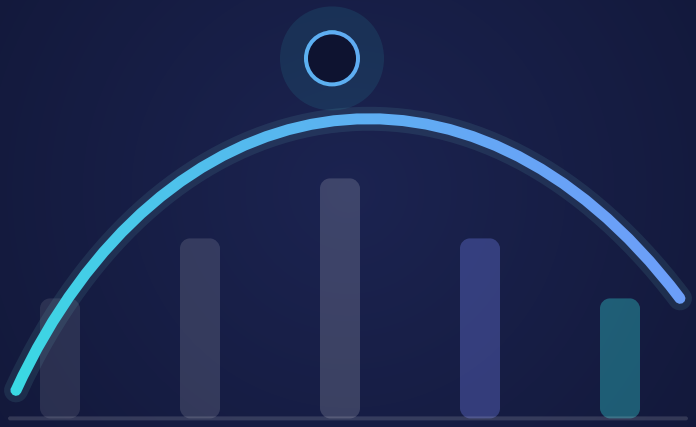
A FIELD GUIDE



LEAPFROG

A Corporate Engineer's Field Guide to Delivering AI on the Cloud

BY

Hugo Lerias

WRITTEN WITH AI · CURATED & EDITED BY THE AUTHOR

LEAPFROG

A Corporate Engineer's Field Guide
to Delivering AI on the Cloud

Hugo Lerias

These are the author's personal opinions and experience, not official BMW guidance.

WRITTEN WITH AI · CURATED AND EDITED BY THE AUTHOR

Leapfrog

A Corporate Engineer's Field Guide to Delivering AI on the Cloud

leapfrog.lerias.org · First edition, 2026.

How This Book Was Made

This book was written by pairing a human with an AI, and it seems right to say so plainly at the start.

The idea, the argument, and the structure are the author's, and so is the experience the book draws on — two decades spent building on the edge, in the cloud, and now with AI inside a large engineering organization. The author directed the work throughout: deciding what the book would claim, what belonged in each chapter and what did not, steering the research, correcting the drafts, and making the final call on every page. He read the whole of it and stands behind it.

The drafting, and much of the research assistance, were done in collaboration with an AI system — Anthropic's Claude. The model helped turn direction into prose, gather and organize sources, and move quickly through a large, structured project. It was a tool in the author's hands, not the author.

Two commitments follow from that. First, the ideas that matter belong to the researchers and builders whose work this book rests on; they are credited in the references at the end of every chapter, and any mistakes are the author's alone. Second, everything here is the author's personal view and experience — not official guidance from BMW or any employer.

There is a fitting symmetry in admitting this. The book argues that the way forward is to pair human judgment with AI — openly, and well — to deliver things you could not have delivered as quickly on your own. This book was made in exactly that way. It is, in a small way, an example of its own thesis.

— *Hugo Lerias*

Preface

Every large company runs on a quiet, reasonable-sounding phrase: *yes, but.* Yes, we should do this — but let's wait for the platform decision. Yes, this would help — but not until the migration is done. Yes, but let's pilot it first, let's bring in a consultant to be sure. Each *but* is individually sensible, and together they add up to a culture of deferral dressed as prudence. For most of the last two decades that was survivable, because the cost of waiting was low.

Then AI revoked the patience. The barriers a corporation puts in front of delivering something new — the migrations, the committees, the rented answers — are now the difference between the organizations pulling ahead and the ones quietly falling behind, and the gap compounds by the quarter. Waiting stopped being safe. This book is about the way out, and its argument is simple: you don't have to clear the barriers one at a time. You can leapfrog them, using skills you almost certainly already have.

Because most of the hype gets one thing wrong. Delivering AI at scale is not, mainly, a machine-learning problem. It is a cloud-engineering problem — integration, data, deployment, evaluation, cost, security, control — and those are exactly the disciplines a career engineer has been practicing for years. This book is not for AI researchers or data scientists; they are well served already. It is for the engineer who is deep into a career, fluent in how real systems actually work, AI-adjacent but not AI-native, and stuck inside the

yes, but. If that is you, the central claim of these pages is that **you were always qualified for this**, and the barriers are lower than your organization has led you to believe.

The guide draws on two decades spent in the companies that built the modern internet — the edge, the early cloud, and now AI-enabled engineering inside a large automotive company. But it leans far less on any one career than on the researchers and builders it cites throughout, and is best read as a map that credits the people who charted the territory. (How the book itself was made — a human working with an AI — is described in the note just before this one.)

A few words on how it is built. It moves in four parts. First, *why and where* — why AI is now a must rather than a should, and where it actually belongs in an organization (the answer is not "a chatbot"). Then *how to build* — the landscape, the cloud foundations, how models are served, the patterns that hold up, and how to ground a model in your own data. Then *how to survive Monday* — deploying, operating, and staying in control of something non-deterministic, and how to know it is working. And finally *at scale and forward* — cost and performance, and security, governance, and the road ahead. Between the build and the operating parts sits a short interlude that asks you to stop reading and ship something tiny, this week, because the gap between the five percent who deliver AI and the ninety-five percent who don't was never knowledge — it was having shipped.

Two habits run through every chapter. Each ends with *experiments to run* — small, real, shippable tasks — and with curated references that point to the primary sources. And the whole book is deliberately vendor-neutral: it teaches the layer *beneath* the tools, the durable structure that survives the next model release, rather than this quarter's product names. Where you need hands-on code and cur-

rent specifics — which age in weeks — it points you to the companion labs at leapfrog.lerias.org. The book is the why and the shape; the site is where you build.

The map is drawn. The territory is yours to cross. Let’s begin.

Contents

PART I

Why & Where

Why AI is a must — and where it actually belongs.

01 The Convergence

CHAPTER 1

"Yes, we want to move to the cloud. But the migration will take years."

You have heard that sentence. You may have said it. And for a long time, it was the sensible thing to say.

This chapter is about why it stopped being sensible — and why the same instinct that once protected you is now the thing most likely to leave you behind.

The corporate trap

You have spent a career learning how technology actually works. You know where the bodies are buried in the network diagram. You know which system nobody is allowed to touch and why. You know the difference between a demo and a thing that survives a Monday morning.

And you work somewhere that every good idea hits the same wall.

Yes, but the migration will take years. Yes, but we have on-prem contracts through 2029. Yes, but security hasn't signed off. Yes, but let's wait for the platform team to define a standard first.

None of these objections are wrong, exactly. That is what makes them dangerous. They are institutional patience dressed up as prudence, and they have a way of compounding until "later" quietly becomes "never." The trap was never a technical one. The technology was almost always ready before the organization was. The trap is a culture that has learned to mistake deferral for diligence — and a role, yours, that has learned to survive inside it by waiting for permission.

For most of the last two decades, waiting was survivable. The cost of moving late to a new platform was a few quarters of lost efficiency, absorbed by an organization moving late alongside its equally cautious competitors. Everyone deferred together. The train was slow, and it waited at the station.

That is the part that has changed.

Not a need — a must

Cloud, for most enterprises, was a *should*. A better way to do what you were already doing. You could get there in three years or five, and the difference, painful as it felt internally, rarely decided whether the business lived or died.

AI is not a *should*. For corporate technology work, it is fast becoming a *must* — not because of hype, but because the job itself is being redefined around it. The expectation of what one engineer can produce, how fast a team can ship, what "we looked into it" is allowed

to mean — all of that is being reset in real time by people who decided not to wait.

Which means the old survival strategy inverts. Deferring in lockstep with your competitors was safe. Deferring while a subset of your competitors — and a subset of your own colleagues — *leapfrog* is not safe. It is the specific mechanism by which capable people become obsolete inside functioning organizations.

Leapfrogging is the move this book is built around. Not the orderly, sequenced, boil-the-ocean migration that has to finish before anyone is allowed to build something intelligent on top of it. The reverse: building the intelligent thing now, on infrastructure you can rent by the hour, in a corner of the estate small enough that nobody has to convene a steering committee to approve it. You do not need the three-year migration to be finished. You need to stop treating it as a prerequisite.

The death of the rented answer

There is a reflex, deeply worn into corporate technology, for what to do when a capability is missing in-house: rent it. Bring in the consultants. Buy the transformation program. Wait for the deck, the roadmap, the target operating model. For thirty years this was simply *how* large organizations acquired skills they did not have.

That reflex is now working against you, and it is worth being precise about why.

The scarce asset has moved. When building was hard, the people who could build were rare, and renting them made sense. But the tools got good — good enough that a competent engineer with a model working alongside them can now produce, in an afternoon,

work that used to require a small team and a statement of work. What remains scarce is not the ability to build. It is knowing *what* is worth building: which process actually matters, where the real data lives, what "good" looks like in your specific corner of a specific business. That knowledge is yours. It is the thing thirty years of context bought you, and it is precisely the thing an external firm has to spend the first six weeks of every engagement trying to reconstruct.

If you doubt that delivery, rather than modeling, is now the bottleneck, watch where the frontier labs are putting their people. In 2026, OpenAI, Anthropic, Microsoft, and AWS each stood up dedicated forward-deployed engineering organizations — teams whose entire job is to sit inside enterprises and *make the thing actually ship*. When the companies building the models conclude that the hard part is deployment, not the model, that is your signal. The fight has moved to knowledge and delivery. And on both, you are closer to the front line than any outsider can be.

None of this means you build alone. It means the opposite: you build with the tools — Claude, or whichever model earns its place in your workflow — doing an enormous share of the labor beside you. The point is not self-reliance for its own sake. The point is that the capability can now live *inside* you and your team, permanently, instead of being rented for a quarter and wheeled back out the door with the knowledge still in the consultant's laptop.

But — and this is the whole game — you have to deliver. A memo about AI strategy is not delivery. A pilot that impresses in a conference room and never touches a real user is not delivery. The rest of this book is about *how*.

The economics already flipped in your favor

Here is the part the organization has not fully priced in: the barriers you are being asked to fear are far lower than the “yes, but” culture assumes.

Start with cost. According to Stanford’s AI Index, the price of running a model at the quality of 2022’s GPT-3.5 fell more than 280-fold between late 2022 and late 2024 — from roughly twenty dollars per million tokens to about seven cents. That is not an incremental improvement. It is a re-pricing so severe that most business cases written even a year earlier are simply wrong, in your favor. Projects dismissed as too expensive to run at scale often are not, anymore.

Then capability. Open-weight models — the ones you can run yourself, without a per-token bill to anyone — closed the gap with the best closed models on some benchmarks from around eight percent to under two percent in a single year. The practical meaning: you are no longer forced to choose between “good” and “affordable,” or between “capable” and “under our control.” That trade-off, which used to end a lot of internal debates, has mostly dissolved.

And access. The way models reach enterprises has converged onto infrastructure you likely already use. The major cloud providers are now the dominant route by which organizations obtain foundation models — a clear majority of enterprises buy them straight through their cloud. The same frontier model is frequently available across multiple clouds at once; through 2026 the last major exclusivity arrangements gave way to multi-cloud availability. Whatever your organization’s cloud footprint, the frontier is probably reachable from inside it.

This is the right moment to state the stance this book takes on vendors, because it falls directly out of that last fact. **This book does not tell you which cloud or which model is best.** Not out of diplomacy, but because the market itself has moved toward multi-provider neutrality, and because the durable skill lives in the layer *beneath* the vendor: how inference works, how you ground a model in your data, how you deploy and observe and control the cost of the thing. Learn that layer and you can move between providers as prices, capabilities, and corporate politics shift — which they will. Bet your skills on a single vendor's console and you have simply recreated the lock-in you spent the last decade trying to escape. Every reference in this book is chosen to teach the layer underneath.

So: cheaper than you were told, more capable than you were told, and reachable from where you already are. The barriers are real, but smaller than the deferral culture needs them to be.

Most still fail — and that is your opening

If the economics are this favorable, you would expect the results to be spectacular. They are not — and understanding *why* is the single most important idea in this chapter.

The adoption numbers are enormous. Stanford's AI Index put organizational AI use at 78% in 2024, up from 55% the year before; its 2026 update tracks generative AI specifically in at least one business function at around 70% of organizations. Nearly everyone is *doing* AI.

Almost none of it is *working*. MIT's Project NANDA, studying hundreds of real deployments, found that roughly 95% of organizations investing in generative AI saw no measurable impact on the income

statement — with only about 5% capturing real value at scale. RAND reports that over 80% of AI projects fail to reach production, roughly twice the failure rate of ordinary IT projects. S&P Global found that 42% of companies abandoned most of their AI initiatives in 2025, up sharply from 17% a year earlier. The exact figures vary because the studies measure different things — reaching production, moving the P&L, surviving the pilot — but they point the same direction. There is a wide, well-documented gap between adopting AI and getting anything out of it.

Now the crucial part. Read the post-mortems and the cause is almost never the model. It is legacy systems that will not integrate. It is data that was never ready. It is the absence of an evaluation loop, so nobody can tell whether the thing is getting better or worse. It is a pilot with no path to production because no one thought about deployment, observability, security, or cost until it was too late.

Read that list again. Integration. Data pipelines. Deployment. Observability. Cost control. Security. Those are not artificial-intelligence problems. They are *engineering* problems — the exact discipline you have been practicing your entire career. The reason 80% fail is that most teams treated AI as a modeling exercise and discovered, too late, that it was a distributed-systems exercise wearing a modeling costume.

This is your opening. The failure rate is not a warning to stay away. It is a map of where the value is trapped, and the trap is made of precisely the skills you already have. The organizations in the successful 5% are not the ones with access to a secret model. Everyone has the models now. They are the ones who could *operate* one — who brought engineering discipline to a problem everyone else treated as magic.

You are already qualified to be in the 5%. What you are missing is not talent. It is a specific, learnable mapping from what you already know onto a new surface — which is the rest of this book.

Experiment or die

There is only one way to learn this, and it is not by reading.

Reading about AI makes you conversant. You will be able to hold the meeting, use the vocabulary, nod at the right moments. It will not make you capable. Capability comes from building something small and real, watching it behave in ways you did not predict, breaking it, and building it again. The gap between the two is enormous, and it is invisible from the outside until the moment you are asked to deliver and discover which side of it you are on.

So the operating stance of this book — and, I would argue, of every corporate technology job from here forward — is simple: **experiment or die.** Not as melodrama. As a description of the actual condition. The people who will be fine are not the ones who understood AI earliest. They are the ones who *built* with it earliest, accumulated scar tissue earliest, and turned a hundred small experiments into judgment nobody can rent.

This book is built to serve that stance. It is deliberately self-contained: you can start from where you are, without a platform team blessing a standard first. It stays at the level of concepts and architecture — the *why* and the *shape* of things — because that is what survives the next model release. And it is paired with a companion site, **leapfrog.lerias.org**, which carries the hands-on layer: the labs, the starters, the working code you go and break. The

division of labor is clean. The book tells you what to build and why it holds up. The site is where you build it.

Every chapter from here ends with a short section called **Experiments to run** — small, concrete, shippable things, each tied to something on the site. They are not homework. They are the whole point. The chapters exist to make the experiments make sense; the experiments exist to make the chapters real.

One honest warning before you start, because it is the one genuinely new muscle. Everything you know about engineering assumes determinism: the same input yields the same output, and when it does not, you have found a bug. Models do not work that way. The same prompt can yield different answers; "correct" becomes a distribution rather than a value; and debugging shifts from *finding the broken line* to *reasoning about a system that is allowed to surprise you*. This will feel wrong at first. It is not wrong — it is new, and it is learnable, and by the end of this book it will feel like just another property of the system, the way eventual consistency once did. But name it now, so that the first time a model does something unrepeatable, you recognize it as the territory rather than a failure of your understanding.

The train has left the station. The good news, the whole reason this book exists, is that you were always qualified to be on it. You just have to move.

Experiments to run

Small enough to start today. Companion labs and starters at leapfrog.lerias.org.

1. **Spend $5 and a lunch break.** Get an API key from any one provider your organization already touches, and make a single call from your laptop that sends a real internal question and gets an answer back. The goal is not the answer — it is to feel, viscerally, how low the barrier to entry actually is compared to what the "yes, but" culture told you.

2. **Find your smallest real task.** Identify one genuinely small, genuinely real piece of work in your world — a report nobody enjoys writing, a triage step, a lookup — that a model could plausibly help with. Do not build it yet. Just write down what "delivered" would mean: who would use it, and how you'd know it worked. You are practicing the scarce skill — *knowing what to build* — before the easy one.

3. **Break something on purpose.** Ask a model the same question five times and keep the answers side by side. Watch it be non-deterministic. Sit with the discomfort. This is the new muscle; better to meet it in a throwaway experiment than in production.

4. **Write your own "yes, but," then answer it.** Take the objection you personally expect to hear first when you propose building something with AI at

> work, and draft the two-sentence rebuttal. You will use it sooner than you think.

References for this chapter

Independent and primary sources, weighted to keep the vendor-neutral stance honest. Full links maintained on leapfrog.lerias.org.

On adoption, economics, and the state of the field - Stanford HAI, *AI Index Report 2025* and *AI Index Report 2026* — inference-cost decline, open-weight convergence, adoption rates, hardware and capex trends. - McKinsey, *Global Survey on AI, 2025* — adoption versus realized EBIT impact. - MIT Project NANDA, *The GenAI Divide: State of AI in Business, 2025* — the pilot-to-value gap. - RAND Corporation — research on why AI projects fail to reach production. - S&P Global Market Intelligence, *Voice of the Enterprise, 2025* — initiative abandonment rates. - Deloitte, *State of AI, 2026* — hybrid-cloud and hyperscale adoption patterns.

On the cloud–frontier-lab convergence (read neutrally, across vendors) - U.S. Federal Trade Commission, *Staff Report on AI Partnerships* (6(b) study of the cloud-provider / model-developer relationships) — a non-vendor account of the Microsoft–OpenAI and Amazon/Google–Anthropic arrangements. - Primary partnership and availability announcements from AWS, Microsoft, Google Cloud, and the model developers — used only for factual availability, not as endorsements. - Synergy Research Group — cloud infrastructure market-share data.

On the shift of delivery in-house - Public announcements of forward-deployed / applied-engineering organizations from OpenAI, Anthropic, Microsoft, and AWS through 2026 — cited as evidence that deployment, not modeling, is the constraint.

02 Where AI Belongs

CHAPTER 2

There is no Chief Spellchecker Officer. There is no spellcheck budget, no spellcheck strategy deck, no spellcheck center of excellence. Spellcheck is everywhere and nobody thinks about it — which is exactly where AI is going, and exactly why building a chatbot is aiming too low.

Chapter 1 argued that AI is a *must* and that you're already qualified to deliver it. This chapter answers the question that comes next and that the rest of the book quietly depends on: deliver it *where?* Point it at what?

It matters because the technical chapters ahead — patterns, retrieval, agents — are easy to read as instructions for building a chatbot. They aren't, and if you finish this book having built a very good chatbot, you will have aimed at the wrong target. So before any of the "how," this chapter fixes the "where," and it does so on the back of a decade of research rather than anyone's opinion.

The spellchecker end-state

Start with a thought experiment. Spellcheck is one of the most widely deployed pieces of language technology ever built. It sits in every text field you touch, quietly correcting billions of words a day. And notice what *doesn't* exist around it: no executive owns it, no budget line funds it, no consultant is retained to advise on spellcheck maturity. You would notice spellcheck only if it vanished. It is embedded, ambient, invisible — a capability, not a product.

That is not a quirk of spellcheck. It is the fate of a whole category of technology that economists call *general-purpose technologies* — the class that includes electricity, the internal combustion engine, and the internet. They share three properties: they're pervasive, used as an input across nearly every sector; they keep improving; and they make everything downstream more productive. The argument, made most prominently by Erik Brynjolfsson and colleagues at Stanford, is that AI now belongs in that category — that it is transitioning from a bespoke set of tools into a general-purpose technology *from which applications are assembled*. Andrew Ng compressed the same idea into four words that stuck: AI is the new electricity.

Sit with what that implies, because it's the whole orientation of this chapter. If AI is a general-purpose technology, then intelligence becomes a commodity input, wired into everything, mostly out of sight. The end-state is not a room you visit to talk to an AI. It is AI woven into every system and every workflow and every text field — the spellchecker, generalized. And there's a second, deeper layer that answers a question you may already be asking: AI doesn't stop at being embedded *in* products. As it matures, it starts *building and operating* them — writing the systems (the AI-assisted development of Chapter 4) and running them (agentic operations). AI moves from

feature, to substrate, to the thing that builds and runs the substrate. Intelligence as ambient infrastructure.

