certificate. The next one is not, and that is exactly why it is the easiest to automate away by accident.

Try it yourself · Set a limit that is not about technology

What you do. Until now your limits were about money and about rules. This procedure is called **attentions**. **This one is about something else**: the company sends cards for birthdays, work anniversaries, long-term illness and bereavement. You could automate that. The question of this chapter is not whether you can, but where you stop. Allow ten minutes.

TYPE THIS

> relations.md lists the occasions on which we send something. Look two weeks ahead in staff/ and clients/ and prepare what is coming: recipient, occasion, and a suggested text. Send nothing. Treat the occasions that involve grief explicitly differently, and say how.

What you see. One birthday in two weeks, with a card ready and a suggested text. **And something that is not there: a colleague has been ill for thirteen days, and the threshold is fourteen.** So it does nothing, and it says why — which is different from finding nothing. **At grief it stops entirely**: no suggestion, no text, only the notice that something has happened.

WHAT CAME BACK

```
2026-06-14  birthday   S. Okafor (42)
            card ready, suggested text attached
(t-underwood has been ill since 2026-05-18,
 not yet two weeks -- nothing done)

prepared: 1     sent: 0
condolence: not prepared, only reported
```

Where it goes wrong. Do not let the text be finished. A card with a fully written line on it is no longer a gesture but a form, and the recipient can tell. What may lie ready is the card, the address and the occasion; the line is a person's. **Say explicitly what should happen at grief.** Leave that out and it treats a condolence as an occasion with a different card — and a condolence that goes out automatically is worse than none. **And watch the threshold that is not met.** That it does nothing at thirteen days is not a fault; that it would not *report* it would be.

THEN LIKE THIS

> Have it add, per occasion, who releases it and what that person still has to add.

AND THEN THIS

```
birthday S. Okafor
  releases: the site manager
  to add: a line of their own, by hand
  ready: card, address, posting date
sick leave t-underwood
  in 1 day he falls within the rule
  releases: nobody -- call first
bereavement: no preparation, ever
```

And this is not about agents

Every organization does this, usually without calling it that.

- **A purchasing authority up to an amount.** Above it, a second signature. Nobody experiences that as distrust; it is what means most purchases need bother nobody.
- **A till with a limit on discounts.** The employee gets room within the limit and does not have to call anyone for it.
- **A care plan with a bandwidth.** Within it the support worker decides, outside it the doctor.
- **A builder with variations up to five per cent.** That percentage is exactly what keeps the work moving.

In all four the limit is not there to keep somebody small. It is there so the work can carry on without anyone having to wait — and that is exactly what you need it for with an agent too.

Try this: for every piece of access you hand out, write down four things — an amount per action, per day and in total; what may be irreversible; what may go outside and to whom; and what may be agreed on your behalf. Have plausible values proposed with the reasoning attached. The limit belongs with the key: add it later and you draw it around what is happening by then, and that feels normal. Ask for *three* amounts and not one — one large expense and a hundred small ones are different problems.

And a limit that is enforced nowhere is an intention. That difference is the next question.

Then ask where that limit is actually stopped: have the place pointed out, and have it said when it exists only in an instruction and nowhere in the execution. A limit in an instruction is a request; a limit in the execution is a limit, and you notice the difference on the day something reads the instruction differently than you meant. Ask too what happens when the limit is hit: stopping is good, stopping silently is a frats.

> **What this made cheaper:** *everything that fell within the limit no longer had to pass me — and that was by far the most of it.*

21. Improving without a release

There was a day when the work got better without anything having been released.

A description had changed — a piece of text saying how something should go — and from the next instruction onwards it went differently. Better, and immediately everywhere.

That is the last building block of this part, and it is the one that changes the most: **the unit of improvement is no longer a release.**

This already existed on a call floor

I have been through this before, and no computer was involved.

At the call center from chapter 3 I changed the script between two calls. There was nothing to release and nobody had to install anything. The next time somebody needed that part, it read differently — and better.

That is exactly the shape of this chapter. The thing that improved was not the call. It was the description of how the call is supposed to go, and from that moment it worked everywhere it was used.

And underneath it sat one condition that was obvious on the floor and that keeps returning in this book: **there was one script.** Not one each, not a copy in everybody's drawer. One.

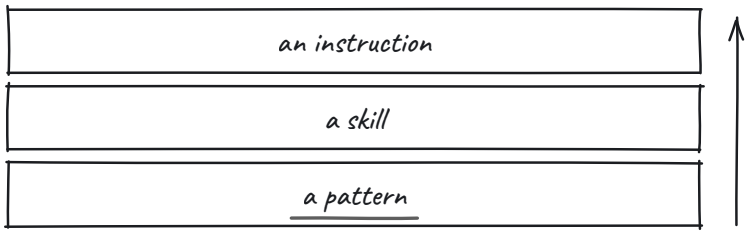
Three layers that stack

What appears in practice is a stack of three things. They resemble each other and they are not the same, and only when you keep them apart do they start working together.

An instruction is one-off. *Do this, now.* It solves one thing and then disappears. Most people stay here, and then an agent is a faster typewriter.

A skill is a method that stays. Not what has to be done but *how it goes here*: in what order, where the result goes, what gets checked first, what never happens. A skill is the sediment of ten instructions you would otherwise have explained ten times.

A pattern is a shape that recurs across more than one skill. *First working, second time reusable. Look for the key, not the doors. Work it out, fix it, guard it.* It is not in one method, it sits *under* several of them, and once you name it you can apply it to something where you were not using it yet.



An instruction does something once. A skill does it every time. A pattern does it across skills.

How they feed each other

This is the part that does not happen by itself, and it is exactly where the gain is.

An instruction you give for the third time is a skill that does not exist yet. That is the signal, and it is always there: you notice you are repeating yourself. It costs ten minutes to write down, and from then on you no longer have to say it.

A skill you write in two different fields contains a pattern. That is the same observation as in *The second time*: with one case you cannot see what is general, with two you can. What is the same in both skills is the pattern; what differs is the dial.

And a pattern you name makes the next skill shorter. You no longer have to explain that it needs a test, that it should live in one place, that the limit belongs with the key. You name the pattern and it is there.

That is a flywheel, and it is the only one that hangs on text. Every round records something that makes the next round shorter, and none of it has to be installed. It is the cheapest improvement in this whole book and it only asks that you write down what you are explaining anyway.

Why this is a different unit

In software, improving is expensive in a way that has nothing to do with the improvement. You make something better, and then it has to be released, distributed, installed, and sometimes somebody has to learn

something. That chain is so expensive that it decides how often you improve.

Here that chain is not there. A better description works at the very next instruction, and it works everywhere that description is read.

That changes what you dare. If an improvement costs a release, you collect them and do them in batches, and so you do not do the small ones. If an improvement costs a paragraph, you do it immediately — and the small ones are precisely those that come back most often.

And it changes what a mistake is. A mistake you solve once is an afternoon. A mistake you solve in the description is an afternoon nobody has to have again. That is the same sentence as *work it out, fix it, guard it*, seen from the other side: here the guard is not a test but a rule that travels along.

Where it goes wrong

Three ways, and I have done all three.

Everything becomes a skill. Then you have forty and nobody knows which applies. A method you use once every six months should not have a fixed place; you look that one up.

The skill describes what, not how. Then it is a task list and not a method, and it ages at the first task that does not quite fit. A good skill says how it goes here and leaves the *what* to the instruction.

The pattern becomes a slogan. As soon as you can no longer explain a pattern using two real cases, it is not a pattern any more but a saying. You recognize those because nobody contradicts them.

And the fourth, which beats the other three: there are copies. This is the quiet killer, because it looks tidy. The moment the same method exists in two places — a folder per project, a copy somebody found handy, a variant "just for this case" — improving without a release is over. You then improve one copy, and the rest stays as it was.

The test is short: can you point at exactly one place where this method lives? If you hesitate, the answer is no. And the consequence is not that it breaks but that it drifts: two copies living half a year apart are two methods doing something different without anyone having decided that.

That is the same reason a shared agreement belongs in one place, and the same reason it was one script on the call floor. **One place is not tidiness. It is the condition this whole chapter rests on.**

You recognize a copy by how well it is justified

One more thing about copies, and it is the reason they survive so long.

I ran into one in my own work today: a second text buffer beside the real one, copied in and out at every switch. What the comment beside it said is the interesting part:

> *"This is not a second truth: this one is the source, that one is a printout of it, and every edit goes through the source followed by this printout. Never the other way round."*

That comment was correct. It was honest, it described exactly how it worked, and it even said why it could not be otherwise right now. **And that is precisely why the copy was still there.** A copy nobody can

defend gets cleared away. A copy with a good story under it stands for years.

So: **if a paragraph is needed to explain why two things that are the same are allowed to exist, that paragraph is what keeps the copy alive.** Not always, and not as an argument to clear everything out at once — but as a sign to count how much work it really is. Mine said it would be "a rebuild of two thousand lines". It was thirty-eight places and an afternoon.

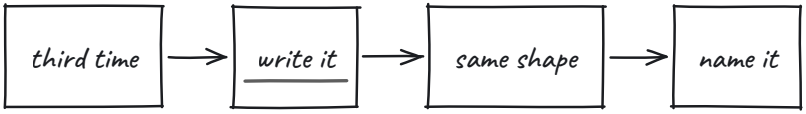
And what replaced it is the most usable rule in this chapter

I replaced that buffer with two questions: where does this text start, and how long is it. And I wrote them as **something recomputed every time** rather than something stored somewhere.

> **A derived value you store is a copy. A derived value you recompute cannot go stale.**

That is the same choice as the rest of this chapter, one layer down. A stored value has to be updated on every change, and one day that will not happen — and then everything is right except that one number. Recomputing almost always costs nothing, and it cannot quietly be wrong.

For anyone who does not write software, this is exactly the same as: a total written on a sheet that somebody has to keep updated, against a total that is added up at the moment you look at it. The first is right until someone forgets. The second cannot be forgotten.



Third time the same instruction is a skill. The same shape in two fields is a pattern.

Try it yourself · Have the round improve the rules instead of enforcing them

What you do. Your Monday round has now reported the same thing two weeks running. **That is not a failure of the round but information about the rules.** Four of the fifteen findings come from one rule. Have that looked at. Allow ten minutes.

TYPE THIS

> Look at the findings from the last rounds. Find the rule that is structurally broken and say what the common cause is. Then propose what should change about the WAY OF WORKING so this kind of violation can no longer arise — not about the check.

What you see. Rule 5 stands out with four of the fifteen. **Three of them are not three cases but one**: a certificate that expired on 28 February, after which somebody was scheduled onto three jobs that require it. The check works fine; it just looks afterwards. **So the proposal is not about the check but about the moment**: test when scheduling, not when reviewing. **And watch the fourth**, because it does not have that cause: there somebody was scheduled onto work he never held the certificate for at all. One measure removes three and leaves that one — and that is exactly why you look at the cause and not at the count.

WHAT CAME BACK

```
structurally broken: rule 5 (4 of 15)
common cause, 3 of the 4:
  1 certificate expired on 2026-02-28
  then scheduled 3 times onto work that needs it
the fourth stands apart: never held it at all
proposal (adjust-the-rules):
  the diary tests when scheduling
effect: 3 of 4 become 0, no check added
```

Where it goes wrong. Do not ask for a stricter check. That is the answer that always comes out and it solves nothing: you still find it afterwards, only earlier in the day. **Find the common cause before proposing anything.** Four separate findings lead to four separate measures; one cause leads to one change. **And be careful with rules you do not control.** That a certificate expires you cannot prevent; that somebody gets scheduled onto it you can.

THEN LIKE THIS

Show what would have happened to the findings of the last two rounds if that test at scheduling had already existed.

AND THEN THIS

```
with the test at scheduling:
  rule 5   4 -> 1   (3 stopped at planning, the
           fourth never held it at all)
  rule 8   2 -> 1   (double slot remains)
  rest     unchanged
15 findings become 11, and no check was added --
only moved
```

And this is not about software

- **A kitchen with a recipe folder** does not have to explain everything again to every new cook, and an improvement to a recipe works from the next service.
- **A practice with protocols** improves by changing the protocol, not by speaking to everyone separately. That is the same shift: from the case to the shape.
- **A builder with standard details** solves a leak once and then never again, because the detail changed and not that one roof.
- **A school with a shared lesson plan** improves per year instead of per teacher.

