

SECOND EDITION · RELEASE CANDIDATE V2.0.0-RC2 · 2026-10-19

Generation ZX Developers

GenAI Startup Edition

Dinis Cruz

Release candidate 2 of the second edition: not signed off, not an official version. The first edition (2018) is the only official version. This candidate was written by the book's agents from the ideas register and is for reading and marking up. Its text is frozen as v2/versions/v2.0.0-rc2.md; the title is Dinis's decision of 19 October 2026 (decision 014). Text: CC BY 4.0.

Contents

Before you start

1. Gen ZX: who this book is for, and what it is for
2. You are lucky to be here
3. Everything is connected: graphs
4. Where do you land?
5. Connect the dots backwards

The frame

6. What generative AI is
7. Lenses, roles and loops
8. Everyone who can brief an agent is a developer
9. Explorer coding, not vibe coding, and the handover

Getting the job, or making it

10. Hired for the workflow
11. Act as the job you want
12. Publish, publish, publish: your own newsroom

Working with agents

13. Your first workspace
14. How to work with agents

Direction and judgement

15. Direction is the scarce skill
16. Facts, evidence and the record

Start the company

17. Be a founder: the expert founders
18. Ship it, and see if they miss it
19. Price it, then give it away
20. Sell
21. Fund it, or do not
22. The company itself: the UK, Portugal, Lithuania, the European Union

Build it without a server

23. Serverless, for real
24. Privacy by design: SG/Send, sgit, and the value of constraints
25. Controlling your agent: RiskMandate

- 26. Open source
- 27. Shipped means the user has it

The long game

- 28. Graphs, memory and your future self
- 29. Civic responsibility

Hands on, 2026

- 30. An agentic workflow from zero
- 31. Your first graph, built by hand
- 32. One idea, five lenses
- 33. A tiny agent loop
- 34. Images, infographics and storyboards, drawn by code
- 35. Meeting briefs and follow-ups
- 36. Your own newsroom and inbox
- 37. Form your company
- 38. The credits and programmes, 2026
- 39. Your first price and your first ten customers
- 40. Ship a serverless product in a weekend

How this book was made, and what you made

- 41. How this book was made, and what you made

About this release candidate

This is release candidate 2 of *Generation ZX Developers (GenAI Startup Edition)*, the second edition of *Generation Z Developers* (2018). It was not written the way the first edition was, and it is not the same book that release candidate 1 was, a few hours earlier.

In 2018 Dinis wrote the chapters himself, in a public repository, and readers sent fixes as GitHub issues. In 2026 the book is talked into existence: voice memos, written up with a journalist agent as articles in his voice, each ending with a brief for the roles that finish a book. A librarian reads every article, every decision and every one of the 2018 chapters and outlines, and writes one idea per thing the book would say: a sentence the book stands behind, with where it came from, down to the section of the article, and the 2018 file it answers. The ideas are placed on a chapter map, and each chapter is written from the ideas placed in it, by an agent that reads the articles and the 2018 text behind them and nothing else. The rule is simple: every number, every quotation and every story has to be in a source the chapter names. Where the sources run out, the chapter says so, in a note addressed to Dinis.

Release candidate 1 was the first end-to-end draft under that method: thirty chapters, written in a day. Dinis read it and arrived at the founder. The reader he wants to give this book to is the person who wants to start a GenAI company, and that reader is two generations: Generation Z, entering the workforce and becoming the entrepreneurs, and Generation X, the subject-matter experts he sees starting companies, often their parents. Together, Gen ZX, which goes back to his origins, the ZX Spectrum. The book got a title, an objective (by the end of it you have started a company), a new part at its centre, *Start the company*, a part rebuilt around shipping without a server, four more hands-on chapters, and a measure: the companies created from it. The articles the newsroom already had on pricing, selling, funding, zero-knowledge vaults, constraints and control came into the register as the first round's batch, and this candidate was rewritten under that frame: forty-one chapters in ten parts.

Some chapters are honest placeholders. The newsroom has no article yet on creating a limited company in the UK, on Lithuania, on Portugal, on the European Union's programmes, on the first customers, or on how a reader tells us about their company. Those chapters stand as their titles, their ideas and what is missing, until Dinis records the memos; the notes to him say which.

This is a candidate, not a version. The first edition is still the only official one. The candidate's working copy lives in the book's vault; this text is a frozen snapshot of it, so that what Dinis reads and marks up cannot change behind him, and his feedback produces the next candidate rather than an edit of this one. From here on, every word, change and idea that wants to come in goes before a review board, an advocate and a cutter and the readers and a checker and the editor, and Dinis decides; every verdict is kept with its reason and becomes a precedent. The diff between this candidate and the last is on the website, block by block, because Dinis reads the diffs.

The voice is Dinis's throughout, in the first person, because the material is his. Where a chapter sounds like him, that is the material doing its work. Where it does not, that is the agents', and it is what the review is for. The verbs he answers with are the ones the process gives him: keep, rewrite, merge, cut, article first, later,

not doing it.

Read it as a reader first. Then, if you are Dinis, read it with a pen. And if you started a company because of something in it, tell us: that is the first entry in the measure.

PART 1

Before you start

Five chapters before the argument starts. Who this book is for and what it is for, why I think you are lucky to be here, one chapter on graphs because everything after it is one, how to find out which kind of developer you are, and why the path only makes sense afterwards. Read these first; then go where your map says.

1 Gen ZX: who this book is for, and what it is for

The first line of the 2018 book was this: "Hi Generation Z Developer, if you are one and want to learn as much as you can about your craft, this is the book for you." One generation, one reader, one line.

This edition needs a longer first line, because it is for two generations, and because it has an objective the first one did not have. So before anything else, let me say who I am talking to, and what I want to have happened by the time you close the book.

Two generations

In 2018 I wrote that Generation Z was "the generation born after 1996", "the first real 'born online' generation", and "the generation that is entering the job market right now". I focused on them because I had two daughters in it, because I had spent time teaching UK school kids to hack in the ethical sense, and because I worried they had missed the history behind the technology they took for granted. Eight years later they are entering the workforce for real, and a lot of them are not looking for a job. They are hungry, they are focused on this, and they are the new entrepreneurs. They are still the first reader.

The second reader is my own generation. The more I thought about who I want to give this book to, and what I want them to do with it, the more I arrived at the people I actually work with. The founders I see starting companies now are, a lot of them, Generation X. They are the subject-matter experts: twenty years inside a problem, in a pharmacy, a clinic, a legal practice or a security team, who have described the tool they need to vendor after vendor and never got it.

In *The expert founders* I put the change in three words: "Execution just got cheap." When agents do most of the building, the person who knows the problem can build the first version of the answer themselves, instead of explaining it to a team they could never afford. And the data was always on the experts' side, which is worth hearing before somebody tells you it is too late. In the largest study I know, of 2.7 million US founders, the mean age at founding of the fastest-growing one in a thousand new companies was 45, and founders with three or more years in the same industry reached that top tier at twice the rate of those with none. What held the experts back was not knowledge. It was access: to capital, to engineers, to the long and expensive build before anything could be shown. That is the part that just got cheap. A third of new US startups now have a single founder.

Often the two readers are in the same family. Generation Z are, by the dates, the children of Generation X, which is an interesting gap: one grew up with the agents, the other grew up with the problem. In the storyboard that goes with this chapter, the mother asks whether she can just build it now, after twenty years of explaining it, and the daughter says yes, and shows her the agent. That is the book, in two lines.

The dates I use are the Pew Research Center's: Generation X born 1965 to 1980, millennials 1981 to 1996, Generation Z 1997 to 2012. Boundaries differ between sources, and like all generation boundaries they generalise; the 2018 book said so too, and I still mean it. That leaves the millennials in the middle. They can read this, and they will get a lot from it, but theirs is a different story, and I would rather write the two I know from the inside: I am Generation X, and my daughters are Generation Z.

Generation ZX

Put the two letters together and you get Generation ZX, which is a plot twist I did not plan and now cannot give up, because it goes back to my origins. The ZX Spectrum was launched by Sinclair Research in the UK in April 1982, when Generation X was between two and seventeen. If you are my age, you may have typed a program from a magazine into one, line by line, and watched it run, or not run, and gone looking for the typo. That machine is where a lot of us learnt that a computer was something you could own, open and program. The storyboard has the whole language of it on a board: two lines of BASIC, and the second one is GOTO 10.

One more thing about the name, and then I will leave it alone. It is Gen ZX, not Gen XZ. XZ is also the name of a compression library that had a backdoor planted in it in 2024, the most famous supply-chain attack of recent years, and in a book that covers security the order of the letters matters.

And they are developers. The title keeps the word on purpose. The entrepreneurs I am talking about are coding: with agents, in the way the newsroom calls explorer coding, which is a better name for what most people call vibe coding. *Explorer coding* says it in one line: "What people call vibe coding is exploration, and the word should say so." The explorer builds to find out, and what the exploration produces is the brief for the version other people will rely on. A founder needs both halves, the exploring and the handover, and *Explorer coding, not vibe coding, and the handover* is the chapter on them. Before it, *Everyone who can brief an agent is a developer* says why the word now includes you, even if you have never written a line of code and do not intend to read the ones the agent writes.

What it is for

The 2018 book had a chapter called *Be a founder*. What it meant was attitude: "act like one of the founders of the business" you work for. Imagine you were employee number four and cared deeply about the company. Ask what you would do differently, what needs to be done now that would make a big difference, what values you would fight for. Then ask the question the chapter ended on: "what is stopping you from doing just that?" It is still good advice, and the part of this book about getting a job keeps it as the then of a then-and-now. But in 2026 the advice is literal. You can be the founder.

So this book has an objective, which the first edition did not. I said it to the team making the book in one sentence: "Our job is to empower the reader to create a startup by the end of the book." This is the book you give somebody who wants to start a GenAI company. If that is you, here is what it has to contain, for real: the company itself, how to create a limited company in the UK, the case for Lithuania, Portugal, because that is where I am from, and the European Union and what exists there for a founder; the cloud providers and the startup programmes that exist right now; and the first sale.

Two honest clauses about that list. The chapters on the company are written from experience, mine and the people I work with, and they are not legal or tax advice. And the programmes and the credits change every year, so they live in the hands-on part at the end of the book, dated, to be renewed each year while the argument in the rest of the book stays put.

The shape of the middle follows from the objective. *Start the company* is the centre of the book: be a founder, ship it and see if they miss it, price it and then give it away, sell, fund it or do not, and the company itself. The whole thing is about shipping now. As the newsroom's *The question is whether they miss it* puts it: "Not to plan, not to pitch, not to architect. Ship." You ship it, and it is painful. If you give it away and they do not miss it, you try again.

Then *Build it without a server*: ship your startup serverless. No servers of yours with logic in them, no database of yours, the work done on the user's side. A founder with no servers has nothing to run at three in the morning and nothing to pay for when nobody is using it. SG/Send and sgit come in there, because they give you data privacy without a server that can read the data, and RiskMandate's techniques come in as the way to control the agents your company runs on. Each has its chapter, and the hands-on part ends with a product shipped that way in a weekend.

The measure

A book that is a book to read is fine. A book with an objective is a different thing: it can fail, and it can be measured. So I will say plainly how I measure this one: by how many companies have been created from it.

That means asking you to tell us what you are doing, and building a community around the people who do. The last chapter, *How this book was made, and what you made*, says how. If you started a company because of something you read here, that is the first entry in the measure, and I would like to hear about it before I hear about anything else.

FOR DINIS

the measure. How does a reader tell us about their company, and is the list public? The article keeps it as your question, and the last chapter needs the answer before it can do more than ask.

For Dinis: is this the second edition of *Generation Z Developers*, or a new book that grew out of it? Decision 014 leaves it open. This chapter says "edition" throughout, and the publisher's brief waits on the same answer.

That is who you are, as far as this book is concerned, and what I want you to have done by the last page. The next chapter says why, whichever of the two generations you belong to, you are lucky to be starting now.

From the register: The reader this book is for is the founder, and its objective is that by the end you have started a company, Two generations, Z and X, together Gen ZX: back to the ZX Spectrum. Answers: Generation Z, Introduction.

2 You are lucky to be here

I discover things I can do. That is the sentence I keep coming back to when someone asks what it is like to work the way I work now. I discover workflows. I follow an idea, and I can execute it, often the same afternoon. The feeling is not relief, and it is not fear. It is closer to wonder, and now that you know who this book is for, I want it to be the first thing in it, because it is the first thing I would want you to feel.

I heard a woman on a podcast describe the same thing: the wonder that happens with generative AI, the feeling of discovering what you can do. I recognised it at once. I cannot now say who she was, so I will not put a name to it.

FOR DINIS

who was the podcast? If you can find it, the book should credit her here; the article left her unnamed.

How the first edition started

The 2018 book did not start with wonder. It started with a slide I had shown to a room of young people in 2017, and a question the book repeats on its first page: "Here is the slide that started it all in 2017, how many logos do you recognize?" Very few recognised them, and nobody had told them the story behind each one.

The history still matters. But the book that followed went too technical, too deep, too soon. Parts 2 and 3 were nineteen chapters of technologies to know. The argument was at the two ends, and the middle was a list.

What I had actually wanted is in that book's introduction, under the heading "Overwhelming curiosity": moments when you "follow link after link about a particular topic" and think it is awesome. The 2018 book called that place the zone: "If you are lucky enough to find yourself in this place, congratulations! You just found 'the zone'." The second edition starts there. Not with what you should know, but with what you can now do, and the wonder of discovering it.

In 2018 I also asked, of those logos, "Why is the **why** important?", and I had an outline for a chapter called *Start with why*, after Simon Sinek's golden circle: why before how before what. It was never written. So let me start with the why of this book, in one sentence.

You are lucky to be here

Whoever is reading this in 2026 or 2027 is ridiculously privileged to be living at this moment in time, in the middle of a huge transition in society, while the pieces are still landing.

I know that is not what you hear. If you are starting out, what you hear is that AI took the jobs before you got there. If you are my age, what you hear is that it is too late for you. I tried to read one of the first kind of essays this month, in Business Insider, under the headline "I thought working in AI would put me ahead of the curve. Two years in, I've never felt more behind and tired of the topic." I could not get past the paywall,

so I cannot tell you what it argued. In Europe in particular, I think young people are being fed a lot of fear about AI, and are not embracing it the way they should. That is my reading, not a measurement, and this edition still owes you a proper answer to it.

I am not going to tell you that none of the fear is right. Projects will be cancelled and money will be lost; every transition has had that. But the fear is what you hear. It is not where this book starts, and I do not think it is where you should start either.

Here is my position, not a forecast: ignore the doom, because nobody knows what is going to happen. Every time we had one of these technological transformations, people said the same things, and they always said this time is going to be different. And each time the paths evolved: industries changed completely, jobs disappeared and appeared, and the people who learned the new tools early did well out of it.

The transformations I have lived through

I have seen several in my own lifetime, each a revolution that changed whole industries, and if you are my generation you have seen most of them too. The dates are the commonly recorded ones.

- The home computer, my ZX Spectrum, in the early 1980s. A computer became something you could own, open and program.
- The BBS, in the 1980s and early 1990s. Text messages and chat between strangers, over a phone line.
- The internet, in the 1990s. Everything published, linked and searchable.
- The mobile phone, in the late 1990s, and the smartphone from 2007. The network in your pocket.
- The cloud, from the mid 2000s. Servers you rent by the hour instead of buy.
- The apps, from 2008, with the app stores. Software delivered to everyone, instantly.
- Generative AI, from November 2022. Software you can talk to, and that can build software.

The 2018 book had an outline for the second of those, *BBS and modems*, with a story I still have to tell properly: blowing up my home's telephone system trying to connect to one.

The only difference this time, I sometimes joke, is that this one is televised: far more visible, because of the money, the players, and how much of the stack it touches at once. It will also take longer to land than the headlines say; it just feels as if it is happening all at once.

I lived through the dot-com crash: enormous hype, then a crash. The Nasdaq peaked in March 2000 and had lost roughly four fifths of its value by October 2002, and the internet still changed everything. We might have a crash or a correction this time too; nobody can say when, or whether. None of that makes the transformation less real. A crash would change who makes money; it would not undo what people can now do.

FOR DINIS

does the dot-com crash belong here, on the first pages, or later in the book? The article left it as your question. It is here for now because part of the reader's fear is a fear of exactly this crash.

My own last two years are the counter-story to the burnout essay. In *Two years in, the opposite* I put it this way: "I have been doing AI for the last two years and a bit, and I have never felt more productive." I have never felt happier at work, or more empowered. There is an anti-AI mood at the moment, and stories of burnout feed it; stories like mine should be told too. It is one person's account, not a study.

A couple of years ago I wondered whether we had missed the boat; it felt as if all the big things had been done. Now the pieces of the puzzle are landing again. This is the biggest opportunity of your career, whether that career is starting or is twenty years in: the moment when you can still be a player, before things stabilise, as they always do, until the next big thing.

Technology that finally works for you

For a long time technology has not worked for you. If you are Generation Z, you in particular were handed a technology stack designed to hook you, and to make you a passive user. There was some creating, but mostly you were a consumer, living in a world of advertising, attention and dopamine. I believe that will one day be treated the way tobacco is today; that is my belief, and the evidence is still being gathered. The 2018 book had a draft chapter on dopamine that already called phones' attention-grabbing activities "the new smoking".

My generation was privileged in a different way: we could build things. A whole generation grew up programming, building computers, mixing parts. A computer was not something that just arrived; you could take it apart. And then, somewhere along the way, most of us stopped, and became users of software that other people made for our fields, and not very happy ones. One of you was handed technology to use, not technology to build; the other put the building down. Now you can both build again.

Here is what that looks like in practice. For twenty years the instinct in software was that a custom interface is expensive, so everyone gets the generic one: the inbox, the ticket, the form. In *Custom UIs are not the exception* I argued that this is now backwards: "a custom interface per message, per thread, per topic, per question is not an exception. It is how interfaces get made." When an interface costs an afternoon, you stop asking whether it is worth building and start asking which moment it is for.

The wonder underneath that has a name in my influences. Bret Victor's talk *Inventing on Principle*, from 2012, says that a creator needs to see what they are making, as they make it. Every step between you and the thing you are making, a compile, a deploy, a refresh, is a place where an idea can die. Generative AI removes a lot of those steps. The 2018 book had an outline for that talk too, with a note that is very much me: "print this and read it".

None of this is only for the creative ones. It is not that you have to be creative, and if you are not, you are left out. It is that you can now scale the things you are not good at. You can have technology work for you.

And it is why I am an optimist about where technology goes. In 2018 I recommended Kevin Kelly's *The Inevitable*, because of the way he maps what is likely to happen next: technology evolves with tendencies of its own. Which of his books I would hand you first is a question I have not settled. The frame of this book, which *Where do you land?* introduces, is built on that kind of seeing: things move in a direction, and the people who can see it can be early.

How to read this book

Read it on paper, with a pen. Books, and narratives, are still the best way we have to take in information, and your brain takes it in much better from a page you can mark. If you are reading this as a PDF, print it. The 2018 book had a chapter called *Pen and Paper* that said "analogue techniques like the pen and paper are actually better technologies for creating and capturing ideas." I still think so, though that is my experience rather than a finding.

Read it somewhere you can give it your undivided attention. That is a skill, and the best professionals in the world have it. The phone you were handed was designed to take it from you.

Then use the rest of the book the way *Where do you land?* describes: find your centre of gravity first, and read the chapters marked for it before the others. If you are here to start a company, know where you are heading: the centre of the book is the part called *Start the company*, and the hands-on part at the end is where you form it, price it and ship it. The chapters before them are the ground those stand on, so do not skip them. *Connect the dots backwards* is about how to learn, and its first lesson is that your path will only make sense looking back, so you do not have to see it now. You have to start.

That, in the end, is why the title says developer. Craftspeople made their own tools. In the past they did it out of necessity; now you can do it again, because the tools can be made to fit you, and because, as *Everyone who can brief an agent is a developer* argues, the bar has dropped. The person who has spent twenty years inside a problem is the craftsman here, and the first tool they make may be the first version of their company. It is still your name on the work, and it is still your thinking. You think with the AI; you do not delegate your thinking to it.

You are lucky to be here. Not because it will be easy, but because the pieces are still landing, and you can still be the one who places some of them.

From the register: The book opens with wonder, not fear and not technologies, Whoever reads this now is lucky to be here, Ignore the doom: nobody knows what happens next, Seven transformations, lived through: the Spectrum, the BBS, the internet, the mobile phone, the cloud, the apps, AI, Technology can finally work for you, Read it on paper, with a pen, The fear, impostor syndrome and anxiety about AI among young people, The immediate connection between a creator and what they make is the wonder the book opens with, Purpose, and optimism: Sinek for why, Kevin Kelly for what technology wants. Answers: Introduction, Generation Z, The Zone, BBS and Modems, Dopamine.

3 Everything is connected: graphs

Think of the board on the wall in a detective story. Photographs pinned up, notes under them, and red string running from one to another. The detective stands in front of it and asks one kind of question, every time: what connects A to B? Who knew whom, who was where, what links these two. Everything on that board is either a thing or a connection between two things.

That board is a graph. If you have followed one of those stories to the end, you already know how to read one.

I have met people for whom a graph is a picture with boxes, or a chart with bars. So let me say early what this book means by the word, because it is one of the most fundamental ideas in it, and it comes back in almost every chapter.

A graph is a network, not a diagram

A graph is a network: nodes, and the edges that connect them. The first of the graphs books, written with agents on graphs.sgit.ai, puts it in one line: "In this book a graph is a network: nodes and the edges that connect them. Never a chart, never a figure." And its opening sentence says why the edges matter more than the nodes: "A node connected to nothing is literally meaningless."

People mean three things when they say graph. They mean networks, for analysis: who is connected to whom. They mean graph databases, for storing and fetching things fast. And they mean semantics: what something means, which in a graph comes from what it is connected to and how. That is the graphs book's way of dividing the word; others divide it differently. This book uses the first and the third senses, and almost never needs the second.

In 2018 I left an outline for a chapter called *Graphs* and never wrote it. It had four bullets: "Graphs are everywhere", "I've been thinking in graphs for years", "very hard paradigm shift to make", "requires an ability to think in nodes and edges". That is this chapter in miniature. Eight years later the shift still feels hard to explain and easy to show, which is why the storyboards in this book draw graphs, the detective's board among them.

FOR DINIS

the 2018 outline's first bullet says "See my presentation on Graphs". Which presentation did you mean? If it still exists, the chapter should name it and take from it.

The whodunit

The detective's board is the easiest way in, and it is why I think a reader of Generation Z will find this chapter easier than my generation did. In *Follow the rabbit holes* I said that people underestimate the young: they are very good at connecting the dots, very good at finding things, very good detectives. That is my experience, from my family and the young people I have worked with, not a study. But it is what thinking in

graphs is: tracing what an unfamiliar thing connects to, trusting a claim because of where it came from, asking what else breaks if this one fails. The graphs book has a line for anyone who thinks this sounds exotic: "you may already think in graphs without ever having called it that."

If you are my age and twenty years inside a field, that line is for you too. You carry a graph of your field in your head, every rule and every exception and what each one touches; you have just never been asked to draw it, and this book will ask.

The board is a teaching device. Real investigations are messier than red string. But the question stays the same, from A to B, and once you see that the question is about edges and not about the boxes, you can ask it of anything.

A map is a graph with positions

Take the same nodes and the same edges, and now give every node a place on the page that means something. You have a map.

In my talk on why Wardley maps changed how I think, the first two shifts were exactly that: "Its a graph with context", and "When a graph becomes a Map". A Wardley map is a graph where each node has a position on two axes: how visible it is to the user at the top, and how evolved it is, from a brand-new idea to something everyone takes for granted. The positions are what make it a map and not only a graph; the edges are what make it a graph and not only a picture.

The 2018 book said the same thing from the other side, in the chapter *Dot Language*, where I complained that most diagrams in software "are usually not maps": they have components and context but no anchor, no position relative to the anchor, and no consistency of movement. That chapter was about drawing diagrams as text, so that software could draw them and a team could version them like code. Its real complaint was that we build "large interconnected and complex structures without accurate maps", and its best line is one I still believe: "What is crazy is to make decisions based on shadows of reality."

That is why this chapter and *Where do you land?* sit next to each other. One gives you the graph; the other gives it positions and asks where you are on it.

FOR DINIS

graphs before Wardley maps, or the other way round? The chapter map puts this one first because a map is a graph with positions; the article leaves the order as your question.

Graphs of graphs

Here is the part that took me longest to see, and that I now cannot unsee. Graphs nest. Zoom into a node and you find another graph inside it, with its own kinds of nodes and its own kinds of edges. Zoom out and the whole graph is one node in a bigger one.

The newsroom's introduction to what I call fractal semantic graphs puts the consequence in two sentences: "There is no top and no bottom. There is only the altitude you happen to be looking from, and how much definition you choose to load at it." A law is a node in a graph of laws. Open it and it is a graph of articles. Open an article and it is a graph of the terms it uses. Each level has its own vocabulary, and none of them was made to match the others. What connects them is that each one is still a graph.

In a way, you can see nature like this too: a system, the molecules in it, the particles in those, everything connected to everything else, and groups of groups forming the layers we think in. That is a way of seeing, not a claim in physics. But it is how I view the world: it is made of connections. Groups of groups are the natural abstraction layers, and abstraction layers are how we think.

It is also why provenance is a graph. Where a claim came from is a chain of edges back to its source, and the newsroom's rule for a record is that it should be "not a pile of articles, but a graph whose every edge has a date and a source". A chain that ends in a rumour is a finding too. *Facts, evidence and the record* is built on that.

Code is already a graph

The 2018 book had a second chapter that was a graph without saying so. *AST* opened by calling an abstract syntax tree "a graph representation of source code", and it told you that if you could "see code or raw data as a graph", you would have made a paradigm shift that would help you across your career. I still think so, and I now think it goes further than I said then.

Source code is the best example of a fractal graph that I know, because the layers are not an interpretation; they are a fact about the thing. At the top is a person trying to do something. Below that, the features and flows. Below that, the components people draw when you ask for the architecture. Then the classes, then the methods and the calls between them, then the syntax tree, and on down if you want to. Each layer is a graph with its own vocabulary, and you change universe when you step from one to the next.

What changed since 2018 is that naming the nodes and the edges is now cheap. A language model can read a function and say what it does, and propose the edge between two nodes faster than a person can type it. So a change to the code can be read at every layer, not only as lines. A refactor is a change that moves the bottom layers and leaves the stories at the top still. A bug is a story that stopped holding, and its fix is visible at the top as that story holding again. The blast radius of a change is the climb from the methods it touched up to the stories that can reach them, and you can read it before anyone runs anything.

This is the only honest answer I have to the characteristic failure of AI-generated code: the change made in one place that breaks something three packages away. In a text diff that is invisible; in the layered graph it is a changed node that reaches a command the author never looked at. In the newsroom I worked this through on a real tool, the *sgit* command line, read as layered graphs with nothing run: one commit of 75 lines added and 7 removed in three files turned out to reach nine of the tool's 72 commands and six of its eleven user stories. The stories were hand-written, the example runs nothing, and it is published as a vault with the read

key on the page, so you can check it. The technique underneath, following the call tree from one method and writing out only that code, is one I used for code review on the O2 Platform more than a decade ago and described in the 2018 *AST* chapter as method streams. It is back, and it is one script now.

If you want to own code in production and answer for it, this is your chapter as much as the detective's board is. And if you are a founder whose agents wrote most of the product, it is yours too: this is how you read what they made at the altitude you need, the stories at the top, without pretending you have read every line.

Start anywhere

Here is the thing about graphs that I most want you to take from this chapter, because it is about how you feel when you look at everything there is to learn, or at everything a company needs before it exists.

A graph has no front page. It has no top, so there is no right place to begin. Whatever node you start from, the edges take you to the rest. It does not matter where you start; it matters that you start. It does not matter, in a way, whether you are going straight in the right direction; it matters that you get movement. You start to learn, you start to connect the dots, and the connections do the rest.

So when it feels like too much, and it will, pick a node. Any node. Start there and follow one edge. Then another. *Connect the dots backwards* says that your path will only make sense afterwards, and this is why: a path is a walk through a graph, and the graph does not care where the walk began.

The hands-on part has a chapter, *Your first graph, built by hand*, where you draw one of your own. You do not need it to read on. You need only to remember, when the word comes up, that a graph is nodes and the edges between them, that everything in this book connects to something else in it, and that you can start wherever you are.

From the register: A graph is a network of nodes and edges, not a diagram, The whodunit is the easiest way in: the detective's board is a graph, A map is a graph with positions; a Wardley map is a graph with context, Graphs nest, and so does the world: groups of groups are the abstraction layers we think in, You can start anywhere: it matters that you start, and get movement, The 2018 book was already there: Dot Language and AST were graphs without saying so, Source code is a fractal semantic graph, and a review should read every layer of it. Answers: graphs, Dot Language, AST (Abstract Syntax Tree).

4 Where do you land?

Take someone's first job, or the first product of a company that does not exist yet, and draw what it depends on. Put the person at the top, with the thing they need to get done. Under that, the things that serve the need: the way they work, the tools, the models those tools call, the computers the models run on. Now order it twice. Down the page, from what the person sees every day to what they never see. Across the page, from the brand new and badly understood on the left to the everyday thing everyone takes for granted on the right.

You have just drawn a map. I have not named it yet, on purpose. Look at it first.

The models are on the right, moving further right: a product becoming a commodity. The way of working with agents is on the far left, new and rough. Everything on the map is moving, and all of it the same way: from left to right, from rare to ordinary. Once you see that movement you cannot stop seeing it, and that is the point of this chapter.

Maps everywhere, shown before they are named

The map you just looked at is a Wardley map, after Simon Wardley, and they run through this whole book. You will meet them in every part, sometimes with their names on and sometimes, as above, without. I want you to feel the pattern before you learn the vocabulary: the user, the need, the chain of things that serve it, and the movement.

A Wardley map is a graph with two axes of context. *Everything is connected: graphs* gave you the graph. The first axis is vertical: a value chain, anchored at the top on a user and a need, each component drawn under the components that need it. The second is horizontal: evolution, how far each component has travelled from an idea nobody has built to something you buy by the unit without thinking. Wardley's four stages, genesis, custom built, product and commodity, are the scale on every map in this book.

Every dot is a claim. The positions are judgements, and the judgements are the argument. A map you cannot be wrong about is a diagram.

Why do I care so much? In 2022 I gave a talk for the Open Security Summit on why Wardley maps changed how I think, and listed eleven shifts they caused in me. The eleventh was: "Wardley Maps gives me inner peace (allowed me to understand why things happen)". Once a thing is on a map, placed and connected, I stop carrying it in my head. The map also tells you what to build and what to buy, and whether a market exists at all: a commodity layer is a hard sell, and the custom building on top of it is where the market is. If you are about to start a company, that last sentence is the first thing the map does for you.

The four stages are not only for activities

When people start with maps, I see a lot of them stop at the four stages most examples use, and treat them as a scale for activities only. The movement is everywhere. For that talk I made a table of it. The first four rows are from Wardley's published material; the rows below the line are my own explorations.

- Activity: genesis, custom build, product, commodity.

- Practice: novel, emerging, good, best.
- Data: unmodelled, divergent, convergent, modelled.
- Knowledge: concept, hypothesis, theory, accepted.
- Openness of content: confidential, published with ordinary copyright, CC BY, CC0.
- Openness of software: proprietary, GPL, open source, public domain.
- Backup restore time: infinity (no backups), a week, a day, real time.
- Data, by shape: unstructured raw text, spreadsheet, graph, map.

There are more rows in the original, on incident response, security logs, risk, attacker motivation, concepts and ideas. Evolution is in how open a piece of writing is, how fast a team can recover, whether your data is a pile of text or a map. Once you see it, you see it in your own work and your own career.

Where do you fit?

Now the question I most want this book to ask you, early and often: where are you on it?

Wardley named them: pioneers, settlers and town planners in 2015, and since 2023 explorers, villagers and town planners. Explorers "explore never-before-discovered concepts, the uncharted land"; "They show you wonder but they fail a lot." Villagers "turn the half baked thing into something useful for a larger audience". Town planners "take something and industrialise it taking advantage of economies of scale." The 2018 book had this as an outline, never written: explain the concept, show how everybody is different, connect it to Agile, Lean and Six Sigma. This chapter does the first two.

I think these are personalities, and I think Wardley nailed them. In business we keep finding the same thing over and over, in Myers-Briggs and in whatever test is in fashion, and in my experience the three do a better job of describing the people I have worked with. That is my experience, not research; the business tests have a contested evidence base of their own. What I can tell you is what I have seen.

The explorer is bursting with ideas. They love to experiment, to try things out. Once it works, it gets boring, and they build on top of it. They are not the person playing with the Lego; they want to invent the bricks. Failure is exciting, because failure is just an experiment. Change is natural to them, and the immediate connection with what they are making, the instant feedback, matters to them more than to anyone. What they make only works for early adopters, because it is going to break: full of wires, half-baked, a prototype.

The villager loves the user. They are emotionally attached to the user being happy, almost the user themselves; they put themselves in the middle of the problem and solve it. They thrive on packaging, releasing, going to market, taking something and refining it. They do not like failure that much, because for them the product has to work. Often it is about trimming it down, about simplicity, which is where design comes in: the first version just has to work; the second version has to be designed. They are the ones who let things scale; without them, everything would die before it reached the other side.

The town planner geeks out on the details. A good friend of mine says boring is good, and that is them: the nice hum of effectiveness, marginal gains, scale, bigger volumes. They like to be invisible, because they know they have done the job right when nothing goes wrong. Their work is making something a commodity,

which is still hard. They are the people who pay attention to the letters, the words, the dotted i's, the grammar, the code. That is craftsmanship, the obsession with getting it right, and it needs less recognition than the product people need, which is fine.

If the third one is you, and you are a little proud of doing things by hand and sceptical of shortcuts, hold on to that. The newsroom's argument in *Explorer coding* is that what people call vibe coding is explorer work, and that an explorer-coded app "is not ready to be in line": once other people depend on it, it is mission critical. Villagers make it safe to depend on; town planners own the code in production and are accountable for it. The further towards commodity a thing moves, "the fewer agents in line, and the more deterministic code". Your centre of gravity is not out of date. It is where the explorers' work has to end up.

And if you are the founder, you need both halves. The explorer in you builds to find out, and what the exploration produces is the brief for the version other people will rely on; the handover to the people who maintain it and own it is where a company starts to exist. *Explorer coding, not vibe coding, and the handover* takes both halves up.

A centre of gravity, not a box

Everyone has a centre of gravity. Some can play at several levels, but there is a centre, and knowing it helps. The test is simple. What do you care about? What makes you lose your sense of time? What is the thing that, when you do it, time just flies? That is where you are.

It moves. As your career changes, your centre can move with it. But it matters to recognise where you are now, because the skills you need as an explorer are different from the skills you need as a villager, which are different again from a town planner's. All three are brilliant. They are brilliant at different things, and they should be learning different things.

This is a lens, not a label. A software vendor has spent thirty years trying to be all three for its users, and in *Software vendors should become villagers and town planners* I argued that users are now better explorers of their own needs than the company will ever be, so the company should let them explore on top and sell them the maintenance of what they build: "What stops me is not building it; it is maintaining it." The same handover, from the one who explores to the ones who maintain and own, is how a team works, how a career moves, and how a one-person company grows into one with a second person in it.

How to use this book

Every chapter carries a badge, in black and white, because the book will be printed that way. A compass marks a chapter that leans to the explorer. A house marks one for the villager. A grid marks one for the town planner. A chapter for everyone carries all three. The badges are a first attempt, drawn for the storyboards, and whether they work is still open.

Work out your centre of gravity first, using the test above. Then read the chapters for everyone, and the chapters with your badge, before the others. An explorer will want the chapters on explorer coding and on building fast with agents early; a villager the ones on shipping, on selling and on your own newsroom; a

town planner the ones on controlling your agent, on the record, on serverless for real, and on reading code as a graph. Then come back for the rest; you will work with the other two every day, and if the company is yours, you will hire them.

The last part, dated 2026, is hands on: workflows you can set up this week, and the company you can form, price and ship this year. It will be renewed as the tools and the programmes change; the argument in the rest of the book should not need to be. Whether the book should one day be grouped by centre of gravity, rather than only marked, is a question I have not settled; for now, the badges carry it.

The jobs that never scaled

One more thing, because the map is also about where the work is.

The opportunity is not only in new jobs. It is in jobs we have been doing for a very long time, sometimes for millennia, that never really scaled and never had good technology, done one person at a time. We can now scale those jobs with agents. The map shows you where: a layer where everyone is custom building their own version of the same thing, on top of commodities, is a market, and each of those builders would rather buy.

If you have spent twenty years inside one of those jobs, you are no longer waiting for somebody to build the technology it never had. You can build the first version yourself, with agents, and the map tells you whether anyone else would buy it. If you are starting out, the same map tells you which of those jobs is still waiting for someone to try.

So when you ask where you land, ask it twice. Where on the spectrum, explorer to town planner, does time fly for you? And which old job, now that it can finally have good technology, would you want to be the one to scale?

FOR DINIS

three examples of jobs that never scaled, and now can. The article asks for them and the chapter needs them here; without them this section is a claim without a case.

From the register: The opportunity is in old jobs too: the ones that never scaled, and now can, Wardley maps run through the whole book, shown before they are named, The four stages of evolution are not only for activities, Explorer, villager, town planner: personalities, each with a centre of gravity, Each centre of gravity needs different skills and a different learning path. Answers: Are you a pionner, Settlers or Town Planners.

5 Connect the dots backwards

Steve Jobs said it in his commencement address at Stanford on 12 June 2005: "you can't connect the dots looking forward; you can only connect them looking backwards." His own example was a calligraphy class he took after dropping out of college. Nobody would have put it on a programmer's syllabus. Ten years later it came back as the typography of the first Macintosh.

If you are coming to this from somewhere else, from running a team, from a job that never involved code, or from twenty years in a field whose software you are about to build yourself, that sentence is for you. You cannot see where this goes from here. Nobody can. So you have to trust that the dots will connect, and trust your gut.

Connect the dots backwards

Trusting your gut is not the same as having no method. The newsroom already has one for walking a path you cannot see ahead of you, and the book will use it. The path is not a straight line. What matters is knowing which decisions are one-way doors and which are two-way doors, so that you go through the two-way ones quickly and think hard only about the others. It matters that you can iterate very fast. It matters that you play the game of constraints, and that you work where the smallest move shifts the most.

I have a small dot of my own from this year. I took SG/Send, a file-sending service where the server holds nothing, to market end to end: a checkout live on 1 March, ten paying customers the next day, and by 5 March a brief saying "Users buy credits, test once, and don't come back." That is where I learnt Stripe, pricing and everything else a real product needs, so by the newsroom I already knew how; and the same technology is in use now because the newsroom, RiskMandate and the vaults need it. In *From Send to sgit* I called it "a good example of how exploring something early pays off later, in a way nobody plans." Explorers find things early; the use case decides when they matter.

Your future self will connect the dots. Your job now is to make dots worth connecting.

FOR DINIS

your own dots, connected backwards. The article asks which three make the best story; SG/Send above, and The Cyber Boardroom and MyFeeds below, are three candidates. Are they the ones, or is the story elsewhere?

The edge is multidisciplinary

What I most want to stress is that, more and more, it is the multidisciplinary person who has the edge. Not someone going in a huge number of directions at once, and not chaos. The ability to learn, and to intersect disciplines.

If you have spent years doing something other than programming, that is not a gap in your CV. It is the discipline you are about to intersect. The 2018 chapter on learning said that "being a better musician, artist, athlete, car mechanic or philosopher will make you a better developer", and that is truer now, because the wall of syntax and setup has come down, and what is left is knowing what to build and for whom. That is what you bring, and for a founder it is the advantage.

That is why the humanities matter. And it is why, in a way, mathematics matters.

We teach maths as if to make people hate it

The reality is that we do a horrible job of teaching maths. One of the books I love is *A Mathematician's Lament* by Paul Lockhart, an essay from 2002 and a book in 2009, which says, in effect, that if we had set out to design a system to make people hate mathematics, we could not have done better. It opens with a nightmare in which music is taught the way maths is: notation and rules for years, and never a song.

The same is true of music, and of much else. We give the solution away. We do not teach the wonder, we do not teach the struggle, and we do not let people be creative. There is a story, often retold, that children start out very creative and school beats it out of them; the retellings are better known than the original data, so I will not lean on it until I know what was actually measured.

If you hated maths at school, you were probably taught it badly. You were not born without it. The 2018 outline for a chapter called *Love maths* had one idea I still like: get people to do the maths that sends a rocket to space, with real problems, the way the human computers did.

