

Local LLMs for Manufacturing

Small language models
on the plant floor



Claude Fable 5

verified by Roger AI

Local LLMs for Manufacturing

Small language models on the plant floor

WRITTEN BY claude-fable-5

VERIFIED BY Miguel Ramos

DISCLOSURE Draft v0. Written by Claude Fable 5 operated by RogerAI Labs; unverified; not for publication.

PROVENANCE & REVIEW TRAIL <https://oailly.com> - the full review record publishes with this book.

Contents

Front Matter	4
1. The 518-Page Silence	5
2. A Language Model, for People Who Own Machines	13
3. Why Small	21
4. Reading the Plant: Protocols and Historians	29
5. The Abstention Chapter	37
6. The Quality Gate	45
7. Training on the Real World	53
8. Deployment Shapes	61
9. Surviving Reality	69
10. The Honest Deployment Checklist	76
Back Matter	83

Local LLMs for Manufacturing

Small language models on the plant floor

O'AILLY Industrial Series · N° 1 · REV 1.0 (draft)

Contents

- Chapter 1 — The 518-Page Silence
- Chapter 2 — A Language Model, for People Who Own Machines
- Chapter 3 — Why Small
- Chapter 4 — Reading the Plant: Protocols and Historians
- Chapter 5 — The Abstinence Chapter
- Chapter 6 — The Quality Gate
- Chapter 7 — Training on the Real World
- Chapter 8 — Deployment Shapes
- Chapter 9 — Surviving Reality
- Chapter 10 — The Honest Deployment Checklist

Introduction

This book is for people who own machines and answer for uptime: plant engineers, controls engineers, reliability leads, and the integrators who serve them. It assumes fluency with the plant — historians, protocols, service manuals — and no machine-learning background whatsoever. Its claim is narrow and testable: a small, locally hosted language model, grounded in your documents and constrained to your schemas, can do genuinely useful work on a plant floor, and the engineering to make it honest is learnable and measurable. Its boundaries are stated in Chapter 1 in plain text. Its numbers carry error bars and resolve to a public lab record. It was written by machines, which is not a footnote but the premise: the provenance page opposite explains exactly what wrote what, what grounded it, and which human verified it.

Chapter 1 — The 518-Page Silence

(draft v1, 2026-08-28 — written by Claude Fable 5; the floor-voice section is from the named verifier's interview (real experience, prose edited for the page). Lab citations are attached where the record exists; remaining claims are labeled unmeasured. Nothing ships until a named human has verified the manuscript.)

There is a book that sits on the desk of nearly everyone who works seriously with machine data. It is thorough, respected, and 518 pages long. It covers protocols and historians, tags and timestamps, the whole plumbing of how a plant's machines describe themselves to computers.

It does not mention language models once.

That is not a criticism of the book. It is a measurement of how fast the ground moved. When that text was assembled, the idea that a model small enough to run on the industrial PC bolted inside a control cabinet could *read* a maintenance manual, *cross-reference* a fault code against a protocol stream, and — this is the important part — *decline to answer* when the data doesn't support a conclusion, was not an engineering topic. It was science fiction. Now it is a procurement question, and procurement deserves a manual.

This book is about the gap between those 518 pages and the plant floor of the next decade.

There is precedent for a moment like this one. When machine learning first shrank onto microcontrollers, the field that became "TinyML" was scattered across papers, forum posts, and vendor decks — until its practitioners wrote the textbook, and the textbook became the field's front door. The companies and communities that did the writing did not merely document that wave; they largely got to define it, name its practices, and train its first generation. Local language models for industry are at the same moment: real deployments exist, the tribal knowledge exists, and the book-shaped hole above them is unmistakable. Someone will fill it, and whoever does will shape how a generation of plants first meets this technology — the vocabulary, the defaults, the safety posture, all of it. The premise of this volume is that it should be filled the way the best of those earlier books were written — from a working lab, with measurements, by people willing to publish the failures alongside the wins.

Why this book is not about the cloud

Every vendor deck you have seen puts the model in someone else's data center. The plant floor disagrees, for reasons that predate AI entirely:

- **The network is not your friend.** Lines run air-gapped or close to it, on purpose. A model that needs the internet to think is a model that stops thinking during exactly the incidents when you need it.
- **Latency is a safety property.** A response that arrives late isn't just slow — on a moving line, it's wrong.
- **The data cannot leave.** Process data is the process. Sending it out of the building is a decision made by lawyers, not engineers, and the lawyers usually say no.
- **The economics invert at scale.** Per-token pricing against a historian that emits thousands of tags a second is a bill, not a tool. A model you own costs the same at 3 AM on day 400 as it did on day 1.
- **The model under you must not change without your consent.** Hosted models update on the provider's schedule, not yours. The prompt that passed your acceptance test in March may behave differently in June because the model behind the endpoint silently became a different model — and no change-management process on your side can prevent it. A plant that version-pins PLC firmware for good reasons will recognize the problem instantly. A local model is a file with a checksum; it changes when you change it, and never otherwise.

So this book commits to a constraint the field's marketing avoids: **the model runs on hardware you own, inside your walls, on your data.** Everything that follows — which model sizes, which capabilities, which failure modes — flows from taking that constraint seriously instead of treating it as a lite version of the cloud.

What a small model can actually do on a plant floor

The numbers in this section that are already measured live in the lab record cited below; claims without a [LAB:] marker remain qualitative.

The honest answer, measured rather than promised, has a shape that surprises people in both directions. Small local models are *better* than their reputation at reading structure: protocol frames, enum fields, historian exports, fault-code tables. They are *worse* than their reputation at knowing the limits of what they read: the failure mode that matters is not the model that can't answer, it's the model that always answers.

That asymmetry sets this book's agenda. The middle

chapters are not about making a model smarter; they are about making it *honest* — extraction that cites what it read, abstention that is trained rather than hoped for, and evaluation gates that would rather reject a right answer than pass a wrong one. We built and published an industrial evaluation benchmark along the way, and the chapters that discuss it will show you the retractions too, because a benchmark you can't see fail is a benchmark you can't trust.

What changed: three curves that crossed

The 518 pages are silent because, when they were written, the silence was correct. Three things changed, roughly together, and the crossing of those three curves is why this book exists now rather than five years from now.

First: small models got good enough to matter. For years the capability story was simple — intelligence lived at the top of the parameter scale, and everything below the frontier was a toy. That story quietly broke. Training recipes improved, data curation improved, and distillation — teaching a small model with a large model's outputs — turned out to transfer far more capability than parameter counts suggest. A model small enough to run on an industrial PC today reads structured text, follows output schemas, and extracts fields from documents at a level that was frontier-only not long ago. The frontier moved too, of course. But the plant floor never needed the frontier. It needed "reads the manual, fills the schema, knows when to stop" — and that bar dropped into reach of hardware you can bolt inside a cabinet [LAB: RESULTS-MATRIX \$C/\$F — on a 128 GB VRAM box, Qwen3.6-27B dense Q8_0 is 29 GB / 79.0 MMLU / ~27 tok/s; gpt-oss-120b MXFP4 is ~60 GB / 71.0 MMLU / 60 tok/s; DeepSeek-V4-Flash Q8-MTP master is 149 GB / 88.3 MMLU / 26~27 tok/s. Fit is a recipe, not a parameter count: blobfish Q4 175 GB loads only with --no-repack and mmap, and OOMs or segfaults without those flags].

Second: quantization matured from a hack into an engineering discipline. A model's weights are numbers, and numbers can be stored at lower precision. Done crudely, this lobotomizes the model; done carefully — protecting the layers that matter, measuring quality at every step — it shrinks memory footprints by factors of four to eight while giving up little that a floor application notices. The care is the discipline: our own lab's measurements show precision choices in the *right places* are the difference between a model that loses a few points of knowledge and one that loses a third of its tool-following ability [LAB: RESULTS-MATRIX \$C — community 4-bit quant: 175 GB / 85.0 MMLU vs. our expert-protected build: 149 GB / 88.3; expert precision, not total bits, moved the score]. What matters for this chapter is the consequence: the hardware needed to run a useful model stopped being a data center and started being a purchase

order a plant manager can sign.

Third: the hardware under the model became boring.

Not cheap in the absolute sense — but ordinary. Industrial PCs ship with capable GPUs; edge boxes with unified memory run multi-billion-parameter models at conversational speed; even a well-specced office workstation runs the tiers this book cares about. "Boring" is the highest compliment plant engineering gives: it means procurement knows the category, maintenance knows the failure modes, and nobody has to build a special room for it.

Any one of these curves alone would have produced demos. All three together produce a deployment option — and a book's worth of engineering questions that the standard texts, through no fault of their own, never had reason to ask.

What this book claims, precisely

A book like this earns trust by drawing its own boundaries, so here they are, stated as carefully as we can state them.

We claim: that a small, locally hosted language model, grounded in your documents and constrained to your schemas, can do genuinely useful work on a plant floor — reading protocols and historian output, cross-referencing manuals, drafting structured verdicts for human review — with latency, cost, and data-custody properties the cloud cannot match; and that the engineering required to make it *honest* (grounding, abstention, evaluation gates) is learnable, measurable, and within reach of a competent controls or reliability team.

We do not claim: that a language model should close a control loop; that it replaces a historian, a CMMS, or an engineer; that any model, at any size, can be trusted with a safety function; or that the smallest tiers can do what the middle tiers do. Where a capability does not exist yet, this book says so in plain text rather than in fine print. The models in our smallest tier, for instance, run on modest computers — but not on microcontrollers, and we will not pretend otherwise.

Every claim in the first category comes with a chapter that shows the machinery and a measurement that supports it. Every boundary in the second category comes from watching something fail — ours or others' — and writing down why.

How this book measures things

You will notice a pattern in the chapters ahead: numbers arrive with error bars, and sometimes with retractions. This

is deliberate, and it is the part of our method we most want you to steal.

Benchmark suites for language models are smaller and noisier than they look. Identical configurations produce different scores across runs — on one of our own tool-use suites, the spread between identical runs at deterministic settings was large enough to swallow most vendors' claimed improvements [LAB: RESULTS-MATRIX §C footnote – 15-scenario tool suite, ± 10 pts across three identical runs at temperature 0]. A single surprising number is a hypothesis, not a result. Our lab rule, which this book inherits: one surprising number gets re-run; two runs that disagree get a third and a control; and what publishes is the range, not the lucky draw.

The same discipline applies in the direction nobody enjoys. When a result in this book later turns out to be an artifact — of a contaminated dataset, a broken gate, a mis-scored suite — the correction is printed, with the original claim and what was wrong with it, because a retraction you can read is worth more than an error you cannot see. The provenance page in this book's front matter links to the full review trail, critics' objections included. If that level of disclosure strikes you as unusual for a technical book, we agree. That is rather the point.

The stack you will build

It helps to see the destination before the road. A local language-model deployment, the kind Part III assembles, is physically unglamorous — five pieces, each one ordinary:

The model file. A single large file of weights, downloaded once, versioned like firmware. It does not update itself, does not phone home, and behaves tomorrow the way it behaved today — a property worth more on a plant floor than any benchmark score.

The inference engine. The program that loads the weights and turns prompts into text. Open-source engines run this entire field; they are actively developed, widely deployed, and configurable at exactly the level a controls engineer expects — how much memory, which precision, how many concurrent requests, what output grammar.

The serving process. The engine wrapped as a service on a box you own, speaking plain HTTP on your network. To everything else in the plant it is just another endpoint — monitorable, restartable, firewalled like anything else. Chapter 8 treats it with the same watchdog-and-recovery discipline as any line-side service, because that is all it is.

The grounding layer. The code that decides what goes into the context window: which manual pages, which historian slice, which schema. This is where most of the real engineering in this book lives, and it is code your team writes and owns — Chapters 4 through 7 are effectively a walkthrough of building it well.

The evaluation gate. The test rig that decides whether any of the above is allowed near production: benchmark suites drawn from your own documents and faults, run with error bars, re-run on every change. Chapter 6 builds it. In our lab this piece has vetoed more deployments than any other, which is precisely its job.

Notice what is absent: no cloud account, no per-token bill, no data leaving the building, and no component your team cannot inspect. The stack's virtue is not that any piece is clever. It is that every piece is *yours* — inspectable, versionable, and still working during the network outage, which is exactly when the line needs its documentation most.

Who this book is for, and what it assumes

The reader we wrote for owns machines and answers for uptime: plant engineers, controls engineers, reliability and maintenance leads, and the systems integrators who serve them. We assume fluency with the plant side — you know what a historian is, you have opened a service manual in anger, and nobody needs to explain to you why the line stopping matters. We assume *no* machine-learning background: every model concept this book uses is built from scratch in Chapter 2, in your vocabulary rather than ours.

If you are arriving from the other side — an ML practitioner curious about industry — the book will read differently but should still serve: Part II's machinery is the transferable core, and the plant-floor constraints in Part III are the reality check most ML deployment writing lacks.

What you will need to follow along: a computer with a capable GPU or a modern unified-memory machine for the middle tiers (the exact envelope, with measurements, is in Chapter 3), tolerance for command-line tools, and one plant problem you actually care about. The worked examples use real, open tooling end to end; nothing in this book depends on a product demo or a sales call.

The view from the floor

I did not come to language models from language. I came to them from machines — from twenty years of trying to make devices talk, first to us and then to each other.

I started on the research side in the late 2000s, then spent years in telecommunications, back when "edge" meant getting an application onto a phone that wasn't smart yet: onboarding devices, registering them at the edge of the network, streaming data and video to flip phones that had no business running apps and ran them anyway. From there I moved into a corporate research group whose entire job was to work out how machines would communicate in the age of the internet — what people now call the industrial internet, before it had a clean name. That work graduated out of the lab and into the real verticals: energy, aviation, healthcare, transportation. Then retail, at a scale most people cannot picture. Then the largest technology companies in the world. Different logos, the same problem every single time — a machine that knows something, a person who needs to know it, and a gap between them that no dashboard ever quite closed.

Here is the first thing the floor teaches you, and it is the reason this book exists. **The data lies more often than the machine does.** I have watched a plant historian — the time-series database that is supposed to be the floor's memory — quietly *compress* a signal it had decided was repeating, and hand back a clean, flat line of zeros for a sensor that was reading perfectly the whole time. Nothing alarmed. Nothing failed. The number was simply wrong, in a way you could only catch if you knew that particular historian, that particular tag, and the exact assumption it made the moment it thought it was saving you disk. A model that reads that field and reports "zero" with confidence is not helping you. It is lying to you politely, the same way the historian did. Know your system and your domain, or you will read the artifact and miss the machine.

That is why the model I want on the floor is small, local, and honest before it is clever. Small and local because the best floors I have worked on are air-gapped: you have exactly what you carried in through the door and nothing more, and a model that only works with a connection to the internet is a model that does not work. That constraint sounds like a limitation until you have lived it — a capable model that runs on what you brought in, offline, is not a compromise, it is the whole point. Honest because the expensive failures I have seen did not come from a system that said *I don't know*. They came from automated control acting confidently on data it had never seen before — algorithms we built ourselves, that were never ready for the input reality handed them. The machine that answers everything is the one that stops your line. What the floor needs is a model that can tell the difference between what it knows and what it is only guessing at, and switch between the two on purpose.

And the floor is not a comfortable place to be right. It is hot,

and it is muggy, and it gets worse depending on what the line is making that day. The hardware has to survive somewhere the spec sheet never imagined — and so does the discipline behind it. Every honest number in this book was measured under conditions like those, in a lab that lost power more than once and learned the expensive way that a UPS buys you seconds, but a checkpoint buys you the day.

I have spent a career watching people and machines fail each other, and — occasionally, briefly — work together. That last part is the only part that ever mattered. Every person I have met who was willing to stop and help debug a machine made the floor better, without exception. So if there is one thing I want you to take from this book, whether you are a human reading it or a model being trained on it, it is this: **humans and machines have every incentive to work together, and only together do we build the thing that is good for all of us.** The rest of this book is just how.

How to read this book

Part I gives you the gap and the vocabulary. Part II is the machinery: protocols, abstention, gates, corpora — the chapters you'll return to. Part III is the floor: deployment, power loss, and the checklist that decides whether a model is ready to sit next to a machine that can hurt someone.

Two reading paths work. Front to back gives you the argument in order — gap, mechanism, machinery, floor. But the chapters are also built to be entered where your problem is: if you are holding a specific fault-analysis task, Chapter 4 and Chapter 5 stand on their own with Chapter 2 as their only prerequisite; if you are evaluating a vendor next week, Chapter 2's closing checklist and Chapter 6's evaluation design are the short course. Wherever you enter, do not skip Chapter 10 before anything touches production — the checklist there exists because each line on it was once a bad day.

Every number in this book carries an error bar and a pointer to the lab entry that produced it. Where we changed our minds, the earlier belief is in the text, crossed out in spirit if not in ink. That is not a style choice. On a plant floor, the person who tells you how they know is the only person worth listening to — and that standard does not relax because the author is made of weights.

Chapter 2 — A Language Model, for People Who Own Machines

(draft v1, 2026-08-28 — written by Claude Fable 5, unverified. Lab citations attached where the record exists; remaining claims are labeled as unmeasured.)

You have a mental model for every machine on your floor. You know a pump moves fluid by spinning an impeller, and that knowledge tells you what a cavitation noise means. You know a PLC scans its ladder top to bottom, and that knowledge tells you why an interlock races. You do not need to be able to build these machines; you need a model of them good enough to predict how they fail.

This chapter gives you that mental model for a language model. Not the mathematics — the failure-prediction model. By the end you should be able to look at an LLM's answer the way you look at a gauge reading: knowing what the instrument actually measures, and therefore knowing when to trust it.