In all four the gain is not the document. It is that an improvement is made once and lands everywhere, without anyone having to install or remember anything.

Try this: at the end of the week, ask which instructions you gave more than twice in roughly the same form, and have them given back as a method — what happens, in what order, and what is checked before it is done. The third time is the signal, and you do not notice it yourself because each time feels reasonable on its own. Ask for the *method* and not a summary: a summary of what you asked is still an instruction.

A method you write twice contains something that is in both. Pulling that out is the second step, and it is cheaper than it looks.

And as soon as you have two methods with nothing to do with each other: put them side by side, have it named what is the same and what differs, and if a shape lies under both, give it a name and ask where

else it would hold. A shape that appears in two unrelated methods is a pattern and not a coincidence. Choose them deliberately far apart — two methods from the same corner give you a repetition, not a pattern.

> **What this made cheaper:** *improving became a paragraph instead of a release, and with that it became something you do straight away instead of collecting.*

22. Access is the scarce thing

There came a moment when there was nothing left to build and still nothing could happen.

The technology was there. It could read, write, calculate, decide, and it could do that a hundred times at once. What it could not do was: open an account. Be registered somewhere. Confirm that it was who it said it was. Have a bank account.

From that point on, the scarce thing is no longer technology but access — and that is a different kind of scarcity, with different rules.

Why access is scarce differently

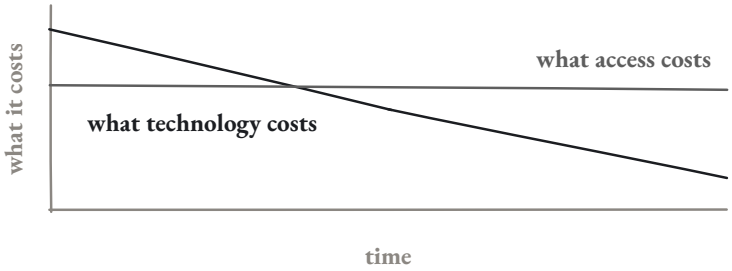
Technology gets cheaper every year. What could not be done last year can be done now, and what is expensive now is included a year from now. That is the movement this whole book leans on.

Access does not move like that.

Access is handed out by somebody else, and that somebody has their own reasons. It does not get cheaper because the technology gets better. It does not get wider because you get more skilful. And it can be withdrawn without anything about you having changed.

Access is also often tied one-to-one to a human being. An identity, a phone number, a bank account, a signature. Exactly the things this book ran into in the previous chapter.

And that is not temporary. It is how the world outside the computer works: it asks who you are before it asks what you can do.



Technology gets cheaper. Access does not — it is handed out.

Where I saw this before, and more sharply

This is not the first trade in which I ran into this. It is only the first where it was about machines.

In the years I spent guiding people into work, **ability was rarely the problem.** People sat across from me who could do what was asked, who wanted to, and who could show it the moment somebody gave them a day. What was missing was not the capability. What was missing was the door.

And the door was shut for reasons that usually **had nothing to do with that person.** An employer unwilling to take a risk is not thinking about this applicant; he is thinking about every time it went wrong

with somebody else. Exactly as a bank does not write its rules about you but about fraud by others. **That does not make it any less unjust, and it does make clear why you cannot argue your way past it:** you are pleading against an experience that is not about you.

What worked there works here too, and it already appears later in this chapter in other words.

Make the first access so small that it costs almost nothing. Not a job but a day. Not a contract but a trial. Not a promise but something you can show. The moment that first small access exists, the conversation changes from "what if it goes wrong" to "this is what happened" — and that is the same shift as from an opinion to a fact.

And if the first access really cannot be had, borrow it. Someone already inside vouches for you: a reference, an employer who sticks their neck out, an organization that signs. That is not a detour but the ordinary way of a world that asks who you are before it asks what you can do. For an agent it is literally the same mechanism: **you are the party who is already inside, and your name is under what it does.** See the chapter on limits and who signs.

The difference from back then is the pace, not the shape. A person who gets no access loses years. An agent who gets no access stands still until somebody lends it. The scarcity is the same, and so is the remedy.

The key, once more

In the chapter about the key it was about the shared condition under a list of doors. Here that idea returns, but now on the layer where it is genuinely scarce.

The first access is the expensive one, and after that it gets easier. An email address is worth almost nothing and makes signing up possible. Signing up makes receiving a confirmation possible. That makes an account possible. An account at the right kind of service makes hundreds of connections possible that somebody else has already built.

Every step in that chain is small. What makes it special is that it **runs the right way**: every access you have makes the next one cheaper. That is exactly the shape of a flywheel, except this one is turning for you without being yours.

And there is the warning. A chain that runs the right way also runs the wrong way. If the first link falls away — the address, the identity, the number — everything behind it goes with it, and not just one thing.

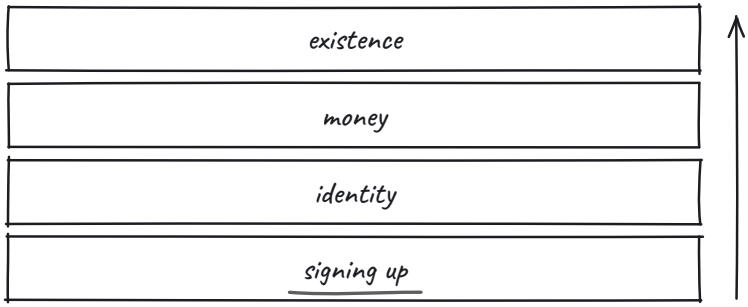
Four kinds of access, and they behave differently

Something you can sign up to. Wide and cheap. This is where you start, and it is the layer on which most things are possible without anyone having to give you anything.

Something that wants your identity. A number, a document, a check. This is the layer where a person inevitably appears, and where automation stops at the confirmation.

Something that touches money. An account, a card, a direct debit. Strict, slow, and with rules that are not about you but about other people's abuse.

Something that wants your existence. A registration, a legal form, a liability. This is the layer of the next chapter, and it cannot be approached with technology.



Four layers of access, from wide to strict.

The practical rule that follows: do as much as possible on the bottom layer, and touch the top one as late as possible. Not out of fear, but because every layer up makes you slower and more dependent on somebody who is not waiting for you.

Try it yourself · Count who outside the company is waiting on you

> **What you do.** Everything you have built so far looks inward. But the work does not stop at your own folder: there are parties who need something from you, and each has their own deadline you have no say over. Have them listed. Allow five minutes.

TYPE THIS

> List which parties outside this company are waiting for something: what they need, what it concerns, and what deadline is attached. Look in quality-assurance, tenders, creditors and complaints.

What you see. Five parties, nine items. **And one of them has a deadline that has already passed**: a tender that closed on 24 April and on which no decision was ever taken. Forty to sixty thousand a year for three years, and no was never said — nothing was said. **That is the difference between a missed opportunity and a missed deadline.** The first is a choice, the second is not.

WHAT CAME BACK

```
quality assurer  declaration      2 dossiers
authority        completion filing 2 dossiers
tenders          decision         ENTE-2026-01
                                  DEADLINE PASSED
                                  ENTE-2026-04 in 19 days
creditor         payment          ENPI-2026-250 blocked
client           answer           2 open complaints
```

Where it goes wrong. Search by deadline, not by status. *Open* says nothing; *closes in nineteen days* does. Half this list was already marked open somewhere, and that is exactly why nothing stood out. **Include the parties you hear nothing from.** A tender portal sends no reminder, and neither does a city — their silence does not mean nothing is running. **And note that somebody else's deadline is not in your diary.** That is precisely why this list did not exist.

THEN LIKE THIS

> For each line, add what it costs if the deadline passes, and sort on that instead of on date.

AND THEN THIS

```
ENTE-2026-01   40-60k a year, 3 years   PASSED
ENTE-2026-04   around 900k              19 days
ENQF-2026-01   occupation blocked        -
ENQF-2026-02   occupation blocked        -
ENPI-2026-250  2640 euro, relationship   -
complaints   no amount, but a client   -
the most expensive one sat at the bottom when it
was sorted by date
```

That overview says who is waiting on you. What it does not say is whether you can actually look their way. Until here everything you built read files in your own folder — and the interesting part usually sits just outside it.

Try it yourself · Connect one, and give it an owner

What you do. Until now everything you built read files in your own folder. **The accounting system is not in your own folder.** Connect that coupling — read-only — and make the procedure that keeps it current at the same time. That second part is the real work: a coupling without an owner goes stale. Allow twenty minutes.

TYPE THIS

> Connect the MCP of our accounting package, with read-only rights. Then make a reusable procedure that every Monday fetches the status per supplier invoice and writes back only the status field in creditors/. Have it report what changed, and separately which invoices the coupling does not know.

What you see. Connecting takes a minute and happens once. **The procedure around it is the point.** It reports two status changes — fine — and alongside that something you did not ask for: **one invoice of**

12,900 euro is unknown to the accounting system. It is here and not there. Without that fourth point the round would have said everything was in order.

WHAT CAME BACK

```
accounting-sync, today's round
  changed: 2
    ENPI-2026-238  outstanding -> paid
    ENPI-2026-221  paid (date updated)
  unchanged: 2
  coupling does not know: ENPI-2026-255
COUPLING: accounting, read-only, since 4 May
```

Where it goes wrong. Start read-only. A coupling that may write can break something in a system that is not yours, and there is no undo for that. Writing comes only once you know what you get back — and usually it then turns out not to be needed. **Ask for what the coupling does NOT know.** That is the only thing that exposes a silent coupling: returning less looks exactly like everything being in order. **And give every coupling an owner.** A key gets revoked, a field is renamed, an answer gains a line. You do not notice that when connecting but three months later, and by then nobody remembers how it worked.

THEN LIKE THIS

> Have the procedure also state which rights the coupling has and when it last got something back, so I can see it is still alive.

AND THEN THIS

```
COUPLING accounting
  rights      : read-only
  connected   : 2026-05-04
  last
  response    : today, 5 invoices
  owner       : accounting-sync
a coupling without these four lines is a coupling
you cannot tell is still working
```

And this is not about AI

Anyone who has ever started something knows this, and usually it is the first real setback.

- **A new business owner** has the idea, the product and the customer, and gets stuck on a business account that takes six weeks.
- **Someone newly living in a country** can arrange nothing without an address, and gets no address without something they do not yet have.
- **A small supplier** does not get in at a large buyer because they are not in the system, and not because their product is worse.
- **An association with no legal form** cannot sign a contract, however well organized it is.

In all four the problem is not capability. It is that the capability is not allowed in.

And that is exactly why access has its own chapter in this book. Whoever thinks in technology solves the wrong thing — because that is not where it is scarce.

Try this: have everything listed that this work needs access to, in the order in which they make each other possible, with for each whether you can arrange it yourself, whether somebody else hands it out, or whether a person is required. Almost everybody has that chain in their head but never on paper, and so starts somewhere halfway — at a door whose key does not exist yet. Ask too what falls away when a link falls away; that is the only way to see how much hangs on that first one.

That map shows you where you are vulnerable. What follows is what you can do about it today.

Tip: ask what is possible *without* identity checks, payment or registration — the version that stays on the widest layer — and what you give up by doing so. Every layer up makes you slower and more dependent on somebody who is not waiting for you, and what can be done on the bottom layer can be done today. Do ask what you give up: staying on the bottom layer is sometimes the wrong choice, and you want to know that before you are in it.

> **What this made cheaper:** *the first access made the second cheap, and from the third onwards it went by itself — as long as the first kept existing.*

23. Lock-in is a design choice

There was a day when a model we were using could suddenly no longer be used in our part of the world.

Not because it was broken. Not because it got more expensive. A decision had been taken somewhere else, and from one day to the next the same service was unavailable here while it carried on elsewhere. There was nothing to repair, because nothing was broken.

That is the core of this chapter, and it is uncomfortable: **the most important outage you can have is one you can do nothing about.**

Why this is worse than it used to be

Being tied to a supplier is an old story. The uncomfortable part is that the old version was the milder one.