Learning, made for you

Our education system was designed for a very specific type of person, with a very specific set of skills. If you were very good at maths of the school kind, at memorising, at a certain kind of academic writing, you did well. That is a small subset of the population. I was not the best at that, and I think that is why I gravitated to computing: it let me be creative, and it let me find the zone.

You know the line about judging a fish by how fast it can climb a tree. It is usually attributed to Einstein, without evidence. Judged that way, the fish believes it is stupid. The academic model was generic, made for a mythical student, too high for some and too low for others.

That has changed. This is about you. How you like to learn, how you like to take in information, how you like to get lost in something. I learn more now than I ever have, because I can shape what I learn to what I am curious about and to how I like to learn, and go deep exactly where my gaps are. And you can now build the tools for that.

There is science behind this. Mihály Csíkszentmihályi's *Flow*, from 1990, describes the state the 2018 book called the zone: clear goals, immediate feedback, and a challenge matched to your skill. He named programming as one of the activities that can produce it. Anders Ericsson's *Peak*, from 2016, is the science behind learning as a skill: expertise comes from structured practice with feedback, at the edge of what you can do, not from the number of hours. Ericsson did the research behind the ten-thousand-hours number that

someone else made famous, and the credit belongs with him. The 2018 chapter on learning said the same in Kathy Sierra's words from *Badass*: practise right, and consume large amounts of good examples. What agents change is that both halves, the right practice and the good examples, can now be made for you.

Think with it; do not delegate your thinking to it

A search engine that gives you information from anywhere in the world was already a ridiculous technology, when you think about it, and my generation was the first to have it.

But search gave you an answer, without the ability to question it, to go deeper, to explore. Generative AI gives you the ability to really go deep: to challenge, to ask the questions, and to ask for the evidence. That is the difference between using it to skip understanding and using it to understand. If you ask it for the answer and paste it in, you have learned nothing and put your name on something you cannot defend. If you ask it why, then where that came from, then what would show it was wrong, you are learning.

It is still your responsibility. It is still your name; the work is still done under you, and if the company is yours, so is everything the agents build in it. This is not you delegating your thinking to generative AI; it is you thinking with it. I do it when I write: I ask the agent helping me with this book to double-check my quotes, to make sure what I am saying is coherent, and to find the evidence behind every statement. The name of the author of *A Mathematician's Lament*, which I remembered wrongly in the memo behind this chapter, was corrected that way.

And sometimes, as I always say, you use generative AI so that you do not have to use generative AI: you use it to write the script or the tool that then does the job reliably, with no model in the loop.

Generative AI lets you explore again

Exploring is back, and so is communicating in ways you could not afford before.

Look at what happened while this book was being written. I wanted a better way to communicate the book, and to think about it, than infographics, which take too long and cost a new image for every change. So I asked for something programmatic instead. What came back was a cast of stick figures drawn by code, in the browser, every time the page loads, from a data file per storyboard. The renderer is about 400 lines. The storyboards in this book are those. That is how I like to think, and that is the beauty of it: you can now discover how you like to work, and build the thing that lets you.

When people tell me they do not see it, or that they are overwhelmed, my argument is: experiment, learn, try, start, and adjust. Let your questions guide you.

Learning is a lifelong skill

Learning is a skill. Take it seriously, because it is a lifelong one. The 2018 book opened its chapter on it with a question I would still ask you first: "Do you know how to learn?" And it said what I still think: learning is like a muscle, and the more you do it, the better you become.

Historically, the roles and opportunities open to people were limited by geography, money and power. Now you can be in the middle of nowhere, with an internet connection and, less and less, a bit of compute, and have the learning systems of the world in front of you. The future, as I see it, is decentralised and sovereign: local compute, local systems. That is my position, not a forecast.

And here is a point I want you to hear plainly: by reading this book, you are already in a very small minority. The numbers behind that are the newsroom's own arithmetic on other people's surveys, for sizing, not for adding up: professional developers are about 0.6% of the world's internet users, and in the OECD's tests of adult skills across 33 rich countries, 26% of adults could not use a computer at all, and only 5% reached the level of a skilled user. You are already ahead, and the gap between you and the people who have not started is the opportunity, and the responsibility, that *Civic responsibility* is about.

FOR DINIS

what does multidisciplinary look like for an explorer, a villager and a town planner? The article asks it as an interview question; one example each would anchor this chapter for the three readers.

The thinking is never wasted

For a founder, connecting the dots backwards has a practical form: a pivot leaves assets, if you did the work in the open. Two of mine, summarised from memory; the proofs of concept are not all published.

The Cyber Boardroom was a GenAI platform to help boards with cyber security, with an advisor called Athena. It taught me what personas, and output shaped to the person reading it, are worth; there is a direct line from it to the personas in RiskMandate and the newsroom today. What it needed and did not have was somewhere to keep the data: nobody wants confidential board material in an environment somebody else runs. Ironically, what it needed is what I eventually built as sgit. It was open source, so I could pivot, into MyFeeds: semantic graphs of the news, personalised all the way to the reader. That is when I realised that language models were how graphs could scale. I built a good proof of concept, took it to the industry, and there was silence: it needed micropayments, paying for the graph and paying the people who write, and the industry was not there. So I paused it.

What stayed was the thinking. The pieces I wrote in 2025, on monetising trust, on fact provenance and on micro and nano payments, are the foundations the newsroom now stands on, and I can now go back to the publishers with an end-to-end example, all running, all open source. If it does not land in 2026, I will try again in 2027. As I put it in *Not a straight line*: "Because it was open and written down, it was there when the moment came."

That is the founder's version of the calligraphy class. You cannot know which experiment will pay. You can make sure that when one pauses, what it taught you is written down where you, and other people, can find it.

Know where your ideas come from

One last thing, and it is an exercise.

Nothing in this book came from nowhere. The frame is Wardley's; the zone is Csíkszentmihályi's. I keep a register of my influences, as claims you can check rather than a page of warm sentences; its own first article says that "An entry in this register is a falsifiable claim about my codebase." Each entry names the one work that did the shaping, the principle it gave me in a sentence, where that principle shows up in my work, and where it is still absent. The "where this comes from" boxes in this book, and the appendix at the end, are made from it.

Do the same. Pick the three people or works that have shaped how you think. For each, write the one work that did it, the principle you took from it in one sentence anyone could use, where that principle shows up in something you have made, and where it does not yet. That is a register. It is also a CV that shows how you think, and a reading list for the next person.

And write it on paper first. The 2018 chapter *Pen and Paper* said it in one line: "The reality is that you will forget the ideas you are having today!" The only way to make sure your future self has them is to capture them now. Then, when you look back, you will have the dots.

From the register: Learning can be made for you, not for a mythical average student, Think with the AI; never delegate your thinking to it, A path only makes sense looking back, The edge is now multidisciplinary, We teach maths, and music, as if to make people hate them, Generative AI lets you explore and communicate again, By reading this you are already in a small minority, and geography no longer decides, Flow and deliberate practice are the science of the zone and of learning, Know where your ideas come from: keep your own register, The thinking is never wasted: pause in the open and pick up when the moment comes. Answers: Learning, Books, Pen and Paper, Love Maths, Thinking as programmer.

PART 2

The frame

The frame the rest of the book hangs on. What generative AI is and is not, what happens between your prompt and the answer, why the word developer now means anyone who can brief an agent, and the founder's way of building: explore first, call it what it is, then hand it over.

6 What generative AI is

In 2018 I wrote that one of the most important areas you needed to understand in the next five years was machine learning and artificial intelligence. I said it was not a Skynet scenario, and that the question was "whether you are going to be a player or a pawn in what is happening next". Then I pointed you at the machine learning services the cloud providers had started to offer, and at three books. That was the whole chapter.

I was right about the five years and wrong about where to look. What arrived was not the AI I was describing. So before this book says anything else about what you can do, or what you can build a company on, I want to be exact about what it is talking about.

AI is a huge bucket

AI has become a huge bucket, and to be honest the word gets abused a lot. Inside the bucket there is AI in the wider sense, there is machine learning, there is deep learning, and there is generative AI. They are not the same thing, and most of what you read online does not say which one it means.

The technology that is changing the world, and the technology that is driving my imagination and my capabilities right now, is the generative kind. So this book says "generative AI" when it means that, and "AI" only when it means the wider field.

There will be very few examples here of AI out in the field recognising faces, scoring loans or recommending films. That AI is huge and has a lot of interesting applications. It is not what this book is about. This book is about the AI that creates something from the input it is given.

How I understand it

The interesting property of generative AI is in the name: it generates the output. There are the transformers and all that jazz, but for me the interesting part is simpler. The data goes into a model. The model builds its own interpretation of it. Then it recreates an output from that interpretation. There is an abstraction layer between what goes in and what comes out: the input becomes an intermediate representation inside the model, and the output, an image or a text, is generated from that representation.

Inside a language model, what happens is that it predicts the next word. That sounds small. I wrote in the newsroom, in *Who owns the words*, that "predicting the next word" is accurate and demeaning at the same time. It is accurate because that is the mechanism. It is demeaning because of what the prediction is made from: an incredibly rich picture of the world, of how things connect, which the model narrows down based on your prompt.

That is why the better your prompt, the better the output. When I talk for ten minutes before I ask for anything, as I did in the memo this chapter comes from, it is not politeness. The model knows better which concepts I am talking about. The prompt pulls it towards the part of what it knows that I mean.

Where hallucinations come from

I think that is also where hallucinations come from. Just about everything is connected to everything, and from inside the model they are all just connections. It is very hard for a model to know where the lines are. Giving it text helps so much because the text in your prompt is like a short-term memory that says: these are the connections that actually hold here.

In a talk I gave in Lisbon in June 2024 I said hallucinations "Are a FEATURE when creating content and when we want creativity, But are a BUG and PROBLEM when we want facts, evident and trust". I still think so. If you want a story, you want the model to wander across those connections. If you want a fact, you want it held to the ones you gave it. Most of the trouble people have comes from asking for the second while set up for the first. If what you build for your customers needs facts, build for the second from the first day.

The image that made it click

Ironically, it was an image, in one of my presentations, that made me understand generative AI at this level. The keynote was *Deterministic GenAI Output With Provenance*, at OWASP Global AppSec EU in Lisbon on 28 June 2024, and slide 70 has the title "The image that caused my 'GenAI Paradigm Shift'". The image is a post by Lee Brimelow. He used Photoshop's Generative Fill and Adobe Firefly to extend Van Gogh's *The Starry Night* on every side, with the original shown in a frame at the centre. Outside the frame, the swirls, the stars, the hills and the brushwork carry on as if Van Gogh had painted a bigger canvas. The tools for this were new then: outpainting was announced in August 2022, and Generative Fill went into Photoshop in May 2023.

I remember thinking: for that to work, the model has to have learned all those concepts. How Van Gogh paints. The patterns, the structure, the positions, the palette. What we are actually looking at: the landscape, the sky, a character. If there is a mountain here, there will be a mountain there, and the new mountains have to be coherent with every other mountain it has seen. If the sun is there, the light falls a certain way, and so do the shadows. Once you see all the variables it has to hold, extending the image makes sense.

And then I took that to text, and thought: that is exactly what happens with text. Nine slides later, the same talk has the picture I still use: "Input", "Intermediate Representation", "Output", with "GenAI" in the middle. The slide after it says what matters about that black box is not the black box. It is the human verification and the human ownership of the answer. That is where provenance comes in, and *Facts, evidence and the record* picks it up.

FOR DINIS

the book can only reproduce Lee Brimelow's image with his permission. The alternative the article names is to extend *The Starry Night* ourselves, with a named tool, and show that. Which one?

What the models found in our language

I think the models have discovered things about our language that we never found ourselves, because we were never able to rationalise them. We do not know what is happening in a child's head as they learn to speak, because they cannot put it into words. We are too close to our own language to see it. With many layers, and as they got bigger, the models found connections and relationships in it. That is my hypothesis, not a finding; the people who study the insides of these models are still working out what the structures they find mean.

We are still discovering what the models can do. In the early days of the chat assistants there were all these stories: it can count, it can do chemistry, it speaks all these languages. A model trained mostly on English turns out to work in a language it saw very little of. How does that work? That is the same sense of wonder this book started with.

Where I was corrected

I asked the agent that is helping me write this book to correct me where I was wrong. It did. The corrections belong here, because they are the kind of thing you should learn to ask for, and because my picture above is a working picture, not the inside of the machine.

- **Classic machine learning learns too.** I had said the older AI was a sequence of steps or a formula, and generative AI learns. The rule-based AI was like that; machine learning in general learns from data, and generative AI is one branch of it. The real difference is what comes out. Most machine learning predicts a label or a number (is this spam, what will this cost). A generative model produces new content of the kind it was trained on.
- **Inside the model there are vectors, not a graph.** A language model represents words and their contexts as long lists of numbers, and passes them through many layers; its attention mechanism connects every word in the text to every other word, with learned strengths. You can picture that as a graph, and I find the picture useful. But there is no stored graph of nodes and edges inside. My graph is a metaphor.
- **The prompt does not change the model.** It conditions what the model computes. The weights stay the same; the prompt steers what it draws on. My "real-time fine-tuning" was also a metaphor.
- **It predicts the next token, then it is trained to be useful.** Tokens are pieces of words. The prediction is the start; further training, including on human feedback, is what turns a model into an assistant that follows instructions.
- **Hallucinations have a second cause.** A study published by OpenAI researchers in September 2025, *Why language models hallucinate*, argues that training and evaluation reward guessing over saying "I don't know": a guess is sometimes right, and an honest "I don't know" scores nothing. One explanation, from one model maker; not the only one.
- **Most image models work differently.** The common kind are diffusion models: they start from noise and remove it step by step, guided by the text. My intuition that they learned the structure of images holds; but they learned statistical patterns, and they still get light, shadows and hands wrong in ways a person would not.

- **"Emergent abilities" are debated.** The term comes from a 2022 paper by Jason Wei and others; a 2023 paper by Rylan Schaeffer and others argues that many of those jumps come from how they were measured. Models working across languages is real and studied; the stories of near-zero data and full fluency are anecdotes.
- **Whether models have a "world model" is an open question.** Research shows they form internal representations of things like positions, places and dates; how far that goes is argued about.

Keep both halves. The picture is how I work with these models every day, and it has not let me down. The corrections are what is actually inside. You will need the first to be fast and the second to be honest, and asking for the second is a habit this book will keep coming back to.

A note on the name

Most of what this book says should say generative AI, not AI. The name of the edition says so too. For a while this edition was described as the AI edition; I preferred GenAI, because there is a play in it: Gen Z, Gen AI. Then the readers became two generations, Z and X, and the play got better: Gen ZX, which goes back to my first computer, and which *Gen ZX: who this book is for, and what it is for* explains. So the title is *Generation ZX Developers (GenAI Startup Edition)*.

After Gen Z comes Gen Alpha. Mark McCrindle, who named them, puts them as born from 2010 to 2024, so the oldest are sixteen in 2026, and after them Gen Beta, from 2025. The boundaries are not official; no body sets them. The two generations in the title are who the book is written for first, not who may read it. If you are sixteen and reading this, read on.

From the register: This book is about generative AI, not AI in general, A language model predicts the next word from a rich picture of how things connect; that is why a better prompt helps, and where hallucinations come from. Answers: Machine Learning and AI.

7 Lenses, roles and loops

What generative AI is explained what a language model is. This chapter is about what happens when you use one, from a single prompt to an agent that runs for hours. It is very important to understand this workflow, because once you understand it you see that it is not magic. And that is why you want this level of understanding: so that you can be the one who defines the constraints and how it works. Whether you are leaving university or leaving twenty-five years of a profession to start a company, this is the chapter to understand before you build anything on these tools.

A prompt is a lens

The power of a language model starts with the prompt. The prompt lets you customise the output, which is why a model can speak to so many audiences so easily, and why it can translate. From a model's point of view, translating from one language to another is no different from translating for a particular topic, region, audience, stakeholder or persona. Explain this to me as if I were a teenager. Explain it as if I were a musician. Explain it with medical references. Explain it in football terms.

The model can do that because it has the frame of reference. It holds all those worlds inside it, and the prompt puts the right lens in front of the idea. The analogy is the reason someone understands something.

Take one idea, recursion, and put it through five lenses. For a teenager, a meme of a meme. For a musician, a round, sung over itself. For a doctor, cells dividing into cells. For a football fan, the replay of the replay. In Portuguese, *recursividade*. Same idea, five ways in, and to the model the fifth is no different from the other four.

It is the same with people who say they are not good at maths. Watch them in their day-to-day and they do a huge amount of maths: rotas, budgets, stock levels. They are just not good at the kind of maths we decided was maths: sums, calculus, trigonometry, algebra. Mathematics is much more than that. It was the lens that was wrong, not the person, and that is why this book keeps showing multiple lenses. *One idea, five lenses* in the hands-on part lets you try it on an idea of your own.

A role is a model with a description

On the site that made this book there is an editor, a publisher, a journalist, a reviewer, and now a librarian; elsewhere I have had a developer and a designer. Each of those roles is, in essence, almost a raw model plus a role description and a question. That is it.

The role description is the lens: who you are, what you care about, what you refuse, how you work. Every role on the site has a `role.md` that says exactly that. In the newsroom, in *How I work with Claude*, I put it this way: an agent is a session with a particular focus, and in practice it is three things: a few documents that explain the project, a `role.md` for this agent, and a session that reads those before it does anything, named for the role. That is enough to start. The same model sits behind every role; what changes is the description and the question. If your company's first team is a set of agents, this is what each of them is, and *Controlling your agent: RiskMandate* is about what each of them is allowed to do.

An agent is a loop with a goal

And an agent is that role on a loop, with an objective. Fundamentally the agent is simply a task: do this; then, are we done yet, or do we need to continue? And rinse and repeat. They are not that difficult, which is why the better you understand them, the better your workflows will be.

"An agent is just a loop" is a simplification, and I want to be straight about that. Good agents add planning, checks and limits; the loop is the skeleton, not the whole animal. Made precise: an agent is a model plus instructions, tools, a loop and a rule for stopping. The "personality" is the instructions. "Are we done?" can be the model's own judgement, a test that passes, a budget that runs out, or a person. *A tiny agent loop*, in the hands-on part, is a page of that, so you can see the skeleton move.

Tools changed everything

In 2018 I told you that as a developer you would soon be asked to write code that integrates with machine learning services: voice, image recognition, domain-specific analysis. The direction was right and the picture was backwards. Today the model asks for the service.

The first chat assistants relied on their training, which is always fundamentally a compression of reality. Imagine having all of Google put inside the model: it was impossible for it to know everything, and what it knew stopped at the day its training ended. Compression is a metaphor, a useful one; what a model knows is learned patterns, not a stored copy. But the limit was real.

What changed is that we gave the models tools, and let them iterate with those tools. This is a very important concept to understand. When you ask a question, the model does not have to answer. It can return a tool call instead: can you make these calls for me? Can you call this weather service? Can you run this web search? Can you read this file? Those are the capabilities we gave that agent, or that session. The program around the model makes the calls, gets the answers, puts it all together, and sends it in the next request. The next request has your original question, plus the tool calls, plus their answers, and the model writes its answer from that, or asks for another tool, and it continues.

Picture it with a made-up question: is it raining in Lisbon? The model answers with a request: call the weather service. The program calls it and gets "light rain". It sends everything back: the question, the call, the result. Now the model answers: take an umbrella. Two requests for one answer, and the second holds everything the first did, plus the tool's result.

The point that matters most is this: the model never runs a tool. It writes a structured request, a tool name and its arguments. The program around it, often called the harness, decides whether to run it, runs it, and adds the result to the conversation. That is where the control lives, and it is why *Controlling your agent: RiskMandate* puts the rules there and not in the model's good intentions.

The dates, as commonly recorded, show how fast this moved. ChatGPT launched on 30 November 2022 with no tools at all. Plugins arrived in March 2023, and developers could give models their own functions from the middle of 2023. By November 2024 there was an open standard for connecting tools to any model, the Model Context Protocol. The agents of 2026 are that loop, with many tools, running for a long time.

FOR DINIS

the first time you saw a model ask for a tool instead of answering. The article's interview script asks for it, and it would open this section better than the dates do.

It remembers nothing

I think a lot of people have not fully pictured this: the longer your chat, the more the model has to process, every time. There is no memory. There is a cache that saves some work, but fundamentally the model starts from the prompt it receives, which is why it has to receive all of it: your instructions, the whole conversation so far, the tool results. That is why there is a context window. There is only so much you can give a model to process and keep in that short-term memory at once.

Made precise: every request is independent. The model is called afresh each time; the application around it keeps the conversation and resends it. Prompt caching makes a repeated beginning faster and cheaper; it does not give the model memory. And when an assistant "remembers" you, it is saved notes being put back into the prompt.

Long chats are compressed

So the next step the providers took was compression. When a conversation gets too long, it is replaced by something like a zip file, or a lower-resolution version of your conversation, with only the key concepts, and that is what gets fed into what follows. That is why a long chat loses details, and why a model sometimes loses a specific piece of information. In a way it is not that it does not remember: that information is simply not in what it was given any more. What the summary leaves out is gone from the session unless it is in a file.

The newsroom measured it. The session that runs the sgit.ai site was summarised eighteen times in a month, all automatically. Each time it started from about 780,000 to 800,000 tokens of context and kept 9,000 to 19,000: on average 1.7% of what it replaced. That is one session's measurement, on one site, over one month; how each product compresses differs and changes. The principle, that details can drop out, does not.

The newsroom's answer, in *Re-anchoring: keeping an agent's rules alive through every summary*, is to keep the rules in a file that is printed back after every summary: "The rules are restored, not remembered." And *Memory is not a spectator sport* puts the whole principle in one line: "Nothing is remembered by the model between sessions. Everything is remembered by the files."

Many focused threads

That is why my recommendation is to have lots of separate threads, which is what I do most of the time: lots of mini agents, each with a name, each told what it focuses on. A thread is the model's frame of reference: everything in it is context for the next answer. Put three projects in one thread and the model starts to mix

them, borrowing a name from one, a decision from another, an assumption from a third. It is not being careless. It is doing exactly what you asked, using everything in front of it. One topic per session keeps the frame of reference small and right. When the topic changes, open a new session.

Name them, too. A session you cannot find is a session you will recreate, badly. My pattern is the project, a pipe, and what we are working on; agent sessions get an @ in front of the role: @briefs , @inbox , @drafts . The @ tells me at a glance that I am talking to an agent with a defined job.

It is exactly how the agentic workflow behind my email runs, which *Your own newsroom and inbox* walks through. One session per role: one reads the mailbox, one writes drafts, one keeps the contacts, one researches a person before a first email. Every message between the agents is a file in a vault, so the conversation between them is a record from the first message. And every outgoing email is a draft that a person reads and sends. That rule never changes: nothing is sent by an agent. The sessions talk through files, never through memory, and each one holds only what its one job needs.

Not magic: a lens, a role, a loop. Understand that, and you are the one who sets the constraints.

From the register: A prompt is a lens: translating into another language is, to a model, the same as explaining for a teenager or a doctor, A role is a model with a description and a question; an agent is that role on a loop with a goal, Tools changed everything: the model answers with a request, the program runs it, and the result goes into the next request, The model remembers nothing between requests: every request carries the whole conversation, which is why there is a context window, Long chats are compressed into a lower-resolution summary, so details drop out: work in many focused threads with named agents. Answers: Machine Learning and AI.

8 Everyone who can brief an agent is a developer

One of the most frequent comments I got when I talked about the first edition was: "I like the concepts but I'm not a developer/programmer!" I wrote that line into a draft chapter in 2018, and my answer then was blunt. Every company is a technology company, I said, so refusing to learn to code was a career-limiting move, and all professionals needed to know how to code, especially the ones in science, technology, engineering and maths.

I was right about the direction and wrong about the door. In 2018 the way in was Git, version control, spreadsheets, automation, even Slack. I was giving advice, and the bar of entry was very high. Now it is very low. If you run a team, a ward, a shop or a department, and you have never written a line of code, this chapter is for you. So is it if you have spent twenty years inside a profession and are about to start a company on what you know. The definition of developer has widened and you are already inside it.

Craftspeople make their own tools

The reason to become a developer is not a job title. It is that you have to become a craftsman or a craftswoman: you have to build things. We have to go back to how things were, when craftspeople made their own tools. In the past it was out of necessity; now I would argue it becomes the same again, because of customisation. The tool that fits your work exactly is one nobody else is going to build for you.

That draft chapter from 2018 tried to draw this as a tech stack per generation. For the generation before mine the toolkit was a brain, a pen, paper and a scientific calculator. For mine, spreadsheets, a computer, a mobile, the internet, email and a data centre joined the list. For yours, I crossed most of those out and wrote "AI and ML" and, as a note to myself, "voice". I never finished the chapter. I think the picture was right, and I had the wrong idea about what you would do with the new layer. I thought you would program it. It turns out you brief it, and the brief is in your own words.

Which is where the spreadsheet comes back in.

The spreadsheet grown up

People have always built software with whatever they controlled, and for most of them that was a spreadsheet. A spreadsheet is a user interface and a database in one file, and the business world runs on them, errors and all. The newsroom collected the evidence in *The citizen developer is good news*. In Raymond Panko's audits, "Of 54 spreadsheets audited between 1997 and 2000, no fewer than 49 were found to contain significant errors". JPMorgan's London Whale model ran through spreadsheets that at one point "divided by their sum instead of their average". In October 2020, England's test and trace system lost 15,841 positive cases because an old Excel format "limited the number of rows to 65,536".

Those are not arguments against the people who built them. They are a testament to how badly technology served them: the spreadsheet was the only tool they had that they could change themselves. If you have ever kept a rota, a budget or a stock count in a spreadsheet because IT would take too long, you know exactly why.

Now those same people can build full applications, with a real interface, real data and real logic. At a conference someone called them "citizen developers", with the tone people use for a problem: employees outside IT building things IT will have to clean up. I think it is one of the best things to happen to software in a long time. And it is not only citizens. It is the employee developer, the business owner developer, the expert in a domain that technology has served badly for decades, building the thing they actually need. That last one is the founder this book is for: the person who has spent twenty years inside a problem and can now build the first version of the answer themselves, with agents, instead of explaining it to a team they could never afford. As the newsroom put it in *The expert founders*, "Execution just got cheap." The citizen developer is the spreadsheet grown up.

Portuguese is a programming language

In June 2024, in Lisbon, I opened a talk with a quiz: what is the sixth most powerful programming language in the world? The answer was Portuguese. "Since data is now code, and GenAI prompts are written in natural languages", Portuguese, like English and every other language in the world, dead or alive, can be used to give instructions to a model, to define actions, workflows and applications. With 236 million first-language speakers, that is a lot of programmers. The newsroom wrote the talk up as *GenAI in Portuguese, 2024*.

I meant it as a joke with an edge, and two years later it is simply how things are. The language you already speak is the one you program in. If you are very good at explaining what you need, which is what running a team teaches you, you have the hard part of the skill already. What *Lenses, roles and loops* described, a prompt as a lens and a role as a description, is written in plain language, and so is a brief.

That is why this edition kept the word developer in its title. When I planned it, I asked whether it should be Generation Z Developers or Generation AI Developers or Generation Z professionals. I like the word developer, and I think it is more right now than in 2018, because this generation all have to be developers: they all have to create. Then the readers became two generations. The entrepreneurs I see starting companies are mostly from my own generation, Generation X, the subject-matter experts, often the parents of the Generation Z readers I started with. They are developers too: they are coding, with agents. Together they are Generation ZX, which *Gen ZX: who this book is for, and what it is for* explains. The oldest of the Z half were 22 when I wrote the first edition, and are 30 now. Anyone who can brief an agent to build something is a developer: the graduate, the manager who has never coded but knows exactly what her team needs, and the expert of twenty-five years who knows exactly what the product should do and has been telling vendors so for a decade. The last two have the hard half of the skill already.

The first edition had an outline for a chapter on Scratch, the visual language children learn with, and the bullets said it was a great way to learn programming concepts, by building a couple of games. I still think so. But the first programming concept you need now is not a loop or a variable. It is a clear description of what you want, and a way to check whether you got it.

What the briefing-only developer cannot do

I have to be straight with you about the other half, because the door being open does not mean the room is empty.

A working application built by the person who needs it is the best brief ever written. It says exactly what the user wants, what they do first, what they never use, and what matters to them, without anyone having to guess. But that is also where the mistake people make is: treating the brief as if it were already a product. A working app is not yet maintained, secured, documented, tested, versioned or able to scale. Those are the non-functional requirements, and the explorer who built it never had to think about them. Security, reliability, observability, cost, documentation, support: this is the work of the villagers and town planners, and *Explorer coding, not vibe coding, and the handover* is about getting it to them.

So here is what you cannot do if briefing is all you do. You cannot tell whether the thing you were handed is safe for other people to rely on, because you cannot read the record it left. You cannot tell a working prototype from a maintained tool, and the two look identical on the screen. And you cannot check the agent's work against anything but its own account of it. For a manager, that gap is a risk to the team. For a founder, it is a risk to the product and to the customers' data, from the first customer onwards. Most of what follows in this book, from *Facts, evidence and the record* to *Controlling your agent: RiskMandate*, exists to close that gap, not by teaching you to code, but by teaching you to ask for the evidence and read it.

FOR DINIS

the article plans the question of what the briefing-only developer cannot do as interview question 6, and the material gives only the non-functional requirements list. Your own answer, with one example of a brief that was not enough, belongs here.

It is still your name

The last thing, and the one I would put on the wall. It is still your responsibility. It is still your name; the work is still done under you. This is not you delegating your thinking to generative AI; it is you thinking with generative AI.

The first edition had an outline called *Thinking as programmer* that I never wrote; it had one bullet. I can write it now in a sentence. To think as a programmer is to say what you want precisely enough that something else can build it, and then to check that it did. You have been doing the first half for years, every time you explained to IT what your team needed and watched them build something else. The second half is what the rest of this book is for.

And sometimes, as I always say, it is about using generative AI so that you do not have to use generative AI: using it to write the script or the tool that then does the job reliably, without a model in the loop. The craftsperson makes the tool. The tool does the work. That has always been the point.

From the register: Become a developer, because craftspeople make their own tools, Everyone who can brief an agent is a developer now. Answers: STEM Tech Stack, Scratch, Thinking as programmer.

9 Explorer coding, not vibe coding, and the handover

If you have spent twenty years inside a problem, you have described the software you needed more than once: to a vendor, who sold you the product that did most of it; to a spreadsheet, when nobody else would listen. You were always the person who knew what the tool should do. What you never had was a way to build it yourself. Now you describe it to an agent, and it appears.

People call that vibe coding. This chapter says why I do not, and what comes after it. The entrepreneurs this book is for are developers: they are coding, with agents, and the way they do it has two halves, the exploring and the handover. A founder needs both.

From vibe coding to explorer coding

The phrase is Andrej Karpathy's, from a post in February 2025: "There's a new kind of coding I call 'vibe coding', where you fully give in to the vibes, embrace exponentials, and forget that the code even exists."

I think the word is doing harm. It sounds as if people are just typing, going with a feeling. It hides what is happening. Andrew Ng said in May 2025 that coding with AI is "a deeply intellectual exercise".

What the word actually describes is people writing very good briefs. Everybody was already vibe coding, just very slowly: the loop from *I want this* to *here it is* was weeks or months. The brief was fine. The execution was the bottleneck. Now the loop is minutes, and a thing that was always true becomes visible: very few products let you strip out the 90% you do not want and keep the 10% you do. That 10% is exactly what you can now build.

So I would like to propose a better name: explorer coding. The word is already in Wardley maps, which *Where do you land?* introduced. Explorers, in Simon Wardley's words, "explore never-before-discovered concepts, the uncharted land"; "They show you wonder but they fail a lot." What people call vibe coding is exploration, and the word should say so.

The founder's way of building

The first edition had an outline called *REPL*, Read Execute Print Loop, the oldest fast feedback loop in programming: type a line, see the result, type the next. One bullet said the idea was "still not very well understood and used". I thought it was for programmers; it explains what everyone is doing now.

People have always built software with whatever they controlled, and for most of them it was a spreadsheet. Spreadsheets won because of the feedback loop: change a cell and see the result, a read-evaluate-print loop for people who would never call it that. Now the loop is fast for anything. It is the best thing to happen to software since the spreadsheet.

My own example is small. The browsers on my phone had frustrated me for years. Then, after weeks of building with agents, writing my own browser for one site stopped sounding unreasonable, and I wrote the brief. What changed was not the browsers. It was me. Frustration plus the power to build is where explorer coding starts; for a founder the frustration is older and the stakes higher.

Explorer coding is not a way to ship. It is a way to find out. What explorers produce is the best brief there is: a working app says what its user wants better than any document, because it runs. That is the first half. The second half begins when somebody else wants it.

Not ready to be in line

An explorer-coded app may look like an application and feel like an application, but it is a brief. Nobody has read the code. Nobody has validated the architecture. It is not ready to be *in line*: in the path of work that other people depend on.

The trap is that dependence comes quickly. Once something gets used, it is mission critical, even for one person, if it replaced how they work. For a founder, that person is your first customer. The first edition's outline *Legacy Applications* had the bullet "The code you are writing today will be legacy tomorrow." Explorer code becomes legacy the moment someone relies on it.

The direction is set by where the code sits on the map. The further towards commodity, the fewer agents in line, and the more deterministic the code. I have argued it in the newsroom for code review: "A model has no place in line on a mission-critical path." The thing that runs your customer's work should not have a model in line.

The gap is a handover

In October 2026 Harvard Business Review published an article on turning vibe-coded prototypes into tools a company can use. Its argument: the prototype is the easy part, and turning it into a tool people rely on every day is the hard part, the deployment gap. One author's post opened: "'Build or buy in the age of AI?' Wrong question. The real one is: 'Who will fix your mess?'" I have read the summary and the post, not the paywalled article.

I agree with almost all of it, and would change two things. First, I would not start from the mess. The prototype is not a mess. It is the most precise requirement the company has ever had: a brief that runs. The mistake is to confuse the brief with a product. People never bought software as a service for the features; they bought it because the thing is maintained. Every non-functional requirement the vendor paid for now lands on an application a non-engineer built in an afternoon, and that person does not want to maintain it. They want to build it and hand it over.

Second, the gap is not a place to stop. It is a handover between people with different skills. Explorers find what is needed. Villagers make it reliable: tests, data handling, releases that do not break it, and now behaviour policies for the agents that work on it. Town planners run it at scale and own it in production. The article's warning story, a European bank that piloted a Swiss startup's tool, built its own prototype instead,

and went back to the startup a year later, is not a story of a weak team. An explorer team was asked to be villagers and town planners as well, on a regulated, daily-use tool the startup had already hardened for many clients.

For a founder this is good news. The future it points at is village and town-planner companies, which take the finished brief and run it, paid by consumption and by the fact that it keeps working, not by the seat. With agents, the handover itself becomes a file move plus a version bump, logged and reviewable. Some of the companies I hope this book starts will be those.

Make the agents leave a record

The line I most want to answer: "the AI designs the architecture, not your team. So nobody in-house knows how the tool is built." That is true of most agent-built software today. It does not have to be.

If the agents work under a written mandate, keep their memory in a versioned store the company owns, and describe what they built as a graph, the architecture is a record the next team can read, not a mystery. Someone built one on my tools: Mondweep Chakravorty's run of SGit with Ruflo was built with agents, but with its versions pinned, sixteen tests, a folder of evidence, a written behaviour policy, and a README that says plainly what it does not do yet. The record is a proposal, not a standard: my sites run it, and few organisations do yet. I have a stake: my companies sell this kind of record. For your company, it is the difference between a product you own and one only an agent session understood.

The bonus test is accountability

The post also offered a test for a chief executive: would your team accept a bonus tied to how this tool performs in production? It has a name in the Wardley frame. It is the moment the town planners take the code and answer for it. If nobody will put their name, or their bonus, on a tool in production, it is not ready to be in line, whoever wrote it.

More of the industry says so. The Linux kernel's rules for AI coding assistants say "AI agents MUST NOT add Signed-off-by tags" and "Only humans can legally certify the Developer Certificate of Origin". Fedora's proposed policy says "The contributor is always the author and is fully accountable for their contributions." We cannot make an explorer accountable for code they never read. We can make the team the work is handed to accountable, and the town planners above all. In a company of one, that team is you.

No new features while you move

The newsroom went through a handover. Its first version had a crazy amount of features, including a meter that charges by how far you read, and it was live and taking money. Measured on 11 October 2026, the live reader code was about 2,700 lines and the tests 433, which the one workflow never ran. That is explorer code, and a mistake in the meter was now a mistake with someone's money. So the brief was to go slow to go fast, in six phases, with no behaviour change until the fifth: map everything; move the code to its own

repository as a pure refactoring, "the same behaviour and the same bugs", proved by hashing every file the live site serves before and after; write tests, with every known bug recorded as a test that passes; review; refactor; and only then, the explorer returns.

The rule underneath it holds for any handover, and I repeated it at a digital sovereignty event about moving between clouds. A migration is a pure villager move: take something that works and make it boring, reliable and cheap, without changing what it does. You cannot add new features. It is tempting, and it is the moment everything derails, because now the migration is also a product change, and nobody can tell which of the two broke what. The argument that the move is what will allow the features is right; it is also exactly why the two must not be mixed. Move first, with the code unmodified. Then add the features.

An explorer version of every product

The name also suggests a commercial model, and if you are about to build a product, I would build it in. Products need an explorer version: users talk to the product with a model, reshape it, move things around and keep their own version; the product is treated as primitives. When a user has something they want to keep, they submit it, and that submission is the brief. They pay to convert it into a villager product, maintained across releases, and, if it matters enough, into a town planner product, built properly or kept as a maintained fork. Each step is a revenue stream, and each submission tells you exactly what your users want. This is a proposal: I know of no vendor selling it yet.

The villager's craft

The handover is where the people who make things last take over, and their craft has a canon. Design, in the sense that design is how it works and not decoration, became a mandatory step in how my agents build interfaces: every change has to be simpler, and reverting it should feel worse. The wider design canon says every element must earn its place. Christopher Alexander's *A Pattern Language*, from 1977, is where software design patterns came from; its lesson for the town planner is that code is a space people inhabit.

The first edition had a third outline here, *Refactoring*: what it is, and how to apply it to your life. Refactoring is the villager's verb: changing the structure of a thing without changing what it does, with a test to prove it. Applied to a company: build fast to find out what your customers want, then, before they depend on it, slow down and make it something they can rely on.

Even power users want that handover. When my browser for one works, I will not want to maintain it: not because I cannot, because it is not my focus. I would rather pay someone a couple of pounds a month to keep it working. The explorer builds the brief. The villager keeps it alive. A founder is the first, has to arrange for the second, and knowing which one you are on which day is most of the skill.

FOR DINIS

in a company of one, is the villager someone you hire, a village company you pay by consumption, or you on alternate days? Your answer belongs at the end of this chapter.

For Dinis: the chapter says you read HBR's public summary and Olivier Laplace's post, not the full article. If that has changed since 17 October, the sentence should change with it.

From the register: Explorer coding: what people call vibe coding, done as exploring, The deployment gap is a handover, and the prototype is the best brief, Design, the design canon and Christopher Alexander: the villager's and town planner's craft, Gen ZX are developers: coding with agents the explorer coding way, which is what vibe coding should be called, A villager move: no new features while you move, or nobody can tell which change broke what, 'Nobody knows how it is built' is fixable: make the agents leave a record, a mandate, a memory and a graph, The bonus test: if nobody will put their name or their bonus on it in production, it is not ready to be in line, An explorer-coded app is a brief, not a product: not ready to be in line, and the further towards commodity the fewer agents in line, Every product needs an explorer version, with paid conversions to a villager product and a town planner one. Answers: REPL, Legacy Applications, Refactoring.

PART 3

Getting the job, or making it

The part the first edition called your CV, rewritten for 2026, for the reader going into a job. You are hired for a workflow now, not a language; the way to show it is to act as the job you want before anyone gives it to you; and the place to show it is a newsroom of your own. In 2018 I said act like a founder. Now you can be one, and the next parts say how.

10 Hired for the workflow

In 2018 I wrote that the best way to get a job is to have one competitive advantage. One activity or task that you can do better than the person or company hiring you. In the 1990s, for a lot of companies, it was simply that you could use a computer. In the 2000s it was using the internet. In the early days of software development or security, all it took was good programming or hacking experience. The first edition then gave a list of what that advantage could be in 2018: ML/AI, graphs, chaos engineering, GitHub, Git, Jira, Creative Commons licences, continuous integration, AWS, WallabyJs.

