Welcome

Build a small QA model. Understand every piece of it.

The best way to understand a system is to build a small version of it yourself. That is what this book is: a guided build of a **local, QA-specialized language model** — one that reads a failing Playwright test and proposes a grounded fix — from first principles to a green re-run, entirely on your own machine.

You will not need a cloud account, an API key, or anyone's permission. CI logs are proprietary, so everything here runs on-device: the reading, the data, the fine-tune, and the test execution. When you finish, you will have three things — an understanding of how a language model actually works, a small model you shaped yourself, and at least one real automation test that your model helped you diagnose and repair.

> This book is the companion text to the **Scholarly QA Model Builder** app. Everything interactive here mirrors what the app does live: the numbers in the charts come from a real MLX LoRA training run, and the failures you'll study were harvested from 260 real test executions on two live sites. Nothing is invented for the sake of a tidy example.

The book is linear on purpose, and the ramp is deliberate. **Part I teaches the AI itself, from zero** — what a model is, how it reads, how it learns — using everyday examples before any test-automation ones; it assumes no AI background. **Part II turns the lens on QA**: failures, evidence, grounding. **Part III you build** — data, fine-tune, chat. **Part IV you apply and lead** — repair a real test, then roll the practice out to a team. Field exercises along the way turn reading into practice. Use the arrows above, or `←` and `→` on your keyboard.

> **Who this is for.** Three readers, one box opened. **New QA engineers** — read linearly; if you can read a stack trace you have every prerequisite. **Seasoned SDETs** on Playwright, Cypress, or Appium — the build and repair chapters are your hands-on core. **QA leaders** — follow the "Leader's lens" notes in every chapter, then land on the rollout chapter: the whole leadership track takes an evening.

LEADER'S LENS

If you lead a team, read this book the way you'd read a vendor's architecture doc before a seven-figure commitment — except this one lets you rebuild the product yourself. Two hours here buys you the vocabulary to challenge every AI-testing pitch you'll hear this year.

Foreword: from the author's chair

Why a test engineering leader spent his nights building a language model.

I have spent more than twenty years in quality engineering — running test organizations at places like Tinder, Amazon's Ring, Spokeo, and Live Nation, and today as a Sr. Manager of Test Engineering at Motorola Solutions. In that time I have watched three waves of tooling promise to end flaky tests forever. Record-and-playback didn't. Codeless automation didn't. The current wave — language models — actually might change the daily texture of this job, and that is exactly why I refuse to treat it as magic.

Every durable skill I've seen in great QA engineers comes down to the same move: *open the box*. Read the stack trace instead of re-running the suite. Read the DOM instead of blaming the framework. This book is that move, applied to the model itself. We will open the box far enough that when a model hands you a fix, you'll know what it saw, what it couldn't see, and whether to trust it — because you'll have built a small one with your own hands.

Three readers are welcome here, and each gets a different payoff. **If you're new to QA — or new to AI**, read linearly: Part I assumes no AI background and almost no automation background, and every concept arrives with an everyday example before a QA one. **If you're a seasoned SDET**, the build chapters are yours — real data, real fine-tune, real red-to-green loop, all on your laptop. **If you lead a quality org**, watch for the "Leader's lens" notes threaded through every chapter: they translate each concept into the decisions you actually own — budgets, gates, trust, and rollout order.

One promise, carried over from how I try to run teams: the numbers in this book are real, and where they are not yet human-validated, the text says so out loud. A benchmark you can't interrogate is a rumor. We don't ship rumors.

— **Suneet Malhotra**
Sr. Manager, Test Engineering · Motorola Solutions
suneetmalhotra.com

What is AI, really?