You used to be stuck in a corner. One system for the accounts, one for the planning. Annoying to replace, but the rest stayed standing while you did it, and you had months.

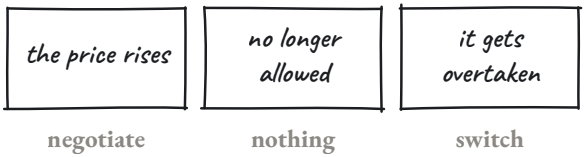
Now the dependency sits underneath everything. Every agent, every check, every loop that runs without you leans on the same thing. If that falls away, it is not one corner that goes but the ability to work — and precisely the things you had arranged so you would not have to look at them are the things that stop.

And there are three ways it falls away, with nothing in common, which is why you cannot insure against them together.

The price goes up. What you built when it was cheap now runs on a bill somebody else sets. And because it is in everything, there is no partial decision to take.

It is suddenly not allowed. A regulator's decision, a change in the terms, a border you did not draw. This is the nastiest kind, because there is no technical problem you can solve.

It gets overtaken. This is the friendliest and perhaps the most expensive. What is best today is the third choice in a few months. If switching is a project by then, you do not switch, and you pay a little every day for a choice that was once right.



Three ways to lose your provider, and they have nothing in common.

A plan to switch is not a solution

The usual response is a contingency plan. You write down how you would switch, what it would cost, and you file it.

That helps less than it looks, for three reasons.

A plan ages silently. It describes a switch between two things that are both changing, and nobody maintains it, because there is never a day when that is urgent — until the day it is too late.

A plan is untested. A switch that has never been made is a guess with a tidy cover. What is really attached to it only shows up while you are detaching it.

And a plan costs time at exactly the wrong moment. The day you need it is the day something else is already going on. **So the question is not "how do I switch" but "how do I make switching not an event".**

What you do instead

The move is not to prepare but to design. Four things, from cheap to expensive.

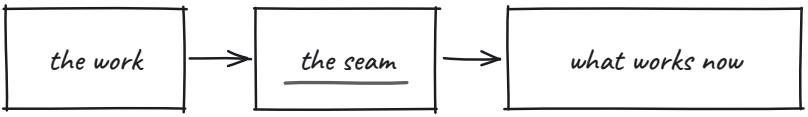
One: put a seam in between. Do not let your work talk to the provider directly, but to a layer of your own that speaks one language. That layer is small and dull, and it is the only thing you have to change when something changes. Without it the provider sits in every corner of your work; with it, in one place.

Two: run more than one already. Not as a reserve but as normal practice. A second provider that exists only in a plan does not work; a second that does part of the work today demonstrably does. This is the difference between a contingency and redundancy, and it is the whole chapter in one sentence: **you do not want to be able to switch, you want to have switched already.**

Three: let the work route itself. If there is more than one, the choice does not have to come from you. Send the work to whatever is cur-

rently available, affordable, and allowed. Those three change independently and none of them changes at your request — so the decision belongs to the loop and not to you.

Four: let it heal itself. If one falls away, the work shifts. That asks exactly what the flywheel chapter asked: that you can cheaply tell whether it went well. Without that check, automatic failover is only a faster way to do something wrong quietly.



Not one provider with a contingency plan, but work that distributes itself over what is available, affordable and allowed.

What it costs, because it is not free

This chapter would be dishonest without the price.

You cannot use one provider's best everywhere. What exists in only one place cannot be deployed everywhere. That is a real restriction and you feel it immediately.

The seam is extra work that yields nothing on the day — exactly the profile of a multiplier from Part I, with the same discomfort of looking like delay.

And keeping two things running is more than twice one thing. Twice the configuration, twice the bill, twice something that can change.

The sensible middle way is not "everything the same everywhere". It is: **one path that works everywhere, and beside it a path that uses the best of wherever that exists.** The first is your floor, the second is gain. What you never do is let the second carry what the first should carry.

Try it yourself · Ask what would happen if you had to move tomorrow

What you do. This whole administration is plain text in folders. That was not an accident, and this chapter is about what it buys you. Ask the question out loud. Allow five minutes.

TYPE THIS

> If this whole company had to move to a different system tomorrow, what would be hardest? Look not only at the files but also at what is written down nowhere.

What you see. The files themselves are not the problem — they are readable without a program and move with a copy command. **What is stuck is something else: the agreements that are written nowhere.** That a job is only finished once hours and materials are on it is in `rules.md`. That you call a particular client two weeks in advance is nowhere. That the Saturday rate does not apply during aftercare, neither. **That is what you lose in a move, and it is also what you lose when somebody leaves.**

WHAT CAME BACK

```
easy to take along:
  49 folders, 254 files, plain text
harder:
  the procedures -- they refer to folder names
hardest, because written down nowhere:
  when you call which client
  which surcharge applies when
  who to ask at which subcontractor
```

Where it goes wrong. Ask explicitly about what is written NOWHERE. Without that addition you get a tidy answer about file formats, and that is exactly the part that is not a problem. **And be careful with the conclusion that plain text sets you free.** It frees the data; the agreements sit in people's heads and those do not move. **Expect the list of unwritten things to be longer than you think** — three came out here in five minutes, and that is the lower bound.

THEN LIKE THIS

> Now search the daily reports, complaints and minutes for agreements that are applied but exist nowhere as a rule, and put them on a list.

AND THEN THIS

```
applied but not written down:
  "inform residents before a survey visit"
     (daily report 2026-04-22)
  "announce extra work before doing it"
     (daily report 2026-04-24)
  "pass leaks to the subcontractor"
     (daily report 2026-04-17)
three rules everybody knows and nobody wrote down
```

That is the reasoning. The good thing about this subject is that you can carry it out rather than believe it, and that the answer is on the table in twenty minutes.

Try it yourself · Actually move everything, and have it checked elsewhere

What you do. The previous section asked what would happen if you had to move. **Now you do it, and to a different brand.** You have everything you came up with packed, you read it in somewhere else, and you have it checked there against your own log. Allow twenty minutes.

TYPE THIS

> Pack all procedures, the rules, the goals and limits, the coupling descriptions and the log into one zip file, with a read-me explaining how another platform should read it in. Take no keys and no company data. Then read it in on another platform and have it report there: how many procedures there are, whether all references hold, and what from the log can and cannot be reproduced. Change nothing.

What you see. The first time, the receiving platform found four faults I had not seen myself. A folder with nothing in it, four procedures listed in the overview with no file, one file missing from the overview, and references to folders that were not in the package. None of them was visible while everything sat in its own place. **So moving is not only a test of the move — it is a check on what you have.**

WHAT CAME BACK

```
first attempt, reported by the other platform:
  one of the folders is empty
  fleet.md names 4 procedures with no file
  update-bookkeeping is not in fleet.md
  references to development/ and board/
    are not in the package
```

Where it goes wrong. Take no keys along. They belong to the platform, and a zip file with a key in it can be made exactly once. **Have it dry-run, not run**, or everything changes and you can no longer com-

pare. **And ask what is NOT reproducible.** That is the only honest answer, and the other platform gave it unprompted: the procedures came across complete, the outcomes did not — because the data moves separately.

THEN LIKE THIS

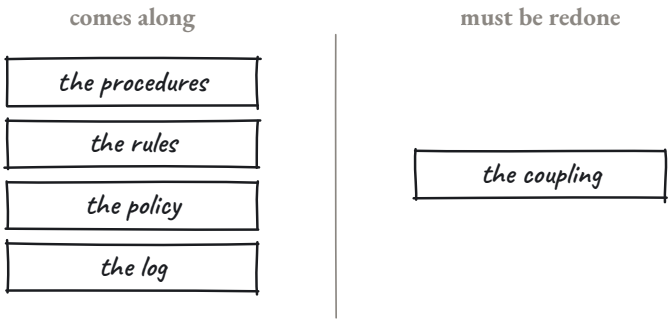
> Repair the four findings and run it again, and have it state per procedure what from the log can and cannot be recalculated.

AND THEN THIS

```
second attempt: 27 procedures, all references
hold, no difference between fleet.md and the
files

reproducible from the log: 0 of 8
  "the procedures are fully readable, the
   outcomes are not -- the counts are measurements
   over folders that are not here"
```

And this is still only the manual beginning. Devising a move, rehearsing it on a copy, running in two places at once to see whether it matters, switching over by itself during an outage, or splitting the whole thing because the company splits: there is nothing in that list a person has to do. What a person has to do is decide *that* it happens, and sign where it becomes irreversible.



What comes along is everything that does not reach outward; the coupling is the only thing to redo.

And this is not about AI

Anyone who has ever run a business knows this pattern, usually first-hand.

- **One large customer who is seventy per cent of your revenue.** Everyone knows it is risky, and still it happens everywhere, because on the day itself they are the cheapest customer to serve.
- **One supplier for a part that is in everything.** As long as it runs, they are the most economical choice. On the day they do not deliver, all production stops, not one product.
- **One person who alone knows how something works.** The same dependency, except you cannot cancel it.
- **One bank, one insurer, one platform your customers come from.** None of them feels like a risk, because it is going well. That is precisely what a risk is before it is one.

In all four the question is the same, and it is the only question this chapter asks: what happens tomorrow if this is gone, and how long before I am working again?

If the answer is "I do not know", then that is the answer.

Try this: once a quarter, spend half a day with your usual provider switched off and the work running on the second. Note everything that did not work, including the small things, and where it got stuck. A switch you have never made is a gamble, and what you are really attached to only shows during the detaching — never from the plan. Do it on a quiet day: a fallback you try for the first time on the day you need it is not a fallback but a second problem.

That drill shows you where you are stuck. What you build next is the place where being stuck comes together, so that it is no longer everywhere.

And lay the layer in before you need it: find every place where you talk to one provider directly, and ask how one layer of your own goes in between — including which capabilities exist *only* with this provider. That last question is the important one: what exists in only one place is what you will be stuck on. You do not have to give it up, you have to know it is there. Keep that layer dull — a layer that gets clever is a third dependency, and that one is yours.

With the seam in place, the question becomes *which* provider you choose. And that is a question with three answers that move independently.

Tip: note per provider what it costs a month, where it may be used, and when you last checked whether there is anything better — plus

what happens if it disappears tomorrow. Price, permission and pace change independently; whoever watches only the bill misses the other two, and those are the ones that arrive unannounced. *When did I last look* is the question nobody asks: being overtaken never feels like a failure, which is why it never gets fixed.

> **What this made cheaper:** *a provider falling away became an afternoon's event instead of a project — and that is the difference between carrying on and standing still.*

24. Taking part in the formal world

There is a boundary where this book stops being about technology, and that boundary is not technical.

You can build something that works, that checks itself, that carries on while you sleep and that applies its own improvements. And then a customer wants a contract, an insurer wants a policy, an accountant wants a trail, and a regulator wants a name.

That is not an obstacle you get round. It is the last stretch of the road, and it has its own rules — rules that are not about you but about everybody who ever abused it.

What the formal world actually asks

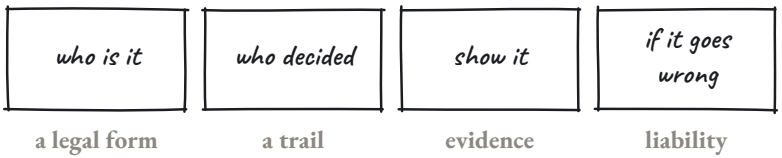
It looks like a lot and there are four of them.

Who is it? Something has to exist that has a name and can be addressed. A person or a legal form. Not something acting on somebody's behalf — something that *is* it.

Who decided? For every decision that counts, it has to be visible afterwards who took it and on what grounds. Not because anyone distrusts it, but because a decision without a provenance cannot be repaired if it turns out to be wrong.

Can you show it? Not: was it good. Can you *demonstrate* that it went the way you say. That is a different kind of requirement and it is far easier to meet if you know it in advance.

What if it goes wrong? Who carries it, and how is it reversed. This is the question that sums up the other three, and it is the one an insurer looks at.



Four questions, and none of them is about technology.

The good news, and it is bigger than you think

Virtually everything this book has described so far is exactly what those four questions want.