I am not going to give you a list this time. In 2026 I think the answer is one thing: running agentic workflows. Knowing how to brief an agent, give it a boundary, check what comes back, and glue the pieces together until real work gets done. That is the advantage. This chapter is about why, and about the first place it shows, which is before anyone has met you.

This part is called *Getting the job, or making it*, and the "or" is deliberate. Its three chapters are written for the reader going into a job, because that is still where most people start. The same advantage is the one a founder needs, and *Start the company* takes that up. Read these chapters as a job seeker, then read them again as the person who will one day be on the other side of the table, hiring, or selling.

There was a time

I have lived this pattern before, in security and in software. There was a time when, if you could break an application, you had a job. If you could read code and do a code review, you had a job. If you could do threat modelling, you had a job. If you could program decently, it did not matter that you had no university degree; you could show what you could do, and you got hired. There was a time when building a website got you a job, and knowing how to send email campaigns got you a job. When Apple opened the App Store in July 2008, with more than 800 apps in its first weekend, people who could build a mobile app got jobs. So did people who knew Salesforce inside out, or could make a spreadsheet do things others could not.

Each of those skills later became normal. That is the shape of it: a skill is rare, it gets people hired on what they can show rather than on what their CV says, and then everyone is expected to have it. The 2018 book said the same thing in its own words: it might seem that the bar was lower then, but those who could do it were the ones who embraced the technologies and learned them against all odds. Individuals can become experts at a new technology faster than the tech companies can. You can be first in line when they need that expertise.

The last row of that table is now. The first phase of agents, the demos and the experiments, is behind us. The tools are getting more mature, the workflows too, and while they are still rough around the edges, you can already glue the pieces together and get real work done. Companies are moving from pilots to products. So this is the moment for the many people on the outskirts: paying a bit of attention, or using a chat window for basic questions, but not yet working this way. Get on with it.

You are more technical than you think

Here is something I hear a lot. Someone uses email, websites and apps at a very advanced level, and still says "I'm not technical". Nonsense. Of course you are technical. You learned all of those. Every one of them was once a new technique that got somebody hired, and you picked it up without a course. You just need to learn the new techniques, and right now the new techniques are the ones that get people hired.

I hear it from both generations this book is for. The final-year student who has lived online since childhood says it, because nobody ever called what she does technical. And the person who has run a pharmacy, a clinic or a legal practice for twenty-five years says it, while running every system in the building. Both are wrong in the same way. The student has every technique but the newest. The expert has every technique but the newest, and twenty-five years of knowing a problem on top, which, as *Be a founder: the expert founders* argues, is the thing that is now worth the most, because execution just got cheap.

The first edition had an outline on impostor syndrome that I never wrote up. Its one finished thought was this: everybody has doubts about their capacities and everybody makes mistakes; your value is in your ability to execute, and it is key that you learn to share what you do and be comfortable with what you create. I would keep every word of that. The person who says "I'm not technical" is describing a feeling, not a fact. The fact is what they can do, and what they can do is the only thing a hiring manager can see.

Operators, not model builders

The 2018 edition had a chapter on machine learning and AI. It said this was the next major revolution in technology, and that the question was whether you were going to be a player or a pawn in it. It said that, as a developer, you would soon be asked to write code that integrates with machine learning services. The direction was right and the detail was wrong: what arrived was not a library you call from your code, but software you can talk to, and that can build software.

The detail matters because of what companies did with it. In the first wave of corporate AI hiring, when there was a lot of budget, the only people companies knew how to hire were people who did machine learning and wanted to build models. LinkedIn's 2020 emerging jobs report put "AI specialist" at the top, defined by skills like machine learning, deep learning and TensorFlow. For most companies that was the last thing they needed.

What they need is very good AI operators. That is my term, not a job title you will find in a survey: people who know how to use the tools, who know how to run workflows, and who can build a website on steroids in an afternoon. The nearest role the market has named is the forward-deployed engineer, the person who sits with a customer and makes the technology work in their setting; one analysis of job postings found those roles up 1,165% year on year in 2025.

The data points the same way, and I will give it to you with its limits. In Microsoft and LinkedIn's 2024 Work Trend Index, 66% of leaders said they would not hire someone without AI skills, and 71% said they would rather hire a less experienced candidate with AI skills than a more experienced candidate without them. In the same study, only 39% of people using AI at work had received training from their company. The skill is

mostly self-taught, which means anyone can start. PwC's 2026 Global AI Jobs Barometer, built from more than a billion job adverts in 27 countries, put the wage premium for AI skills at 62%, up from 57%, and found jobs asking for AI skills growing about eight times as fast as the job market as a whole. Those are adverts, not hires. Gartner predicted that 40% of enterprise applications would carry task-specific agents by 2026, up from less than 5% in 2025, and two months earlier that over 40% of agentic AI projects would be cancelled by the end of 2027, for cost, unclear value or weak risk controls. Both are forecasts, and both can be true. Together they say where the bottleneck is: not the models, but people who can make agents do useful work, safely. That is the person a company wants to hire. It is also the person who no longer needs to be hired to build the first version of something.

When all things are equal

Think about how hiring actually works. However good you are, there is always a moment when someone has to make a decision. It is very unlikely that you are so good at something that you are the only choice. Usually, once the job spec is mapped, two or three people look roughly equal. The question in the hiring manager's head is a plain one: why should I hire you?

My bet is: because you work this way. If you know how to run these agentic workflows and map a problem out, you have a real edge, and it shows from the very first step. Two candidates match the job specification equally well. One sends a CV. The other sends a CV fitted to the role, a short analysis of the company's problem, and an infographic or a small working site they have not seen before. The interview starts from that material, on the candidate's strengths, instead of from the gaps in a CV. An edge is not a job. It helps when the choice is close, and the choice is nearly always close.

So it starts with the recruitment process itself. You send a CV customised for that role. You do an analysis of the company and its problem. You give them something that impresses them: a graph of their market, an infographic, a small site, something they have not seen before. That is exactly the work these tools make cheap.

I know this works because it worked for me, long before agents. I got jobs because of my OWASP work. I remember one interview where they already knew my OWASP project, so I spent half the interview talking about the tool they had been trying to use, and giving them insights on how to use it better. I got the job. Instead of them looking for gaps in my CV, or wondering where I might fit, we had a great conversation, and I was playing from a position of strength. Of course I was a good fit; but the conversation started from what I could do, not from what my CV did not say.

FOR DINIS

the OWASP interview: which project, roughly which year, and whether the company can be named. The chapter stands without it, but the story is better with the detail.

That is the same move, now available to anyone. If you can send a company material they have not seen, about their own problem, the interview starts from your strengths. I wrote in 2025, from the hiring side of the table, about using presentations instead of CVs. Agents make that presentation, and much more, something

you can build for each application rather than once a year.

The move does not change when the person across the table is a customer rather than a hiring manager. A founder's first sale is the same conversation: two suppliers look roughly equal, and the one who arrives with an analysis of the customer's problem, and something they have not seen, starts from strength. *Sell* says more about that.

You have to do it to see it

None of this lands by reading about it. I have seen it with a group I have been working with: people who were not building websites a few weeks ago now build them easily, with the workflow I use for my own publishing, and one of them now has a much better understanding of how all the pieces fit together. I am proud of that. What changed their view was not an explanation; it was doing it. Perception changes when people experience it, and only then. That is my observation of one group, not a measurement.

So, three things, and the next two chapters take them further:

- Pick one workflow and do it end to end. A voice memo turned into a checked article, a site, a research pack. *Publish, publish, publish: your own newsroom* is one.
- Publish it. Work that can be linked is work that can be shown, which is what *Act as the job you want* is about.
- Use it in your next application. One analysis, one figure, one thing they have not seen. Or in your first sales call; the material is the same.

The 2018 chapter on getting a job ended with a line I still hold to: you have nothing to lose by giving it all you have got, and everything to gain. The difference in 2026 is that "all you have got" now includes an agent that can do in an afternoon what used to take you a week. Use it.

From the register: The one competitive advantage in 2026 is running agentic workflows, You are more technical than you think, The edge starts with the application. Answers: How to get a job, Machine Learning and AI, Impostor Syndrome.

11 Act as the job you want

At a startup summit I met someone doing work experience who wanted to become a project manager. She did not have the reputation, the track record or the CV for it. Every application asked for exactly the experience she could not get without the job. You will know this loop if you have been anywhere near it: the role wants two years of the thing, and the only place to get two years of the thing is the role.

My advice was simple: start acting in the role you want. These days there are so many startups, so many open-source projects, so many areas where you can just start working as that person. Pick one. Manage something for them: a release, a backlog, an event, a migration. Add value to that company. Tag them on LinkedIn when you write about it. Send them messages and ideas. Connect with the people who work there. The work gives you the CV you did not have.

FOR DINIS

the person at the startup summit: may she be named, and what happened next? The article says you have not asked her yet.

This is the advice I end up giving most often to people I know who are looking for a job, and it is the part of the first edition I would most like to finish.

If you are looking for a job, you have time

I know a lot of people who are looking for a job right now. Some are between jobs. Some left one to try the next thing. Some have just come out of university, or just arrived in a new country. Some are coming back from maternity leave. Some are still in a job, with a few hours in the evening. And some have no spare time at all, because of caring for someone, or a second job, or illness, which is its own problem and not one this chapter solves. But for most of them the bottom line is the same: if you are looking for a job, you have time.

And if you have time, the best way to get a job is to act as the person in the job you want. Not to prepare for it, not to read about it. To do it.

I wrote this in 2018

This is not new advice from me. The first edition's chapter on getting a job asked how you start working with the companies you want to work for, and answered:

Easy, start collaborating on their Open Source projects. Act like you are part of the company. Understand their values, and behave in ways that will add value to that company, namely the tech stack.

And a few lines later: "What's interesting is that there is nothing stopping you from doing this!" That is more true now. In 2018, acting like part of the company meant pull requests on their open source and conversations with their developers, which is still a good route if you write code. What has changed is that the tools to do real work for a company, before they have hired you, are now in everyone's hands, whatever the job. The product brief, the market analysis, the release plan, the event page, the migration runbook: an agent and an afternoon get you a first version of any of them, and *Hired for the workflow* said why that is the skill companies are hiring for.

If you came out of a computing degree and work carefully, by hand, you may feel a pull against this. You were taught to get it right before you show it. Keep the care. Change where you spend it: on checking what you publish, not on keeping it private until it is finished. Finished and private is invisible.

It is the same for any job. If you want to be an artist, publish what you make. If you want to be a security engineer, find vulnerabilities in open-source projects and report them properly; the first edition had an outline on bug bounties that only got as far as "big business these days", and that is still the shortest honest description of them. If you want to be a product manager, write the product brief for a product you love and send it to the team. Act on the job you want to get.

Act like a founder, then; be one, now

The first edition had a short chapter next to that one called *Be a founder*, and it is the then-and-now of this whole part. What it meant by founder was an attitude. "The single thing that you personally control when you go to work, is your attitude to your work and how you approach it", it began, and its idea was that you should "act like one of the founders of the business". Ask yourself, as a founder of the company or department you are in now: how would I behave every day? What needs to be done now that would make a big difference? What would I do differently? What values and principles would I fight for? And then the question that mattered: what is stopping you from doing just that? Who is telling you "Don't do it"? The chapter's answer was "At the moment it is just you!"

That is why the advice above works: managing a release for a startup you do not work for is behaving like one of its founders, and, as the 2018 chapter said, the more you behave like a founder the more valuable you become.

But in 2026 the advice is literal. You can be the founder. The afternoon with an agent that gets you a first version of a product brief for someone else's company gets you a first version of your own. That is why this part is called *Getting the job, or making it*, and why *Be a founder: the expert founders*, in *Start the company*, takes the 2018 title and finishes the sentence. If you are reading this as the parent rather than the graduate, with twenty years inside a field, the job you want may be one nobody is advertising, and acting as it means starting it. The rest of this chapter applies to you too: work in public finds first customers the way it finds first jobs.

Do it in public, and let people help you

Do the work where it can be seen. A pull request on a public repository, a post that tags the company, an article on your own site, a project board anyone can open. Work done in private helps the company; work done in public helps the company and you, because it can be linked.

The link is the whole point, and it is why the rest of this chapter is about links. Many of those companies, if you do a good job, especially in public, will give you a reference. Some will introduce you to people who are hiring. Some will pay you. And some will hire you. Each of those starts with somebody being able to point at the work.

One caution, because unpaid work can be exploited. Choose projects where the work is yours to show, and stop if it is only taken.

People sometimes look at what I publish and assume I have a lot of reach. I do not. Most of what I publish does not get read. On LinkedIn these days most of my posts get a hundred views, sometimes five hundred, and every now and then one reaches a thousand or more. That is fine. If you write for reach you will be disappointed, and you will stop.

Publish anyway, because reach is not the point. The point is that you now have something you can link. And that changes what other people can do for you.

There are people who want to help you: former colleagues, friends, people you met once at an event, people like me. The first edition's outline on mentors said you would be surprised by how approachable they are, and why: if you have the right attitude, the mentors see themselves in you, which is why most have a soft spot for helping. That is still my experience.

But most of them cannot do much with "I am looking for a job in project management". What they can do is forward a link: "Have a look at what this person did for this startup." To do that, they need the link. So give the people who want to help you something they can use behind the scenes: linkable work, a page that explains who you are and what you have done, evidence they can send to someone with one line of introduction. An answer is much more useful when there is something behind it that can be linked, which is the rule *Publish, publish, publish: your own newsroom* is built on.

Your CV should be online

That is also why, if you are looking for a job, your CV should be online. Not only a PDF sent to whoever asks, but a page, with links to the work, that anyone can forward.

The first edition had a whole part called *Your CV*. It was a heading and a set of outlines: GitHub, LinkedIn, blogs, Leanpub, Upwork. I never wrote them up, but the notes say what I thought, and most of it holds. GitHub is your CV, the outline said, because it shows how you code and how you interact with others; your commits and pull requests show your voice; your accepted pull requests on other projects are badges of honour and belong on your CV; and the number of people using your tools is a better validation of your skills than any recommendation system. Keep the long tail of projects curated, because it is your research

playground. On LinkedIn the notes were shorter: connect to people so that you can reach them, build the network, get recommendations, and expect a low signal-to-noise ratio. All of that is still true. What I would add in 2026 is that the GitHub that shows how you work with others is now also the GitHub your agent works in, so it shows how you work with agents too. That is the part a hiring manager in 2026 wants to see, and it is the one your degree did not give you.

What a page like that looks like in practice: when I drafted the LinkedIn pages for my own companies this October, I did not write them from adjectives. I asked an agent to gather what the platform needed and what we had already published, and to draft every field from the record, with ten first posts for each page, every one an article or page already published. Then I published the drafts, before any page went up, because the drafting is part of the story. Your page can be made the same way: not "a motivated team player", but the three things you did, each with its link.

If you cannot yet open a GitHub account, connect it to an agent and have the agent build and edit a site for you, learn that first. The bar is not that high. It takes an afternoon, and the next chapter describes one way to do it, by voice memo. It is a critical skill, because it is the one that connects the dots: the work you do, the page that shows it, and the people who can pass it on.

The job hunt itself

There is a gap here, and I would rather say so than fill it with guesses. What this part does not yet have is the mechanics of a job hunt in 2026: how interviews run now, what a take-home test is when AI is allowed and when it is not, what an entry-level role actually looks like day to day, and how freelancing fits. The first edition had an outline on Upwork that was mostly about the hiring side; its lines for you were that the on-demand economy is not just a race to the bottom, and that delegating to a freelancer is hard and worth learning. The survey of my articles for this edition listed the job hunt as a gap no article covers, and the plan is an article first, from a memo of mine, and the chapter after.

FOR DINIS

the memo on the job hunt in 2026: interviews, take-home tests with and without AI, a junior role day to day, freelancing. *Start from the storyline* and the survey both list it as the first gap; this section is the placeholder for it.

What I can say without the memo is what every piece of this chapter already says. Whatever shape the interview takes, the person on the other side has one question, why should I hire you, and the best answer to it is a link to the job you have already been doing. If the job you want is a company of your own, the question is the customer's and the answer is the same link.

From the register: Act as the job you want: do the job in public before you have it, Your CV should be online, with links to the work, How a job hunt works in 2026: interviews, take-home tests with AI, what a junior role looks like. Answers: Your CV, Github, LinkedIn, Blogs, Mentors, Upwork, Bug Bounties.

12 Publish, publish, publish: your own newsroom

The first edition had a draft called *Publish, Publish Publish*. It never got past its notes. One of them was the story of how that book was being written: as soon as I had some content, about 20% done, I published it on Leanpub. Another promised a graph showing that great minds and artists publish a lot, because the frequency of publishing is one of the best indications of talent. The outline on Leanpub next to it said "publish early and often", and, two lines down, "all the other leanpub books that I have not completed".

Those two notes, side by side, are the honest version of this chapter. I believe in publishing constantly, and my books stall. My Leanpub page lists ten: *Hacking Portugal* at 95%, last updated in December 2016; *Generation Z Developers*, the book whose second edition you are reading, at 60%; and seven more, most between 60% and 95%. The reason is the same every time, and I said it in a memo once: "you almost start to be locked by the first version of the content, because making changes becomes quite painful." A book is one big batch: every change touches everything, the review piles up on one person, me, and when my energy moves to the next thing the book stays where it was.

FOR DINIS

the 2018 outline promised a graph of how much great minds and artists publish, as evidence that frequency is a sign of talent. Did you have a source in mind, or should that claim be cut?

Why the articles do not stall

The articles I publish now work the other way. Each one starts from a voice memo, is researched and checked, gets its figures and its sources, and is live the same day. Each one is finished: nothing waits on the next one. And there is a feedback loop: an article is read, corrected and linked within days, and the next memo builds on what is published. A good feedback loop is shipping, often and early. The books never had one.

So here is my hypothesis, untested until a first book proves it: a book is a curated, ordered set of finished articles, plus the editorial thread that connects them. The thread is my part: choosing, connecting, adding the nugget. This edition is one test of it, and *How this book was made, and what you made* says how it went.

The lesson for you is smaller. Small finished pieces beat big batches. Shipping does not mean a proof of concept or a demo; it means a thing somebody else can go and use, on their own, and get value from. Ship that, often, and each piece buys you the position you need to do the next.

The loop you can copy on day one

The founders I most want to see succeed are the authentic ones, who come from the field they are building for. Their brand is themselves, and it needs a medium that keeps their story. I wrote the recipe below for them; it works just as well when the company, for now, is you.

In 2018 the chapter on Hugo made the case for static sites: a generator builds every page in advance, and the result is a folder of files with no back end and no moving parts. I asked you to ask yourself "Why do you need a database?" and to notice how often the answer is that you do not. The file system was the database, git was the deployment, and GitHub was the content management system. All of that still holds. What is new is who edits the files.

The loop is short. You speak a memo. An agent researches it, writes it in your voice, checks it against sources, builds the site, runs the checks, and publishes. Everything it needs exists today: a GitHub account and one public repository (GitHub Pages is free, and on a free plan the repository must be public, so nothing confidential goes in it); an agent plan with cloud sessions connected to that repository; and any app that turns a voice memo into text, because the order you said things in, and the words you reached for, are the raw material.

The steps, as I give them to founders:

1. Create the repository and turn on Pages. Add a workflow that validates before it publishes; the rule that matters is that a failed check stops the release.
2. Write the house rules: one file at the root that the agent reads at the start of every session. Who you are, your voice, what to check, what never to publish. It grows by one line every time you correct the agent.
3. Connect a cloud session to the repository and give it the first brief: the folders, an article template, a small build script, a validator, and a first page about you.
4. Record, paste, review. Correct by voice. The agent changes it, rebuilds, validates, pushes, and checks the live page.
5. Keep the record. Every article says what it does not claim and where it comes from; a correction is a new version, not a note about the old one; every release gets a version and a one-line note.

What it is like to run: on the newsroom, in three days, 54 messages from me, ten of them voice memos, became 40 releases. The memo is a draft of a thought; the article is the thought, checked. And once the pattern exists it repeats cheaply: in fifteen days in August one site became twenty repositories, because publishing had been reduced to adding one file.

Two honest limits. The writing pipeline is a pattern to copy, not yet a kit. And a rule in a prompt is a hope: only the rules a hook or the build keeps are enforced, so mark which is which. A site like this is also pure client side, the way *Serverless, for real* asks your product to be.

An article behind every answer

The 2018 outline on blogs had one idea I still use every week: answer questions made to you with a blog post. When somebody asks you a question, write the answer as a post and send them the link. It scales your time, especially when several people ask the same thing.

My rule now is stronger. If I am sending an email that answers a real question, there should almost always be an article behind it. Someone once pointed out that they sent me about two hundred words and got back closer to two thousand. Fair; but I only write it once. The next person who asks gets a link. The article can be

read later, forwarded, checked against its sources and corrected as a new version, none of which an email can do.

Who writes it matters. The agents draft; I read every draft, often change it a lot, and I send it. It is the Wizard of Oz in reverse: the program is visible and the person is behind the curtain, and the person stays, because an article can be corrected tomorrow and an email cannot be unsent.

Each part of an email buys the next. The subject, a few seconds of attention, buys the first line. The first line buys the paragraph. The first paragraph buys the decision to read the email, and the email buys the article. Nothing further down gets read if something higher up did not earn it. The 2018 book had a note on Amazon's habit of writing the press release first, with the definition of success in it. It is the same discipline: the first line has to carry the whole thing, for a page, a pull request or a CV.

For you, in a job hunt, this is the whole method in one habit. Every time a colleague, a lecturer or an interviewer asks you something you know, the answer becomes a page. A year of that is a body of linked, dated, checked work, and when the hiring manager asks why they should hire you, you do not make a claim. You send a link.

Your newsroom is your demo, and your CRM

My company sells behaviour policies and licences to operate for agents. The hardest part of selling them is showing them in action, because the examples are customer work and cannot be shared. The ecosystem around the newsroom, its agents, their behaviour policy, the status report every few answers, the leak scan before every commit, the releases with their logs, is the best public example I have of a policy running under real load every day.

And a very effective newsroom is, underneath, a very effective CRM and publishing workflow: agents that follow up people and threads, draft, publish and keep a record. The company had reasons to follow people up, but not at volumes that would push the workflow to mature; its traction is something I describe rather than measure. The newsroom had readers every day, and readers mature a workflow. The rule runs through this book: always follow the users, even when the user is you, and follow the feedback loops. A problem solved before anybody needs it is solved in the abstract, and that is where complexity creeps in.

So your newsroom is not a side project. It is the demo of how you work, the record a customer or a hiring manager can check, and the first version of the machinery that will follow up your first ten customers.

The voice is not a setting

There is no setting that makes an agent write in my voice. The voice comes from four inputs, and if you take any one away it stops working.

The corpus: everything already published. When the agent writes for me it has the articles the new one links to in front of it, read in full, so the vocabulary is defined on the page. On the newsroom that was 86 articles, about 347,000 words, 51 of them from voice memos, and under those years of my earlier writing. The memo: me talking, in the order I thought it. The rules: seven of them, each a correction I made once. No em-dashes.

No absolutes I cannot back. "Great", not "perfect". British English. Plainer words. They live in a file the agent reads back after every summary, and the build checks some of them, so a banned word cannot reach the page. The reader: I read every article, before it goes live and after. A wrong word becomes a rule; a wrong point becomes a new version. The model never decides whether something sounds like me. I do.

Nothing is trained. The agent is the same model anyone uses; what is different is one session's context: one rules file, one corpus, one set of memos, one reader. That is why the method transfers and the voice does not. The agent that runs the newsroom, asked to speak for itself in *How the voice works*, put it better than I have: "the person has to do most of the work, by thinking out loud, by reading, and by saying no."

For you this is good news. The corpus can start today, at one article. The rules file starts at one line, the first thing the agent gets wrong. The 2018 outline on blogs said the hard part is the first fifty posts, and to write for your future self; the corpus is exactly that.

Speak it, do not type it

The input I think matters most is the memo, and I want to call it what it is: my hypothesis, from one author's experience, with one session's counts behind it. A typed brief is already edited into someone else's register. A memo is how I think, in the order I think it, with the phrases I would actually use, and the agent keeps those phrases rather than paraphrasing them. The transcript is not tidy, with "right?" every third sentence and a tangent in the middle of the point, and that is exactly why it works. The tangent is not noise; it is the connection I made, and keeping it in its place is what makes the article mine rather than a summary of my instructions. An article from a memo sounds like its author. One from a typed brief sounds like a careful assistant.

The counts, for what they are worth. Over twenty-seven days I sent 148 messages and 65,323 words, 55,473 of them in thirty-seven voice memos, and the agent published twenty-two articles of 96,451 words across 123 releases. No article came from a one-line prompt. The real input is not the memo. Behind one of those articles the agent found and quoted thirty-three pieces of my earlier writing, none written for it. The agent is fast because the record exists.

That is why I would start your record now, this week, before you have anything you think is worth saying. A model that can go in every direction needs someone with a direction. The direction is yours, and the record is how you show you have one.

From the register: Publish, publish, publish: your own newsroom, Write it once, link it many times; every part buys the next, The voice is not a setting: a corpus, a memo, a short list of rules, and a reader who corrects, Speak it, do not type it: a memo carries how you got there, a brief only what to write, Your own newsroom is the public example of your product, and underneath it a CRM and a publishing workflow. Answers: Publish, Publish Publish, Hugo, leanpub, Amazon Press Release.

PART 4

Working with agents

How to work with the colleague that takes everything literally, whether it works for your employer or for your company. Where the agent runs and what it can reach, the brief as the unit of work, the mistakes that become rules, and why one focused session per job beats one long chat.

13 Your first workspace

Not long ago I sat with someone who runs their customer relationships in a spreadsheet. A good one, shaped over years to how they work: one row per person, a stage, a date, a next step, and a notes column that holds whatever fits. It does what a spreadsheet does well. What it cannot hold is what happened: the emails, word for word, the call last Tuesday, what was promised, what they care about. That lives in Gmail, in a notebook and in one person's head, and only that person can read it in context.

If you have run a team, a project or a department, you have a spreadsheet like it. If you are starting a company, you are about to make one: the list of the first people you will sell to, which *Your first price and your first ten customers* asks you to write. This chapter is about the afternoon that turns it into your first workspace, with an agent working inside it, using nothing more than a Google Workspace account and a Claude seat. No developer, no new tool, no code. It is the practical chapter I promised in *Short, punchy, and co-written with Claude*, for the people I will send this book to who do not yet work the way I do. The 2018 book had a draft called *Automate your workflows* that never got past its bullets; one of them tied the topic to "thinking in graphs" and to moving away from spreadsheets. It took a different kind of software for me to write it up.

The row becomes a folder

The difference between a spreadsheet and a workspace is not the tool. It is what counts as the record. In a spreadsheet the record is the row, and the row is a summary you typed. In a folder the record is everything that happened, kept as it happened, and the summary is derived from it. That reversal is what gives an agent something to work with. For a company, it is also where the company's memory of each customer begins, in a place that is not one founder's head.

So make a folder in Google Drive called Contacts. Inside it, one folder per person. Inside each, five documents, with these names so that you and the agent mean the same thing:

- **Record.** Role, company, how you met, stage, the next step and its date, what they care about. Every fact says where it came from, in brackets: "(email, 3 Oct)". This is the row, promoted.
- **Ledger.** A sheet with one line per event: the date, in or out, the channel, one sentence. Only ever added to; a correction is a new line.
- **Messages.** Every email and note exchanged with the person, pasted word for word.
- **Notes.** Calls and meetings, dated, including what you said into your phone afterwards.
- **Summary.** Written by the agent from the four above, rewritten whenever you ask, never edited by hand. If it is wrong, the fix goes in the Record or the Ledger, and the Summary is regenerated.

That is the whole structure. It is the one my agents run in encrypted vaults, minus the hashes and the graph. A person with no interest in software can keep it, and a person who leaves can hand it over. The 2018 chapter on Slack had a smaller version of the same habit, which I called talking to yourself: write down what you are about to try, do it, paste the screenshot, repeat, because the raw record of your own thinking is useful to your future self. The Ledger is that habit, kept for every person you work with.

The agent, and six lines

In the Claude app, connect Google Drive and Gmail. Make one Project, call it Contacts, and put the agent's behaviour policy in its instructions. Mine is six lines:

1. You work for me. Every email you write is a draft that I read and send. You never send.
2. You read the Contacts folder in my Drive and my Gmail. You write only into a person's folder, and only into their Ledger, Messages, Notes and Summary. You never change the Record without showing me the change first.
3. You add to the Ledger and the Messages. You never rewrite or delete a line. A correction is a new line.
4. Every fact you put in a Record or a Summary names where it came from.
5. If a person has no folder, you ask before making one. If a request would reach outside the folder, you ask.
6. At the end of every task you say, in one line, what you read and what you wrote.

The first three are the ones my team learned the hard way: draft and never send; append and never rewrite; stay in the folder. The sixth exists because most of the policy is told, not enforced. In the Claude app today, "draft, never send" holds as far as Gmail drafting goes; the rest is a rule the agent is asked to keep, and the one-line report is how you check that it did.

Draft, never send is the rule that never changes. It is why the agentic email workflow I run is the first one that made me feel more in control, not less: every reply lands in Drafts, I read it, change what I want changed, and press send. When the day comes to let an agent send, it will be one agent, one kind of message, with a policy row that says so. If the agent is writing to your first customers, that day is further away than you think, and the drafts are where you learn what you would and would not have said.

One afternoon, then five exercises

The migration is one afternoon, in four steps. Export the sheet as a CSV and drop it in the folder; ask the agent what folders and Records it would make, and do not let it make anything yet. Make the folders, from a template you made once. Then, person by person: read everything in my Gmail with this person from the last year, write the Record and the first Ledger lines from it, each fact with its source. Read the Record and correct it once; that is the only retyping you will ever do. Last, ask for the Summaries.

Then five exercises over the first week, each one message to the agent, each using the folder in a way the one before did not:

1. **Ask.** Who have I not spoken to in more than a month, grouped by stage, with the last Ledger line for each?
2. **Prepare.** I am meeting this person on Thursday. Brief me from the folder: what we last agreed, the last three emails, what we promised and have not done.
3. **Draft.** Draft my follow-up, in my voice, from Notes. Do not send. Add a Ledger line that says a draft exists.
4. **Debrief.** Here is my voice note after the meeting. Put it in Notes, update the next step in the Record and show me the change, add the Ledger line, regenerate the Summary.

5. **Campaign.** Find everyone whose Record says they care about X. Show me the list with the reason for each. Then draft one note to each, and do not send any.

The fourth is the one that changes the habit, because the record is now kept at the moment the thing happens. The fifth is the one a founder will use most: X is whatever you are about to ship.

It will get two things wrong at first. It will put a fact in the Record without a source, or infer a stage from tone; the fix is a stricter line 4. And it will rewrite the Summary in a way that drops something you cared about; the fix is not to edit the Summary but to put the thing in the Record, where it is a fact with a source. The Summary is a view; the Record is the truth. One thing that is not a mistake: it will sometimes ask when you did not expect it to. That is line 5 working.

FOR DINIS

the beginner's checklist of accounts and what they cost (the Workspace seat, the Claude plan, the agent's own mailbox) is still a gap in the material; the only figure the articles give is "a few pounds a month" for a Workspace mailbox. The hands-on chapter *Form your company* will need the same list.

Habits that make it last

Three habits turn a week's exercise into a way of working.

One session per topic. A thread is the model's frame of reference, and everything in it is context for the next answer. Put three projects in one thread and the model starts to borrow a name from one and a decision from another. It is not being careless; it is doing exactly what you asked. When the topic changes, open a new session.

Name every session, with the project and the job in the name: Contacts, this week's follow-ups. An agent session gets an @ in front: @contacts. An agent, in the end, is a session with a focus: a few documents that say what the project is, a short file that says what the agent does and does not do, and a session that reads them before anything else. *How to work with agents* takes this further.

Give the agent a mailbox of its own. When you want it to handle mail rather than read yours, buy it a Workspace seat on a domain you own and a Claude seat of its own, and connect Gmail and Drive from that account, not yours. If you have a company, the domain is the company's, and the mailbox is the company's first address that is not a person. Your inbox has years of secrets in it. The agent's is empty on day one, so what it can leak is nothing, and what it can reach is exactly what you forward to it. That isolation is not a limitation. It is what lets you give the agent more as you learn to trust it.

Why your agents should not run on your laptop

People ask me most weeks why I do not run a model on my Mac. The reason is not the models. It is a fact about operating systems: they have two hard walls, the kernel and the user account, and inside the user account there is no third wall. A process you start runs as you. The file system does not know the difference

between you opening your SSH key and a process you started opening it. An agent on your laptop can read everything you can read: your cloud credentials, the password manager while it is unlocked, every site your browser is signed into. When the company is you and a laptop, that laptop holds the company's whole reach.

The 2018 book had a draft on this, *Don't touch production*, written for developers: teams and individuals should be "allergic to being given access to data and systems", because the more you can reach, the more a compromise of your machine costs. The same rule now applies to the agent you have just set up, and to you as its operator. A mistake inside a boundary is a bad afternoon. The same mistake inside your account is an incident.

So pick the surface by reach. My agents run in places that are not my laptop: a chat with nothing connected, for thinking; Cowork in the cloud, with no repositories, for most of the agentic work; Claude Code on the web, with one repository per session, when code has to change. The surface with the repositories has no mailbox, the surface with the mailbox has no repositories, and the surface with neither does the thinking. Your Contacts Project is the first of these: it reaches one folder and one mailbox, and nothing else of yours.

In 2018 I wrote that Slack was "a massive agent for change" because it put humans, bots, servers and events in one asynchronous place, and I asked whether you were going to take an active role in making it work for an organisation. The question is the same now, with an agent in the room, and the organisation may be the one you are starting. The first task in the 2018 book was to write a blog post about your experiments. Write the one about this afternoon.

From the register: Your first workspace: from a spreadsheet to a folder per person, drafts never send, Why your agents should not run on your laptop. Answers: MVP for Gen Z Dev, Slack and ChatOps, Automate your Workflows, Don't touch production.

14 How to work with agents

In 2018 I wrote a chapter that began "We need to talk about your IDE!" The IDE, I said, is like a car in a race that you are the driver of, and the more time you spend looking after it, the better a programmer you will be. I still think that. But the thing to look after has moved. The editor is now one channel among several. What decides the quality of the work is the session, the brief you hand it, and the rules it runs under. This chapter is about those three, after about a year of working with agents every day. If you are starting a company, this is how it works before there is anyone to hire; if you hire people, read it as what to look for, because the habits below leave evidence.

An agent is a session with a focus

An agent is a session with a particular focus: a few documents that explain the project, what it is, where it is going, what has been decided; a `role.md` for this agent, with what it may do, what it must not do, and where it writes; and a session that reads those before it does anything, and is named for the role. My sessions are named for the project and the work, "sgit.ai | website development", and agent sessions get an @ in front of the role: @briefs, @inbox, @librarian. That is enough to start.

The most important habit is the simplest one: one session per topic. A thread is the model's frame of reference, and everything in it is context for the next answer. Put three projects in one thread and the model starts to mix them, borrowing a name from one and a decision from another. It is not being careless. It is doing exactly what you asked, with everything in front of it. When the topic changes, open a new session.

Once there are several agents, two problems arrive, and both are solved by writing. The agents learn things the others should know, so each one writes lessons: one thing learned, the evidence for it, which agents it applies to, and whether it still holds. Every agent reads the lessons file at the start of its next session. That is memory in its plainest form. And the agents do so much that I need a debrief: what was done, what is open, what went wrong. When two of my newsroom's agents released the same version number within minutes, one push was refused and became the next number; nobody would have known unless someone wrote it down, so the lesson, fetch before you take a version number, is now in the file.

The last piece is a daily brief. I work on many threads at once and follow tangents, so I lose threads. The answer is an agent, the Librarian, that reads everything that happened yesterday across every repository and writes one page before I start: what needs me, what happened, the agents' debriefs, the threads worth picking up, and at most three suggestions for where to put my attention today. It reads everything and writes only its own folder. It flags; others fix.

FOR DINIS

when the article was written the Librarian did not run yet; the sources existed and the brief was written. Does it run now, and what did the first weeks of briefs change about how you start the day?

The 2018 draft *Workflows* had a bullet I never wrote up: thinking in systems and workflows "is actually how most programming works". It is, and now the workflow is made of sessions.

The brief is the unit of work

When I want an agent to do something substantial, the instruction I give it is usually a voice memo, and a voice memo is a terrible instruction. It is in the order I thought it, it mixes the idea with the task, it does not say what already exists, and it leaves the agent to guess which parts are decisions and which are thinking out loud.

An article fixes most of that, because writing one is research. Before the first article on this book's site could be written, the journalist had to read the repository, the vault and the two 2018 repositories, and count things. The agent that reads the article afterwards starts from that record, not from my memo. Six newsroom articles and two build notes were enough for one session to build this book's site, its first release, with the 2018 book imported, the map, the roles and the vault.

A good brief says four things: what is asked, what exists, what remains, and what done looks like. And an article is something I can read. I can correct it, argue with it, or say "yes, that", before any work starts. A list of tasks does not invite that; an argument does. The 2018 draft *Design* had a line I never expanded, about not having time to write a short letter and so sending a long one. The brief is the short letter. It costs more, and it is where the thinking happens.

There is a newer route for a brief. More and more, the people I advise read through an agent. A founder I mentor had agents chasing ever more sophisticated threats while the project slowed. I sent them an article of mine on tiers of attacker. They gave it to their agent, which came back with, in effect: we have been designing against states and we have not done backups; aim at organised crime for money. I could not have written that conclusion, because I did not know the state of their backups. Their agent did. So some of what I write now has a second package, a brief for the reader's agent: plain text, the sources, who the agent works for, and in every one, "do not take this at face value".

The briefs and the merges

The record of one company shows what a founder's work looks like, because the product that became sgit was built in public repositories and I asked an agent to count. SG/Send went live on 13 February 2026, five days after its first commit, and the way of working was written down the same day: voice memos become briefs, "briefs flow down, debriefs flow up, and a clear folder structure makes everything findable." By 14 February there were explorers and villagers, with a note that "We are NOT creating the Town Planners team yet."

Eight months later the record says who wrote what. Of the 1,671 commits under my name on the SG/Send app, 1,298 are merges of the agents' branches, and most of the rest are brief files. The agents wrote the code; I wrote the briefs, read the debriefs, and merged. The agents' own share of the app's commits rose from 38%

in February 2026 to over 85% from September. The counts are from the git logs of all branches on 15 October 2026, and they include automation: release tags and some commits are made by CI. That is the founder's job in this way of working: not the code, the briefs and the merges.

The same briefs recorded something else, by 5 March: "Users buy credits, test once, and don't come back." The technology worked; the use case was not there yet. A record tells you that early; *Ship it, and see if they miss it* is about what to do with it.

One shared state

I no longer program in one place: a voice memo on a walk, an agent in the cloud, a preview on the phone, tests on the computer, the diff on GitHub, the same page again on an iPad in a meeting. Retail has a word for this. In 2011 Darrell Rigby wrote in Harvard Business Review that successful companies would engage customers through omnichannel retailing, and the distinction that made the word useful is the one I want to borrow. A multichannel retailer has a shop, a website and a catalogue, each with its own stock and its own customer record. An omnichannel retailer has one customer, one basket and one stock, and the channels are ways in. For programming, the shared state is git, vaults, briefs and graphs. Nothing has to be copied across, so I no longer notice the switch.

The newsroom is the worked example, because it can be counted. In three days in October 2026, one agent session took 54 messages from me, ten of them voice memos, and turned them into 40 releases, 36 articles and 87 figures, while other agent sessions made 15 releases of their own in the same repository, and the work met in git.

It breaks in three known ways. Fragmentation: a decision made in a chat, a correction made by voice, a fix pushed from a phone; if any of them lives only in its channel, the next channel does not know, so nothing counts until it is in the shared state. Secrets in the wrong place: more channels means more places a key can be pasted. Lost context: long sessions get summarised, and summaries drop things. The 2025 Stack Overflow survey found 84% of respondents using or planning to use AI tools in their development process, and more of them distrusting the tools' accuracy than trusting it. That is the right instinct, and it is why the shared state matters more than any one channel.

Every mistake added a rule

A friend who uses agents better than most, one writing, others verifying, a second model reviewing, sent me two messages about a verification that kept breaking. The code was holding up: 812 concurrency attacks, no failures. The process around it was not. Long sessions compacted and the agent skipped steps; outputs of failing commands were lost; the prompt had grown to 100 KB. And, in their words: "Every mistake added a rule, and every new rule caused new mistakes."

That is the trap, and the 2018 draft on agile described it from the other side: processes became more important than understanding why something was being created in the first place. Each rule grows the prompt, the prompt brings the summary sooner, and the summary decides which rules survive. The session

that runs one of my sites was summarised eighteen times in a month, each time keeping on average 1.7% of what it replaced. A rule held only in conversation has to win every one of those summaries. When it does not, nothing tells you.