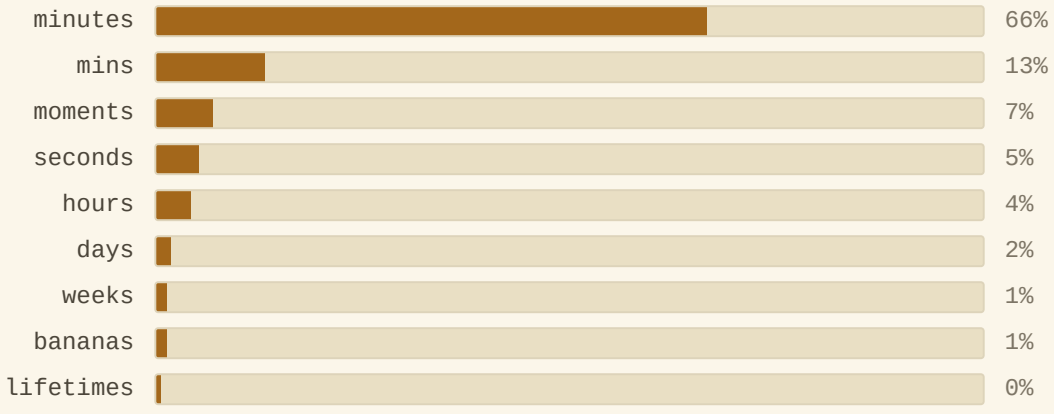
No QA knowledge needed for this chapter. No AI knowledge either — that's the point.

Strip away the branding and the doom headlines, and the "AI" everyone suddenly works next to is one specific thing: a **language model** — a program that reads some text and predicts what text comes next. That's it. Not a mind, not a database, not a search engine. A prediction machine for words, grown very large.

You already use a tiny one every day: the word suggestions above your phone keyboard. Type "I'll be there in five" and your phone offers *minutes* — not because it understands lateness, but because in the mountains of text it learned from, that's what usually comes next. A modern model like the ones in this book is that same idea with billions of learned settings instead of a few thousand. Try being the model yourself:

FIG. 01 · PREDICT THE NEXT WORD — NO TECH REQUIRED temperature 0.80

"I'll be there in five ▌"



temperature ●——— SAMPLE

Every candidate word gets a probability; the model draws one. The temperature slider is a real control you'll meet again: low keeps it predictable ("minutes", every time), high lets unlikely words win. Slide it to 2.0 and press Sample a few times — congratulations, you've just seen why AI sometimes says something weird.

Three terms, sorted once so the rest of the book never has to: **AI** is the umbrella marketing word for machines doing things that look smart. **Machine learning** is the actual method — programs that improve from data instead of hand-written rules. A

large language model (LLM) is a machine-learning program trained on text to do next-word prediction. ChatGPT, Claude, Gemini, and the small local models in this book are all LLMs — different brands of the same species.

FIG. 02 · THE NESTING DOLLS — where LLMs sit

TERM	WHAT IT ACTUALLY MEANS	EXAMPLE
AI	umbrella word — machines doing "smart-looking" things	chess engines, spam filters, LLMs
Machine learning	learn behavior from data, not hand-written rules	your inbox learning what's spam
Neural network	the flexible math structure that does the learning	the layers you'll open in Part I
LLM	**a neural network trained on text to predict the next word**	**ChatGPT, Claude, gemma — and yours, by Part III**

When a vendor says "our AI understands your tests," translate it: "our LLM predicts plausible next words about your tests." The translation isn't cynical — prediction this good is genuinely powerful — but it keeps you asking the right questions.

Two consequences follow from "it predicts words," and they explain almost every AI behavior that will ever puzzle you. First, the model's knowledge is *whatever was in the text it learned from* — impressively broad, invisibly dated, and silent about your private world. Second, it always produces *plausible* text, which is not the same as *true* text. A prediction machine asked a question it can't answer doesn't say "error" — it predicts what an answer would look like. Fluent, confident, and sometimes wrong: you now understand "hallucination" better than most of the industry's marketing departments.

FIELD EXERCISE

Before the next chapter: explain to a colleague, in three sentences and zero jargon, what an LLM does. Then find two next-word predictors you already use (phone keyboard, email autocomplete, search suggestions) and watch them work for a minute. The rest of this book is just these machines, opened up.

LEADER'S LENS

Run this chapter as a 30-minute team session before any AI tooling rollout. In my experience the single biggest source of misuse isn't bad tools — it's engineers holding one of two wrong models in their head: "it's Google" (so they trust retrieval it never did) or "it's magic" (so they never question output). "It predicts the next word" resets both.