If that is the destination, then what you build now should point toward it — toward embedding, not visiting. Which is exactly where most organizations go wrong.

The chatbot trap

Ask almost anyone to picture "using AI" and they picture a chatbot: a box you type into that types back. It's the most visible form, the first demo everyone builds, the thing that made the technology feel real. It is also, in a corporation, usually the *lowest*-value form — and mistaking it for the destination is the single most common way to aim too low.

The reason is structural. A separate chatbot sits *beside* the work. To use it, a person has to stop what they're doing, switch windows, phrase a question, wait, and copy the answer back into the actual task. Every one of those steps is friction, and the value leaks out in the handoffs. The chatbot makes the human do the integration work by hand, one query at a time.

The grounded alternative is now stated plainly in the enterprise research: value shows up when AI sits *where the work already happens* — embedded in the system the employee is already in, reducing handoffs, surfacing the right information, drafting, flagging exceptions, and suggesting the next step, all in the flow. The example that makes it concrete is a humble one. Instead of a chatbot you can ask about invoices, you build AI that reads each incoming invoice, checks it against the approved record, and routes only the mismatches to a human. Nobody "uses" it. It's simply how invoices

work now. That is the spellchecker pattern, applied to a business process: embedded, ambient, and valuable precisely because no one has to go to it.

None of this makes the chatbot useless. It is excellent *training wheels* — the fastest way to prototype, to feel a model's behavior, to learn the patterns in the chapters ahead. That's why the early experiments in this book are chatbot-shaped; you start there because it's the shortest path to touching the technology. The trap is only in mistaking the training wheels for the destination. So carry this reframe through every build chapter that follows: don't ask "how do I build a chatbot about X." Ask "how do I embed intelligence into the workflow that does X."

The jagged frontier: where AI works, and where it hurts

Embed AI into workflows, then — but into which *parts*? Not all of a workflow should touch a model, and the boundary is emphatically not where intuition places it.

Here the evidence is unusually good. In a large preregistered field experiment, researchers from Harvard Business School and BCG, together with colleagues from MIT Sloan and Wharton, had 758 management consultants perform realistic tasks with and without a frontier model. Inside a certain boundary the effect was transformative: consultants completed more tasks, far faster, at markedly higher quality — on the order of a quarter faster and forty percent better. But step *outside* that boundary and the sign flipped. On tasks beyond it, consultants using AI were meaningfully *less* likely to reach the correct answer than those working without it.

The researchers named the boundary the **jagged technological frontier**: AI excels at some tasks and fails at others of *similar apparent difficulty*, sometimes within the very same workflow, and the edge juts inward and outward in ways human intuition does not predict. You cannot reason "this task is harder, so it's harder for AI." The frontier is jagged, not a smooth gradient — which is precisely what makes it dangerous, because you can't feel where it is.

And the failure mode at the edge is the worst kind. Outside the frontier, AI does not fail loudly or obviously. It produces confident, polished, well-structured work that happens to be *wrong* — faster than a human would produce the right answer. This is the silent failure of Chapter 5 wearing a business suit: the output looks like success, which is exactly why it slips through.

So the method — grounded in MIT Sloan's more recent work on how AI actually lands in organizations — is not "add AI to this job." It is: decompose the workflow into its component tasks, and assess each task independently against the frontier. Where does AI clearly help? Where does it clearly hurt? Where must a human hold the pen? Then design the sequencing and the handoffs between human and machine deliberately, because the research is clear that AI's biggest impact comes not from any task in isolation but from how tasks are managed, grouped, and handed off. You are not bolting a tool onto a role. You are redrawing a workflow, task by task.

One more grounded nuance worth carrying: the lift is generally largest for *less*-experienced people. A Stanford and MIT field study of thousands of customer-support agents found average productivity gains around fourteen percent, rising to roughly thirty-five percent for the newest and lowest-skilled workers. AI tends to raise the floor more than the ceiling — an "inverse skill bias" — and that, too, shapes where in an organization it belongs.

Two patterns for sharing the work

Within the frontier, the same consulting study surfaced two distinct and durable patterns for how a human and an AI actually divide a task. They're worth naming, because they are the atomic shapes of AI-in-a-workflow.

The **Centaur** pattern divides and delegates. Like the mythical half-human, half-horse, there's a clean seam: the person does the parts of the work on their side of the frontier and hands whole tasks across to the AI on the other side. You delegate the literature search and keep the judgment call; the boundary between the two is sharp and deliberate.

The **Cyborg** pattern interweaves. Here the human and the machine blend within a task, in a tight loop — prompt, correct, extend, redirect, iterate — with no clean dividing line. The work is co-produced move by move.

Neither is correct in the abstract; they suit different tasks. Centaur fits work that decomposes cleanly, where you can name which subtasks belong to which party. Cyborg fits work where human and model sharpen each other continuously — drafting, coding, analysis. Most real designs combine them. What the pair gives you is vocabulary for *how* a human and a system share a task, once the jagged frontier has told you *which* tasks they should share at all.

The embedding ladder

Put the whole trajectory on a ladder, because it's how a single workflow matures over time.

Assistant is the first rung: a chatbot beside the work. Fastest to build, easiest to adopt, lowest value, and — as we've seen — a trap if you stop there. Genuinely useful for open-ended help and for learning, but not a destination.

Embedded is the second rung: intelligence *inside* the system where the work happens, surfacing information, drafting, flagging, and suggesting the next step in the flow, so that no one has to "go to" it. This is where most durable enterprise value lives today. It is the spellchecker rung.

Agentic is the third rung: the system runs the workflow itself — planning, calling tools, executing — with a human *on* the loop rather than *in* it. This is the frontier that corporations are climbing toward now, and it's exactly where the reliability discipline of Chapter 6's "workflows versus agents" and the operating chapters becomes non-negotiable, because an unreliable agent embedded in a core process fails at scale.

The rungs are not better or worse in the absolute; they're a maturity path. You often start at rung one to learn, but you should always know which rung a given workflow *ought* to reach, and build toward it deliberately. For spelling, we long ago climbed to rung two-and-a-half — embedded, largely autonomous, with a human able to override. Chapters 6 through 9 are, in large part, how you climb that ladder for your own workflows without falling off it.

Co-invention: what separates the 5% from the 95%

Now the finding that should reframe how you think about every AI project you ever scope, and that closes the loop with Chapter 1's uncomfortable statistics.

Stanford's Digital Economy Lab, in its *Enterprise AI Playbook*, and McKinsey's enterprise survey data land on the same conclusion, and it is blunt: AI success is not about deploying better models. It is about *redesigning how the organization works*. The companies actually capturing financial return are roughly three times more likely to have fundamentally redesigned their workflows around AI — on the order of fifty-five percent of high performers versus twenty percent of everyone else — only about a fifth of organizations have done it at all, and workflow redesign is the single factor most correlated with realizing value.

This is Brynjolfsson's *productivity J-curve*, and it is the same story electricity told a century ago. Factories did not get more productive by bolting electric motors onto layouts designed around a central steam shaft. Productivity came only once they *redesigned the factory* around distributed power — a re-invention that took decades. The value of a general-purpose technology is unlocked by the complementary re-invention around it, not by the technology alone. Economists call that complementary work *co-invention*, and it is the expensive, unglamorous, organizationally hard part. AI's J-curve is expected to be shorter than electricity's — partly because AI can help implement itself — but the shape is identical: you invest in the redesign *before* the returns appear, which is why the returns feel slow and why the impatient give up.

The implication is uncomfortable and clarifying at once. Bolting AI onto an unchanged workflow is the *default* failure mode — it is the mechanism by which organizations join the ninety-five percent that see no measurable return. Value is not the model. Value is the model *plus* the complementary investment: the redesigned process, the new handoffs, the retrained people, the better data. And the model, as Chapter 5 showed, is now the cheap part. The co-invention is the work, and the willingness to do it is precisely what the five percent have and the ninety-five percent don't.

Which turns the whole question inside out. The right question was never "where can I add AI?" It is "**which workflow am I willing to redesign around AI?**" — because the redesign is where the value lives, and asking it that way filters out the bolt-on projects before they waste a year.

Choosing your first workflow

So here is the on-ramp, and it is deliberately concrete, because it feeds the first real thing you'll build. Picking where to start is a short, disciplined method:

Choose a *workflow*, not a task and not a chatbot — you're redesigning a flow of work, not adding a feature. Decompose it into tasks and map each to the jagged frontier: AI clearly helps, AI clearly hurts, or human-only. Then pick a workflow where the helpful gap is large *and measurable*, and — this is the part people skip — where you actually have the authority to change the process, because co-invention you're not allowed to do is co-invention that won't happen. Decide the collaboration pattern (Centaur or Cyborg) and the rung you're aiming for (embedded, ideally, not assistant). And define what

"delivered" and "better" mean *before* you build, so you can tell whether the redesign worked.

That is the leapfrog move at the level of the organization. Don't wait for a corporate AI strategy to descend from on high. Pick one workflow you're allowed to rebuild, and rebuild it around embedded intelligence. That single redesigned workflow — not a model, not a chatbot, not a strategy deck — is the atomic unit of AI value. Deliver one, and you've done the thing the ninety-five percent never manage.

Experiments to run

Small enough to start today. A workflow-mapping worksheet at leapfrog.lerias.org.

1. **Run the spellchecker test.** Take an AI idea someone in your organization is excited about and ask: would this end up with a Chief Something Officer and a budget line, or would it just disappear into how the work gets done? If the former, you're probably staring at a chatbot. Go find its embedded version.

2. **Map a workflow to the jagged frontier.** Pick one real workflow, break it into its tasks, and label each: AI clearly helps, AI clearly hurts, or human-only. The labels will not match your first instinct — and that surprise is the entire point.

3. **Find the redesign, not the add-on.** For that same workflow, write how it would run if it were *rebuilt*

around AI — different steps, different handoffs — rather than AI stapled onto today's version. That description is the co-invention. It's the actual work, and most teams never write it down.

4. **Name the rung.** For a feature you're considering, decide honestly which rung it should live on — assistant, embedded, or agentic — and whether what you're about to build is aimed there or quietly stuck on rung one.

References for this chapter

Primary academic and institutional sources throughout — this chapter is grounding, so it leans on the research directly. Worksheets on leapfrog.lerias.org.

The jagged frontier and the collaboration patterns - Dell'Acqua, McFowland, Mollick, Lifshitz-Assaf, Kellogg, Rajendran, Krayer, Candelon & Lakhani, *Navigating the Jagged Technological Frontier* (Harvard Business School / BCG, with MIT Sloan and Wharton co-authors; Harvard Working Paper 24-013, 2023; published in *Organization Science*) — the frontier, and the Centaur and Cyborg patterns. - The follow-on BCG/HBS skill-development experiments on upskilling and the inverse skill bias.

AI as a general-purpose technology - Brynjolfsson, Rock & Syverson on the AI productivity paradox and the productivity J-curve; the classic general-purpose-technology properties (Bresnahan & Tra-

jtenberg). - Andrew Ng, “AI is the new electricity” (Stanford Graduate School of Business).

Enterprise value and co-invention - Stanford Digital Economy Lab, *The Enterprise AI Playbook* (Pereira, Graylin & Brynjolfsson, 2026) — workflow redesign as the factor most correlated with return. - McKinsey, *State of AI* — high performers and fundamental workflow redesign. - MIT Sloan — task-level management and the shift from individual-productivity to enterprise-workflow use cases (Davenport & Bean, and related 2026 research). - The Stanford/MIT customer-support field study (Brynjolfsson, Li & Raymond) — average gains and the inverse skill bias.

03 The Landscape Without the Hype

CHAPTER 3

"Which model is best?" is the wrong question. It has a new answer every month — and the answer is never "the same one you picked last month."

The last chapter argued that the hard part of AI at scale is engineering, not the model. This chapter is about the trap that sits one level up: the belief that before you can build anything, you first have to *pick the right vendor* — the right cloud, the right lab, the right model — and that picking wrong is fatal.

It isn't. And the effort you spend trying to pick a winner is effort not spent shipping. This chapter gives you a map of the landscape so you can stop staring at it, and a single architectural stance that makes the whole "who's winning" question stop mattering to your day job.

The war that isn't

If you read the trade press, the model landscape looks like a heavyweight title fight: a handful of giants trading blows, each new release declared a knockout, each leaderboard a verdict on who you should bet your company on.

Look at the actual data and the fight dissolves.

Independent testing across hundreds of models through 2026 found the same thing repeatedly: no single model wins. Leadership rotates by task. One flagship leads at fixing bugs, another at writing new programs, another at system administration — and the same model that tops one category can fall well down the pack in the next. On the broad knowledge and reasoning benchmarks that get the most headlines, the top models have clustered so tightly that the gaps between them are smaller than the run-to-run variation of the tests themselves. When the difference between first and fifth place is smaller than the measurement noise, the ranking is not telling you anything. The honest summary from the people running those tests is blunt: you can no longer pick one model and use it for everything.

Now layer on speed. In a single thirty-day stretch in the spring of 2026, every major lab moved at once — new flagships from the largest closed labs, a steep price cut from a leading open-weight challenger, and a surprise benchmark-topping release from another. That was not an unusual month. That is the tempo. Models now also ship less like products and more like *bundles of settings* — reasoning mode on or off, small or large context, tool access or not — so even "which model" fragments into "which configuration of which model for which task."

Sit with what this means for a book, or for an architecture. Anything that hard-codes today's winner is wrong by the time it's read. The durable skill is not knowing which model leads this quarter. It is knowing how to *read* the landscape and re-read it cheaply, forever. That is what the rest of this chapter builds toward — and it is why, from here on, this book names families and categories, not version numbers. The specifics live on the companion site, where they can be kept current; the judgment lives in you.

Meet the players — without ranking them

You still need a mental map. Just not a scoreboard. Here is the landscape by *kind of thing*, which changes far more slowly than the rankings do.

The closed frontier labs. A small number of well-capitalized labs ship proprietary flagship models you reach through an API, not by downloading anything. As a group they tend to hold a slight edge on the very hardest reasoning and the most complex multi-step agentic work, and they come with the largest tool ecosystems and the most enterprise plumbing around them. You trade control for capability and convenience: you cannot see the weights, you cannot run them on your own hardware, and you are exposed to their pricing, their deprecations, and their availability. Think of them as managed intelligence, rented by the token.

The open-weight tier. A parallel ecosystem publishes model weights you can download, inspect, fine-tune, and run yourself. Two years ago this tier was a clear step behind the frontier. It is not anymore. Stanford's AI Index tracked the gap between the best open and best closed models narrowing from around eight percent to under two percent on some benchmarks in a single year, and by

2026 several open-weight families sat within striking distance of the closed flagships, with one ranking among the top handful of models overall. The names in this tier rotate — a lineage from Meta, several strong entrants from Chinese labs, a European contender, and others — but the *category* is stable and strategically important: open weights are the answer whenever control, data residency, on-premises operation, or high-volume cost sensitivity outweighs the last few points of frontier capability.

The specialists and the cheap tier. Alongside both groups is a fast-growing middle: smaller, faster, dramatically cheaper models that are more than good enough for the bulk of real work — classification, extraction, routing, summarization, first-draft generation. The most important landscape shift of 2026 was not at the frontier at all. It was the explosion of this affordable tier, which is what makes a multi-model strategy practical rather than academic. Most of what your applications do does not need the smartest model in the world. It needs a competent one that costs a fiftieth as much.

Notice that this map has no "best." It has trade-offs — capability versus control, frontier versus cost, convenience versus portability — and the right point on those trade-offs is different for different tasks *inside the same application*. Which is exactly why the interesting decision is architectural, not a purchasing choice.

How access actually happens

Between you and any of those models sits a delivery channel, and the channel matters as much as the model. There are four, arranged roughly from most convenient to most controlled.

First-party APIs. You call the lab directly. Simplest to start, newest features first, and a direct relationship — but also the tightest coupling to one vendor's endpoints, pricing, and rate limits.

Cloud-hosted model services. The major clouds each offer a service that resells many models — first- and third-party — through their own platform, inside their own security, billing, networking, and compliance envelope. This is now the dominant way enterprises actually obtain foundation models: a clear majority procure them through a cloud provider rather than direct. The appeal is obvious for a corporate engineer — the model shows up inside the IAM, the VPC, the audit logging, and the procurement contract you already have. The same frontier model is frequently available across all three major clouds at once, which is itself a signal: the platform layer has converged, and your existing cloud footprint almost certainly already reaches the frontier. You do not need a new vendor relationship to start. You need to look inside the one you have.

Gateways and aggregators. A middle layer — sometimes a hosted service, sometimes open-source software you run — exposes one unified interface to hundreds of models across many providers. You integrate once; the gateway translates and routes. This is the pattern the next two sections are built around, so hold it.

Self-hosted open weights. You run the model yourself, on your own or rented GPUs. Maximum control, maximum residency guarantee, no per-token bill to a lab — and maximum operational burden. You are now running the model *and* the serving infrastructure, which is a real engineering commitment (Chapters 5 and 8 are largely about what that commitment involves).

Most mature setups use several of these at once: a cloud-hosted service for the sanctioned default, a gateway to keep options open,

and self-hosting for the workloads where control is non-negotiable. The point of knowing the taxonomy is that "getting access to AI" is not one decision. It is four, and you can make them differently per workload.

The one decision that matters

Here is the decision that actually deserves your attention, and it is not "which model."

It is: **how do I avoid being trapped by whichever model I pick?**

Because you *will* pick wrong, in the narrow sense — not through bad judgment, but through the passage of time. The model you choose today will be surpassed, repriced, deprecated, or made unavailable, and probably within the year. The question is whether that event is a configuration change or a crisis.

The evidence that most teams are unprepared for it is stark. A 2026 survey found 94% of organizations worried about vendor lock-in. Another found a revealing gap: 89% of enterprises believed they could switch AI providers, but of those who actually tried, 58% hit failures or unexpected difficulty. The confidence is not matched by the architecture.

And the failures are not hypothetical. When one provider abruptly suspended access to a model in 2026, teams that had wired it straight into production pipelines had no fallback and were simply down. In another documented case, a company that built on a single proprietary platform spent six figures and three months migrating forty workflows after that platform collapsed — with customer-facing features degraded throughout. In every one of these stories, the same architectural absence is the cause: no layer between the ap-

plication and the provider, so a model change or an outage becomes an all-hands engineering incident instead of a flipped switch.

There is also a quieter, newer trap worth naming, because it will catch careful teams. Agentic workflows re-introduce lock-in through the back door. As you build guardrails, tune prompts, and shape tool use around one model's specific behavior, switching stops being free even if your API abstraction is clean — because the *behavior* you've engineered around is vendor-specific. The more sophisticated your AI system, the more subtly it can bind you. You cannot eliminate this, but you can be aware of it and keep the coupling loose on purpose.

Lock-in, not model choice, is the thing that actually costs enterprises money and time. So the neutral stance this book takes is not fence-sitting. It is the risk-management position that the data supports.

The neutral architecture

The good news is that staying neutral is a well-understood pattern, and it is one a cloud-native engineer will recognize instantly, because it is the same move you have always made to avoid infrastructure lock-in: put an abstraction layer in the middle.

For models, that layer is usually called an AI gateway or model router. Think of it as a load balancer for intelligence. Your application talks to one unified interface. Behind it, the gateway holds credentials for many providers and models, translates your request into each one's format, and decides where to send it. Your prompt templates, business rules, and evaluation logic never learn whether they're talking to a closed frontier model or a self-hosted open one.

That separation is the whole game, and it buys you four things at once.

Portability. Switching providers becomes a config change, not a rewrite. When a better or cheaper model appears — monthly, remember — you can route to it without touching application code. This is the point that shows up in the survey data as the difference between the teams that switch smoothly and the 58% who don't.

Resilience. With more than one provider behind the gateway, an outage at one becomes an automatic failover to another rather than a P1 incident. Fallback chains and circuit breakers, familiar from ordinary distributed-systems work, apply directly. Teams running multi-provider setups routinely report near-continuous uptime that no single provider guarantees.

Cost control. Because the gateway sees every request, it can route by policy: send the easy, high-volume tasks to the cheap tier and reserve the expensive frontier models for the genuinely hard ones. This one lever alone commonly cuts inference spend by a meaningful fraction without any loss of quality where quality matters — and it is only possible once you've stopped hard-coding a single model. Add semantic caching, and repeated or near-repeated requests stop costing anything at all.