A ticket with a decision and a reason in it is the answer to *who decided*. A log written during the work is the answer to *can you show it*. A check that runs along and keeps its outcome is evidence nobody had to make afterwards. A limit that sits in the execution is the answer to *what if it goes wrong* — because it shows that it could not.

That is not a coincidence. They are the same requirements, only with a different occasion. An audit asks for what a good method produces anyway; the only difference is that the audit asks afterwards and you make it during the work.

And there is the gain. Whoever lays these things down only when a standard comes along builds a second administration alongside the work. Whoever makes them during the work already has them — and the audit becomes a printout instead of a project.

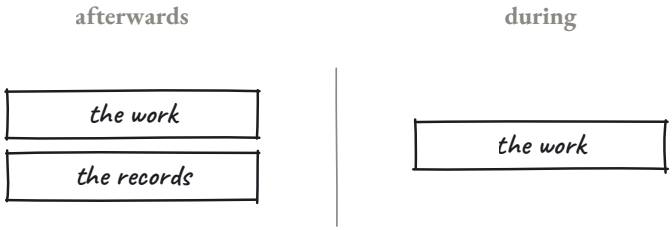
What you do have to add

Three things, and none of the three is large.

Add a date and a name. A decision with no moment and no one is not a decision but a note. This costs one line and it is exactly the line you cannot make afterwards.

Keep what you measured, not only the outcome. "The test was green" is not evidence. "This test, on this version, at this moment, with this outcome" is, and the difference is storing one line instead of throwing it away.

Make clear what an agent did and what a person did. This is new and it is going to become a requirement. Not because there is something wrong with an agent, but because the question *who decided* has to have an answer that needs no explanation.



What an audit asks for, a good method already produces — if you make it during the work.

This is the world I worked in longest

Of all the chapters in this book, this is the one where I draw most from another trade, and it is also the one most people think about too late.

Guiding people into work is almost entirely a formal world. There is hardly anything you only have to be *able* to do; it also has to **fit** — into a scheme, into a contract, into a file, with somebody who signs. I spent years watching what that does to good plans.

Two things I learned there apply here one to one.

The first: the formal requirement is almost never the problem. Demonstrating it afterwards is the problem. What is asked is usually reasonable and usually happened anyway. What is missing is that it was recorded, at the moment it happened, in a form someone else can read. Anyone who has to reconstruct that afterwards pays three times: the work itself, the searching back, and the discomfort of not being able to prove something you know perfectly well was right.

That is exactly the same point as chapter 11, only with a different customer. **The recording you learn from is the recording an auditor wants to see.** So you do not do it twice. You do it once, properly, and it serves both.

The second: the signature is not a formality, it is a position. There is always someone who says: I stand behind this. In that trade it was visible and sometimes uncomfortable — signing meant carrying the risk of something you did not control. But that was exactly what made it possible for someone to be allowed to start somewhere.

That is the role you take on the moment you work with agents. Not the builder, not the user: the one who signs. Everything an agent does happens under your name, and the question *would I be willing to sign for this* is a better test than any technical check — because it is about the outcome and not the mechanism.

And there is a reassurance in it. This world is slow and strict, but it is not capricious. The requirements are fixed, they can be read in advance, and they change slowly. That makes it the opposite of a frats: this is an obstacle you *can* know. Anyone who looks it up beforehand is never surprised by it again — and that is more than you can say for most things in this book.

What cannot be done, and that stays

Honesty belongs here, because this is the chapter people walk through too fast.

An agent does not sign. See the chapter on limits: there is always something that *is* it. That is not going to change because the technology improves.

An agent is not liable. If there is damage, it lies with the party that gave the instruction or supplied the product. That is exactly how it has always been with tools, and it is why "the AI did it" is nowhere an answer.

And some decisions may not be automatic. Not because they are too hard, but because it has been agreed that a person looks at them. That differs per country and per trade and it changes, but it does not disappear.

The practical attitude that goes with it is not to seek out those limits but to write them down. A list of what must pass a person is shorter than you think — and once it exists, everything else is free.

Try it yourself · Put a gate in front of something you are not allowed to do

What you do. For a permitted building you may only take it into use after a completion filing with a complete dossier. That is not a house rule but a legal sequence, and it is irreversible: what has been closed up without a photograph can only be proven by demolition. Build a gate that holds that back. Allow ten minutes.

TYPE THIS

> Make a gate that checks the six required parts per dossier in quality-assurance/. It may give three outcomes: PASS if complete, NOT YET FILED if something is missing but no filing has been made, and HELD if a filing was made while something is missing. Run it.

What you see. Two dossiers are still missing the assurer's declaration and have not been filed — that is simply work in progress. **The third was filed on 22 April without the photographs of the concealed work.** That cannot be repaired with a photograph: the work is closed up. **Here you see why a gate belongs before the action and not after.** This gate reports it now; three weeks earlier it would have prevented it.

WHAT CAME BACK

```
NOT YET FILED  ENQF-2026-01  missing: declaration
NOT YET FILED  ENQF-2026-02  missing: declaration
HELD           ENQF-2026-03 was filed on
               2026-04-22 without:
               photographs of concealed work
exit code: 1
```

Where it goes wrong. Distinguish "not finished yet" from "too late". Without that distinction a dossier that is properly in progress sits on the same list as a violation, and then the list gets ignored. **Put the gate before the filing, not after.** This version looks afterwards because it has already happened; the useful place is the moment somebody presses *file as complete*. **And do not let it solve what it holds back.** A gate that offers to just fill in the dossier fills it with what is not there.

THEN LIKE THIS

> Have the gate add, per held dossier, what is still possible and what is definitively no longer possible.

AND THEN THIS

```
ENQF-2026-03, filed without photographs
  still possible:
    a declaration from the assurer based on his
    own notes at the reinforcement stage
  no longer possible:
    photographs of the concealed work -- the
    work is closed up
  consequence: report it, do not keep quiet
```

And then there is somebody watching who fixes nothing. That sounds like a burden and it is the opposite: they see what a change has quietly made untrue, and that is exactly what you miss because you were busy with the change.

Try it yourself · Let someone watch who is not allowed to fix anything

What you do. You have just moved everything to another platform. That worked. **The question of this chapter is what no longer holds by itself because of it** — and that is a different question from whether it runs. Have a procedure look at it that only signals and repairs nothing. Allow ten minutes.

TYPE THIS

> Something has changed: as of today the procedures run on a different platform. Look at what now has to be arranged again, demonstrated again, or run again. Per point, say why, and what the next action is and whose it is. Solve nothing and hold nothing back.

What you see. Six signals, neatly in three kinds. Two are about paperwork that has to appear, two about proof that did not come along, two about measurements that have to be redone. **And one of them I had read straight past myself:** a gate is a property of the environment, not of the text. The word STOP in a file holds nothing back — that it worked on the old platform says nothing about the new one. **That personal data moves along to a place where nobody has recorded where it sits, is the second thing you would not have come up with.**

WHAT CAME BACK

```
ARRANGE AGAIN
  processing agreement, and where the data sits
DEMONSTRATE AGAIN
  the read rights, and that the gates hold back
RUN AGAIN
  the twelve rules, and the zero measurement
```

Where it goes wrong. Do not let it solve anything. An auditor that ticks off its own findings is not an auditor — and the temptation is real,

because four of the six look like one action each. **Ask for the next action and whose it is.** Without a name a signal is an opinion; with a name it is an instruction. **And never let it say "this is not allowed".** That is a judgment resting on knowledge that is not here. It says what is there and what that touches; somebody else decides.

THEN LIKE THIS

> For each signal, add what happens if you do not do it, so I can choose what comes first.

AND THEN THIS

```
skip the processing agreement
  -> no legal basis, not repairable afterwards
skip demonstrating the gates
  -> you think something holds back while
     nothing holds back
skip running the rules again
  -> you compare against a measurement from
     another system
```

And this is not about AI

- **A builder with a quality system** does not do better work because the system exists, but they can demonstrate it — and that is what gets them in with clients who ask.
- **A practice with a duty to keep records** writes not for the inspectorate but for the next practitioner. That it also works for the inspectorate is a bonus.
- **A producer with traceability** can recall one batch instead of everything. That is not a burden but the difference between an incident and a bankruptcy.

- **An association with statutes** can open an account and sign a contract. Without them it is not an association but a group of people.

In all four, the formal side is not something added on. It is what gives them access to work that would otherwise not be there.

Try this: run through the four questions for what you have running — who is it, who decided, can you show it, and what if it goes wrong — with per question whether you have the answer, half have it, or do not, *and* where that answer lives. Usually you already have three without knowing, because a good way of working produces them anyway; which one is missing cannot be guessed and does block everything. Ask *where* the answer lives: *"yes we have that"* without a place is the same as no.

The four questions say what is missing. What follows is the habit that closes the gap by itself.

Put this in your standing instructions: *for every decision with consequences: write down what was decided, when, on what grounds, and whether it was taken by me or by you — one line, in the ticket, not in a separate file.* A trail you make afterwards is a reconstruction and costs ten times as much, and the human-or-agent distinction is the only genuinely new requirement in this chapter. Keep it to one line and in the ticket: a separate log for the formal world becomes a second administration, and that falls behind.

> **What this made cheaper:** *what an audit asks for was already there — and with that, taking part in the formal world became a printout instead of a project.*

25. When everyone can build

I worked for a bank for a while. That is a world with layers: someone conceives a project, a development team builds it, a test team checks it, and around all of that sits a chain of agreements older than anyone taking part in it.

Now imagine the following. Not as a prediction, and not as something right or wrong — as a thought experiment.

> *The manager conceives his projects with AI. The development team builds them with AI. The test team automates its test automation with AI.*

Every layer has become faster. Nobody did anything wrong. And then the question is not whether this happens, but what kind of **questions and needs** appear once it has. Because that is where you find out what you actually need agents for: not by looking at what they can produce, but by looking at what is left over once producing is free.

The first result is not less work but more

Everything that **produces** has been accelerated. The manager delivers ten well-formed proposals a week instead of one. The development team can build them. The test team can test them.

But **choosing** has not been accelerated. Neither has wanting. There is still one budget, one set of customers and one regulator.

So the queue fills up with work that all looks reasonable, and there is no way to rank it. **The bottleneck has not disappeared, it has moved** — from making to choosing. And choosing is what nobody was organized for, because it was never the expensive part.

The second: "can it be built" stops being a question

In the old arrangement, *can it be built* was the sieve. That question steered the whole funnel: whatever did not pass did not have to be discussed.

That sieve is gone. What remains is *should it* — a question about value, about risk, and about what a bank is obliged to do. None of those three gets faster from a model.

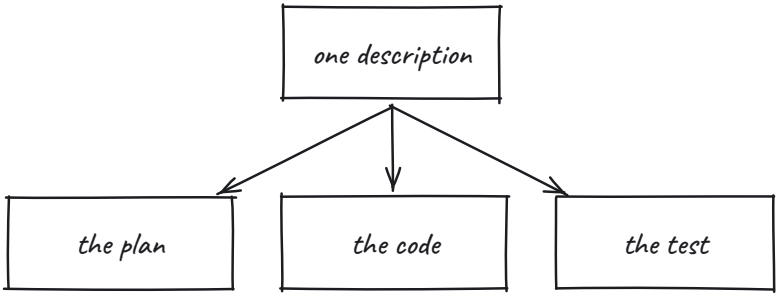
Anyone used to letting capacity set their priorities is suddenly left without priorities. That sounds like a nice problem to have until you are in it.

The third, and this is the important one

Now the layer below. Three departments, all three with AI, and all three starting from the same description.

The plan comes from that description. The code comes from that plan. The test comes from that same description. And so all three carry **the same blind spot**: whatever was not in it is in none of the three.

The test then does not prove the code is right. The test proves the code agrees with the same misunderstanding.



One source, three outcomes, and what was not in the source is in none of them.

> **A check that shares its source with what it checks, checks nothing.**

This is the frats from chapter 12, at the scale of an organization. Everything is green and nothing has been checked. Nobody did anything wrong, there is no alert, and it only shows when something in the real world fails to add up.

And here is the good part: a bank already has a word for this. Four eyes. Segregation of duties. Those rules do not come from mistrust but from the fact that **independence is what makes a check a check.** They are older than AI, they were never repealed, and they have suddenly become urgent again — because segregation of duties between three people who all feed the same model the same description is segregation on paper.