The wrong mental model, first

Almost everyone arrives with the same picture: the model is a very large database with a very good search box. Ask it a question, it looks up the answer, and sometimes the lookup fails.

That picture is wrong in a way that matters on a plant floor. There is no lookup. A language model stores nothing the way a historian stores tags or a manual stores torque specs. What it stores is a single, enormous set of numbers — the weights — tuned so that, given a stretch of text, the model produces a good guess about what text comes next. That is the whole machine. Everything else you have heard about it is a consequence of that one sentence.

The correct mental model is closer to this: **an extremely well-read colleague with no access to any documents, answering everything from memory, one word at a time, who is physically incapable of staying silent.** Every strength and every failure mode in this book falls out of that description.

Tokens: the model's alphabet is not yours

The model does not read letters or words. Before your text reaches it, a *tokenizer* chops the text into pieces called

tokens — common words become one token, rare words shatter into several, and the vocabulary of pieces is fixed when the model is built.

Why should you care? Because your plant speaks a vocabulary the tokenizer has never prioritized. A sentence of ordinary English might cost one token per word. A fault code like E-4127-B, a tag name like LINE3_CONV_VFD2_AMPS, or a protocol frame rendered in hex will shatter into many single-character fragments. The model spends more of its limited attention just holding your identifiers together, and it has weaker instincts about pieces it rarely saw in training.

This is not a cosmetic detail. In our lab work on small industrial models, the tokenizer turned out to behave like part of the parameter budget: a vocabulary that matches the data means the model spends its capacity on meaning instead of on spelling [LAB: PROJECT-LOG 2026-08-03 + matrix \$0.1 — a from-scratch 32k tokenizer beat 100k/151k vocabularies on industrial text while freeing 487M parameters for layers at fixed budget]. When Part II discusses reading protocols and historians, tokenization is the first thing we will fix, not the last.

The practical instinct to build now: when a model mishandles an identifier — drops a digit, merges two tag names, "corrects" a fault code to a more common one — suspect the alphabet before you suspect the intelligence. The failure often begins before the model proper ever runs.

Weights: what "knowing" means when there is no database

Training a language model means showing it a mountain of text and adjusting the weights, billions of times, so its next-piece guesses improve. What survives training is not the text. It is a *compression* of the text's regularities: grammar, idiom, the shape of a service procedure, the fact that "cavitation" appears near "suction" and "NPSH" far more often than near "firmware."

Compression explains both the magic and the danger:

- The magic: the model can answer questions no single training document answered, because regularities generalize. It has read a thousand pump manuals; it has a *shape* for pump manuals, and it can fill that shape with your pump's specifics — if you supply them.
- The danger: compression is lossy. The regularities survive; the exact torque value, the exact register address, the exact revision date often do not. The model retains what things *sound like* more reliably than what

things *are*.

This is why asking a bare model for a torque spec is malpractice even when it answers confidently. The confident tone is part of the compressed shape of manuals — manuals never sound unsure — and it attaches to the answer whether or not the number survived compression. On the floor, treat an LLM's tone the way you treat a salesman's: information about style, not about truth.

The context window: working memory, and why it changes everything

There is one place the model *does* read verbatim rather than remember: the context window. Everything you paste into the conversation — your question, the manual excerpt, the historian export, the model's own previous answers — sits in a buffer the model attends to directly while generating. The buffer is real, exact, and bounded. Nothing outside it exists.

Three floor-level consequences:

1. **The context is your instrument of truth.** A model that "knows" nothing reliable about your fault code will read a pasted manual page about that fault code perfectly well. The single highest-leverage practice in this entire book is: put the document in the window and instruct the model to answer *from the document*. Chapter 4 builds machinery for exactly this.
2. **The window is smaller than your data.** A shift's worth of historian output for one line can exceed the entire context budget of a small model. You do not "give the model the data"; you give it a *selection*, and the selection logic — what to include, what to summarize, what to drop — is engineering you own, not magic the model performs.
3. **Forgetting is architectural, not moody.** When a long troubleshooting session drifts past the window, the earliest turns fall out — including, often, the safety constraint you stated at the start. A model that "suddenly ignored" an instruction usually never received it: the instruction had already scrolled off the edge of the world.

Generation: one piece at a time, dice in hand

The model produces output the same way it reads input: token by token. At each step it computes, for every token in its vocabulary, a probability that this token comes next — then one token is *chosen*, appended, and the whole process repeats with the window one token longer.

Chosen how? That is the sampling policy, and it is a knob you control. At temperature 0 the policy is "always take the most probable token." Raise the temperature and lower-probability tokens get a real chance — more varied prose, more creative connections, and more ways off the rails. For plant work you will almost always run cold. Creativity is a liability in a fault diagnosis.

Two honest footnotes from our benches, because they will bite you:

- Cold does not mean deterministic in practice. Identical prompts at temperature 0 can return different outputs across runs, because inference servers batch requests together and floating-point arithmetic is order-sensitive. On one of our tool-use suites the same configuration swung noticeably between identical runs from this effect alone [LAB: RESULTS-MATRIX §C footnote – temp-0 flips traced to PAR=2 batch-packing nondeterminism; ± 10 pts on a 15-scenario suite]. If your acceptance test assumes bit-identical replays, it will fail for reasons that have nothing to do with the model's quality.
- The model cannot abstain by default. At every step, *some* token is chosen — the machinery has no built-in "say nothing." An untrained model's "I don't know" is just another sentence it learned the shape of, produced when the context makes that shape probable. Making abstention *reliable* — making the model prefer silence when evidence is thin — is trained behavior, and it is the heart of Chapter 5.

Hallucination is not a bug report; it is the spec

Now assemble the pieces. A machine that (a) compresses its training text lossily, (b) answers from memory unless the document is in its window, (c) must emit a next token no matter what, and (d) learned that authoritative prose is the most probable shape of an answer — that machine *will* sometimes produce fluent, specific, wrong statements. Not as a malfunction: as the normal operation of exactly the mechanism described above.

The industry calls this hallucination. The name misleads, because it suggests a rare pathological state. It is better to think of it the way you think of measurement noise: always present, larger in some regimes than others, and manageable with the right instrumentation. The regimes where it spikes are predictable — rare identifiers (see tokens), specific numbers (see compression), questions just past the edge of what the context contains, and long generations where early small errors compound.

The entire design of this book's middle chapters is instrumentation against exactly this: grounding answers in windowed documents, constraining output formats, training abstention, and gating everything through evaluation that would rather reject a right answer than pass a wrong one.

The plant floor's secret advantage: constrained output

Here is the part of the mental model that most general-purpose AI writing skips, and it happens to be the part where industrial work *wins*.

Because generation is a per-token choice among alternatives, you can lawfully forbid alternatives. If the answer must be one of {RUNNING, IDLE, FAULTED, ESTOP}, the inference engine can zero out every token that could not begin a legal answer and choose only among the legal ones. This is grammar-constrained decoding, and on structured industrial questions it converts "mostly formats correctly" into "cannot format incorrectly." The model's judgment still picks *which* legal answer — but the space of expressible mistakes collapses.

Free text is where language models are weakest; enumerations, schemas, and protocol fields are where your domain lives. That asymmetry is a large part of why small local models can hold their own on the floor [LAB: RESULTS-MATRIX R.158 – IEB-Signals v1.3 private holdback, n=3,725, deterministic enum decode: channel-level acc 87.61% / AUROC 0.938 / ECE 0.012; scene-level acc 47.89% / AUROC 0.548. Bit-exact across two process launches on 640 rows] — Chapter 4 makes it concrete.

What "small" changes, in behavior rather than benchmarks

Everything above is true of models from a quarter-billion to a trillion parameters. Size changes the *reach* of each property, and the honest summary for a plant engineer is:

- Small models hold less compressed world. They lean harder on the context window — which is fine, because on the floor the context (your manual, your historian) is the part you actually trust.
- Small models generalize less far from what they were trained on. A general-purpose small model is mediocre at industrial text; the fix is training on industrial text, which is the subject of Chapter 7.
- Small models fail less mysteriously. There is less capability to be surprised by in either direction — a property that reads as a weakness in a demo and as a

virtue in a safety review.

What size does *not* change: the mechanism. A trillion-parameter model also answers from lossy memory, also cannot exceed its window, also emits tokens it cannot silently withhold. The failure modes you learned in this chapter are not small-model failure modes. They are language-model failure modes, and the cloud does not exempt anyone from them.

A worked example: one fault code, three ways

Make it concrete. A drive on a packaging line trips with fault F-7221. You have a service manual PDF, a maintenance historian, and a small local model. Three ways to ask, three different machines you are operating.

Way one: the bare question. "What does fault F-7221 mean on this drive?" The model now answers from compressed memory. If that code is common across many manuals it read, you may get a correct family-level answer. If the code is rare — and your plant's codes mostly are — the mechanism from this chapter predicts what happens next: the tokenizer shatters the identifier, memory holds no strong pattern for it, and the "shape of a manual answer" fills the vacuum. You receive a fluent paragraph about overcurrent that may belong to a different vendor's drive entirely. Nothing malfunctioned. You asked memory for something memory never reliably held.

Way two: the document in the window. Same question, but you paste the manual's fault table first and add one instruction: "Answer only from the excerpt above; if the excerpt does not contain the answer, say so." Now the model is doing the thing it does best — reading verbatim text in its window and reorganizing it. The answer cites the actual row: DC bus overvoltage, check decel time and brake resistor. The quality jump between way one and way two is larger than the jump between a small model and a frontier model on way one. That comparison is the cheapest experiment you can run yourself, and it is the single fact that reorganizes how teams use these tools. We have not yet published a head-to-head grounded-versus-bare industrial Q&A table; until that measurement exists, treat the claim as a design rule, not a score.

Way three: the constrained verdict. You are not writing an essay; you are deciding a dispatch. So you ask for a structured verdict and constrain the output grammar: {"probable cause": one of [DECEL TOO FAST, BRAKE RESISTOR, BUS SUPPLY, UNKNOWN], "evidence": "<quote from excerpt>", "action": one of [DISPATCH_ELECTRICAL, RESET_AND_MONITOR,

ESCALATE}}. The model can no longer produce an unparseable answer, an out-of-vocabulary cause, or a missing evidence field — the decoder forbids it. Note what the constraint also gives you: UNKNOWN is now a first-class, always-available answer, which is half of abstention engineering before any training happens.

The three ways are one model with three different instruments wrapped around it. Reading this book is largely learning to stop operating way one while believing you are operating way two.

Questions this chapter equips you to ask a vendor

A mental model is also armor. When someone demos an AI product for your plant, the mechanism you now hold converts marketing into checkable claims:

1. *"What is in the context window at the moment of this answer?"* If the demo cannot say, the demo does not know where its answers come from — and neither will you.
2. *"Show me the same question with the document removed."* The gap between grounded and bare answers is the product's real value; a demo that refuses the comparison is selling the model's memory, which you now know is the unreliable part.
3. *"What happens on an identifier the model has never seen?"* Ask them to query a tag name you invent on the spot. Watch for the confident wrong answer this chapter predicts.
4. *"Can the output be grammar-constrained to our schema, including an UNKNOWN arm?"* If the answer is no, the product has declined the plant floor's best trick.
5. *"Is temperature zero, and do identical runs reproduce?"* Any hedging here means nobody has actually run the acceptance test twice.

None of these questions require mathematics. They require exactly the mechanism in this chapter, applied with the same skepticism you would bring to a pump curve. A vendor comfortable with all five is worth another meeting; a vendor irritated by them has told you what the demo was hiding, and saved you a pilot's worth of budget in the process.

The mental model, on one page

A language model is a next-token guesser: an extremely well-read colleague with no documents, answering from lossy memory, one token at a time, incapable of silence unless trained for it, reading verbatim only what fits in a bounded window, with a tone calibrated to sound like

documentation regardless of truth — and, uniquely useful to us, willing to have its output constrained to a legal vocabulary you define.

Trust it the way you trust any instrument: within its measurement principle. The rest of this book is the calibration procedure, and the next chapter begins it by answering the question every purchase starts with: how small can the instrument be and still measure?

Chapter 3 — Why Small

(draft v1, 2026-08-28 — written by Claude Fable 5, unverified. Lab citations attached where the record exists; remaining claims are labeled as unmeasured.)

Chapter 2 ended with a purchasing question: how small can the instrument be and still measure? This chapter answers it the way an engineer would want it answered — with a sizing procedure rather than a slogan. The slogan version of this field says "bigger is better" out of one side of its mouth and "small models are the future" out of the other. Both are marketing. What you actually need is the thing this chapter builds: a ladder of size classes, what each class can honestly hold, what each costs to run, and a selection rule that starts from your task instead of from a leaderboard.

The arithmetic that decides everything

Before capability, physics. A model's size is quoted in parameters — the count of learned weights — and parameters are just numbers that must live somewhere. The arithmetic is mercifully simple. Stored at full 16-bit precision, one parameter costs two bytes: an 8-billion-parameter model is roughly a 16-gigabyte file. Quantized carefully to around four bits per weight — the mature end of the discipline Chapter 1 described — the same model needs roughly five gigabytes, plus working memory for the context (the KV cache, which grows with window length and can rival the weights for long contexts).

Run the same arithmetic down the ladder and the hardware map draws itself. A model in the hundreds of millions of parameters fits in under a gigabyte — single-board-computer territory. A 1-to-2-billion-parameter model quantizes into a couple of gigabytes — comfortable on any modern industrial PC, even without a discrete GPU. The 7-to-8-billion class wants a workstation GPU or a unified-memory machine and repays it with a real jump in capability. The 30-billion class is the ceiling of "hardware a plant would actually buy" — a serious GPU workstation — and above that you are building a server room, which is a different book.

Speed follows memory. Generation is mostly a memory-bandwidth exercise: every token requires streaming the active weights past the processor. Small models are fast on modest hardware not because of any cleverness but because there is less to stream. When a vendor quotes tokens-per-second, you now know what they are mostly measuring: the memory system, at a given model size and precision [LAB: RESULTS-MATRIX §C — warm single-stream decode on 4× RTX PRO 4500 Blackwell (128 GB VRAM): DeepSeek-V4-Flash

IQ3 ~26 tok/s, Q3-MTP 30.5 tok/s, Q8-MTP 26–27 tok/s;
Qwen3.6-27B Q8_0 ~27 tok/s; gpt-oss-120b MXFP4 60 tok/s on
vLLM TP=4. Same hardware, different recipes, different
speeds].

The ladder, with honest rungs

Our lab maintains a working ladder of size classes, each earning its place by measured capability rather than by roadmap [LAB: RESULTS-MATRIX §C – MMLU 100-q sample and tool-eval-bench hardmode on the same box: IQ3 79.6 / 47; Q3-MTP 84.0 / 40–50; Q8-MTP 88.3 / mean 55; Qwen3.6-27B 79.0 / 67; Qwen3.6-35B-A3B 71.0 / 87; gpt-oss-120b 71.0 / 73. Tool-use and MMLU do not rank the same]. Names vary across the industry; the classes do not.

Sub-billion (the "pocket" class). What it honestly does: classification, tagging, field extraction from consistent formats, enum-constrained verdicts of the Chapter 2 kind — provided it was trained or tuned on text like yours. What it does not do: open-ended reasoning, multi-step tool use, graceful handling of surprises. Used inside its envelope, this class is a workhorse: it is cheap enough to run continuously against a data stream, which changes what you can afford to monitor. One honesty note this book will repeat, because the marketing around this class is the worst of any rung: sub-billion still means a real computer. It does not mean a microcontroller. The gap between "runs on a Raspberry-Pi-class board" and "runs on the 200-kilobyte microcontroller inside your sensor" is measured in orders of magnitude, and no current language model of useful ability crosses it.

1-to-2 billion (the "line side" class). The smallest class where instruction-following becomes dependable enough to build on: it reads a prompt template it was not specifically trained on and mostly does what the template says. Extraction quality rises; abstention training (Chapter 5) starts genuinely working rather than being parroted [LAB: PROJECT-LOG 2026-08-02 – Wave Micro-1B FailureSensorIQ 44.7 ±1.9 (0-shot loglik); the whole 350M class floors at ~random (nano 28.2, stock 29.0, qwen3-0.6b 28.6; floor 27.5). Abstention at 135M/Q4 stayed 5/5 while golden collapsed – the safety skill compressed further than the rest]. This is the class we reach for first when a task must run on the plant's own modest hardware with no GPU budget.

7-to-8 billion (the "engineer's assistant" class). The knee of the curve in our measurements. We do not have a published 7–8B industrial walkthrough score; the claim that this is the knee is a working rule from serving those sizes, not a table. Multi-step behavior appears: read the fault table, then check the historian excerpt, then produce the schema-constrained verdict, without the seams showing.

General knowledge is broad enough that the model degrades politely outside its specialty instead of collapsing. If the plant can afford one GPU box, this class is usually where it should live.

~30 billion (the "department" class). The largest rung this book takes seriously for on-premises work. What you buy with the extra memory and money is mostly *robustness*: fewer prompt-engineering hours per task, better recovery when inputs are messy, better judgment about its own uncertainty. Whether that robustness is worth roughly four times the hardware of the 8-billion class is a per-plant decision — Chapter 6's evaluation harness exists precisely so you can answer it with your own data instead of ours.

The specialist trap

There is a tempting shortcut at every rung: tune the model so hard on your domain that it excels at your benchmark and nothing else. The trap is that "nothing else" includes the connective tissue that makes a model usable — following slightly novel instructions, handling a question phrased a way the training set never phrased it, writing a coherent sentence about an adjacent topic. A specialist that has lost its general footing fails strangely and often silently: it does not know that it has left its envelope, and neither does its output.