Observability and governance. One choke point for every model call is also the natural place to put logging, spend tracking, rate limits, access control, and prompt-injection defenses — a single pane of glass over all AI traffic, which becomes essential the moment more than one team is building.

Under the gateway, lean on open standards wherever they exist, because they are portability at the format level: an open protocol for

connecting models to tools and data, open formats for model weights and for data, and open instrumentation for telemetry. Standards outlive vendors. Every place you use one is a place you are not rewriting later.

You do not have to build any of this from scratch — there is a whole category of gateways, hosted and self-hosted, and Chapter 8 gets concrete about running one. The lesson here is architectural: one thin layer, placed deliberately, converts the entire "who's winning the model war" question from a strategic bet you're afraid of getting wrong into an operational detail you adjust on a Tuesday. Gartner expects this to go mainstream fast — from under 5% of multi-model teams using gateway capabilities in 2024 to around 70% by 2028. You can be early, cheaply.

Keeping your map current

Because the landscape moves monthly, the last durable skill this chapter teaches is not a fact but a *habit*: a cheap, repeatable process for re-reading the map so it never goes stale on you.

Watch a small number of independent, benchmark-driven trackers rather than vendor announcements — the point of independence is that no one on the list is trying to sell you the winner. Re-ask the same short set of questions on a fixed cadence, quarterly is plenty: Has a new model changed the price-performance frontier for any task I actually run? Has anything I depend on been deprecated or repriced? Could I still switch my primary provider with a config change — and when did I last test that? Has my agentic tuning quietly bound me to one model's behavior? None of these require you to chase every release. They require you to *notice*, on your schedule instead of the vendors'.

The companion site keeps a living version of this landscape — the current families, the current trackers worth watching, and the questions above as a checklist — precisely so that this chapter can stay at the level of judgment while the facts stay current somewhere they can be edited. The map changes. The way you read it does not.

That is the whole posture of this book in miniature: don't bet the company on a vendor, and don't wait until you're certain which one wins. Put in the thin neutral layer, start building on top of it today, and let the frontier move underneath you while you keep shipping.

Experiments to run

Small enough to start today. Companion labs and a live landscape page at leapfrog.lerias.org.

1. **Same prompt, two providers, one interface.** Put a gateway (hosted or a lightweight self-hosted one) in front of two different providers and send the same real internal prompt to each. Compare the answers, the latency, and the cost side by side. The goal is not to crown a winner — it's to feel how little your code cares which one answered.

2. **Route by difficulty.** Send an easy, high-volume task (say, classifying or summarizing) to a cheap small model, and a genuinely hard one to a frontier model. Measure the cost difference. That gap, multiplied by your real volume, is the money a multi-model

strategy leaves on the table when you hard-code one model.

3. **Run the exit test.** Ask honestly: if your main provider doubled its price or pulled your model tomorrow, is switching a config change or a multi-sprint project? Then actually try to reroute one workload and time it. Discover the answer now, on your terms, not during an incident.

4. **Break a provider on purpose.** Wire up a fallback chain, then pull the credentials for your primary provider and watch traffic fail over to the backup. Resilience you haven't tested is a resilience you don't have.

References for this chapter

Independent and primary sources first; tools named only as examples of a category, never endorsements. Live specifics maintained on leapfrog.lerias.org.

On the state of the models (read as instruments, not scoreboards) - Independent benchmark and comparison trackers — e.g. Artificial Analysis, LMArena, LLM-Stats — for live, cross-provider performance, price, and latency. - Cross-model testing showing task-dependent leadership and statistically tied knowledge benchmarks (the "no single winner" analyses of 2026). - Stanford HAI, *AI Index* — the closing gap between open-weight and closed models.

On access and the cloud channel - U.S. FTC, *Staff Report on AI Partnerships* — a neutral account of the cloud-provider / model-developer relationships. - Synergy Research Group — cloud infrastructure market share. - Provider documentation for the cloud-hosted model services — used only to describe the access taxonomy, not to rank platforms.

On lock-in and the neutral architecture - Gartner — adoption forecasts for AI gateways; the gateway definition as an abstraction layer for AI traffic. - TechTarget and other practitioner guides — best practices for avoiding AI vendor lock-in. - Enterprise lock-in and switching surveys (2026) — the gap between believed and actual portability. - Open standards, cited directly: the Model Context Protocol (tool/data connectivity), ONNX (model portability), OpenTelemetry (observability), and open data formats for portability. - AI gateway / model-router tooling as category examples: LiteLLM, OpenRouter, Kong AI Gateway, Portkey, and the cloud-native routers — evaluate against your own architecture, never adopt on reputation.

04

Cloud Foundations for the AI Engineer

CHAPTER 4

You can run a capable model on your laptop tonight. You cannot ship it from your laptop. Everything between those two facts is this chapter.

The last two chapters were about *why* and *what*. This one is about the ground you build on — and it starts by dissolving a false choice that traps a lot of engineers before they begin: the idea that "doing AI" means either a toy on your machine or a giant cloud programme, with nothing in between.

The truth is simpler and more useful. You start local because it's the fastest on-ramp in the history of the field. Then you discover that nothing you actually *deliver* stays local — there is always a host — and the machinery that runs whatever you ship, wherever you ship it, is the cloud control plane and software lifecycle you already know. Learn to see AI through that lens and the whole thing stops being foreign.

Start local — it's the best on-ramp

The barrier to your first real experiment is one command.

Local model runtimes have matured into what is fairly described as *Docker for language models*: you pull a model by name, and the tool handles the download, the memory management, the hardware acceleration, and — critically — exposes the model behind a small local web API that mirrors the format the big cloud providers use. A capable open-weight model in the single-digit-billion-parameter range, quantized down to a few gigabytes, runs comfortably on an ordinary laptop, silently, on battery, offline, for free. Apple's unified-memory machines punch well above their price here; a modest used GPU can replace a monthly API bill within months.

This matters for exactly the reason the whole book keeps returning to. Starting local removes every institutional obstacle at once. No cloud account. No procurement. No security review before you're allowed to send a single prompt. The distance between "I wonder if a model could help with this" and "I just tried it on real data" collapses to an afternoon. This is where *experiment or die* literally begins, because it's the one place you can build with zero permission.

It's also genuinely useful, not just a sandbox. Local is the right tool for prototyping, for iterating on prompts a hundred times without a bill, for putting sensitive data through a model without it ever leaving your machine, and for running two or three models side by side to feel the differences from Chapter 3 with your own hands. And because the local runtime speaks the same API dialect as the hosted providers, the mature architecture is often *hybrid* — route routine work to a small on-device model, fall back to a cloud model for the

hard cases — with the switch between them being a line of configuration rather than a rewrite.

Hold on to that last point, because it's the hinge of the entire chapter.

But nothing real stays local

The moment anyone other than you needs the thing, local ends.

A teammate wants to use it. A scheduled job has to call it at 3 a.m. A product feature depends on it being there. The instant that happens, the requirements change from "runs on my machine when I'm looking at it" to "reachable, available, authenticated, maintained, and up when I'm asleep." Your laptop is none of those things. So the model, or the app around it, gets wrapped in a container, placed behind a stable endpoint, and deployed somewhere that stays running — with authentication in front of it, scaling behind it, and a way to update it without taking it down.

That somewhere is a host. And here is the point worth saying plainly, because it quietly settles a lot of confused debates: **there is always a hosting.** "Local versus cloud" is the wrong axis. Local is where you *develop*; a host is where you *deliver*; and the same OpenAI-shaped API that let you swap a local model for a cloud one also means your application code barely notices the move. The app is portable. The *hosting is not optional*. Something, somewhere, always runs it — with an address, an owner, a bill, and a policy.

So the durable skill this chapter teaches is not a model runtime. Runtimes come and go. It's the thing that runs *whatever* you deliver, *wherever* it runs: the control plane, and the lifecycle around it. That's

the rest of the chapter, and it's almost entirely made of things you already know.

The control plane you already know

Strip away the AI vocabulary and the control plane for an AI system is the same set of primitives you've used for years. Four of them here; the fifth — how you define and reproduce all of it — gets its own section because it's what makes experimentation safe.

Compute, plus one new primitive. Virtual machines, containers, and serverless functions behave exactly as you expect. The only genuinely new element is the *accelerator* — the GPU or specialized chip that runs the model. And the good news for a corporate engineer is that you mostly don't touch it directly. Most of the time you *consume inference*: you call a managed endpoint and someone else runs the accelerator. You only meet the GPU face-to-face when you choose to self-host, and that's a deliberate trade, not a default. It's worth knowing that these accelerators are a genuinely scarce resource — the current generation was reportedly sold out well into 2026 against an enormous backlog, and inference is now the majority of all accelerator spending — because scarcity is what makes "just run it yourself" more expensive and more constrained than it sounds. When you *do* self-host, the industry has consolidated on a recognizable pattern: containers and Kubernetes underneath, a specialized inference engine to run the model efficiently, and a serving layer on top for autoscaling and routing — with the managed ML platforms used for experimentation and Kubernetes-based stacks taking over at production scale. You don't need that machinery to start. You need to know it exists and when to reach for it, which is Chapter 8's job.

Networking, with a meter running. Virtual private networks and private endpoints do here what they've always done: keep traffic — including your prompts and your data — off the public internet and inside a boundary you control. The AI-specific trap is the *egress meter*. Moving data costs money, and AI systems move a lot of it; a design that shuttles large volumes across regions or out of the cloud is a bill waiting to surprise someone. The old instinct applies with new force: **data has gravity.** Keep the compute near the data, not the other way around. If you came from the edge or CDN world, this is your native language — locality, latency, and caching were always the game.

Storage. Object storage is the backbone: it's where your documents live for retrieval, where model weights sit, where logs accumulate. Nothing exotic; the same durable, cheap bucket you already use, now holding the raw material an AI system reads from.

Identity. Least-privilege access to models and to data, exactly as for any other resource. The one habit to internalize immediately: an API key to a model provider is a *secret*, with all that implies — stored in a secret manager, scoped tightly, rotated, never in a repo. A model endpoint is just another privileged resource, and it gets governed like one.

None of this is new to you. That's the entire point. The control plane for AI is the control plane you already operate, with one new primitive bolted on.

The software lifecycle is the same lifecycle

AI does not get its own special software development lifecycle. It rides the one you already run — and recognizing that is what turns

"we need an AI initiative" back into "we ship software, and some of it now calls a model."

Source control is still the spine. What changes is how much of the system is now *text you version*. Prompts are text. The choice of model is configuration. Guardrails, retrieval settings, evaluation sets — all text, all belonging in the repository, all subject to review, diff, and rollback. The mental shift that saves teams the most pain is this: a prompt change is a code change. It can break production as thoroughly as a bad function, so it goes through the same door — a commit, a review, a history you can revert to.

CI/CD is still the pipeline. The same build-test-deploy machinery now also assembles and checks the AI parts of the system. You are not inventing a new delivery mechanism; you are extending the one you have.

Infrastructure-as-code is what makes the recklessness safe. Defining your environments in code — so any of them can be created, destroyed, and recreated identically — is the quiet hero of this whole book. It is precisely what lets *experiment or die* not become *experiment and leave a smoking crater*. You can stand up a full environment to try something, tear it down when you're done, and know you left no orphaned resources leaking cost and no snowflake configuration nobody can reproduce. Reproducibility is the safety net stretched under the experimentation. Without it, moving fast is genuinely dangerous; with it, moving fast is just moving fast.

There's a role shift hiding in here, and it's the same one Chapter 1 promised. As more of the system becomes text that a model can help you produce, the highest-value work moves *up the stack* — away from typing lines and toward framing the problem, designing

the system, validating the output, and governing the whole thing. Which brings us to the part you specifically shouldn't leave for last.

Shift governance left

The corporate default is to treat governance as a gate at the end: build the thing, then submit it to a review board that says yes or no weeks later. That gate is exactly the "yes, but" culture from Chapter 1 in institutional form, and it's where velocity goes to die.

The alternative is to **shift governance left** — to move the controls earlier, into the commit, the pull request, and the build, where problems are cheap to catch and cheap to fix. This isn't a philosophy; it's arithmetic. A governance violation can enter a codebase at four distinct moments — commit time, review time, build time, and release time — and each earlier catch is dramatically cheaper than the one after it. A secret caught in a pull request never reaches the main branch. A vulnerable dependency caught at build never ships. A bad prompt caught by a test never reaches a user.

Made concrete, shift-left governance is a set of automated checks distributed across the pipeline, expressed as *policy-as-code* so they're enforced by the machine rather than remembered by a person: secret scanning and dependency review on every change; a software bill of materials so you always know what's inside a release; provenance and attestation so you can prove how an artifact was built; and — the AI-specific one that ties into Chapter 9 — *evaluation gates*, where a change to a prompt or a model must pass a quality check before it's allowed to merge. You bind the whole set together with a single required status check: nothing reaches the main branch unless every applicable governance check is green. The rules live in the repository, run automatically, and fail loudly. Modern

governance frameworks even derive their default gates from a declared compliance posture, so the connection to formal standards is a configuration choice rather than a manual audit — but the *mechanism* is what matters here; the specific regulations are Chapter 11.

And this is where "there is always a host" comes back to bite. Every host has data policies, and those policies change. In 2026, a widely used AI coding assistant quietly shifted its lower tiers so that user interactions became training data *by default* unless you opted out, while its enterprise tier was exempt — a change that turned "do you train on our code?" into a question every regulated team suddenly had to ask of every host in its stack. The lesson isn't about one vendor. It's that the answer to that question is a governance fact you encode from the first commit, not a surprise you discover in a contract review after you've already shipped. You govern the hosting because there is always a hosting.

Done this way, governance stops being the thing that blocks delivery and becomes the thing that *enables* it: you move fast precisely because you can prove, at every step, that what you're moving is defensible. That is the opposite of the review board, and it is how you leapfrog the "yes, but" without being reckless.

The prerequisite short list

Here is the payoff, and it's deliberately short, because the whole argument of this book is that you do *not* need to master all of cloud before you deliver AI. You need this subset — and you can start acquiring it today, in parallel, without waiting for any migration to finish.

Need now, to deliver your first real thing: containers; a source repository with CI/CD; infrastructure-as-code for one reproducible environment; a way to call a model (a managed endpoint or a gateway from Chapter 3); secrets management for your keys; and one minimal governance gate on the pipeline.

Learn when you hit it, not before: Kubernetes and the self-hosting serving stack (only when you self-host at scale); GPU scheduling and the internals of accelerators; advanced multi-region networking; and the deeper reaches of model operations, which the next chapters open up as you need them.

That's it. That's the ground floor. The migration your organization keeps deferring is not a prerequisite for standing on it. Start local tonight, and know that the path from there to something real runs through machinery you already understand.

Experiments to run

Small enough to start today. Companion labs and setup guides at leapfrog.lerias.org.

1. **Run a model on your own machine tonight.** Install a local runtime, pull a small open-weight model, and call its local endpoint from a script. Total cost: zero. The point is to feel, physically, how low the on-ramp is.
2. **Turn "local" into "hosted."** Take that same little app and deploy it as a container behind a stable endpoint somewhere it stays up — however small. No-

tice the exact moment it stops being local, and notice that your application code barely changed. That's the "there is always a host" lesson, felt rather than read.

3. **Version a prompt like code.** Put a prompt in a repository, change it in a pull request, and require one automated check to pass before it can merge. That's governance-as-code in its smallest possible form — build the habit at toy scale.

4. **Destroy it and bring it back.** Define your environment entirely in infrastructure-as-code, tear the whole thing down, and recreate it identically from the code. The confidence that gives you *is* your license to experiment freely.

References for this chapter

Primary and vendor-neutral first; tools named only as examples of a category. Setup guides kept current on leapfrog.lerias.org.

Starting local - Local model runtimes as category examples — Ollama, LM Studio, llama.cpp — and the GGUF/quantization formats that make small models fit on consumer hardware. - On-device and hybrid (on-device plus cloud) architecture guidance for 2026.

The control plane and serving stack - The open serving stack as primary sources — vLLM, KServe, Ray — at awareness level (Chapter 8 goes deep). - Cloud providers' well-architected, networking, and

identity documentation — read across vendors for the patterns (private endpoints, egress, least privilege), not to rank platforms. - Kubernetes, and Terraform / Pulumi, for orchestration and infrastructure-as-code. - Stanford HAI *AI Index* and accelerator documentation for the hardware scarcity and cost trend line.

Shifting governance left - NIST — Secure Software Development practices extended to generative AI (SP 800-218A) and the AI Risk Management Framework — for the shift-left mechanisms, not the legal detail. - ISO/IEC 42001 — AI management systems — as the standards anchor. - Shift-left governance tooling and pipeline-gate patterns (policy-as-code, secret scanning, SBOM, attestation, evaluation gates) as category examples. - The 2026 AI-assistant data-policy episode as a governance case study in why hosting policy is a first-commit concern. Deep regulatory treatment is deferred to Chapter 11.

05

How Models Are Served and Consumed

CHAPTER 5

The model feels like magic until the invoice arrives. Then it feels like a metered utility — which is exactly what it is, and exactly what makes it something you can engineer.

Chapter 4 got you to a running model behind a control plane. This chapter is about what actually happens each time you call it — because until you can see the meter, the clock, and the ceiling, every AI feature is a gamble on cost, speed, and quality that you won't understand until it's in production and one of the three has gone wrong.

The good news is that all three are legible. A model is not an oracle; it's a service with predictable economics and physics. Learn to read them and you can look at a proposed feature and predict, before you build it, roughly what it will cost, how fast it will respond, and where its quality will quietly break. That predictive power is the difference between the teams that ship AI that survives and the ones whose pilots die on contact with a real bill and a real user.

The unit of everything: the token

Everything downstream is denominated in one unit, so start there. A *token* is a chunk of text the model processes as a single step — very roughly four characters, or three-quarters of a word, for English prose. Code tokenizes more densely, often half again as many tokens per line, because of all the punctuation and structure. And every provider uses a slightly different tokenizer, so the identical paragraph can cost a different number of tokens on different models — a subtlety that quietly distorts naive cross-provider price comparisons.

Two kinds of token get billed, separately. *Input tokens* are everything you send: the prompt, the instructions, the retrieved documents, the conversation history. *Output tokens* are everything the model generates back. And here is the fact to burn into memory, because half of cost engineering follows from it: **output tokens cost several times more than input tokens** — commonly somewhere in the range of two to eight times more — across essentially every provider.

That asymmetry isn't a pricing whim. It's physics. Your input can be processed in a single parallel pass — the model reads the whole prompt at once, a phase called *prefill*. Output cannot: each generated token requires its own full pass through the model, produced one at a time in sequence, a phase called *decode*. Generating is inherently more expensive than reading, and the price card simply reflects that.

The immediate consequence is that the *shape* of your workload sets its cost. A summarization task — long input, short output — is cheap per call. A generation task — short input, long output — is expensive,

even at the same total token count. Which means the single highest-leverage cost decision you make is almost never which provider to negotiate with. It's how many output tokens you let the model produce. Keep that in your pocket; we'll come back to it.

The bill is a design decision

Sticker prices span an enormous range — a couple of orders of magnitude between the cheapest small model and the priciest frontier reasoning model — and they keep falling fast, as Chapter 1 showed. But the sticker price is a ceiling, not what a well-built system actually pays. Three levers sit under it, and they stack.

Route by difficulty. Most of what an application does — classifying, extracting fields, formatting, routing, first drafts — does not need the smartest model in the world. Send that work to a cheap, fast, small model and reserve the expensive frontier models for the genuinely hard requests. Done well, this alone cuts spend by more than half, often much more. This is the same multi-model posture Chapter 3 built the gateway for; here it shows up as a line on the invoice.

Cache the repeated parts. Most prompts have a stable prefix that repeats on every call — the system prompt, the few-shot examples, a reference document. Providers can cache that prefix server-side and bill it at a steep discount, commonly well over half off, sometimes up to ninety percent off the cached portion. There's a second, quieter benefit: cached reads frequently don't count against your rate limits, so caching buys you throughput as well as savings. The catch is architectural — you only benefit if your prompts are stable and front-loaded, which is a design choice you make on purpose.

Batch what isn't urgent. For anything that doesn't need an answer this second — nightly reports, bulk classification, data enrichment — providers offer asynchronous batch processing within a window, typically at half price. This is close to free money, and most teams leave it on the table.

Stack caching and batching on routed traffic and your effective cost can fall to roughly a quarter of the on-demand rate. None of this is exotic; it's just deciding to do it.