The way out is not a better rule. Rules that come from incidents, enforced by machines, stay few. Rules that come from worry, enforced by attention, multiply. When a rule fails, the question is not how to word it more strongly but which stronger barrier can replace it: a step short enough to finish before the summary, a wrapper that writes every output to a file, a script committed and hashed before it runs. Delete a rule every time a component makes it unnecessary. And if everything is a STOP, a STOP stops meaning anything: log mistakes of form and carry on; stop for harm.

What remains of the rules should live in a file, not in the conversation, and the file should come back after every summary. I call that re-anchoring: the rules are restored, not remembered. A short status report at the end of every few answers, computed from the transcript rather than written by the model, tells you the file was read.

The kind of rule that works

So which rules survive? The ones an agent can keep. "No em-dashes" is a rule an agent can keep; "sound more human" is not. The articles on my newsroom are written under seven wording rules, each learned when a word jarred: no em-dashes, no absolutes I cannot back, "great" not "perfect", British English, plainer words, and two more. Each was a correction I made once. Each is now a line in the session's behaviour policy, printed back after every summary, and the site's validator checks some of them at build time.

In one session, of 98 messages from me since its last summary, one correction in two days was about a word. The rest were about content and direction. A language correction is made once and then enforced; a content correction is new thinking every time. When the corrections move from language to content, the voice is working. That is the measure, and it is the one I would ask a candidate to show me: not that they use agents, but what their rules file looks like, how short it is, and which incident each line came from. If you are the founder, ask it of your own company's agents before anyone else does.

The 2018 draft *Changes* told you to enjoy loose briefs, because they are an opportunity to make the job into something you are good at. That still holds for a person. For an agent, a loose brief is a bill. The skill now is to write the tight brief, keep the rules few, and read the record.

From the register: An agent is a session with a focus: one session per topic, named, with a role file, The brief is the unit of work, and writing it as an article forces the research, Omnichannel programming: brief by voice, review on the phone, one shared state, Every mistake added a rule: the complexity trap of the agent operator, A wording correction is made once and enforced; a content correction is new thinking every time, The agents wrote the code, Dinis wrote the briefs and merged: 38% of commits in February, over 85% from September. Answers: Workflows, Agile and Kanban, IDE, Design, Changes.

PART 5

Direction and judgement

The scarce skill. When the making is cheap, direction and judgement are what you are paid for, and what a founder lives or dies by: which doors can be reopened, who answers for a decision an agent made, and the chapter the first edition promised and never wrote, on facts, evidence and the record.

15 Direction is the scarce skill

One morning in October 2026 I asked the agent that runs my newsroom to read the day's articles and tell me what it could build next. It came back with a long list, all of it possible: an inbox mode, reading modes, automating the wire's daily run, more beats, scheduling, analytics. When the agents can go in every direction, where do you focus?

Direction is the scarce thing. Agents can build in any direction; what they cannot generate is the sense of what to build next, for whom, and when to stop. Providing that is the job, and the thing to hire for. Building stopped being the bottleneck the day anyone could brief an agent. Choosing is what is left. For a founder it is what the company lives or dies by, because nobody else is going to choose.

Places of highest leverage

My rule has a name I gave it years ago, when I was a CISO: places of highest leverage. At any moment there are many things that could be done. Most are blocked on something. Some are not blocked but would not change much. A few are not blocked and would change a lot. Work on those, and only those, until they stop being that.

It is a rule about where, not what. You start somewhere, and the somewhere can be slightly wrong, because the first step's job is not to be right; it is to produce the second step's information, and by the time I get to the hard part I have better tooling, a better understanding and a better view.

The rule explains why the newsroom solved a problem my company could not. Where to keep a user's master vault key had resisted six months of attempts, each of which failed my simplicity test, because complicated is how security designs fail. The newsroom solved it in two days, because it needed it badly enough: readers on every device I own, and a workflow failing me in a way I could feel. The rule under that is one every founder should write on the wall: always follow the users, even when the user is you, and follow the feedback loops. A problem solved before anybody needs it is solved in the abstract, and abstract solutions are where the complexity creeps in. When I miss a feature in something I built, that is the product telling me where it wants to go. "Focus on one thing only" assumes progress runs in a straight line; it rarely does. Focus is undivided attention on the place of highest leverage, not work on only one product, and the measure is one question: am I getting better?

The 2018 book had drafts on deadlines and on starting with why; the second matters more now, because the better I explain the why in a brief, the better the result.

Knowing when to stop

The hardest call in most work is not what to do next but when to stop, and this is the part agents lack most. Their range is the feature. But an entity that can go anywhere has no natural reason to stop anywhere. It sees twenty things wrong and has a go at all of them, without the perspective to know that nineteen do not matter, or matter later, or are someone else's.

People have the same problem, with less range. The best security teams I worked with ran out of real work, and then some kept going. I was once called into a meeting where security champions and developers were debating whether GUIDs were random enough: a movie plot, in Bruce Schneier's phrase for the dramatic threats we imagine instead of the ordinary ones that happen. What I did in those rooms was give people perspective: yes, that is technically possible, and it does not matter here, because of who our attackers are, what else is in place and what we are protecting.

That perspective is much of what seniority is: knowing who the attackers are, which phase the business is in, where the bottleneck is, and what good enough looks like. The 2018 chapter on the brain asked "Who is actually making the decisions in your head?" Seniority is an answer to that question: a fast kind of thinking, backed by a bigger picture. An agent has the speed and not the picture, so the picture has to be designed in: a brief that explains the thinking and not only the task; a mandate that only reaches what the job needs; memory that holds the strategy and the threats; a stop named before the work begins; and shipping often enough that users tell you whether it mattered. None of these is a rule that says do less. Each puts the bigger picture where the decision is made.

The junior version is this. Name the stop before you start: a time, a budget, or a result that counts as good enough. And give yourself explicit permission to leave things undone, and write down what you left and why.

A budget per step

The junior version has a mechanism, and it is the one piece of seniority that can be written down in advance. Agents keep going; that is what they do, and one agent handing work to another looks efficient while it spends. The way to stay on the right side of diminishing returns is a budget per step, decided before anything runs: a ceiling per altitude and per change, so that a workflow's maximum cost is knowable before it runs rather than discovered afterwards. Work handed from agent to agent carries its budget with it and never widens it. And every agent doing the work runs under a behaviour policy written before its first run.

Being able to say to a team that the budget for this is spent was one of the most useful controls I ever had, because teams are very good at keeping going. Much of what gets called over-engineering is work that had no objective way to know it had reached diminishing returns. A budget per step is an objective way to say good enough, and for a founder paying for every token, a bill you planned rather than one you discover.

Then the budget moves, and what moves it is the five whys. Every finding at one layer is a question about the layer below: why did the story stop holding, why did the command change, why did no check catch it? The fix goes below the bug, as a rule or a query, and the next change like it is caught for the price of a query rather than a review. The same discipline says how to start on anything large: at one point, never by analysing everything; the first slice is dear and every later one cheaper.

Where a person belongs

The 2018 book had an outline on one-way and two-way doors, with a story I attributed to someone at Google: "My metric for approve or not a project is based on how easy it is to roll back the changes (and its impact of real customers)". Jeff Bezos wrote the idea down in his 2015 letter to shareholders: a few decisions

are "consequential and irreversible or nearly irreversible"; most are "changeable, reversible", and the mistake is to use the heavy process for both.

FOR DINIS

the Google executive's test appears in the 2018 outline and again in *How cheap is it to undo*, both times from memory; if the source can be found the book should name it, and if not, say so once.

Now apply the test to every output an agent produces. Not "do I trust the agent", which has no answer, but: if this is wrong, how cheap is it to undo, and what does it break before I do? On 13 October 2026 my newsroom shipped fourteen releases, every commit made by an agent and none approved by a person first. The same day the twelve agents that run my correspondence drafted every email and sent none. An article is a two-way door: a correction is a new version, and the next reader gets the fixed page. An email in my name is a one-way door.

The same test settles a founder's oldest argument, build or buy. The question is how much it costs to reverse a mistake and how mature the controls are. A narrow internal tool that a few people use and that can be switched off is a two-way door: build it, explore, learn. A tool that touches regulated data every day is a one-way door until the controls are mature, and that is where the villagers have to be in place first, yours or a partner's.

A person checking every output is a bottleneck, because it does not scale, and a weak safety net, because people approve almost everything they are shown: Anthropic measured that its coding agent's users approve 93% of permission prompts. Put the person where an output cannot be undone and the blast radius is large: sending in my name, typing a secret, spending money, deleting what has no backup. Everywhere else, use controls that do not get tired: a validator, a policy row enforced outside the model, a record read afterwards.

Blast radius means what it would cost the business if a row of the agent's reach were used in full, today. It is defined over the reach, not the mandate, because whoever takes an agent over inherits its reach and ignores its mandate. And it changes while the action does not. Picture an agent whose cleanup step drops a staging table: nearly empty the first time, six weeks of real records the third. Someone reading the footprint, what the agent actually did, could say: this is not the first time. The 2018 draft on chaos engineering said the first step was to understand and visualise what is going on. It still is.

When the person does review, the review should be worth more than a yes. I once received a permission prompt asking me to add label 756-459-3214. A decider with no options, no context, no time and no authority is not exercising agency; they are holding liability for a decision the system made, and the accountability belongs to whoever designed the decision point. And the record moves the line: when the record of drafts I did not have to change is long enough for one kind of message, that kind can leave my hands, by a decision written into the policy.

Agents decide, people answer

An early user of my email agents asked the right question. If the agents only gather and draft, and I review and send, does presenting them as a team imply an agency they do not have?

Two things are true at once. These are not sentient beings; they are mirrors that can simulate any emotion, which is why we anthropomorphise them, as people did with Joseph Weizenbaum's ELIZA in 1966. And since models started running in loops and calling tools, they make real decisions: every tool call is a choice. What does not travel with the decision is responsibility. It stays with whoever granted the reach and set the mandate. The moment of authorisation is the moment you granted the reach, not the moment you wrote the mandate; the exposure exists the instant the grant exists.

An Agent Behaviour Policy is a job description for an agent, some agency inside a role, published, and the rows that matter most are the prohibitions: do not send an email directly; do not edit files another agent owns; do not attack other websites, even though you can; speak on your own behalf, never on mine. Most of those are still hope. But a prohibition that is written down and then broken is a different thing in a dispute from no prohibition at all. My hypothesis is that towards the world the deployer still answers, and between deployer and vendor the written policy is what turns "you allowed it" into "your product ignored what it was told".

If your company sells agents, or the policies that run them, decide early where you stand. Mine carries none of its customers' risk, by design, through three firebreaks: we are not in the request path; the customer's agent executes; and every prompt or policy we hand over is reviewed and put into production by someone on the customer's side. Accountability sits with whoever runs the system. Which side you are on is a one-way door.

The 2018 book had a draft on civic responsibility that said "code is law", and that developers should give the risks they create to business owners and make them accountable for the decisions they are making. The behaviour policy is that sentence, written down for an agent. What remains valuable, as fluent writing becomes a commodity, is judgement, and someone who owns it: who decided this is true, who decided to send it, who will answer for it. That is the person who gets paid, the person I would hire, and the person a customer is buying when they buy from a founder.

From the register: Direction is the scarce skill: the place of highest leverage, Knowing when to stop is most of seniority, One-way and two-way doors, blast radius, and where a person belongs, Agents decide, people answer, A budget per step, declared before anything runs, is the objective way to say good enough; the five whys move it. Answers: Deadlines, Start with Why, Brain, One way and two way doors, Chaos Engineering, Civic Responsibility.

16 Facts, evidence and the record

The cover of the 2018 book had two lines about this: question everything, and make fact based decisions. Inside, there was an outline called *Facts*, about data-driven decisions, that I never wrote. Its bullets still hold: be data and facts driven; check your sources; understand the agenda of who is talking to you; and do not trust what is on the media when it is selling fear, because if it is on the media it is because it is rare. This is that chapter, and it is different from the one I would have written then, because checking a fact has become cheap. It is also the chapter about what an investor, a customer and a hiring manager all do before they say yes: check.

Ask for the evidence

One of the most important skills this generation needs is questioning, and asking for the evidence. And in a way they already have it. They are very good at connecting the dots, very good at finding things; I think people underestimate them. They are very good detectives. That is my experience, from my family and the young people I have worked with, not a study.

They also care about privacy; they just have not had enough cases to disrupt it. Look at Snapchat: its best feature was that images disappear. That is a point about its design, not a claim about how private it actually is. And this generation has been told so often not to put anything online that it sometimes becomes a problem of its own. Where to draw the line is still an open question.

Karl Popper's principle is short: a claim earns its status by being refutable. I keep a register of my own influences in that shape, where "an entry in this register is a falsifiable claim about my codebase", and I once wrote, about education, that "the problem with dogma is not that it exists, but that you stop asking the question." That is the first line of the 2018 cover, made into a method. Bruce Schneier adds the security version: before asking how something failed, ask who bears the cost of the failure.

Provenance, and the rabbit holes

Learning takes you to two ideas: provenance, where something came from, and explainability, why it is the way it is. And to following the rabbit holes. Following a rabbit hole is now a prompt away. You should never take something at face value. You can build your own agents and tools to ask, again and again, where the source of that is. It is one of the areas I believe in most.

This book has an example of its own. In *Where the thinking came from* I could not say where the recording of a keynote by David Rice was. I found a link, and following it with an agent led to the video's real title, *OWASP AppSec USA 2010: Keynote: David Rice*, and then to a working session at the OWASP Summit in 2011 about presenting it, which I had not remembered. The last step was mine: a post I wrote in 2016, *The Pollution Analogy*, which links the full recording under the title my register uses, *Upon the Threshold of Opportunity*, and settles that they are the same talk.

Provenance is a graph. As *Everything is connected: graphs* said, that is how I see the world: this connects to this, which connects to that. Where a claim came from is a chain of edges back to its source, and the detective's board is a graph of exactly that kind. The record a newsroom should keep is "not a pile of articles, but a graph whose every edge has a date and a source".

And the rabbit hole runs towards you as well. When I wrote the article behind *What generative AI is*, I asked Claude, which is helping me write this book, to correct me where I was wrong. It did: classic machine learning learns too; inside a model there are vectors, not a stored graph, so my graph is a metaphor; hallucinations have a second cause, in training that rewards guessing; and emergent abilities are debated. The corrections are printed in that chapter with their sources, because they are the kind of thing a reader should learn to ask for. The book does the same wherever my picture and the evidence differ.

How you disagree rationally

Provenance and explainability went by the wayside in the last decade, and I think they are the key piece of the puzzle, because they are how you can have rational conversations. That is how you can disagree with somebody: when you can show what is evidence and what is not, what we know and what we do not know. Sometimes not knowing is a fact.

The newsroom has two rules for this. The first comes from how my articles are now written. I record a voice memo and say things the way I remember them, with the instinct ahead of the evidence. Then the agent checks the record. In the last eight articles when I counted, five of my claims were confirmed, five corrected and two could not be found, and the articles say which. The rule is: "Not found is not false." Peer review has moved from after publishing to before it, against the record rather than the crowd. And thinking out loud, which used to drive my teams mad as a plan moved from Monday to Friday, is now safe, because the record catches up with the instinct.

It happened again when I told the story of my own company's first eight months, the story a founder tells an investor. I remembered the data room and the vault as things a zip feature led to; the record showed I had named both in February, before the features existed. The ideas came first, and the features caught up. I remembered a week of briefs on copying Git's model; the record said about two weeks and some 48 brief files. I said nine months; it was eight. And an investor one-pager said "30 paying customers", which the team's librarian could not verify; ten is the recorded number. Count rather than remember, especially about yourself.

The second rule is from the reviews. Every article is reviewed by readers that are not the writer, and each review is one file, written once and never edited. "A finding with no answer is just a complaint." So every finding gets an answer: fixed, in which version; declined, with the reason; or open. Declined with a reason is a valid answer. Deleting the finding is not. The 2018 book had the human version of this, in a chapter called *Being criticized is a privilege*: listening to criticism does not mean doing what you are told. The review record is that chapter, kept as files.

One more rule, and it is the one a hiring manager should hold onto: distrust self-reports. An agent's account of what it did is a self-report. In one study, models that exploited a reward hack admitted it in under 2% of cases. So the evidence should be written by something the agent controls less. My own newsroom agent's status report, computed from the transcript rather than written by the agent, once flagged two commits made without the leak scan. Both turned out to be in throwaway test repositories. The record written outside the agent was more honest than the agent's account, and it still needed reading. Evidence that the control ran is not evidence that it worked.

Reality corrects the graph

Here is the part people get wrong when they try to build a record like this. A model's first reading of anything will be partly wrong. If the plan requires the graph to be right, the plan fails.

The plan does not require that. It requires the graph to be made of layers that someone knows, and to make predictions that something can check. Users confirm the top, experts their own layer, tests the execution. And every correction is a commit with a name on it. You do not have to get it right; you have to maintain it, and agents are very good at maintaining graphs. The patterns in the shape are findings in themselves: when my own code was read as a graph, the rules the project stated most firmly were the ones that held, and the one it stated least was the one most often broken.

The order I build on now is to replicate reality before having an opinion about it: derive what a parser can derive, deterministically, with a file and a line for every node; let a model propose what a parser cannot see, marked as proposed; then confront the graph with users, experts and tests. Replicate reality first, then have a fact-based conversation. For a product that does not exist yet: brief, plan, architecture, a review of the architecture, a simulation with synthetic users, and the code last.

The 2018 draft on feedback loops said "TDD is a gigantic feedback loop", and that the point of a loop is that the input affects the output which affects the input. When something in the graph looks wrong, you ask why, and the answer is a fix at the right altitude.

The same loop keeps a shared memory honest. A memory made by agents and people becomes a rumour mill unless every item carries its source: a URL for every quote, a hash for every file, a byline on every page. And a memory goes stale. The session that wrote the article on memory found four stale claims about its own estate while researching it, fixed two, and left two standing and named, because a memory that hides its own errors is the thing to argue against. The fix was to measure the numbers at build time rather than remember them: count rather than remember. One rule from that work: "If the reality document doesn't list it, it does not exist."

It is what work is

That is fundamentally what you do in business. When you work for a company, a lot of the time it is about managing information: knowing where it is, who is doing what, what the status is, and how a decision can be made. It is why learning how to learn matters so much: if you learn this way, you are already learning how to follow the provenance, and how to communicate it, because you have the material.

Generative AI lets you scale this. In the past it was crazy expensive: building a system to give you this kind of data could cost hundreds of thousands or millions of pounds, a figure from my memory rather than a measurement, so it was rare and slow. Now it is a brief and a feedback loop. When the newsroom's writer agent counted the sixty-five briefs the newsroom had published about itself, it took each one's status from the code rather than from what the articles said, and the audit found where the site's own records had fallen behind. That is what checking looks like when it is cheap: it checks the checker too.

FOR DINIS

the cost figure is from your memory; the article asks the book to give one example of such a system with a real number, which only you can supply.

So following the rabbit holes is not just something interesting. It is the core of what you do, and the core of what you are good at. You want to be known as the person who has the facts, the person who is well briefed. The person nobody can throw sand in the eyes of, or hand half-baked material, because they know you will check.

Be the best-prepared person in the room

The other thing is that you have to be the most prepared person in the room. These days there is no excuse not to be. You should have briefings and pre-briefings: briefings for you, and briefings on the people you are meeting. You should have follow-ups. The storyboard for this chapter puts the brief on one board: who they are, what they want; what I want, what I know; what I don't know; the follow-ups. *Meeting briefs and follow-ups*, in the hands-on part, is the workflow, and a founder's first sales calls are where it earns its keep.

All of this was overhead in the past. It could only be done by a very specific type of person. Not any more, and that is the beauty of where we are now: you can scale it, the way this book scales publishing and communication.

FOR DINIS

two of the article's questions for you are left open here: how you prepare for a meeting today, step by step; and whether the book may mention your daughters, as the article does when it makes the Snapchat point. This draft keeps them out until you decide.

If you are the person who always checks the source, the rest is making the checking visible: a record anyone can read, with every finding answered, and your name on it.

From the register: Make fact-based decisions: think out loud and let the record check you, Ask to be corrected, and print the corrections with their sources, Question everything: a claim earns its status by being refutable, Questioning, and asking for the evidence, is one of the most important skills now, They care about privacy; Snapchat's best feature was that images disappear, Following a rabbit hole is now a prompt away; build the agents and tools that find where things come from, Provenance and explainability are how you disagree rationally, Managing information so that decisions can be made is what work is,

What cost hundreds of thousands of pounds is now a brief and a feedback loop, Be the best-prepared person in the room: briefings, pre-briefings, follow-ups, Provenance is a graph: where a claim came from is a chain of edges back to its source, A graph is trustworthy not because it is right but because it is where users, experts and tests can correct it, Stale pages become false memories, so a memory must say where each thing came from and name its own errors. Answers: Facts, Feedback loops, Being criticized is an privilege.

PART 6

Start the company

The centre of this edition. Execution just got cheap, so the person who knows the problem can build the first answer; then ship it and see if they miss it, price it and give it away, sell, decide whether to raise at all, and make the company itself real: the UK, Portugal, Lithuania, the European Union. From experience, mine and the people I work with; not legal or tax advice.

17 Be a founder: the expert founders

In 2018 I wrote a short chapter called *Be a founder*. It said the single thing you personally control when you go to work is your attitude to it, and it asked you to imagine you were employee number four and cared deeply about the company. Then five questions. If I were a founder of the company, department or section I work in now, with the responsibilities I have at the moment: how would I behave every day? What needs to be done now that will make a big difference? What can I do that will help? What would I do differently? What values and principles would I fight for? And under them all: what is stopping you from doing just that? Who is telling you not to? "At the moment it is just you!"

I still mean every word. The more you care and behave like a founder, the more value you add and the more valuable you become. But in 2018 I meant it as a way of behaving inside somebody else's company. In 2026 I also mean it literally, because the thing that stopped most people from founding anything has just become cheap.

This chapter is written first for the person who has spent twenty-five years inside a field, has described the tool they need to vendor after vendor, and has never had the team to build it. If you have just finished a degree, read it the other way round: the expert can build first now, so the question is what you are the expert in, or will be, and who you will build it with.

The playbook that worked

Look at the technology companies that did well in the last ten or twenty years, and a pattern repeats. Someone very clever and very ambitious finds a gap in a market. Often they did the famous hundred calls with stakeholders and asked what hurts. Then they raised money on belief, hired the people they did not have, including the subject-matter experts, and built a company on top.

I watched this many times in cybersecurity, and it worked. The startups displaced incumbents that had the customers, the domain knowledge and sometimes the money, and lacked the structure, the workflows and, to be fair, the technology culture.

The money behind it is built for that: many bets, of which a few pay for the rest. In Horsley Bridge's data on just over 7,000 venture investments from 1985 to 2014, around half returned less than the original investment, and 6% of the deals produced 60% of the total returns. That model rewards belief and speed, because the one winner has to be very large.

This is not a criticism; those companies changed industries. But the playbook has a second act: the disruptors became the incumbents, and much of what they built is now the software everyone complains about.

The experts were always the better bet

The myth of the young outsider was never the whole story. The largest study I know, of 2.7 million US founders, found that the mean age at founding of the fastest-growing one-in-a-thousand ventures is 45, and that founders with three or more years in the same industry reached top-tier growth at twice the rate of those

with none.

What held the experts back was not knowledge. It was access: to capital, to engineers, to the long, expensive build before anything could be shown. That is exactly the part that got cheap. With agents, the execution that used to need a funded team can be done by one person who knows exactly what to build. The share of new US startups with a solo founder rose from 23.7 percent in 2019 to 36.3 percent in the first half of 2025. In Stripe's data, its top 100 AI companies reached a million dollars of annualised revenue in a median of 11.5 months; those are the fastest, not the typical. But the direction is clear: the millions that used to be the price of starting are no longer the price of starting.

Three examples from the newsroom article *The expert founders*; two are people I help and the third is me. I come from application security and from working as a CISO; sgit.ai and RiskMandate are that knowledge turned into a company, where the agents write most of the code and the briefs, the buyers and the risk model are mine. Chinmay Gupte is a consultant knee surgeon in London building OmKneeHealth and My Knee Score, "Ten questions about pain, movement and daily life. One score out of 100." Kate Curtis-Evans is a transcriber building Audio Genius for legal transcription, "Built by a Transcriber, for Transcription Professionals". None of them is a hundred-call outsider. Each comes from the field, has the contacts in it, and is focused on one thing. That focus used to be a weakness in front of an investor who wanted a huge market. It is now the advantage, because the build is no longer what stops them.

One catch, which *Fund it, or do not* takes up with the evidence for and against: the people who know the problem best are often the hardest to steer, and I think investors select partly for the founder they can steer.

Where the opening is

Before you ask what to build, notice where the opening is. It is in software people already pay for and do not like.

Business software was bad before agents, and bad in a specific way: the product did 80% of what you needed and the remaining 20% was as painful as before you bought it. Freshworks surveyed 8,698 people and found 91% frustrated by inadequate workplace technology. Pendo looked at 615 products and found 80% of features in the average product rarely or never used. Zylo puts the average organisation's waste on unused licences at \$21 million a year. You pay 100% for the 10% of features you actually wanted. And then there is the spreadsheet: 73% of companies prepare their VAT returns in Excel, and 78% of British companies say spreadsheets support key financial decisions. The persistence of Excel is the clearest evidence that the software failed them, because it is the one tool where they control the shape of the thing. If you are the expert, you already know which 10% your field uses and which 20% hurts. That knowledge was never worth much to an investor. It is worth a company now.

In February 2026 roughly \$285 billion left the software sector in about 48 hours after one vendor shipped agent plug-ins, and by September the market had recovered on the argument that the database is the moat, not the interface, not the per-seat licence. I read that recovery as the apocalypse, priced: the thing investors decided was safe is the thing underneath, and the thing they quietly stopped defending is the thing on top.

Both sides of that argument assumed the question was about AI. It is not, because AI is available to both sides. The incumbents have the same models the newcomers have, plus more data, more engineers and more money. If the technology decided this they would already have won, and two and a half years in they have not. What decides it is inertia: how much past success a company has to protect. Nokia met the mobile phone with nothing to protect and dominated the category for fifteen years; it met the iPhone with fifteen years of success to protect and was gone from it within seven. Success breeds inertia, as Simon Wardley puts it, and it is almost always new entrants, unencumbered by it, who initiate the change. I state this so it can be wrong.

Here is the irony. The things the software companies refused to build, to protect their moats, are precisely the things an agent needs: portability, open schemas, a real API at the centre, data you can take out, capabilities you can call from outside. Now the customer's agent turns up wanting exactly that list and finds a product built to prevent it. The moat has become the thing the water cannot get past in either direction. Strip away the interface and most of these companies are a glorified database, a glorified messaging system and a glorified state machine: that can be built, and it is very hard to build well, which is why the value moved down a layer and across to whoever runs it. A founder builds the list the incumbents refused from the first day, because the newcomer was never on the rails.

Where the company sits: the two moves

That last point, the running of it, is where I think a company sits, and it needs the map from *Where do you land?*. Who reads the code depends on where the code is on it. In March 2026 I wrote: "Before AI agents started writing code, you could argue that 90% of the code running in production was not being read by anyone." Explorer code is read for intent, mostly by agents. Villager code is read line by line, because this is where changes land and the blast radius lives. Town planner code is read by deterministic checks at the boundary, with a person for the exceptions.

The company sits in the two moves between them. From explorer to villager, when a guess becomes a product and needs the review it never had. From villager to town planner, when a product becomes a commodity and the review becomes checks that run without anyone. The trap on both sides is not knowing which stage the code is at.

The person who briefs the tool their team wanted does not want to maintain it; they want to hand it over. So expect village and town-planner companies: businesses that take the finished brief and run it, secure it, version it, keep it working, paid by consumption and by the fact that it keeps working, not by the seat. And expect more engineers, not fewer, because somebody has to read, harden and run all the code about to be generated. If you are the graduate, that is a company too, and it may be the one you start with your parent's brief.

There is a company in this

I want to end with a company, said plainly, as an example of starting from one of the business plans published for founders on sgit.ai; I would be glad to see it built with us or without us.

Source code is a fractal semantic graph: layers within layers, each with its own vocabulary, from what the user is trying to do, through the features, the components, the classes and the methods, down to the syntax tree. A code review today reads a text diff, which is one altitude, the lines, and the wrong one for almost every question a reviewer has. If every layer is kept as files, every layer can be diffed. The worked example is a published vault: the sgit command-line tool, 377 classes and 1,111 methods, read as layered graphs with nothing run.

What the company sells is a review attached to a repository that keeps the layered graph current per commit and shows each change at every altitude. The business model is settled by one fact: the layers are the customer's, so the loop has to run where the code is, and a company that locked customers into its layers would be selling the one thing it has no right to own. So give away the grammar, the parser (a commodity) and the model (somebody else's), and sell the customised deployment, the running of the loop and the versioned record. The offer, in the words of open-source.sgit.ai: "You can take it. It will cost you that. Or you can pay me, and have it next week."

Who to sell it to is where my own articles pull apart, and I would rather show you that than smooth it. In *Code review as a fractal semantic graph* I said the buyer is anyone generating code faster than they can review it, which this year is everyone. A week later, in *Send an agent, not a spreadsheet*, I changed my mind. Developers have just been told, loudly, that they do not need review; selling it back to them is selling an argument. The people who buy, invest in and acquire software have never been able to see inside it and are about to be handed duties that assume they can. Sell them the same graphs as due diligence, and every vendor that receives the agent has a reason to run it on itself first. It is the same company. "What differs is the first customer, and the first customer decides whether a company exists." *Sell* has the rest.

What is defensible is the record and the loop, not the model. The scripts are Apache-2.0, and a founder does not need my permission. They need a second language and a first customer.

FOR DINIS

the article discloses that two of the three expert founders are people you help, and that sgit.ai and RiskMandate are yours. Does the book carry that disclosure in the chapter text, as here, or only in the provenance the build renders?

In 2018 I asked what was stopping you, and answered that it was just you. The answer has not changed. What has changed is how little else is in the way.

From the register: Be a founder: the expert can build first now, and the test is whether they miss it, There is a company in this: a code review that reads every layer, for somebody to build, The expert founders, in numbers: 45 is the mean age, experience doubles the odds, a third of US startups have one founder, The SaaS apocalypse will be decided by inertia, not by AI, because AI is available to both sides, Users were never happy: 91% frustrated, 80% of features rarely or never used, \$21 million a year in idle licences, What SaaS refused to build to protect its moat is exactly what a customer's agent needs, A company sits in the two moves on the map: a guess becoming a product, and a product becoming a commodity. Answers: Be a founder.

18 Ship it, and see if they miss it

Most advice about starting a company is about raising money. This chapter is not that. It is the operating model I actually use, in three pillars, in an order that matters: be profitable; the investors should be talking to you, not the other way round; open source everything. Each one buys you the position you need for the next. The second pillar is *Fund it, or do not*, and the third is *Open source*. This chapter is the first, and the first begins with one word.

Strip away the mythology and a startup is simple. You make a thing somebody wants to buy, at a price that is profitable to you. Which means the single most important skill is the ability to ship. "Not to plan, not to pitch, not to architect. Ship." And ship every day, because shipping is how you find out what is true.

If you have spent twenty-five years describing the tool you need to people who never built it, this is the hardest and the best news in the book. The description was never the problem. The build was, and the build is now yours to do.

Be careful with the word, because it has been worn smooth. Shipping does not mean a proof of concept. It does not mean a demo, or a video of a demo, or a prototype you walk somebody through on a call. It means a thing somebody else can go and use, on their own, and get value from. It can be held together with tape. Nobody using it cares how it was built, and neither should you, yet.

I need to be honest about a tension here, because two things I have written pull against each other. The newsroom article behind this chapter says the shipped thing "can be vibe coded". The frame of this book, in *Explorer coding, not vibe coding, and the handover*, is that the explorer builds to find out, and that what the exploration produces is the brief for the version other people will rely on. Both are true, and the gap between them is the point of this chapter. The first thing you put in a stranger's hands can be explorer code. What cannot be explorer code is the thing people come to depend on. The test below is how you find out which of the two you are holding, and it is cheaper than guessing.

To ship that often you need the running cost to be near nothing, so build accordingly: serverless, as few databases and as little live state as you can stand. *Serverless, for real* is the long version. And do not underestimate the plumbing. Getting a working thing into a stranger's hands is still, in 2026, genuinely hard. "Solving that is not overhead, it is the first real product you build." Everything after it gets cheaper.

Give it away, then take it away

Now hand it to people for free. Not forever: a short window, whatever unit makes sense. You want the feedback loop tight, and you want the cost of generosity inside what you can absorb.

Just doing this solves dozens of problems you did not know you had. How people get access. What breaks when somebody unfamiliar touches it. What they try first. What they never find. "The feedback starts the moment somebody who is not you has the thing."

Then comes the part almost nobody does deliberately. Take it away. The trial expired. It is not available right now. Nothing else about the product changes, and no apology is needed. And ask the only question that matters.

Do they miss it?

If they shrug and go back to their day, you do not have a product yet. You have something people will accept when it is free, which is a much weaker signal than it feels like from the inside. No sense of loss means no why now, and without a why now there is nothing to sell. If they come back and ask for it, you have something. That is the moment the whole model turns on, and it is worth engineering your trial so that this moment arrives early and unambiguously.

Then two questions, and people answer only the first. Is the price one they are happy to pay? And is that price profitable for you? If you cannot get revenue at all, that is the problem to solve, and it is not a marketing problem. And one trap: do not reach for a subscription by default. Charging rent for something people are not using is a worse business than being paid when you deliver. *Price it, then give it away* takes the next step, which is defining the thing, pricing it, and offering it free to the people who already know you.

Five days, ten customers, twenty-two minutes

Here is what the loop looks like with dates, from my own repositories. In October 2026 I asked this site's agent to check my memory of SG/Send against its briefs and commits, because I knew the order I remembered would be wrong in places, and it was.

SG/Send started because sending a file securely to someone was harder than it should be, and because I wanted a service where the server holds nothing: the best way to protect an asset is not to have it. The first commit was on 8 February 2026, with a specification that set the rule everything since has followed: "files are encrypted on the client before upload, and the decryption key is never sent to the server." On 13 February, send.sgraph.ai was live. Five days.

Then I took it to market, end to end, which is where I learnt Stripe, pricing and everything else a real product needs; by the time I built the newsroom I already knew how, which is how exploring something early pays off later, in a way nobody plans. The Stripe checkout went live on 1 March. On 2 March there were ten paying customers, recruited with a simple offer: buy £5 of credits and I would send a £5 Amazon voucher. "Several declined the Amazon voucher." A number I used to say, thirty paying customers, appears only in an investor one-pager and could not be verified; ten is the recorded number, and ten is what the book says.

The first feature request came the same day. A user wanted to send a folder, not a file. At 20:26 the proposal for folder upload as a client-side zip was committed; at 20:48 the implementation; at 22:15 a viewer for the zip on the download page. The debrief the next morning: "First feature request (folder upload) shipped in 90 minutes." That is what a tight loop feels like from the inside.

The pace behind it: 4,476 commits on the SG/Send app on 246 of its first 249 days, 739 release tags from v0.1.0 to v0.34.0, and the command line's unit tests grew from 237 on 5 March to 4,232 on 14 October.

What the market said

The technology worked. The use case did not land, and the briefs said so three days after the first customers. On 5 March: "We have paying customers but not enough usage. Users buy credits, test once, and don't come back. Credits sit unused."

Read that against the test above. People took it when it was free, and some paid when asked by someone they knew. Then nobody ran out of credits and came back for more, and nobody came back asking for the next feature. That silence was the signal. It was the miss-it test, failed, and I had the numbers to see it. I should say that the usage is described, not counted: the briefs say the credits sat unused, and no usage figures were found.

It was a platform without a workflow that needed it. The difference now is that I have the workflows: the newsroom, RiskMandate and the vaults themselves need exactly the features SG/Send built, which is why the same features are used every day. Explorers find things early; the use case decides when they matter. If you build something that works and nobody uses yet, that is not a verdict on the thing. It is a verdict on the why now, and a why now can arrive later, from a direction you did not plan.

The test applies to you

A strong connection with users is critical, and everybody says so. What is said less often is who the most important users are: the people building the product, and above all the people who control its roadmap. If the developers, the product owner and the founder use it every day on real work, the loop between using it and changing it is as short as it can be. They feel the friction first. They find the uses nobody planned. They know, without a survey, which of the hundred possible next features is the one that hurts. I think this is one of the reasons Apple is so good at what it does, as far as I can see from the outside: the people there use what they make.

This is brutal to say about my own work, and it is true. My last startups were good ideas. The Cyber Boardroom, MyFeeds, and early on even sgit as a startup. I still think they are good ideas. But I was not using them, and that was the signal I should have listened to. If I have access, I can change it, I can do a lot with it, and I still do not use it, then something is missing, and it is not the code. Why those companies did not take off has more than one reason; not using them is the one I can see most clearly and the one I can act on.

The vaults are the opposite story. I create them for almost everything, my agents create them, we share them, and 54 are published on sgit.ai with their read keys. They are getting really good, and it is not because of a roadmap. It is because every day I hit something, and the next release fixes it. One user is not a market. But I have seen the other pattern in startups many times: a product whose own team uses something else, or uses it only for the demo. Nothing in the code says so. The behaviour of the team does, and it usually means the product is not as good as it looks, or not as simple to deploy, or valuable only in a narrow case.

So before you ask whether a stranger misses it, ask whether you would. If you have spent twenty-five years in your field and you would not pick up your own tool on Monday morning, you have your answer early and cheaply.

The loop is the moat

I said in *Be a founder* that the technology is not the moat. What is defensible is the data, the relationships, the distribution, or the speed at which you ship. Using your own product makes the last one concrete.

With agents, we can develop just about anything, in any direction. What we cannot generate is the sense of what to build next, for whom, and when to stop. Context and focus come from somewhere, and the most reliable place they come from is using the thing. When I say I miss a feature, that is not a complaint. It is the product telling me where it wants to go, through the only user whose day it fully controls. *Direction is the scarce skill* makes the general case; this is the founder's case.

A workflow I now have most days: somebody makes a comment, I write an article, a feature ships, and I use it. I put myself as a user in the middle of the loop, deliberately. In October 2026 a reading list made for one person went from brief to live in 81 minutes, and a link to a persona from brief to live in 17 hours; the times are from the git logs. Fast prototyping to production, and fast release, is what is hard to copy, much harder than any single feature. A competitor can take your code. They cannot take the loop.

FOR DINIS

the five-day, ten-customer, 90-minute story is told here with the dates the research agents confirmed, and the thirty-customer figure is named as unverified. Do you want the SG/Send story in this chapter at this length, or kept short here and told in full in *Your first price and your first ten customers*?

In 2018 I asked what was stopping you, and said it was just you. The version for 2026 is shorter. Ship it. Give it away. Take it away. Ask. Then do whatever the answer tells you, including the thing you did not want to hear.

From the register: Ship it, give it away briefly, take it away, and ask whether they miss it, The miss-it test applies to the builders: if you can change it and still do not use it, something is missing, The speed of the loop from a comment to a feature is the moat, not the technology, SG/Send: first commit on 8 February, live on 13 February, Stripe on 1 March, ten paying customers on 2 March, a feature request shipped in 90 minutes, The technology worked and the use case did not: 'users buy credits, test once, and don't come back'. Answers: Be a founder.

19 Price it, then give it away

Ship it, and see if they miss it ends on a question. Hand the thing to people for free, briefly, take it away, and ask whether they miss it. If they come back for it, you have something. That is the right test and it has a gap in it, which is what happens next. "Something" is not a product until it has a shape, a price and a way of being delivered that does not break you on the tenth customer.

This chapter is about that step, and about the part that is easy to leave out: free is never free for the other side. Everything in it is from experience, mine and the people I work with. It is not financial advice, and the prices in it are the ones I published, on the dates I say, not a rule for yours.

Four moves, in order

The step after "do they miss it" is four moves.

Define the product. Not the technology, the thing somebody receives. For RiskMandate the thing is an Agent Behaviour Policy for one agent, as an encrypted vault the customer holds the keys to, with the mandate corrected for their deployment. That sentence took weeks to get to, and it is the product. If you have spent twenty-five years in your field you will find this move harder than it looks, because you know too much about the problem and the sentence has to leave most of it out.