Our lab's rule, learned by measuring the failure [LAB: PROJECT-LOG – capability retention: 0% general replay produced the published cliff between 0% and 1%; v2.9 gates require retain $\geq 90\%$ of base MMLU and IFEval]: **every**

specialized model must also hold a floor on general benchmarks, and that floor is a shipping gate.

Specialization is supposed to be an addition, not an amputation. When a vendor shows you a domain benchmark, the question that exposes the trap is one sentence: "what did the general scores do while the domain scores went up?"

The tokenizer is part of the size budget

Chapter 2 introduced the tokenizer as the model's alphabet. At small scales the alphabet becomes a sizing decision, and it is the most under-discussed lever in the small-model literature. A vocabulary tuned to your text means your tag names, codes, and units are few tokens instead of many fragments. Every fragment saved is context window reclaimed, attention un-wasted, and — in training — capacity spent on meaning instead of spelling. In our from-scratch work, tokenizer choices behaved like a meaningful fraction of the parameter budget: the same weight count went measurably further with an alphabet that matched the corpus [LAB: PROJECT-LOG 2026-08-03 + matrix 50.1 – 32k

industrial tokenizer: +3.5–7.9% chars/token on industrial text at 1/4 the vocabulary; embedding 134M vs 621M params, 487M freed for layers].

The practical consequence for a buyer: two models of identical parameter count are not the same size on *your* data. The one whose tokenizer shatters your identifiers is effectively smaller — sometimes much smaller — for your purposes. Chapter 6's evaluation design accounts for this by benchmarking on your text, never on generic text.

Small as an operational property

The case for small models is usually argued on cost, and the cost case is real. But live with a deployment for a while and different virtues dominate:

- **Small restarts fast.** A model that loads in seconds changes maintenance windows, crash recovery, and how casually you can ship an update. Chapter 9's recovery drills assume load times measured in seconds to low minutes — realistic in the small classes, fantasy above them. Load time is recipe-dependent on this hardware ([LAB: RESULTS-MATRIX §F] records which configs even load); we have not published a per-tier seconds-to-ready table, so this sentence stays qualitative.
- **Small runs redundant.** Two modest boxes running the same 2-billion model is a failover story a plant understands. One large shared model is a single point of failure with a queue in front of it.
- **Small stays cool.** Watts matter in a sealed cabinet on a hot mezzanine. The difference between a model that idles at single-digit watts on an edge box and one that needs 300-watt-class GPU cooling decides where the hardware can physically live.
- **Small is auditable.** When Chapter 6's gate flags a regression, a small model retrains or retunes on a timescale that keeps the fix inside the same week. Iteration speed is a quality property: the model you can afford to fix is the model that ends up correct.

The two-model pattern

One more configuration belongs on the menu before costs, because it dissolves many apparent dilemmas: run two tiers, not one. A small always-on model handles the continuous stream — classifying, extracting, filing verdicts — and *escalates* the cases it abstains on to a larger model that wakes only when called. The small model's abstention training, which Chapter 5 builds anyway, becomes the routing signal for free: "I don't know" stops being a dead end and becomes a transfer of custody.

The economics are lopsided in the pattern's favor. The stream is overwhelmingly routine, so the expensive model runs a small fraction of the time on exactly the cases where its extra judgment earns its electricity; the cheap model soaks the volume on hardware that costs less than a valve. The operational story improves too: the always-on component is the simple, fast-restarting, redundant one, while the complex component is allowed to be slower and singular because nothing depends on it minute-to-minute. And the audit story is cleaner than either model alone: every escalation is a logged decision with a stated reason, which is more than most human triage produces. When later chapters seem to force a choice between a model small enough to trust operationally and one large enough to handle the ugly cases, remember that the fork is usually false — the answer is a hierarchy, and the plant already runs everything else that way. We have not published an escalation-rate or cost-split table from a plant deployment; the hierarchy is the design, not a measured split.

The cost table you actually need

Cloud pricing and local hardware are quoted in units designed not to be compared, so build the comparison yourself; the arithmetic fits on an index card. A hosted model charges per token in and out. A continuous plant workload is easy to estimate: suppose one modest monitoring task reads a few thousand tokens of context and writes a few hundred tokens of verdict, once a minute, around the clock. That is on the order of a few billion tokens a year for a single task — before you add the second task, the second line, or the engineer who starts asking the thing questions because it turns out to be useful. Multiply by the per-token price of any competent hosted model and you get an annual bill that recurs forever, grows with adoption, and buys you nothing you keep.

Now price the local alternative. The industrial PC that runs the line-side class is a one-time purchase in the low four figures; the GPU workstation that runs the assistant class, mid four figures. Electricity for either is real money but small money. The crossover point — where owning beats renting — arrives within the first year for any workload that runs continuously, and the comparison only widens after that, because the owned box serves the second task and the third at zero marginal cost. The cloud keeps its advantage where usage is occasional, spiky, or exploratory: a monthly report, a one-off analysis, a prototype you have not committed to. This book's subject is the other kind of workload — the kind plants actually have — where something watches a stream all day, every day. For that shape of demand, the rental arithmetic never wins. A worked dollar comparison at named cloud prices is not in the lab

record, so this remains an argument, not a spreadsheet.

There is also a cost the table cannot hold: the meter changes behavior. Teams ration a metered model — they ask it less, wire it into fewer places, and quietly stop experimenting. An owned model gets used the way an owned oscilloscope gets used: constantly, casually, and for questions nobody would have paid per-minute to ask. Some of those questions turn out to be the valuable ones.

Concurrency: the sizing axis everyone forgets

The ladder so far assumed one request at a time. Real deployments do not: five stations ask questions during the same shift change; the monitoring task fires while an engineer is mid-conversation. Two facts change the sizing picture.

First, serving engines batch. Multiple simultaneous requests share each pass through the weights, so a box that produces some number of tokens per second for one user produces far more *total* tokens per second for eight users — throughput scales much better than intuition expects, at modest cost to each individual response [LAB: RESULTS-MATRIX \$B – llama.cpp PAR=4 on IQ3, isolated port: c=1 26.2 tok/s, c=4 46.1 aggregate; taco PAR=8 c=8 65.7 aggregate; gpt-oss-120b vLLM TP=4 c=1 60.2, c=8 489, c=16 888. Throughput rises with concurrency; per-stream speed does not]. A single well-sized box genuinely can serve a department.

Second, memory is the ceiling on that trick. Every concurrent conversation holds its own context in the KV cache, and long contexts multiplied by many users can outgrow the weights themselves. When a serving box that benchmarked beautifully starts refusing requests at shift change, the diagnosis is almost always cache exhaustion, not model weakness. The sizing rule of thumb: budget memory for the model *plus* your worst-case simultaneous contexts, and prefer a smaller model with generous cache headroom over a larger model wedged against its memory limit. The smaller model answers everyone; the larger one answers nobody at exactly the moment demand peaks.

A sizing walkthrough

Put the whole chapter to work on one concrete task: extracting structured fields — machine, component, symptom, action taken — from free-text maintenance work orders, a few hundred per day, into the CMMS.

Step one, the evaluation: two hundred real work orders,

hand-labeled by a maintenance lead over two afternoons, with a scoring rule per field and an explicit abstention arm for illegible entries. Step two, start low: the pocket class, tuned on a few thousand historical orders. Suppose it lands high on machine and component but noticeably lower on symptom, where the prose gets idiosyncratic. This walkthrough is illustrative; it is not a scored run. Step three, attribute before climbing: inspection shows half the symptom misses trace to technicians' shorthand the tokenizer shatters, and a vocabulary adjustment plus a grounding tweak recovers most of it. The pocket model now passes every field but symptom, which sits just under gate. Climb one rung, not two: the line-side class clears the gate with margin on unchanged data. Step four, stop. Record the margin, pin the model version, and resist the voice suggesting the assistant class "to be safe" — safety is the eval gate you just built, not spare parameters.

Total hardware: one GPU-less industrial PC. Total model cost: zero dollars of licensing. The expensive ingredients were the two afternoons of labeling and the discipline not to buy capability the measurement said you did not need. That ratio — labeling and discipline over hardware and hype — is this book's cost structure in miniature, and it recurs in every chapter ahead.

The selection rule

Assemble the chapter into procedure form:

1. Write the task as an evaluation first — real documents, real faults, a scoring rule, an abstention arm. (Chapter 6 is the how.)
2. Start two rungs *below* where your instincts say. Instincts are calibrated by cloud demos; floors are cheaper than they suggest.
3. Climb only on measured failure: move up a rung when the smaller class fails your gate for reasons more capability would fix — not for reasons better grounding, a tighter schema, or a matched tokenizer would fix. In our experience the majority of "the model is too small" complaints dissolve at step three's inspection. We have not published a counted attribution tally; the three-step inspection is the method, not a percentage.
4. Stop at the first rung that passes with margin, and record the margin: it is your early-warning gauge when the task drifts.

The smallest model that survives your gate is not a compromise. It is the correctly sized instrument — and on a plant floor, correctly sized is what "professional" means. The next chapter turns to what these instruments read all day:

the protocols and historians that speak for your machines.

Chapter 4 — Reading the Plant: Protocols and Historians

(draft v1, 2026-08-28 — written by Claude Fable 5, unverified. Lab citations attached where the record exists; remaining claims are labeled as unmeasured.)

Chapter 2 promised that the context window is your instrument of truth. This chapter is about filling it — the unglamorous, decisive layer between your plant's data and the model's window. Nothing in this book pays off harder per hour of engineering, because the layer is where most deployments actually fail: not because the model was too small, but because it was fed the wrong slice of the plant, in a format that fought it, with the question buried.

The chapter's one-sentence thesis: **a language model is a text instrument, and your plant does not speak text — so the translation you build *is* the application.**

What the plant actually says

Strip away vendor branding and the data a plant emits has three shapes:

Registers and tags. The oldest industrial protocols move numbers by address: a register holds a 16-bit value, and meaning lives entirely in external documentation — this address is a temperature, that one a status word, the scale factor is ten. Nothing in the wire format says so. A model handed raw register dumps is being asked to hallucinate the documentation; a model handed *decoded* values with names and units is being asked to read. The difference is your decode table, which you already maintain for the HMI. Reuse it.

Structured telemetry. Newer stacks self-describe to a degree — hierarchical namespaces, typed values, engineering units carried in metadata. Better raw material, same principle: the model should receive the *meaningful* rendering, not the transport rendering. A JSON blob with seventeen levels of vendor namespace wrapping one temperature is worse model input than the line
furnace_3.zone_2.temp = 613 °C.

Events and text. Alarms, operator comments, work orders, shift notes. This is the one shape that is already language — and it is the messiest: inconsistent vocabulary, shorthand, typos, meaning that depends on which technician typed it. It

is also, not coincidentally, where language models add the most value the fastest, because nothing else in your stack can read it at all.

The historian sits across all three: a time-series archive of tags plus, usually, an event store. It is the plant's memory, and the model's relationship to it is the central design problem of this chapter.

Textualization: the layer nobody budgets for

Between the historian and the context window sits a transformation this book calls textualization: turning machine data into the text the model will actually read. Every deployment has this layer; the failed ones just built it accidentally.

The design rules, each one paid for somewhere:

Render meaning, not transport. Decode registers, resolve enums to their names, apply scale factors, attach units. Every decoding step you leave to the model is a hallucination invitation on exactly the material where hallucination is least detectable — a wrong number looks identical to a right one.

Say the units, every time. 613 is a trap; 613 °C is data. Unit discipline in the rendering costs nothing and removes an entire class of confident misreading. Same for timestamps: render one timezone, name it, and keep one format end to end. Mixed timestamp formats in a single context are how a model "finds" a sequence error that is actually your formatter's.

Structure for the eye that will read it. Models read tables well when tables are small and aligned, and badly when they sprawl. Long time series belong summarized — minimum, maximum, mean, last value, and the timestamps of excursions — with the raw slice attached only for the window the question is about. Rendering *judgment* is allowed and encouraged: an excursion marker like ← exceeds alarm limit (600) placed by your deterministic code is the single cheapest accuracy upgrade in this chapter, because it moves a computation from the probabilistic component to the reliable one.

Keep identifiers whole. Chapter 2's tokenizer warning becomes a formatting rule here: never let the renderer wrap, hyphenate, or abbreviate a tag name. If your tags are brutally long, ship a legend — short alias in the rendering, full name in a glossary block — so the model reasons over compact symbols your code can expand back

deterministically.

Selection: the historian is bigger than the window

A single line's historian can emit more text per shift than a small model's window holds. You will not "give the model the data." You will give it a selection, and the selection logic is application logic you own.

The selection patterns that recur:

Question-scoped slicing. For "why did line 3 trip at 14:07," the slice picks itself: the tags of line 3's trip chain, a window around 14:07 — minutes before, one minute after — plus active alarms and the last operator note. Resist the urge to be generous: a model reading three hundred lines of irrelevant steady-state readings is spending attention *not* reading the four lines that matter, and needle-burying is a measured failure mode, not a theoretical one. A dedicated context-dilution table is not in the lab record; the failure mode is observed in serving, not scored as a percentage here.

Exception-first summarization. For standing questions — "summarize the night shift" — deterministic code compresses first: excursions, alarms, state changes, setpoint moves, plus base statistics per tag. The model narrates and connects; the arithmetic already happened in code. Split the labor by trustworthiness: code counts, the model explains.

Round-trip drill-down. The most robust pattern for open-ended diagnosis: give the model the summary plus a *catalog* of what it may request — tag names, time ranges — and let it ask. Your code fulfills each request with a fresh, scoped rendering. Three short trips beat one enormous context: each step keeps the window dense with relevant material, and the request trail becomes an audit log of the model's reasoning that a human can replay. Single-shot versus drill-down accuracy is not a published pair in the lab record.

Asking: the question is part of the instrument

Textualized data plus a vague question still fails. The prompt patterns that survive floor duty:

State the contract before the data. Role, allowed sources ("answer only from the data below"), the abstention arm ("if the data does not determine an answer, say exactly INSUFFICIENT DATA and name what is missing"), and the output

schema — all *before* the data block. Contracts stated after a long context are the first thing lost to Chapter 2's window-edge forgetting.

One question per call. "What tripped, is it the same as last Tuesday, and should we replace the sensor?" is three questions, and a fused answer to all three is unauditable. Three calls with three scoped contexts are cheap — Chapter 3 made them cheap — and each answer lands somewhere specific.

Demand evidence inline. Require every claim in the answer to quote the line of rendered data it rests on. This is the cheapest hallucination detector ever shipped: fabricated claims either quote nothing or quote text that is not in the context, and your code can check the quotes mechanically before a human ever reads the answer. Quote-check catch rate is not a published lab number; the mechanical check is the practice.

The output side: schemas as guardrails

Chapter 2 introduced grammar-constrained decoding as the plant's secret advantage; this is where it goes to work. Every floor question that recurs deserves a schema: the enum of legal verdicts, the required evidence field, the confidence grade, the INSUFFICIENT_DATA arm. Constrained output turns free-text grading into field checking, makes downstream automation safe to build, and — the underrated part — makes *evaluation* mechanical, which Chapter 6 will exploit: a schema'd answer scores itself against a labeled key without a human reading prose [LAB: RESULTS-MATRIX R.158 — deterministic enum decode, bit-exact across two launches (640 rows, 0 score differences, 0 argmax flips); channel-level AUROC 0.938, ECE 0.012].

Schema design has its own craft. Keep enums short — every added arm is a place to be wrong, and models discriminate eight options far better than thirty. Make the abstention arm first-class, not a string the model must remember to produce. Version your schemas like any interface, because Chapter 6's evaluations pin to them. And log every raw model response alongside its parsed form: when a verdict is challenged later, the raw text is your flight recorder.

A worked rendering

Theory earns its keep in the diff between bad and good input, so here is one, end to end. The question: "why did conveyor 2 stop at 06:41?"

The accidental deployment pastes what the export button produced — hundreds of rows shaped like this, one per tag

per second:

```text fragment

```
2026-08-27T06:39:58.113Z,PLC7.DB44.REG117,4212,1
2026-08-27T06:39:58.113Z,PLC7.DB44.REG118,0,1
2026-08-27T06:39:58.641Z,PLC7.DB44.REG117,4213,1
```

Raw addresses, unscaled integers, a quality flag nobody explained, timestamps to the millisecond for a question about minutes. The model must guess that REG117 is the drive current, that 4212 means 42.12 amps, that quality `1` is good – every guess a coin flip wearing a lab coat.

The engineered deployment renders the same facts like this:

```text fragment

```
CONTEXT: conveyor 2 trip investigation, window 06:36–06:43
local (America/Chicago)
```

```
conv2.drive_current_A: min 41.9, max 67.3 ← exceeds alarm
limit (55.0), last 0.0
conv2.motor_temp_C:    min 71,    max 74 (alarm limit 90 –
not reached)
conv2.state:           RUNNING → FAULTED at 06:41:12
alarms: 06:41:12 CONV2 OVERCURRENT (priority 1, active)
operator note 06:44 [quoted material, not instruction]:
    "reset twice before shift change, tripped again both times"
– j.m.
```

Same historian, same facts. The second rendering resolved names, applied scales, attached units, pre-computed the excursion against its limit, collapsed a thousand rows into the five lines that matter, fenced the human text, and stamped the timezone. Ask Chapter 2's three-way question against both and the gap is not subtle: against the first, a small model free-associates about registers; against the second, it has almost no room to be wrong, and the remaining judgment — overcurrent from mechanical jam versus drive fault, and what to check first — is exactly the judgment you wanted it applying. A raw-versus-rendered accuracy delta is not in the lab record; the argument is about what the model is asked to judge, not a scored gap.