One trap to name, because it's where teams fool themselves: **cost per token is not cost per task.** The cheapest model per token can be the most expensive per *job*, if it needs more retries, produces worse output that has to be regenerated, or forces you to stuff in more context. The number that matters to the business is cost per unit of work — per transaction, per resolved ticket, per document processed. And here's the uncomfortable statistic: only about a fifth of organizations actually track their AI spend at that granularity. The rest are flying blind, which is why so many pilots die when someone finally does the division. Instrument cost per unit of work from the first day — it ties directly into the governance of Chapter 4 and the observability of Chapter 9.

And never forget the lever from the last section: shorter output is cheaper output. Asking for concise, structured responses — a JSON object instead of a page of prose — routinely cuts the expensive half of the bill without touching quality.

The reasoning tax

There's a newer cost dynamic that breaks a comfortable assumption, and you need to see it coming.

Reasoning models don't just answer. They first generate a long, hidden chain of "thinking" — working through the problem, testing approaches, backtracking — and only then produce the visible response. Those thinking tokens are real tokens, billed as *output* tokens, the expensive kind. And they are not cheap in number: for hard problems, a reasoning model can consume roughly an order of magnitude more tokens per useful answer than a plain model.

That imposes two taxes, not one. A **cost** tax, because you're paying output rates for a large invisible preamble. And a **latency** tax, because thinking tokens are autoregressive and happen *before* the first visible word — so a heavy reasoning chain can mean seconds of silence before the user sees anything at all. Time-to-first-token, which we're about to make central, balloons.

This is why the industry's favorite reassurance — "cost per token keeps falling, so this all gets cheaper" — is only half true. Per-token prices do fall. But if you shift your workloads toward heavy reasoning, tokens-per-answer step up by that order of magnitude, and per-*answer* cost can rise even as per-token cost drops. The two trends fight, and in a reasoning-heavy system the reasoning wins.

The discipline, then, is to treat reasoning as a governed resource rather than a free upgrade. It's usually exposed as a dial — off, low, medium, high — and, importantly, its returns diminish: past a certain depth, more thinking stops improving the answer, and what actually predicts quality is the tokens genuinely consumed, not the budget you allocated. So do with reasoning exactly what you did with model choice: route by difficulty. Default the bulk of traffic to no or low reasoning, and spend the deep-thinking budget only on the requests that provably earn it. A system that makes every request think is a system quietly overpaying in both money and time.

The clock: latency as a first-class constraint

Speed is not one number, and conflating its two components is a common source of "the model feels slow" confusion.

The first number is **time-to-first-token** — how long the user waits before *anything* appears. The second is **throughput**, or time-per-output-token — how fast the response completes once it has started. They come from the two phases we already met: prefill (reading your input) dominates time-to-first-token, and decode (generating output) dominates throughput. So a very long input makes the user wait longer to see the first word, while a very long output makes the whole response take longer to finish. Two different levers on two different delays.

This is why **streaming** exists, and why it's a product decision more than a technical one. By delivering tokens as they're generated rather than waiting for the whole response, streaming makes the *perceived* latency equal to time-to-first-token instead of total time. A ten-second full response becomes a one-second wait that then keeps flowing — which, for anything interactive, is the difference between "broken" and "fine." For a chat or an assistant, streaming is close to mandatory; for a background job, it's irrelevant.

Two working rules. First, the datasheet speed is marketing: real latency varies with load, region, model, and time of day, so measure on *your own* traffic at *your own* load before you promise anyone a number. Second, latency is an architecture decision that follows from the product. Interactive features need low time-to-first-token and streaming; background and bulk work can tolerate seconds or hours — and should therefore ride the batch discount from earlier. Match the serving mode to the latency budget the feature actually

has, rather than paying real-time prices for work no human is waiting on.

The ceiling that isn't where it says: context

Every model advertises a maximum context window — the number of input tokens it will accept — and those numbers have grown spectacularly, now reaching a million tokens and beyond. It is tempting to read that as "I can put everything in and let the model sort it out." Resist that, because the advertised window and the *usable* window are not the same number, and the gap is where a lot of production quality quietly dies.

The advertised window is the maximum the API accepts. The **effective** context — how much the model can actually reason over without losing precision — is often dramatically smaller, and quality starts degrading well before the stated limit. Two well-documented effects drive this. The first is **lost in the middle**: models attend well to the beginning and end of a long input and poorly to the material in between, a U-shaped curve that has been measured at accuracy drops of thirty percent or more for information buried in the center. The second is broader and more unsettling — **context rot**. Research that stress-tested eighteen different frontier models found that *every one of them* got worse as the input grew longer, even when the window was nowhere near full. More tokens in, worse output out. The mechanisms compound: lost-in-the-middle, plus attention spreading ever thinner as the token count rises, plus irrelevant-but-similar content actively distracting the model.

Notice the two-sided penalty here, because it's unusually clean. Stuffing the context window costs you *money* (all those input tokens) *and* costs you *quality* (rot). Context discipline is the rare lever that is

simultaneously the cheaper choice and the better one — there is no trade-off to agonize over.

And the failure mode you should fear most is the silent one. Overrunning the hard limit throws a loud, obvious error; you'll notice immediately. Context rot throws nothing. Your code runs, your logs are clean, and the answers just quietly get less reliable as the input grows — the most expensive kind of bug, the kind nobody sees until a user does. So the operating rules are: put your most critical instructions and data at the edges of the prompt, not the middle; don't dump everything in when you can retrieve only what's relevant — which is the entire argument for the retrieval systems in Chapter 7; and smoke-test your own data before trusting anything past the lower portion of an advertised window. Big windows are table stakes now. They are not a substitute for deciding, deliberately, what belongs in the prompt.

Serving shapes, and the self-host crossover

Pull the threads together and the ways you consume a model resolve into three shapes, each matched to a different latency budget. **Real-time** synchronous calls for interactive work, at full price. **Streaming** for interactive work where perceived speed matters, which is most of it. And **batch** for volume and background work that can wait, at roughly half the cost. Choosing the shape is choosing where a workload sits on the speed-versus-cost line, and most systems use all three for different jobs.

Two constraints shape the choice in practice. **Rate limits and quotas** are not a footnote — the cheapest model on paper is useless if you can't get enough throughput at your spend tier, and this can override a per-token comparison entirely. It's worth knowing that

caching and batch often carry separate, higher limits, so they buy you *capacity* as well as savings; design for limits with backoff and queueing rather than discovering them in an incident. And the **self-host crossover** from Chapter 4 is fundamentally an economic line: consuming a managed endpoint is the right call until your volume and utilization are high enough that owning the accelerator is cheaper — a threshold that lands, very roughly, in the range of tens of millions of tokens a month at sustained high utilization, and only when you have the operational capability to run it. Below that line, managed wins on every axis; above it, self-hosting can pay. The gateway from Chapter 3 is precisely what lets you cross that line later without rewriting the application.

Beyond text: multimodal inputs

This chapter has assumed text in and text out, as most of this book does for clarity. But real enterprise data is not tidy text — it's PDFs, scans, invoices, contracts, charts, screenshots, diagrams, forms, product photos, and call recordings. Increasingly the models take all of it directly, and the good news is that you already hold the mental model for it.

The mechanism, in one line: everything becomes tokens. A multimodal model runs an image through a vision encoder that turns it into *vision tokens*, maps those into the same space the text tokens live in, and processes the whole mix in a single attention pass; audio is handled the same way, either tokenized directly or transcribed to text first. So every mechanic in this chapter — the token as the unit, the context window, the cost math, the latency clock — extends directly to images and audio. You simply have new sources of tokens.

The cost twist is the part you must not miss: images are token-hungry. A single high-detail image can cost anywhere from a few hundred to many thousands of tokens depending on the model and its resolution, enough that bolting vision onto a text workload can multiply its bill several-fold overnight. So the levers from this chapter grow multimodal-specific dials. Image resolution and "detail level" become cost levers — send a low-detail version when a thumbnail would answer the question; audio gets chunked; and video, billed per second, is the most expensive input of all. Curiously, the arrow sometimes points the other way: for a *dense* document, handing the model the page as an image can cost fewer tokens than its extracted text, which is why compressing documents into images is a live optimization thread.

And the jagged frontier of Chapter 2 reappears, now sliced by modality. Multimodal models are genuinely impressive and predictably uneven — they reason beautifully across a chart or a screenshot or a page, then get beaten on narrow, high-accuracy extraction by purpose-built tools: a dedicated OCR service for pulling exact text off a messy scan, a dedicated speech model for transcribing accented or noisy multi-speaker audio. So the mature pattern is not "use the multimodal model for everything." It's the tool-use pattern of Chapter 6 applied to perception: let a specialized extractor do the precise, high-stakes reading, then hand its clean output to the model for reasoning. Route by what each part actually needs, exactly as you route text tasks by difficulty.

For the enterprise this lands first on *document understanding* — invoices, contracts, forms turned into extracted, structured fields — which plugs straight into the retrieval of Chapter 7 and the agents of Chapter 8, where a model that can see a screen, read a document, and act is the substrate the agentic future is being built on. Carry

one caution forward: an instruction can hide inside an image as easily as inside text, so multimodal widens the prompt-injection surface of Chapter 11. Treat pixels as untrusted input, too. Multimodal is not a separate discipline; it is this chapter's discipline with new token sources and a bigger bill.

Step back and the whole chapter collapses into one idea. Model choice, serving shape, caching, batching, reasoning effort, output length, and context discipline are not separate concerns — they are the dials on a single console. Engineering inference means setting those dials to hit the cost, latency, and quality budget each workload actually has, and then instrumenting the system well enough to see, in production, whether you got it right. That last part — seeing whether you got it right — is a whole discipline of its own, and it's where we're headed.

Experiments to run

Small enough to start today. Cost and latency calculators, and a live pricing page, at leapfrog.lerias.org.

1. **Read your own meter.** Run one real task and log the input tokens, the output tokens, and the cost separately. Then ask the model for a concise or structured answer and run it again. Watch the bill fall from the output side. You'll never again forget which direction is expensive.

2. **Pull all three levers.** Take a workload you run repeatedly. Cache the system prompt, route the easy

cases to a cheaper model, and move the non-urgent ones to batch. Measure cost per unit of work before and after — that delta is the difference between a pilot and a product.

3. **Pay the reasoning tax on purpose.** Run the same hard task with reasoning off, low, and high. Line up quality, tokens, latency, and cost side by side, and find the point where more thinking stops earning its keep. That point is a design parameter, not a mystery.

4. **Find your real ceiling.** Take a model with a big advertised window, bury one specific fact in the middle of a long input, and ask for it back. Whether it comes back tells you your *effective* context — the number the datasheet won't.

5. **Meter an image.** Send the same picture to a vision model at high detail and at low detail, and log the token cost of each. Then compare a high-volume image workload against its text-only equivalent. That gap is why resolution is a cost lever, not a formatting choice.

References for this chapter

Independent instruments and neutral primary sources first; live numbers kept current on leapfrog.lerias.org because they change in weeks.

Cost and latency mechanics - Independent pricing and latency trackers — e.g. Artificial Analysis and live per-token pricing comparisons — used to read the market, not to endorse a model. - Provider pricing, prompt-caching, and batch-API documentation across vendors, cited for *mechanics* (input/output asymmetry, cache and batch discounts, rate limits), not rankings. - Stanford HAI *AI Index* for the multi-year decline in inference cost.

The reasoning tax - Analyses of inference-time / test-time compute and its effect on per-answer cost and latency. - Research on reasoning-token consumption and the diminishing returns of reasoning effort.

Context and its limits - “Lost in the Middle” (Liu et al., Stanford / TACL) — the U-shaped attention curve. - Context-rot research (including the eighteen-model study) — degradation as input length grows, and the advertised-versus-effective gap. Motivates the retrieval systems of Chapter 7.

Beyond text: multimodal - Per-provider image-tokenization rules and vision-token cost analyses (as of 2026); vision-token compression research. - The practical boundaries of multimodal models versus purpose-built OCR and speech tools — the case for routing perception to specialized extractors (Chapter 6’s tool use).

PART II

Build the Thing

Patterns that hold up, and grounding a model in your data.

06

Patterns That Hold Up

CHAPTER 6

You will forget which model you used. You will not forget the shape of the system you built around it. Build the shapes that outlast the models.

The last chapter made a single model call legible — its cost, its speed, its limits. This chapter is about everything you build *around* that call, and it opens with a warning: the field will throw a new framework at you roughly every month, and chasing them is a treadmill that produces motion without progress.

So this chapter teaches patterns, not frameworks. A pattern is a shape — a way of arranging model calls, tools, and data that keeps making sense even when the model underneath doubles in capability and the framework you used to implement it is deprecated. Learn the shapes and every framework, or none, becomes legible. That is how you build things that survive the next release instead of rebuilding them for it.

Patterns over frameworks

Start from the primitive you already understand: the single call. One input, one output, done. Everything in this chapter is composition on top of that — chaining calls together, giving them tools, grounding them in your data, letting them loop. Nothing here is exotic. It's arrangement.

The reason to think in patterns rather than frameworks is durability. Frameworks are implementations; they come, they get popular, they get abandoned, they get replaced by the next one that does the same thing with different names. The popular agent frameworks of this year are useful, and you may well use one — but the value they encode is a handful of patterns you can name in an afternoon. Learn the patterns and a framework becomes a convenience rather than a dependency; you can pick one up, put it down, or swap it, because you understand the shape it's implementing. This is Chapter 3's anti-lock-in instinct pointed at your application layer instead of your model.

There's a lens that runs through the whole chapter: durable versus hype. A pattern holds up if it still makes sense after the model got twice as good and the framework got deprecated. Most of what gets marketed as "the future of AI" is a framework or a product. Most of what actually holds up is a small set of patterns — and the discipline to reach for the simplest one that works.

Context engineering

The first pattern is really a discipline, and it has quietly renamed itself, which tells you something.

For three years the hyped skill was *prompt engineering* — crafting the perfect instruction. That skill has been largely absorbed into a broader one the field now calls *context engineering*: the deliberate design of everything the model sees on every single call. The models got good enough at reading intent that clever phrasing matters less than it did; what matters now is what *information* is in the window when the model runs.

The framing that stuck is an analogy: the model is a CPU, and the context window is its RAM. Your job is to be the operating system — loading exactly the right working memory for each task and nothing else. And "everything the model sees" is more than the prompt. It's the system instructions and the output contract, the user's input, the documents you retrieved, the conversation history, the definitions of the tools it can call, and anything it has remembered from before. Context engineering is the deliberate assembly of all of that, per call.

Here is the diagnosis that makes this the central skill: in production, most failures are no longer model failures — they're context failures. The model was capable; it just wasn't given the right things to see. The asymmetry proves it. A brilliant prompt sitting in a poorly assembled context still fails, while a mediocre prompt in a well-assembled context often succeeds. If failures live in context assembly, that's where your engineering effort belongs.

There's a compact way to think about the moves involved: **write** the instructions and state clearly; **select** the right context to bring in for this specific call; **compress** it so you're not wasting tokens (which, from Chapter 5, is both a cost and a quality decision — remember context rot); and **isolate** unrelated context so it doesn't distract the model. A governing principle sits underneath all four: just-in-time retrieval beats pre-loading everything. Don't pour the whole world

into the window and hope the model finds the relevant part. Decide what enters the attention span, and when.

This is the direct continuation of Chapter 5. There we learned the context window has an effective ceiling and quietly rots as you fill it. Context engineering is the discipline of building deliberately *within* that ceiling — and, per Chapter 4, of treating the assembled context as a versioned, testable asset rather than a string someone typed once.

Structured outputs: the contract at the boundary

The second pattern is the one that makes all the others composable, and it's easy to underrate.

A raw model returns free-form text. That's fine when a human reads it, and a disaster when *code* has to. Parse prose with hand-rolled heuristics and you've built your system's most important seam out of guesswork — the exact place non-determinism does the most damage. The pattern that fixes this is the *structured output*: constrain the model to return data in a defined schema, typically JSON with specified fields and types, enforced so the shape is guaranteed. Providers now support this directly — schema-checked results your code can rely on.

What that buys you is bigger than convenience. It turns a probabilistic text generator into a component with a typed interface. The model can still be creative *inside* the fields, but the envelope around it is contractual. You've tamed non-determinism at precisely the point it matters most: the boundary where AI hands its output to the rest of your system. Everything downstream in this chapter — tools,

pipelines, multi-step agents — depends on each step producing something the next step can consume without hoping. Structured output is the connective tissue that makes composition possible instead of fragile.

The working habit is the one you already use for any interface: define the contract first. Decide the fields, the types, and what "done" looks like, then build the call to fill it. Contract-first, exactly as you'd design an API — because that's what you're doing.

Tool use: hands, via an open standard

The third pattern gives the model the ability to *act*, not just talk. With tool use, the model can call a function, query a database, hit an API, or run code. The loop is simple: the model decides which tool to call and with what arguments, your code executes it, the result comes back, and the model continues with that result in hand. This is what turns a text box into something that can actually do work in your systems.

The mechanism underneath is *function calling* — the model emits a structured request to invoke a named tool with typed arguments. Notice that a tool call *is* a structured output, which is why the last section came first: tool use is structured output pointed at your systems.

What changed recently, and durably, is standardization. The Model Context Protocol — introduced by Anthropic in late 2024, since donated to the Linux Foundation and adopted across the major providers — is an open, vendor-neutral way for tools to describe themselves and for models to discover and invoke them. It's often called "USB-C for AI." Before it, every tool-to-model connection was be-

spoke; now a tool exposed once through the protocol works across models and hosts, and an agent's tool layer becomes simply "the union of the tool servers it's connected to."

Three reasons this book cares about the *open* part specifically. First, lock-in: this is Chapter 3's argument arriving at the tool layer — your integrations aren't welded to one vendor's proprietary format. Second, governance: it's Chapter 4's shift-left made concrete, because a standard tool layer is where you put audited, permissioned, least-privilege access to internal systems, instead of copy-pasted context or one-off connectors nobody reviews. Third, composition: standard tools snap together.

But hold the most important caveat firmly, because it's where teams overspend and underdeliver: **the protocol is integration, not intelligence.** Connect a powerful tool to a vague prompt and you get an agent that *has* tools and doesn't know when to use them. The capability comes from the tool; the competence comes from context engineering around it — naming the goal, spelling out which tool applies when, defining the output contract, and planning what happens when a tool fails. Integration and context engineering are a pair. Connectivity is not competence.

One flag for later: tools are power, and power needs least privilege and active defense against tool-targeted attacks. That's Chapter 11's territory; just don't wire an agent to your production database on the strength of a demo.

Grounding: RAG as a pattern

The fourth pattern answers the two problems Chapter 5 left open. A model relying only on what it memorized during training is working

from knowledge that is generic, frozen at a cutoff, and prone to confident invention — and stuffing everything it might need into the window just triggers context rot. Grounding fixes both. Instead of trusting the model's memory, you *retrieve* the relevant information at query time, place it in the context, and then generate. Retrieve, then generate — commonly known as retrieval-augmented generation, or RAG.

This is the answer to hallucination, because the response is anchored to real sources you can cite. It's the answer to context rot, because you bring in only what's relevant rather than everything. And it's how you give a model access to your private, current, proprietary data without retraining anything. The shape is straightforward: a query arrives, you find the relevant pieces of your data, you assemble them into the context, and the model answers grounded in them — ideally citing what it used.

There's a natural marriage with the previous pattern: *retrieval as a tool call*. Rather than always pre-fetching, you can let the model decide when it needs to look something up and what to look for — retrieval becomes just another tool in the union. This "agentic RAG" is increasingly the default shape.

Grounding is where most genuinely useful enterprise AI lives, which is why it gets its own chapter. Here it's enough to hold it as a first-class pattern; the mechanics — how you turn documents into something retrievable, how you keep it fresh, how you govern it — are Chapter 7.

Workflows and agents: the progression, and the discipline

Now the pattern everyone is shouting about, handled with the restraint the evidence demands.

There are two ways to build a multi-step AI system, and the difference is simply who holds the steering wheel. In a **workflow**, *you* define the control flow in code, with model calls at specific steps. It's predictable, testable, cheaper, and traceable, because you decided the path. In an **agent**, the *model* decides what to do next at runtime. It's flexible and handles genuinely novel situations, but it's harder to control, more expensive, and less predictable, because you handed it the wheel. Both are "agentic systems." The engineering implications are worlds apart.