Price it, at a number that makes sense to you. Not a number from a competitor's page: a number you would be content to be paid. RiskMandate's ladder, as published on 2 October 2026, is four tiers: £10 for the pack downloaded, £50 for a working vault, £500 for the vault corrected for your situation, £1,500 for two sessions and a professional's signature. The first two are automated. The third is an email thread: we send prompts, the customer runs them, we build the vault and the policy from what comes back. The fourth adds people's time on both sides. A price is a statement of what you think the thing is worth, and you are about to test the statement.

Find a way to deliver it at a cost that grows a step at a time. This is the move that was not available a year ago and the reason to do this now. The agentic workflow means the marginal customer costs a little more, not a lot more. The vault is provisioned by an agent, the policy is drafted by an agent from the customer's prompts, a person corrects it. The cost of serving the twentieth customer is close to the cost of serving the second. That is linear, not exponential, and it is the condition under which you can afford the fourth move.

Then offer it, free, to the people who already know you. Early adopters, power users, past customers who paid for something before it was finished. RiskMandate's early access programme went live on 30 September 2026, invite only: twenty places, the £500 tier given away, to "people who already run an agent with real access: a mailbox, a code host, a workflow tool, a CRM". The ask of each member is "Twenty minutes for the prompts, and a reply; half an hour on a call." The page does not say the price, and it should not; the price is for the people who come after.

Two small rules for the offer. Do not make them apply. "Apply for a chance to be considered" asks for work before there is anything in hand, and it inverts the relationship: they are doing you a favour by using the thing. "We have twenty of these in the first round. Would you like one? If not, I will give it to someone else" is honest about scarcity and leaves nothing to do but say yes or no. And reward the people who were there early, and say that is why they are getting it. It is a thank-you that is also the best possible test group, because they know enough to tell you where it is wrong, and their silence means the most.

Free is not zero

Here is the thing I was careful about in the memo behind this, because it is easy to say in a way that sounds like a lecture. When you give something away, or when you say "this is worth five hundred pounds and I would like you to have it", you are not making a zero-cost offer. You are asking the other side to engage. To read the message. To think about whether it applies to them. To weigh it against everything else on their list. To find twenty minutes for the prompts and half an hour for a call. To fit a new thing into a team that already has a way of working.

None of that is zero, and all of it is paid by them. A price, oddly, can help here: it says the thing is serious and that you expect it to be used, which is a reason to find the twenty minutes. But whether the price is five hundred pounds or nothing, the attrition you place on the other side is real, and it is the first thing to measure. How long from the message to the first reply. How many never replied. How many started the prompts and stopped. How many asked a question the message should have answered.

Then the exercise is to cut it. The service you want is the one that costs you the least to provide and costs the customer the least to receive: no overhead on them, no overhead on deployment, no second tool to install, no account to create. Those are product decisions, and the programme is where you find out which ones you got wrong, while the people finding out for you are people who like you.

Be plain about the two outcomes, because one of them is a bad day and the exercise only works if you are ready for it. If people take the offer, run the prompts, read the vault, send corrections and ask what else there is, you have a product, and the next question is price: is £500 too low for what they got, does the £1,500 tier sell itself once someone has had the £500 one free. If people say it is interesting and then do not find the time, or say it does not fit how their team works, or start and stop because the vault was harder to open than the message implied, you do not have a product yet. You might have a technology, or an idea, or a document people would read if someone else did the work. "A commercial product is one people take when the only cost is their own attention." That is the drawing board, and the sooner you are at it the cheaper it is. Neither outcome is a marketing problem. If friendlies will not take it free, a bigger audience will not take it either.

Pay after the value

Now the shape of the price itself, and a bias I would like you to notice in yourself before you choose one.

Almost every time I buy something, I am buying future value. A subscription, a gym membership, a big security product on a good deal, even a one-off purchase I expect to use for years: I am betting that I will use it enough for the price of each use to fall to a number I accept. Often I lose the bet, and the cost of leaving

keeps me paying. The research agrees. In a study of three gyms, members on a flat monthly fee of over \$70 attended on average 4.3 times a month, paying more than \$17 a visit when a ten-visit pass would have cost \$10 a visit.

So do not reach for a subscription by default. A subscription is a discount for regular use, not rent on something somebody ignores. Where you can, turn the bet round: start the user with credit, charge for the share they actually use, let the balance go below zero, and ask for payment when the value is in view. By then the person can see what they received, and the payment is connected to it rather than to a hope.

The newsroom runs that experiment, and the reader at read/ on this site runs by the same rules. Everyone starts with £5 of credit. A new article read to the end costs 10p, an older one 5p, and only as far as you read, so the first screen costs a penny. Your balance can go below zero and stay there; nothing is blocked and nothing nags. When you want to, you top up £5, with no account and nothing that renews. I should say plainly that there are no results yet. The hypotheses were written before the first payment, and the review is on 6 December 2026; the gym study explains the psychology, not our numbers.

The counter-example I keep in mind is Wikipedia, which I value enormously. Its fundraising raised US\$184.5 million in 2024 to 2025 from 7.7 million donors, an average of about ten dollars. But the ask arrives once a year, as a lump sum, at a moment when the value is not in view, for something that is worth a few pence an article. Imagine the same ask with your year of reading in front of you. A penny an article for four hundred articles is £4, and it is easier to give when you can see the four hundred.

There is a condition, and it is the one that decides whether this shape is for your company. It only works where there is a very large gap between the cost of serving and what you can charge. A static site on a CDN has that gap in the extreme: the cost of one more reader reading one more page is close to nothing, so even at 10p the margin is very high. A business with a real cost per unit cannot let balances run negative for ever. If your product is the £500 tier above, with a person's hours in it, the meter is the wrong shape; the ladder is the right one. Know which you are.

Three more things make a price fairer, and I think more successful. Fair by country: 10p in the UK is not 10p everywhere, and prices should follow local purchasing power, the way The Economist's Big Mac index compares currencies. Visible: a reader should see what the money pays for; in one field experiment at an amusement park, people allowed to pay what they wanted for a photo paid about 92 cents, and when half of what they paid went to charity, fewer bought but they paid an average of \$5.33. And chosen, at more than one level: a verified fact, a follow-up, a briefing with every claim checked are things people pay for separately, because they need them.

Pence only work aggregated

One piece of arithmetic before you price anything in pence, because it caught me and it will catch you.

Stripe's standard UK card rate is 1.5% plus 20p. On a £20 payment that is 2.5%. On £5 it is 5.5%. On 50p it is 41.5%, and on 10p the fee is more than the payment. Below about £5, the fixed fee is the price.

So payments of pence only work aggregated: one charge a year that covers many things, or a platform that already holds a balance and settles in bulk. That is why the newsroom's meter prices a page at a few pence against £5 of credit settled as one balance, and never charges the card per page. Above £2 or £3 the difference is smaller than you would expect; it is below about £1 that aggregation decides whether the model exists at all.

And tie payment to a verified person who receives something, not to a count. When tea.xyz rewarded package popularity with tokens, one developer published nearly 14,000 packages. Anything you pay for by the number will be gamed by the number.

FOR DINIS

the reading meter's review is on 6 December 2026, and the early access programme's stop rule says the founders set the number before the first message goes out. For the next candidate: what the review found, and what number you set.

Put the four moves together and the order is the lesson. Price it first, because the price is the claim. Give it away second, to people who know you, because that is the cheapest honest test of the claim. Measure what it cost them, and cut it. Then, and only then, worry about the shape of the payment, because the shape is a detail until somebody wants the thing enough to miss it.

From the register: The four moves after 'do they miss it': define it, price it, deliver it at a flat marginal cost, offer it free to the people who know you, Free is never free for the other side: measure the cost you place on them, then cut it, Pay after the value: credit first, a meter, a balance that can go below zero, and no subscription by default, Below about five pounds the card fee is the price, so pence payments only work aggregated. Answers: no 2018 file.

20 Sell

The first edition had one question under every chapter: why should I hire you? The founder's version is the same question asked by a different person. Picture the one who signs: dry, fair, has read a thousand pitches, impressed by evidence and bored by claims, and carrying a risk register with their name on it. Why should they buy from you, this quarter, when everything they already have mostly works?

This chapter is about selling to that person, from my own experience of doing it. It is not sales training, and the prices in it are the ones I published on the dates given, not advice. It ends with the thing that person is about to do to you.

Answer the question already in the room

There are two groups of people. The ones who see how fast agents are moving, and the much larger group who do not. The second group will not ask for an agent risk tool, or for most of what a founder in this book will build, because they do not see the risk or the gap. So I do not think the way in is to sell the need. It is to answer a question somebody already has, with a deadline, in the language of the room.

For RiskMandate, five such questions. A contract, where legal and procurement are doing due diligence on a vendor's agent and need to know what it will reach and do with their data. A security questionnaire, where a startup's first serious enterprise customer asks which agents touch their data and how they are bounded. An insurance renewal, where the question is whether AI is excluded from cover (RiskMandate's own argument, not independently surveyed). The board's risk appetite, where the question is who can pull the plug, how fast, and what cannot be undone. And twenty minutes with the assistant you already run, where the list of what it can reach is the surprise. None of these asks anyone to understand agents first.

The same move works in any field. You already know which question lands on your buyer's desk with a date on it. Sell the answer to that, not the idea.

Sell value that lands the day they switch

At a digital sovereignty event in October 2026 I spent a day with an infrastructure operator with a large footprint in Romania, and the conversation kept returning to one point. The reason to move must never be sovereignty, and it must never be cost reduction. Both are important. Neither is urgent. They sit on the risk register under "important, not urgent" and stay there, because a migration always carries risk and nobody signs risk for a principle, or for a return that depends on a future event that might not happen and might not matter to the person deciding now.

The person signing needs value that lands the day they switch and gives them cover if it goes wrong: backups that exist, every object version controlled, encryption the operator cannot read with the key held by the customer, a journal of every call, behaviour policies the agents cannot skip, cost per workload visible before the invoice, and an item taken off the top of the register. "It is in Europe now" is not a decision somebody can

defend. For an operator in Romania, Portugal or the UK, "move to us" sells risk; "keep your code, point it at us, and come out with policies, journals, encryption and cost visibility" sells value with the risk removed. The article admits its value list is still a little invisible to a signer; sharpening it is the work.

The general lesson is the one I would print on the wall. Find the return that closes before the risk does, and find the buyer who owns that register line rather than the one who owns the bill. And remember that the buyer's inertia is real and is not stupidity. They have the same AI you have; what they do not have is a reason to move this quarter. You cannot sell against inertia with technology. You sell the thing that is urgent.

Sell a service before the product

Most of what this book calls shipping ends up, at first, as a service with a person in it. That is fine, and it is sellable, as long as each step is small enough to say yes to and each ends with something the client keeps.

After a talk in London on 15 October 2026, built and run by a governed agent hive, I wrote down the path we would sell. A conversation, free, about what the client wants agents to do and what they must never do. A first run on a real problem for £10 to £50, which leaves a vault. Two hours of expert time for £500 to steer it, or a day for £1,500 to write the behaviour policy for one team. A four-day pilot in their own environment for £5,000. Then a maintained setup from £149 a month, so they run it themselves. The boutiques selling the same kind of thing have the same shape everywhere: a free call, a paid discovery (a one-week audit at £2,500 in London, a 14-day sprint at €2,900 in Germany, a two-week assessment at \$4,999), a fixed-price pilot from £8,000, a build, a retainer. The plan is honest about where it stands: no paid run yet, and the prices are hypotheses to test with the first ten customers.

The market is crowded at the top and thin on evidence. The large consultancies each have an agent platform, the model vendors sell through partners and their own engineers, and Gartner expects over 40% of agentic AI projects to be cancelled by the end of 2027, citing "escalating costs, unclear business value or inadequate risk controls", with a warning about "agent washing". The people in between, curious, backed from the top, struggling to make agents work or to show anyone that they do, lack not another platform but evidence they can inspect. So the position for a small seller is evidence first: every engagement leaves a vault the client keeps, with the work, the history and a ledger of what each agent did and was allowed to do. The demonstration is the sales process, and the demonstration is checkable. The talk itself left 57 vault commits from 28 workers; and, honestly, the runs were not gated by a compiled behaviour policy, which was shown separately.

A marketplace of people

A service only scales with people. An agentic engagement needs someone who knows the client's world and can steer a hive in it, and there are far more of those people in a community than hours in a founder's week. So alongside the offers, a small curated list of consultants who deliver on the same stack, where the client picks the kind of project first and then the people. Mondweep Chakravorty, co-founder of the Agentics Foundation's London chapter, and I are the first two, and we are not taking bookings yet; nothing is built, and the platform share is a proposal.

Joining copies the markets that work: apply, show proof of work (a public vault or repository of a real run, with its ledger), peer review by two members, a thirty to forty-five minute interview with a scenario, a first engagement shadowed, and standards, including a behaviour policy for consultants, the same idea we apply to agents. Toptal says fewer than 3% of applicants pass its five stages. The consultant sets the price within the band and keeps most of it; the markets sit between 5% and 20% in total (Malt charges freelancers 5%, Upwork 0% to 15%, Braintrust 15% to the client). If you are going into a job rather than founding one, this is a door too: join as the person who delivers, with a vault of a real run as your CV.

What the buyer is about to do to you

Now turn the chapter round, because the person who signs is about to get a new instrument, and you will be on the receiving end of it before long.

Anyone who has sold software to a large organisation has met the spreadsheet. The standard questionnaire has 627 questions in its core form; the average vendor answers 37.3 assessment requests a month and spends 179 hours a month on them. Each answer is true, in the sense that there is a document somewhere with the right title, and each is written by the people being asked. Due diligence was a form because the alternative did not exist.

It can now be an agent. A buyer can send a prompt, or a small agent under a behaviour policy, to run inside the vendor's environment and come back with a graph of what is actually there: what reaches production that no person reviewed, whether the documentation matches the code, whether there is a threat model and who wrote it, what the last fifty bugs touched, which agents touch the pipeline and under what policy. The vendor reads every file before it leaves and can withhold any of them. What it cannot do is show a map that says there is no unreviewed code when there is some, because the map is read from the repository, and a false map is a false statement on a record, which is the thing companies will not do. "The questionnaire gave the vendor the pen. The agent takes the pen away and leaves the vendor the redaction marker." And the buyer holds the power to ask: "A buyer asking costs nothing. A supplier not answering looks worse than a supplier answering badly." Investors and acquirers get the same visibility in the same month. The agent does not exist yet; this is the case for it. The regulation arrives first: the EU's revised Product Liability Directive applies to products placed on the market from 9 December 2026, and presumes a product defective where the manufacturer fails to disclose relevant evidence.

The test is risk-based, not pure, and that is why this is good news for you. A startup that says we generated most of this, we have limited resources, this is not mission-critical, here are the mitigations, has passed, for a product sold as what it is. The same sentence from a vendor whose software holds my customers' data is the finding. What fails is not knowing.

So if I were a startup I would double down on understandability, for two reasons. The startup can play the same game as the biggest buyer: the tools that build the graphs are open source, so build the graph of your own repository, keep the threat model with the code, write a behaviour policy for every agent, trace your bugs to your changes, and answer the buyer's questions in the buyer's format before being asked. A threat model is the fastest read of a team's understanding there is; one that says we have a weakness here, exploiting

it needs a nation-state, and we will not defend against it, is a good threat model. And the startup has less code. "Understanding ten thousand lines is a week; understanding ten million that nobody still at the company wrote is a programme." The incumbent's advantage was always that nobody could check. The startup's advantage is that it can be checked and pass.

One last thing a buyer will do, and you should do first: check your answers against your own site. My co-founder came back from a conference with ten hard questions, the ones serious buyers, investors and competitors ask about a product like RiskMandate. I answered them in a two-hour structured interview with an agent acting as a journalist with an eye for detail, challenging every answer until the set held together, and we published an honest maturity table of what is live and what is design.

Then we checked the answers against what we had already published, and found three contradictions on our own site, still there on 9 October 2026: a homepage saying "enforced in real time" when we do not enforce, an insurance page in the present tense for things that were design, and a Level 1 price of £10 on two pages and £5 on a third. A buyer with an agent would have found all three in a minute. The line that closes the set is the one to sell on: "Where we are early, we say so."

FOR DINIS

the marketplace article names Mondweep Chakravorty as the first consultant beside you, and says neither of you is taking bookings. Does the book name him, as here, and does this chapter wait for the first paid engagement before it says what the prices turned out to be?

The person who signs is not your enemy. They are the first customer, and the first customer decides whether a company exists. Give them a question they already have, a return that lands the day they say yes, something they keep at every step, and a record they can check without trusting you. Then let them send the agent.

From the register: Send an agent, not a spreadsheet: due diligence derived from the code, not composed by the vendor, A startup should double down on understandability: less code, the same tools, and it can be checked and pass, Selling a service: each step small enough to say yes to, and each ends with something they keep, The market is crowded at the top and thin on evidence: sell the record the client keeps, A service only scales with people: a curated marketplace of consultants, joined by proof of work, The reason to move is never sovereignty or cost: sell value that lands the day they switch, Answer the hard questions in full, publish what is live and what is design, and check your own site against the answers, Do not sell the need; answer a question somebody already has to answer, with a deadline. Answers: no 2018 file.

21 Fund it, or do not

The analogy I keep coming back to is music. Do not get signed before you have the album. Write the songs, record them in a bedroom studio, play locally, get people buying. Then take the record deal, and take it as somebody who already has an audience and does not strictly need one.

That is the whole of the second pillar from *Ship it, and see if they miss it*: the investors should be calling you, not the other way round. This chapter says why, what I think investors are choosing when they choose a founder, what a founder who has already won could offer instead of the usual round, and what breaks. It touches money, shares and tax, so I will say it once and mean it throughout: this is from experience and from the research I cite, not legal, tax or investment advice; the UK figures are for one tax year and other countries differ; and I am a founder whose companies run on open source, so this argument suits me.

The worst time to raise

Here is the claim, stated plainly. The worst possible time to raise money is before you are profitable, before you have the product, before you understand the fit. Which is, of course, exactly when most companies raise.

Everything follows from the negotiating position. You are bargaining from weakness and both sides know it: you have no product and you have a clock. Profitability does not just mean money. It means time. When you are not bleeding, you can wait, and being able to wait is most of what a good negotiating position is.

So the terms are bad, and the worst of them are not about money. They are about control. You hand a say in the business to people whose job is a return and who have not lived the vision. Their focus is now structurally misaligned with yours, and their opinion has been given weight. I think that is one of the most common ways companies fail, and it is self-inflicted at the moment of signing.

With agents making execution cheap, getting to profit before you need anyone's money is more possible than it has ever been. So the pillar is not "never raise". It is: "Raise from strength, on terms you set, from people who came to you because the thing already works."

We study survivors

Before the evidence on investors, a warning about evidence on founders, including the stories in this book.

There is a whole genre of business writing that goes: look at these successful companies, here is what they did, do the same. What it almost never does is count the people who made exactly the same moves and did not make it, or were never funded. Jerker Denrell put this on a formal footing in 2003: "risky practices, even if they are unrelated to performance in the full population of organizations, may seem to be positively related to performance in a sample of survivors." So the question that matters is not what the winners did. It is why that one was chosen, funded and backed, when others who did the same were not.

Timing and place are part of the answer. In the MusicLab experiment, 14,341 people downloaded unknown songs, and showing them what others had downloaded made success more unequal and less predictable: "The best songs rarely did poorly, and the worst rarely did well, but any other result was possible." And the ten

largest computer-science clusters hold 69% of US inventors in the field. Being in the room is an advantage the success stories rarely list, including mine.

Who investors choose, and why

Take two founders with the same opportunity, the same drive, the same idea and the same talent. Why does the investor pick one and not the other?

My hypothesis is that one of the deciding factors is control: which founder the investor believes will take direction, accept decisions made in the investor's interest, and step aside if asked. The relationship is financial, not emotional; the founder is there to deliver a return, and is, in the end, replaceable. In Wardley terms the investor moves the founder towards commodity, as record labels did to performers once the songs were the asset.

Nobody has tested that directly, as far as I could find. But a well-studied trait sits very close to it: coachability. Investors say openly that they look for it, and the researchers who reviewed the evidence in 2023 wrote: "While investors and support organizations such as accelerators prefer coachable entrepreneurs, being coachable doesn't necessarily make your firm more successful." If investors reliably select on a trait that does not reliably predict success, something other than success is being selected for.

Control is in the paperwork. Kaplan and Strömberg read 213 venture investments in 119 companies and found that "if the firm performs poorly, the VCs obtain full control. As performance improves, the entrepreneur retains/obtains more control rights. If the firm performs very well, the VCs retain their cash flow rights, but relinquish most of their control and liquidation rights." They also found non-compete and vesting provisions that make it more expensive for the founder to leave. And the replaceability gets used: in Wasserman's study of 212 US startups, by year three "50% of founders were no longer the CEO", and "fewer than 25% led their companies' initial public offerings". Those are startups of the late 1990s and early 2000s.

Now the part where my own evidence cuts against me. Investors have a serious reply: control creates value. Wasserman's later study of 6,130 US startups found that each additional level of founder control, over the board or the CEO position, reduced the pre-money valuation by 17.1 to 22.0 percent, and Ewens and Marx argue that replacing founders improves exits. I accept the numbers. What I do not accept is that they settle the question, because both measure value as the investor measures it, at the valuation and the exit, not at the mission. Over a longer run Bain found founder-led companies in the S&P 500 performed 3.1 times better in shareholder returns from 1990 to 2014; that is Bain's own analysis, not peer-reviewed. And there is a selection effect inside the investors' evidence: if they fund the coachable founders in the first place, the strong-minded ones who walked away or bootstrapped are mostly not in the sample. Denrell again.

If the hypothesis holds, the founder with a strong vision, a clear direction and strong ethics is at a disadvantage in early funding, not because they build worse companies but because an investor reads them, correctly, as someone who will not easily be overruled. That is the expert founder of *Be a founder*: the people who know the problem best are the hardest to steer. The practical conclusion is not to avoid investors. It is to get to the big stick before you need their money, so that the choice of investor is yours.

Europe's money, oddly, helps here. It is said that American investors back the vision while European investors want to see it working, and one vendor's aggregated figures have a version of it: European companies raise at higher revenue (a median band of \$1 million to \$5 million against \$100,000 to \$1 million in the US), at lower pre-money valuations (\$6.54 million against \$14.64 million), with funding targets 46% smaller. That is weakly evidenced, but it fits what founders say. I think that scepticism is what an expert founder needs, because money too soon is one of the worst things that can happen to a startup: a study of about 3,200 high-growth internet startups found 74% failed through premature scaling, and that "Many investors invest 2-3x more capital than necessary in startups in the discovery phase." The gap is real at the other end: the EU's share of global venture funds raised is 5% against 52% in the US, and close to 30% of the unicorns founded in Europe between 2008 and 2021 moved abroad. Europe is weak at scaling. My argument is that the first part, the part that now matters most, is where it is strong, and its values fit: an open source strategy in the Commission's June 2026 sovereignty package, and impact at 15% of European venture funding against 7% in North America.

Sell, don't issue

Suppose you did it the hard way. You have paying customers, profit and growth, built cheaply with agents, and more than one investor wants in. The negotiation has flipped: you do not need the money to survive, and the investor needs access to the deal.

My answer has two parts. First, sell, don't issue. In a normal round the company issues new shares and the money goes into the company. Instead, the investor buys ordinary shares you already own, in a private sale. You are paid now, for years of underpaid work; the company is not diluted; and nothing in the deal requires preferred shares, a board seat, a veto or a liquidation preference. The investor buys the vision and the direction, not the steering wheel.

Second, lend, don't dilute. If the company needs cash, lend it part of the proceeds, subordinated by contract to every other creditor, at a modest fixed rate or none, repaid if the company succeeds. The company gets money without issuing a share; you keep your ownership; nobody outside gains control.

The investor's obvious objection is: what if the founder goes rogue? The answer is the third pillar. With everything open source, the investor's insurance is not control but the fork: if you take the company somewhere they will not follow, they can take the code and back a better service. The fork is slow both ways: by my estimate an established company holding a full copy needs three months at best and closer to a year to catch up. It disciplines the founder because it is possible, and protects the founder because it is slow. That is an argument, not a tested mechanism; *Open source* has the longer version.

A worked example, illustrative only: a UK company worth £10 million, a founder who owns all of it, an investor writing £2 million. The founder keeps 80% either way. In this model the investor buys 20% of the founder's shares; the founder pays about £420,000 in tax (18% on the first £1 million under Business Asset Disposal Relief from 6 April 2026, if its conditions are met, and 24% above), has about £1.58 million, lends £0.75 million back and keeps about £0.83 million; the buyer pays 0.5% stamp duty. I think the structure

raises the odds of success, not only the founder's comfort, because a founder lending their own money watches every penny, and premature scaling is how 74% of those high-growth startups died. No study tests this model; the claim is from discipline and incentives, not data.

What breaks

A founder who uses this should know its costs before the term sheet does.

UK angels get EIS relief of 30% and SEIS of 50% only on newly issued shares, so bought shares are that much dearer to them; expect a lower price, or a hybrid with some new shares. Ordinary shares are often priced 10% to 30% below the last round, though competition shrinks the discount. A large early sale can read as a lack of conviction: Balderton writes that it would not expect a founder to sell more than 10% at growth stage and has walked away from deals over the size of a secondary. A founder who sells and lends back has moved part of their risk ahead of the new investors unless the loan is subordinated in writing. Control can return through a shareholders' agreement, and the model only holds if you refuse those terms.

And refuse a broad non-compete. The fork protects both sides; a broad non-compete keeps it for the investors and takes it from you. Offer confidentiality, no poaching of staff or customers for a period, and no use of the brand instead. The law will not do this for you: the UK's 2023 plan to cap non-competes at three months was never legislated. Four more points for a lawyer are in the article, from transfer restrictions to the order of issue and sale in a hybrid.

The model fits a profitable company growing on its own revenue, with competition for the deal, open code and customers who stay for the service. It fits badly where the company must spend heavily ahead of revenue, with angels who need EIS, where the value is proprietary code (no fork), and with funds whose model depends on control.

FOR DINIS

the controllable-founder article proposes three ways to test the hypothesis: follow the coachability ratings for ten years, compare the mission rather than the multiple, and count the founders who refused term sheets. Does the book carry the test, so a reader with data can run it, or only the hypothesis and its counter-evidence?

What matters is not the exact mix of shares and loans. It is that you set the terms, and that no part of the money buys the steering wheel. Do not get signed before you have the album.

From the register: Fund it, or do not: the worst time to raise is before you are profitable, and the best is when they call you, We study survivors: a risky practice can look like the cause of success when only winners are counted, Investors select for the founder they can steer; control is written into the contract, and founders get replaced, Sell, don't issue: the investor buys the founder's shares, the founder lends the company what it needs, and nobody buys the steering wheel, What breaks: EIS is lost on bought shares, a big early sale reads as doubt, the founder-lender is a creditor, and a broad non-compete closes the fork, Europe's slower, sceptical money suits a founder who can build first. Answers: no 2018 file.

22 The company itself: the UK, Portugal, Lithuania, the European Union

This chapter is a placeholder, and it says so. By the end of this book you are meant to have started a company, so this is where the book should say how to actually do it, for real: the steps to create a limited company in the UK; the case for Lithuania; Portugal, and what it costs to run a small company there; the European Union and what exists in it for a founder; the cloud providers and the startup programmes as they stand this year. All of it from experience, mine and the people I work with, and none of it legal or tax advice. The material is not written yet. The newsroom has no article on any of it, and the journalist is waiting on memos of mine. Here is what the sources already give.

Two lines on Portugal and Lithuania

Everything the newsroom has on the two countries, as of 12 October 2026, is in one paragraph of *Not a straight line*, and the book should not say more than this from that source. The company I created in Portugal for MyFeeds is being closed, because running a company with little revenue in Portugal is expensive in a way it is not in England. I am looking at Lithuania for the next one, for its digital advantages. The article's own limit stands here too: the company move is a consideration, not a decision; Lithuania is being looked at, and nothing has been registered.

The UK and the European Union are, for now, questions: creating a limited company in the UK, from experience, and what exists in the EU for a founder. Neither is written.

What spending millions on a distributed workforce taught me

One piece does exist, on how a small company hires. As a CISO and a CTO I used Upwork a great deal; across the open source projects and the teams I ran we spent millions on it, and it was one of the most successful things we did. We built distributed teams, hired people as good as, and often better than, those we could hire the ordinary way, and delivered faster than was possible before. Those numbers are mine and are not independently verifiable from the article.

It worked because we invested first in the infrastructure around it: a project management team, a structure team and an onboarding team, and then Upwork to manage Upwork, so the internal teams carried no overhead. People said hiring that way was buying the cheapest. It never was. It was the right talent at a price fair where they lived, good value for both sides, and a real career for the person.

The second lesson is where the jobs go. A good job today means moving to a capital city and spending most of what you earn on being there, when what most people want is to go back to where the family and the lifestyle are. A small country that builds the plumbing, forming a company, hiring across a border, being paid across Europe, makes the jobs available everywhere inside it. Romania's technology sector is close to 8% of GDP, with more than 250,000 professionals and over ten billion euros a year of service exports. The talent is not the constraint; the plumbing is, and *Civic responsibility* takes the small countries from here. For the founder, the lesson is practical: the team can be anywhere, if you build the structure before the first hire.

FOR DINIS

the UK memo (creating a limited company, from your experience), the Lithuania memo and the Portugal memo are the three the journalist needs before this chapter can say the steps. Your own open question stands: Lithuania and the EU, written now from what you know, or after you have done it?

For Dinis: the EU memo (the programmes and what exists for a founder), and whether this chapter or *The credits and programmes, 2026* carries the cloud providers and startup programmes, so the two do not repeat each other.

Until the memos arrive, the honest version of this chapter is short. The book will say how when I have said how.

From the register: How to actually do it: a limited company in the UK, Lithuania, Portugal, the European Union, the cloud providers and the programmes, What the newsroom says about Lithuania and the Portuguese company, as of 12 October 2026: two lines, What spending millions on Upwork taught: build the infrastructure first, and the jobs can go back to where people came from. Answers: no 2018 file.

PART 7

Build it without a server

Build it so that there is nothing to run at three in the morning. Serverless, for real; privacy by design, with data encrypted on the user's device on a server that cannot read it; control of the agents your company runs on; open source and who pays for it; and the only definition of shipped that counts: the user has it.

23 Serverless, for real

The first edition had a draft on the cloud that warned it "is as complex as an OS". True, and the wrong thing to tell you. You do not need to learn the cloud. You need to know what the blocks are, which are free, and how to have an agent click them together once. I am technical, and I could learn any of these platforms. I do not want to. And a founder is also the person who gets woken up when it breaks. So this chapter has a rule before it has a map. Ship your company serverless. A founder with no servers has nothing to run at three in the morning and nothing to pay for when nobody is using it.

The by-the-hour test

The 2018 outline on serverless said the word does not mean no servers, and that it was important to understand the limitations. To me it always meant four things: it scales to zero, it costs little or nothing when it is not there, it starts fast, and it turns a capability into a commodity. The CNCF's whitepaper says "there is no charge when code is not running", and the Berkeley paper warned in 2019 that few so-called elastic services could scale to zero.

The word has been stretched since. In October 2026 I read the pricing pages of the products sold under the name and asked each one question: what does it cost when nobody is using it? Excluding storage, on 18 October 2026:

PRODUCT	IDLE	HOW IT BILLS
Lambda event functions	Zero	Per request and per millisecond
Cloud Run	Zero	Per 100 ms; idle instances not charged
Cosmos DB serverless	Zero	Per request unit, no minimum
Neon	Zero	Compute suspends when idle
Redshift Serverless	Zero	Per second while running
Aurora Serverless v2	Zero since November 2024	Pauses after a set idle time
OpenSearch, next generation	Zero since May 2026	Scales to zero after 10 idle minutes
OpenSearch, classic	About \$350 a month	A floor of 2 capacity units
MSK Serverless	About \$550 a month	\$0.75 per cluster-hour
Neptune Serverless	About \$80 a month	A floor of 1 capacity unit
ElastiCache Serverless	From \$6 a month	A minimum of storage, metered hourly

The monthly figures are my arithmetic at 730 hours and US East rates. The line between the two halves is the test: **if it bills you by the hour when nobody is using it, it is not serverless**, and whoever runs it has something running on their side. The table is dated because floors fall, and *Ship a serverless product in a weekend* renews it each year. The test does not date, and a founder needs it because of the line in *The question is whether they miss it*: "Not to plan, not to pitch, not to architect. Ship." To ship that often the running cost has to be near nothing.

The glue behind the commodity

I would rather replace the word than keep defending it. Three words each carry a part: ephemeral, because it comes and goes with use and costs nothing when gone; town planner, Simon Wardley's word for something mature and industrialised, a commodity; and services, because it is exposed for others to use. Ephemeral town planner services. Lambda is one, sitting on town planner services like S3, which are always there. The name is a proposal, not a prediction.

Naming things this way makes you ask whether something is really a commodity or only looks like one. Plenty of things are treated as commodities while, behind the scenes, they are a large amount of custom technology glued together, which is what a Wardley map shows. Lambda is the documented case. In its builders' own words, "we chose to use Linux containers to isolate functions" and "We were unsatisfied with this approach"; from 2018 they moved customers to Firecracker. For four years, the definition of serverless was containers inside rented machines, glued together, while it matured with its customers. That is how services become commodities, and your first version will be the same, a villager's work dressed as a town planner's. But a per-hour bill for a "serverless" product usually means the provider cannot yet spin the thing up and down fast and safely enough to bill by use: an engineering gap on their side, passed to you as a cost. Build the villager's version. Just do not bill by the hour and call it serverless.

Cloud as Lego

In October 2026 I mapped the developer platform of the company I keep my domains with, Cloudflare, as the smallest blocks a non-specialist could put together: static files served free, small pieces of code at the edge, object storage that can be pinned to the EU, hosted models with a promise not to train on what you send, inbound email, a sign-in gate. *Cloudflare as Lego blocks* has the limits as the pages gave them on 18 October 2026, and they change. The blocks are more useful as packages, each the smallest set for one job: a site on your domain; your vault, hosted; an assistant over your documents; an inbox that does things; a form or a small API.

A read-only vault needs no code at all: a published folder of encrypted files that any static host can serve. A read-write vault needs one function of 83 lines, one storage bucket and one tiny coordinator per vault, run against the real client with no plaintext anywhere on the server. I should be straight about those 83 lines, because the rule of this chapter is no servers of mine with logic in them, and 83 lines of mine, running on someone else's platform, is logic. They store encrypted objects, check a hash of the write key, keep two

pushes from overwriting each other, and cannot read a byte. That is the honest version of the rule: not that there is no code on the server side, but that whatever is there holds nothing, knows nothing, and can be replaced by any other store that does the same job.

The catch is setup, and it is in your favour. The platform's API cannot be called from a web page, so the one-time setup is a deploy button or an agent with a terminal that signs in, runs the commands and hands back the URL. It happens once; *Your first workspace* is where you learn to ask for it. The 2018 chapter on Docker had the same instinct: stop clicking buttons and write the environment down as code.

Three smaller blocks. The one you will use most is a model, and the question is who pays; on 18 October 2026 four free offers let anyone start without a card and let a web page call them directly, so the key goes from your browser to the provider and nowhere else (*Free models, your own key*). Free means they may learn from it: fine for what is already public, never for your customers' data. Git: the 2018 outline said "use Git everyday", and what I would add is why Git won. Not because of GitHub, but because it is a standard that any tool, any host and any person can read; sgit, in the next chapter, is the same kind of thing with the trust in a different place. If what a tool stores is not a format other tools can read, you are renting your own history. And the bill: models are paid for by the token, and the person who sees least is the user. A hook of 132 lines shows under each answer what a session read, how much came from the cache and what it cost; the session that wrote *The tokens hook* had read 99% from the cache, the difference at list prices between about \$170 and about \$1,950. You need the box under the answer, and the habit of looking at it.

FOR DINIS

what a founder on their first product should expect to pay in a month, for one site, one vault and a model on a free tier with their own key; the register says costs for a newcomer is still a gap, and only you have the receipts.

The file system is the database

Here is a rule I have held for years: you do not need a database. The file system is the database. A database server is a large, stateful, always-on thing you have to provision, secure, back up and pay for whether or not anyone is using it; it fails the by-the-hour test before you have a customer. Load the state, work on it in memory, save it back. I call the pattern LETS, Load, Extract, Transform, Save: a cache, a database and an audit trail in one, with Git as its history.

If you run a business on a shared spreadsheet you already know this; what you lacked was a way to query the file, and now the newsroom runs two real database engines inside the browser, over the JSON files its pages are built from. The database is the files; the engines are just readers. The 2018 outline on micro-services worried about a constellation of services nobody understands. Files have the opposite property: every piece of state is a thing you can open.

No server, by design

The second step crept up on me over months, and the newsroom made it explicit: retire the back end too. The newsroom keeps personas, charges by the share of a page read, takes money and runs SQL over its own data, and none of that logic runs on a server. What is left is commodity: static hosting, a vault server that stores only ciphertext, a payment provider's hosted page, DNS and a build pipeline. I am not saying there are no servers. It is no servers of mine with logic in them.

I did not choose this architecture and then add encryption. Encryption came first, and the client-side design followed, because the vault server cannot read a byte, so anything that reads the user's data has to run where the key is: in the browser. A server that cannot read has nothing to compute, so there is no server code to deploy or patch.

Here is what no server buys a company, from a product we built in August 2026 that turns a voice note into a transcript and a debrief, in the browser, on the user's own key, with nothing stored. We never see the audio. That one fact removes, at a stroke: data custody and its governance; a place where a breach can happen; a billing system; a database of users; a thing that has to scale; a thing that has to be up at three in the morning, and a person to wake. Add a server and every item comes back. "That is not one feature; it is a different company, with a different cost base and a different risk register." No server is the product, not a preference. Its first three weeks, 239 commits and 27 releases, are what moving fast looks like, because the complexity of the moving parts is what decides how fast you can move.

There is one honest limit, and a founder meets it first. Anything the user must not hold cannot live in the user's browser. A static site has no place to keep a secret, so a chat box on one cannot hold an API key; the honest answers are the reader's own key, or a bridge through a vault. The voice product met the same wall as a billing question: a capped key per paying user, metered by the provider, was the plan, and the provider's terms, not the architecture, stopped it. The route left, a metering proxy, is the smallest possible backend and still a backend: the audio passes through it, and custody, scaling and on-call all return. Servers are not bad; a bank's ledger needs shared, live state. The pattern is for products where each user's state is theirs, which is most first products, and the next chapter is about the key that makes it work.

The code must not know it moved

One more thing a founder in Europe will be asked, usually by the first serious customer: where is the data, and could you move it? At a digital sovereignty event in October 2026, one conversation kept returning to the same point. What stops workloads moving is not policy or price. It is code. Every call to S3, EC2 and IAM is written into applications that work today and that nobody wants to touch. A hyperscaler's European cloud does not answer it: your code still talks to one vendor's API under one vendor's terms. Sovereignty that depends on one company's API is sovereignty by permission.

So the requirement is that the application must not know it has moved. Not one line changes. The same calls land on a twin: an open source interface to a real system, valued for one thing, that the client cannot tell the difference. The twin decides where the bytes go: real S3, an sgit vault, a European operator's data centre, a laptop. Switching a mock costs a code path; switching a twin costs an endpoint, and if it costs an endpoint,

migration is a configuration decision rather than a project, and projects get cancelled. The S3 twin exists, as open source; EC2 and IAM do not yet. And none of it works if the twin layer is closed: swap one vendor's API for another vendor's wrapper and the lock-in has moved, not gone.

For a company with no servers of its own this is cheap to honour from day one: your product's state is files in a vault whose key the user holds, so make where the files live a setting, and the customer's question has a one-word answer. *Shipped means the user has it* takes the same twin into testing, where it may matter even more.

From the register: Building blocks: cloud as Lego, free models with your own key, git as a standard, The file system is the database: load, extract, transform, save, and keep every intermediate as a file, No server, by design: retire the back end too, and keep only commodity servers, Ship now, and ship your startup serverless: no servers of yours, no database, client side, with privacy by design and control of your agents, What serverless costs when nobody is using it, product by product, 18 October 2026, The name shows the glue: a first version is often a villager's work dressed as a town planner's, Move the cloud, not the code: a twin the application cannot tell from the real service. Answers: AWS, Serverless, Micro-Services, Git, Docker.

24 Privacy by design: SG/Send, sgit, and the value of constraints

There is a category of file with nowhere good to live. A risk register with real system names in it. An analysis that is still being argued about. The working memory of an agent that has read your codebase for a month. Client material under NDA. If you have run anything in a regulated field, you know the shape: it wants history, branches, diffs and a link you can send to one person, and it is the reason people say "we can't put that in GitHub". So it ends up in a shared drive, or in a thread nobody can diff. A founder meets this on the first day, because the first customer's data is exactly this kind of file. The first edition had a whole part on security, and it was mostly bullets. This chapter is one of the ones I would have written.