Takeaway. AI, for our purposes, means a large language model: a prediction machine for text. Everything it does well and everything it does badly follows from that one sentence — and by the end of Part I you'll have watched every internal step of that prediction happen.

The red pipeline

It's 8:47 on a Tuesday and the pipeline is red again.

Not your code. Not the app. A test — the one that clicks the checkout button on saucedemo. You didn't touch it. Nobody touched it. And yet:

```
1) [chromium] › checkout.spec.ts:42:5 › completes purchase
   Error: locator.click: Timeout 30000ms exceeded.
   Call log:
     - waiting for locator('.btn.primary')
       - locator resolved to 0 elements
   at checkout.spec.ts:42:14
```

You stare at it. Thirty seconds of waiting, then nothing. The button is *right there* when you open the site by hand. Somewhere in your team's chat, someone is already typing "can you look at the flaky checkout test again."

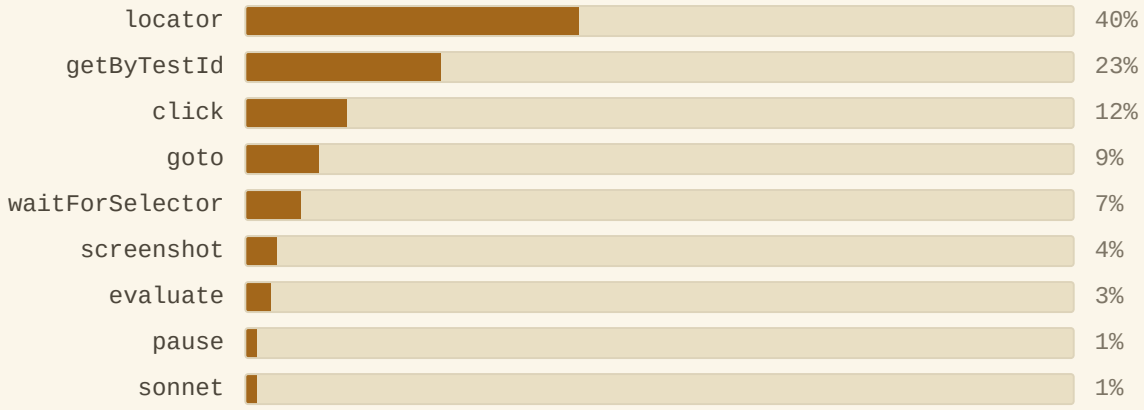
This book exists because a language model can read that failure and, in a few seconds, tell you what a tired human takes twenty minutes to see: the front-end renamed a class, the selector went stale, and the one-line fix is to target the `data-test` attribute instead. But it can only do that reliably if *you* understand what the model is — what it sees, what it can't see, and when it's guessing.

Under all the ceremony, a language model does exactly one thing: given some text, it predicts what token comes next. That's the whole trick. Every diagnosis, every suggested fix, every confident paragraph is that one move, repeated. You played this game with everyday words in the opening chapter; here is the same machine mid-way through writing a Playwright line, deciding what follows `await page.`

FIG. 03 · PREDICT THE NEXT TOKEN

temperature 0.80

```
await page.
```



temperature

SAMPLE

The model assigns a probability to every candidate continuation. Low temperature sharpens the distribution toward `locator` — the safe pick; high temperature flattens it and stranger completions start winning. Press Sample a few times at each extreme.

Everything you learned in Part I — tokenization, embeddings, attention, sampling — exists to make that one prediction good. And "good," for us, has a narrow, testable meaning: given a failing test's error and the relevant evidence, the next tokens should form a **grounded diagnosis with a citation**, or the honest admission that there isn't enough evidence to give one.

> **Takeaway.** A language model predicts the next token; nothing more. The craft is in shaping what it has read — and what you show it — so that the next token is the start of a correct, checkable answer about *your* failing test.

> LEADER'S LENS
>
> "It just predicts the next token" is the single most useful sentence to carry into procurement meetings. Every capability claim a vendor makes must reduce to it — and the ones that can't are marketing.

What is a QA model?

A general model answers everything. Yours will answer one thing well.