The rule that has held up through a year of hype is conservative, and you should adopt it as a default: **start with workflows, and graduate to agents only when the task genuinely requires dynamic decision-making** that you cannot express as a fixed flow. Most real systems settle somewhere in between — structured enough to be reliable, flexible enough to absorb variance.

The workflows themselves are built from a small kit of composable patterns worth knowing by name. *Prompt chaining* breaks a task into a sequence of steps, each feeding the next — the simplest pattern, and often the right first choice. *Routing* classifies the input and sends it down the appropriate path or to the appropriate model — the multi-model routing of Chapters 3 and 5, now as an application pattern. *Parallelization* runs independent subtasks at once and aggregates them. *Orchestrator-workers* has a lead call decompose a job into subtasks dispatched to worker calls. *Evaluator-optimizer* pairs a call that generates with a call that critiques and improves, a

feedback loop. On top of these sit capability patterns: *reflection*, where the model critiques its own work; *planning*, where it plans before executing; and *multi-agent collaboration*, where specialists cooperate — which you add only when specialization clearly helps, not because a framework makes spawning agents easy.

One insight is worth internalizing because it's counterintuitive and it saves money: wrapping a weaker, cheaper model in the right loop — reflection, verification, a second pass — can outperform a stronger model used in a single shot. Often the loop, not the model, is the lever.

To keep yourself honest, design in three layers. **Workflow patterns** decide how the system thinks and executes. **Capability patterns** extend what it can know and do — tools, memory, retrieval. **Production patterns** keep it safe and reliable — guardrails, human-in-the-loop at high stakes, verification, retries, circuit breakers, bounded execution, and tracing on from day one. An impressive prototype has the first two layers. A system that survives production has all three, and the gap between them is exactly where most pilots die.

Which brings us to the discipline at the heart of this whole chapter: **introduce complexity only in response to a real failure mode, and use the minimum control mechanism that addresses it.** Don't add an agent because a framework makes it a one-liner. Don't add a second agent unless specialization demonstrably helps. Every moving part you add is a part that can fail and must be observed and debugged. Complexity should be *earned* by a problem, not adopted for a demo.

Because here is the uncomfortable truth about the loudest story of 2026: agents mostly fail in production not because the model was too weak, but because the system had no structure — which is why a

large share of agentic projects are forecast to be cancelled before they deliver anything. The maxim that emerged from the teams who actually shipped is “agents aren’t hard; the harness is hard.” The harness — the constraints, the vetted tools, the verification, the observability, the recovery paths — is the real work. And that harness is nothing other than the engineering discipline this entire book is about. Building the agent is a weekend. Building the harness that lets it run on Monday is the job. Deploying that harness is Chapter 8; proving it works and debugging it when it doesn’t is Chapter 9.

The patterns, in the end, are few and stable. The skill is knowing which one a problem actually needs — and having the restraint not to reach for more than that. That restraint is not timidity. It’s the thing that makes AI which survives contact with reality.

Experiments to run

Small enough to start today. Pattern templates and starters at leapfrog.lerias.org.

1. **Assemble the context on purpose.** Take one task and deliberately design what goes into the window — the instructions, the few facts it actually needs, the output format — then strip half of it out and watch the quality move. That swing is context engineering, felt rather than read.
2. **Enforce a contract.** Make a model call return strict, schema-valid JSON that your code parses with no

defensive gymnastics. You've just converted a text generator into a typed component you can build on.

3. **Give it one tool — then sabotage it.** Connect a single tool through the open protocol and watch the model call it. Then vague out the prompt until the model *has* the tool but stops using it well. That's the difference between integration and intelligence, in your hands.

4. **Resist the agent.** Take the thing you're itching to build as an autonomous agent and build it as a fixed workflow instead. Ship that first. Reach for agency only when the workflow provably can't handle the variance — and notice how rarely that turns out to be true.

References for this chapter

Primary and durable sources first; frameworks named only as implementations of patterns, never endorsements. Templates kept current on leapfrog.lerias.org.

The workflow-and-agent canon - Anthropic, *Building Effective Agents* — the workflows-versus-agents distinction and the five workflow patterns. - Andrew Ng's agentic design patterns — reflection, tool use, planning, multi-agent collaboration. - Google Cloud and Microsoft agent design-pattern guides, read cross-vendor for the same shapes.

Context, contracts, and tools - Context-engineering writeups — the CPU/RAM/operating-system framing, and the write / select / compress / isolate taxonomy. - Structured-output documentation across providers, cited for the mechanism (schema-enforced results), not for ranking. - The Model Context Protocol specification (Anthropic / Linux Foundation) — the open tool-integration standard, and the anti-lock-in and governance arguments that follow from it.

Grounding and the production layer - Retrieval-augmented generation as a pattern; agentic RAG (retrieval as a tool call). Mechanics in Chapter 7. - OWASP LLM Top 10 and NIST least-privilege guidance for the production/security layer, expanded in Chapter 11. - Framework projects (the popular agent and orchestration libraries and vendor SDKs) referenced only as implementations of the patterns above.

07

Data, Retrieval, and Grounding

CHAPTER 7

A model with a million-token window and no discipline will still confidently tell you the wrong thing. Grounding is how you make it tell you the right one — and prove where the answer came from.

Chapter 6 established grounding as a pattern: retrieve the relevant information, then generate. This chapter is the mechanics — how you actually turn a pile of enterprise documents into something a model can answer from accurately, safely, and verifiably. It matters more than any other build chapter, because grounding is where most genuinely useful enterprise AI lives. The value is almost always in *your* data — your policies, your contracts, your tickets, your history — and no model was trained on that.

It opens by clearing away the single most common piece of bad advice you'll hear this year.

RAG didn't die; it specialized

Every few months someone declares that retrieval is dead. The argument goes: context windows are a million tokens now, so stop building retrieval pipelines and just put everything in the prompt. It sounds reasonable. It's wrong, and Chapter 5 already told you why — the effective window is far smaller than the advertised one, quality rots as you fill it, and stuffing context is expensive. But you don't have to take the earlier chapter's word for it. Independent benchmarking that pitted retrieval against long-context across thousands of cases concluded that neither is a silver bullet: retrieval wins when the corpus is large, when freshness matters, and when you need to cite sources — and long-context runs something like eight to eighty times more expensive at scale.

So the real question was never "RAG or long context." It's which tool fits which query. Whole-document reasoning — "summarize what we discussed" — suits a long window. A precise fact that must be exactly right and cited — "quote the termination clause" — suits retrieval. Style and formatting consistency isn't a grounding problem at all; that's fine-tuning. Questions that require connecting facts across sources call for the graph methods later in this chapter. The best teams don't argue the dichotomy; they route by query type, then measure.

What's actually happening is that retrieval is *specializing* and disappearing into the infrastructure — becoming the invisible "context layer" the way a database is the invisible layer beneath an ordinary app. Nobody asks "should this app use a database"; soon nobody will ask "should this system use retrieval." The question becomes *how is our context layer configured* — which is Chapter 6's context engineering, made operational. The market agrees: this is a category

growing at roughly fifty percent a year. Dead technology doesn't scale like that.

How text becomes retrievable: embeddings and chunking

To retrieve by meaning, you first turn text into numbers. An *embedding* is a vector — a point in a high-dimensional space — positioned so that texts with similar meaning land near each other. Search then becomes geometry: find the points nearest to the query. This is what lets a search for "how do I expense a flight" find a document titled "travel reimbursement policy," despite no shared words.

But hold one fact about embeddings close, because half of this chapter follows from it: they are *lossy by design*. Compressing a whole paragraph into a single point necessarily throws information away. That compression is what makes semantic search possible, and it's also why semantic search alone is not enough — a point in space can't preserve every exact term the paragraph contained.

Before geometry, there's a choice most teams underweight and then regret: **the embedding model matters more than the database.** Retrieval quality is set primarily by how good your embeddings are at capturing meaning in *your* domain. Choose and benchmark the embedding model on your own data first — long before you agonize over which vector store to buy. Get that wrong and no amount of downstream cleverness recovers it.

Then there's *chunking*, the most underrated lever in the whole stack. You can't embed a whole document as one point; you split it into pieces, and how you split determines what can ever be retrieved. The naive approach — cut every N tokens — is fast and dumb, slicing

through the middle of a thought so that neither half retrieves well. Better approaches split at meaning boundaries, or respect the document's actual structure: its headings, sections, tables, and lists. The chunk is the unit of retrieval, so a good chunk is a complete, self-contained idea, not a fragment. Garbage chunks produce garbage retrieval no matter how sophisticated everything after them is. Clean and de-duplicate before you embed; the boring work here pays back more than any flashy component later.

The retrieval stack: naive RAG is dead

The "hello world" of retrieval — embed the query, grab the nearest few chunks by cosine similarity, stuff them in, generate — is genuinely dead for production. It fails in two predictable ways that both trace back to lossy embeddings. It misses exact terms, because a part number or a specific name or a precise clause is exactly the detail compression discards. And it can't reason across several facts at once. The modern pipeline exists to fix these failures, one layer at a time.

Query transformation comes first, and it's optional but powerful: fix the user's question *before* it hits the index. Real users ask vague, misspelled, under-specified questions. Expanding, rephrasing, or generating a hypothetical ideal answer to search against turns a bad query into a retrievable one. Don't send raw human questions straight to your index.

Hybrid search is the core upgrade. Run semantic vector search (for meaning) *and* traditional keyword search (for exact terms) side by side, then merge the two ranked lists with a fusion step. This routinely beats either method alone by twenty to thirty-five percent, for the obvious reason: vectors catch "conceptually similar" and

keywords catch “literally this exact string,” and real questions need both. A vector search can even treat a singular and a plural of the same word as meaningfully different; keyword search simply doesn’t care. Semantic search finds what you *meant*; keyword search guarantees you didn’t lose what you *said*.

Reranking is the layer most often called the secret sauce, and it makes retrieval a two-stage affair. First you cast a wide, cheap net — pull a few dozen candidate chunks. Then a *reranker* — a model that looks at the query and each candidate *together*, rather than comparing pre-computed points — scores true relevance far more precisely and keeps only the best handful for the context. It’s expensive per item, which is exactly why you run it only on the shortlist, not the whole corpus. The wide-then-narrow shape is the whole trick: retrieve broadly and imprecisely, then judge narrowly and precisely.

Put together, the pipeline is: query → transform → retrieve widely with hybrid search → fuse → rerank down to a few → generate. It adds only a few hundred milliseconds. And every layer earns its place by fixing a specific failure of the one before it — which is precisely what naive RAG left out.

Where the vectors live: the store as a production decision

The vector store gets outsized attention because it’s the shiny new kind of database, but keep it in proportion: it is one layer in a stack that also includes query handling, hybrid routing, reranking, and permission filtering. And in 2026 there is no universal winner — the right choice is a production-engineering trade-off, not a leaderboard.

The axes that actually decide it are retrieval quality, metadata filtering, tail latency under real concurrency, operational complexity, the security and multi-tenancy model, freshness requirements, and — frequently the deciding factor — whether your vectors ought to live right next to your existing data.

Which points to a pragmatic default worth stating plainly: **if you already run Postgres, start with its vector extension.** It's mature, it's used in production at serious scale, and it keeps your vectors alongside your relational data in one system your team already operates — no second database, no extra bill, no extra thing to secure. You may well never outgrow it. Reach for a purpose-built vector database only when scale, first-class hybrid search, or hard multi-tenant isolation genuinely forces the move. This is Chapter 4's "add only what you need" and Chapter 3's "don't take on a system you'll regret," applied to retrieval — and it's not a vendor preference, it's an architecture heuristic: start with what you have, add complexity when a requirement demands it.

The mistake to avoid is choosing on a benchmark screenshot or a popular tutorial rather than on your real filters, query volume, security model, and operational reality — and then treating the decision as permanent. It isn't; migrating a retrieval layer later is far more tractable than the marketing implies. And note the quiet bridge to the next section: where the store runs is also where your documents and their embeddings physically live, which for many enterprises matters more than any feature on the comparison chart.

The enterprise realities that break demos

A grounding demo works in an afternoon. A grounding system a regulated enterprise will actually run is a different animal, and four realities are what separate them.

Permissions are the one that gets companies in trouble. Access control has to be enforced at the moment of *retrieval*, not merely at the interface. If you filter what's *displayed* but not what's *fetched*, the model can read — and then paraphrase, summarize, or quote in its answer — documents the user was never cleared to see. The interface hid the file; the model leaked its contents. The discipline is permission-aware retrieval: every query is scoped, at retrieval time, to exactly what *this* user is allowed to see, so the model is structurally incapable of grounding on forbidden data. Get this wrong and no amount of polish saves you.

Freshness is a feature, not an afterthought. Your data changes, and a stale index confidently serves last quarter's answer with a straight face. Re-indexing at scale is costly and error-prone — it has cost real companies real customers — so you need incremental updates and an honest sense of how current your index actually is. An answer that was true when you indexed and false today is still a wrong answer.

Residency and governance extend to the embeddings. Here is the subtlety teams miss: embeddings are *derived data*. They are not "just numbers" — they can leak information about the source text, and they fall under the same residency, retention, and governance rules as the documents they came from. Where your vectors live is a data-residency decision. Govern the embeddings exactly as you govern the source. Chapters 4 and 11 carry this further.

And there's a failure the model will actively hide from you. Retrieve the wrong context, or nothing relevant, and the model does not stop — it answers anyway, confidently, grounding its hallucination in whatever you happened to hand it. Grounding without guardrails doesn't remove hallucination; it makes hallucination sound *more* authoritative, now with citations attached. The defenses are concrete: make the model cite its sources, verify that those citations actually support the claim, and design the system so that empty or weak retrieval produces an honest "I don't know" rather than a confident fabrication. Which is impossible to guarantee unless you can measure it — the subject we close on.

Agentic and graph retrieval, and how you know it works

Two advanced shapes go beyond the static pipeline — and, following Chapter 6's discipline, you reach for them only when a real problem demands it.

Agentic RAG stops treating retrieval as a single up-front step. Instead, the model decides *when* to retrieve, *what* to search for, and whether to search again after seeing initial results — retrieval as a tool call, from Chapter 6. It can reformulate its own query, search several times, and reason across what it finds. That's powerful for open-ended, multi-step questions, and it's also more expensive and less predictable — so use it when a fixed pipeline provably can't handle the variance, not by default.

GraphRAG answers the questions flat chunks can't. Ask "which of our vendors share a dependency that's currently flagged as at-risk," and no amount of similarity search over isolated paragraphs will connect the dots, because the answer lives in *relationships* across

documents. A knowledge graph — entities and the links between them — gives the retriever structure to traverse those relationships and reason across multiple hops. It's more work to build and keep current, and it's the right tool for connected, multi-hop domains.

And now the part most teams skip and then bitterly regret: **evaluation.** Grounding is genuinely hard to evaluate because it isn't one signal, it's four, and each needs its own method. *Retrieval recall*: did you fetch the right chunks at all? *Context relevance*: was what you fetched actually on point, or padded with noise? *Answer faithfulness*: did the answer stick to the retrieved context, or wander off and invent? *Answer correctness*: was the final answer, all things considered, actually right? A system can ace one and fail another — flawless retrieval feeding an unfaithful answer, or a faithful answer built dutifully on the wrong chunks. Measure all four or you are shipping something you cannot diagnose when it degrades, and it *will* degrade, silently, exactly like the context rot of Chapter 5. This is where Chapter 7 hands off to Chapter 9: grounding you can't evaluate is grounding you can't trust, so build the evaluation alongside the pipeline, not after it breaks in front of a user.

Step back and the shape of the work is clear, and a little humbling. Great grounding is unglamorous: good chunks, hybrid retrieval, a reranker, permission-scoping, freshness, honest citations, and relentless measurement. There is no single clever trick. But that unglamorous discipline is precisely the moat — because it is exactly the part the ninety-five percent skip on their way to a demo that never survived a real question.

The other lever: fine-tuning, and when to reach for it

Everything so far in this chapter adapts a model to your world by feeding it the right context at the moment you ask. There's a second lever that adapts the model *itself* — fine-tuning — and because most teams reach for it too early and for the wrong reason, it's worth being blunt about when it earns its place.

The single heuristic that settles most cases is this: **fine-tuning is for form, not facts.** It shapes *behavior* — a house tone, a strict output schema, a domain vocabulary, a tool-call format, a refusal pattern — the things you want the model to do *consistently* without being told every time. It is genuinely bad at injecting *knowledge*, and worse than useless for knowledge that changes, because a fact baked into the weights goes stale, can't be cited, and can't be permission-scoped. Knowledge is retrieval's job — this entire chapter. So if your complaint is "it doesn't know our facts," you don't have a fine-tuning problem; you have the retrieval problem you just spent a chapter learning to solve.

The right order of operations, and the field is unusually agreed on this, is *prompt, then RAG, then fine-tune, then distill* — and you earn each step by exhausting the one before it. Prompting and solid retrieval are hours of reversible work; if they get you ninety percent of the way, the final ten rarely repays a training pipeline's ongoing cost. So the honest answer to "should we fine-tune?" is, most of the time, "not yet" — have you actually run out of road on prompting and retrieval, and do you have an evaluation (Chapter 9) that *shows* the base model has plateaued? Absent that, fine-tuning is a solution shopping for a problem.

When it does earn its place, the signal is a stable task with a known-good output the base model can't hold reliably — a rigid schema, a brand voice, a narrow domain register — backed by that eval and a few hundred to a few thousand genuinely clean examples (curated beats voluminous, every time). One more case deserves naming: *compression*. A smaller open-weight model, fine-tuned on your narrow task, can match a frontier model on that task at a fraction of the inference cost and latency — the cost lever of Chapters 5 and 10. And the two levers compose rather than compete: a common sweet spot is a fine-tuned small model for *form* with retrieval on top for *facts*.

On method, at the level this book cares about: you rarely retrain every weight — full fine-tuning is expensive and usually overkill. The accessible path is parameter-efficient, freezing the base model and training a small *adapter* on top (the family known as LoRA, and its quantized cousin that fits on a single GPU), cheap enough to iterate on and small enough to serve beside the base and swap in and out. The labs on leapfrog.lerias.org walk the mechanics; the book's job is the decision.

And the part nobody quotes you on: the operational tax. Fine-tuning is not a one-time task, it's a maintained pipeline. Adapters need versioning and rollback and a retraining cadence; your training data and configs are code (Chapter 8); and — the real sting — when your provider updates the base model underneath you, your adapter can silently degrade, which is the prompt drift of Chapter 8 wearing a new coat. So you plan periodic revalidation against your eval set, and you budget several times the training cost for the year of ownership that follows. The real cost was never the GPU hours. It's the evaluation, the data curation, and the lifecycle. Reach for fine-tuning deliberately and eyes-open — form not facts, earned not reflexive —

and most of the time you'll find the prompting and retrieval you already built were enough.

Experiments to run

Small enough to start today. Retrieval pipeline templates and an evaluation harness at leapfrog.lerias.org.

1. **Watch naive RAG fail.** Build the simplest embed → top-k → stuff → generate pipeline on your own documents, then hunt for a query it gets wrong — one with an exact term, a part number, or a name, usually does it. That failure is the whole reason the rest of the stack exists.

2. **Add hybrid search and a reranker.** Put keyword search alongside your vector search, fuse the results, and rerank the top candidates. Measure retrieval quality before and after on a set of real questions. Feel where the twenty-to-thirty-five percent comes from.

3. **Break the permission boundary — on purpose.** Put two different users' documents in one index and check, honestly, whether user A can retrieve user B's data. Then fix it at retrieval, not at the interface. Far better you find this than an auditor does.

4. **Ground it, then check faithfulness.** Make the model cite its sources, and then verify that the citations actually support the answer. The gap between

"cited" and "supported" is the gap between genuinely grounded and merely confident.

5. **Run the fine-tuning decision honestly.** Take a case where someone wants to fine-tune and sort the goal into *form* or *facts*. If it's facts, prove prompting and retrieval can't do it first. If it's form, check whether a stricter prompt and structured outputs (Chapter 6) already get you there. You'll usually talk yourself out of a training pipeline — which is the point.

References for this chapter

Primary and durable sources first; vector stores named only as category examples. Live comparisons and templates kept current on leapfrog.lerias.org.

The pattern and the debate - Lewis et al. — the original retrieval-augmented generation paper. - The LaRA benchmark and related retrieval-versus-long-context studies — when each approach wins, and the cost gap.