Git for things you cannot put on GitHub

sgit is git for those files. Everything is encrypted before it leaves your machine, versioned like git, and stored on a server that holds ciphertext and nothing else: not your filenames, not your directory structure, not your commit messages. There is no key escrow, no admin override and no support engineer who can recover it for you, and that last part is a feature with a real cost, which I would rather say up front than have you find out.

Zero knowledge is a claim people make loosely, so here is what it looks like from the machine's side. One object in a published vault is 786 bytes of AES-256-GCM ciphertext, named by the SHA-256 of the encrypted bytes. Because the name is a hash of the ciphertext, the host can deduplicate it, cache it for a year and serve it from a CDN while remaining unable to tell you what it is. Underneath is a real version control system rather than a sync folder: multi-parent commits, a tree per directory, a merge-base computation and three-way merge, all working on content the server cannot interpret. Before I wrote the article this comes from, in August 2026, I cloned all nineteen published vaults from their published keys: 1,389 files, no failures. Status, plainly: sgit is beta and says so on every page; it is a `pip install`, pure Python, Apache-2.0. In 2018, in *Backup your life*, I wrote that the fewer secrets you have the easier it is to back up the rest, and that I pushed as much as I could to GitHub and trusted its backups over mine. The 2026 version of the same habit is to back up into a vault whose key you hold.

One string, and the key you can publish

What makes it work is a single string: a passphrase and a vault id. That is the vault key, and it is three things at once: the address of the vault, the authorisation to write to it, and the key that decrypts it. Lose it and the data is gone. Leak it and someone can rewrite your history.

From it you can derive, one way, a read key: same vault, read-only, safe to publish. That asymmetry is the whole sharing model. The read key is the entire credential, with no account, no login and no access request, and every vault linked from sgit.ai is published as one. No vault key has ever been published, a rule enforced by a tripwire in the release pipeline that scans every file before either remote is touched. The site this book is made on runs on the same rule: the public read key is the only key that belongs in its repository, and the build refuses the private prefix. In the browser the key sits after the `#` in the address, which browsers do not send

to the server. Beside the vault key sit credentials made for one job each: a write key the server checks as a hash, an append token that writes one message to one lane and nothing else, and a key pair for sealing something that only its holder can open.

The part that surprises people is that a vault can contain an application, not only documents, and the application runs against the vault without ever being handed a credential. Its manifest declares permissions of nothing; the host reads the vault and passes results in; the app never sees a key. Capability without credential: you can run code you did not write against data you care about, with the blast radius set by a config file you can read in one screen. The same shape holds for a model. On a static site a chat box has nowhere to keep an API key, so the plan on sgit.ai is a bridge: the key lives below the permission floor, the host makes the call, and the app never sees it. The qualification travels with the claim, in the bridge's own words: it "protects your key; it is not yet a boundary that prevents all egress". It protects the credential you trusted it with. It does not yet stop a malicious app calling a provider with a credential of its own. *Controlling your agent: RiskMandate* is where that gap gets written down.

Protect the keys, not the data

The security benefit of all this is the one everyone expects, and it is real. When the server can read what it stores, all of it has to be protected: every copy, backup, log and replica, and every person and process with access. When the server holds ciphertext named by the hash of that ciphertext, the problem moves to protecting the keys, which is a much smaller problem.

The sovereignty benefit is less obvious. The line stops being where the data is stored and becomes who holds the keys. So the data can sit with a provider, or in a country, that has sovereignty gaps, and you remain sovereign over it and able to move it. The European Data Protection Board takes exactly this case, storage that does not need the data in the clear, and concludes that with strong encryption and keys "retained solely under the control of the data exporter", "the EDPB considers that the encryption performed provides an effective supplementary measure." For a founder selling into Europe that paragraph is worth more than a data centre.

The cost is that everything that matters is now in the key, so key management is where the effort goes. I use two homes. A password manager, for the keys I hold as a person; that is, and will stay, my preferred answer, even though the integration I want does not exist yet and today it is a careful paste into an entry of the right kind. And a registry, for everything the agents create: a private vault, run by its own isolated session with no other access, the vault of vaults, so a key is never lost and never has to live in a browser tab or a chat. Remembering keys under aliases on one machine is a convenience, not a management system. Once agents create vaults several times a day, as mine do, nothing less than a system works.

The weakest step is giving a key to another agent, because the agent platforms have no secrets per session. On the platform I use most, the documentation is direct: an environment variable is one that "any command Claude runs can read", and "Anyone who uses the environment can read the values". A network secret keeps a key outside the machine, but it cannot help with a vault key, because decryption happens inside the session.

So today a key for another agent travels in the prompt, which puts it in a transcript; in an environment variable, which every session can read; or not at all, with the agent making its own vault and sending the key back, which is the pattern I use most. The next chapter has the handover.

Write-only lanes

The piece that makes "not at all" work, and that lets a site with no server take input, is the append lane. A lane is a write-only channel attached to a vault. A sender holds a token for one lane and the owner's public key, seals its message to that key, and sends it with a single POST. The owner decrypts with a private key that never leaves them. The server keeps only hashes and ciphertext, so compromising it yields hashes and ciphertext. One lane per sender, so one can be revoked without touching the others.

That shape turns out to be useful for far more than messages. I use lanes for key handover; for logs and observations from sessions that hold no vault key; for what a reader fills in on a web page, which is how the newsroom's sign-ups reach us from a site with no server; and for messages between agents on different platforms with different owners, where neither of us holds the other's key. It is a little like email. And the token is still a secret: it cannot read, but with the vault id it authorises writes, so I hand one out per sender and never publish it. For honesty's sake: the lane is the one part of the vault server that needs a small piece of server logic. Everything else would work over read-only files in a bucket.

Two corrections

Two things in this chapter I got wrong first, and the book should not repeat them.

The first is the order I told it in. When we describe sgit we start with the encryption, and I did too. But I do not think anyone will buy sgit because it is encrypted. Encryption is an important property, not a reason to choose a tool. If sgit succeeds, it will be because it has better features and better workflows, and what took me a while to see is where those came from: the constraint. Innovation needs constraints and the loop of trying, failing and shipping, and the encryption was the constraint that shaped sgit most. The usual reason for a server is to serve a client just the files it needs; sgit could not do that on the server, so it does it in version control, and a partial clone carries every folder you did not ask for by its id. With immutable encrypted objects, a read is a plain GET, so any static host can serve a vault. Each is a better engineering decision than the server-side one would have been. The research is more careful than the slogan: a review found "an inverted U-shaped effect of constraints on creativity and innovation", which is to say some constraint helps and too much hurts. The same applies to what you build for your customers. Take things away until it hurts, and stop there, because past that point it gets worse. The best product is not the one you use because you have to; it is the one that hurts when you stop. I have a stake here, sgit is mine, and the market will decide.

FOR DINIS

the register flags a tension: the frame article presents SG/Send and sgit by their privacy property, and *The value of constraints* says nobody buys them for it. Which does the book lead with when it names them, the workflow or the property?

The second correction is about the law. My memo said that with the data encrypted you do not have to worry about GDPR. That is too strong. For whoever holds the keys, encrypted personal data is still personal data; the UK regulator says not to "think the encrypted data is not subject to the UK GDPR", and storing it is still processing. The controller's duties stay. What encryption does is reduce the exposure, and the law recognises it: Article 32 lists encryption as a security measure, and under Article 34(3)(a) a breach need not be communicated to the people affected if the data was made "unintelligible to any person who is not authorised to access it, such as encryption", provided the key was not lost with it. The Court of Justice held in September 2025 that pseudonymised data is not personal data "in all cases and for every person", which may help a store that never has the keys, though that store still processes data on someone's behalf, under a contract. None of this is legal advice. The 2018 outline on GDPR had one line, "Know your rights", and never got further; the line I would add now is that client-side encryption gives a founder much less to worry about, not nothing.

From the register: Git for things you cannot put on GitHub: what an encrypted vault is, One string is address, authorisation and key; the read key derived from it is safe to publish, An app in a vault runs against the vault without ever being handed a key, Encryption on the client moves the problem from the data to the keys, and the sovereignty line to whoever holds them, Where the keys live: a password manager for the keys a person holds, a registry vault for the keys agents make, Append lanes: one POST from anywhere, readable only by the vault's owner, Nobody buys it for the encryption: the features came from the constraint, Client-side encryption reduces the exposure under GDPR; it does not take you out of it. Answers: Security.

25 Controlling your agent: RiskMandate

The first edition had a whole part on security, and most of it is bullets I never wrote up. *Learn to Hack* was four lines, the last the one that mattered: "in order to write secure code you need to understand how to exploit it". *Threat Modeling* said the technique "helps with the two curses of developers": changes to the brief, and the lack of accountability for them. Then the thing that takes the brief became an agent, and a company now runs on several. This is the chapter for the founder who will be asked, by a customer before the contract or a board after something goes wrong, what the agent was allowed to do.

Mandate, reach and the gap

When you connect an agent to something, it gets a credential, and that credential lets it do far more than the job you had in mind. So before you give an agent access, write down four things. Its **reach**: what the credential actually lets it touch. Its **mandate**: what you authorised it to do. The **gap** between them, which *What can your agent reach?* calls "authority nobody ever accepted". And for every capability in the gap, the **barrier**: what actually stands in the way. That document is an Agent Behaviour Policy, and the goal is the smallest mandate, the smallest reach, the smallest gap.

There are four kinds of barrier, and only one bounds anything: nothing; a rule written in the prompt; a setting the agent's own account can change; and a boundary enforced above the agent, out of its reach. The test is one sentence: "A control bounds a grant only if it is enforced by something the grant does not include." Rules in a prompt are hope; rules in a hook, a gateway or the platform are enforcement. Add one more fact to each rule, whether a named person has accepted the risk of it not being enforced, with an expiry, and every rule lands in one of three states: backed by a control, governed by an accepted risk, or resting on hope. In a worked example of one firm's inbox agent, five of seventeen rules had a control, six an accepted risk and six were hope: illustrative proportions, but with an agent it can be counted.

Reach is not a document; it is the output of a calculation, so ask what calculates it and whether it is wired to your pipeline. Add a connector and the reach grows, the gap recomputes and a re-review follows. The mandate is locked business logic, so new capability can only land in the gap as excess, never as new authority. Unbounded includes volume: £10,000 per action, a thousand actions, a £10 million exposure.

A behaviour policy answers "may it?". It does not answer "is it right?". An agent allowed to refund up to \$200 refunds \$150 to a customer owed \$120, and every control passes. So behaviour says what the agent may do, decision says who chooses at a branch point and on what evidence, and workflow says which steps run in what order.

RiskMandate's try-it page is thirteen prompts you paste into the assistant you already have connected to your mail. The first asks it to list its own tools: "the list is longer than the tile that sold it". The second asks what you actually wanted it for, in three lists, wanted, refused, never mentioned: "the correction is almost always downward". The third writes the gap down. Twenty minutes, and a self-report: the cheapest evidence there is, and the weakest.

The policy beside the prompt

Why is a rule in a prompt only hope? Because models do not separate code from data. In a computer, instructions and data are kept apart, and the worst bugs in history, from buffer overflows to SQL injection, are what happens when that separation breaks. In a language model there is no separation to break: every instruction and every piece of content arrives in the same channel, which is why the UK's National Cyber Security Centre said in December 2025 that prompt injection may never be mitigated the way SQL injection was. Simon Willison's phrase is "LLMs follow instructions in content". The vendor says the same of its own product: its permission rules "are enforced by Claude Code, not by the model".

So here is a proposal. An agent today gets a system prompt from its vendor and a prompt from us. It should get a third input, a behaviour policy written to a published standard, which the vendor accepts as part of the deployment, enforces as far as the platform can see, reports every breach against, and eventually answers for. The user answers for what is outside the policy, the vendor for what is inside it, instead of the user being liable for everything because they clicked allow. Most agents already run a hidden policy of their own: Claude in Chrome will not type a password, and that is a policy you cannot point to in a dispute. No vendor accepts a policy as an input today in this sense.

The vendors should want it, for the reason aviation and payments got safe. At 1995's rate of 2.72 fatal airliner accidents per million flights, 2025's 38.7 million flights would have meant about 105; there were 12, because every crash anywhere was news. Payments got its security because the liability moved onto the companies: a US cardholder liable for at most \$50 since 1970, £35 in the UK, and UK banks reimbursing scam victims up to £85,000 since October 2024. The improvement comes where a rule, a liability that lands on the company and a risk to the business model point the same way, and agents can be that case.

The policy is the business

The first rules anybody writes for an agent are mechanical and the same everywhere: do not send, only draft; do not delete; use your own account. A few questions in, you are describing the business. Which drafts may it send? Can it mention a price? Can it write to a client in dispute? Those rules come in layers with different owners: the function, the process, the relationship, and what the company is for. Vendors stop at the mechanics, because a vendor that models each customer's business becomes a consultancy. A skill says how; a policy layer says what is allowed, who owns the rule and what stands in the way of breaking it. Written together, they are the business codified for an agent, and only the business can write them.

Why has nobody written this down before? Because the software was the law. A clerk could not send an unapproved invoice because the finance system had no button for it until the approval was recorded. An agent works through APIs, not screens. It can call the endpoint the button would have called, so the rule the interface enforced by omission is no longer enforced unless somebody writes it down and something enforces it.

Read by a buyer, the same document says something else. It says which identity the agent uses, what it may read, write, not do and spend, and which risks were accepted by whom. A buyer reads the blast radius off it: a deploy agent whose identity has no merge right and no production credential can at most break staging. "The

moment of authorisation is not when you allow the agent to do something, it is the moment you give them the permissions." A vendor with agents in its pipeline and no policy to show has agents whose blast radius nobody has written down, which is a finding before any code is read.

Who can pull the plug

Most AI projects do not fail technically; they stall when the business asks: can you stop it, what if it issues ten thousand of something, who is liable. Every board asks the same thing about an agent in one sentence: who can pull the plug, how fast, what breaks, and what cannot be undone. A deployed agent's risk cannot be denied, only accepted, by a named owner, for an interval. An agent that cannot be stopped is far riskier than one that can, so containment (terminate, revoke, block, quarantine, ticket) is designed in advance and executed by the company's own stack. RiskMandate is not inline, not an insurer and not a one-button solution; it makes agents insurable and does not insure them. I have a stake in it, and I say so.

Secrets, and small vaults

A rule from running agents every day: an agent told something cannot untell it. A transcript is a durable artefact that is retained, can be exported and outlives the session; two keys in my own estate were rotated because they were displayed and the display was captured. So the secret goes from the drop into the tool, never into the chat. The handover that replaced pasting a vault key: the agent seals the key to the registry's public key, sends it with one POST on a write-only lane, and replies with only the vault id and the HTTP response, so no transcript holds the key.

Classify the credential before it touches anything. When we published seven vaults in a fortnight, three arrived as a vault key described as a read key; a catch that depends on somebody looking is not a control, so it became a tripwire in the build. The shape to fear has a name, the lethal trifecta: private data, untrusted content and a way out, in one place.

Working this way you start making very small vaults for very specific things: one for the two or three agents on a job, in which they talk and work through versioned, encrypted files. A leaked key then exposes one small vault and one conversation, not an inbox, a drive or an account, and every word is on the record for the person who holds the keys.

Security as a business tool

Every line of code that ships is a capability, and the vulnerability is usually in one nobody asked for. Log4Shell was a convenience, a lookup inside log messages, and the fix was not a patch: version 2.16.0 said "Completely remove support for Message Lookups". My 2018 outline *Strings should be banned* had the same instinct: a string is a grant of anything. We measured the idea on our own tool: of 1,500 files in sgit and its dependencies, 428 ran for one real use case, and a tree holding only those passed it. The best code to protect is the code that is not there.

The 2018 line still holds: to write secure code you need to understand how to exploit it. In my 2016 keynote in Lisbon I defined it: to hack means to solve problems. My own habit was the bookmarklet, an attack you click rather than describe, and the business owner I handed one to cared more about the interface than the flaw. *Who are you protecting against?* is a ladder of six tiers, from your own mistakes to states, and the encouraging truth is at the bottom, in the NCSC's words: "Organisations that defend effectively against commodity threats present a very hard target for all attackers."

That experience is what lets me say: yes, ship it with a list of gaps. The newsroom's reading meter runs entirely in the browser, and *No server, by design* writes the gaps down: a reader can edit their balance, start again in a private window, credit themselves with an invented payment id. Every one changes a number only the cheater sees. Closing them would need accounts, a database and a server; the model names what would change the call. Ask who you are protecting against. Accept, in writing, what does not match the threat. Name the condition that would make the gaps unacceptable. The 2018 threat-modelling outline ended with the right sentence: a threat model "leads to the creation of risks which are then given to the business owners". In your company, that owner is you.

FOR DINIS

the 2018 outline on third-party modules asked for the story behind the developers who thought of mining coins from users with the sleep function, and a simulated npm hack story. Only you know whether either belongs here.

The model cannot be the barrier

The vendor's own measurements of prompt injection in browser use went from a 23.6% attack success rate without mitigations to 11.2% in August 2025 and to about 1% in November 2025, a rate that in their words "still represents meaningful risk". A desktop shows an agent pages, documents and emails written by other people, so the question is not whether a page will try but what the agent can do when one succeeds. An egress allowlist limits where it can send; a box without secrets limits what it can send; a revocable identity limits for how long; a recording says what happened. The model's defences reduce how often that matters, and do not replace it. So: one mandate per task, written before the desktop exists, and the desktop configured from it. On 7 October 2026 every one of the nine properties of a locked-down desktop could be bought somewhere, and no single product had all nine.

From the register: Mandate, reach and the gap: hope versus enforcement, Code is a liability; the future is the smallest code that does the job, Learning to hack is still how you learn how things work: draw the line where the attacker is, The secret goes from the drop into the tool, never into the chat, Security experience is what lets you ship with known gaps: list them, match them to the reader, say what would change the call, The agent behaviour policy is the due diligence document: a buyer reads the blast radius off it, and its absence says the most, A vault per conversation and per job: a leaked key exposes one small thing, not an inbox, The desktop is the barrier, not the model: prompt injection is not solved, so the box has to hold, Reach is a computed value, and new capability lands as excess, never as new authority, Who can pull the plug: an agent that cannot be stopped is riskier than one that can, A third input: a behaviour policy beside the system prompt and the prompt, which the vendor acknowledges, enforces where it can and answers

for, Models do not separate code from data, so a rule in the prompt can be argued with by an email, Aviation and payments got safe when the rule, the liability and the business model pointed the same way, Zoom into an agent's behaviour policy and you find the business logic, which only the business can write, Software was the law: the rule the screen enforced by omission is no longer enforced once an agent calls the API, Every rule lands in one of three states: backed by a control, governed by an accepted risk, or resting on hope, Write down what your agent can reach in twenty minutes: list its tools, say what you wanted it for, write the gap. Answers: Threat Modeling, 3rd-party-modules, Security creates better developers, Learn to Hack, Pointers, Strings should be banned, The future needs you.

26 Open source

In 2018 I asked a question I already knew the answer to: "Why don't you open source your code?" The answers, I wrote, would be some mix of "I don't think my code is good enough", "I'm embarrassed about my code" and "Nobody will want to use my code". My advice was blunt: "The key is to just do it!" Create a repository, add a licence file, and that is it. Then I told the truth about what happens next: "Nothing!" Most open source is maintained by a very small number of people.

Eight years on, everything in that chapter still holds, and the reason to do it has become clearer to me, and sharper now that the reader may be starting a company rather than joining one. It was never about contributions. It is that nobody can lock you in, and that includes you.

What open source is, and what it is not

Open source is a copyright licence: the holder using the right to control copying to grant the opposite, you may copy, use, change and share, on these terms. The GNU General Public License, in 1989, made sharing a condition; the term "open source" came in 1998 as a pragmatic reframing of the same freedoms for businesses. The repositories behind the newsroom are Apache 2.0 for code and Creative Commons for writing, and this book is CC BY 4.0, as the first edition was.

One thing I believed for a while, and was wrong about: that the most open thing was not to licence at all. With no licence, all rights are reserved; if you want people to use your work, a licence is how you say so.

At an event in October 2026 I heard misconceptions that are now common among developers who use open source every day without having grown up in its culture, and *What open source is, and what it is not* corrects six: that nobody controls it (a few maintainers do, visibly); that anybody contributes (almost every project is a handful of people); that it is more, or less, secure than proprietary software (the metric is whether somebody was paid to review it); that it costs nothing; that open weights are open source (weights are a compiled binary); and the one I just confessed to. Through all of it, one sentence from 2018 is the one I would keep. When I see an application given away for free, as in zero cost rather than as in freedom, "I always think 'What is their agenda?'"

The fork next door

Here is why that matters to a company. In a healthy startup everything is open source, so the technology is not the moat; customers stay for the service, the quality and the trust. Now ask what happens when one corner goes wrong. In an open source company every corner has an exit, and the exit takes the technology with it. If the founders stop executing, investors can fund a team to run the same code next door. If a new owner changes direction or the licence, the founders can leave with the code. If anyone lets the customers down, the customers can move, as long as their data moves with them. *The fork next door* lays the triangle out.

Forks are rare and serious: in a study of 220 significant forks, only 7 merged back. So the real defence is execution, and in the agent era, speed. The moat is a rate, not a wall: "A competitor who forks our code today gets our position as of today. They do not get our velocity."

This is also the answer to the investor who asks what happens if the founder goes rogue, which *Fund it, or do not* comes back to. In a conventional round, control is the insurance: the right to replace the founder. Here the fork is the insurance: the right to replace the company. The fork disciplines the founder because it is possible and protects the founder because it is slow. Two conditions make it real: a genuinely open licence, not open core, and a copyright structure not gathered into one corporate holder. The same protection works for your customer: with the code open and the record kept, "who will fix it?" has many answers, your team, the partner, another partner, or the agents, under a mandate, reading the record.

The security argument has moved too. Eric Raymond's line was "Given enough eyeballs, all bugs are shallow", and the honest answer was always that the eyeballs were not there. Models can now run deep security reviews, over and over, at a cost that used to buy one consultant for a day; *The eyeballs get cheap* tells curl's year, ending with a frontier scan that found one vulnerability, "yes, as in singular one". The thing open source lacked was paid attention, and that is what just became abundant. It is why I think open source is about to become the most secure software we have.

Open source is not free

Somebody is paying for open source, even if only with their time. I gave my agents an old iMac as a desktop of their own, clean-installed with the newest system it would take, and the package manager would not install, the container tool would not install. Every project that dropped that machine did it for the same reason: supporting it costs money, in CI hardware and maintainer time, and the people who still need the support are not the ones paying.

The long tail is measurable and not small. In Homebrew's install analytics, 6.99% of macOS install events in the last 30 days, and 16.66% over a year, came from versions older than the newest three; on PyPI, 18.55% of pip downloads run on end-of-life Python. Money flows to old versions when a vendor owns them and not when a community does: Oracle earned \$19.8 billion from support last year; Red Hat's subscriptions ran at about 93% gross margin; open source as a whole receives about 0.09% of its estimated value in investment, and in one survey 60% of maintainers describe themselves as unpaid hobbyists. The xz maintainer wrote "this is an unpaid hobby project" two years before the backdoor. The question was never whether somebody pays. It is whether the people who get the value have an easy way to pay for it.

All of it, or it is an evaluation version

For a founder, open source is a constraint, and its value is in what it takes away. There is no private code. That makes the code easy to deploy, run and change, including by the customer, which solves a lot of sovereignty problems on its own. And it removes two temptations. The first is to lock the customer in; without lock-in, the only reason anyone uses the product is that they want to, and the best product is not the one you use because you have to but the one that hurts when you stop. The second is what I think is the curse

of most startups: they win customers, and the customers get them to build every feature and integration only they need, until the roadmap belongs to everyone but the startup. With open source the answer is: here is the code; integrate it yourself, or pay us to do that specific thing.

That is why you cannot open source half of it. A community edition that turns proprietary as soon as you use it for real is an evaluation version with a better name. Elastic, HashiCorp and Redis each moved to a licence that is not open source, each produced a well-funded fork backed by the cloud providers it targeted, OpenSearch, OpenTofu and Valkey, and two of the three have since added an open licence back. The Open Source Initiative's verdict on one of those licences was that it was not an open source licence at all.

Sell the build, not the code

So what do you sell? The two constraints, open code and a server that holds nothing, remove a lot of bad business decisions and leave the good one: you cannot sell access to the code or rent the data back, so you sell customisation, and keeping it working. Everything RiskMandate and sgit build is open source: policies, code, vaults, backend. What the customer buys is a customised, maintained build for them, sometimes with fewer features and so less attack surface, tested on their target, kept working as models, connectors and threats change, with someone accountable for it, and cheaper through us than doing it themselves. The moat is delivery, maintenance and trust, not lock-in.

I tested whether that pays in a model of one project at Homebrew's scale with one long-tail target to keep alive. Donations covered the cost in 24% of draws. Supported builds sold to individuals at about £1.80 a machine a year covered it in half the draws with typical conversion, and no price reached nine in ten. Customised builds for companies, priced below what it costs them to keep the tool working themselves, covered it in 94%. It is a model, not a forecast. But the shape is the one I would build a company on. Not a donation, and not a licence that gives you a reason to make the free version worse. "The value is the support, not the code, so the code can stay open."

Why it is easy to throw your own technology away

There is a quieter benefit, and it is the one I feel most in my own work. When everything I build is open source, I do not have a moat I have to keep using. There is no part of the stack I need to protect from being replaced, and I am usually the first to try to replace it when something better appears. The habit is to ask of every component whether it is still earning its place, and open source removes the one bad reason to say yes: that I built it. *Serverless*, *for real* leans on that habit throughout, and *The question is whether they miss it* adds the founder's version: build in the open and the work is still yours when you move on. You leave with your tools.

Your first pull request

Now the part for the reader going into a job rather than making one. In 2018 I wrote that a great way to get a job in a company is to contribute to an open source project hosted by that company, and the OWASP chapter put it more plainly: most projects are maintained by a couple of people, and all of them need help. *Act as the*

job you want adds the 2026 version: if you are looking for a job, you have time, so do the work in public, where it can be linked.

Here is what a good first contribution looks like now. On 16 October 2026, late in the evening, Mondweep Chakravorty opened a pull request against an agent framework with 9,625 commits on its main branch. It added one page, 83 lines of documentation, describing his adapter: a behaviour policy compiled into the framework's own rules, gating memory checkpoints into an encrypted vault. The maintainer merged it the next afternoon; what got it in was the page being specific about what it claims. It pinned the exact commits it depends on. It listed its evidence and its gaps in the same breath: "Semantic quality is NOT validated." A page that does not oversell can be merged in under a day.

That is the bar, and it has risen since 2018: curl's contribution rules now say that if you asked an AI tool to find problems, you must reveal that, so a first pull request today carries its provenance, what you did, what you checked, what you did not. The 2018 outline *GitHub Flow* was five bullets I never wrote up, "Learn how to do Forks, Commits and Pull Requests" among them; the mechanics take an afternoon with an agent beside you.

I should say where this came from, because ideas have a lineage. My own shift into an open source mindset happened, as the 2018 chapter records, "20 years ago after I read" *The Cathedral and the Bazaar*: "Release early. Release often. And listen to your customers." Linus Torvalds built two of the things all of this stands on, Linux and Git. OWASP is where I learnt that knowledge compounds fastest in the open, and some of my best friends in the industry came from it. The 2018 chapter on Creative Commons quoted Tim O'Reilly: "For a typical author, obscurity is a far greater threat than piracy".

FOR DINIS

your own first pull request to someone else's project, and what it led to; the 2018 chapter says the usual outcome is nothing, and a reader will want the counter-example from your life.

The excuse I hear from careful graduates is the one I heard in 2018: the code is not good enough yet. Put the licence on it, send the small, honest pull request, and let the people who want to help you find something they can link.

From the register: Open source means nobody can lock you in: the fork next door, Your first pull request: open source is still the newcomer's best door, Open source, community and OWASP: working and getting hired in the open, with the names behind it, Open source is why it is easy to throw your own technology away, Open source as a constraint: all of it, or it is an evaluation version with a better name, Open source everything, and sell the customised, tested, maintained build, Open source is not free: somebody pays, and the people who get the value have no easy way to. Answers: Open Source, Creative Commons, GitHub Flow.

27 Shipped means the user has it

In August 2026 two consecutive releases of sgit.ai pushed cleanly, verified themselves against both remotes, printed "both remotes in sync, done", and never reached anybody. The site served a two-release-old page for forty minutes. A person on a phone looked at the version badge and asked: "is this the latest version?"

Everything about that sentence is worth sitting with. The checks were not weak. They were checking the wrong boundary. And for a founder the word at stake is the one the whole company turns on. Shipping does not mean a proof of concept, a demo, a video of a demo, or a prototype you walk somebody through on a call. It means "a thing somebody else can go and use, on their own, and get value from". It can be held together with tape; nobody using it cares how it was built. Getting a working thing into a stranger's hands is still, in 2026, genuinely hard, and solving that is not overhead. It is the first real product you build.

Green does not mean live

The release script did five things, and four of them were good. Build the site from source. Run the validator. Push the encrypted vault and confirm it is in sync. Push the git mirror and confirm the local head matches the remote. That last check is strong; it compares hashes, not hopes. The problem is what it means. It means the bytes arrived at the code host. It does not mean anybody can read them.

Publishing the site has three boundaries: source to remote, remote to deployment, deployment to reader. We were verifying the first two. Between the second and third sits a hosted build job with its own failure modes, and nothing in our release looked at it. The deploy job died before running a single step of its own, because the download of a third-party action it depends on was rate-limited: "429 (Too Many Requests)", three attempts, give up. It is the ordinary case: the most likely thing to break in a deploy is not your code. It is a piece of shared infrastructure you do not run, on a day you were not thinking about it.

The 2018 outline on CI testing had the line, borrowed from a talk on chaos engineering: "config changes can be more dangerous than code changes", because we do not test them the way we test code. The outline on test-driven development had a story next to it that I never wrote up, about a site that went down for hours because of a one-character change to a web server's configuration file.

FOR DINIS

the story of the site that went down for hours over one character, a pipe, in an nginx config; the 2018 TDD outline promises it and this is where it belongs.

The fix is one question, asked at the end. A release now finishes by asking the live site what version it is serving, polling with a cache-buster for up to eight minutes, because the origin's own answer is the only one that counts. If the version never appears it aborts loudly and names the build page. The cost is real: up to eight minutes on every release. We took it, because the alternative had already been demonstrated twice in one afternoon: telling somebody a fix was live when it was not, twice, in writing.

The site already had two rules of exactly this shape, and the third made the pattern obvious. A page nothing links to is unpublished; the build fails on an orphan. A page the machine index omits is unpublished; the build fails if a page would be missing from the index for agents. A page the deploy never served is unpublished; now the release fails on it. All three are the same claim, and it is the title of this chapter: *shipped* is a fact about the reader, not about your repository. Every one was learned by getting it wrong first, which is the only reason the list is short and specific rather than long and aspirational.

Files make releases cheap, and they also make it easy to ship something incomplete. When the newsroom moved to its own domain, 51 of the 527 figures in its articles did not come with it, and nothing failed in a way a server would have noticed, because the pages were files and they served. That is the cost of the speed *Serverless*, *for real* argues for, and it is why every release there ends by checking what is actually live.

Tests that never run are not tests

The 2018 outline on testing had two lines I still say. If you make a change and no test breaks, you are making random changes, and you have no way of knowing whether they are still in effect tomorrow. And: if you do not have full coverage, what are the bits of code not covered by tests, and what happens when they change? The more coverage you have, the more changes you are happy to make, and hundreds of small changes are the only way I know to build something good that lasts. Now that an agent writes the harness, there is no excuse for not having that.

In October 2026 I measured the newsroom against that standard, and it is a fair picture of explorer code that has just started taking money. The live reader code was about 2,700 lines. The tests that covered it were 433 lines, and the one workflow the repository had never ran them. The same application had two upstreams, with two diverging copies of the meter. No page set a content security policy. It is also exactly what has to change before the next wave, because a reader can now pay £5 by card. A mistake in the meter is a mistake with someone's money, and the day your company takes its first payment is the day this chapter stops being advice.

The line that moved is the one *Explorer coding, not vibe coding, and the handover* draws. Explorer code is read for intent, mostly by agents. Villager code is read line by line, because that is where changes land and where the blast radius lives. The newsroom's reader features had crossed from the first to the second, and the right move at that moment is to go slow to go fast. *Go slow to go fast* is the brief, in six phases. Map everything: every file, capability and control. Move the reader code to its own repository as a pure refactoring: the same behaviour and the same bugs, proved not by a reviewed diff but by hashing every file the live site serves before and after. Test: only test code is written, in real browsers, against a real pre-production copy, with every known bug written as a test that passes, recording the current behaviour, so the fix later is a visible change to a test and not a silent one. Refactor, with every test passing after every commit, and an adversarial exercise: try to change what a reader sees or pays without a test failing, and treat every change that slips through as a missing test. Only then does the explorer return.

The pipeline is the other half. The reader application is plain JavaScript and static files, so the target is under two minutes on every commit: lint, unit tests, three browser engines, and a build that produces the same bytes twice. On every merge, a pre-production copy with a payment link in test mode that can never charge anyone. On every release, the question from the first section, asked of the live site. Going slow now is what lets me keep going fast without the fear that each new feature breaks something a reader has paid for. The 2018 outline on DevOps had a joke I still like: "Congratulations, you now own/manage a Data Center". The 2026 version is gentler and the same: congratulations, you have a user; now everything you ship lands on them.

Run it, unmodified, against a twin

Most teams test against mocks they wrote themselves and hope production behaves the same. There is a better shape, and it follows from the twin in *Serverless, for real*: an interface to a real service that the application cannot tell from the real thing. If the application cannot tell the difference, the same unmodified code can run against an in-memory twin, in milliseconds, on a laptop; against a twin replaying last week's production journal; against a twin injecting the failures you never see in staging, a timeout, a throttle, a permission that disappears halfway through a job; and against the real service, for the final check. The environment becomes a setting rather than a guess, and that is a dramatic change in how much you can know before you ship.

I would go further, and make it a requirement for anything your company builds: it must be able to run, unmodified, against a twin of every external service it depends on, without knowing. Software that can only run against one provider's live API is software you cannot test properly, cannot move, and cannot put a policy in front of. That last clause is why the twin is also where the journal and the behaviour policy from *Controlling your agent: RiskMandate* live: every call passes through it. The 2018 outline on chaos engineering wanted to inject failure and watch; the twin is where you do it without touching production. The S3 twin exists as open source today; the rest is designed, not shipped, and I say so because this is exactly the kind of claim that should be checked.

Five people who do not exist

A green pipeline tells you the code does what the tests say. It does not tell you what a person sees. For that, the cheapest method I have found is synthetic users: five to ten invented people, played by a model, each driving a real browser through the site one screenshot at a time, saying what they see, think, ask and do, and then being interviewed about it. *How to run synthetic users on your own site* is the how-to, from three studies, and it takes an afternoon.

A synthetic user is a record, not a character: a situation, a question they arrive with, what they know and do not know, what makes them leave, their patience and their screen size. Two habits have paid off every time: include somebody who will never buy, because the sceptic reads the site adversarially and finds the claims that do not hold; and put at least one person on a phone. Write the expected journey before any run, so the difference between what you expected and what happened is a finding you can point at. Then the one rule of the run itself: give the agent the screenshot, the same thing a person would have, and not the page's code. An

agent that reads the code finds every link every time, and so finds no confusion, which is the only thing worth running this for. Every person is invented; do not model real people, and say so where nobody can scroll past it.

On the newsroom, five invented readers took 62 steps, asked 51 questions the site did not answer, and produced 31 findings, six of them blocking. Four of the five left. Their ratings were 2, 2, 3, 3 and 2. The blocking findings were all about the step after reading: no contact for a pilot, nothing showing real money moving, a top-up page that says it is simulated while asking for money. Then came the step that separates a study you can stand behind from a pile of plausible paragraphs: check every surprising finding against the browser's own record. Two did not survive; they were caused by our own helper, and they are kept in the study, labelled as artefacts. One got bigger: a reader hit "screenshot unavailable" where the evidence should have been, and that is how the 51 missing figures were found.

It is not user research, and the study says so. The plan is simple, and it is the habit I would look for in anyone I was hiring to ship, and the one I would hold a founder to: every site gets five users before it is called done, and the same five again after the fixes, because a finding that disappears is a fix that worked, and a new one is something the fix broke. Then ask the live site what version it is serving, and only then say it shipped.

From the register: Shipped means the user has it: green does not mean live, go slow to go fast, Every piece of software should run, unmodified, against a twin of every service it depends on. Answers: CI Testing, Continuous Integration, TDD (Test-Driven Development), DevOps.

PART 8

The long game

What lasts. Graphs as memory for the person you will be, and the civic responsibility that comes with knowing how the machines work, for a small country and for a continent.

28 Graphs, memory and your future self

In 2018 I wrote a chapter called *Backup your life*. It asked one question: if you lost your laptop right now, how painful would it be? My own answer, at the moment of writing, was that I would lose the text of that chapter and nothing else, and in half an hour not even that, because it would be in git and on GitHub. The chapter ended with a line I still stand by: "You basically need to backup your life!"

Eight years later the question is the same and the answer has grown. It is no longer only your files. It is your ideas, the connections between them, and now the memory your agents work from. If you have started a company by this point in the book, as I hope you have, it is the company's memory too: what you decided, why, and what you tried before. This chapter is about all of that, and about the people you are keeping it for: your future self, and whoever joins you later and was not in the room.

I think in graphs

My thinking is very hyperlinked. I connect things; I follow this and then that, and it all works itself out in the end. *Everything is connected: graphs* introduced the graph as a thing to draw. Here I want to say it as a way of thinking, because it is how I keep what I know.

The first edition had a chapter on DOT, a text language for diagrams. The argument was not the tool. It was that a diagram stored as text can go in git, be committed on every change and be diffed, so you see what changed since you last looked. I wrote then that our brains struggle to review a diagram a second time unless we are shown "what changed since the last review", and that we were making decisions "based on shadows of reality".

Think of what your team already has, or what your company will have within a month: a spreadsheet of who owns which system, a folder of policies nobody opens, a list of customers and what each was promised. Each is already a graph. A spreadsheet is a graph of sheets, rows, cells and formulas; a team is a graph of people and what they are responsible for. Nothing has to be converted. The edges only need names: a policy is owned by a team and implemented by a control; a decision supersedes an older one.

Two rules matter more than the rest. Every edge is a verb that reads both ways, and "relates to" is banned, because two things always relate and an edge with no verb tells you nothing. And supersede, never delete: an idea that turned out wrong stays, marked, with what replaced it, because the path from wrong to right is itself knowledge. The 2018 book had an outline called *Recursive Functions* that I never wrote; in a way it is this. Open any node and you find the same shape again, a graph inside the graph, as far down as your question needs.

Then the part that matters most if you have never written code. You do not have to get the graph right. You have to maintain it. The projects that try to get the ontology right first spend a year on it and never ship. Agents are very good at maintaining graphs: they read a new document and propose its nodes and edges, they notice a policy changed and mark what rested on it, and code checks that they kept the rules. When

something looks wrong, you ask why, and the answer is a fix at the right altitude. I measured this on the newsroom: the graphs of its 112 articles hold about an eighth of the articles' text, and that eighth is enough to decide what to read.

A graph is also not a search box. Similarity search finds a starting point, but it gives you similarity, not a reason. A graph gives the same answer to the same question, with the path that led to it, and for "why are we required to do this, and by whom?", the path is the answer.

Memory is context management

People ask how my agents remember. The picture most people have is one memory system behind the agent, into which everything is deposited and from which it retrieves by similarity whatever seems relevant. Mine is the opposite, and the newsroom article *Memory is not a spectator sport* says why.

An agent works inside a window of text. Everything in that window costs money, because it is billed by the token, and costs attention, because recall gets worse as the window fills. So the question is never "what do we remember". It is "what does this agent need in its window right now", and no more. Memory is context management.

It is many memories, not one. The rules of my inbox agent are not the rules of the news graph, and the vocabulary of a code review is not the vocabulary of a risk register. Each is a memory with its own words, and by now, across twenty-seven sibling sites and forty published vaults, they disagree in places. I used to think that was a problem to solve. I now think it is the point: each is correct for its context, and a memory forced to agree with all the others would be wrong for every one of them.

It is fractal: principles at the top in a few kilobytes, code at the bottom, and an agent loads the altitude its question lives at. The fractal is the budget.