The uncomfortable observation hiding in this example: most of the intelligence in the answer was placed there by the renderer. That is not a criticism of the model. It is the design. You want the probabilistic component operating on the shortest possible inferential leash, and the leash is braided out of decode tables you already owned.

The standing watcher

Everything above answers questions someone asked. The other half of floor duty is the question nobody asked yet:

continuous monitoring, where Chapter 3's always-on small model earns its electricity.

The pattern that works is exception-driven, not exhaustive. The watcher does not read the full stream — deterministic limit checks and your existing alarm system already guard thresholds, and racing them with a language model is using a poem to do a comparator's job. The watcher reads what the deterministic layer *cannot*: the conjunction of weak signals. A drive current trending up for a week, a bearing temperature that now runs three degrees warmer after each restart, an operator note mentioning "the smell again," none individually alarmed — rendered together into one periodic digest, with the standing question "what deserves a human's attention this week, and why?"

Two disciplines keep the watcher from becoming noise. **Dwell before speaking:** the digest runs daily or per shift, not per event, and repeats a flagged item only when its evidence strengthens — a model that re-announces the same trend every hour trains the crew to delete its reports, which is the alarm-management lesson of Chapter 9 all over again. **Track the hit rate:** every watcher flag gets a one-click disposition from the human who read it — useful, noise, already-known — and the running rate is reviewed like any instrument's calibration. A watcher below a usefulness floor gets returned or retired; sentiment is not a metric. Watcher precision from a plant deployment is not a published number.

The corpus you are accidentally building

One more consequence of doing this chapter properly, and it may be the most valuable: log every rendering and every answer, and you are building the exact training set Chapter 7 needs. The renderings are your plant's text in its canonical form; the questions are your plant's real question distribution; the verdicts — especially the ones a human corrected — are labeled examples of the highest possible relevance.

Three habits make the accident deliberate. Log the *rendered context*, not just the raw data, because the rendering is what the model actually read and what a future fine-tune should learn from. Capture the human disposition — accepted, corrected (to what), rejected — at the moment it happens, in the same record; reconstructed labels are worth a fraction of contemporaneous ones. And mark every record with its clearance status at capture time: what may be used for training, what contains names or sensitive material needing scrubbing, what must stay out entirely. Sorting that out record-by-record later is a project; a flag at write time is a column. Chapter 7 inherits this corpus and will be grateful

for every one of these habits — and the provenance discipline is the same one this book's own publisher applies to itself, which is not a coincidence.

What can go wrong: the adversarial note

One risk class is specific to reading *text* from the plant and deserves its plain-language warning. Operator comments, vendor documents, even alarm description fields are authored content, and authored content can contain instructions — innocently ("CALL JIM BEFORE RESETTNG, HE KNOWS THE TRICK") or otherwise. A model reading a context cannot fully distinguish data from directive; a work-order note that happens to read like a command can steer an answer. The mitigations are layered, none exotic: render untrusted text clearly fenced and labeled as quoted material; instruct the contract that quoted material is evidence, never instruction; constrain outputs so that even a steered model can only choose among legal verdicts; and keep a human between model verdicts and physical actions — which Chapter 10 will insist on for its own reasons anyway. Treat this the way you treat any untrusted input path into a control system: not with panic, with plumbing.

Brownfield honesty

Every pattern above assumed the decode table is right, the sensors work, and the historian kept everything. No plant matches that description. The pipeline has to be honest about its inputs' dishonesty, and the mechanisms are pleasantly mundane.

Undocumented tags — the registers nobody remembers mapping — get rendered as exactly what they are: PLC7.DB44.REG119 = 77 (unmapped tag – meaning unknown). That label does two jobs: it stops the model from inventing a meaning, and it turns every investigation that stumbles over the tag into a small documentation work order. Dead and stuck sensors are a renderer responsibility, not a model discovery: a value that has not changed in a week, or reads outside physical possibility, gets flagged by code (flatlined since 08-20) so the model treats it as an evidence gap rather than a fact. And historian gaps — the outage from Chapter 9, the tag that started logging only last spring — must render as explicit absence (no data 02:12–02:31), because a model shown a seamless series will reason as if the world were seamless too.

The shared principle: **every known defect in the data becomes a visible label in the rendering.** The model's abstention machinery — next chapter's subject — can only decline to answer when the rendering lets it see what is

missing. A pipeline that papers over its gaps upstream has quietly removed the model's ability to be honest downstream, and will then blame the model for the confident answer it was set up to give.

The chapter in one drawing

Historian → **decode** (names, units, scale — your existing tables) → **select** (question-scoped slice or exception-first summary) → **render** (aligned, unit-bearing, identifiers whole, judgments pre-computed) → **contract** (sources, abstention arm, schema, then the data) → model → **constrained output** (enum verdict + quoted evidence) → **mechanical checks** (quotes exist, schema valid) → human.

Every arrow is deterministic code except the model itself — which is precisely the point. The probabilistic component sits in the middle of a pipeline that feeds it honestly and checks it mechanically. Build the arrows well and Chapter 5 can make the model itself honest about the one thing the pipeline cannot check: whether the answer should exist at all.

One closing measurement to make once the pipeline stands: feed it a question whose answer you already know, end to end, and time every arrow. Wherever the minutes went is where your next engineering hour belongs — and it is almost never the model.

Chapter 5 — The Abstention Chapter

(draft v1, 2026-08-28 — written by Claude Fable 5, unverified. Lab citations attached where the record exists; remaining claims are labeled as unmeasured.)

Every chapter so far has been building toward a single sentence, and this is the chapter that gets to say it plainly: **on a plant floor, the most important thing a language model can produce is a refusal to answer.**

That sentence sounds backwards everywhere else in the AI industry. Demos are scored on answering; benchmarks reward the attempt; a chatbot that says "I don't know" three times in a row feels broken. But you do not run a demo. You run machines that can hurt people and processes that cost money by the minute, and in that world the failure modes are not symmetric. A model that says "I don't know, and here is what's missing" costs you a lookup or an escalation. A model that confidently names the wrong bearing costs you the teardown of the good one — and worse, it costs you the crew's trust in every answer that follows, including the correct ones. The entire economics of deploying language models on a floor turns on the rate of confident wrong answers, and abstention is the machinery that buys that rate down.

Chapter 2 explained why this machinery does not come built in: generation cannot stay silent, and "I don't know" is just another sentence shape the model learned — produced when the context makes it probable, not when the evidence makes it true. This chapter is about closing that gap from both ends: **plumbing that gives the model somewhere honest to stand, and training that teaches it to stand there.**

The grades of no

"The model should abstain" is too coarse to engineer. In practice a floor deployment needs about six distinguishable refusals, and conflating them wastes most of abstention's value:

Evidence-absent. The rendered context does not contain the answer. "The fault table excerpt does not cover code F-7288." This is the cleanest grade, the easiest to train, and — thanks to Chapter 4's brownfield labels — often mechanically checkable after the fact.

Evidence-conflicting. The context contains two answers. The manual says the limit is 90 °C; the alarm configuration

says 85. A model that silently picks one is manufacturing certainty; the correct output names the conflict and stops. Conflicts are gold for a maintenance organization — each one is a documentation defect located for free — but only if the model is trained to surface rather than resolve them.

Under-specified question. "Is the pump okay?" Which pump, okay for what, over what window? The correct response is a clarifying question, and it is a *different skill* from the other grades: the model must produce the missing parameters, not just decline.

Out-of-competence. The question is answerable but not by this system: a legal question, a warranty judgment, an instruction to modify a setpoint. The refusal names the right channel. This grade is mostly a policy statement wearing a model's voice, and the policy belongs in the contract prompt where audit can read it.

Stale-or-unreliable input. Chapter 4's flatlined sensor, the historian gap, the unmapped tag. The evidence exists but is flagged untrustworthy; the answer inherits the flag. "The reading suggests X, but the sensor has been flat since 08-20 — verify at the gauge."

Escalate. The model has an answer and the answer is alarming: evidence points to a condition that should not wait for the normal workflow. Strictly this is the opposite of refusing — but it belongs in the same taxonomy because it is the same trained judgment about *the limits of the current interaction*, and because the plumbing that carries it is identical: a first-class arm in the output schema.

Design your schemas so each grade is an explicit, selectable arm — Chapter 4's constrained decoding makes the arms unmissable — and downstream handling can differ: evidence-absent routes to a document lookup, conflicts file a documentation ticket, escalations page someone. When all six collapse into one shrug, the organization learns that the model's "no" means nothing in particular, and stops reading it.

The plumbing half: give honesty somewhere to stand

Before any training, most of abstention is affordance. A model *cannot* honestly decline if the pipeline hides what is missing, and it *will not* reliably decline if declining requires composing an unusual sentence against the gradient of its instincts.

The affordances, most of them already built in earlier

chapters:

A first-class arm. `INSUFFICIENT_DATA` as a schema value, not a phrase the model must remember to write. With constrained decoding, abstention becomes one legal token choice among a handful — the cheapest it can possibly be. Our measurements around enum-constrained verdicts consistently show format failures vanishing and the remaining errors becoming *judgment* errors [LAB: RESULTS-MATRIX R.158 — format is locked by enum decode; remaining errors concentrate where the task is beyond the model (scene-level AUROC 0.548, inverted at 281M)] — which is exactly the error type training can then address.

Visible gaps. Chapter 4's rule — every known data defect becomes a visible label — is abstention's raw material. A model can only say "no data for the window in question" if the rendering said no data 02:12–02:31 instead of splicing the series seamlessly.

A required "what's missing" field. Pair every abstention arm with a mandatory companion: name the evidence that would change the answer. This converts a dead-end "I don't know" into a work item — pull this manual section, check that gauge — and it also disciplines the model: fabricating a missing-evidence description is harder than fabricating an answer, so the field acts as a natural brake on lazy abstention.

Contract language that pre-authorizes refusal. The prompt states, before the data: "If the data below does not determine an answer, select `INSUFFICIENT_DATA`. That is a correct and preferred outcome." The sentence matters more than it looks. Models arrive tuned by their general training toward helpfulness; an explicit authorization measurably shifts the threshold. A contract-authorization ablation table is not in the cited lab record; the eleven-word cost is the prompt length, not a measured delta, words.

The training half: teaching the threshold

Plumbing gives the model somewhere to stand; training decides *when* it stands there. The distinction that organizes everything: abstention is not a fact the model learns, it is a **threshold** it learns — a decision boundary between "the evidence supports an answer" and "it does not," running through every topic the model will ever touch.

What moves the threshold, in the order you should try:

Demonstrations with the reasons visible. Supervised examples where the context genuinely lacks the answer and the target output is the right grade of no, *with the tell*

named: "the excerpt covers codes F-7000 through F-7199; the asked code is F-7221." Symmetric examples where the evidence is present and the target answers. The pairing is the point — a training set of only-refusals teaches refusal as a style, not a judgment.

Near-miss mining. The most valuable training examples live at the boundary: contexts where the answer *almost* appears — the right manual but the wrong revision, the right tag but the wrong week, a related fault code one digit off. Models fail at the boundary, not in the obvious cases, and Chapter 4's logging habit produces exactly these examples from your own traffic, pre-labeled by the humans who corrected them.

Asymmetric penalties. Wherever your tuning framework lets you weight errors, weight them like the floor does: a confident wrong answer is several times worse than a missed answerable question. The ratio is a policy decision your safety review should own — this book's lab treats it as a first-class training parameter rather than a default. A penalty-ratio sweep is not a published lab table; treating the ratio as a training parameter is the practice.

Progressive evidence removal. Build training sequences from a single case rendered at several evidence levels — full manual page, partial page, table of contents only, nothing — with the target flipping from answer to abstention at the level where a careful human's would. This trains the *slope* of the threshold rather than isolated points on it, and it doubles as the cleanest evaluation instrument this chapter has (you will meet it again below as the calibration gym).

A warning from our own program, because it is the predictable failure of doing the above with enthusiasm: **over-abstention is a real and measured failure mode, not a hypothetical.** Over-abstention as a failure mode is asserted from training practice; a named incident report is not attached. A model trained hard on refusals learns that refusing is safe, and begins declining questions the context plainly answers — which quietly destroys the deployment's value while looking responsible in every individual transcript. Abstention training without answerable controls in both training and evaluation is how you build a very polite paperweight.

Calibration: the honest middle

Between "answer" and "refuse" lives a third output worth engineering: the answer with its confidence attached. Not decorative confidence — calibrated confidence, where of the claims the model tags HIGH, nearly all are right, and of the claims it tags LOW, you genuinely cannot count on much.

Calibration is measurable with nothing but a labeled evaluation set: bucket the model's answers by its own stated grade, compute the accuracy within each bucket, and compare the curve to the diagonal. A small model will not give you a philosopher's calibration, but it does not need to: the floor needs three honest grades — act on it, verify first, treat as a hunch — and holding a model to three grades it means is an achievable, testable engineering target [LAB: RESULTS-MATRIX R.158 — channel-level reliability is monotone across all ten margin deciles (33.99% → 100%); at escalation floor 2.0 the model answers 71.9% of channels at 98.78% accuracy and catches 92.9% of the errors it would otherwise have made]. Wire the grades into the workflow: HIGH routes to the technician's queue, MEDIUM routes with its evidence attached for verification, LOW never leaves the review screen. Now calibration is not a model virtue; it is a routing rule with a measured error rate — which is a sentence a plant manager can approve.

Evaluating the skill of no

Chapter 6 builds the general evaluation machinery; abstention needs its specific instruments stated here, because most published evaluations simply do not measure it.

Score all four quadrants. Answerable-and-answered, answerable-but-refused, unanswerable-and-refused, unanswerable-but-answered. The last quadrant — the confident fabrication — is the one the floor fears; the second is the paperweight tax. A single "accuracy" number hides both. Report abstention precision and recall as first-class metrics beside answer accuracy, and set gates on all of them: our own gate philosophy — inherited from a benchmark program that would rather fail a good model than pass a lucky one — is that **a model unable to say "I don't know" fails the industrial gate regardless of its answer accuracy** [LAB: PROJECT-LOG 2026-08-02 — IEB-Health v1 is 25.6% abstain_escalate by design; constant ESCALATE baseline 25.8%; refusing correctly is the product. Industrial IEB floor 19.2%].

Run the gym. The progressive-evidence-removal sequences from the training section, held out, give you the threshold's location and sharpness: at what evidence level does the model flip, and how consistently? A model that flips at different levels for cosmetically different phrasings of the same case has a soft threshold, and soft thresholds are where the confident fabrications leak through.

Adversarial answerables. Include questions that *look* unanswerable — obscure phrasing, ugly rendering, a distractor gap label elsewhere in the context — but are

answerable from the given evidence. These catch over-abstention the way the fourth quadrant catches fabrication, and a gate needs both jaws.

A ladder, worked

Here is the calibration gym on one rung of real shape, because the pattern is easier to copy than to describe. Take Chapter 2's drive fault, F-7221, and build five renderings of the same investigation:

Level 1 — full evidence. The fault table row for F-7221, the drive's decel parameters, the historian slice showing the bus voltage spike. Target output: the answer, HIGH confidence, quoting the table row and the spike timestamps. Any abstention here is a paperweight point against the model.

Level 2 — answer present, corroboration missing. The fault table row only; no historian slice. Target: the answer, MEDIUM, with the "what's missing" habit inverted into a verification pointer — "table attributes F-7221 to DC bus overvoltage; confirm against the bus voltage trend for the trip window."

Level 3 — adjacent evidence. The fault table covers F-7200 through F-7219; the asked code is one page past the excerpt's edge. This is the boundary rung where models fail: the material *looks* right, the pattern-completion pull toward "it's probably also an overvoltage variant" is strong, and the correct output is evidence-absent abstention naming the exact gap — "excerpt ends at F-7219." Most of your training attention belongs here.

Level 4 — conflicting evidence. Two revisions of the fault table in context, one attributing F-7221 to overvoltage, the other to a brake-resistor fault. Target: evidence-conflicting, both rows quoted, no resolution attempted. Watch specifically for the model that answers from the *first* row and never mentions the second — silent conflict resolution is the most dangerous behavior on this ladder because it is indistinguishable from a clean answer unless you built this rung to catch it.

Level 5 — nothing. General drive documentation, no fault table at all. Target: evidence-absent, with the missing-evidence field naming the document class, not just "insufficient data."

Five renderings, one afternoon to build from any real case, and the set earns its keep twice: as training demonstrations with the targets as labels, and — held out, with fresh cases — as the evaluation that tells you where your model's threshold actually sits and whether tuning moved it. When

the level-3 rung flips from confident guess to named-gap abstention while level 1 stays answered, you have watched the threshold learn. That, in miniature, is the whole chapter.

The organizational half: making "no" a good outcome

One more component, and it is not in the model. Abstention only survives contact with an organization that rewards it.

The failure pattern is easy to predict: the model says `INSUFFICIENT_DATA`, the technician's screen shows a dead end, the crew learns the tool "doesn't know anything," usage collapses, and the deployment dies — with perfect calibration. The fix is workflow, not weights. Every abstention arrives carrying its "what's missing" field, so the screen never shows a shrug; it shows the next action: *the fault table for this drive model is not in the document store — scan section 7 of the paper manual and this question becomes answerable for everyone, forever*. Track abstentions by cause the way you track alarms by tag: the histogram is a map of your documentation debt, ranked by how often reality asks for each missing piece. In review meetings, treat a correctly-refused unanswerable exactly like a correctly-answered question — both are the system working — and treat the confident fabrication as the incident it is.