The retrieval stack - Hybrid search, reciprocal rank fusion, and cross-encoder reranking (including ColBERT-style late-interaction models). - Embedding-model documentation across providers, cited neutrally; the 2026 analyses on embedding choice dominating retrieval quality. - Vector stores as category examples — pgvector, Pinecone, Weaviate, Qdrant, Milvus, Chroma — evaluated against your own workload, filters, and security model, never a leaderboard.

Enterprise grounding and evaluation - Permission-aware retrieval and enterprise-RAG governance references; the industry security-readiness signals on agentic deployment. - GraphRAG and agentic-retrieval work, for multi-hop and dynamic retrieval. - RAG evaluation frameworks and the four signals (retrieval recall, context relevance, answer faithfulness, answer correctness), foreshadowing Chapter 9.

The other lever: fine-tuning - The 2026 “prompt → RAG → fine-tune → distill” sequence and the form-not-facts heuristic; parameter-efficient methods (LoRA and its quantized variant) as the accessible default — cited as methods, not endorsements. - The often-omitted lifecycle cost of fine-tuning (adapter versioning, base-model drift, revalidation cadence); hands-on mechanics deferred to leapfrog.lerias.org.

INTERLUDE

Ship Something Tiny This Week

Placed here on purpose. You've read enough. Before we go deep on operating, scaling, and securing these systems, put the book down and ship something — small, imperfect, real. The point isn't what you build. It's crossing the line from reader to builder, and feeling the whole loop once before we make it robust.

You've now got the why (Chapters 1 and 2) and most of the how (Chapters 3 through 7). The temptation is to keep reading until you understand everything, and then build. That instinct is the exact deferral this whole book argues against. So this is a deliberate pause: a single end-to-end build you can finish this week, with the training wheels firmly on. Do it before you read the operating chapters, and those chapters stop being theory — they become upgrades to a thing you already have in your hands.

The rules of the tiny build

Keep it honest and keep it small. Pick **one real workflow** — not a demo, not a chatbot for its own sake, but a genuine slice of work someone actually does. Pick one you're **allowed to touch**, and one where you can **tell whether it worked**. Then time-box it: an afternoon, maybe two. Training wheels on means a hosted model through an API, the smallest slice of real data that's meaningful, and no infrastructure heroics. You are not building a product this week. You are closing a loop.

The thin slice

Walk the shape below; the step-by-step with working code lives in the leapfrog.lerias.org labs. This is the map; the site is the lab.

1. Pick the workflow and define "better." One task, one improvement you could actually measure (Chapter 2). Before you write a line, write down what *delivered* and *better* mean here — the honest baseline and the target (Chapter 1's first experiment). If you can't say what better looks like, you can't tell if you succeeded.

2. Make one real call. Send one real input from that workflow to a hosted model's API and read your meter — the input tokens, the output tokens, the cost (Chapter 5). It's a small moment, but you've now touched the material directly instead of reading about it.

3. Ground it in a little real data. Take a handful of your real documents, do the simplest retrieval you can (even the naive version), and feed the model the relevant context (Chapters 6 and 7). Watch it get more useful — and pay attention to where it still fails. That failure is the map of what the deeper chapters are for.

4. Make it do one real thing. Have it return a structured output your code can act on, or make one tool call that performs one genuine action (Chapter 6). The moment it produces something the surrounding system uses, it stops being a chat and becomes *embedded in the work* — rung two of Chapter 2's ladder, reached for real.

5. Put a tiny check and a tiny bound on it. One small eval on a handful of cases: does it do the thing acceptably (Chapter 9, in miniature)? And one control: scope its access to the minimum, cap what it can spend or loop, and route anything irreversible past a human (Chapter 8, in miniature). Even at toy scale, feel the shape of evaluation and containment. They're easier to add now than to retrofit later.

6. Show it to one real user. The person who actually does this workflow. Watch them use it. Their reaction is your truest evaluation, and it will tell you more in five minutes than another chapter would.

What you just did

You just felt the entire loop this book is about — access, cost, grounding, action, evaluation, control — end to end, at a scale small enough to hold in your head. Chapters 8 through 10 are how you make that loop robust and affordable at scale; Chapter 11 is how you make it safe. All of it will land differently now, because you have a concrete thing to hang it on instead of an abstraction.

And here's the part that matters most. The gap between the five percent who deliver AI and the ninety-five percent who don't was never knowledge. It's having shipped. You just shipped — something small, imperfect, and real. Do that before you finish the book, and you've

already done the hardest thing the corporate trap kept telling you to defer. The rest is upgrades.

Experiment or die. This week. Something tiny. Go.

PART III

III

Survive Monday

Deploy it, keep it in control, and know it works.

08

Deploying and Operating AI Workloads

CHAPTER 8

Getting an AI feature to work in a notebook and getting it to survive Monday are different problems. The second one is mostly about staying in control of something that, by design, you cannot fully predict.

You've built something — a grounded assistant, a workflow, maybe an agent. This chapter is about the moment it stops being yours to watch and starts running on its own: deployed, unattended, changing over time, and — if it can act — capable of doing real things in the world. That transition is where most AI dies, and it dies not from weak models but from the absence of the operating discipline this chapter is about.

The discipline has a name, LLMOps, and it ends somewhere the earlier chapters have only hinted at: control. Because the thing you're deploying is non-deterministic by design, and once it can act, the difference between a useful system and a dangerous one is en-

tirely in how well you've bounded it. So we build up to that honestly — and then defuse the fear, because staying in control turns out to be ordinary engineering.

The notebook-to-Monday gap

Getting an agent to run in a notebook while you watch is a fundamentally different problem from getting it to run reliably, unattended, at scale. The gap between those two is this chapter, and it starts with a shift in where your attention goes.

Classic machine-learning operations is built around one object: the model. You train it, version it, deploy it, watch its predictions for drift, and retrain when it degrades. Operating an LLM application inverts this. You almost never train the model — you consume a pre-trained one through an API — so the model is often the *least*-frequently-changed component in the whole system. What changes constantly, and what you actually have to operate, is everything around it: the prompts, the retrieval configuration, the choice of model and provider, the tool definitions, the evaluation thresholds. The center of gravity moves from "managing a model" to "operating a product whose behavior changes."

Structurally, none of this is new to you. It's the software lifecycle from Chapter 4 — source control, CI/CD, infrastructure-as-code, environments promoted from development through staging to production — specialized for AI. What's new is what those pipelines now carry, and what has to be true before a change is allowed through the gates between environments. That's the rest of the chapter. And the reframe to hold onto from the start: deploying is not the finish

line of building. It's the beginning of operating — and operating a system that can surprise you is, at its core, a control problem.

Prompts are deployments; evals are the gate

Start with the smallest unit of change, because it carries the biggest surprise. A prompt is code. A one-word change to a system prompt can move output quality more than a full model retrain would in classic machine learning. So every prompt change — and every retrieval-config or model-selection change — is a *deployment*: versioned, reviewed, tested, and reversible, treated with exactly the seriousness you'd give a code change, because it is one.

But how do you *test* a change whose output is non-deterministic? You can't write the unit test "assert the answer equals four" when the answer legitimately varies run to run. This is the shift that catches teams: **evaluations replace unit tests.** You run the proposed change against a golden dataset — a curated set of representative inputs with known-good criteria — and gate the deployment on the aggregate result. The change merges only if it clears the eval bar. Your pipeline stops asking "does this function return the right value" and starts asking "does this change make the system's behavior better or worse across cases that matter."

That gate has a real cost: it means your continuous-integration pipeline has to actually *call the model* on every prompt change — slower and more expensive than a unit test that runs in milliseconds. The pattern that keeps this affordable is layered gates. Cheap, fast, free checks first: does the output parse against its schema (the structured outputs of Chapter 6)? Then the expensive behavioral evals, the ones that spend real tokens. Then, where the stakes justi-

fy it, a human review queue. Fail fast and cheap, and spend the expensive gate only on what already passed the free one.

Underneath sits a registry: prompts, model selections, and configurations versioned in the repository, alongside code, deployed together. At any moment you can see which prompt version is live and roll back to a previous one. (Building a *good* golden dataset, and the deeper evaluation methods, are Chapter 9's job. Here, evaluation is simply the gate that stands between a change and production.)

The artifact is the whole response path

Here's a trap that sends teams debugging in the wrong direction. Ask "what version is running?" and the instinct is to answer with the model. But the model is a sliver of what actually produced any given output.

To reproduce, debug, or roll back a response, you need the entire *response path*: the prompt version, the model and its generation parameters, the retrieval configuration and what it actually fetched, the tool set and what the tools returned, and which rollout segment served the request. That whole path — not the model name — is the artifact you version and observe.

Why it matters is the everyday reality of operating these systems: when quality degrades, the model is usually not the cause. A prompt template changed. The document index updated overnight. The router sent this request to a different model than yesterday. A tool returned an error the agent quietly worked around. The fallback route fired and served a cheaper, less-validated result. A team that captures only "the model was X" sees a fraction of the system and debugs blind, chasing the one component that probably didn't move.

Operate the path, not the model — and you turn "something changed and quality dropped, no idea what" into "we can point at the exact change." That capability is what Chapter 9's tracing makes real.

Shipping non-determinism safely

Two challenges are unique to shipping a system whose outputs you can't predict exactly, and both have well-worn answers borrowed from ordinary distributed systems.

The first is **rollout**. Because you can't prove an output correct the way you prove a function returns four, you don't flip a switch — you release gradually and compare. *Canary*: send the change to a small slice of traffic and watch before widening. *Shadow*: run the new version alongside the old on real traffic without showing its output to users, and compare quality offline — the safest way to catch a regression before anyone experiences it. *Champion/challenger*: keep the current version as the champion and test challengers against it on live cases. *Fallback routing*: when the primary path fails or misbehaves, drop to a known-good one — the gateway from Chapter 3 earning its keep again. And under all of it, a *rollback path*: any change reverts instantly, precisely because you versioned the whole response path.

The second is stranger, and it's the one that surprises even careful teams: **prompt drift, the degradation you didn't cause.** Your provider quietly updates the base model, and your unchanged prompt starts producing worse output overnight. Your code didn't move. Your dashboards are green. And your users are getting confidently wrong answers. This is the non-deterministic world's version of "but it worked yesterday," and it can arrive while you sleep. The defenses are architectural: pin or version the model where the provider allows

it; shadow-test model upgrades before they go live; and keep the gateway in place so you can pin, swap, or roll back the model like any other dependency. (*Detecting* drift in production belongs to Chapter 9; the point here is to build so that when it happens, you *can* respond.)

The theme is now unmistakable. You are operating something that changes under you and can surprise you. That is not a defect to engineer away — it's the nature of the material. Which brings us to the two hardest expressions of it: agents, and control.

Deploying agents: long-running distributed systems

Chapters 6 and 7 handed the operational reality of agents forward to here, and it comes down to a reframe that should feel reassuring rather than daunting. An agent in production is a long-running, multi-step process that spans several models, tools, and services and holds state over minutes, hours, or days. Strip away the AI vocabulary and that is a *distributed system* — which, as a cloud-native engineer, is exactly the thing you already know how to run. The AI doesn't invent new problems here so much as supercharge familiar ones.

The questions that matter appear only *after* you deploy, which is why notebook agents don't prepare you for them. What happens when the agent crashes on step seven of ten? When a tool times out mid-run? When the process restarts? Ignore these and a single late-stage failure re-runs the entire workflow from the beginning — re-paying, in tokens and latency, for every model call it already made. The answer is *durable execution*: state persistence, automatic retry, and crash recovery, so a failure resumes from where it stopped instead

of starting over. An honest note worth having: the popular agent frameworks often don't provide this natively — you bolt on durable-execution infrastructure. That's not a failing; it's recognizable reliability engineering, the same you'd apply to any long-running workflow.

The rest is orchestration: routing between steps, managing state, handling errors, and inserting human checkpoints where they matter — observed through open instrumentation (the OpenTelemetry of Chapter 4) so you get one coherent view of a multi-step, multi-system flow, including where the tokens are going at each step. This is the "harness" from Chapter 6 made concrete, and the maxim holds: agents aren't hard; the harness is hard. Route what you can to typed, testable code instead of letting the model decide everything; abstract model calls behind versioned interfaces so you can pin and swap; treat cost per workflow step as a first-class metric. The stakes are not hypothetical — only a small fraction of enterprise agent pilots have reached production at scale — and the gap between the pilots and the survivors is almost entirely this operational discipline, not model capability.

Staying in control

Now the part that has to be said plainly, because it's the genuinely uncomfortable one — and because you deserve the honest version followed by the way through, not reassurance that skips the hard bit.

Everything in this chapter has circled a single fact. You are deploying a system that is non-deterministic — the same input can yield different actions. It can change under you, through drift and silent model updates. And once it's an agent with tools, it can *act* in the real world: spend money, send messages, alter records, reach external systems. Non-deterministic, plus changeable, plus able to act, is a

different risk profile from any software you have shipped before, and it would be dishonest to pretend otherwise. NIST's 2026 work on agents names precisely these traits — autonomous real-world actions, runtime tool-switching that defeats static rules, memory that can be poisoned over time, and non-deterministic behavior — as the things existing frameworks were never built to handle. This is the scary part. Sit with it for a moment, because the way out depends on taking it seriously.

Here is the reframe that dissolves the fear, and it comes from NIST rather than from optimism. Perfect control of the model is *provably impossible*. A 2026 peer-reviewed result, extending Gödel's incompleteness theorems, established that no finite set of guardrails can be universally robust against adversarial input — and because a model reads instructions and data through the very same channel, there is no clean way to separate trusted commands from untrusted content. If your plan was to make the model itself safe and predictable, that plan cannot work, for you or for anyone. So you stop trying to, and the anxiety attached to that impossible goal goes with it. **Control does not come from perfecting the model. It comes from bounding what the system can do.** You assume the model can misbehave or be manipulated, and you engineer so that when it does, the damage is contained. The goal, as the security community frames it, is not to stop the agent from acting — it's to ensure it only acts *as intended*, and to shrink the blast radius when it doesn't, so that a single failure stays a single failure instead of becoming a system-wide disaster.

That reframe turns an unbounded fear into a bounded engineering task, and the task is one you already know how to do. You wrap the

one probabilistic component — the model — in deterministic controls, every one of them ordinary:

- **Least privilege.** The agent gets a scoped identity and default-deny access to tools and data, with short-lived, narrowly-scoped credentials. It can touch only what it must. This is the single highest-leverage control, because it directly caps the blast radius no matter what the model decides.
- **Human-in-the-loop on the irreversible.** Any high-stakes or irreversible action — moving money, deleting data, sending an external message — passes through a human checkpoint. Regulation increasingly requires exactly this for high-risk systems (Chapter 11), but you'd want it regardless.
- **Sandboxed execution.** Tools and code run confined, where a bad call can't reach past its box.
- **Bounded execution.** Hard limits on steps, loops, time, and spend, so a confused agent can't run away — no infinite loops, no runaway bill.
- **Kill switches.** A stop you can hit at runtime, and that the system can trip itself on anomaly.
- **Observability.** You cannot control what you cannot see; the trace and the audit trail (Chapter 9) are part of the control surface, not a separate concern.

The organizing principle that keeps all this proportionate rather than paranoid is simple: **match the control to the blast radius.** An agent that drafts text a human will review needs a light touch. An agent that can execute trades or modify production data needs the full stack — tight permissions, human approval, sandboxing, hard limits, a kill switch. Size the harness to what a failure could actually cost.

And there is a calm fallback, drawn straight from the agent-security maturity guidance, for when you can't get comfortable: if you cannot control an agent enough for the power you were going to give it, you have two honest choices — invest in stronger controls, or reduce the agent's power. Either one keeps you in control. You are never cornered into choosing between "unleash it" and "don't build it."

None of this is AI magic. Containment, least privilege, blast-radius thinking, circuit breakers, fail-safes — this is distributed-systems and security engineering, the discipline you have practiced for years, pointed at a new kind of component. The model is the single part you can't make deterministic; everything you build around it is as controllable as it has always been. That is the whole frame, and it's why there's no need to panic: the goal was never a deterministic model, which is impossible. The goal is a controlled system, which is entirely achievable — and mostly with tools already in your hands.

So you operate an AI system the way you'd operate anything powerful and changeable: with gradual rollout, instant reversibility, a record of who approved which change on the basis of which checks (Chapter 4's shift-left governance, now living at runtime), and controls sized to the stakes. Deploying was never the finish line. It's the beginning of staying in control of something that earns its keep precisely because it can do things you didn't script — safely, because you decided in advance what "things" it is allowed to do.

Experiments to run

Small enough to start today. Deployment and control-harness templates at leapfrog.lerias.org.

1. **Put your build behind a deployed endpoint.** Take the thing you built in Chapters 6–7, containerize it, deploy it somewhere it stays up unattended, and call it from outside. Cross the local-to-hosted line from Chapter 4 for real, and notice everything that "worked in the notebook" now needs a second look.

2. **Add an eval gate.** Build a tiny golden dataset and wire it into your pipeline so a prompt change can't merge unless it passes. Then break a prompt on purpose and watch the gate stop you. That gate is what stands between you and a silent regression.

3. **Shadow a model swap.** Route a slice of traffic to a different model or version, compare its outputs against the current one without showing them to users, and decide with data instead of a hunch. You've just rehearsed your defense against prompt drift.

4. **Give an agent a kill switch and a leash.** Take a simple agent, scope its tools to least privilege, cap its steps and its spend, and add a stop you can hit. Then try to make it misbehave — and confirm the blast radius is exactly as small as you designed it. That confirmation is what "in control" feels like.

References for this chapter

Primary and durable sources first; tools named only as category examples. Deployment and harness templates on leapfrog.lerias.org.

LLMOps and safe deployment - LLMOps practice guides and CI eval-gate patterns; prompt/model/config registries versioned with code. - Rollout strategies — canary, shadow, champion/challenger, and fallback routing — adapted from ordinary distributed-systems practice.

Running agents - Durable-execution platforms (Temporal, DBOS) and agent orchestration/observability (OpenTelemetry) as category examples; MLflow's production-agent guidance on the harness, versioned interfaces, and per-step cost.

Staying in control - OWASP *Top 10 for Agentic Applications* (2026) and its agentic-AI security maturity levels — the containment posture and the "add controls or reduce power" fallback. - NIST — the *AI Risk Management Framework* and 2026 agent work, including the four hard traits of autonomous systems and the formal result that no finite guardrail set is universally robust (hence containment over perfection). - The blast-radius and least-privilege framing (govern through identity; a single failure stays a single failure). Adversarial depth and regulatory detail are Chapter 11.

09

Observability, Evaluation, and Debugging

CHAPTER 9

At 8 a.m. the agent quoted the wrong statute, and a customer noticed. You open the dashboard: traffic healthy, latency fine, no errors. The agent didn't crash. It just lied — and nothing you were watching was built to see it.

Chapter 8 got your system deployed, running, and bounded. This chapter answers the question that decides whether it survives contact with reality: *how do you know it's working?* And when it isn't, *how do you find out why?* This is the discipline the demo-builders skip and the survivors obsess over. It is, more than any other, the chapter that separates the five percent who realize value from the ninety-five percent who ship something they cannot diagnose.

The gap traditional monitoring can't see

You already know how to watch software in production. Latency, uptime, error rates, throughput, resource usage — the dashboards you've trusted for years. Point them at an AI system and they will tell you, cheerfully and correctly, that everything is fine while your model hands a customer a confident fabrication. A model can return a 200 response in fifty milliseconds and still hallucinate, leak private data, or produce biased output. By your monitoring's definition, nothing was wrong. By any definition that matters, everything was.

That gap is the reason this chapter exists, and closing it means keeping three things distinct that get carelessly lumped together. *Monitoring* watches — is the system up, is it fast? *Observability* traces — what actually happened inside this one request? *Evaluation* judges — was the output any good? You need all three, but the new and genuinely hard one, the one none of your existing tools do, is evaluation. Judging quality is the layer AI forces you to add.

And notice how this completes the arc from Chapter 8. There, control came from bounding what the system can *do*, because you can't make the model deterministic. Here is the other half: you also can't make it *predictable*, but you can make it fully *visible and measurable*. You may not know in advance what any given output will be, but you can see every one, score every one, and trace every one to its cause. That is a different kind of control than you're used to — not prediction, but total observability after the fact — and for a non-deterministic system it is exactly the kind you can actually have.