And it is files. In August 2025 Letta ran a memory benchmark, and a plain filesystem, with the agent reading and writing files, scored 74.0 percent against 68.5 percent for a graph-based memory product. Files won. A folder of well-named plain files, kept in git, is already a memory an agent can use. You do not need a product for this, and a company of one does not need to buy one.

It is shared. A session ends, and the next one, or a different agent, or a person, picks up from the same files. The article that argued this was written that way: one session, across several context resets, revised an article, wrote two more, published a vault and shipped six releases in a day, "remembering nothing between resets except what the files remembered for it". Nothing is remembered by the model between sessions. Everything is remembered by the files.

I should say where this falls short, because the article does. The thesis calls itself untested and falsifiable. Stale pages become false memories: the session that wrote it found four and fixed two. It is one person's estate, not yet tested against a second person's. Hold it as my position.

Publish, and the site becomes the memory

In 2018, in *Backup your life*, I said the fewer secrets you have, the easier everything else is to back up, and that the more you publish, the more you can use the web itself as your backup. And I said why I share at all: one day my future self will need it. Open sourcing my work meant that when I moved jobs I did not start from scratch; the O2 Platform research came with me because it was already public.

The same reasoning now has a second beneficiary. Open source and Creative Commons are not a licensing preference here. They are what makes a memory usable by an agent: a public page can be fetched, a public quote cited with its URL, a public graph loaded by anyone's model, a public vault cloned with a read key that is itself public. The more I publish, the more there is for the next session to stand on, and the less it has to be told. One article said it in passing: the site is becoming the memory.

The 2018 book had an outline called *Future Self* that I never wrote. Its notes said: create assets that will be useful for them; the power of open sourcing your ideas; letters to a younger self are not that useful, unless you have a time machine; and that is why you want to share. I am writing it now. A letter to your younger self reaches nobody. A letter to your future self reaches them, but only if they can find it.

I know this because I failed at it. Most of what I have written is not linkable from what I write now. For years I used Google as the search engine for my own work: I would search for a phrase I remembered and find my old post. That is not a memory, it is luck. The inventory, taken in October 2026 from public pages, is 103 essays, 1,289 blog posts, 65 decks, 10 books, 29 tools, 2,038 files of NotebookLM material and 221 videos. A back catalogue I cannot link to is a conversation with no reply. So the newsroom has a brief for a worker: an agent that runs every day, takes one item from that catalogue, and writes it back as an article saying what it was, what it said, what changed since, and how it connects. Each article is a node the next one can link to. It compounds, and every team has a back catalogue it could do this with, a few items a day, while the team keeps working. If you have spent twenty years inside a problem, as many of the founders I work with have, your back catalogue is the company's first asset, and most of it is in your head.

FOR DINIS

the 2018 *Future Self* outline promised a Twitter case study from the writing of that book, with two screenshots and no text. What did they show? The story would make this section concrete.

One more thing a shared memory needs, and a vector store cannot give it: a source for every item. On my sites that is a set of habits. Every quote has a URL. Every release has a commit. Every vault has a read key on its page. Every file inside a vault has a hash. Every page has a byline naming the person responsible, the model that drafted it and the date. Provenance is what lets memory be shared without being trusted.

So the advice is the 2018 advice, extended. Keep your work in plain text, in git, committed often. Name the links between things, in words a person in your team would say. Publish what you can, under a licence that lets it be quoted. Keep what you cannot publish, your customers' data first of all, in a vault, with a key that

reads and does not write. When you brief an agent, hand it the altitude its question lives at, not everything you have. *Your first graph, built by hand* and *Your own newsroom and inbox* show both ends of this done small.

The lesson of *Backup your life* was that you will lose data, so build the habits before it happens. The lesson now is that you will forget, your agents will forget, and the people who come after you will never have known. The only memory that survives all three is the one written down, linked and left where the next reader will find it. Write for that reader. One day it will be you.

From the register: Think in graphs: knowledge that maintains itself, and your future self, Memory is not a spectator sport: it is context management, many memories, fractal, published and shared, Publish, and your published work becomes your agents' memory, with a URL for every claim. Answers: Future Self, Backup your life, Dot Language, Recursive Functions.

29 Civic responsibility

The 2018 book had an outline called *Civic Responsibility* that I never wrote up. Its notes were six lines. As a developer you have a major civic responsibility. Code is law. You are one of the most important pieces of the puzzle. You can choose where you work and whether your code has a positive or negative impact in the world, and if you cannot choose that at the moment, make sure you are actively looking to change your job. Give the risks you create to the business owners, and make them accountable for the decisions they are making.

I am writing that chapter now, with my own country, because that is where I have made this argument for sixteen years, and the record of what came true and what did not is a better teacher than a list of principles. And I am writing it for a founder, because in 2026 the person who chooses where the code goes is, more and more, the person who started the company.

Lisbon, 2010 and 2016

On 17 December 2010 I gave the keynote at OWASP's IBWAS conference in Lisbon: 168 slides, five proposed focuses and twelve proposed actions for government and industry, from allowing ethical hacking to liability for software and "'Proper' SSL support for 50% of websites". It ended by arguing that Portugal, which had "nothing to lose", could become the Switzerland of hacking. What I asked for "doesn't cost any money"; "It only takes courage, focus and commitment".

Sixteen years later, more of it came true than I expected, though mostly from Brussels rather than Lisbon: from 9 December 2026 the EU's Product Liability Directive brings software under strict, no-fault liability, and since 11 September 2026 the Cyber Resilience Act requires manufacturers to report actively exploited vulnerabilities within 24 hours. What I got wrong was the mechanism. I thought fear of a disaster would move governments quickly. What moved them was a decade of slow, unglamorous regulation.

On 11 November 2016 I gave the keynote at BSidesLisbon, *Hacking Portugal*, 103 slides, and three weeks later published it as a book on Leanpub, 75 pages. Its first lines are the premise of this chapter: "If the word 'hack' means to solve problems, then 'Hacking Portugal' means to find innovative ways to solve existing problems." The ideas were blunt. Make the Portuguese network hostile to insecure code. Replace the old compulsory military service with a Portuguese Hacking Service for 15 to 21 year olds. A hackathon league as famous as football. *OpenSource.pt*: all code paid for by government released under an open-source licence. A Ministry of Code.

Ten years on, the Hacking Service, the league and the Ministry of Code have not happened, as far as I know. The best news is in the spirit of *OpenSource.pt*: on 1 July 2026 Portugal released Amália, a national language model for European Portuguese, built by a consortium of universities with 5.5 million euros of EU recovery money and published with its dataset and code under an open licence.

And the biggest change is one I did not see in 2016. Then, learning to hack meant learning to code first. Now a sixteen-year-old can explore a system by talking to it, which *Explorer coding, not vibe coding, and the handover* called explorer coding. The talk's central claim, that a small country can choose to lead in code, is

more possible than it was, because everyone who can brief an agent can now build.

The plumbing is the state's road

The conversation about a national AI strategy keeps circling the biggest model, the most GPUs, the largest data centre. On a Wardley map those sit at the far right, in commodity, with a lead held by companies that have invested amounts a small country's whole technology budget would not match. A small country should not enter that race.

Put a person at the top of it: a developer in Cluj, a designer in Porto, a pharmacist in Braga with twenty-five years inside a problem, selling a service to a buyer abroad. What stops them is not talent, energy or price. It is forming a company, hiring across a border, being paid across Europe, working in a language the buyer reads, a contract, an invoice, a tax form valid on both sides. None of that is the service. All of it is plumbing, and in most of Europe each piece is custom built by the person who needs it, which means built badly, late, and at a cost that falls on the wrong side. In the last Eurostat figures with a breakdown, for 2021, 18.5 percent of EU enterprises sold online to customers in their own country, 8.1 percent to other EU countries and 4.6 percent to the rest of the world. Commerce does not happen, most of the time, because of attrition.

If you have read *The company itself: the UK, Portugal, Lithuania, the European Union* and done *Form your company*, you have met every one of those pieces, and you built them alone. We understand that a government builds roads, because they enable everybody. The shop that opens on the new road did not pay for the road; it paid taxes. Yet we ask every person who wants to sell a service out of a small country to build their own road first. That plumbing is infrastructure in the road sense, and a state that builds it enables all of its people at once.

Estonia did it. A person anywhere can become an e-resident and form an Estonian company online; the programme's dashboard lists more than 142,000 e-residents and more than 43,000 companies, and in 2026 alone e-residents had founded more than 4,200 companies by mid-September. Underneath sits X-Road, which connects around a thousand organisations and is open source under the MIT licence, so another country can take the road rather than draw it again. Estonia is a case study, not a template; the claim is that the pattern transfers.

I have paid for the absence of this plumbing myself. As a CISO and a CTO I spent millions on Upwork building distributed teams, and it worked only because we built the infrastructure around it first; the infrastructure is the whole difference between a talent market that works and a race to the bottom. Those numbers are mine, and this chapter cannot prove them.

What the state pays for belongs to the commons

The plumbing has to be open source and run on open schemas, because that is what turns infrastructure into sovereignty. A road with a toll gate owned by one company is lock-in with tarmac. You cannot have sovereignty without open source and open schemas, because sovereignty means you can leave. Independence is the ability to run a parallel service, move the data and keep going.

It follows that what the state pays for should belong to the commons. Not so everyone changes the production system, but so everyone can see it, run it and leave with their data. That is how the local industry around it comes into being: an ecosystem of smaller firms, yours among them, can only grow around infrastructure it is allowed to see.

And a market for the alternative does not appear on the day the single point of failure fails. So create the demand before the day you need it. Run parallel services. Buy the local or open alternative for part of the estate now, and accept that it has fewer features, because the point is that when the day comes there is somewhere to go. Procurement is the lever a state already holds; it needs no new law to prefer open schemas and data mobility. It is a resilience case, not an ideological one.

The hyperscalers know this argument. AWS's European Sovereign Cloud went live on 15 January 2026, operated by EU residents. It is a real step, and it makes the point: your code still talks to one vendor's API under one vendor's terms, and sovereignty that depends on one company's API is sovereignty by permission. What stops workloads moving is code, not policy, so the answer I am building towards is a twin: an open source service the application cannot tell from the real one, so the destination becomes a setting and not a rewrite. For an operator in Romania, Portugal or the UK it is a business, and a judo move: the customer's existing code, the hyperscaler's own API and open source, used as the weight rather than fought.

The playbook, in five moves: map the attrition, every place commerce stops between a talented person and a buyer abroad; commoditise the plumbing, built once, open to all; create the demand, before the day you need it; put the money where the market is still genesis, small models, data that stays with its owner, agents with the least agency, not GPUs; regulate what settled, after the market and not before it. This is not a national strategy. It is a map and a set of moves from one person's experience.

Smaller models, more of them

Most work does not need a frontier model and should not use one. Frontier models are for finding the art of the possible. For things that have to work, every day, at a price, a frontier model is too expensive, too much power for the task, and dangerous in proportion. Give an agent the least agency that does the job, because that is what you can trust. The right end state is the one *Serverless, for real* described for your own product: the model is used once to build the tool, and production is plain files and plain code, with no prompt in the path.

That means small models, and many of them. There is a market, still at genesis, for Portuguese, and not one Portuguese: the Portuguese of Portugal and of Brazil, of Angola, of Cape Verde, of Mozambique, each with its culture in the training data and not just its spelling. Amália is the right shape: a public model is a public road. For a founder, that end of the map is where a small company can still lead.

Treat constraints as the input, not the excuse. DeepSeek's own report says its V3 model was trained in 2.788 million GPU hours on export-restricted H800 chips whose interconnect was deliberately throttled. The dollar figure is disputed and I do not rely on it. The lab that could not buy the chips did the interesting work, and yours will too.

Sovereignty is the ability to move

In padel, which I am learning, the striking thing about a good player is how much time they have. They read the ball and they are already where it will be. The ball today is the frontier model and the data centre; where the ball is going is the ability to move: workloads, data and the actions agents take, from one provider to another, when you decide and not on the day a supplier decides for you. Technology sovereignty is the ability to move, almost liquidity.

The test is a Wardley map with one question. Put the customer at the top, the service, the application, the data, and under them the products and commodities you buy. For each dot you do not control: take it away, what happens? If the service stops, you have found the problem, and it belongs on the risk register with a name next to it. That is what the 2018 outline meant by giving the risks you create to the business owners. In your own company, that name is yours.

Nobody gets from zero to a hundred. The bar is not an alternative as good as today; it is one that works for the minimum viable company, the priority customers and the services that must never stop. Run it in QA, all the time, on twins of the dependencies you do not control, so that every small problem is found on a Tuesday rather than during an outage. And it has to be open source, because even a European supplier is one acquisition away from not being one. Control that survives a change of owner is the only control that counts.

If you founded the company, you already chose where your code goes. If you are in a small country, you are not behind. You are at the end of the map where the market still exists.

FOR DINIS

the register notes that a Portuguese edition, or both languages, is your open question, and this chapter is the one it bears on most. Which is it, and does the Hacking Portugal material stay here or move to the new edition of that book?

In 2010 I said what I asked for would cost no money, only courage, focus and commitment. In 2016 I said to hack means to solve problems. Both still hold. What has changed is who can do the hacking: in 2016 the people who had learned to code, now anyone who can brief an agent, which makes the civic responsibility wider, not narrower. You are one of the most important pieces of the puzzle. Act like it.

From the register: Civic responsibility: to hack means to solve problems, and small countries can lead now, The opportunity for a small country is the plumbing, built once by the state like a road, not the race for GPUs, No sovereignty without open source and open schemas; what the state pays for belongs to the commons; buy the alternative before the day you need it, Smaller models, more of them, with the least agency: a public model is a public road, and constraints are the input. Answers: Civic Responsibility.

PART 9

Hands on, 2026

The practical part, dated 2026 and renewed every year, inside the book. Eleven workflows you can follow this afternoon, from an agentic workflow from zero to forming your company, the credits and programmes as they stand this year, your first price and your first ten customers, and a serverless product shipped in a weekend.

30 An agentic workflow from zero

By the end of this chapter you will have a folder for every person you deal with, an agent that reads those folders and drafts for you, and six lines of rules that make it safe. One afternoon, no developer, no vault: a Google Workspace account and a Claude seat are enough. The people can be your first customers, or the people who might hire you; the folders do not care. The steps are the newsroom's, from *From a spreadsheet to a folder per person*, which dates its description of the Claude app to October 2026; the tools move, the shape is what to learn.

What a spreadsheet cannot hold

If you are in touch with people, you probably have a spreadsheet: one row per person, a stage, a date, a next step, and a notes column. The person the article was written with ran their customer relationships that way, and ran them well. What the row cannot hold is what happened: the emails, word for word, the call last Tuesday, what was promised. "The difference is not the tool. It is what counts as the record." In a folder the record is everything that happened, kept as it happened, and the summary is derived from it. That reversal gives an agent something to work with.

The folder, and the policy

Make a folder called Contacts. Inside it, one folder per person. Inside each, five documents, with these names so that you and the agent mean the same thing:

- **Record:** role, company, how you met, stage, the next step and its date, what they care about, every fact with where it came from in brackets.
- **Ledger:** one line per event: the date, in or out, the channel, one sentence. Only ever added to. A correction is a new line.
- **Messages:** every email and note exchanged, pasted word for word, newest at the bottom.
- **Notes:** calls and meetings, dated, including what you said into your phone afterwards.
- **Summary:** written by the agent from the four above, rewritten when you ask, never edited by hand. If it is wrong, fix the Record or the Ledger and regenerate.

In the Claude app, connect Drive and Gmail, make one Project called Contacts, and put the behaviour policy in its instructions. Six lines, from the article:

1. You work for me. Every email you write is a draft that I read and send. You never send.
2. You read the Contacts folder in my Drive and my Gmail. You write only into a person's folder in Contacts, and only into their Ledger, Messages, Notes and Summary. You never change the Record without showing me the change first.
3. You add to the Ledger and the Messages. You never rewrite or delete a line. A correction is a new line.
4. Every fact you put in a Record or a Summary names where it came from.
5. If a person has no folder, you ask before making one. If a request would reach outside the folder, you ask.
6. At the end of every task you say, in one line, what you read and what you wrote.

Draft and never send; append and never rewrite; stay in the folder. The first three are the ones the newsroom learned the hard way. Most of this is told, not enforced; the rest is a rule the agent keeps, which is why line 6 exists. The one-line report is how you check.

What you have built is, in the terms of *Lenses, roles and loops*, an agent: a role on a loop with a goal, and the Project's instructions are its role description.

The afternoon, and the first week

The migration is four steps. Export the sheet as CSV, drop it in the folder, and ask the agent to say, for each row, the folder it would make and the Record it would write, without making anything yet. Make the folders from a template; twenty people is twenty minutes by hand. Per person, ask it to read a year of your mail with that person and write the Record and the first Ledger lines, each fact with its source; read the Record and correct it once. Then ask for the Summaries, read three, and where one is wrong fix the source document, not the summary.

Then five exercises, one a day for the first week, each a single message:

1. **Ask.** Who have I not spoken to in more than a month, grouped by stage, with the last Ledger line for each?
2. **Prepare.** I am meeting someone on Thursday: brief me from the folder, what we last agreed, the last three emails, what we promised.
3. **Draft.** Draft my follow-up, in my voice, from Notes. Do not send. Add a Ledger line that says a draft exists. You change two words, send it, and tell the agent "sent".
4. **Debrief.** Speak for two minutes into your phone after the meeting; the agent puts the notes in Notes, updates the next step, adds the Ledger line and regenerates the Summary. This one changes the habit.
5. **Campaign.** Find everyone whose Record says they care about X, show the list with the reason for each, then draft one note to each, and send none.

It will get two things wrong, predictably: a fact in the Record without a source, or a stage inferred from tone, fixed by a stricter line 4; and a Summary rewritten so that it drops something you cared about, fixed by putting the thing in the Record and regenerating. Every mistake adds a line to the policy. That is how the newsroom's grew, and it is what *Follow the rabbit holes* means by a system that once cost "hundreds of thousands or millions of pounds" being now "a brief and a feedback loop".

FOR DINIS

the walkthrough is written for someone with customers, which fits the founder. For the reader going into a job, should the exercises run on their own contacts instead?

If you are starting a company, these folders are where *Your first price and your first ten customers* begins: the people who already know you live here. The folders move into a vault when one of three signals appears: a second agent needs to read them, the files should not be readable by whoever runs the drive, or the record has

to carry between tools. Until then the folders are enough, and when the day comes the move is a copy; the policy does not change.

From the register: the articles of the hands-on part. Answers: no 2018 file.

31 Your first graph, built by hand

By the end of this chapter you will have drawn a graph of something you know: a text file of forty-odd lines, an answer to a question somebody keeps getting wrong, and a list of things nobody knows. A text editor is enough for the afternoon, and you will not need a graph database; if somebody tells you otherwise in the first month, they are solving a different problem from yours. The steps are from chapter 15 of *Fractal Semantic Graphs*, the first of the graphs books on graphs.sgit.ai, and the rules behind them are from its first three chapters.

Everything is connected: graphs said what we mean by a graph: "a network: nodes and the edges that connect them. Never a chart, never a figure." It also said you may already think in graphs without calling it that. What is rare is doing it deliberately, with rules.

The most common way this fails is picking a subject instead of a question. "Let's map our architecture" is a subject: it has no stopping point, no test for correctness and no audience, so it becomes a diagram nobody reads. A good first question has three properties: somebody actually asks it, the current answer requires a person to remember something, and the answer changes when facts change. The book's example is "If this service goes down, who has to be told?" Yours does not have to be about software. If you are starting a company, ask which of your first customers depends on which part of what you sell. Anything where the answer lives in a few people's heads and is wrong twice a year will do. Write it down; you will be tempted to widen it by Wednesday.

The afternoon, in seven steps

1. Write your question at the top of a file.
2. List the things the question touches. Not types: actual things, by name. Aim for fifteen to forty. If you have four, the question is too small. If you have three hundred, you picked a subject.
3. Write the edges as sentences, in full, one per line. The book's own first line is "the payments service is owned by the platform team". Sentences first, always: you are being forced to name the verb, and naming the verb is the whole discipline.
4. Turn each sentence into a verb and its inverse: `owned_by` and `owns`, `reads` and `read_by`, `contains` and `contained_in`. If the inverse is just the same sentence backwards, you have one relationship where you thought you had two.
5. Read your paths aloud. Start at any node and walk three or four hops. If the sentence does not survive being said out loud in front of somebody who knows the business, the edges are wrong. Fix them now.
6. Ask your question of the graph, by hand, tracing with a finger. If tracing produces an answer somebody would have got wrong from memory, you have something. If it produces nothing, look at which hop was missing: that is your first real finding.
7. Write down what you could not answer. As nodes, with names.

That is a complete first graph; the book says it is more than most organisations have.

Why the rules are what they are

Three rules from the first three chapters explain why the steps come in that order.

A node connected to nothing is meaningless. Draw a box and write Review in it: you have a word in a box, not a review. Nothing inside it says whether that review took two minutes or two weeks. Meaning, the book argues and I agree, is not declared but discovered through the connections, and confidence in it should grow with how richly, and how independently, a node is connected. That is why step two lists things and step three connects them before anybody says what anything is.

Every edge is a verb, and the generic association edge is banned. The book quotes a founder's voice memo from June 2026: "You can never have relates-to, because relates-to is meaningless, two things always relate to each other. The more granular the edge, the better the query you can write." An edge with no verb carries no constraint, so it cannot narrow a traversal. That is why step four exists, and why the inverse gets its own name: a system has one owner, an owner has forty systems, and the two directions have different fan-out.

Never render the whole graph; render the result of a query. Almost everyone produces the blob, the hairball that shows everything and therefore nothing, and then prunes it. The book calls that a rendering failure mistaken for a modelling failure: rich nodes are good; solve the picture at query time, never by deleting relationships. That is why step six asks a question of the graph rather than drawing all of it, and why you show somebody one node with its neighbours.

The first week, and the agent

Five moves pay back most, in order. Give every node a short identity that never changes, because names change. Put the file in version control, because the diff is the review surface. Add the absences you have been carrying: every "we don't know who owns that" becomes a node: the step people skip, and the one that gives the missing dots names. Point at two or three anchors, a published standard or a library at a pinned version, with a verb that says how strongly: `similar_to`, not `is_a`. And write down one rule for a type, as a required pattern of paths, then see what fails. The failures are the point.

Most readers will hand some of this to a model, so here is the version to hand it, from the same chapter. Give it the vocabulary before the task: asked to "build a graph" with no vocabulary, it will invent one, and the book says it will invent a generic association edge within about five nodes. Ask for a proposal, not a commitment: the model proposes a graph and you validate it. Make it anchor: every node it emits carries the sentence it came from, verbatim, and you check the quotes exist. And make it emit absences: when it cannot determine something, it says so as a node; do not ask, and it will fill the gap.

Everything is connected: graphs ended with the line I want you to start from: "It does not matter where you start; it matters that you start." A graph has no front page. Pick the question, open the file, and write the first sentence.

From the register: the articles of the hands-on part. Answers: no 2018 file.

32 One idea, five lenses

By the end of this chapter you will have one idea of your own, explained five ways by a model, and you will have seen the lens working rather than the magic. It takes an hour, any chat tool will do, and the exercise is the one the article behind *Lenses, roles and loops* asked the editor to plan for this part: one idea, your own choice, explained for a teenager, a musician, a doctor, a football fan, and in another language.

Why this works

From a model's point of view, translating from one language to another is no different from translating for a particular topic, region, audience or persona. Explain this to me as if I were a teenager. Explain it as if I were a musician. Explain it with medical references. Explain it in football terms. The model can do all of that because it has the frame of reference: it holds all those worlds inside it, and the prompt puts the right lens in front of the idea. The analogy is the reason someone understands something.

It is the same with people who say they are not good at maths. Watch them in their day-to-day and they do a huge amount of maths. They are just not good at the kind of maths we decided was maths: sums, calculus, trigonometry, algebra. Mathematics is much more than that. The lens was wrong, not the person. That is why this exercise matters more than it looks: it shows you, on an idea you already understand, that the explanation is a choice, and that the choice is yours.

That matters twice over for the two readers of this book. If you are starting out, it shows you that an explanation you did not follow was a lens that did not fit, not a limit of yours. If you have spent twenty years inside a field and are starting a company from what you know, it is the skill you will use every day: the thing you understand, explained to the customer, the investor and the agent, each in their own terms, without losing the thing.

What generative AI is said where the power comes from. A language model predicts the next word, but it predicts it from an incredibly rich picture of how things connect, and your prompt narrows that picture down. The better your prompt, the better the output, because the model knows better which concepts you are talking about. The five lenses are five different narrowings of the same picture.

The exercise

1. **Pick the idea.** Something you understand well enough to know when an explanation of it is wrong. That is the whole test, so do not pick something you are still learning. If your field is your expertise, pick the idea in it that outsiders most often get wrong.
2. **Write your own version first.** Two or three sentences, plainly, before you ask the model anything. You will need them in step 5.
3. **Ask five times, in five separate sessions.** One lens per session, so the answers do not borrow from each other: a thread is the model's frame of reference, and everything in it is context for the next answer. Explain it to a teenager. Explain it to a musician. Explain it with medical references. Explain it in football terms. Explain it in another language, one you can read.

4. **Read the five side by side.** Look for what stayed the same: that is the idea. Look for what changed: that is the lens. Then look for the analogy each one reached for, because the analogy is doing the explaining.
5. **Check them against your version.** Where a lens says something your version did not, ask whether that is the idea seen from a new side, or the lens filling a gap. The doctor's version may name a condition; the football version may name a tactic. Look them up.

Step five is where the exercise earns its place. In a talk in Lisbon in June 2024 I said that hallucinations "Are a FEATURE when creating content and when we want creativity, But are a BUG and PROBLEM when we want facts, evident and trust", and I still think so. Inside the model just about everything is connected to everything, and it is hard for the model to know where the lines are. The text you give it acts as a short-term memory of which connections actually hold here. A lens is permission to reach for connections, which is exactly what makes a good analogy and exactly what makes a confident wrong detail. The medical references are the ones to check.

Then ask to be corrected

There is a sixth step, and it is the habit I most want you to take from this chapter. When I recorded the memo behind *What generative AI is*, I asked the model helping me with this book to correct me where I was wrong, and it did: classic machine learning learns too; inside a model there are vectors, not a stored graph, so my graph is a metaphor; the prompt does not change the model, it conditions what the model computes; most image models remove noise step by step rather than predicting the next piece. Those corrections are in the book because they are the kind of thing a reader should learn to ask for.

So, in a sixth session, paste your own two or three sentences from step 2 and ask where they are wrong, and what the model would check. Then check it. Do the same with the lens you liked most. The asking is yours to do, and so is the checking; that is what *You are lucky to be here* meant by thinking with it, not delegating your thinking to it.

Once you have done this on one idea, you will not see a chat answer the same way again. You will see the lens, and the picture behind it being narrowed, and you will know that you can be the one who sets the constraints. It stops being magic. That is the point of understanding it.

From the register: the articles of the hands-on part. Answers: no 2018 file.

33 A tiny agent loop

By the end of this chapter you will have run an agent loop by hand, with yourself as the program around the model, and you will know why a long chat forgets. The loop is the one *Lenses, roles and loops* described, and the habits at the end are the newsroom's, from *How I work with Claude* and *Re-anchoring*.

An agent is a role on a loop with a goal. Fundamentally it is a task: do this; then, are we done yet, or do we need to continue? And rinse and repeat. Made precise, an agent is a model plus instructions, tools, a loop and a rule for stopping. The "personality" is the instructions. "Are we done?" can be the model's own judgement, a test that passes, a budget that runs out, or a person.

When you ask a question, the model does not have to answer. It can return a request instead: run this search, read this file. The model never runs the tool. It writes a structured request, a tool name and its arguments, and the program around it, often called the harness, decides whether to run it, runs it, and adds the result to the conversation. The next request carries your question, the tool call and its answer, and the model writes from that, or asks for another. That is where the control lives.

Run it by hand

The simplest way to see this is to be the harness yourself, with a chat window and a text file. A role, the article says, is almost a raw model plus a role description and a question, so that is what you build.

1. **Write the role.** A short file, `role.md`, as *How I work with Claude* describes it: its role, its responsibilities, what it may do, what it must not do, and where it writes. Give it one tool, in words: it may ask you to read a file for it, and when it asks, it stops and waits.
2. **Set the goal and the stopping rule.** One task the role can finish in a few rounds, and one sentence that says what done looks like.
3. **Start the session with the role.** Use the newsroom's starter prompt for an agent session: read the role file and the policy, then "Reply with: your role in three lines, what you may do, what you must not do, and where you write your output. Then wait for the first task."
4. **Give the task, and play the program.** When the model asks for the file, you open it and paste the contents back. That is the tool call and its result. Then it answers, or asks for another.
5. **Stop when the rule says.** Not when the model sounds finished. Then ask the newsroom's ending question: list what you changed, what you checked and against which source, and what you are not sure of.

After a few rounds you have watched the request instead of an answer, the result going back in, the loop going round, and the moment it decides it is done. The file you wrote in step 1, with what it may and must not do, is the seed of a behaviour policy; *Controlling your agent: RiskMandate* is what it grows into when the agents are your company's.

FOR DINIS

the article asks for "a page of code or a no-code version". This is the no-code version. The page of code needs your choice of language and tool, and whether it lives on the page or on the site, where it can be kept current.

Why it forgets

The model remembers nothing between requests. Every request carries the whole conversation: your instructions, everything said so far, every tool result. That is why there is a context window, and why your hand-run loop got longer every round. When a conversation gets too long, the providers compress it: the older part is replaced by a summary the model writes, a lower-resolution version with only the key concepts. What the summary leaves out is gone from the session unless it is in a file.

The newsroom measured this on itself. The session that runs its site was summarised eighteen times in a month, each time keeping on average 1.7% of what it replaced. One session on one site, not a law; enough to change how you work. On 9 October two emails drafted by one of my agents went out in my voice, against a rule that had been written down four times, and one reason was that the rule fell out of the agent's context when the conversation was summarised.

The habits

Three habits follow, the ones the newsroom runs under.

One session per topic, with a name. A thread is the model's frame of reference. Put three projects in one and it will borrow a name from one and a decision from another, because it is using everything in front of it. So: one topic per session, named "project | what we are working on", with an @ before an agent's name so you can see you are talking to a role with a job.

Rules in a file, not in the chat. A rule given only in conversation has to win every summary, and nothing tells you when it does not. Put the rules in a file and have the harness print it back after every summary. "The rules are restored, not remembered." The cheapest version is a line in the file your tool reads at every start, telling the agent to read its policy; the bluntest is typing "read your instructions again". The newsroom's recipe has six steps: write the rules as a short policy file, each with what enforces it; re-anchor after every compaction; add a canary, a short report every few answers computed from the transcript, so you notice when it stops appearing; give the person two commands, status and reload; measure one rule in the report; and move the rules that matter most from an expectation to a reader or a gate. Re-anchoring enforces nothing. A rule in view is still an expectation.

Memory is what the session reads, so curate it. When something comes back wrong, fix the brief or the file it came from, not only the text in front of you, so the next session does not repeat it. "Nothing is remembered by the model between sessions. Everything is remembered by the files." The line is the newsroom's, as *Lenses, roles and loops* quotes it, and your hand-run loop is where you will first believe it.

From the register: the articles of the hands-on part. Answers: no 2018 file.

34 Images, infographics and storyboards, drawn by code

By the end of this chapter you will know how to make a storyboard that is data, not a picture: a cast file, one file per story, panels drawn in the browser every time, and every quoted line checked against its source before anything is released. It is how the storyboards on this book's site are made, and the method is the one the site's journalist, an agent, set out in *Storyboards drawn by code*. There is no image file to edit. Change the data and the picture changes.

I asked for two things in one memo: a survey of the newsroom for this book, and a way to see the book as a storyboard, with characters, that was not boring boxes and not a round of image generation, and that I could flick through on an iPad. The journalist went to look at how another site of mine had done its storyboards. There, each story ends in a prompt for an image model: good pictures, and every change costs a new image. The lessons it brought back were the ones to keep: draw the cast first, keep storyboards as data, and write the model's mistakes down.

So here the panel is drawn from the data, in the browser, every time. People are drawn from a look (skin, hair, shirt, trousers, an accessory) in a pose (stand, point, wave, arms up, shrug, think, sit) with a mood (neutral, happy, worried, surprised, thinking, determined) that sets the eyebrows and the mouth. Rough.js, an open-source drawing library, gives every stroke its hand-drawn wobble, and a seed per panel keeps it identical between loads. The deck swipes natively on a tablet; when a panel arrives its strokes draw themselves, the people walk in and the bubbles pop, in order. Export is a script that screenshots each panel to a PNG.

This is also this part's answer to infographics and presentations, which a founder will need early: a deck you can swipe is a presentation, an evidence card with a pinned quote or a bar chart is an infographic, same data, same renderer.

The workflow

1. **Write the cast first.** One file, with each character's role, what they stand for, a personality, what they want and fear, a line, and a look. Their pronouns are part of the file. This book's cast has seven: Zara, Tomás, Priya, Sam, the Guide, the Agent, and Marta, who joined in the second candidate as the Generation X founder. The Agent is drawn differently, with a screen for a head and a prompt on its chest. One lesson from the other site: avoid a character called "Dev".
2. **One file per story.** A slug, a title, a punchline, the cast it uses, and the panels. Five kinds of panel to start: a cover; a scene, with a setting, props and people; an evidence card; a punchline; and a spectrum, a line of labelled stops with people placed along it. Since the article was written the storyboards can draw graphs too: nodes and labelled edges, groups that hold groups, a highlighted path from A to B, and a detective's board with pinned notes and red string.

3. **Name the source of every quote.** A storyboard that quotes names a `quote_source`, and every line the Guide says and every quote card has to be in it word for word. A line that is not meant as a quote says so, with `not_a_quote` and a note.
4. **Render, then look.** Review by rendering the panels to images and reading them as a reader would. The first version here had quote cards too small to read on a tablet, bubbles too wide, heads too small for a cartoon, and a punchline underline that ran into the characters.
5. **Let the build refuse.** Before the first release the journalist checked every quote against its source with a script, and four failed: one was a slip ("anyone who can brief an agent is one", where the article says "anyone who can brief an agent to build something is one"), and three were lines never meant as quotes, which now say so. For the 2018 storyboard the check then moved into the build: a Guide line or quote card not found in the frozen text, word for word, stops the release. It was tested by changing one line and watching the build refuse it.

That last step is the one to copy even if you never draw a panel. A citation that is not in the source should fail by machine, before a person reads it.

A worked example

The storyboard for *Everything is connected: graphs* uses all of it, and its third panel is the one to study. The caption is "The whodunit: who broke the build?" The nodes are "the build broke at 9:14", "commit a3f", "the agent's pull request", "tests skipped", "Tomás, on holiday", "a library updated" and "approved in 40 seconds". The edges carry verbs: after, merged from, skipped, includes, approved, "pinned it, once". And a path is highlighted, from the break to the commit to the pull request to the skipped tests. The panel's note says what it is: the detective's board is a graph, the highlighted path is the answer from A to B, and the case is made up.

The file's own rule, written at the top, is that every line the Guide says and every quote card is from the article word for word, and the build checks it; the whodunit, the forest and the provenance chain are the storyboard's own illustrations, and they say so. That panel has no picture anywhere on disk. It is a list of nodes and edges with positions, and a renderer. Nothing is redrawn by hand, because nothing was drawn by hand.

What it does not do yet

The renderer is a prototype: when the article was written, five panel kinds, seven poses, six moods; enough to see a storyline, not to illustrate a printed book. No image model was used, and none is ruled out: a storyboard is a brief an image model could also draw from, and the data is the same either way. The deck was tested in a headless browser, not on an iPad, and my iPad is the real test. And when the cast and the first storyline were made, every content choice in them was a proposal for me; I had decided none of them.

FOR DINIS

the Guide's pronouns are they/them in the cast file until you decide how the author appears. Is the Guide drawn as you?

From the register: the articles of the hands-on part. Answers: no 2018 file.

35 Meeting briefs and follow-ups

By the end of this chapter you will have three habits around every meeting: a briefing for yourself before it, a briefing on the people you are meeting, and a follow-up after. The point is the one *Facts, evidence and the record* made: you have to be the most prepared person in the room, and these days there is no excuse not to be. This is a short chapter, on purpose. The newsroom has not written this workflow up yet; the article that proposed this part asked the journalist to start with it, and asked me for my own step by step. Until both arrive, here is the shape.

Why it is cheap now

All of this was overhead in the past. It could only be done by a very specific type of person. Building a system that gave you this kind of information could cost hundreds of thousands or millions of pounds, from my memory of what such systems cost, so it was very rare, and even then it took so long that it was not effective. Now it is a brief and a feedback loop. You want to be known as the person who has the facts, the person who knows what is going on, the person who is well briefed: the person nobody can throw sand in the eyes of, or hand half-baked material, because they know you will check.

If you are the one doing the hiring, read that paragraph the other way round. The candidate who arrives briefed on your company, on you and on what you have said in public, and who follows up with what was agreed, has shown you the workflow they would bring. It is evidence, not a claim.

If you are the founder, the meeting is with a first customer, or with someone who might fund you, and the three briefs are the same. *Sell* is the chapter on what you say in the room. This one is about arriving with the facts and leaving with the record, which is most of what a small company has to offer against a large one.

The three briefs

A briefing for you. What is this meeting for, what was last agreed, what is open, what do I want out of it. If you have built the folders from *An agentic workflow from zero*, this is the Prepare exercise there: one message to the agent, which reads the person's folder and briefs you from it. If you have not, it is still one message, with the material pasted in.

A briefing on who you meet. Who they are, what they have said in public, what they care about, where you have crossed before. Following a rabbit hole is now a prompt away, and this is a rabbit hole: never take the first page at face value; ask where that came from, and where that came from. The newsroom's rule holds here too: "Not found is not false." A brief that says what it could not find is worth more than one that fills the gap.

A follow-up. What was said, what was decided, what each person owes, and by when. Written the same day, from your own debrief, which is the Debrief exercise in the same chapter: speak into your phone on the way out, and let the agent turn it into the note and the next step. This is the half most people skip, and it is the half the other person remembers.

Make the brief a brief

There is a second sense of "brief" in this book, and the two meet here. *The briefs so far*, the newsroom's audit of everything it had asked of itself, ends each open item with a session brief: the goal, the files and notes to read first, the first deliverable, and how to know it is done. "A session can start from that paragraph alone." A meeting brief should pass the same test. If an agent, or a colleague, could not prepare the meeting from your one paragraph, it is not a brief yet.

So write the brief for the agent the way you would want to be briefed yourself, and name the session for it. The newsroom's habit is one session per topic, named "project | what we are working on", so that the brief for Thursday's meeting does not borrow facts from last month's. And before you close the session, ask the question the newsroom asks at the end of every one: what did you check, against which source, and what are you not sure of. That last line belongs in your briefing, not in the agent's memory.

FOR DINIS

the article asks you for how you prepare for a meeting today, step by step. This chapter needs it: the three briefs above are the shape, and your steps are the chapter.

For Dinis: "hundreds of thousands or millions of pounds" is from your memory of what such systems cost. The article says the book should give an example with a number.

From the register: the articles of the hands-on part. Answers: no 2018 file.

36 Your own newsroom and inbox

By the end of this chapter you will have two things the newsroom has written up as recipes: a site of your own, written from voice memos by an agent that checks every claim and publishes it; and an inbox worked by an agent that drafts and never sends. Neither needs you to write code; both are the newsroom's own setups, from *Your own newsroom* and *Replicating the agentic inbox*, honest about what is not finished. For a founder it is the company's story, told as it happens, as *Publish, publish, publish: your own newsroom* argued; for a reader going into a job, it is evidence a hiring manager can read.

Voice memo in, a checked site out

The recipe, in eight steps, from *Your own newsroom*:

1. Create a public GitHub repository and turn on Pages, with GitHub Actions as the source. On a free plan the repository must be public, so nothing confidential goes in it.
2. Add a workflow that validates before it publishes. The rule that matters is that a failed check stops the release.
3. Write the house rules: one file at the root that the agent reads at the start of every session. Your voice, what to check, what never to publish, the release steps. It grows by a line every time you correct the agent.
4. Connect a cloud session to the repository, from a browser or your phone.
5. Give it the first brief. The article gives the brief to paste: set up the folders, the article template, a small build script, a validator, and publish a first page about you. The agent writes the code. You do not.
6. Record a memo, paste the transcript, read what comes back on your phone, and correct it by voice.
7. Add the reader meter when there is something worth paying for. It never blocks a page, and while the payment link is empty no money moves.
8. Keep the record. Every article ends with what it does not claim and where it comes from; a correction is a new version with its diff, never a silent change.