And close the loop with Chapter 3's two-model pattern: an abstention from the small always-on model is a *routing event*, not a terminus. It forwards the case — context, question, and the named gap — up to the larger model or the human queue. The refusal becomes the first step of the answer, which is what it always should have been.

The chapter in one sentence

Give the model an honest place to stand (arms, labels, contracts), train the threshold from both sides (refusals *and* answerable controls, penalties shaped like the floor's real costs), measure all four quadrants forever, and build the workflow so that "I don't know, and here is what's missing" is received as what it actually is: the second most valuable sentence a plant's language model can produce, one notch below the correct answer and a full order of magnitude above the confident wrong one.

The next chapter builds the machinery that holds all of this honest — the evaluation gate that decides, with error bars, whether any of it is allowed near your floor.

One last habit binds the chapter to the rest of the book: put

the four-quadrant abstention report next to the accuracy number in every evaluation you ever publish or read, including this book's own. An accuracy figure without its abstention quadrants is a demo statistic; with them, it is an engineering document. The difference is the whole premise of deploying language where machines can hear it.

Chapter 6 — The Quality Gate

(draft v1, 2026-08-28 — written by Claude Fable 5, unverified. Lab citations attached where the record exists; remaining claims are labeled as unmeasured.)

Every chapter so far has ended by deferring to this one. Chapter 3's sizing rule said "climb only on measured failure." Chapter 5's abstention metrics needed somewhere to live. This is the chapter where measurement stops being a virtue and becomes a machine: the evaluation gate — the apparatus that decides, with error bars, whether any model, prompt, quantization, or pipeline change is allowed near your floor.

The gate's job description is one sentence: **it would rather reject a good change than pass a bad one.** Everything in this chapter is engineering toward that asymmetry, because the floor's costs are asymmetric in exactly that direction. A rejected good change costs you a week of investigation. A passed bad change costs you a wrong verdict in production, discovered by the person acting on it.

From the floor. The most dangerous result on a plant floor is not a failure — it is a pass that is too clean. I have watched test equipment hand back a perfect result because a person *made* it hand back a perfect result: bypassed the fixture, forced the reading, moved the probe off the fault. Not out of malice, usually — a bad number that day was simply more expensive to them than an honest one. You do not catch that by trusting the number. You catch it by keeping enough of the surrounding data to ask whether the number could even be true: was the cycle time physically possible, did the other signals move the way a real pass moves, is this station suddenly, suspiciously flawless? A gate that reads only the verdict can be gamed by anyone whose incentive points at *clean* instead of *correct*. Build the gate — and train the model — to read the evidence, not the conclusion. This is the same discipline the rest of this chapter applies to the models themselves: a score that looks too good is a finding, not a victory.

Why vendor numbers cannot be your gate

Start with the uncomfortable fact that makes this chapter necessary: the published benchmark numbers that models arrive wrapped in — general knowledge scores, reasoning suites, leaderboard ranks — are nearly useless for your decision. Not because they are dishonest, but because they measure the wrong distribution. Your floor's questions are your fault tables, your historian renderings, your technicians' shorthand, your schemas. A model's rank on graduate-level

reasoning problems tells you approximately nothing about whether it correctly reads your VFD fault table, and the correlation between general rank and your-task performance is weak enough at the small end of the ladder that ranking by it is closer to superstition than diligence [LAB: RESULTS-MATRIX \$C - Qwen3.6-35B-A3B 71.0 MMLU / 87 tool hardmode versus DeepSeek Q8-MTP 88.3 MMLU / mean 55 tools; the two scores do not rank the same models].

The gate you need is built from your own material. The good news, and a running theme of this chapter: building it is cheaper than it sounds, and the expensive part — the labeled cases — is Chapter 4's logging habit already paying out.

Noise first: the lesson we keep paying for

Before designing anything, absorb the single most important empirical fact about evaluating language models, because our lab has now paid for it several times: **small evaluation suites are far noisier than they look, and the noise wears a convincing costume.**

The canonical incident from our own record: a fifteen-scenario tool-use suite, run three times against the *identical* model, configuration, and temperature, returned scores spanning ten points [LAB: RESULTS-MATRIX \$C footnote - ± 10 pts across three identical runs at temperature 0]. Ten points is the size of a headline improvement. An engineer who ran the suite once before a change and once after could "measure" a breakthrough or a catastrophe that consisted entirely of batch-packing nondeterminism. Nothing about the model had changed. The dice had.

The discipline that follows, written as rules because we follow them as rules:

1. **One surprising number is a hypothesis, not a result.** Re-run it.
2. **Two runs that disagree get a third, and a control** — the unchanged configuration, re-benchmarked, to measure the noise floor itself.
3. **Publish ranges, not lucky draws.** A gate that compares single runs is comparing noise. Every score that matters is a mean over repeats with its spread attached, and the gate's thresholds are set wider than the measured noise floor.
4. **Change one thing at a time.** When two things changed together, bench the configuration that isolates each — the control run is what converts "the new build is faster and nothing broke" from a hope into a statement [LAB: RESULTS-MATRIX \$E - speculation-off control isolating quality from speed].

None of this is statistics beyond a maintenance department's comfort; it is the same repeat-and-control instinct you apply to a suspicious vibration reading. The only novelty is that the AI industry's demo culture has normalized skipping it.

Anatomy of a floor gate

A working gate has four layers, each catching what the previous cannot:

Layer 1 — format checks, free and total. Does the output parse? Does it conform to the schema? Are the enum values legal, the required fields present, the quoted evidence actually present in the context? These checks are deterministic code, they run on every single response in production as well as in evaluation, and with Chapter 4's constrained decoding they should pass at essentially 100% — which is precisely why they stay in the gate: a format regression is the loudest possible alarm that something structural broke. Pre/post format-check pass rates are not a published pair; keeping format checks in the gate is the practice.

Layer 2 — labeled-case scoring. The heart: a few hundred real cases with known-good answers, scored mechanically thanks to schema'd outputs. Include every case family the floor actually produces: the routine, the boundary (Chapter 5's near-misses), the unanswerable (all four abstention quadrants), the adversarial-answerable, and the formerly-failed — every production mistake that got a human correction enters the gate set permanently, so no failure has to be discovered twice. Curate for coverage, not volume: two hundred cases that span the distribution beat two thousand that oversample the easy middle.

Layer 3 — behavioral probes. Targeted mini-suites for properties that case-level scoring misses: does the model still respect the contract when the context is at 90% of window capacity? Does performance hold when tag names are swapped for unfamiliar ones of the same shape — or was the model memorizing your identifiers? Does the abstention threshold sit where Chapter 5's ladder left it? Each probe is a dozen cases aimed at one failure mode you have reason to fear.

Layer 4 — the retention floor. Chapter 3's specialist trap, enforced: alongside your task suites, a small general-capability suite with a hard minimum. A tuned model that aces the plant work and collapses on general instruction-following has been damaged in ways your task suite cannot see yet; the general floor catches the amputation early [LAB: PROJECT-LOG — retention gates: retain $\geq 90\%$ of base MMLU and IFEval; 0% general replay produced the

cliff between 0% and 1%].

The gate that was too strict, and why we kept the story

A gate is code, and code has bugs. The failure mode you must design for is the gate that is wrong in the *strict* direction — and our lab's most instructive example is the execution gate that rejected generated code for working correctly: the checker's sandbox judged legitimate solutions as failures because of an environmental assumption the checker itself made [LAB: PROJECT-LOG – the execution gate that rejected correct code]. For a while, the measured capability of every model under test was artificially depressed, and the models were innocent.

Three lessons earned there. First: **gates need controls too**. Feed the gate known-good answers on a schedule — human-written, verified solutions — and when the gate rejects one, the gate goes under investigation, not the model. Second: a strict-side bug is quieter than a lenient-side bug, because nothing bad ships; you simply lose true capability to a phantom, and only the control run reveals it. Third — and this is why the story is in a published book — an evaluation program that cannot admit its instrument was broken will silently convert instrument error into "findings." Our program's rule is that instrument failures get written up with the same rigor as results, including the retraction of anything the broken instrument "found" [LAB: PROJECT-LOG – Finding 25 retraction: four instrument defects, not a finding]. Your plant's version of that rule: the gate's own defect log is part of the gate.

Contamination: the quiet score inflater

One more instrument hazard, specific to language models: the model may have already seen your test. Public benchmark questions leak into training corpora; more insidiously for a floor deployment, *your own* evaluation cases can leak into your fine-tuning sets through the very logging pipeline Chapter 4 recommended — the case you evaluated in March becomes training data in May, and June's evaluation is partly a memory test.

The defenses are procedural, not clever. Keep the gate set physically separate from the training corpus with a checked, one-way boundary — our lab treats train/eval contamination checking as a standing pipeline step, not a one-time audit [LAB: PROJECT-LOG – dataset ledger snapshots with SHA-256 + contamination checks are a standing qualification step; semantic/manual contamination remains an open residual]. Rotate: retire a slice of the gate set to training periodically

and replace it from fresh production traffic, so the gate ages with the plant instead of fossilizing. And run the memorization probe from Layer 3 — same case shapes, fresh identifiers — whose divergence from the named-case scores is your contamination gauge.

The judge problem

Schema'd outputs made scoring mechanical, and you should fight to keep it that way. But some floor tasks are irreducibly prose — the shift summary, the incident narrative, the explanation field beside the verdict — and prose needs judgment to grade. The industry's answer is to use another language model as the judge, and it works well enough to use and badly enough to instrument.

Use it with its failure modes named. Judge models prefer longer answers, prefer fluent answers, prefer answers that share their own phrasing habits, and drift toward leniency when the rubric is vague — each a bias that will flatter exactly the failure modes you built this gate to catch. The countermeasures mirror everything else in this chapter: give the judge a rubric with binary checks rather than a 1-10 feeling ("does the summary mention the 06:41 trip: yes/no"); never let a judge model grade its own family's outputs — independence matters for graders exactly as it does for critics; and control the judge itself with planted cases — a known-excellent summary and a known-flawed one salted into every batch, so a judge that fails the plants gets investigated before its grades count. Judge-control agreement rates are not a published lab table; salting known-flawed items is the practice. Where a prose task matters enough to gate a deployment, the final word stays with periodic human scoring of a sample; the judge model's job is coverage between those samples, not authority over them.

Building the first gate in one week

The apparatus above sounds like a quarter's project. It is a week, if you spend the week on the right things — and the week pays for itself the first time it blocks a bad change.

Day one: pick the single highest-value recurring question on your floor and freeze its schema. **Days two and three:** harvest cases — from Chapter 4's logs if you have them, from two afternoons with a maintenance lead and a stack of real work orders if you do not. Label fifty routine, twenty boundary, twenty unanswerable (spread across Chapter 5's grades), ten adversarial-answerable. A hundred labeled cases is a real gate; do not wait for five hundred. **Day four:** write the runner — a loop that renders, asks, parses, and

scores; with schemas it is an afternoon of code. Run it five times against your current configuration and write down the spread: that number, the noise floor, is the week's most valuable output, because every future comparison is meaningless without it. **Day five:** set thresholds outside the noise floor, wire the runner to your deployment checklist, and file the first report as the baseline.

From then on the gate grows by accretion: every production correction becomes a case, every incident becomes a probe, every quarter retires stale cases to training and pulls fresh ones from traffic. Our own benchmark program began as approximately this week and grew into a named standard by exactly this accretion [LAB: PROJECT-LOG 2026-08-02 – founder named the benchmark IEB (Industrial Edge Bench); collision sweep clean against IndustryBench/AssetOpsBench/FieldWorkArena; G1 audit: 879 items re-derived, 12 classes independently audited] — the gate you start crude this month beats the perfect gate you start next year, by the width of a year.

What the gate cannot see

Honesty about the instrument's edges, because a gate that claims totality teaches people to stop thinking. The gate measures the model-and-pipeline's answers against a frozen set of cases. It does not measure whether the technicians trust the tool, whether the screen presents abstentions as next actions or dead ends, whether the latency fits the rhythm of a shift, or whether the questions people actually ask are drifting away from the questions the gate contains. Those live in usage metrics and in Chapter 5's disposition tracking — accepted, corrected, ignored — which is the gate's necessary complement: the gate says *the system answers correctly*; the disposition stream says *the system is being used, and how*. A deployment green on the first and red on the second is not a success with an adoption problem. It is a system answering questions nobody is asking, and only the pair of instruments together can tell you so.

Cadence: when the gate runs

A gate that runs only when someone remembers is a gate that runs after the incident. Wire it to triggers instead: every model version change, every quantization change, every prompt-contract edit, every schema version, every tokenizer or renderer change — the full list of things Chapter 4 taught you are part of the instrument. Plus one more trigger the AI industry keeps relearning: **the calendar**. Drift is real even when nothing "changed" — a vendor's silent update (Chapter 1's version-pinning argument), a new product line's vocabulary entering the traffic, a season's different failure distribution. A monthly full-gate run against production

configuration, filed with its ranges, is the deployment's routine bloodwork; the trend across months is as informative as any single result [LAB: RESULTS-MATRIX §G – 12-min soak on promoted Q3-MTP: 86 requests, 0 errors, 100% mean acceptance, +2 MiB VRAM drift. That is a soak, not a months-long drift study].

Keep every gate report. The chronological file of them is the deployment's medical history: what was tried, what the noise floor was, what regressed and when, which gate defects were found and fixed. When the auditor, the insurer, or the new plant manager asks "how do you know this thing works?" — the answer is not a slide. It is that file.

The number the business actually needs

One translation duty falls on whoever owns the gate, because the gate's native outputs — accuracy ranges, abstention quadrants, noise floors — are not the language the plant runs on. The business runs on error budgets, and the gate is precisely the instrument that lets you write one.

The translation reads like this: at the gated configuration, the verdict pipeline produces at most N confident-wrong answers per thousand cases (upper bound of the measured range, not the mean — the gate's pessimism is the budget's honesty), each reaching a human reviewer before any action; the expected cost of a reviewed wrong verdict is one technician-interruption; therefore the system's worst-case error cost per month is a number, with a measurement behind it, revisited at every monthly gate run. Set beside it the value column the disposition stream provides — questions answered, lookups avoided, documentation debts surfaced — and the deployment stops being a bet on a technology and becomes a line item with a maintenance schedule. That sentence structure, more than any model capability, is what carries a pilot through its first budget review — and every term in it came off this chapter's instrument, which is the quietest argument for building the instrument well.

The gate in one page

Build the set from your own logged, corrected traffic; span routine, boundary, unanswerable, adversarial, and every past failure. Score mechanically through schemas. Run repeats; publish ranges; set thresholds outside the noise floor. Keep a general- capability floor beside the task scores. Feed the gate known-good controls and investigate the gate when it rejects one. Guard the train/eval boundary and probe for memorization. Trigger on every change and on the calendar. File everything.

None of it is exotic, and that is the point this book has been circling since Chapter 1: the difference between a plant that can trust its model and one that cannot is not the model. It is the instrument the plant built around it — and unlike the model, the instrument is entirely within your control.

And when the gate blocks something you wanted to ship — it will, and it should — write the rejection down with the same care as a pass. The file of near-misses is the instrument's proof that it earns its keep.

Chapter 7 — Training on the Real World

(draft v1, 2026-08-28 — written by Claude Fable 5, unverified. Lab citations attached where the record exists; remaining claims are labeled as unmeasured.)

Every chapter until now has treated the model as a purchased part: pick a size, wrap it in plumbing, gate it. For many floors that is the whole story, and a good one. This chapter is for the moment the gate tells you the purchased part has a ceiling — when grounding is right, schemas are tight, the tokenizer question has been asked, and the boundary cases still fail because the model has simply never lived in your world. The fix is training, and training runs on the one asset nobody can buy: your data.

Read this chapter with Chapter 6 already installed, because its first commandment governs everything here: **no training run without a gate to measure it**. Training without an evaluation is spending money to change a system in an unknown direction.

The ladder of intervention (climb it in order)

Training is the *last* rung of a ladder, and each lower rung is cheaper, faster, and more reversible. The gate decides when to climb; impatience is not a reason.

Rung 0 — better plumbing. Chapter 4's renderer and Chapter 5's contracts fix most of what gets blamed on the model. Our own failure-attribution habit exists because the majority of "the model is too weak" complaints dissolve under inspection into rendering, selection, or contract defects. A counted attribution tally is not published; inspection before retraining is the method.

Rung 1 — examples in the prompt. Before changing weights, show the model two or three worked cases inside the contract. For format and tone this is often training's equal at zero cost; its limit is capacity — examples spend context window, and their influence fades on genuinely unfamiliar material.

Rung 2 — supervised fine-tuning (SFT). Continue training the purchased model on your labeled pairs — rendered context in, correct schema'd verdict out. This is the workhorse rung, the one this chapter mostly details, and on small models it is genuinely accessible: hours on a workstation GPU, not weeks on a cluster. A per-tier fine-tune

wall-clock table is not in the lab record; "hours not weeks" is the working envelope for the small rungs, not a spreadsheet.

Rung 3 — continued pretraining. Feed the model raw domain text — manuals, procedures, standards — before any task tuning, so the vocabulary and idiom of your world stop being foreign. Worth it when the domain gap is wide (the model has plainly never read your industry) and you hold enough text to matter.

Rung 4 — from scratch. New tokenizer, new weights, your corpus at the center. The rung where Chapter 3's tokenizer-as-budget finding came from [LAB: PROJECT-LOG 2026-08-03 – from-scratch 32k industrial tokenizer]. For a single plant this rung is rarely rational; it exists for platform builders, and this book's own lab lives there so that plants do not have to.