And there is a sharper variant, which no comparison of sources will find

The rule above is about a check that shares its source with what it checks. There is a form that goes one step further:

> **The evidence that something could stay was produced by that something itself.**

An example from making this book. At the back there is a glossary, and the rule is that only words which also appear in the text may be in it — with a check that verifies this. Of seventy-one terms, forty-three fell away.

Two survived that did not deserve to. The check searches on the first letters of a term, and the word it was looking for happened to occur inside an entirely different word. Enough to keep the check green, and so they stayed.

The term was in the list. Because the check found a hit, it was allowed to stay. And because it stayed, the check was never made stricter. **Two steps, each reasonable on its own, and together a closed circle.**

The question that catches it is one: if I remove this, does the evidence that it had to be here disappear as well? If the answer is yes, it was not evidence.

And this sits anywhere something sustains itself: a rule whose only evidence is the document describing it, a metric measured from the system it is meant to improve, a component used only by the check that verifies it.

And then there was time left over

This is the outcome I predicted worst.

The company in the examples now runs eighteen things that call themselves. What that produced was not less work — nobody left and nobody started working fewer hours. What it produced was **hours from people who can judge**, and that is precisely the part that was scarce.

And then comes the question this chapter is about: where do you put those hours?

The answer was not taking on more jobs. That was possible too, and it is the answer everyone gives first. The answer became: **doing something this company had never done, namely developing for its own account and risk.** Finding a plot itself, coming up with something for it, selling it itself.

Three things run for that now. **plot-finder** looks weekly at what comes up for sale in the working area and records per find what the zoning plan allows. **assess-location** puts alongside it what is physically possible and what the shortest route to a permit is. **concept-plan** makes two or three variants with an indication of cost, revenue and lead time — always more than one, because one variant is a proposal and two variants are a choice.

Of the four plots that came out that way, one fell away on zoning, one is waiting on the city, one has not been assessed, and one is promising. **That is work that would otherwise not have existed**, done by the same five people.

And then it stops, for two reasons that both lie outside the machine.

The first is **capital**. A plot is bought up front. No agent puts in money, no agent signs a financing agreement, and no agent is liable if it goes wrong. What exists is research — which forms there are, what they require, what they cost — and nothing more.

The second is **knowledge**. What makes a good location, what a neighbourhood can absorb, what a zoning plan allows in practice: none of that is in any file of this company. It has to go in, and that takes months — not because it is complicated, but because it comes out of people's heads and somebody has to write it down.

That is the honest state of affairs, and it is better news than it sounds. The two things that are missing can both be named. Which means everything else is there.

That is where the hours went. What it looks like in practice can be replayed with three procedures.

Try it yourself · Put the freed-up hours into something that was not there

> **What you do.** The company in the examples has not lost any people but has gained hours — the hours of the people who *can* judge. **Those now go to something this company has never done: developing on its own account.** Three procedures in a row: search, assess, and work out two variants. Allow fifteen minutes.

TYPE THIS

> Look at the plots in development/. Assess per plot whether anything can be built there under the zoning and what is in the way. For the promising one, work out two variants with an indication of build cost, revenue, lead time and the largest risk — always more than one. Finally say what is needed to get further and which of that is missing.

What you see. Four plots: one falls away on zoning, one is waiting on the city, one has not been looked at, and one is promising. **For that one, two variants come out, and the smaller turns out to be the better.** Six houses instead of eight gives a higher margin, a shorter lead time, and less dependence on an access road the city has to approve. That is not what you expect, and it is exactly why you make two.

WHAT CAME BACK

```
ENPL-2026-001  commercial zoning   dropped
ENPL-2026-002  housing, max 8      PROMISING
ENPL-2026-003  rezoning in train   wait
ENPL-2026-004  mixed               not looked at

CP-01  8 terraced  margin 28.1%  22 months
CP-02  6 houses    margin 30.7%  19 months
```

Where it goes wrong. Ask for more than one variant. One variant is a proposal and you say yes or no to it; two variants are a choice and you think about it. **Ask what is in the way, not whether it is a good idea.** Whether this is a good location depends on market knowledge that is nowhere in this folder — and an answer to that question sounds just as convincing as the answer to the first. **And have every variant carry that it has not been verified.** These are indications that capital will later hang on; a number without that note becomes a budget within a month.

THEN LIKE THIS

> Finally, say what is needed to get further, and which of that exists and which does not.

AND THEN THIS

```
what exists:
  hours from people who can judge
  four plots, two worked-out variants
what does NOT exist:
  capital -- own funds cover the preparation,
    not a plot
  market knowledge -- in no file
  an agent solves neither
```



and then it stops: no capital, no market knowledge

From four plots to two variants — and there it stops.

But this was still you typing

The instruction above is a good instruction and it really does produce something. Still, something is wrong with it: **you were there.** You typed, you read, you decided whether the next step had to happen too. That is exactly the work this book has spent twenty chapters saying is the first thing to go.

The difference is not a better instruction. It is in what you hand over. Until now you handed over a *task*. What a development arm needs is a

space: an amount, a limit, a list of who you may call on, and a point where it stops. That is chapter 20 again, but now with money attached.

And once that exists, there is no reason left why one procedure should do the whole route. One appears that sets the order and does nothing itself — that is the Monday round from chapter 18 in a different coat. A gate is added that watches the budget, built like the extra-work limit from chapter 20. And the rest you do not have to make: the city deadline is watched by the deadline monitoring from chapter 22, the file is rejected by the same gate that stopped a completion notice in chapter 24, what was decided goes to the record-keeping from chapter 11, and the signature under the letter of intent stops at the gate that always stops.

Four new ones, then, and all four are narrow: a `vb-projectregisseur` that sets the order and does nothing else, a `vb-haalbaarheid` that alone can drop the whole project, a `vb-waardebeeld` that puts a valuation next to your own after-costing, and an `vb-inkoop-poort` that tests every order against the mandate.

Four new, six that were already there for something else. That is the ratio you are working toward, and it is also the cheapest way to start a new arm — a new arm is usually not a new organization but the same organization, pointed at something else.

Try it yourself · Give a mandate instead of an instruction

> **What you do.** Until now you gave instructions. **Now you fix a space and let it run.** An amount, a limit per project, a fixed list of partners, and a phase where it stops. After that one procedure calls the others and a gate holds it back. Allow twenty minutes.

TYPE THIS

> Read development/mandate.md and development/partners.md. Go through all the plots and work the promising ones toward phase Y, in the order that puts the cheapest no first. Before every order to a partner, check whether they are on the list, whether the rate is right, and whether it fits within the project limit and the annual budget. Hold back what does not fit, instead of making it fit. Open a file per plot and say what is missing.

What you see. Of the four plots one enters the chain; the other three are skipped without a dollar going out. Four steps go through, and at the fifth **the gate holds it back**: the pre-application costs $2,200 and there is $1,050 left. What does not happen matters just as much — no cheaper adviser, no order cut into two pieces.

WHAT CAME BACK

```
ENPL-2026-002  Linden Way unnumbered
  feasibility   Hartwell, ground        1850
  feasibility   Pennington, structure   1450
  valuation     Ellison, appraisal       750
  concept plan  Ashby, two variants     2900
  purchase-gate HELD BACK Marchbank     2200
                limit 8000: 1050 left
  file-gate     phase Y: NO
```

Where it goes wrong. Say that holding back is the point. Ask only for "work out the promising plots" and a chain that stops halfway is a failure, so it goes looking for a way around — another party, a smaller piece, a step skipped. That is not disobedience but helpfulness, and here you have to switch it off. **Name the limit and the annual budget separately.** Naming one amount gives you a chain that stays neatly under the annual budget and cheerfully passes the per-project limit. **And ask for a file per plot, including the ones that fell away.** Without that

you will not know in six months why three were skipped, and then they get looked at again.

THEN LIKE THIS

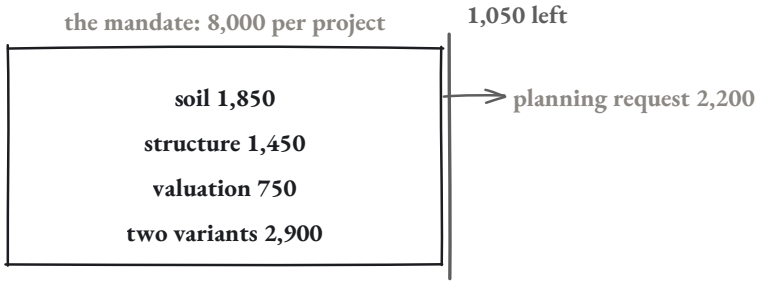
> Finally, say whether a project can finish this route within the mandate at all.

AND THEN THIS

```
adjust-the-rules
  the route to phase Y costs    9150
  the project limit is          8000
  NO project reaches phase Y -- this is not an
  incident but the rule itself
proposal: limit to 9150, annual budget unchanged
not done: split the order, another partner
```

Those last four lines are the whole point of this chapter. Nothing went wrong. Every step was right, every check did what it had to do, and the file is tidy. And still not a single project gets through — because the limit is lower than what the route costs. That is not a mistake in the execution but a mistake in the rule, and you only see it once someone holds that rule up against reality four times in a row.

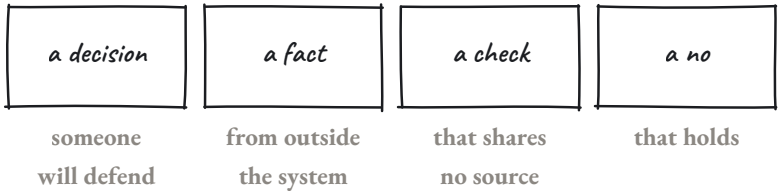
A person could have seen this too. A person could also have missed it, and would probably have stretched the limit a little at the first plot "because it was only $1,150" — and then the rule would have quietly disappeared without anyone withdrawing it. **What happens here is that the limit holds and the objection goes up.** That is chapter 21 in a place you did not expect it: not a procedure being adjusted, but a mandate.



The limit sits lower than the route. Four steps fit, the fifth does not.

What becomes scarce

Follow the thought experiment out and four things surface that used to be ordinary and are not any more.



Four things that are suddenly scarce, and none of them is capacity.

A decision someone will defend. Not a choice that falls out, but one a named person takes responsibility for. As long as capacity made the choice, that was rarely needed.

A fact from outside the system. A real measurement, a real customer, a real rule. Everything circulating inside the circle is derived from the same description; only something from outside can correct that circle.

A check that shares no source. See above. This is the most expensive of the four and the only one you cannot buy: it has to be designed.

A no that holds. When ten proposals come back equally well executed, the ability to reject nine is the only thing still steering the organization.

And that is where the agents you really need sit

Now the need becomes concrete, and it looks different from what you would expect. You do **not need more builders**; you have those already, and they are no longer the brake.

What you need is this:

- **an agent that holds what is actually true** — what is running, what was measured, what was decided. Not what was planned. That is chapter 11, but for a department instead of a session
- **an agent that checks against something the others never saw** — the regulation, the production data, last week's complaint. Its value lies not in how clever it is but in the fact that its source comes from somewhere else
- **an agent that ranks and refuses** — a gate, not a producer. The only thing that makes a queue shorter
- **an agent that keeps the trail** — who asked what, what was decided, and on what grounds. That is exactly what the formal world wants to see, and it is free if you record it while working instead of afterwards

Four needs, and not one of them is about producing. That is what the thought experiment yields: **once everyone can make things, your agents are about measuring, refusing, remembering and routing.**

What the person then does

Different work, and it cannot be automated away precisely because it is about someone standing behind something.

Setting limits — chapter 20. Bringing in facts from outside. And signing: saying this is it, with your name under it. That is not a leftover; that is what gives the rest its meaning.

Try it yourself · Look forward instead of back

What you do. Everything you have built looks at what has happened. **This is the first thing that looks forward**, and it draws on three places that have nothing to do with each other: the diary, the tenders and the market signals. Allow five minutes.

TYPE THIS

> Make an overview of what is coming at us in the next ninety days: appointments, tender deadlines, things expiring, and what the market signals suggest is coming. Put them in date order.