A general-purpose LLM will happily write you a sonnet, a SQL query, and a wedding toast. A **QA model**, as this book uses the term, is narrower and more useful: a model shaped to do one job — given a failing automation test's error output and the relevant docs, produce a **diagnosis and a fix that are grounded in evidence**, with a citation — and to **abstain** when the evidence doesn't support an answer.

That last clause is not a footnote; it is the design center. In test automation, a confident wrong fix is worse than no fix. You apply it, push, and wait through a full CI cycle before learning it was a guess. A model that says `INSUFFICIENT_EVIDENCE` and asks for the trace costs you one sentence. A model that invents a plausible cause costs you the afternoon.

FIG. 04 · WHAT THE MODEL ACTUALLY SEES — no browser · no app · text only

✓ sees the error line · the call log · DOM you paste · your stated intent

✗ never sees your app · the browser · the network · what the test *meant* to do

The model has never opened your app. It cannot click the button and watch what happens. It reads text, and only text — which means the quality of its answer is capped by the quality of what you hand it.

Intent is the piece engineers forget. "Completes purchase" is a title, not a specification. If you don't say that `.btn.primary` used to be the green *Checkout* button in the cart footer, the model is guessing. Good models guess well. But a diagnosis grounded in your intent beats a clever guess every time.

So the model we build will be judged on four axes, and you'll see all four again in the evaluation chapter: **answer correctness**, **citation precision** (does it point at real evidence?), **abstention accuracy** (does it refuse when it should?), and **hallucination risk** (how often does it invent?). Capability is nice. Trustworthiness is the product.

The trap to avoid from day one. A model that always produces a fix *feels* more helpful and is more dangerous. Reward the tool — and write the prompts — that treat "not enough evidence, send me the trace" as a first-class answer. Abstention is a feature.

LEADER'S LENS

Write abstention into your acceptance criteria before you pilot any AI triage tool: "the system must decline to answer when evidence is insufficient, and we will measure how often it declines correctly." Vendors optimized for demo-day confidence fail this line item instantly — which is the point.

Before transformers

Sixty years of teaching machines to finish a sentence — and why it kept failing.

Next-token prediction is not a new idea. The oldest usable version is the **n-gram model**: count, in a big pile of text, how often each word follows the previous one or two, and predict by lookup. No neurons, no training loop — just counting. And on narrow text, counting gets you surprisingly far. Try it: the widget below has learned from a small corpus of CI log lines, and nothing else.

FIG. 05 · A COUNTING MACHINE READS YOUR LOGS n-gram model · trained on CI lines in your browser

context length: 2 words back | GENERATE A LOG LINE

```
test passed user can log in landed on inventory page after click on checkout
button resolved to 0
```

With one word of context the output drifts mid-line — the model forgets what kind of sentence it started. With two, lines hold together longer. That's the whole n-gram story: coherence lasts exactly as long as the lookup window, and not one token longer.

The wall is obvious once you've hit it. A CI failure's meaning routinely spans forty lines — the selector on line 3, the DOM evidence on line 41. An n-gram model with a two-word window can never connect them, and widening the window explodes the table: every extra word of context multiplies the rows you need to count. By the 1990s the field's answer was the **recurrent neural network** — read tokens one at a time, left to right, carrying a compressed summary of everything so far. In principle, unlimited memory. In practice, a game of telephone: by the time an RNN has read a long trace, the beginning has been squeezed through so many updates that it has effectively evaporated. Long-range links — *this timeout relates to that selector* — are exactly what got lost.

The 2017 transformer removed the bottleneck with a blunt idea: stop compressing. Let every token look *directly* at every other token and decide for itself what matters — the attention mechanism you'll dissect over the next two chapters — and the reason a model can bind an error line to evidence forty lines away. The cost is quadratic work as the

window grows; the win is that distance stopped mattering. Almost everything since — bigger windows, faster attention — is engineering against that cost.

UNDER THE HOOD

The lineage in one line each. N-grams (1950s–2000s): probability by counting, memory = window. RNNs and their gated cousins like LSTMs (1990s–2017): probability by a running summary vector, memory long in theory and short in practice, and inherently sequential — you can't parallelize reading. Transformers (2017–): probability by direct token-to-token attention, memory = the whole context window, and fully parallel during training, which is what made scaling to today's sizes economical.