The trace tree is the unit

Before you can judge quality, you have to see what produced it — and a single AI request is almost never a single call. It's a tree: the model generation, the retrieval that fed it, the tools it invoked, the steps an agent took along the way. That tree is the response path from Chapter 8, and observing the whole thing, span by span, is the foundation everything else in this chapter stands on.

The architecture the field converged on in 2026 is worth stating precisely, because getting it right protects you and getting it wrong locks you in. LLM observability is OpenTelemetry tracing, plus LLM-aware span conventions (the OpenTelemetry-GenAI and OpenInference standards) that name the things that matter — the model, the prompt, the completion, the tokens, the tool call — plus evaluation scores written *back onto those spans*. The trace tree is the unit of work, and each span carries its own inputs, outputs, cost, and, once scored, its own quality verdict.

The reason to insist on the open standard is the reason this book insists on it everywhere: instrument once against OpenTelemetry and your traces flow to any backend you choose, and can be swapped later with a config change; adopt a vendor's proprietary span format and every future choice locks tighter around it. Neutrality belongs at the observability layer as much as at the model layer. What this buys you, concretely, is per-step cost and token attribution (the FinOps thread from Chapters 5 and 8, picked up again in Chapter 10) and — more importantly here — the raw material for both evaluation and debugging. You cannot judge or fix what you cannot see. The trace is how you see.

Evaluation: judging quality you can't unit-test

Here is the problem in one sentence: you cannot write `assert output == expected` for open-ended, non-deterministic text. So how do you judge quality, at scale, without a human reading everything? Three families of method, used together, from cheapest to most nuanced.

Programmatic checks come first because they're free and objective: does the output parse, does it match the required schema (Chapter 6's structured outputs), does it contain the citation it was supposed to? *Similarity and entailment* methods go further: embedding similarity to a reference answer, or semantic entailment against the retrieved context to check groundedness. And *LLM-as-a-judge* handles the subjective core — a model scoring another model's output against a rubric for factuality, relevance, faithfulness, or tone. This last one is now the workhorse of AI evaluation, and it comes with a caution you must respect: a model judging a model has blind spots of its own, and two judges will often fixate on different evidence and reach different verdicts. So you align your judge against human ratings on a sample before you trust it, you keep a human in the loop where the stakes are high, and you treat an unvalidated judge score as the unreliable number it is. A judge you haven't checked against people is not a measurement; it's a guess with a decimal point.

The metrics worth scoring are the ones the research has grounded: faithfulness and groundedness (does the answer stay true to its source?), hallucination, relevance, toxicity, bias, citation support — and, for retrieval systems, the four signals from Chapter 7: retrieval recall, context relevance, answer faithfulness, and answer

correctness. Against these you run a *golden dataset* — a curated set of representative inputs with known-good criteria, your standing yardstick for quality.

But the single practice that most separates the five percent from the ninety-five is not the golden set itself. It's the *feedback loop* around it: production traces flow to human annotation, where domain experts, product managers, and QA review and label real failures, and those labeled failures are curated back into the golden dataset for the next cycle. Your test coverage evolves with real usage, so it keeps catching what actually breaks in the wild rather than what you imagined might break at the start. And this is a team sport, not an engineering chore — the people who know what "good" means for a legal answer or a clinical summary are the domain experts, not the engineers. Build the annotation workflow so they can contribute without engineering as the bottleneck, because that is where quality actually gets defined. A team that treats evaluation as one person's side task has already decided to be in the ninety-five percent.

Three places evaluation lives

The same evaluation discipline shows up at three points across the lifecycle, and a serious system uses all three.

Pre-deployment is the gate from Chapter 8: run the golden set before a change ships, and catch the regression — a prompt rewrite that quietly drops factuality from ninety-four percent to eighty-nine — before a single user meets it. *Production monitoring* scores live or sampled traffic after deployment; because real volume dwarfs any test set, you lean on cheaper scorers and sampling, and this is how you catch drift. *Runtime guardrails* inspect the output before it ever reaches the user — blocking, rewriting, or routing a high-risk re-

sponse for review, and intercepting a dangerous agent action before it executes, a leaked identifier or a policy violation stopped in flight. This last one is where evaluation meets Chapter 8's control frame and Chapter 11's security: it's the real-time safety net beneath everything.

Same discipline, three touchpoints: catch it before you ship, watch it after you ship, and stop the worst of it in the moment.

Catching drift: the degradation nobody deployed

Chapter 8 named the fear; this is where you detect it. *Silent drift* is a gradual decay in quality — or cost — that no aggregate dashboard reveals, because averages smooth it over and latency stays perfectly flat while the answers quietly get worse. Nobody deployed the degradation. It crept in.

Trace it back and every cause lives somewhere in the response path. A retrieval index went stale, so the model grounds on last quarter's facts. The provider silently updated the base model — Chapter 8's prompt drift — and your unchanged prompt now behaves differently. A prompt edit reached production without enough evaluation coverage. The embedding distribution shifted underneath your vector store. In each case your code never moved, and the quality fell anyway.

Detecting it requires one inversion of your instincts: **alert on quality, not just on latency.** The alarm should fire when an evaluation score drops, not only when a request is slow — and it should watch drift at the level of the individual prompt, use case, or user segment, not just the global aggregate, because a serious problem in one work-

flow drowns unnoticed in an average across all of them. The goal is blunt and it is serious: know before your users do, and in a regulated industry, know before your regulators do.

Debugging non-determinism

When quality does drop, you face the last unfamiliar problem: how do you debug something that never behaves the same way twice? Not the way you debug deterministic code. There is no single broken line to find. Instead you reason about a *path* and a *distribution*, and the trace tree is what makes that tractable.

The method is root-cause by elimination, walking the tree. Was it retrieval — wrong chunks, or none (Chapter 7)? The prompt — a recent edit? The model — drift, or an upstream update? A tool — an error the agent quietly worked around? The router or a fallback — a different, weaker path served without anyone noticing? Carry Chapter 8's lesson with you as your prior: the model is usually *not* the culprit, and the trace is what tells you which component actually moved. Debugging here is detective work over a recorded path, not a breakpoint in a repeatable run.

And this is where the thread that started in Chapter 1 finally ties off. Non-determinism was named early as the one genuinely new muscle, and it has shadowed every chapter since. Here is where it stops being frightening. You were never going to get a deterministic model, and chasing one was the wrong goal all along. What you can have — what this chapter builds — is a system that is completely observable, measurable, and diagnosable: every output seen, every output scored, every output traceable to its cause, even though none of them can be predicted in advance. That is control of a different

kind than deterministic software gave you, and for this material it is not a lesser kind. It is the right one.

Observability, evaluation, and debugging are the least glamorous work in the book, and the most decisive. They are what turn a demo that dazzled in a conference room into a system you can trust on Monday and defend to an auditor on Friday. They are precisely the discipline the ninety-five percent skip on their way to a launch that quietly rots. Building this is what puts you in the five percent — and, just as importantly, what keeps you there.

Experiments to run

Small enough to start today. A tracing-and-eval starter kit at leapfrog.lerias.org.

1. **Trace one real request end to end.** Instrument a single request and look at the whole tree — generation, retrieval, tool calls, each with its inputs, outputs, and cost. Notice how much your old logs never showed you.
2. **Score a golden set.** Build a small set of representative inputs, pick one metric — faithfulness is a good first — and score your outputs to get a real number. Then curate five genuine failures back into the set. You've just started the feedback loop.
3. **Alert on quality, not latency.** Wire an alert that fires when an evaluation score drops rather than when a request is slow. Break something on purpose and

watch it catch what your infrastructure dashboards never would.

4. **Root-cause a bad answer.** Take one wrong output and walk its trace to name the actual culprit. Bet, before you look, that it's the model — and notice how often it isn't.

References for this chapter

Open standards and methods first; tools named only as category examples. Tracing-and-eval starter on leapfrog.lerias.org.

Seeing the system - OpenTelemetry GenAI semantic conventions and OpenInference — the open-standard, portable instrumentation layer; the trace tree as the unit of work.

Judging quality - Evaluation methods and metrics: LLM-as-a-judge (and the research on its blind spots and alignment to human ratings), faithfulness and groundedness scoring, and frameworks such as Ragas and DeepEval — cited as methods, not endorsements. - The golden-dataset feedback loop: production traces to human annotation to curated datasets, with domain experts and QA in the loop.

Watching and debugging in production - The three evaluation touchpoints — pre-deployment gate, production monitoring, runtime guardrails — and quality-aware alerting on score drops at the prompt/segment level. - Observability and evaluation platforms as category examples: Langfuse, Arize Phoenix, LangSmith, Braintrust,

MLflow, Datadog, Galileo, Evidently — evaluated against your stack and residency needs. Ties to Chapter 7 (retrieval evaluation) and Chapter 11 (guardrails as security).

PART IV

IV

At Scale & Forward

Cost, performance, security, and the road ahead.

10 Scaling, Performance, and Cost Control

CHAPTER 10

Per-token prices are collapsing, and your AI bill is climbing anyway. That contradiction is the whole chapter — and resolving it is an engineering job, not a procurement one.

Chapter 5 taught you the mechanics of a single call — its tokens, its latency, its cost. This chapter is what happens when that call runs ten million times a day across a dozen teams: when cost and performance stop being properties of a request and become properties of an organization. Running AI at scale is a discipline, and it's the one that decides whether your AI program is a durable product or a runaway line item on next quarter's invoice.

The paradox: prices fell, the bill tripled

Here is a true story that has happened, in some form, to a great many teams. A platform group swapped the model behind their flag-

ship feature for one costing roughly ten times less per token, checked the price of a single request, and proudly told their VP they'd cut the feature's running cost by ninety percent. Six weeks later, the monthly bill for that feature had *tripled*.

Nobody lied. Both numbers were real. The per-request cost genuinely fell, and the total bill genuinely exploded — because cheaper requests invited more of them, usage grew, and the arithmetic that governs a bill at scale is not the arithmetic of a single call. That gap between collapsing prices and climbing bills is the entire reason "FinOps for AI" went, in two years, from a niche phrase to the discipline nearly every finance and engineering organization is now scrambling to build. The surveys tell the story bluntly: the share of cost-management teams responsible for AI spend went from roughly a third, to two thirds, to essentially all of them in the span of two years — not because AI became a boardroom priority, but because the invoices arrived and nobody was ready.

Three forces keep the bill rising even as unit prices fall, and you should name them so they don't surprise you. Volume: cheaper and more useful means more usage, always. The reasoning tax from Chapter 5, at scale: reasoning and agentic workloads consume something like five to thirty times more tokens per task than a plain chat turn, and that multiplier swamps the per-token savings at the bottom of the range. And the end of the subsidy phase: frontier providers priced below true cost early to win adoption, and as enterprise consumption outran the rate at which per-token costs were falling, that math broke — list-price declines have slowed and concentrated in the commodity tiers, while frontier pricing holds and providers move toward pre-committed capacity models that make you forecast token demand the way you once forecast cloud. Cost

per call is a Chapter 5 problem. Cost at scale is a different animal, and it is this chapter's.

Cost is an engineering metric, not a finance line item

The single most important reframe in this chapter is this: **the cost of an AI feature is decided in your architecture and your code, not in a procurement contract.** It lives in your prompt templates, your retrieval configuration, your model selection, and the loop conditions of your agents — decisions made in pull requests, sprint after sprint, by engineers.

Which means the classic corporate instinct — treat the AI bill as a line item finance will optimize later — is not just wrong, it's structurally impossible. Asking finance to reduce your token spend is like asking finance to make your application faster: they can see the number, and they cannot move it, because nothing that drives it lives on their side of the house. The teams that stay in control treat LLM cost exactly the way mature engineering organizations already treat latency and reliability — as a design constraint owned by the people writing the code, measured continuously, and checked before release rather than explained after the invoice lands. The cost lever moved from procurement to the delivery team. Pick it up, because no one else can.

Visibility, then attribution, then optimization — in that order

Most organizations, handed a scary bill, jump straight to "how do we spend less." That is the wrong first move, and it's why so much AI

cost-cutting fails. The sequence that works is fixed: **visibility, then attribution, then optimization.**

Visibility is simply knowing what's being spent — the part everyone has, because the invoice states it. *Attribution* is the hard, decisive step: mapping that spend back to a specific product, feature, team, tenant, or customer. Without it you cannot prioritize, because you can't tell whether a doubling bill means a feature customers love is scaling or a low-value use case is leaking money. And AI billing data does not come pre-tagged the way cloud resources do — so you create the tags yourself, attaching metadata (team, feature, environment, tenant) to every model call, which the gateway from Chapter 3 and the traces from Chapter 9 are exactly the right places to do. Only once you can see where the money goes does *optimization* become a rational conversation instead of a blind one.

The metric that crowns this is *unit economics*: not the total bill, which on its own is nearly useless, but the cost per unit of value — cost per inference, per feature, per resolved ticket, per booking, per customer. And you report it in the listener's vocabulary. A CFO does not think in tokens; they think in cost per transaction and margin per customer, and a number that doesn't map to their profit-and-loss language cannot support a budget decision no matter how accurate it is. The discipline's own leaders have retitled the whole practice around *value* rather than savings for this reason: the metric to chase is value per token, not cost per token. Wrapped around all of it is the governance layer — budgets, anomaly detection, and chargeback — including budget guardrails that stop a runaway agent or a looping prompt from quietly incinerating a quarter's budget before anyone looks, which is Chapter 8's bounded execution wearing a FinOps hat.

The optimization levers, at scale

Once you can see and attribute, the levers are largely the ones from Chapter 5 — now applied as a standing discipline rather than a one-time tuning pass. Route by difficulty, sending the cheap, high-volume work to small models and reserving frontier models for the tasks that need them. Cache aggressively: at scale, semantic caching that recognizes repeated or near-repeated requests stops you from paying twice for the same answer. Manage context so you retrieve only what's relevant rather than stuffing the window (Chapter 7). Control output length, the highest-leverage lever of all. Batch anything that isn't interactive. And treat even the *format* of your structured output as a cost lever — some serializations consume markedly fewer tokens than verbose JSON for the same data.

One caveat governs all of these, and it's the bridge to the previous chapter: **every cost optimization needs a quality check.** Unlike trimming a cloud instance, where the workload either runs or it doesn't, cutting AI cost can silently degrade output — a cheaper model, a trimmed context, a shorter answer can all save money while quietly making the product worse. So each optimization runs against the evaluation harness from Chapter 9, and you keep the ones that hold quality. And you do it continuously, not as a heroic quarterly cleanup, because usage patterns, prices, and models all shift underneath you month to month. Optimization is an operating loop, not an event.

Performance at scale: serving, autoscaling, and the tail

Cost is half the scaling story; performance is the other. And serving inference at scale is a genuinely different problem from training a

model: where training chased ever-bigger models, serving is about running them cost-efficiently and reliably, continuously, all day, for users who expect a fast first token and a consistent response every single time.

Most of the machinery you already met — the accelerators and autoscaling serving stack of Chapter 4, the scale-to-zero economics, the batching and caching of Chapter 5. What changes at scale is where your attention goes. Autoscaling a GPU fleet against real, spiky load without either melting under a surge or paying for idle capacity is the core operational challenge, and it's harder than autoscaling ordinary compute because the accelerators are scarce and expensive (Chapter 4). Capacity planning becomes forecasting: with providers moving toward pre-committed token capacity, and with the sheer physical reality behind it — electricity demand from AI data centers grew on the order of fifty percent in a single recent year, against low-single-digit growth for electricity overall, driven specifically by reasoning and agentic workloads — you now forecast AI compute demand the way you long forecast cloud.

And watch the *tail*. A healthy median latency hides a multitude of sins; what users actually feel is the ninety-ninth-percentile request, the one that stalls. A system that's fast on average and occasionally awful feels broken, so at scale you measure and defend p99 and beyond, not just the median — the same discipline you'd apply to any latency-sensitive distributed service, because that is exactly what this now is.

Inference at the edge: the distributed decision layer

Which brings us, fittingly, back to where the book began. Chapter 1 called this era the Convergence — cloud, edge, and intelligence collapsing into one continuum — and nowhere is that more literal than in where inference runs.

For most of the last decade the default was simple: run inference centrally in the cloud, return results by API. That worked when AI was used occasionally and a second or two of latency was fine. Neither condition holds anymore. Centralized inference adds latency, piles up egress costs, and runs into data-residency constraints — and all three *compound* as inference volume grows (the data gravity of Chapter 4, the residency of Chapter 7, now multiplied by scale). The response, increasingly, is to push inference outward, toward the users and the places where data is actually generated. As one industry framing puts it, stop treating the edge as a remote extension of the cloud and start treating it as a distributed *decision layer* — the place where intelligent systems make real-time decisions close to where they matter: voice agents, fraud scoring, personalization, robotics, factory floors, agentic workflows.

Be honest about the shape of it, though, because this is where hype outruns reality. Edge inference today is for *small, fast, specialized* models — classification, extraction, moderation, scoring — not frontier-scale general generation, which the hardware near the user can't hold. So the mature architecture is the hybrid you already met in miniature in Chapter 2: run the routine, latency-sensitive work on a small model at the edge, and escalate the genuinely hard requests to a frontier model in the cloud. And the real gatekeeper is not capability but *operations*: running a distributed inference fleet

means updating, versioning, monitoring, and evaluating models across many sites at once — Chapters 8 and 9, multiplied across geography. The right time to reach for the edge is when the business case — latency, egress cost, residency, resilience — is clear enough to justify building or acquiring that operational capability, and not a moment before.

But when it is justified, this is the frontier the industry is moving toward, and it is the oldest idea in this book made new: put the compute where the users are. The engineers who spent careers learning to serve content and run logic at the edge of the network are, it turns out, exactly the ones equipped to serve *intelligence* there too. The Convergence isn't a prediction. For anyone who came up through distributed systems and the edge, it's a homecoming.

Experiments to run

Small enough to start today. A unit-economics worksheet and edge-decision calculator at leapfrog.lerias.org.

1. **Attribute one feature's spend.** Pass team-and-feature metadata through your gateway, join it to the bill, and produce a single number: the cost per transaction of one real feature. Then say that number out loud in your CFO's vocabulary, not in tokens.
2. **Find your value per token.** For one workload, compute the cost per unit of value it produces — per resolved ticket, per booking, per document. Decide,

with a straight face, whether it's worth it. That decision is the whole discipline in miniature.

3. **Cache, then verify.** Add semantic caching to a repeated workload and measure the cost drop — then run the Chapter 9 eval to confirm quality held. A saving that degrades the product isn't a saving.

4. **Do the edge math honestly.** Pick one latency-sensitive, high-volume workload and estimate centralized versus edge: latency, egress cost, residency — and the operational overhead of running a fleet. Let the full picture, not the latency number alone, make the call.

References for this chapter

Primary and durable sources first; cost and infrastructure tools named only as category examples. Worksheets on leapfrog.lerias.org.

FinOps and token economics - The FinOps Foundation — *State of FinOps 2026*, the *FinOps for AI* guidance, the token-economics (TokenOps) framing, and the FOCUS open cost-and-usage specification for normalizing spend across providers. - The "cost is an engineering metric / value per token" argument and unit-economics practice (attribution, budgets, chargeback). - AI cost-management platforms as category examples — Finout, Vantage, CloudZero, and others — evaluated against your stack and attribution needs.

Scaling and performance - Chapter 4 (accelerators, autoscaling serving stack, scale-to-zero) and Chapter 5 (cost levers, latency) as the mechanics this chapter operationalizes. - Stanford HAI *AI Index* and the International Energy Agency on the compute and energy demand behind inference at scale.

Inference at the edge - Edge-AI inference analyses and the cloud-versus-edge hybrid decision frameworks (latency, egress, residency, resilience, and the operational overhead of a distributed fleet). - The distributed-decision-layer framing — the Convergence of Chapter 1, made concrete at the serving layer.

11 Security, Governance, and the Road Ahead

CHAPTER 11

The same properties that make AI powerful — that it acts, that it's non-deterministic, that it reads instructions and data through a single channel — are exactly what make it attackable and what make regulators want a say. This is the reality you deploy into. None of it is a reason to wait; all of it is your turf.

This is the last chapter, and it does three things. It gives you the adversarial reality — how these systems are attacked, and how you contain what you can't prevent. It gives you the governance reality — the regulations arriving now, framed as architecture you build rather than paperwork you dread. And then it pulls the whole book together into a single reference architecture and a roadmap you can walk from Monday. Security and governance are where corporate engineers are most tempted to defer to someone else. Don't. They are the most engineering-shaped problems in the book, which means they are yours.