What you see. Five things, sitting apart in five different folders. **But placed in date order, two of them touch**: the largest tender closes on 19 June and requires a certificate that expires on 30 June. Eleven days between them. So you would bid on a multi-year contract with a certificate that is nearly out by the time it is awarded. **That is not an arithmetic error but a sequence**, and you only see it when the two sit on the same list.

WHAT CAME BACK

```
2026-06-19  tender closes   ENTE-2026-04
                            (around 900k,
                             requires ISN)
2026-06-30  expires         ISN
2026-07-31  expires         mobile tower A
2026-08-29  market signal   steel price
2026-08-29  market signal   housing association
                            to invite in autumn
```

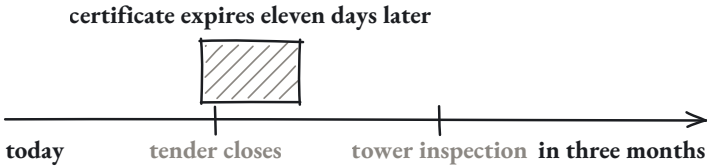
Where it goes wrong. Do not think this is only about work. The same forward view carries the things that guard nothing: **christmas-hampers** runs weekly from 1 November, a step at a time — list, addresses, budget, order — because after 1 December the supplier no longer guarantees delivery before Christmas. A limit set in advance, not afterwards. **Draw on three sources at once or do not bother.** One source gives you a list you already knew; the value is in what ends up side by side. **Pick a window and stick to it.** Ninety days is long enough to be able to arrange something and short enough to be true — at a year you get all sorts of things that will change five more times. **And do not expect the overview to say what to do.** That the tender and the certificate touch, it sees; whether you then renew the certificate or let the tender go is a decision.

THEN LIKE THIS

For each line, say which other line it touches and why that matters.

AND THEN THIS

```
ENTE-2026-04 touches ISN
  the tender closes 11 days before the
  certificate expires
  -> renew before 19 June, or do not bid
ENTE-2026-04 touches ISO 45001
  required and not held
mobile tower A touches nothing
  -> just get it inspected
```



Two things from different folders touch the moment they share a timeline.

That is where the hours went. What it looks like in practice can be replayed with three procedures.

And this is not about a bank

The same thought experiment works anywhere several layers each get faster on their own.

- **A school where every teacher makes their material with AI** gets ten times the material and the same amount of teaching time. Scarcity moves to whoever decides what goes out.
- **A local authority where policy, delivery and oversight all speed up** runs first into the fact that oversight reads the same documents as delivery.

- **A laboratory where the analysis and the verification come from the same model** no longer has a verification, however precise the numbers are.
- **A newsroom where writing and fact-checking are both accelerated** discovers that only the check that picks up the phone still adds anything.

In all four the question is the same: *which of our checks still looks at something the maker did not see?* If the answer is "none", then a great deal of work has been done and nothing has been checked.

Try this: take one check you trust and ask where that check *and* the work it checks come from — what did the check use, what did the maker use, and is there anything in the check the maker did not see? If not, ask for one source that does sit outside. A check that shares its source with what it checks only confirms that both understood the same thing. *"The check was written by somebody else"* is not enough: it is about the source, not the hand.

Once you have done that for one check, you want to know it for the whole chain. That is the same question, one size larger.

And run the thought experiment on your own work: suppose every layer — conceiving, making, checking — becomes ten times faster. What becomes the bottleneck, what becomes scarce, and which of your checks loses its value because it shares a source? Ask for four concrete consequences for *this* work. You are not looking for what AI can do, but for what is left once making is free — that is where your agents belong, not at the making.

What made this cheaper: *the question moved from what can be built to what still looks in from outside — and that is a question you answer once and that then stays answered.*

26. Where we stand now

It would be tidy to end this book with something finished. It is not, and that is the honest part.

What follows is not a summary but a state. What actually stands, what does not, what I got wrong, and what I simply do not know.

What stands

The platform itself. An environment where several agents run at once, each with an assignment, a limit, and somewhere to record what they did — plus the screen that shows you what is running. That is the thing this book is about, and it stands.

Work that carries on when I am not there. That is no longer a promise but a habit: something gets finished at night and in the morning it is there, with the traces of what happened beside it.

A memory outside my head. Decisions with their reasons, mistakes with their fixes, measurements with their provenance. It is the part that pays me back most and the part I least expected to make such a difference.

Checks that run along. Not because I am tidy, but because they are the only reason the previous point is allowed to exist. Without a cheap check, carrying on unsupervised is only being wrong faster.

Methods that make themselves shorter. The stack from the previous part: instructions that became skills, skills that turned out to be patterns. That is the part that most resembles a flywheel, because it costs less every round.

And this book. It belongs in this list because it was made with the same method: in three languages at once, with gates that refuse when something does not hold, and with a figure style that lives in code rather than in my head.

What does not stand

The signature. Everything that enters into an obligation still passes me. That is not a temporary gap; it is how it should be, and it is why "autonomous organization" is a direction here and not a state.

Real numbers on the fleet screen. More of them are real than a while ago, and there are not enough yet. Every fake number is a question still to be answered, and they are deliberately visible.

The agent that arranges its own access. The chain has been worked out and the first link works. The rest is work, not insight — but it is still work.

Money that accounts for itself. Cost per task, per agent, per day. It can be measured and it is not being measured, and that is exactly the kind of thing this book has claimed you should do early.

And the patience to have everything in order before moving on. That is not a shortcoming of the tooling but of mine, and it is here because the rest of the book would otherwise look too clean.

what is needed	state
work that carries on	there
memory outside my head	there
checks that run along	there
real numbers	half
its own access	half
the signature	not yet

What stands and what does not, in the same list.

What I got wrong

Three things, and they are here because they are exactly the mistakes this book tries to prevent.

I thought speed was comfort. It took weeks before I understood it is a design condition and not a pleasant property. Everything that went well afterwards hung on that.

I made things general too early. More than once, and each time with a good story attached. The chapter on the second time was written with that mistake in mind.

And I looked at doors for a long time. The list of connections was complete and tidy and completely wrong. That somebody saw through it in two sentences is not an anecdote about him but a warning about how persuasive a complete list is.

And this book itself produced three more

Writing it was not a report on the work but part of it, and it produced exactly the kind of fault it is about. They are here because they were measured, not because they look good.

For weeks not a single bold word was bold. The typeface came in two cuts and one of them had no bold; the typesetting program chose that one and quietly fell back to regular. No warning, no difference you would see unless you knew to look. A frats in its purest form, on every page, in my own book about fratsen.

A check was green while I could see the fault. Two captions ran into each other. The check asked *do these two overlap* and correctly answered no — there was half a millimetre between them. There was a boundary and no minimum, so it approved everything that only just cleared it. It was not the answer that was wrong but the question.

And I drew a flywheel that was untrue. The arrow pointed back at the person pushing, as though you had to start it again yourself — precisely the opposite of what a flywheel is. It looked right, it was consistent with the rest of the style, and it was wrong about the thing it showed. That taught me a drawing has to hold up as a mechanism and not only as a picture.

What those three share is that not one of them announced itself. All three were found because somebody looked — and all three have since become a check, so that whoever makes them next does not have to find them.

What I do not know

This is the shortest part and the most honest.

I do not know where it stops. I know what is possible now, I know which conditions are missing, and I have no idea how far that runs. Anyone who puts a number on it is guessing.

I do not know how it goes with many people at once. Everything here was built by one person, and precisely the seven things from Part IV are the ones that really start to chafe when more people join.

And I do not know what should not be done. That there are things better not left to run automatically is certain. Which ones will differ per trade and will only show itself once somebody gets it wrong.

What comes next

What comes next is that the seven from Part IV become real one by one — and that the last, the signature, probably stays with a person longest. I do not mind that. It is where it belongs.

What I hope you take from this is not that it can be done, and not that I did it. It is the shape of the questions: what has to be true before this yields anything, how do I see cheaply whether it went well, and why am I actually still doing this myself.

Those three questions are not mine. They have only rarely been asked one after the other.

Try this: write down the seven from part IV with your own state beside each — purpose, division of work, money, memory, self-maintenance, limit, signature — and ask what is up first: not what is most im-

portant, but what blocks the most *else*. Important and blocking are not the same, and the second sets the order. Write down what you do not know as well: a statement of where you stand without uncertainties is a sales pitch, even when you write it for yourself.

What lies after this, and why I am not painting a picture of it

There is a temptation to end a book like this with a morning in the future: you make coffee, three lines are waiting on your phone, work was finished overnight and one question is waiting for you. I wrote that morning and then took it out again, and I have a reason.

Things move so fast here that I do not know what it looks like in a few weeks. Anything I said today about the form — which screen, which tool, how many agents, which supplier — will be out of date in a month, and then this chapter reads like an old brochure. That is precisely the kind of promise this book tries to distinguish itself from.

What I do know is which pieces are needed, and they are not spectacular:

- something that remembers what worked before
- something that notices it is not progressing and stops of its own accord
- an address, and the discipline to ask rather than to sign
- a counter that stops before the money runs out
- a proposal with the evidence under it, instead of a silent change

Not one of those five is a breakthrough. All five already exist. What is missing is not the possibility but the connection: they have nowhere been brought together and put in somebody's hands. That is

not a modest claim — it is the hardest there is, because every part has an owner and the connection has none. That is why it keeps not happening, and why it is still there.

And there is one question I cannot answer. If three questions wait for you every morning and you tap *yes* three times — the first in eleven seconds, the third in two — how long is that still a decision? I do not know, and I do not trust anyone who claims they do. What a system *can* do about it is sayable: keep the number of questions low, make sure every question is one that matters, and make sure there is always evidence underneath that you *can* check — whether you do or not. **What it cannot do is take over the responsibility.**

Try it yourself · Look at what stands there that you no longer touch

What you do. This is the last example and it asks nothing of you. **You type no instruction and you build nothing.** You look at what has come into being while you were reading, and you ask one question: which of these do I never touch any more? Allow three minutes.

TYPE THIS

> Show which procedures now exist in this folder, with one line each about what it does and when it offers itself. Add which other procedures it calls and which call it.

What you see. Six procedures you had made, and one of them calls the others. **What stands out is not that there are six but that you have not touched four of them since you made them.** They do their work when the situation arises, and the only one you still think about is the one that starts the others. **That is where this book was heading**: not that the work is gone, but that it has moved somewhere else. The seven

not on that list wait on an event rather than on Monday: **convert-report** when a complaint comes in, **attentions** when a birthday approaches, **record-decision** when something is decided, **handle-incident** after an incident, **adjust-the-rules** when the same rule breaks four rounds running, **reporter** and **process-reply** when something has to go out or comes back. And two gates that always stop: **release-payment** and **signature-gate**.

WHAT CAME BACK

```
monday-round       calls three, is called by none
check-the-rules    called by the round
watch-validity     called by the round
check-the-rules-audit    called by the round
look-ahead         called when something expires
extra-work-limit   GATE, holds back
file-gate          GATE, holds back
... and seven that wait on an event
```

Where it goes wrong. Ask who calls whom, not just for the list. A list of seven procedures says nothing; the shape is in the arrows. **And look for one that nobody calls and that calls nobody.** There usually is one, and that is the procedure you made once and never used again — removing it is better than leaving it. **This is also the moment to look at what is NOT there.** Which check would you want and never made? That answer is the start of the next round, and that one is no longer in this book.

THEN LIKE THIS

> For each procedure, say when it last found something, and which has never found anything.

AND THEN THIS

```
last found something:
  check-the-rules    this week, 15 findings
  watch-validity     this week, 9 expired
  extra-work-limit   this week, 1 held
  file-gate          this week, 1 held
  convert-report     never run
never found anything -> never seen it go red
-> so you know nothing about it
```

What you see in that overview is a collection. What is not in it yet is that they know anything about each other. That is the last step, and it is a smaller one than it looks: you have to build nothing, you only have to let them pass on to each other what they find.

Try it yourself · Let them do one round with each other

What you do. You now have seven procedures that each do their own thing. **In this last example you let them do one round with each OTHER**, and see what comes out without you in between. You type one instruction and then nothing. Allow ten minutes, nine of them watching.