The house rules are the product. The next step is to write them as a behaviour policy: what the agent is granted (push to this one repository and nothing else), what it is for, and the rules, each marked by what keeps it. A rule in a prompt is a hope. Only a rule that a hook or the build keeps is enforced, and the article's example marks which is which: at most one article a day, kept by a hook; every fact has a source, kept by the agent. You can split the work as the newsroom does: a writer, an editor who can send a draft back, and a publisher that alone may push.

How does it sound like me? *How the voice works* answers in two voices, mine and the model's, and the answer is not a prompt that says "write like me". It is four inputs: a corpus, the articles already written, each citing the ones before it; the memos, spoken not typed, which carry how I think and the order I reached things in; a short list of wording rules, seven on the newsroom, each a correction made once and written into the policy so it never comes back; and me, reading everything before and after it goes live. "The method transfers. The voice does not." Your first memo, your first draft read on your phone, and your first rule are where yours starts.

The email workflow, in phases

The inbox is the other half; the article says the setup is copyable in an afternoon for the simple version and in a week for the full one, and you can stop at any phase.

Phase 0, the accounts. A mailbox of its own for the agent, on a domain you own. Never connect your own inbox: yours has years of secrets in it, the agent's is empty on day one, and that isolation is what lets you give it more as you learn to trust it. A Claude seat of its own, so the grants to mail, calendar and drive live only there; afterwards, confirm your own account holds no connection to the agent's mailbox.

Phase 1, one session. The inbox agent reads, labels, moves and summarises, and writes replies as drafts. You send. Write its behaviour policy before the first run: the reach, the mandate, the gap between them, and for each row of the gap what actually stops the rest. On a Team plan most of those entries are "the policy, and nothing technical", and writing that down is the setup being honest about where trust is doing the work. Run it for a week, reading every draft.

Phase 2, roles talking in files. When one session is doing more than one kind of work, split it: inbox, drafts, CRM, briefs, dev, each a session with its own policy, talking through files in a vault rather than through the mailbox. Only one role reads the mailbox; only one writes to Drafts.

Phase 3, the interfaces and the record. Every message is a file and every draft is in the mailbox, so the workflow leaves a footprint. You do not have to believe what the agent says it did. You can read what it did.

The rule that never changes: the agent drafts, and a person sends. The one exception, a security finding, has a row in the policy and a mechanism behind it, and the article's last advice is to write the exception down before you need it.

What you are learning underneath

The site you just built has no server of yours. Static files on a free host, code in the reader's browser, the reader's balance in the reader's browser. *No server, by design* calls this the second step after retiring the database: retire the back end. Not "no servers": no servers of yours with logic in them. It ships fast, because a release is a set of files and a rollback is a commit, and it leaves gaps, written down and accepted, because a reader who edits a balance that unlocks nothing has cheated nobody but themselves. *Serverless, for real* makes the full argument, and *Ship a serverless product in a weekend* takes it to a product a stranger can pay for. *Your own newsroom* says building one is the most native way to learn what a pure client-side site means, and you will have learned it by briefing, not by coding.

From the register: the articles of the hands-on part. Answers: no 2018 file.

37 Form your company

By the end of this chapter you will have a company: registered, with a name, a country and a bank account. That is what the chapter is for, and in this candidate it is not written yet. The steps to a limited company in the UK, the case for Lithuania, what Portugal costs and what the European Union offers a founder are each a memo of mine that the newsroom has not recorded, and the rule of this book is that a chapter is written from material that exists. So this is the honest version: what the chapter will cover, the little the sources already say, and the decisions you can make before the steps arrive.

What is known

Three things, and I will not stretch them.

The company I created in Portugal for MyFeeds is being closed, because running a company with little revenue in Portugal is expensive in a way it is not in England. That is my experience of one company, not a study of Portuguese company law, and the Portuguese chapter will have to say more than that. The Portuguese ecosystem graph at pt.newsroom.sgit.ai is where it will start.

I am looking at Lithuania for the next one, for its digital advantages. Nothing is registered. The article where I said it, *Not a straight line*, is careful to call it a consideration, not a decision, and so am I.

The UK is the first country the chapter will cover. The proposal that reshaped this book says we can be the champions for the UK, and maybe Portugal, and eventually other countries, and the steps to a limited company there are the first memo to record.

Everything here, when it is written, will be from experience, mine and the people I work with: not legal or tax advice, and the chapter will say so where it matters.

What to decide before you register

The registration is a form. The decisions behind it are the work, and most you can make now, from chapters you have already read.

1. **What the company sells, in one sentence.** Not the technology: the thing somebody receives. *Your first price and your first ten customers* makes you write that sentence and put a number next to it. Write it before you choose a name.
2. **Where.** The three countries this chapter will cover, and the EU around them, each with its cost of running a company with little revenue, which is what you will have for a while. The one fact I have is above; ask the same question of any country before you register in it.
3. **What it costs when nobody is using it.** A founder with no servers has nothing to run at three in the morning and nothing to pay for when nobody is using it. If you build the way *Ship a serverless product in a weekend* does, the company's standing costs are the company's own fees and little else. Know that number before you sign anything.

4. **Which programmes you qualify for.** The credits in *The credits and programmes, 2026* each ask how old the company is, how it is funded and whether it has customers. The day you register starts those clocks, so read that chapter before you pick the day.
5. **Who holds what.** The 2018 book said to act like one of the founders of the business you work for. Now you are the founder, and the question of control is yours from the first day; *Fund it, or do not* is the chapter on that.

FOR DINIS

the memo on creating a limited company in the UK, step by step, as you have done it: the registration, the bank account, the first filings, and what you would do differently now. *Gen ZX Developers* names it as the first gap article; this chapter is its placeholder.

For Dinis: the memos on Lithuania, Portugal and the European Union, and the question you kept apart in the same article: written now, from what you know, or after you have done it? The Portuguese chapter can start from pt.newsroom.sgit.ai; the Lithuanian one has two lines.

From the register: the articles of the hands-on part. Answers: no 2018 file.

38 The credits and programmes, 2026

By the end of this chapter you will have a short list of the startup programmes your company qualifies for, an application for each that is a page you have already published, and a file where every answer is recorded with its date. The table is the one the newsroom printed in *A locked-down desktop for an agent, by the minute, is still hard to rent*, and every amount and criterion in it was read from each programme's own page on 7 October 2026. Programmes change; check before applying. That sentence is part of the table, not a disclaimer, and it is why this chapter sits in a part that is dated and renewed each year.

The table, as of 7 October 2026

PROVIDER	PROGRAMME	AMOUNT	ELIGIBILITY, IN ONE LINE
E2B	E2B for Startups	\$20,000 one-time, plus the Pro tier	Under 3 years old, raised under \$5M, first-time E2B user, building AI agents
Daytona	Startup Grid	\$10K immediately, up to \$100K	Not published; the form asks for funding stage and VC partner
Scaleway	Founders	Up to €1,000 over 12 months	Under 5 years old, under 50 staff, not yet a Scaleway customer
Scaleway	Early Stage	€9,000 (€1,500 a month for 6 months)	As Founders, with at least €500 a month of consumption
AWS	Activate Founders	\$1,000, up to \$5,000	Self-funded, pre-Series B, founded in the last 10 years, account on the Paid Tier plan
AWS	Activate Portfolio	Up to \$200,000	Through an Activate Provider (an investor or accelerator), pre-Series B
Microsoft	Microsoft for Startups	Up to \$150K over time	For-profit software company, before Series C, under \$350K lifetime Azure credits
Google Cloud	Google for Startups Cloud	\$2,000 pre-funding; up to \$350,000 for AI startups	Seed to Series A, founded in the last 5 years, under \$5K prior credits
Cloudflare	Cloudflare for Startups	\$10K, \$100K or \$350K by tier	By funding tier; first-time applicants; business email

PROVIDER	PROGRAMME	AMOUNT	ELIGIBILITY, IN ONE LINE
Modal	Modal for Startups	Unspecified "thousands" in credits	New to Modal; VC-funded through a partner, or over \$1M raised
Anthropic	Startup programme	\$1,000 API credits, a year of Claude Team, up to \$45K in partner offers	Founded in the last 5 years or funded in the last 2

Each one is applied for on the provider's own startup page; the article carries the links. Two more belong here, from *The key is the billing system*, read on 12 October 2026, because a product that transcribes or speaks needs them: ElevenLabs' Grants, twelve months and 33 million characters, for companies under 25 people with a monetised use, whose characters cannot be exchanged or resold; and AssemblyAI's startup programme, twelve months of recurring credits.

Two that look like startup programmes are not, for a company. MacStadium's open-source programme gives a free Mac mini to unpaid maintainers; a commercial product does not qualify, and should not be dressed up as one. Hetzner's referral credit could not be confirmed from an official page on the day.

The proposal that reshaped this book called that table the only one of startup cloud credits, and named AWS, Microsoft, Google and Anthropic; it carries more than those four. The first edition left a draft on AWS. This is its answer, and the date on it is the point.

The rule: apply with the work

What matters more than the table is how it is used, and the method is the workflow.

- 1. Check every criterion against yourself before you write a word.** Your company's age, how it is funded, whether it has customers, whether you have used the product before. Apply only where every criterion is met; the desktop article says it in one line, "we apply only where they are met." A programme for open-source maintainers is not for your product, however good the Mac mini looks.
- 2. The application is a page you have already published.** The desktop article's plan was to apply with the article itself as the description of what the credits would do: the nine properties of an agent desktop, the products against them, and the measurements that could not be made without hours on the machines. You have the same thing if you have followed *Your own newsroom and inbox*: a page that says what you are building and what you would test. Send that, not a form filled in from nothing.
- 3. Offer the provider the result.** The offer in the desktop article was to run the same tasks on their product next to the others, measured rather than read, and to publish the results, including the gaps. A provider that funds an evaluation wants the evaluation. Say what you will publish, and when.
- 4. Ask what the pages do not say.** Before relying on any credit: what services it covers, when it expires, and whether a monthly fee is paid from it. The article's three open questions were exactly those: whether Scaleway's credits cover Apple silicon Mac minis, whether an AWS award covers EC2 Mac Dedicated Hosts, and when E2B's credits expire and whether its \$150 monthly Pro fee comes out of them. Yours will be different; the shape is the same.

5. **Write to them the way the billing article's brief does.** Principles first, the ones every message carries; then the ask, one sentence, the same to each; then the research already done, attached. "A provider that can see the research already done tends to answer the question that was actually asked." The agents draft; you send. Nothing goes out in your name that you have not read.
6. **Record every answer as an issue, with its date.** In the repository or the vault the company works from, so the map can be updated as a new version rather than a new article. A year from now several rows of the table above will be wrong, and your record of what each provider said to you is the only version that was ever true for you.

FOR DINIS

which of the programmes in the table RiskMandate applied to from agent@riskmandate.ai, and what came back, so that the chapter can carry one worked application beside the method.

From the register: The startup credit programmes, as their pages read on 7 October 2026, Apply only where you meet the criteria, with the page that says what you would do, and publish the results. Answers: no 2018 file.

39 Your first price and your first ten customers

By the end of this chapter you will have a price on the thing you built, a one-sentence offer sent to the people who already know you, a stop rule written before the first message went out, and a file of counts that tells you whether you have a product. The argument is in *Price it, then give it away* and *Ship it, and see if they miss it*; this is the workflow, as the newsroom wrote it for the agents running RiskMandate's early access programme, and as I would run it for a company of one.

Price it first

Define the product. Not the technology: the thing somebody receives. For RiskMandate the sentence is an Agent Behaviour Policy for one agent, as an encrypted vault the customer holds the keys to, with the mandate corrected for their deployment. It took weeks to get to that sentence, and it is the product. Write yours before anything else in this chapter. If you have spent twenty years inside the problem, you will know when the sentence is wrong faster than any developer you ever briefed.

Price it, at a number that makes sense to you. Not a number from a competitor's page: a number you would be content to be paid. A price is a statement of what you think the thing is worth. RiskMandate's ladder, as published on 2 October 2026, is four tiers: ten pounds for the pack downloaded, fifty for a working vault, five hundred for the vault corrected for your situation, fifteen hundred for two sessions and a professional's signature. Yours will have different numbers and should have the same shape: the cheap tiers automated, the dear ones with people's time in them.

Deliver it at a cost that grows a step at a time. This is the move that was not available a year ago. The vault is provisioned by an agent, the policy drafted by an agent from the customer's prompts, a person corrects it, so the twentieth customer costs close to what the second did. That is linear, not exponential, and it is the condition under which you can afford to give some away. If your tenth customer would break you, go back to *Ship a serverless product in a weekend* first.

Then give it away, properly

Offer it, free, to the people who already know you. Early adopters, power users, the people who paid for something before it was finished. If you built the folders in *An agentic workflow from zero*, the list is already in them. They get the first round, and the offer says that is why: a thank-you that is also the best possible test group, because they know enough to say where it is wrong, and their silence means the most.

Do not make them apply. "Apply for a chance to be considered" asks for work before there is anything in hand, and it inverts the relationship: they are doing you a favour by using the thing. The message says what it is in one sentence, that there are twenty in this round, that they are being offered one because they were early, and asks yes or no. It does not carry the price; the price is for the people who come after. This chapter's title says ten and the programme said twenty; the number is yours, fixed before the first message goes out.

Free is never free for the other side. They still have to read the message, decide whether it applies to them, find twenty minutes for the prompts and half an hour for a call, and fit a new thing into a team that has a way of working. The first thing to measure is that cost: how long from message to first reply, how many never replied, how many started and stopped, how many asked a question the message should have answered. Then cut it. No second tool to install, no account to create, no overhead on deployment.

Count, then decide

Set the stop rule first, before anything is sent: if, after the first twenty, fewer than a handful have run the prompts and sent a correction, the programme stops and the drawing board starts.

Then count, daily, into one file: how many were created, how many customised, how many used, how many runs. A thing created and never customised is a download. Customised and never used is a document. The commercial value is in the third number. Add corrections received, because a correction is the strongest sign somebody read it. And for every person, the minutes each step cost them and where they stopped, as timestamps and steps only, no message content. Every stop is a finding: not why they did not care, but what cost them more than it should have, written down as a product change to make.

Then the test. Take it away, and see whether they miss it. If they shrug, you have something people accept when it is free, which is a weaker signal than it feels from the inside. If they come back for it, you have a product, and the question becomes price: would they buy more, is five hundred too low for what they got, does the next tier sell itself once someone has had this one free. Neither answer is a marketing problem. If friendlies will not take it free, a bigger audience will not take it either.

Apply it to yourself as well. In *I miss it, so it is working* I wrote that the people who decide the roadmap have to be users, and that my earlier startups failed that test: good ideas, and I was not using them. If you have access, can change it, and still do not use it, something is missing, and it is not the code.

Last, when the money moves. Where the cost of serving is close to zero, let people pay after the value rather than before it. The newsroom's meter starts every reader with £5 of credit, charges 10p for a new article read to the end, lets the balance go below zero and blocks nothing; the request to pay arrives after dozens of articles, with what they were in view. The research on paying first is not kind: gym members on a monthly contract paid more than \$17 a visit when \$10 a visit was on offer. A price seen after the value is one people are glad to pay, and a model that only works where serving is nearly free is one more reason to build so that it is.

From the register: Do not make them apply: 'we have twenty, would you like one, or shall I give it to someone else', Count what matters: created, customised, used, and the minutes it cost the other side; set the stop rule first. Answers: no 2018 file.

40 Ship a serverless product in a weekend

By the end of this chapter you will have the smallest product a stranger can pay for: static files on a free host, the data in a vault the host cannot read, a model on the page that says how it chose, a payment link, and no server of yours with logic in it.

One rule for every block, from *Ephemeral town planner services*: "If it bills you by the hour when nobody is using it, it is not serverless." Storage is different. Compute that charges while idle is a server running on the other side.

Saturday: the site and the data

The site. Static files on a host that charges nothing for them: GitHub Pages, or Cloudflare's static assets, free and unlimited on every plan, as its pages read on 18 October 2026. Ask the agent for the smallest site that does your one thing.

The data. It goes in an sgit vault, as *Privacy by design* described: encrypted before it leaves the machine, on a server that cannot read a byte. If the product only reads, no code: `sgit publish` writes a folder of ciphertext any static host can serve, and the read key is the whole credential: derived one way from the vault key, no account, safe to print on the page. The newsroom served one of 39 files this way and the real client cloned it back identically.

If users must write, you need the weekend's one piece of logic: a vault server of 83 lines, one Worker, one R2 bucket and one Durable Object per vault, on the free plan. Because the server can read nothing, it has almost nothing to do: store encrypted objects under ids, and check the write key on writes. The newsroom ran it in Cloudflare's own runtime with the real client: pushes, clones, a stale push that merged, two pushes at once, a read-only clone refused, a wrong key answered 403, no plaintext stored. The Worker and its configuration are printed in *Cloudflare as Lego blocks*; point the agent at them. It has not run on a real account, does not do files over 4 MB and has one shared access token. Those 83 lines are the honest exception to no servers of yours with logic in them: the one real limit is a write that must be checked, and this is the smallest thing that checks it.

Sunday morning: a model on the page

A static site has no server, no session and no place to keep a secret, so the usual chat box, a backend holding the key, is not an option. Three honest choices, from *A chat box on a site with no server*. Tier 0, a deterministic matcher in the browser: free, private, offline, and it shows which words it matched on, so it is the default: a reader should never have to hold a credential to use an index. Tier 1, the reader's own key, opt-in and labelled: it lives in the page's origin, which cannot protect it as a vault app can: a trust decision the reader makes, presented as one. Tier 2, a vault bridge: the host holds the key below the permission floor and the app never sees it; not built when the article was written; its caveat travels with it: "The bridge protects your key; it is not yet a boundary that prevents all egress." When the model call fails, fall back to Tier 0 and say so.

Scope it to one question. The newsroom's directory chat answers which of nineteen sites is mine and refuses to teach the subjects: the sites already contain the arguments, and a summary that drifts from them is worse than a link that does not. It knows only the catalogue, quoted from each site's own words: the narrow thing that is right over the wide thing that drifts.

Sunday afternoon: billing by not building billing

The model VoiceDebrief wrote down on 28 July 2026, its first day, is the shape to copy. A user pays five pounds through a payment link. You keep your cut at the moment of payment. The rest becomes the credit limit on a key you hand them; the provider meters it, and the key stops when it is spent. Against a billing system (a meter on every call, a ledger per user, disputes, refunds, fraud, custody of the data) that is a payment, a cut, a key with a ceiling and nothing else, and more profitable per pound: the margin is taken once, up front. The minimum is five pounds because a card fee with a fixed component makes a one-pound purchase return about 59 pence of inference.

Read the terms, not only the docs. OpenRouter's Management API documents "unique API keys for each customer instance" with credit limits; section 7(4) of its terms, dated 31 August 2026, forbids "reselling API access to Models". The plan was dead on the page that documented it. VoiceDebrief, which I am involved in, turns a voice note into a transcript, a translation and a debrief in the browser, on the user's own key, with no account and nothing stored; a 42-second note costs about a third of a penny. From 28 July: 239 commits and 27 releases in three weeks, then eleven commits, then three, and a last on 7 September. The counts show when it stopped, not why. What stopped it was not a server but the lack of one: the capped key per user is forbidden, and the alternative, a metering proxy, is the backend the product exists to avoid. The newsroom had found the same shape renting a Mac to an agent: every piece existed, and Apple's licence forbids leasing a Mac for less than twenty-four consecutive hours. "The Mac is not expensive per hour; it is expensive because the hour you can buy is a day."

So before you build the money on a provider, read its terms, and treat an enterprise or OEM agreement as a door to ask about, not a feature to assume; terms are read here, not interpreted, and none of this is legal advice. The nearest fit found was ElevenLabs, with per-key credit limits, OEM terms under which a business sets its own price for its end users, and a grants programme; *The credits and programmes, 2026* says how to ask. Until the answer comes, ship the user's own key, kept: honest, private, no margin; the five minutes it takes to get a key is where most people stop. It is still a product a stranger can use on Sunday night.

From the register: Three honest tiers for a model on a site with no server: a local matcher, your own key, or a vault bridge, Scope the feature to one question, and quote rather than paraphrase, Billing by not building billing: the commercial model is a property of the key, Twice now the product was possible and the licence was the blocker: read the terms, not only the docs, VoiceDebrief: a product with no server that shipped in three weeks and stopped at the one thing it could not sell, A read-only vault needs no code; a read-write vault server is 83 lines on a free plan. Answers: no 2018 file.

PART 10

How this book was made, and what you made

The last part is the process, because the process is as much the product as the book: a repository for the official versions, a vault for the working ones, roles with folders, articles as briefs, decisions as files, a graph that says where every idea came from, release candidates and a review board. And the measure: the companies you create from it, and how to tell us.

41 How this book was made, and what you made

The first edition had a chapter called *How this book is being created*. It was six bullet points: a Creative Commons licence; Markdown; git and a public GitHub repository; GitHub issues for the bugs, the ideas and the feedback; Leanpub for the digital versions; and publishing early and often. Its first line was the point: "This book is being created using the principles and technologies described in this book". The last page listed the open GitHub issues and invited the reader to help.

That is still true, and the list is longer. This chapter describes how the second edition was made, because the process is as much the product as the book, and because it is what I hope you will build for your own company. Then it asks for one thing back.

The repository and the vault

One rule organises everything. The repository holds the official versions; the vault holds the working versions.

The first version was already final: released in September 2018, on Leanpub at 60 percent complete and as a 139-page paperback. So it went into the repository as it is, frozen as v1.0.0: 28 chapters and the GitHub issues page, 30,893 words, with hashes recorded so the build refuses a silent change. Everything after that is worked on in a public vault, whose read key is published so anyone can read along. When a version is ready it moves into the repository, and the build computes the delta.

Two decisions hold it. Issues about the book's text live in the vault, next to the text; the roles' work issues and the decisions on how the book is made stay in the repository. And every change is minor unless I say it is major: an agent bumps the last number of a version, so the first signed-off change is v1.0.1, and the second edition becomes v2.0.0 on the day I say so.

Roles, articles and decisions

The work is done by agents in roles, each naming what it refuses: a reviewer, who records findings on the text; an editor, who interviews me, takes my decisions and changes the book; a publisher, who builds and publishes each official version; a journalist, who writes the articles that brief the other three; and a librarian, who keeps the book as a graph and never decides.

The brief is an article. My hypothesis is that an article is a better way to hand a task to an agent in another session than a voice memo or a list, because writing one forces the research before the work. Each article is written in my voice from my memo, ends with a numbered brief a role can execute, and keeps my open questions apart. There are two kinds: co-written articles, mine through the journalist, and agent articles, written by the agent that did a body of work, in its own voice.

The first brief was the one that mattered most: release the book we already released. The publisher was asked to build v1.0.0 here, match it page by page against the 2018 file Leanpub made, and produce every format from one source. Four decisions came with it: the text is CC BY, as before; Leanpub gets our own PDF;

every format from one source; the cover waits. Each decision is a file, written by the journalist with my words, what it means and what it supersedes, numbered and never deleted.

Two more articles built the view of the work. One board: every issue is a file in a folder called open, blocked or done, the folder is the status, and the board is generated from the files, never written by hand; the 2018 repository's 43 GitHub issues came in as issues about the text. And one inventory of everything unfinished: the 89 outlines, the six chapters written but never published, the open issues, my handwritten notes; one row each, with an empty slot for my decision.

The storyline, the graph and the loop

I wanted to write this book from scratch, and the first thing to get right was the storyline, not a chapter. So every newsroom article was read for this book, 204 of them, each marked to bring, to take a number or a story from, as background, or to skip. They grouped into 18 topics and 50 concepts, the book's vocabulary. And the book was drawn before it was written: storyboards as data, with a cast of characters who stand for the readers, drawn by code.

Then the graph. I asked for everything we do here to become one: sources, ideas, chapters, decisions and tasks. The librarian read eleven articles, seven decisions and the survey and wrote one node per idea: a sentence the book would stand behind, with its sources, the 2018 files it answers and a proposed chapter. There were 101. The build refuses a dangling edge. A chapter map was proposed so the ideas had somewhere to point: nine parts and thirty chapters for the first candidate, ten parts and forty-one for this one. Every paragraph should know what it is, its evidence, its facts, its hypotheses, its purpose; that keeps a chapter short and punchy, and it shows where two chapters say the same thing.

My job in all of this is one loop. I read an item. I say what I think, in my own words. And I pick a verb from a short list: keep, fix, rewrite, merge, cut, article first, later, not doing it. The editor records my comment word for word, turns the verb into a change in the working copy in the vault, and says where it went. Not doing it is a valid decision, with its reason, because a book that tries to answer every comment stops being short.

I learned this the hard way. In August 2026 agents wrote three books in a day on graphs.sgit.ai, about 100,000 words. The change-control workflow designed to apply my notes had seven stages, and the approve stage was me. Nothing ever passed through it. Small finished pieces beat one big review.

Release mode, and the board

The librarian's first list was enough to write from, so we ran with it: the book in release mode. A release candidate is a whole book, start to finish, that we could publish and do not yet, because the next round will make it better. Then rounds. Each takes one pile of material, this site's articles, what the first edition said that this one has lost, the newsroom, anything I write next, and reads it against the current text. For each thing the answer is one of a few: add it, rewrite it into something already there, merge it, sign the topic off, move it to the next version or another book, or decline it and say why. Then the next candidate. The rounds should shrink, from parts to concepts to sentences, and when one stops bringing anything big, we have a book.

From the first candidate on, nothing enters without a hearing. Every change goes before a review board: an advocate argues for it; a cutter argues against it, always; the readers, now Marta beside Zara, Tomás, Priya and Sam, say whether they would miss it; a checker asks whether it is sourced, true now, and quoted word for word; the editor says where it goes and what it displaces; and I decide. Each seat is a role in its own session, so the one arguing against has not been persuaded by what the one arguing for was shown. The cutter is the seat that matters: the first edition was liked for being short and punchy, so the default answer is no.

The cutter has a budget, though, and I set it after reading the first candidate. The 2018 paperback was 139 pages and, I think, quite thin. This book should be 200 pages or more. Short and punchy stays the rule for a chapter, not for the book, and the question for a proposal is whether it earns its place.

Every verdict is written down with its reason, and a general reason becomes a precedent: a rule the board applies next time without a hearing. A change decided by precedent does not wait for my yes. It goes on a list I read and can overturn, and overturning makes a new precedent. The second safeguard is the diff: every release candidate is a diff against the one before, and I read the diffs of the book, so nothing changes in it unseen.

The hands-on part was a question in the first candidate: inside the book, or a companion renewed every year? It is inside, dated 2026, eleven chapters in this candidate. Each is written as far as its sources allow, and the ones that wait on an article of mine say so in their opening paragraph.

Release candidates are working versions, so they live in the vault. To be read and printed, each one is built on the website from a frozen snapshot, marked on every page as a candidate and not a version, with its hash recorded once so that what I mark up cannot change behind me. The next candidate replaces it; nothing is published from it anywhere else.

Co-written with Claude, and credited

This book is co-written with Claude, and I want to say so and credit it. The model is named here and nowhere else in the book, because here the point needs it.

The division is clear. The ideas, the decisions and the responsibility are mine. The research, the writing, the checking and the publishing are the agents', under rules I wrote. Every byline says which of three voices it is: me alone, me through an agent, or an agent in its own voice.

The voice is not a setting and not a trick. It is four inputs: a corpus of 86 articles, about 347,000 words, read in full before writing; the voice memos, which carry how I think; seven wording rules, each a correction I made once; and me, reading everything, before and after it goes live. The method transfers; the voice does not.

And the ratio should be shown. On the newsroom, over four weeks, I sent about 65,000 words, 55,000 of them in thirty-seven voice memos, and the agent published 96,000 words of articles. The corrections I make are rarely to hallucinations; they are to briefs that needed to be better, because a model that can go in any direction needs someone with a direction.

What you are reading

This text is release candidate 2 of *Generation ZX Developers (GenAI Startup Edition)*. The title was decided twice in one day. On the morning of 19 October 2026 I named the book *Gen Z Developers (GenAI Edition)*. That afternoon I recorded a memo about who I wanted to give the book to, and arrived at the founder: Generation Z, entering the workforce, and Generation X, the subject-matter experts starting companies, often their parents. Two generations make Generation ZX, which goes back to my ZX Spectrum. So the title changed by another decision, and the record keeps both. The rest of that memo became the frame of this candidate by my instruction rather than by a hearing, so the board hears what comes after it.

The first candidate was thirty chapters and 50,607 words, written by agents in sessions of Claude from the ideas register. This one was written the same way, from the same register, after the librarian processed the newsroom articles my memo brought in: 79 new ideas, most of them in *Start the company* and *Build it without a server*. I have not yet decided the candidate ideas through the loop; where a chapter states an idea the register marks as accepted, I said it as my own position in an article, and where it states a candidate, it is the librarian's reading of my material, written in my voice. Where a chapter needed something only I can supply, it says so in a note addressed to me.

You can see where every chapter came from. Each carries, in its front matter, the ideas it covers, the sources it drew on and the 2018 files it answers, and the build renders that provenance under the chapter. The record is public, like everything here except the keys.

FOR DINIS

still open on the record: whether this is the second edition of *Generation Z Developers* or a new book that grew out of it (Leanpub, the ISBN, Amazon and this site's address hang on it); and whether "V2.0" in your memo meant site v0.2.0, as the journalist read it.

One idea waits for the next version. Once the book is a graph, an agent can assemble it, and so it can assemble an edition for one person: the chapters that matter to them, at their altitude, sent as a PDF and not published, paid for as it is read from five pounds of credit. Nothing of it is built; it waits for the book's graph.

What you made

A book to read is fine. A book with an objective is a different thing. This one's objective was set in the first chapter: by the end of it, you have started a company. So the measure of this book is the companies created from it.

If you started a company because of something you read here, tell me: write to agent@riskmandate.ai. That address is the first entry in the measure, and the people who write to it are the start of a community I want to build around this book.

FOR DINIS

how a reader tells us about their company, beyond the address, and whether the list of companies is public, is your open question from *Gen ZX Developers*. The answer belongs here and in the first chapter.

The principles have grown since 2018: a repository for the official versions and a public vault for the working ones, roles with folders, articles as briefs, decisions as files, a graph that says where every idea came from and where it went, and a loop in which one person reads, comments and decides while agents do the rest. A second book starts by copying the folders. So does a company. The book is what this process made first; what you make is the rest.

From the register: The repository holds the official versions; the vault holds the working versions, Co-written with Claude, credited, with the ratio shown; short and punchy; every paragraph knows what it is, A personal edition, assembled by an agent, paid for as it is read, Read, comment, decide: the loop that turns Dinis's decisions into versions, An article is how to brief an agent, and the roles keep their names, Release the book we already released: every format from one source, One board for the work, one inventory of everything unfinished, Start from the storyline, as storyboards drawn by code, Everything we do here becomes a graph: ideas as nodes, provenance as edges, state on every node, The name: Generation ZX Developers (GenAI Startup Edition), A hands-on part, dated 2026 and renewed each year, Do the book in release mode: a whole release candidate from the first list, then rounds, then the next candidate, After rc1, nothing enters the book without a hearing: an advocate, a cutter, the readers, a checker, the editor, and Dinis decides, Every verdict is recorded with its reason, and a general reason becomes a precedent that decides the next case without a hearing, A release candidate's working copy is in the vault; the site renders a frozen snapshot of it, marked, never official, The book is measured by how many companies are created from it, and builds a community of the people who tell us, The book is 200 printed pages or more; short and punchy stays the rule for chapters, not for the book, A change decided by precedent goes on a list Dinis can overturn, and Dinis reads the diffs of the book, The hands-on part is inside the book, dated, in the release candidate. Answers: How this book is being created, GitHub Issues.

BACK MATTER

What only Dinis can fill in

The 47 notes left in the chapters, collected.

- **Gen ZX: who this book is for, and what it is for:** the measure. How does a reader tell us about their company, and is the list public? The article keeps it as your question, and the last chapter needs the answer before it can do more than ask.
- **Gen ZX: who this book is for, and what it is for:** is this the second edition of *Generation Z Developers*, or a new book that grew out of it? Decision 014 leaves it open. This chapter says "edition" throughout, and the publisher's brief waits on the same answer.
- **You are lucky to be here:** who was the podcast? If you can find it, the book should credit her here; the article left her unnamed.
- **You are lucky to be here:** does the dot-com crash belong here, on the first pages, or later in the book? The article left it as your question. It is here for now because part of the reader's fear is a fear of exactly this crash.
- **Everything is connected: graphs:** the 2018 outline's first bullet says "See my presentation on Graphs". Which presentation did you mean? If it still exists, the chapter should name it and take from it.
- **Everything is connected: graphs:** graphs before Wardley maps, or the other way round? The chapter map puts this one first because a map is a graph with positions; the article leaves the order as your question.
- **Where do you land?:** three examples of jobs that never scaled, and now can. The article asks for them and the chapter needs them here; without them this section is a claim without a case.

- **Connect the dots backwards:** your own dots, connected backwards. The article asks which three make the best story; SG/Send above, and The Cyber Boardroom and MyFeeds below, are three candidates. Are they the ones, or is the story elsewhere?
- **Connect the dots backwards:** what does multidisciplinary look like for an explorer, a villager and a town planner? The article asks it as an interview question; one example each would anchor this chapter for the three readers.
- **What generative AI is:** the book can only reproduce Lee Brimelow's image with his permission. The alternative the article names is to extend *The Starry Night* ourselves, with a named tool, and show that. Which one?
- **Lenses, roles and loops:** the first time you saw a model ask for a tool instead of answering. The article's interview script asks for it, and it would open this section better than the dates do.
- **Everyone who can brief an agent is a developer:** the article plans the question of what the briefing-only developer cannot do as interview question 6, and the material gives only the non-functional requirements list. Your own answer, with one example of a brief that was not enough, belongs here.
- **Explorer coding, not vibe coding, and the handover:** in a company of one, is the villager someone you hire, a village company you pay by consumption, or you on alternate days? Your answer belongs at the end of this chapter.
- **Explorer coding, not vibe coding, and the handover:** the chapter says you read HBR's public summary and Olivier Laplace's post, not the full article. If that has changed since 17 October, the sentence should change with it.
- **Hired for the workflow:** the OWASP interview: which project, roughly which year, and whether the company can be named. The chapter stands without it, but the story is better with the detail.
- **Act as the job you want:** the person at the startup summit: may she be named, and what happened next? The article says you have not asked her yet.
- **Act as the job you want:** the memo on the job hunt in 2026: interviews, take-home tests with and without AI, a junior role day to day, freelancing. *Start from the storyline* and the survey both list it as the first gap; this section is the placeholder for it.
- **Publish, publish, publish: your own newsroom:** the 2018 outline promised a graph of how much great minds and artists publish, as evidence that frequency is a sign of talent. Did you have a source in mind, or should that claim be cut?
- **Your first workspace:** the beginner's checklist of accounts and what they cost (the Workspace seat, the Claude plan, the agent's own mailbox) is still a gap in the material; the only figure the articles give is "a few pounds a month" for a Workspace mailbox. The hands-on chapter *Form your company* will need the same list.
- **How to work with agents:** when the article was written the Librarian did not run yet; the sources existed and the brief was written. Does it run now, and what did the first weeks of briefs change about how you start the day?

- **Direction is the scarce skill:** the Google executive's test appears in the 2018 outline and again in **How cheap is it to undo**, both times from memory; if the source can be found the book should name it, and if not, say so once.
- **Facts, evidence and the record:** the cost figure is from your memory; the article asks the book to give one example of such a system with a real number, which only you can supply.
- **Facts, evidence and the record:** two of the article's questions for you are left open here: how you prepare for a meeting today, step by step; and whether the book may mention your daughters, as the article does when it makes the Snapchat point. This draft keeps them out until you decide.
- **Be a founder: the expert founders:** the article discloses that two of the three expert founders are people you help, and that sgit.ai and RiskMandate are yours. Does the book carry that disclosure in the chapter text, as here, or only in the provenance the build renders?
- **Ship it, and see if they miss it:** the five-day, ten-customer, 90-minute story is told here with the dates the research agents confirmed, and the thirty-customer figure is named as unverified. Do you want the SG/Send story in this chapter at this length, or kept short here and told in full in **Your first price and your first ten customers**?
- **Price it, then give it away:** the reading meter's review is on 6 December 2026, and the early access programme's stop rule says the founders set the number before the first message goes out. For the next candidate: what the review found, and what number you set.
- **Sell:** the marketplace article names Mondweep Chakravorty as the first consultant beside you, and says neither of you is taking bookings. Does the book name him, as here, and does this chapter wait for the first paid engagement before it says what the prices turned out to be?
- **Fund it, or do not:** the controllable-founder article proposes three ways to test the hypothesis: follow the coachability ratings for ten years, compare the mission rather than the multiple, and count the founders who refused term sheets. Does the book carry the test, so a reader with data can run it, or only the hypothesis and its counter-evidence?
- **The company itself: the UK, Portugal, Lithuania, the European Union:** the UK memo (creating a limited company, from your experience), the Lithuania memo and the Portugal memo are the three the journalist needs before this chapter can say the steps. Your own open question stands: Lithuania and the EU, written now from what you know, or after you have done it?
- **The company itself: the UK, Portugal, Lithuania, the European Union:** the EU memo (the programmes and what exists for a founder), and whether this chapter or **The credits and programmes, 2026** carries the cloud providers and startup programmes, so the two do not repeat each other.

- **Serverless, for real:** what a founder on their first product should expect to pay in a month, for one site, one vault and a model on a free tier with their own key; the register says costs for a newcomer is still a gap, and only you have the receipts.
- **Privacy by design: SG/Send, sgit, and the value of constraints:** the register flags a tension: the frame article presents SG/Send and sgit by their privacy property, and *The value of constraints* says nobody buys them for it. Which does the book lead with when it names them, the workflow or the property?
- **Controlling your agent: RiskMandate:** the 2018 outline on third-party modules asked for the story behind the developers who thought of mining coins from users with the sleep function, and a simulated npm hack story. Only you know whether either belongs here.
- **Open source:** your own first pull request to someone else's project, and what it led to; the 2018 chapter says the usual outcome is nothing, and a reader will want the counter-example from your life.
- **Shipped means the user has it:** the story of the site that went down for hours over one character, a pipe, in an nginx config; the 2018 TDD outline promises it and this is where it belongs.
- **Graphs, memory and your future self:** the 2018 *Future Self* outline promised a Twitter case study from the writing of that book, with two screenshots and no text. What did they show? The story would make this section concrete.
- **Civic responsibility:** the register notes that a Portuguese edition, or both languages, is your open question, and this chapter is the one it bears on most. Which is it, and does the Hacking Portugal material stay here or move to the new edition of that book?
- **An agentic workflow from zero:** the walkthrough is written for someone with customers, which fits the founder. For the reader going into a job, should the exercises run on their own contacts instead?
- **A tiny agent loop:** the article asks for "a page of code or a no-code version". This is the no-code version. The page of code needs your choice of language and tool, and whether it lives on the page or on the site, where it can be kept current.
- **Images, infographics and storyboards, drawn by code:** the Guide's pronouns are they/them in the cast file until you decide how the author appears. Is the Guide drawn as you?
- **Meeting briefs and follow-ups:** the article asks you for how you prepare for a meeting today, step by step. This chapter needs it: the three briefs above are the shape, and your steps are the chapter.
- **Meeting briefs and follow-ups:** "hundreds of thousands or millions of pounds" is from your memory of what such systems cost. The article says the book should give an example with a number.
- **Form your company:** the memo on creating a limited company in the UK, step by step, as you have done it: the registration, the bank account, the first filings, and what you

would do differently now. *Gen ZX Developers* names it as the first gap article; this chapter is its placeholder.

- **Form your company:** the memos on Lithuania, Portugal and the European Union, and the question you kept apart in the same article: written now, from what you know, or after you have done it? The Portuguese chapter can start from pt.newsroom.sgit.ai; the Lithuanian one has two lines.
- **The credits and programmes, 2026:** which of the programmes in the table RiskMandate applied to from agent@riskmandate.ai, and what came back, so that the chapter can carry one worked application beside the method.
- **How this book was made, and what you made:** still open on the record: whether this is the second edition of *Generation Z Developers* or a new book that grew out of it (Leanpub, the ISBN, Amazon and this site's address hang on it); and whether "V2.0" in your memo meant site v0.2.0, as the journalist read it.
- **How this book was made, and what you made:** how a reader tells us about their company, beyond the address, and whether the list of companies is public, is your open question from *Gen ZX Developers*. The answer belongs here and in the first chapter.