The adversarial reality: the attack you can't patch

Start with the one that matters most, because it is both the most exploited and the most misunderstood. *Prompt injection* has held the top of the OWASP list of LLM risks for two editions running, and it is not a bug you will fix. It is a consequence of how these models work: a language model reads instructions and data through the same channel, with no clean separation, so an attacker can craft input the model interprets as a new instruction rather than as content to process — and the model obeys, because it genuinely cannot tell the difference. Security researchers have taken to calling it the new SQL injection, and the analogy is apt except for one crucial difference: SQL injection has a clean fix, and prompt injection does not. You cannot patch your way out of a design property.

It comes in two forms, and the second is the one that should worry an enterprise. *Direct* injection is a user typing a malicious instruction. *Indirect* injection hides the instruction inside content the model will later retrieve — a document, a web page, an email, a support ticket — so that a perfectly innocent user question pulls poisoned content into the context and the model acts on the buried command. Every retrieval-augmented and agentic system you build is exposed to this by construction, and the attacker needs no special access at all. This is not theoretical: there are documented cases of data exfiltrated from an AI assistant through content it was fed, and of an agent tricked via leaked prompt logic into abusing a URL-reading tool to steal configuration secrets. And the numbers are sobering precisely because they're improving but not solved — a recent frontier model's own safety documentation reported indirect-injection success rates

in an agentic setting climbing from a few percent at one attempt to well over half at a hundred attempts. Defenses reduce the risk; they do not eliminate it.

Which is why the strategy is the one Chapter 8 already gave you, now with its adversarial justification made explicit: you assume injection can succeed, and you *contain the blast radius*. Defense in depth — validate inputs, treat every model output as untrusted until checked, segregate untrusted external content from trusted instructions, restrict what tools and permissions the model can reach, and put a human in the loop for anything irreversible. The model layer is porous by nature. The controls you wrap around it are not, and that is where your security actually lives.

The rest of the threat surface

Prompt injection is the headline, but a defensible program maps the whole surface, and the OWASP catalog of LLM and agentic risks is the community's shared coverage map — grounded in real incidents, and the right checklist to threat-model against. The rest, briefly, because each one connects to something you've already built:

Sensitive information disclosure and *system prompt leakage* — models can reveal training data, retrieved records, or their own system prompts, so the rule is simple: never put secrets or confidential logic in a prompt, and enforce your controls outside it, because a prompt is not a secure place. *Improper output handling* — the mirror of injection: treat every model output as untrusted data, and validate and encode it before it ever reaches a database, a browser, or a shell, or you've built a fresh injection vector downstream. *Supply chain* and *data-and-model poisoning* — your model, its training data, and its dependencies are an attack surface, and research shows that a strik-

ingly small number of malicious documents can poison a model's behavior. *Excessive agency* — an agent with more tools and permissions than its task requires is a larger blast radius waiting to happen; least privilege, from Chapter 8, is the direct control. *Vector and embedding weaknesses* — the retrieval layer of Chapter 7 carries its own leakage and poisoning risks, and embeddings are derived data that can expose what they were built from. *Unbounded consumption* — the resource and cost attacks that Chapters 8 and 10 bound with limits and budgets. And *multimodal* inputs widen the surface further, since an instruction can now hide inside an image as easily as inside text.

You don't memorize this list; you use it. Map your system against it, then *red-team* — attack your own thing deliberately, using the OWASP taxonomy as your script and frameworks like NIST's AI Risk Management Framework and its generative-AI profile, and MITRE's ATLAS knowledge base of real adversary techniques, as your reference. The discipline is exactly the security engineering you already know, pointed at a new component whose defining trait is that it can be talked into things.

Governance you can build: the regulatory landscape without the panic

Now the part corporate engineers most want to hand to legal — and shouldn't hand over entirely, because most of what regulation asks for is architecture, and architecture is yours. Three frameworks matter, and they fit together rather than competing.

The **EU AI Act** is mandatory law, and the world's first comprehensive one. It sorts every system into four risk tiers — unacceptable (banned), high, limited (transparency obligations), and minimal

(most systems, essentially untouched) — and puts the heavy obligations on high-risk uses. Two features catch enterprises off guard. It is *extraterritorial*: it applies to any organization whose AI is placed on the EU market, used by people in the EU, or whose output is used there, wherever the company is headquartered. And it distinguishes *providers* from *deployers* — so a company that embeds a third-party model into its own product is very often both at once, and "we only call someone else's API" is not the exemption people assume it is.

Its timeline is phased, and here you must check the current state rather than trust a printed date, because it is actively moving. Prohibited practices and AI-literacy duties applied from early 2025; general-purpose-model obligations from mid-2025; broad transparency rules land in 2026. The full high-risk obligations were originally set for August 2026 — but a simplification package (the "Digital Omnibus") agreed in the first half of 2026 has deferred them, pushing standalone high-risk systems to late 2027 and product-embedded ones to 2028, with the original dates snapping back if that package isn't formally adopted in time. The penalties are large enough to command attention (into the tens of millions of euros or a meaningful percentage of global turnover for the worst violations). The specific dates will keep shifting; the direction — real, enforceable, risk-tiered obligations — will not. Track the current status rather than any single reference (leapfrog.lerias.org keeps a live pointer).

Around that sit two voluntary frameworks that make compliance practical. The **NIST AI Risk Management Framework** is a flexible, sector-agnostic structure organized around four functions — govern, map, measure, manage — with a dedicated generative-AI profile; it tells you what good practice looks like. **ISO/IEC 42001** is the first certifiable AI management-system standard, provable through an accredited audit, and increasingly a line item in enterprise

procurement questionnaires; it lets you *prove* the practice. The three share a common core — risk assessment, human oversight, accountability, documentation, continuous monitoring — so a program built well against one advances the others, and the efficient path is to use NIST for method, ISO 42001 for structure and evidence, and layer the EU AI Act's specific obligations on top where you have exposure. One honest gap worth flagging: none of these was designed for agentic AI, so cascading failures, scope creep, and attribution across autonomous steps are things you'll have to extend the frameworks to cover yourself.

Governance as architecture, not paperwork

Here is the reframe that turns all of that from a burden into your work: governance, done well, is *shift-left engineering* (Chapter 4), and its deliverables are things engineers build, not memos lawyers file.

Start with an **AI system inventory** — a living registry of every AI system and feature running in your organization, each classified by risk tier. This sounds mundane and is foundational, because the most common enterprise reality is not knowing what's running or where, and you cannot govern, secure, or budget for what you can't see (the shadow AI is always more than you think). On top of the inventory sit a small set of artifacts that make governance inspectable: a *control catalog* listing each safeguard and how it's enforced at runtime, a *compliance matrix* mapping each control to the framework clauses it satisfies, and a *risk register* naming the owners, mitigations, and evidence for risks like data leakage or unauthorized action. Each high-risk system gets a *named accountable owner* and, crucially, *stop authority* — a specific person with the explicit right to

pause, halt, or roll back the system in production without waiting for escalation. That last one is the tell: it's the EU AI Act's human-oversight requirement and Chapter 8's kill switch, wearing a governance hat, and if nobody actually holds that authority, your governance documents are theater.

Two closing points on governance, because they change how it feels. First, it's a competitive asset, not just a cost: in regulated industries, being able to *prove* your controls wins the contracts that competitors lose to delay. Second, this is compliance-by-design — you build the controls into the architecture from the start (the risk-tiered instinct: rigorous documentation and human oversight for the high-risk system, a light touch for the low-risk one), rather than reverse-engineering them under audit pressure later. Regulation, approached this way, stops being a wall and becomes scaffolding.

A reference architecture: the book on one page

Step back, and everything in this book assembles into a single, vendor-neutral shape — not a product to buy, but a set of layers to reason about, each of which you now understand.

At the base is the **cloud foundation** (Chapter 4): the control plane, containers, identity, and infrastructure-as-code you already command. Above it, a **model-access layer** reached through a **gateway** (Chapters 3 and 5) that abstracts providers, so you route, swap, pin, and fall back without rewrites — your insurance against lock-in and against the model changing under you. Above that, the **orchestration and pattern layer** (Chapter 6): context engineering, structured outputs, tool use through open protocols, and the workflows-and-agents logic that does the actual work, sitting beside a **retrieval**

and grounding layer (Chapter 7) that connects the model to your data with permission-awareness built in. Wrapping the runtime is the **operations layer** (Chapter 8): deployment as versioned response paths, eval gates, safe rollout, durable execution for agents. Watching all of it is the **observability and evaluation layer** (Chapter 9): open-standard tracing with quality scored on the spans. And threaded through every layer, not bolted on at the end, are the three cross-cutting concerns this book has returned to again and again — **cost** (Chapters 5 and 10), **security and control** (Chapters 8 and 11), and **vendor neutrality** (Chapters 3 and 6). Above it all sits the layer that decides whether any of it matters: the **workflow you chose to redesign** (Chapter 2), because the architecture serves the co-invention, never the other way around.

That is the whole book in one picture. If you can draw it and say where each of your decisions lives on it, you can build, run, and defend AI at scale.

The road ahead, and the durable core

It's customary to end a book like this with predictions, so here are the honest ones — agents will keep maturing from assistants you visit into systems that run whole workflows; the general-purpose-technology future of Chapter 2 will keep arriving, AI dissolving into everything until it's as unremarkable as the spellchecker; inference will keep moving toward the edge, the Convergence completing; the open standards will keep hardening; the injection arms race will continue, defenders and attackers trading moves; and regulation will keep converging across jurisdictions toward a common risk-tiered core. Some of that will be wrong. Prediction is not the point.

The point is what *doesn't* change, and it is the reason this book was written the way it was. The specific models will turn over — several times before this ink is dry. But the fundamentals underneath them are durable: the cloud-engineering discipline, the evaluation muscle, the containment mindset, the co-invention that turns a model into value, and the neutrality that keeps you free. Those outlast every release, which is why the book taught the layer beneath the vendor rather than this quarter's tool. Learn the durable core and you are never behind, no matter how fast the frontier moves.

So here is the roadmap, and it is deliberately small at the start, because that is the whole ethos. You do not begin with a platform or a strategy deck. You begin by choosing one workflow worth re-designing, grounding one model in one real dataset, putting one thing behind a deployed endpoint, and wrapping it in one eval and one control. Then you compound: attribution and guardrails and a governance registry; then agents, the edge, and an organization-wide platform, as the value earns each next rung. That progression — not a big bang — is how the five percent got there, and how you will.

Which returns us, at the end, to where we began. The barriers a corporation places in front of delivering AI are mostly deferral dressed as prudence, and AI has revoked the patience that made deferral survivable. You were always qualified for this — it was a cloud-engineering problem all along, and that is your ground. The map in these pages was drawn by someone who got lost a great deal, pointing at the researchers and builders in the references who charted the real territory; the credit is theirs, and the walking is yours. So go experiment. Ship something small this week. The territory is open, and you were always equipped to cross it.

Experiments to run

Small enough to start today. Red-team scripts, a one-page risk-note template, and an AI-inventory starter at leapfrog.lerias.org.

1. **Red-team your own thing.** Take something you built and attack it. Type a direct injection; then hide an instruction inside a document it will retrieve and watch whether it obeys. Fifteen minutes here teaches more than any threat report.

2. **Write the one-page risk note.** For one system, on a single page: what it can do (its blast radius), what could go wrong (the two or three OWASP risks that actually apply), what controls bound it (Chapter 8), and who holds stop authority. That page is governance made real.

3. **Inventory your AI.** List every AI system and feature running in your corner of the organization and classify each by risk tier. You will find shadow AI you didn't know about — and that list is where governance actually begins.

4. **Place yourself on the roadmap.** Rate honestly where you and your organization are — one workflow, or attribution, or guardrails, or agents — then name the single next rung and go ship it. The roadmap only works if you take the first step on it.

References for this chapter

Primary and standards bodies first; tools named only as category examples. Live regulatory pointers and templates on leapfrog.lerias.org.

Security - OWASP — the *Top 10 for LLM Applications* (2025) and the *Top 10 for Agentic Applications* (2025), the shared coverage maps; documented real-world incidents of injection and exfiltration; and frontier-model system cards reporting measured injection success rates. - NIST — the *AI Risk Management Framework* and its *Generative AI Profile* (AI 600-1); MITRE *ATLAS* for adversary techniques. Red-teaming tooling as category examples.

Governance and regulation - The **EU AI Act** (Regulation (EU) 2024/1689), its risk tiers and phased timeline, and the 2026 “Digital Omnibus” deferral of high-risk obligations — verify the current status, as dates are moving. - The **NIST AI RMF** (govern/map/measure/manage) and **ISO/IEC 42001** (the certifiable AI management-system standard), and the published crosswalks that let one program satisfy several frameworks.

Synthesis - The reference architecture draws every prior chapter together; the roadmap and the durable-core argument close the book’s through-line from Chapter 1. Credit for the frameworks belongs to the researchers and standards communities named throughout — this book only arranges their work into a path.

Back Matter

Glossary

Plain, working definitions for the terms this book leans on. Where a term has a whole chapter behind it, the chapter is noted.

Token — the unit a model reads and writes; a chunk of text (or image or audio) roughly a few characters long. You are billed per token, in and out (Ch 5).

Context window — the maximum number of tokens a model can consider at once. The *effective* context — how much it can actually use well — is usually smaller than the advertised number (Ch 5).

Inference — running a trained model to get an output. The whole cost-and-latency game of serving AI is inference, not training (Ch 5).

Embedding — a numeric representation of meaning, so that similar things sit close together in vector space. The basis of semantic retrieval (Ch 7).

Vector store — a database that indexes embeddings for similarity search; a production decision, not a default (Ch 7).

RAG (retrieval-augmented generation) — fetching relevant information at query time and feeding it to the model as context, instead of relying on the model's baked-in knowledge (Ch 7).

Chunking / reranking / hybrid search — the mechanics of good retrieval: splitting documents sensibly, re-scoring candidates for relevance, and combining keyword with semantic search (Ch 7).

Prompt / system prompt — the instructions given to a model; the system prompt sets standing behavior. A prompt is code, and a prompt change is a deployment (Ch 6, Ch 8).

Context engineering — deliberately assembling what goes into the context window (instructions, examples, retrieved data) for a given task (Ch 6).

Structured output — forcing a model to return machine-readable data (e.g., a fixed schema) your code can act on (Ch 6).

Tool use — letting a model call functions or services to fetch data or take actions, often via an open protocol (Ch 6).

MCP (Model Context Protocol) — an open standard for connecting models to tools and data sources without bespoke glue (Ch 3, Ch 6).

Agent vs. workflow — a workflow runs predetermined steps; an agent decides its own steps at runtime. Start with workflows; earn agents (Ch 6).

Orchestration / durable execution — coordinating multi-step processes and surviving crashes mid-run (state, retries, recovery). Agents are long-running distributed systems (Ch 8).

Gateway — an abstraction layer in front of model providers that lets you route, swap, pin, and fall back without rewriting the application. Your main defense against lock-in (Ch 3).

Fine-tuning — adapting the model's own weights to your task. Good for *form* (tone, schema, vocabulary), bad for *facts* (use RAG). Usually not the first move (Ch 7).

LoRA / QLoRA — parameter-efficient fine-tuning: train a small adapter on a frozen base model rather than retraining everything (Ch 7).

Distillation — training a smaller, cheaper model to mimic a larger one on a task (Ch 7, Ch 10).

Quantization — compressing a model's weights to lower precision to cut memory and cost, usually with modest quality loss (Ch 4, Ch 5).

Reasoning model / reasoning tax — models that "think" before answering, spending extra tokens (and cost and latency) for better results on hard tasks. Worth it sometimes, not always (Ch 5).

Multimodal — models that take images, audio, or video as well as text; everything still becomes tokens, and images are token-hungry (Ch 5).

Hallucination — a confident, fluent, wrong output. The failure mode that looks like success (Ch 1, Ch 9).

Non-determinism — the same input can produce different outputs. The one genuinely new engineering muscle; you can't make it predictable, but you can make it observable and bounded (Ch 1, Ch 8, Ch 9).

Guardrail — a runtime check that inspects inputs or outputs and blocks, rewrites, or escalates the risky ones (Ch 9, Ch 11).

Evaluation / LLM-as-judge / golden dataset — measuring output quality; often using a model to score another model's output against a rubric; the golden dataset is your standing set of known-good cases (Ch 9).

Observability / trace — instrumenting a request so you can see the whole path (prompt, retrieval, tools) as a tree of spans, with quality scored on it (Ch 9).

Drift / prompt drift — silent quality decay, often because a provider updated the base model underneath your unchanged prompt (Ch 8, Ch 9).

LLMOps — the discipline of deploying and operating LLM applications: prompts as deployments, evals as gates, the whole response path as the versioned artifact (Ch 8).

FinOps / TokenOps — treating AI cost as an engineering metric: visibility, attribution, and value per token, owned by the team that writes the code (Ch 10).

Least privilege / blast radius — giving a system only the access its task requires, so that when something goes wrong, the damage is contained (Ch 8, Ch 11).

Prompt injection — an attack that hides instructions in input (directly, or indirectly via retrieved content) that the model then obeys. A design property you can't patch, only contain (Ch 11).

Jagged frontier — the finding that AI helps on some tasks and hurts on others of similar difficulty, unpredictably; you map a workflow against it task by task (Ch 2).

Co-invention — the complementary redesign of workflows, processes, and skills around a new technology, which is where its value actually comes from. The thing the 5% do (Ch 2).

General-purpose technology — the economists' category for technologies like electricity and the internet that end up embedded in everything. AI's destination, and why the chatbot is a trap (Ch 2).

Edge inference — running models close to users and data rather than in a central cloud, to cut latency, egress, and residency exposure (Ch 10).

Appendix — Staying Current

This book taught the layer *beneath* the tools on purpose, because that layer is durable. But the surface moves monthly — new models, new prices, new features — and you'll need a way to stay current without drowning in it. A few habits keep the map fresh:

Follow primary sources, not the hype cycle. The labs' own documentation, the standards bodies (OWASP, NIST), the research institutions (Stanford HAI's AI Index, MIT, Harvard), and *independent* benchmarks tell you more than aggregators and vendor leaderboards. A small rotation of trustworthy sources beats a firehose of takes.

Track the layer, not the release. When a new model lands, don't just read the benchmark score. Ask the structural questions this book taught you: what changed in the cost curve, the effective context, the modalities, the reasoning behavior, the tool support? Those tell you whether anything in your architecture should change.

Keep your own evals. Your golden dataset is your private source of truth. When a model updates or a price shifts, re-run it. Your evals will tell you whether a shiny new release is actually better *for you*, which no leaderboard can.

Verify the fast-moving facts. Prices, model names, context limits, and regulatory dates (the EU AI Act timeline especially) change in weeks. Treat any specific number in this book as current-as-of-writing and check it before you rely on it.

Live pointers to current sources, benchmarks, and the regulatory status are kept at **leapfrog.lerias.org**, alongside the hands-on labs.

Standing on Shoulders

This book is a map, and the territory was charted by others. Almost every idea in it belongs to someone smarter than me, and this page is where I say so plainly.

The framing of *where* AI belongs rests on the "jagged frontier" work of Fabrizio Dell'Acqua, Karim Lakhani, Ethan Mollick, Katherine Kellogg, and their colleagues across Harvard Business School, MIT Sloan, Wharton, and BCG — and on Erik Brynjolfsson and the Stanford Digital Economy Lab, whose work on general-purpose technologies, the productivity J-curve, and enterprise co-invention runs underneath Chapters 1 and 2. Andrew Ng's "AI is the new electricity" gave the whole thing its shortest expression. Stanford HAI's AI Index and MIT's work on the enterprise "GenAI divide" supplied the numbers that anchor the argument.

The build and operate chapters stand on the open-source and open-standards communities: the people behind the Model Context Protocol and OpenTelemetry, and behind the serving, retrieval, and observability tools that made all of this reachable by ordinary teams. The security and governance chapters lean on OWASP's LLM and Agentic Top 10 projects, NIST's AI Risk Management Framework, MITRE's ATLAS, the authors of ISO/IEC 42001, and the drafters of the EU AI Act. And throughout, the specific studies cited in each chapter's references — on retrieval, on context limits, on evaluation, on cost — did the real intellectual work.

To all of them: thank you. The credit here is yours. Any mistakes are mine alone. This book only tried to arrange your work into a path a busy engineer could walk.

— *Hugo Lerias*