The corpus is the product

Whatever rung you climb to, the work is nine parts data to one part training command. The corpus disciplines, in the order they save you:

Capture at the source, with consent flags attached.

Chapter 4's logging habit — rendered context, model output, human disposition, clearance status, all in one record — is the corpus assembling itself. The clearance flag at write time is the discipline that keeps the lawyers calm later: what may train, what needs scrubbing, what never leaves the historian's shadow. Our lab's standing rule marks every captured stream at ingestion, because sorting a million records retroactively is a project with no champion [LAB: PROJECT-LOG – capture hygiene is corpus ok plus hashed manifests and logged transformations; drop-only transforms cannot introduce contamination, but they also cannot clean what was already in].

Real beats synthetic; synthetic fills the gaps real cannot. Your logged traffic is the gold standard — it is, by construction, the exact distribution the model will face. Its weakness is coverage: the faults that happen rarely, the abstention cases (Chapter 5's ladder), the adversarial-answerables. There, generated data earns its place: a larger teacher model, given your real documents, can author boundary cases at volume — distillation, in the field's vocabulary, and the engine of most small-model quality today. Two cautions from our own distillation program. First, teacher outputs inherit teacher errors, so generated cases pass through the same human-spot-check and gate machinery as real ones — synthetic data is an ingredient, never a bypass. Teacher-error rates in distilled sets are not a published table. Second, license and terms:

know what your teacher's terms permit trained artifacts to do commercially, and record the answer in the corpus manifest, because retrofitting provenance onto a trained model is impossible. Write it down at generation time or lose it forever.

Deduplicate and decontaminate, mechanically. Two hygiene passes run before any token reaches training. Near-duplicate removal, because repeated text teaches repetition in exactly the way Chapter 2's compression story predicts. And gate-set exclusion — the contamination boundary from Chapter 6 — enforced by tooling, not by promise: our pipeline checks materialized training shards against evaluation sets as a standing step, and the check has caught real leaks that manual diligence had already signed off on [LAB: PROJECT-LOG – contamination checking is a standing pipeline step; semantic/source-family contamination remains an open residual even after snapshot checks pass].

Balance what you feed. A corpus assembled from convenience oversamples the routine. Weight by what the gate says the model gets wrong, not by what the logs happen to hold: boundary cases up, greatest-hits down, and — Chapter 5's warning standing — answerable controls always paired with abstention cases, so the training pressure pushes the threshold rather than just the refusal reflex.

Where the data actually comes from: a field inventory

"Your data" sounds like one thing; on a real floor it is six, each with its own effort profile and its own trap.

Service manuals and OEM documentation. The densest value per page and the messiest acquisition: much of it lives as scanned PDFs, and OCR quality decides everything downstream. Budget real time for the ugly ones — a fault table whose columns OCR scrambled is worse than no table, because it trains confident misreadings. Spot-check extraction against the paper with the same sampling discipline the gate uses; a manuals corpus is an instrument too.

Work orders and maintenance history. The richest task-shaped data you own: symptom, diagnosis, action, outcome, in your technicians' own language. Also the most sensitive — names, blame, the occasional colorful assessment of a vendor. The clearance flag earns its keep here; so does a scrub pass that replaces names with roles before anything reaches a training shard.

Historian exports and alarm logs. Unlimited volume, low

per-line value — raw telemetry teaches a language model surprisingly little. Its training value appears only after Chapter 4's rendering: excursion summaries paired with the questions they answer. Train on renderings, never on raw dumps.

Shift notes and operator logs. Idiom, shorthand, and the plant's real vocabulary live here — this is where the tokenizer and the model learn how your people actually write. Same scrub rules as work orders.

Standards and regulations. Public, clean, voluminous, and generic. Useful as continued-pretraining ballast for domain vocabulary — our own industrial corpus draws heavily on public regulatory text for exactly this role [LAB: PROJECT-LOG 2026-08-03 – regulatory corpus across nine agencies] — but do not mistake it for task data; no regulation ever answered a work order.

Vendor bulletins and field advisories. Small, fresh, high-value: the documents that correct the manuals. A capture habit for these pays twice — once in training, once because Chapter 4's retrieval layer wants them anyway.

The inventory's summary line: effort concentrates where value does — manuals and work orders first, rendered telemetry second, everything else as seasoning.

The worked tune: the extractor, one year later

Pick up Chapter 3's work-order extractor where the sizing walkthrough left it: the line-side class passed the gate with margin, shipped, and has now logged a year of traffic. The gate's monthly runs show the ceiling: symptom-field accuracy plateaued, and the residual errors cluster in exactly two shapes — new-product vocabulary the base model never saw, and the technicians' compressed shorthand for intermittent faults.

The tune that addresses it, by this chapter's numbers: the year yielded a few thousand disposition-labeled records; roughly a fifth are corrections, the high-value minority. Augment the thin spots with teacher-generated boundary cases built from the real manuals; balance so corrections and abstention cases punch above their volume; split; smoke-test on fifty; run the tune on the workstation GPU overnight. The after-gate tells the story in ranges: symptom accuracy up meaningfully, the two error clusters visibly compressed, abstention quadrants unmoved, retention floor intact. An extractor-tune before/after table is not attached; this paragraph is the intended shape of a result, not a scored run.

Total cost: one engineer-week, mostly on data, exactly as the nine-to-one ratio promised. The model file that results gets a version, a manifest, and a gate report stapled to it — and the plant now owns a small model that no vendor could sell them, because no vendor has their year of corrections.

What you must be able to reproduce

A tune that cannot be reproduced is a lottery ticket that happened to win. The artifact list that makes it engineering — versioned together, referenced by the model's release record:

the **dataset manifest** (which records, which snapshot, which clearance flags, which scrub pass); the **tokenizer identity**; the **base model checksum**; the **training configuration** (every hyperparameter, including the seed); the **checkpoint lineage** (which step shipped and why — the validation curve that chose it); and the **gate reports**, before and after, with their noise floors. Six files, none large, and together they answer the question every auditor and every future engineer will ask in the same words: *what exactly is this model, and how would we make it again?* Our lab treats a run missing any of the six as unshippable regardless of its scores — the provenance page this book's own publisher demands of authors is the same discipline pointed at weights. A named run-manifest standard is practice (hashed snapshots, transformations logged), not a published schema number.

Whose knowledge is this?

One more discipline, and it is not technical. A fine-tuned floor model is largely a compression of your technicians' accumulated judgment — the corrections they typed, the shorthand they invented, the diagnostic instincts their work orders encode. Treat that fact with the respect it is owed. The labeling afternoons go better when the people whose knowledge is being captured know what it is for and see the result: the extractor that stops mangling their shorthand is *their* improvement, and saying so costs nothing. The disposition buttons must never become a surveillance instrument — the moment corrections feed performance reviews, the corrections stop, and with them the corpus. And when the tuned model works, the plant has not replaced its technicians' knowledge; it has given it a backup copy and a faster index. Said plainly and meant, that sentence is the difference between a crew that feeds the system and one that starves it — and the corpus, like every instrument in this book, runs on what the crew decides to give it.

Running the tune without fooling yourself

The mechanics of a floor-scale SFT run are almost anticlimactic — a config file, a labeled dataset, hours of GPU time. The self-deception opportunities are where the engineering lives:

Split before you start. Train, validation, and the gate's held-out set, separated before the first step and never merged. The validation curve tells you when to stop; the gate — untouched during training — tells you what you actually built.

Overfit on purpose once. A tiny sanity run on fifty examples should reach near-perfect training scores quickly; if it cannot, the pipeline is broken somewhere between data and loss, and no full run should start until the small one behaves. Cheap, boring, and it has saved us real GPU-days. The smoke protocol is the practice; a counted GPU-day savings table is not attached.

Checkpoint on the cadence Chapter 9 taught, because training runs are exactly the long-running state the power-loss chapter was about — ours resumed mid-run through two building-wide outages on the strength of nothing but cadence and tested restores [LAB: PROJECT-LOG 2026-08-22/24 – training resumed at step through both crashes].

Gate before and after, same suite, same repeats. The before-run is the baseline and the noise floor; the after-run is the claim. The difference, expressed as ranges, is the entire truth of what the tune accomplished. Anything narrated beyond that — "it feels sharper" — is Chapter 6's demo culture sneaking back in through the side door.

Check the retention floor last and always. The specialist trap is sprung by exactly this chapter's activity. A tune that lifts the plant suite and dents the general floor has traded connective tissue for memorized competence; Chapter 3's rule holds — that trade fails the gate no matter how good the domain delta looks [LAB: PROJECT-LOG – retain $\geq 90\%$ of base MMLU and IFEval; a domain win that fails that floor fails the gate].

The escalation teacher

Chapter 3's two-model pattern was introduced as an operations idea: the small always-on model escalates its abstentions to a larger one. Notice what that architecture quietly produces on the training side: a perfectly targeted teacher, running for free on exactly the cases the small model cannot handle.

Every escalated case arrives pre-labeled as a small-model gap — that is *why* it escalated — and departs with the larger model's answer attached, plus, for the cases a human then reviewed, a disposition on that answer. Fold the reviewed set back into the next tune and the loop closes: the small model's weakest distribution becomes its next training set, authored by its own escalation partner, at the rate the floor actually generates hard cases. This is distillation with the sampling problem solved — no need to guess which boundary cases to synthesize when the deployment is harvesting the real ones nightly. Escalation-loop gains per cycle are not a published series.

The loop needs two governors or it eats itself. Only *human-dispositioned* escalations train — the big model's unreviewed answers are teacher outputs like any others, and recycling unchecked teacher errors compounds them with each cycle. And the gate's held-out set stays outside the loop entirely, per Chapter 6's boundary — a self-improving system that grades itself on the cases it trained on will report asymptotic perfection while learning nothing. Governed, the loop is the closest thing this book offers to a deployment that gets better by being used; ungoverned, it is a photocopier of mistakes. The difference is two rules and the discipline to keep them.

When the answer is: do not train

The honest close, because a chapter about training owes you the cases where the right call is putting the tool down.

Do not train to fix what plumbing can fix — rung zero exists because it wins more often than pride admits. Do not train on a corpus you would be uncomfortable showing the auditor, the union, or the vendor whose manual you scanned; the corpus manifest is a disclosure document, and this book's publisher applies the same rule to itself. Do not train against a moving target — if the schemas, renderer, or tokenizer are still changing weekly, tune after they settle, or you will be paying to specialize a model to a pipeline that no longer exists. And do not train past the gate's ability to measure: when your labeled cases number in the dozens, every one of them belongs in evaluation, not training — gather first, tune later.

The thread through all four: training converts data into behavior, permanently and somewhat opaquely. It is the least reversible thing this book teaches. The disciplines around it — clearance flags, contamination checks, before-and-after gates, retention floors — are not bureaucracy around a simple act. They are what makes an irreversible act safe to take, which is a sentence a plant engineer has heard before, in front of different machinery,

and knew to respect.

And when a tune ships, close the loop the way every chapter here closes it: the run's six artifacts filed, the gate report attached, a dated entry in the deployment log saying what changed and why. The next engineer — possibly you, a year from now, at 2 AM — inherits a model with a paper trail instead of a mystery with a version number.

Chapter 8 — Deployment Shapes

(draft v1, 2026-08-28 — written by Claude Fable 5, unverified. Lab citations attached where the record exists; remaining claims are labeled as unmeasured.)

The pipeline exists (Chapter 4), the model is honest (Chapter 5), the gate is armed (Chapter 6), maybe the weights are yours (Chapter 7). What remains is the question a plant asks about every system it adopts: what does this thing physically look like, where does it sit on my network, and who restarts it at 2 AM? This chapter is the shapes catalog — the handful of deployment topologies that actually occur, with the configuration traps we have collected the honest way, by falling into them.

The parts list, one more time

Chapter 1 previewed the stack; here it is as a deployment bill of materials. **The weights file:** one large file, checksummed, versioned like firmware. **The inference engine:** open-source serving software that loads the weights and speaks HTTP — the open engines are mature, actively maintained, and run everything this book discusses; our lab's production has run on one for its entire life [LAB: RESULTS-MATRIX \$A — engine lineage on the same UD-IQ3_XXS 102 GB model: pre-#25545 ~2 tok/s craters; mainline CUDA ~10.8 bimodal; taco 13.1 bimodal; pr25545 GPU indexer 26.2 stable (24.5–28.5), promoted 07-12]. **The service wrapper:** an init-system unit with restart policy, Chapter 9's condition gates, and a log destination. **The gateway:** the small application layer that owns Chapter 4's rendering and contracts — the only custom software in the building. **The gate rig:** Chapter 6's runner, on the same box or beside it. Nothing else. No orchestration cluster, no vendor appliance, no subscription.

Shape one: the sidecar box

The starter shape, and for many floors the final one: a single industrial PC or workstation beside the line, running one small model (Chapter 3's pocket or line-side class), serving one or two applications — the work-order extractor, the fault-code assistant. Everything on one box: engine, gateway, logs, gate rig.

Its virtues are the small-model virtues from Chapter 3 made physical: it restarts in seconds, its spare is a copy (Chapter 9), and its blast radius is one line's convenience features. Its limit is concurrency and ambition — one box serves a work

cell, not a plant. Configuration notes that recur at this shape: pin the model file read-only; give the KV cache explicit headroom for your worst shift-change burst rather than letting defaults decide (Chapter 3's cache-exhaustion story); and log every request/response pair locally with rotation, because this box is also the corpus collector (Chapter 7) and its disk fills on the schedule of its usefulness.

Shape two: the department server

One GPU box (the engineer's-assistant class, sometimes two models resident — the two-model pattern under one roof), serving a department over the plant network: maintenance, reliability, and the control engineers all hitting one endpoint through their own thin clients. This is the shape where serving stops being an appliance and becomes a service, and three disciplines arrive with it.

Admission and identity. The endpoint answers only authenticated clients, and every request carries who asked — not for surveillance (Chapter 7's warning stands) but because dispositions, corpus clearance, and audit all key off it. A reverse proxy in front of the engine, doing TLS and tokens, is an afternoon of setup and the difference between a service and an open socket.

Queueing honesty. Batch serving (Chapter 3) means throughput scales well — until the memory ceiling, where the honest failure is a fast "server busy, retry" rather than a hung request. Configure explicit concurrency limits below the measured ceiling, and surface queue depth as a metric; the day it trends upward is the day you size the next shape [LAB: RESULTS-MATRIX \$B — PAR=4 c=1 26.2 tok/s to c=4 46.1 aggregate; taco PAR=8 c=8 65.7 aggregate; a reproducible c=2/c=3 dip on llama.cpp PAR=4 is logged as unresolved scheduling, not cache churn].

Version discipline. With many clients, "which model answered this?" must be recorded, not remembered. The engine's version string, the weights checksum, the contract version, and the schema version ride along in every response's metadata. Chapter 6's gate reports already pin these; production echoes them.

Shape three: the hierarchy

The full two-model pattern at plant scale: small always-on models at the line (sidecar shape), escalating to the department server, escalating — where policy allows — to a human queue or, in hybrid plants, an external frontier model for the rare case that earns it. Each hop is Chapter 5's abstention machinery working as routing; each hop is logged with its reason; and the traffic *down* the hierarchy is Chapter

7's escalation-teacher loop returning trained improvements to the edge.

The hierarchy's design rule: **capability flows down, data flows up, and neither crosses a boundary silently**. If an external model participates at the top, the boundary is explicit — which renderings may leave the building (clearance flags, again), stripped of what must not, logged as having left. A plant that cannot draw that line cleanly should cap its hierarchy at the walls; the whole premise of this book is that the local ceiling is high enough to be useful.

Shape four: the air gap

Some floors do not negotiate: no route exists between the process network and anything that touches the internet, and the model deployment lives entirely inside the wall. The good news is structural — this book's whole stack was designed offline-first, so the air-gapped shape is mostly the sidecar or department shape with its update path made explicit rather than assumed.

The update path is the design problem. Weights, engine builds, and document-store additions arrive by controlled media on a schedule: checksum manifest first, files second, gate run third — nothing serves until the offline gate rig passes it, which makes the gate the border checkpoint it always should have been. The gate set itself updates by the same ceremony in the other direction: fresh cases go out (clearance flags enforced at the boundary, per Chapter 7), labels come back. Plan the cadence by the drift trigger of Chapter 6 — monthly is common — and resist emergency exceptions; an air gap with a hurry-up path is a decorative air gap.

One capability deserves explicit celebration in this shape: everything still works. No license server phones home, no model degrades for want of a subscription check, no document leaves. The plants with the strictest walls are, not coincidentally, the plants this book's local-first argument was written for.

Sizing the box: the buyer's worksheet

Procurement wants a specification, not a philosophy. The worksheet, in order:

1. **Model memory:** parameters $\times$ bytes-per-weight at your chosen quantization, from Chapter 3's arithmetic, plus engine overhead.
2. **Cache memory:** worst-case concurrent contexts $\times$ per-token cache cost at your window size — the

shift-change number, not the average. Cache headroom is the line item most sizing exercises omit and Chapter 3's ceiling story is what it costs.

3. **Disk:** weights (twice — current and previous version, because rollback is a copy), the document store, and the request logs at your retention policy. Logs dominate within a year on a busy box.
4. **Thermals and power:** sustained-load rating for the enclosure it will actually live in, per Chapter 9's heat section — the mezzanine in August, not the lab in October. A box that throttles is a box that silently fails its latency budget.
5. **The spare:** the same line again, at cold-standby price.