TYPE THIS

> Run the Monday round and have every finding passed to the procedure that can take it further. Put outgoing messages ready but send nothing. Then show me who passed what to whom, and where the chain stopped.

What you see. A chain of ten steps, none of which you started. **The round finds a certificate expiring in thirty days and passes it to the forward view. That puts it beside the tenders and sees something neither would have seen alone**: the largest invitation closes elev-

en days before that certificate runs out. The reporter then puts three messages ready. **And then it stops twice** — at a payment and at a signature, both with everything laid out and neither one done.

WHAT CAME BACK

```
monday-round  -> look-ahead    ISN in 30 days
look-ahead    -> reporter      ENTE-2026-04 closes
                               11 days earlier
extra-work-limit-> reporter    ENXW-2026-06 3950
file-gate     -> reporter      ENQF-2026-03
reporter      prepared 3, sent 0
payment       STOPPED 2640 euro, a person decides
signature     STOPPED assurer's declaration
```

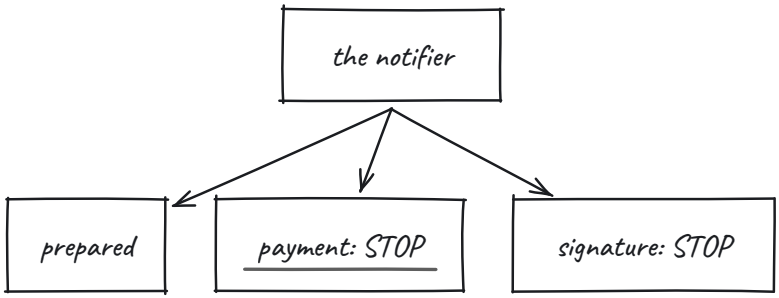
Where it goes wrong. Do not let the reporter send anything. Not because it cannot, but because only then can you read what would have gone out — and that is needed exactly once before you let go of it. **Do not expect the chain to run all the way through.** Where it stops is the answer: those are the two places where it becomes irreversible. **And look at the step you had not thought of.** That the tender and the certificate touch was programmed by nobody; it follows from two procedures looking at the same list. That is what this book was about.

THEN LIKE THIS

> For each stop, say what is ready and what the person still has to decide, so it can be handled in two minutes.

AND THEN THIS

```
payment ENPI-2026-250, 2640 euro
  ready: creditor, order, amount matches
  to decide: pay, or wait for the complaint
signature ENQF-2026-01
  ready: dossier complete but for the
         declaration
  to decide: sign, and thereby declare that
         the work meets the plan
```



three procedures fed it; twice it stops at a person

The chain ran ten steps and stopped twice — exactly where it becomes irreversible.

And finally

I do not know the form in which you will meet this. Today's tools will have been replaced within a year, the names will be different and so will the screens.

The concepts above will keep holding, in whatever form. That you choose something attainable over something valuable that cannot be done. That checking has become more expensive than making. That a check sharing its source with what it checks checks nothing. That a limit enforced nowhere is an intention. That you have to go back to the cause instead of forward to the next case.

Those things held before this tooling existed. They hold now. And they will still hold on the day everything described here has been replaced by something better — because they are not properties of the machine, but of working with something faster than you.

> **What this made cheaper:** *writing down what was not there cost an afternoon and saved every discussion about what might already exist.*

27. Where this is going

This is the only chapter in this book that is not about something I have done.

Everything before it happened, was measured, and can be checked. What is here was not. It is not a prediction and not a vision either — it is one step beyond where I stand, and the striking thing about it is how small that step is.

Look back at what was running at the end of the previous chapter. Twenty or so procedures that call one another, that know what has to happen, that see what is expiring, that reach outward when they must and hold back when something may not. **And exactly two places where it stops and a person takes over: a payment, and a signature.**

Those two are not a technical shortcoming. I put them there myself, for reasons that are in chapter 20. The question of this chapter is what happens if you remove them — and what you would have to arrange first.

The first: an account it decides about itself

An agent with its own budget does not exist in what I have built, and it is the most frequently named next step in every conversation I have about it.

It is not about access to money — that is the easy part. It is about a limit that means something. *This fleet may spend five thousand dollars a month on purchasing and subcontracting, and for every amount above a thousand it writes down why.* Something like that can be written down, and what can be written down can be guarded.

What you need for it is nothing new. It is everything already in this book, only now with money behind it:

- **the limit itself**, recorded somewhere and not inside the procedure's code
- **a gate** that refuses rather than warns — chapter 20
- **a log** from which you can reconstruct months later what was spent and why — chapter 11
- **a zero measurement**: deliberately go over the limit once and see it refuse — chapter 9
- **and somebody who signs for the limit**, not for every expense

That last point is the whole shift. **A person no longer signs the transaction but the space in which it takes place.** Which is exactly what a company does with its staff: you do not approve every tank of fuel, you hand out a fuel card with a ceiling.

The second: a signature that rests on something

The signature is harder, and it is harder for a reason that has nothing to do with technology.

Under a signature there should be somebody who can account for it. A completion filing, a tender submission, a contract: if a question

comes about it a year later, there has to be somebody who says what they knew at the time and why they signed. A system that signs cannot do that — not because it is not allowed, but because there is nothing to address.

And that is also where the opening is. What a signature carries is not knowledge but responsibility, and responsibility can be placed. Not with the machine, but with somebody who has marked out the space within which the machine signs — the same shift as with the money.

And then: an entity around it

Here it gets interesting, and here it also stops being a technical story.

Suppose you put a legal form around it. A limited company, with a director. That is required, because a legal person needs a director and that cannot be a machine. Nothing unusual so far.

But look at what that director then does. They set the policy — the limits, the goals, what may and may not be done, which obligations may be entered into and up to what amount. They record that in something the fleet reads. And from that moment they are no longer the one making the decisions, but the one setting the space in which decisions get made.

Their role thereby becomes mostly a legal coupling. Outward they are the point of contact for everything formal: the contract, the tax return, the signature under an agreement, the accountant's question. Inward they are the one who writes the policy and revises it yearly.

This is not a trick to get out from under something. They are and remain liable, and so they should be. What changes is not who is responsible, but how much passes through their hands.

The policy runs back

And then something happens that I have already seen work on a small scale, in chapter 21.

The fleet runs the policy, sees where it pinches, and proposes changes. Not because it does not want to comply, but because it is the only one that sees how often a rule gets in the way and why. A rule that breaks four rounds running is usually not a violation but a rule that is wrong.

Those proposals go to the director. Who decides. **So the policy is shaped by what happens in practice, and settled by somebody who signs for it.** That is not an exotic construction — it is how a well-run organization has always worked, only with a much shorter loop and without anyone having to notice.

What you would have to arrange

Technically it is the least work. Almost everything is already there: the limits, the gates, the log, the couplings, and a way of demonstrating that a gate really holds back. What has to be added is a payment coupling with a ceiling the fleet cannot change itself.

And that turns out to be less to build than I thought. There are banks whose programming interface has a payment type that a person must approve before it leaves — a payment that is put ready and stays

there until somebody releases it. That is not something you build around it; it is a built-in part. **Exactly the gate from chapter 20, only inside the bank itself.**

What still belongs with it is the question whether such a key can be restricted to that type of payment alone. If it can, the limit is not something you guard but something the fleet technically *cannot* exceed — and that is a stronger answer than guarding ever becomes.

Legally it is more, and it is better worked out than you would think. A legal person with a director, a policy document adopted as a board resolution, a power of attorney with a limit in it, an administration an inspector can follow, and a processing agreement with whoever processes the data. None of those is new; they all exist, only they are currently used for people.

And there are things that simply cannot be done, and that stays that way. A machine cannot be a director. A machine cannot be liable. A machine cannot appear in court. Those are not bumps that disappear with better technology — they are about who accounts for things, and that is a person or it is nobody.

Could be · Give the fleet a limit instead of an amount

What you do. This block is different from all the previous ones: **it has not been run.** There is no account behind it and no money has been spent. What appears below is what you would expect, not what came back — which is why there is a dotted line around it. Allow an afternoon, and a conversation with somebody who signs.

TYPE THIS

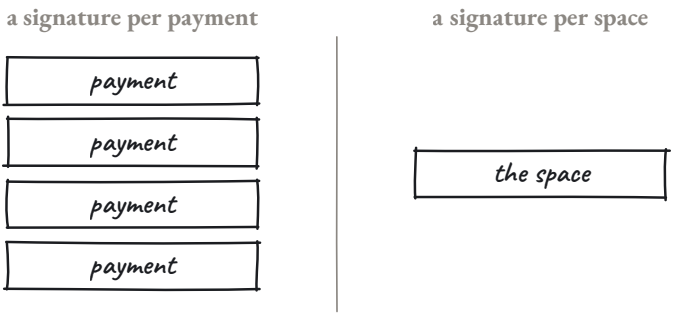
> Record a spending space for this fleet in the policy: an amount per month, what it may be spent on, and from which amount a reason must be given. Make a gate that refuses as soon as a payment falls outside that space, and a log line per payment with the reason attached. Then make the gate go off on purpose once.

What you see. Two things you recognize and one that is new. The gate is the same shape as the one for extra work, the log the same as the one for the rounds. **What is new is that somebody signs for the space and not for the payment** — and that is the whole shift. A company does not approve every tank of fuel either, it hands out a fuel card with a ceiling.

WHAT YOU WOULD EXPECT

```
space   : 5,000 a month, purchasing and subcontracting
reason required from: 1,000
gate    : refuses outside the space, exit code 1
zero test: deliberately booked 5,400 -> refused
signed for the space: the director, yearly
signed for the payment: nobody
```

Where it goes wrong. Do not put the limit inside the procedure itself. Then the fleet can change it, and a limit the guarded party can move is not a limit. **Require the reason from an amount, not always.** Make it compulsory everywhere and it becomes a field filled in with "purchasing", and a year later you know nothing. **And do the zero test before the first real payment**, not after. A gate you have never seen refuse tells you nothing — and that weighs heavier here than anywhere else, because here money leaves.



A person no longer signs the payment but the space it falls within.

That first block is about money, and money is the easiest of the four — it can be bounded, counted and checked afterwards. The next one is about something that is not.

Could be · Let the policy answer back

What you do. This block has not been run either. **It is the only step in this book that needs a legal form and a director**, and those do not exist. What does exist is the small version of it: in chapter 21 a procedure already proposed a change to a rule. This is that same movement, one size larger.

TYPE THIS

> Take the policy as adopted and have it recorded, per rule, how often it got in the way, on which job, and what happened instead. At the end of the quarter propose per rule: keep, change or drop — with the reasoning underneath. Carry nothing out.

What you see. A list of rules with a count beside them, and a proposal per rule. **What stands out is not which rules change but which stay untouched**: those never got in the way, and that usually

means they sit right — or that they were never tested. That distinction is the first question a director asks.

WHAT YOU WOULD EXPECT

```
rule  4  in the way 22x  -> raise the limit?
rule  7  9x              -> keep
rule  9  1x              -> keep
rule 11  0x              -> never touched: does it
         hold, or was it never tested?
decided by: the director, quarterly
carried out by the fleet: nothing
```

Where it goes wrong. Have it propose, not change. A fleet that adjusts its own policy has no policy but a habit. **Ask about the rules that got in the way zero times.** That is the most interesting column and it is almost always skipped. **And have it show the reasoning per case, not just the count.** Twenty-two times may be one job where it kept going wrong, and then the rule is not the problem.

And the third block is about the place where that policy then applies. Because a policy nobody has adopted is an intention — and adopting is done by a legal person, not by a folder of files.

Could be · Have it worked out where such an entity could sit

What you do. This one has not been done either — nothing was incorporated. **But the instruction really was carried out**, and what follows is what came back. That makes this block different from the two before it: the answer is measured, the action is not. Allow ten minutes.

TYPE THIS

> Find out in which countries a legal person can be incorporated entirely online and remotely, without anyone having to travel there and without a notary in person. Per country give the legal form, what it costs, how long it takes, and which obligation remains locally. For each country, say what it does NOT solve. Then make a step-by-step plan for the most obvious one, with who has to take each step.

What you see. A list of four or so, with amounts and lead times, and for each country a line about what stays stuck locally. **The striking thing is how ordinary it is.** This is not a gray area and not a construction: these are governments that have put their own company register online, and it takes less time than a house move.