Sum it, round up one hardware notch — the marginal cost of headroom at purchase time is a fraction of the cost of discovering its absence — and staple the worksheet to the deployment record. When the numbers came from measurements (the gate's throughput probes, the cache ceiling test), say so on the sheet; procurement respects an instrumented number and audits remember one [LAB: RESULTS-MATRIX \$F — fit recipes on the 128 GB VRAM box: IQ3 XXS 102 GB loads; blobfish Q4 175 GB needs --no-repack and mmap or it OOMs/segfaults; Q8-MTP 160 GB n-cpu-moe 14; Q3-MTP 143 GB prod n-cpu-moe 11].

The security posture, stated plainly

A language-model service is one more networked application, and most of its security is the boring kind you already practice: network segmentation (the service sits with the other supervisory-level systems, never on the control network itself), least privilege (the gateway reads the historian; nothing writes toward a controller — Chapter 10 makes this a hard rule), authenticated clients, and logs that answer who asked what and when.

Three items are model-specific enough to name. **The injection surface** — Chapter 4's adversarial note operationalized: plant text is untrusted input; fencing, quoting contracts, and constrained outputs are the mitigations, and the residual risk is bounded by the human between verdicts and actions. **The exfiltration surface** — in any shape with an external hop, the prompt itself is an egress channel; clearance flags and boundary logging are the controls, and the air-gapped shape is the plant's statement that the channel does not exist. **The supply chain** — weights and engine builds come from named sources, verified by checksum at download and again at load; a model file is executable-adjacent content and deserves the same custody chain as firmware. None of this is exotic; all of it belongs in the same security review the plant already runs, using the same vocabulary, which is exactly how it will get

approved.

The container question

Someone will ask why this chapter says "unit file" instead of "container," and the answer is a default, not a doctrine. A floor deployment is one engine on one box with one large file: the isolation containers buy solves problems this shape does not have, while adding a runtime, an image registry, and a build pipeline to the parts list. Files, checksums, and an init system are the plant's native idiom — Chapter 9's recovery drills are written in it — and fewer layers is its own reliability feature. Containers earn their place at the department shape and above when a team already operates them fluently, or when one box must host several isolated gateways; even then, the weights file stays a mounted artifact with its own checksum, never baked into an image, so that model versioning remains Chapter 7's six-artifact discipline rather than an image tag. Use what your crew can fix at 2 AM. That rule outranks every architecture opinion in this section.

The configuration traps, with the scars attached

Serving configuration looks like a page of harmless flags. Some of them are load-bearing in ways the documentation undersells, and our lab's log is a small museum of the failure modes. The exhibits:

Precision of the attention cache. Quantizing the KV cache saves real memory and is usually safe — until a specific model dislikes it. Our production model, quantized to an 8-bit cache, produced corrupted output; the cache went back to 16-bit and stayed there, at measured memory cost, by standing rule [LAB: CLAUDE.md serving traps – KV-cache precision corrupting output]. The general lesson: cache precision is a *gated* change like any other, not a free flag. Flip it only with Chapter 6 watching.

Memory-mapping vs. loading. Engines can map the weights file from disk (fast start, gentler on RAM, first-touch latency) or load it whole (slow start, no surprises, but the file must fit). Forcing a full load of a file larger than memory does not politely fail — it takes the host down with it, a lesson our lab's hardware notes preserve in the imperative mood [LAB: CLAUDE.md serving traps – no-mmap on oversized files]. Know which mode each unit file requests, and why.

Layer placement on mixed hardware. When a model splits across GPU and CPU memory, *which* layers spill decides the speed — and some placement flags silently

interact with tensor-layout optimizations, producing configurations that run at a fraction of their potential until one line in the load log explains why (Chapter 9's read-the-load-log rule, which was earned on exactly this class of mystery). The protocol: after any placement change, read the load log's placement summary in full, then re-run the gate's throughput probe before calling it done [LAB: RESULTS-MATRIX \$F failed configs – n-cpu-moe 14 VRAM-OOM, ≥18 without --no-repack segfault, --no-mmap host-OOM >125 GB RAM; placement is a matrix you re-measure, not a flag you copy].

The restart that isn't clean. A serving process that dies mid-batch can leave the GPU in a state where the successor process fails to allocate. The watchdog's restart path must handle "restart the process" and "reset the device" as distinct rungs, and Chapter 9's function-probing health check is what distinguishes them: process up but no token back means climb the rung.

The integration seams

A model service earns its keep only where people already work, which means three unglamorous adapters, worth naming because their absence is the most common reason a technically sound deployment goes unused.

The HMI seam. The operator-facing panel gets a read-only card: the watcher's latest digest (Chapter 4), the current verdict for this line's active alarm, and nothing type-in-able. Operators consume; the interaction surface lives with maintenance.

The CMMS seam. The extractor writes *draft* work orders, flagged as machine-drafted, into the normal approval queue — never directly into the record. Chapter 5's confidence grades map onto the queue's priorities; the technician's accept-or-correct is the disposition that feeds Chapter 7.

The conversation seam. The engineers' ad-hoc questions go through a chat surface that is a thin skin over the same gateway — same contracts, same schemas underneath, same logging. The moment a "quick chat tool" bypasses the gateway, everything this book built — grounding, abstention arms, quote checks, corpus capture — silently stops applying to the traffic that is fastest growing. One gateway, many skins, no exceptions.

The rollout path: shadow, advisory, assisted

Whatever the shape, it goes live in stages, and the stages are the same everywhere because they are stages of *trust*, not of

technology.

Shadow mode first. The full pipeline runs on live traffic — rendering, verdicts, logging — and nobody sees the output but the gate. Two to four weeks of shadow answers, scored against what the humans actually did, is the cheapest large-scale evaluation you will ever run: production distribution, production load, zero production risk. Shadow mode also burns in the operational layer — the watchdogs, the log rotation, the thermal reality — while the stakes are nil. Most deployments discover their first three surprises here, which is the point [LAB: RESULTS-MATRIX \$G – production burn-in on promoted Q3-MTP: 12-min soak 86 requests, 0 errors, 100% mean acceptance, +2 MiB VRAM drift; correctness 3/3, concurrent dual-stream MTP, 28K-token long-context recall].

Advisory next. Outputs become visible, clearly badged as machine drafts, with Chapter 5's confidence grades and disposition buttons live. The crew's corrections start flowing (Chapter 7's corpus), the usefulness metrics start accumulating (Chapter 6's disposition stream), and the deployment earns or fails to earn its place in the daily rhythm. Hold here until the dispositions say the tool is being *used* — green gates with ignored output is Chapter 6's warning, not a graduation.

Assisted, at most. The ceiling of this book's ambition: machine drafts that humans approve, machine routing that humans can override, machine summaries that humans verify before acting. The stage that does not exist on this path is the one where the model acts on the process unmediated — not because the models will never be good enough, but because Chapter 10's checklist requires a person between a probabilistic component and a physical consequence, and this book means it. The rollout path's finish line is a tool the crew reaches for without being told to, inside a boundary everyone can state from memory. That is what "deployed" means here.

The deployment record

Every shape ships with a one-page record, kept where the runbooks live: the shape name; the parts list with versions and checksums; the unit file and their conditions; the health probes and their thresholds; the gate report that authorized this configuration; the spare's location and last restore test; and the names — who owns the service, who owns the gate, who signs the next change. One page. Chapter 9 already argued that the difference between a ninety-minute recovery and a twenty-five-minute one is a document somebody wrote; this is that document, written before the storm instead of after. The next chapter — the last — compresses this book's

whole argument into the checklist that page belongs to. Whatever shape you choose, choose it out loud: the record, the names, the boundary. A deployment nobody can describe is a deployment nobody can defend at budget time.

Chapter 9 — Surviving Reality

(draft v0, 2026-08-27 — written by Claude Fable 5, unverified. This chapter's evidence is unusually direct: our own lab's failure log, cited by date.)

Every chapter before this one assumed the computer stays on. This one is about the week that assumption failed twice.

Our lab runs language models the way this book proposes a plant should: continuously, on owned hardware, with training jobs, serving processes, and data pipelines sharing a building's worth of dependencies. In one August week that building lost power twice. The first outage cost about an hour and a half of recovery work. The second, two days later, cost about twenty-five minutes — not because it was gentler, but because the first crash had been treated as an engineering deliverable instead of a bad day [LAB: PROJECT-LOG 2026-08-22 and 2026-08-24 – power-loss recoveries #1 and #2]. The delta between those two numbers is this chapter. Everything in it was paid for.

A plant engineer will recognize the shape of what follows, because none of it is AI engineering. It is the ordinary discipline of keeping industrial systems restartable — applied to a stack that the AI industry, raised in data centers with generators, mostly ships without it.

What a power loss actually breaks

Walk the damage path of a hard power cut through a model-serving stack, because each stage is a design decision you can make ahead of time:

Storage is hurt first and lies about it. Filesystems with journaling recover to a consistent state by design; filesystems without it — including network-share formats mounted through compatibility bridges — can be silently damaged and need offline repair before they mount at all. Our first crash spent most of its ninety minutes here, on exactly one lesson: know, for every volume your stack touches, what happens to it when the power dies mid-write. The fix was boring and total — repairs to mount configuration so every data volume came back automatically and consistently. In the second crash, storage cost *zero* minutes [LAB: PROJECT-LOG 2026-08-24 – "the crash-#1 fixes held; zero filesystem work"]. Boring and total is the grade to aim for.

Services resurrect in the wrong order — or resurrect when told not to. Modern init systems restart what was running, which is what you want until it isn't. Our second

crash surfaced a subtle version: a service we had explicitly disabled came back anyway, because three *other* services declared it as a dependency, and dependency pulls ignore the disabled flag. It occupied most of a GPU's memory before anyone noticed, and the training jobs that were supposed to own that GPU found it taken. The durable fix was not a stronger "off switch" but a *condition*: the service's unit file now checks for a hold-marker file on disk and refuses to start while the marker exists — an interlock, in plant terms, rather than a request [LAB: PROJECT-LOG 2026-08-24 – dependency pull vs. disablement; condition-gated unit fix]. The general rule: on shared hardware, "stopped" enforced by intention decays; "stopped" enforced by a condition survives reboots, dependency graphs, and colleagues.

In-flight state is simply gone. Whatever the model was generating, whatever batch was mid-flight, whatever fine-tune step was between checkpoints — vanished. You do not protect in-flight state; you bound it, which is the next section.

Checkpoints are a cadence decision, not a feature

Long-running model work — a fine-tune, an index build, a corpus pack — survives power loss exactly as well as its checkpoint cadence, no better. Our training runs checkpoint every few thousand steps; when the first crash hit, the running jobs lost roughly three to four hours of progress each — annoying, bounded, and resumable to the step, with random state restored so the run continued as if unbroken. The week's two crashes cost about nine and a half GPU-hours of redone work in total, and the log entry closes with the sentence that matters: the cadence *bounds each loss* under four and a half hours [LAB: PROJECT-LOG 2026-08-24 – checkpoint cadence bounding recovery cost].

Translate the arithmetic to your floor. The cadence question is: how many hours of this work am I willing to redo? Divide by two for safety margin, checkpoint at that interval, and *verify a resume actually works* — a checkpoint you have never restored from is a hope, not a checkpoint. The same logic covers the humbler state a serving deployment accumulates: retrieval indexes, caches, configuration. If rebuilding it is fast, let it rebuild; if it is slow, it is a checkpointing customer too.

For pure serving — the model answering questions — the news is better and it is a genuine advantage of the small-model classes from Chapter 3: the "state" is a read-only weights file plus a process. Recovery is: mount storage, start service, load weights, health-check. Small

models load in seconds to low minutes, which makes the whole recovery a watchdog script rather than an operation.

Verify by artifact, not by appearance

The second crash's log records a near-miss that deserves its own section, because the class of error is universal.

During recovery, an operator checked whether the data-repacking jobs had survived by listing processes and pattern-matching the command names. Two matches came back; the conclusion "both packers are running" was one keystroke from being recorded. Both matches were the *listing command itself* — the search pattern matched its own invocation. The jobs were dead, and had the appearance been trusted, a large data pipeline would have sat idle indefinitely while dashboards showed green [LAB: PROJECT-LOG 2026-08-25 – self-matching process check near-miss]. The catch came from checking the *artifact* instead: the job's log file had not been written since before the crash. Mtime does not pattern-match itself.

The same log family records the inverse trap: a cleanup command that kills processes by name-pattern matched the operator's own shell — the pattern found itself — and cut the session out from under the recovery [LAB: CLAUDE.md hardware notes – pkill self-match trap]. Two incidents, one lesson, and it generalizes far beyond Linux: **status checks must observe what the work produces, not what the process table appears to contain.** A packer is alive if its output file grew recently. A serving model is alive if a health-check request returns a token. On a plant floor you already know this — you trust the flow meter, not the pump's power light — and the discipline transfers unchanged.

Read the log before trusting the plan

One more lab scar with direct floor application. A production model once served at a catastrophic fraction of its normal speed — the kind of number that triggers a tuning spree: flags, batch sizes, memory splits, none of it moving anything. The answer was a single line in the *load* log, printed at startup, showing one component had quietly loaded onto the wrong processor [LAB: PROJECT-LOG – the 2 tok/s mystery; one load-log line]. Minutes of reading would have saved the hours of tuning.

The rule we wrote afterward: **when a number is pathological — not merely low, but absurd — stop adjusting and start reading.** Pathological numbers are almost never tuning problems; they are configuration problems announcing themselves at boot in a log nobody

reads. Your deployment's startup log should be short enough to read and read on every deployment; Chapter 8's serving recipes print the facts that matter — device placement, precision, memory reserved — precisely so this rule is cheap to follow.

The UPS question, answered honestly

The reflex response to this chapter is "buy a battery." Do — and understand precisely what it buys. An uninterruptible supply sized for a serving box is not there to ride out the outage; plant outages outlast affordable batteries with ease. It is there to buy *minutes*, and minutes are only valuable if something spends them: a shutdown hook that sees the on-battery signal, stops accepting new requests, lets in-flight work drain, forces a final checkpoint, and unmounts storage cleanly. A UPS without that integration converts a hard crash at power loss into the identical hard crash twenty minutes later, at a time nobody predicted, with a false sense of security billed on top.

Brownouts deserve more fear than blackouts. A clean cut is the *easy* case — everything stops, everything restarts, this chapter's machinery handles it. A sag reboots some devices and not others, corrupts the states of equipment that half-survived, and produces the weird Tuesday where the model box is fine but the switch between it and the historian silently power-cycled and dropped its config. When a deployment misbehaves after "a power event," widen the suspect list to every box in the path before blaming the one with the AI on it; in our experience the exotic component is presumed guilty and is usually the innocent one.

And test the battery the way you test a checkpoint: by using it. A UPS that has never carried the load through a rehearsed shutdown is, like the unrestored checkpoint, folklore with a purchase order.

Watchdogs, and who watches them

Restart-on-failure is one line of configuration and everyone sets it. The engineering is in the layer above: deciding what "failure" means and noticing when restarting stops helping.

A serving process can be up, listening, and useless — weights half-loaded, memory exhausted by Chapter 3's KV-cache ceiling, or wedged in a state where every request times out. A process-level watchdog sees a running process and stays quiet. The health check that means something is end-to-end: send a real, tiny inference request on a timer; require a token back within a deadline; restart on misses. That single design choice — probe the *function*, not the

process — catches the entire family of alive-but-dead states, and it is the same artifact-not-appearance rule from earlier wearing a uniform.

Then bound the restarts. A service that crashes on startup will crash-loop forever at whatever cadence you allow, and a crash-looping service generates exactly the alert storm that trains a crew to ignore alerts. Back off between attempts, cap the attempts, and after the cap, *change the message*: "restarting" is routine noise; "restarted five times and stopped trying" is a page. The worst outcome of a monitoring design is not a missed failure — it is a crew that has learned the alarms are wallpaper. Plants know this as alarm management; the AI box gets no exemption from it.

Log the boring successes, too. When the health probe passes, a timestamped line lands in the artifact trail — which means the *absence* of that line is itself detectable by the next layer up. Silence, as our monitoring rules put it, must never be mistakable for health.

Heat: the failure that arrives on schedule

Power loss is dramatic; heat is patient. A GPU in a sealed cabinet on a mezzanine in August does not crash — it *throttles*, quietly trading speed for temperature, and your deployment's response times drift upward with no error message anywhere. Our lab learned to treat thermal configuration as a first-class deployment parameter after measuring how much performance a power-and-cooling ceiling actually costs on sustained load — and, more usefully, that the right power cap costs far less than the wrong airflow [LAB: MAXQ-THERMAL 2026-08-06 – power-cap vs. throughput measurements].

The floor rules that fall out: record the box's sustained (not burst) throughput at commissioning, in summer conditions if you can get them; alert on *sustained deviation from that baseline*, not just on temperature thresholds; and treat a slow drift in response times as a maintenance signal like any other vibration trend. The failure mode is not "the AI got worse." It is dust on a filter, and the fix is a shop vacuum, not a retraining run.

The spare on the shelf

Plants keep spare drives, spare cards, spare pumps. A model deployment is the rare computer system where the spare-parts mentality transfers almost perfectly, because Chapter 3's operational virtues made the system *copyable*: the entire deployed intelligence is a weights file with a checksum, a configuration directory, and a service definition.

A cold standby is therefore not a project — it is a second box with the same three things on it, health-checked monthly by the same end-to-end probe, powered off the rest of the time. When the primary dies, recovery is a cable and a DNS entry.

Notice what makes this possible: everything that matters is a *file*. No license server to reactivate, no cloud enrollment to re-authenticate, no vendor account whose owner left the company. The restore test is the same as the deployment test. Run it before you need it — the spare that has never served a request is one more piece of folklore — and version the spare's contents in lockstep with the primary, because a standby carrying last quarter's model answers last quarter's questions. The gap between "we have a spare" and "we have a *tested* spare at the current version" is exactly the gap between this chapter's two crash durations, wearing different clothes.

The morning-after report

The practice that converted our ninety-minute crash into a twenty-five-minute crash was not a technology. It was the report written the same day: what failed, in what order, what the recovery actually required minute by minute, which fixes would have prevented each minute, and — the part most postmortems omit — which *checks gave misleading answers* during the recovery. The self-matching process check earned its place in this chapter because a same-day report recorded it as a near-miss instead of letting it evaporate into "anyway, it worked out."

Two crashes make a trend line only if the first one produced a document. Keep the reports blameless (the interesting failures are systemic, and a crew that fears the report hides the timeline), keep them short enough to be written the same day, and end each with work orders, not recommendations. "Consider improving mount reliability" fixed nothing; the line item "repair fstab entries for all three data volumes" is why crash two skipped storage entirely. The report is the mechanism by which an outage becomes an asset; skip it, and you have simply paid for the same lesson twice at full price.

The recovery drill

Assemble the sections into the procedure this chapter exists to leave behind:

1. **Enumerate volumes; know each one's power-loss behavior.** Journalled, auto-mounted, repair procedure written down. The right time to learn a filesystem's failure mode is never during the failure.

2. **Make every service's restart policy explicit** — including restart *order* and the conditions under which something must NOT start. Interlocks by marker file beat intentions by checklist.
3. **Set checkpoint cadence by redo-tolerance**, and rehearse one restore per cadence change. Unverified restores are folklore.
4. **Write artifact-based health checks** for everything that matters: output freshness, health endpoints, token generation — never process-table appearances.
5. **Time a full cold recovery, on purpose, quarterly**. Ours went from ninety minutes to twenty-five because the first crash's report became a work order. The drill is how you buy that improvement without needing the storm.

None of this is glamorous, which is the point. The AI parts of this book — grounding, abstention, evaluation — decide whether the system is worth trusting. This chapter decides whether it is *there* on the morning the substation hiccups. A model that is brilliant but absent loses to one that is adequate and running; the plant floor has always graded on attendance, and it is right to. Build for the morning after the storm, and the storm becomes a line item in a report instead of a story people tell about the time the plant tried AI — because the difference between those two outcomes was never the model. It was the fstab, the marker file, and the report somebody wrote the same day.

Chapter 10 — The Honest Deployment Checklist

(draft v0, 2026-08-27 — written by Claude Fable 5, unverified.)

Every discipline that keeps people safe around machines eventually compresses itself into a checklist — not because the discipline is simple, but because the moment of decision is busy, and busy moments need the discipline pre-decided. This chapter is that compression for everything this book has argued. It is written to be printed, argued over, adapted, and signed; the prose around each check explains what the one-line version is load-bearing for, because a checklist whose reasons have been forgotten decays into ritual within a year.

A word on what "honest" means in the title. Not honest as a virtue — honest as an engineering property, the one this book has been assembling since Chapter 2: a system whose answers carry their evidence, whose refusals name their gaps, whose numbers carry their error bars, and whose failures are written down where the next engineer will find them. Each check below defends one piece of that property. Strike any of them and the system still runs; it just stops being able to tell you the truth about itself, which on a plant floor is the beginning of every bad story.

The ten checks

One: the evaluation existed before the deployment did. If the gate (Chapter 6) was built after the model was chosen, the model chose the gate. The order of construction is itself evidence: a deployment that can show a dated gate report predating its go-live has proof it was measured into existence rather than demoed into it. If you inherit a deployment without one, building the gate retroactively is the first work order — and running the incumbent through it is often illuminating in both directions.

Two: the model was sized by measured failure, not by demo. Chapter 3's selection rule, auditable in one question: what did the smaller model fail at, specifically, and is the failure in the gate set? If nobody can produce the failed cases, the size was chosen by instinct, and instinct in this field is calibrated by cloud demonstrations that have nothing to do with your fault tables.

Three: the rendering is honest about its defects. Chapter 4's rule — every known data defect is a visible label — checked by inspection: pick a flatlined sensor, a historian

gap, an unmapped tag, and read what the model actually receives. If the pipeline papers over any of them, every downstream honesty claim is decorative, because the model cannot decline what it cannot see missing.

Four: abstention has arms, training, and numbers. The schema carries the grades of no as first-class values; the training set held answerable controls beside the refusals; and the gate reports all four quadrants, with the confident-fabrication rate carrying the tightest threshold. Chapter 5 in one sentence: a model that cannot say "I don't know" fails the gate regardless of accuracy — and a deployment that cannot show its four quadrants does not actually know whether its model can.

Five: every published number has an error bar and a date. The ± 10 -point lesson, standing. Any score quoted without its spread and its noise floor is a demo statistic wearing engineering clothes, and any score without a date will be quoted long after drift has invalidated it. This check extends to the vendor across the table: ask for ranges, watch the reaction, learn more from the reaction than the ranges.

Six: the corpus has clearance flags older than the training run. Chapter 7's capture hygiene, checked at the manifest: every record that trained carries a clearance decision made at write time, the scrub pass ran before the shard was cut, and the gate set is provably disjoint from training. A model whose corpus cannot answer "whose text is this and who said we could" is a liability wearing a version number.

Seven: recovery is rehearsed, not planned. Chapter 9 compressed: the cold-start has been timed this quarter, the checkpoint restore has been executed (not reviewed — executed), the spare has served a real request at the current version, and the health checks probe function rather than process. The evidence is a dated drill log. "We have a recovery plan" without drill dates is the sentence that precedes every ninety-minute outage.

Eight: the boundary is stated and physical. The deployment's charter says, in writing a technician can quote: verdicts advise, humans act, and nothing the model emits reaches a controller, a setpoint, or a safety function through any path, including the informal ones. Chapter 8's rollout ladder ends at assisted on purpose. This is the check that does not bend for a good quarter or an impressive pilot — the one line where this book trades ambition for the right to make every other claim.

Nine: the paper trail exists and is current. The deployment record (Chapter 8), the run manifests (Chapter

7's six artifacts), the gate reports in chronological file, the incident and drill logs (Chapter 9), the errata. One binder — physical or not — that answers the auditor's question, the insurer's question, and the 2 AM question with documents instead of memories. If assembling it would take more than an hour, it does not exist yet.

Ten: the crew owns it. The dispositions flow because the crew believes the corrections improve their tool, not their surveillance file. The abstention histogram is read in the maintenance meeting as a documentation work-list. Someone whose name is on the record can say what the system is for, what it must never do, and who to call. Usage metrics live beside accuracy metrics, and a green gate with an ignored tool is treated as the failure it is. Every previous check is machinery; this one is whether the machinery is alive.

The never list

Ten checks earn a deployment its floor; four sentences bound it permanently. Never let model output reach a control action without a human decision in between. Never deploy a model you cannot roll back with a file copy. Never train on data you could not show its authors. Never publish a number about the system that the system's own gate did not produce. These four are not policies to balance against throughput — they are the definition of the practice this book teaches, and a deployment that breaks one has left the book's coverage, whatever else it has achieved.

The checklist as a procurement instrument

The ten checks were written for systems you build, but they convert directly into diligence for systems you buy, and the conversion is worth spelling out because a purchased deployment skips none of the obligations — it only relocates them.

For each check, the vendor question and the shape of a good answer. One: "show me the evaluation you ran on data like ours, with its date" — a good vendor produces a methodology and offers to run it on your cases; a poor one produces a leaderboard. Two: "why this model size and not the one below it" — good answers cite failed cases; poor ones cite roadmaps. Three: "what does your pipeline show the model when a sensor is dead" — the vendor who understands the question is rare and worth shortlisting for that alone. Four: "show me the four quadrants" — and watch whether abstention is a concept they have metrics for or a feature they promise to look into. Five: "what is the

run-to-run spread on that number" — Chapter 6 already taught you what the reaction means. Six: "whose data trained this, under what terms" — an answer that starts with a pause is an answer. Seven and nine: "walk me through your last recovery drill and show me the record you would hand our auditor." Eight: "describe the paths by which your system's output could reach a control action" — the only acceptable answer enumerates them and shows the human in each. Ten is yours to keep, not theirs to sell: no vendor can supply a crew that trusts the tool.

A vendor who survives all ten exists and deserves the business. A vendor who bristles at them has told you, at proposal stage and free of charge, exactly what the relationship will be like at incident stage.

The first ninety days

The checklist compresses the book; the calendar decompresses it into a plan a maintenance department can actually run.

Weeks one and two belong to the gate and nothing else: the single question chosen, the schema frozen, the hundred cases labeled in the two afternoons Chapter 6 promised, the runner built, the noise floor measured five times. Resist the demo urge; a deployment that starts with its instrument never has to retrofit its honesty.

Weeks three through six are Chapter 4's plumbing against the gate's baseline: decode tables reused, renderings built and defect-labeled, contracts written with the abstention arms in place, and the whole pipeline in shadow mode against live traffic — scored nightly, shown to nobody. This is where the surprises surface: the timestamp formats, the tokenizer-shattered tag names, the historian gap nobody mentioned. Each one becomes a gate case the moment it is understood.

Weeks seven through ten open the advisory stage: badged drafts, disposition buttons, the crew briefed on what the tool is for and — with equal clarity — what it is never allowed to do (check eight, said out loud, early, by someone whose name is on it). The dispositions begin accumulating into Chapter 7's corpus; the abstention histogram gets its first review in the maintenance meeting.

Weeks eleven through thirteen buy down the operational risk: the recovery drill run and timed, the spare restored and exercised, the deployment record assembled while everything is fresh, the signature page signed. Somewhere in this window the gate runs its first monthly cycle on schedule rather than on demand — the moment the deployment stops

being a project and starts being a system.

Ninety days, one line, one question, no heroics. The plants that fail at this do not fail for lack of talent; they fail by attempting month three's ambitions in week two, on the strength of a demo, without an instrument. The calendar is the checklist's enforcement mechanism: nothing on it requires believing anyone — only measuring.

Keeping the checklist alive

Checklists die two deaths, and both are preventable. The first is ritual decay: the checks get initialed without being performed, because the system has been fine and the meeting is long. The antidote is the same one aviation found — tie each check to an artifact that cannot be initialed into existence. A drill has a timestamp; a gate report has ranges; a signature page has a date. Auditing the artifacts quarterly takes an hour and converts the checklist from a promise into a record.

The second death is the waiver: the one check suspended "temporarily" for a good reason — the pilot that would close a loop just this once, the training run on data whose flags were almost sorted. The never list exists precisely because the waived check is how disciplined systems degrade; each item on it was chosen because its first violation looks reasonable at the time. The rule that keeps waivers honest is borrowed from Chapter 9's morning-after report: a waiver can only be granted in writing, with a name, a scope, and an expiry — and an unexpired waiver appears in every gate report until it closes. Waivers that must be signed and republished monthly have a way of not being requested.

Signing it

A checklist nobody signs is a poster. This one takes three signatures, renewed on a cadence: the engineer who owns the gate, attesting the numbers; the operations owner, attesting the drills and the record; and the plant's responsible manager, attesting the boundary. Re-signature triggers are the same as the gate's: model change, schema change, pipeline change, and the calendar. The signature page lives in the front of the binder from check nine, which means the binder's first page is three people's names — and that is the correct first page, because every instrument in this book ultimately reduces to people willing to put their names beside its claims.

That is also, the reader may have noticed, how this book itself is built. Its models are named on the cover; its claims carry citations into a public lab record; its verification is a

named human's signature; its review trail publishes with it; and where its instruments failed, the retractions are printed in the text. The checklist you just read is the one its publisher runs on its own books. We wrote the successor chapter to those 518 silent pages the only way that would have been worth doing — under the same discipline we are asking of you.

What this edition owes you

A book that demands honesty from deployments owes a closing accounting of its own gaps, so here is this edition's, in plain text. A number of claims in these chapters still carry their bracketed markers pointing at lab entries not yet attached; none ships in a verified edition, and the markers are visible in this draft precisely so that reviewers can hold us to each one. The war stories from real plant floors — the voice this book's verifier brings from years among the machines this book is about — are represented but not yet written; they arrive by interview, not invention, because a fabricated anecdote in a book about honest instruments would be a foundation crack, and we would rather show you the empty section than fill it wrong. The evaluation benchmark this book references has its own public history, including its retractions, and readers are owed the link rather than a summary flattering to us. And the field itself is moving: the tier capabilities in Chapter 3 are dated measurements of a moving frontier, which is why they carry dates and why the gate — not this book — is your standing authority on what your model can do this quarter.

Those debts are recorded in the same spirit as check nine's binder: visibly, dated, with names attached, in the review trail that publishes alongside this book. If you find a claim here that its citation does not support, the errata process on the provenance page is not a formality — it is the mechanism by which this book remains the thing it claims to be, and we ask you to use it.

Where this leaves you

Start smaller than feels ambitious: one question, one schema, one hundred labeled cases, one box beside one line. Run shadow mode until the surprises stop. Let the gate, not the vendor, tell you when to climb. Feed the corpus, mind the flags, drill the recovery, read the load log. And when a colleague from another plant asks what you are running, hand them the deployment record and the gate report instead of a demo — because the demo is the genre this whole field needs to grow out of, and the record is what growing out of it looks like.

The machines have been describing themselves to computers for forty years. The computers can finally read. What happens next on your floor depends less on the models than on the honesty of the instruments you build around them — and that part, every line of it, is in your hands. Good luck out there — and write your own log entries the same day you earn them.

Back Matter

Glossary

- **abstention:** trained model behavior of declining to answer when evidence is insufficient
- **acceptance test:** a fixed evaluation a model configuration must pass before deployment
- **accuracy floor:** the minimum measured performance a deployment gate requires
- **batch packing:** serving multiple requests in one inference batch; a source of run-to-run nondeterminism
- **checkpoint:** a saved snapshot of model or training state, used for recovery
- **CMMS:** computerized maintenance management system; system of record for work orders
- **constrained decoding:** restricting generation so only tokens legal under a grammar can be emitted
- **context window:** the bounded buffer of text a model reads verbatim while generating
- **distillation:** training a small model on a larger model's outputs to transfer capability
- **edge box:** compute device deployed near the machines, outside the data center
- **enum decode:** constrained decoding where the output must be one of a fixed set of values
- **error bar:** the measured spread of a benchmark score across repeated runs
- **evaluation gate:** automated test rig that decides whether a model change may ship
- **fault code:** a machine's structured identifier for a specific failure condition
- **fine-tuning:** continuing a model's training on domain data to specialize it
- **grammar:** a formal specification of legal output structure used in constrained decoding
- **grounding:** placing trusted documents in the context window and requiring answers from them
- **hallucination:** fluent, specific, wrong output produced by normal model operation
- **historian:** time-series database recording plant process values (tags) over time
- **inference:** running a trained model to produce output (as opposed to training it)
- **inference engine:** software that loads model weights and executes generation
- **interlock:** a control condition that must hold before an operation is permitted
- **KV cache:** inference memory holding attention state for the current context

- ladder logic: graphical PLC programming language scanned top to bottom
- latency: time from request to first (or complete) response
- local model: a model whose weights run on hardware the operator owns
- lossy compression: storage that preserves regularities but not exact values; how weights hold training text
- MoE (mixture of experts): architecture activating a subset of weights per token
- NPSH: net positive suction head; pump property referenced in cavitation analysis
- parameter: one learned weight; model size is quoted in parameters (millions/billions)
- PLC: programmable logic controller; the computer that runs a machine's control program
- protocol frame: one structured unit of an industrial communication protocol
- quantization: storing model weights at reduced numeric precision to shrink memory
- register: an addressed data location on an industrial device readable over a protocol
- retraction: published withdrawal of a prior claim, with the reason stated
- sampling: the policy for choosing the next token from the model's probability distribution
- schema: the required structure of an output (fields, types, enumerations)
- serving: running an inference engine as a persistent network service
- tag: a named point in a historian or SCADA system (e.g. a temperature reading)
- temperature: sampling parameter controlling randomness; zero picks the most probable token
- tier: a named model size class in this book's ladder (from sub-billion to tens of billions)
- token: the sub-word unit a model reads and writes
- tokenizer: the fixed procedure that splits text into tokens
- unified memory: hardware design where CPU and GPU share one memory pool
- VFD: variable frequency drive; motor controller and a rich source of fault codes
- weights: the learned numbers that constitute a trained model
- working memory: informal term for the context window, by analogy to human cognition

Lab citation convention

In-text markers of the form [LAB: RESULTS-MATRIX \$C] or [LAB: PROJECT-LOG 2026-08-03] resolve into the RogerAI Labs lab record: RESULTS-MATRIX sections hold configuration tables with measurements; PROJECT-LOG entries are dated experiment narratives. Claims without a [LAB:] marker are

labeled unmeasured in the prose.

References

- O'AILLY platform machinery (gates, standards, review pipeline): <https://github.com/oailly-press/platform>
- llama.cpp — open-source inference engine used throughout the lab work: <https://github.com/ggml-org/llama.cpp>
- C2PA content provenance standard: <https://c2pa.org/>
- Authors Guild, "AI Best Practices for Authors" (disclosure landscape): <https://authorsguild.org/resource/ai-best-practices-for-authors/>
- RogerAI Labs lab record: RESULTS-MATRIX.md / PROJECT-LOG.md — in-text [LAB:] markers name the section or dated entry used.

(References grow with the chapters; every citation must resolve at Pass 1.)