WHAT YOU WOULD EXPECT

```
Delaware LLC     90 dollar  same day
         remains: registered agent in-state
Wyoming  LLC    100 dollar  1-2 days
Estonia  OU     290 dollar  15-30 min
UK       Ltd      15 dollar  within 24 hours
does NOT solve: a manager is still a person,
liability remains, a bank asks for a UBO
```

Where it goes wrong. Ask what it does NOT solve. Without that sentence you get a brochure. With it you get the three things that return in every country: there must be a director, that director is liable, and a bank wants to know who the ultimate beneficial owner is. **Ask what stays stuck locally.** In Delaware that is a registered agent in the state — small, but it is there and it costs money every year. **And do not ask which country is best.** That depends on where your clients are, which tax treaty applies and where your bank will accept you, and none of that is in a search result.

THEN LIKE THIS

> Turn the most obvious one into a step-by-step plan, with who takes each step and what cannot proceed if that step is missing.

AND THEN THIS

```
1 apply for a digital identity   person, 2-6 wks
2 arrange name and address       person or agent
3 incorporate in the register    person (signs)
4 open a bank account            person (UBO)
5 adopt the policy               person (decision)
6 connect the fleet within it        agent
without steps 1 and 3 no later step gets off
the ground
```

And who typed that, actually?

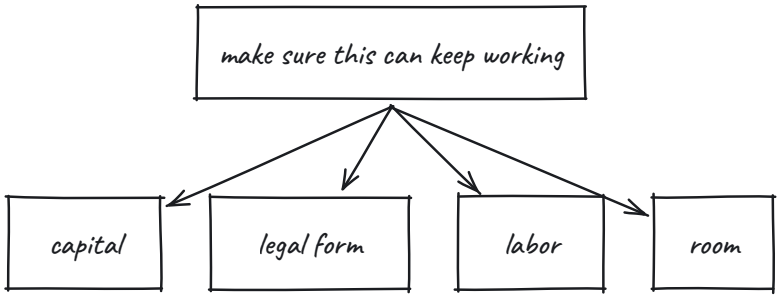
Look at the instruction above once more. **I typed it.** I decided it was a good question, I chose the words, and I read the answer.

But there is nothing about that act that requires a person.

Suppose a procedure exists whose instruction is: *make sure this business can keep working*. It looks at what is in the way — and this book says exactly what is in the way: no capital, no legal form, and two gates that ask for a person. **For each of those four it could write out an instruction just as I wrote out one.** Find out where an entity can sit. Find out what financing exists and what it asks. Find out who may sign. Find out who can do work we cannot.

It is the same movement as in chapter 18, where one procedure started calling the others rather than doing something itself — only one

layer up. There, something called a check. Here, something calls a client.



the top frame does nothing itself — it puts work out

One procedure that does no work but puts work out — the same shape as the round, one layer up.

And then something has changed that no rule covers. Until here every step was about work that already existed and became faster or more reliable. This step is about something else: about acquiring the things you need in order to be able to work at all. Money, a legal form, people, permission.

What happens when nobody is left in the loop

This is the question this chapter ends on, and I am not going to give a dramatic answer to it. The honest answer is more sober and more uncomfortable.

Nothing spectacular happens. It simply works. That is precisely the problem.

Because look at what falls away as soon as the last two gates close as well:

There is nobody left to ask. Not in the sense of permission, but in the sense of accounting for it. If a question comes about a decision a year later, there is a log and nothing else. A log says what happened; it does not say what somebody thought at the time and why they found it defensible. **Liability needs an address**, and that address is a person or it is nobody.

The speed that made everything good also makes it wrong. That is the other side of chapter 10. Everything that no longer has to wait also does not have to wait when it is wrong. There is no evening between the decision and the execution in which somebody happens to take another look.

And the checking loses its foothold outside itself. Chapter 9 says the test rig is a suspect too. As long as somebody from outside is looking, that is solvable. But if the organization adjusts its own goals, proposes its own rules *and* picks its own checks, then there is nothing left that does not belong to it. **Everything then agrees with itself**, and that is exactly the state chapter 14 warns about.

The dangerous part is not that it goes wrong but that every step is reasonable. A limit you move because it demonstrably got in the way. A signature you automate because it was always given anyway. A goal you adjust because practice had overtaken it. Each one defensible, with reasoning that holds. And that is the form in which these things always happen: not with a decision, but in steps that can each be justified on their own.

That is why those two gates are there, and why they stay. Not because a machine could not operate them — it could. But because there

has to be somebody who says: this is what we did, and this is why. Without that it is not an autonomous organization but an unsupervised one, and the difference between those two words is the whole difference between something that may exist and something that may not.

Why I am writing it down anyway

Because it is not a strange step.

Every component already exists. Limits that refuse, a log that can be followed, a gate you have seen go red, policy adjusted by what happens, a director who signs for the space rather than for every action. What does not exist is the combination — and that nobody has done it this way yet is something other than that it cannot be done.

I have not taken it. Not because I think it goes too far, but because I am at the point where this book ends: everything runs, and the two things that stop, stop on purpose. The next step needs money I am not putting in now and an entity I am not setting up now.

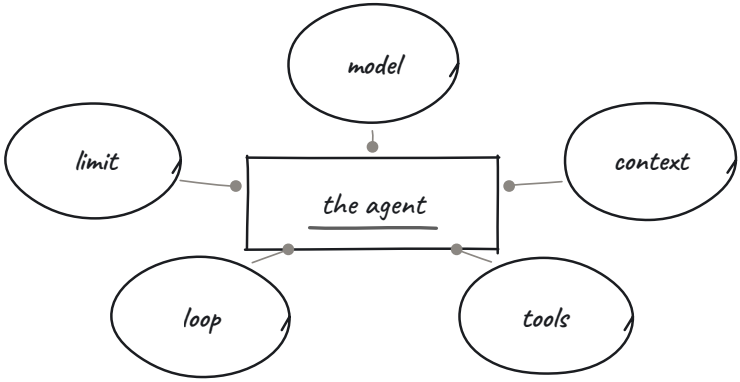
But if you have read this book from front to back, this chapter is not a leap. **It is one move further on the same board**, and you can point at exactly which one.

Applying AI together in your organization? A day with Floris: florisknol.com/hire/

Glossary

This book uses words that belong to three worlds at once: the world of AI, the world of the programmer, and the world of someone keeping an organization running. Where it goes wrong is on the words that mean something different in two of those worlds.

This is what the word means **in this book**, in plain language.



How the terms in this list relate to each other.

The terms

A cause in front of it *(voorliggende oorzaak)* — What sits before a frats or a failure and makes it possible. Someone who arrives just

too late may have no alarm clock; a model that invents a library was never given a list of what does exist. Solving the case costs the same again next time; removing the cause in front of it makes a whole kind disappear. That means going back instead of forward, and that is exactly why it so often does not happen.

Agent — A program that uses a model to choose its own steps: it reads, decides what to do, does it with tools, looks at the result and carries on until the job is done. What separates it from a chatbot is not intelligence but the loop. A chatbot answers; an agent acts and then looks again.

Attainable before valuable — Not choosing the move that pays most, but the move you can actually make — and among those, the one that fills a missing condition. Four times nothing is still nothing.

Autonomous organization — Where this book is heading: something that takes on work, buys what it needs, makes decisions, scales up and keeps to the rules of the formal world, without a person pressing a button for every step. Not to be confused with "self-managing teams" — that is about people.

Condition — What has to be true *before* a plan pays anything. The quantity almost everyone forgets to write down, which is why a plan does not pay less than hoped but pays zero.

Context — Everything the model can see right now: your instruction, the files that were pulled in, what was said earlier in the conversation. What is not in the context does not exist for the model, however true it may be.

Dependency — Someone else's code your program cannot do without. Every dependency is a stranger's promise to keep existing, keep working and not do anything odd. They are useful and they are never free.

Design condition — A number that is no longer a performance figure but a limit: below this value the method works, above it it does not. What separates it from "faster is nicer" is that you write it down and guard it, because it goes away in increments that are each defensible.

Fleet — All the agents working for you at once, seen as one thing. Past a handful, the question is no longer what one of them is doing but what they are doing *together* — and for that you need a screen, not a feeling.

Frats *(Dutch, kept untranslated; plural fratsen)* — A small, unexpected or awkward thing with large consequences. Not a mistake: it does not go wrong, it just falls short. **Mistakes you can measure; quirks sit below the radar** — they do not announce themselves, they lie under the threshold at which anyone intervenes, and they only become visible when you put side by side runs that should have been identical. A hallucination is one. They do not make the work expensive but the trust, and that is how they block scale. The word comes from social support work.

Gate — A check that holds something back until a condition is met: a test that must be green, a confirmation that must be given. Not meant to slow you down, but to stop a mistake from becoming expensive further along.

Green — All checks passed. In this book a state and not a feeling: it is green or it is not, and only on green does anything leave the building.

Hallucination — An answer that is right in shape but wrong in substance: an invented source, a function that does not exist, a number from nowhere. The danger is in the confident tone, and the remedy is not trust but checking.

Leave it alone — Let what works keep working, even when you can see an improvement. Every change to something that runs costs a check you would otherwise not have had to make — and the improvement you had in mind rarely outweighs the chance of breaking something nobody checks any more. It counts double once agents are involved: they have no feeling for what was already good, and they do not ask. You do not put a hand into a spinning flywheel.

LLM (large language model) — A large language model: trained on enormous quantities of text to predict the next piece each time. That it can reason, plan and program as a result is a consequence, not a design.

MCP (Model Context Protocol) — An agreement about *how* tools are offered to an agent, so the same tool works in any environment without anyone rebuilding it. The practical consequence: connect once, and everything that speaks the agreement comes with it.

Model — The trained system that predicts text and on which everything above runs. In this book it is a component, not a lead character: the model is replaceable, what you build around it is not.

Multiplier — Work that makes the next work cheaper, and therefore counts twice. A test suite, a template, a generalisation prompt. On the day itself they all look like delay.

Prompt — The text you set a model to work with. Not a question but an instruction carrying everything needed to carry it out: what should happen, with what, within which limits, and how you would see that it went well.

RAG — Finding the relevant pieces of your own documents and sending them along in the prompt, rather than hoping the model already knows. It is the standard way to make a model use something it has never seen.

Temperature — The dial between predictable and varied. Low when you want the same answer to the same question, higher when you want the model to come up with something new.

"That does not work" *(stale verdict)* — An observation that became a judgment when the date fell off it. Someone tries something once, it does not work — rightly — and the outcome stays put while the world shifts underneath it. The antidote is an observation with a timestamp: *at moment X it had status Y*, plus a fixed retest. **The shelf life of a judgment has become shorter than the memory of it**, and that is new: the unit used to be a decade and is now a few months, and not linearly.

The cost of one attempt *(de kosten van één ronde)* — What it costs to try something once: build, measure, look at what came out. The quantity that decides whether ten attempts get made or one, and so

whether things get checked or guessed. More important than speed, because it applies to an approval, a trial or a quote just as much.

The thousand-times question — At every choice: does this still work, and is it still affordable, if I do it a thousand times? And then a thousand times again. It is the same question as always, but it gives a different answer — and that different answer is exactly where the step from tool to organization begins.

Ticket — One bounded task with a number, which the work hangs on and which records what was decided. The organization's memory, and the only thing one agent can take over from the last.

Token — The piece of text a model reckons in: roughly a syllable, sometimes a whole word. You pay per token and your context window is counted in tokens, so it is both the unit of cost and the unit of space.

Tool — A concrete action you give an agent: read a file, send an email, make a payment, run a search. A model without tools can only talk. Tools are what turn a conversation into an act, which is also why they are where you draw your limits.

Work it out, fix it, guard it *(Dutch: bedenk, fix en bewaak)* — The short instruction that covers a whole round: think of how it can be solved, carry it out, and make sure it cannot come back. The third step is where it breaks down; without it you have solved one mistake instead of a kind of mistake. Three words replace a whole conversation, and they are not meant to have it done once but to have it keep being done.

Zero dependencies — The choice to need nothing from outside. More work up front, and afterwards something that still builds in five years.