The widget above is a real n-gram model — it counts word pairs from its corpus at page load and samples from the counts. What it lacks isn't data; it's any notion of similarity. To an n-gram, *timeout* and *Timeout* are strangers. Embeddings fixed that, and they're a chapter back.

LEADER'S LENS

Vendors will pitch you "AI test healing" that is, underneath, closer to the counting machine above than to a transformer — brittle string patterns with a glossy demo. The question that separates them in a sales call: *can it connect evidence across the whole failure, or only react to the line in front of it?* Now you know why that question works.

Takeaway. Every era answered "what comes next?" with a different memory: n-grams remembered a window, RNNs remembered a summary, transformers remember everything in the context — which is why context, not cleverness, is what you manage when you use one on a failing test.

Tokenizing a CI log

Before a model can read your failure, it has to chop it up.

Time to open the box. The first surprise inside: models don't see characters, and they don't see words. They see **tokens** — sub-word chunks, typically three to five characters of English, learned by a compression algorithm from mountains of text. Common words get to be a single token. Rare ones get split. And CI logs are *full* of rare ones: selectors, hex ids, millisecond counts, file paths.

Paste a line from a real failure below and watch how it breaks apart. Each colored chip is one token — one unit of the model's attention.

FIG. 06 · CI-LOG TOKENIZER 21 tokens · 83 chars

```
Error: locator.click: Timeout 30000ms exceeded. waiting for
locator('.btn.primary')
```

`Error` `:` `locator` `.` `click` `:` `Timeout` `3000` `0` `ms` `exceeded` `.` `waiting` `for` `locator` `('.` `btn` `.` `prim` `ary` `')`

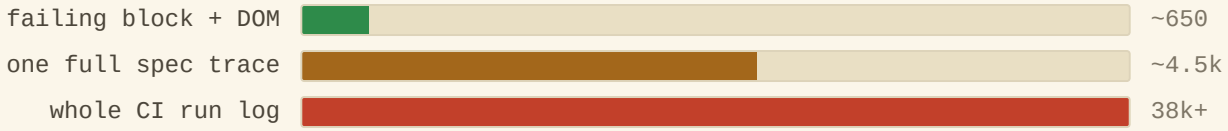
Notice how `Timeout` survives whole while `30000ms` shatters into pieces, and how the selector `.btn.primary` costs more tokens than the English around it. Log text is expensive text.

FIELD EXERCISE

Paste a real failing line from your own suite (or any error message you can find) into the tokenizer above. Count the tokens. Find the single most token-expensive part — it will almost always be a selector, id, or path. That instinct for "expensive text" is the first practical AI skill a QA engineer needs.

Why should a QA engineer care? Because of the **context window**. A small local model might fit 8,000 tokens — and a single failed run with full browser traces can be tens of thousands. Everything you paste competes for the model's attention. Paste the whole log and the one line that matters drowns.

FIG. 07 · WHAT FITS IN THE WINDOW 8,192-token budget



The whole-run log doesn't fit at all — it would be silently truncated. The curated failing block leaves room for the model to think. ***You are the filter. Filter first.***

UNDER THE HOOD

Real tokenizers use *byte-pair encoding*: start from bytes, repeatedly merge the most frequent adjacent pair, and keep a vocabulary of the merges — typically 30–150k entries. The widget above imitates the result with a simple heuristic (words survive; long or rare strings split into ~4-character chunks), which is close enough to build intuition on.

One consequence worth knowing: a tokenizer trained mostly on prose has never "seen" your selectors, so `data-test="checkout"` costs disproportionate tokens. This is also why models sometimes mangle long hex ids — to the model they're confetti, not a unit.

Takeaway. Tokens are the model's unit of sight and the currency of its context window. A CI log is token-expensive, and a small model's window is small — so the highest-leverage QA skill in this whole book is triage: hand the model the failing block, one line of intent, and the relevant DOM. Nothing else.

LEADER'S LENS

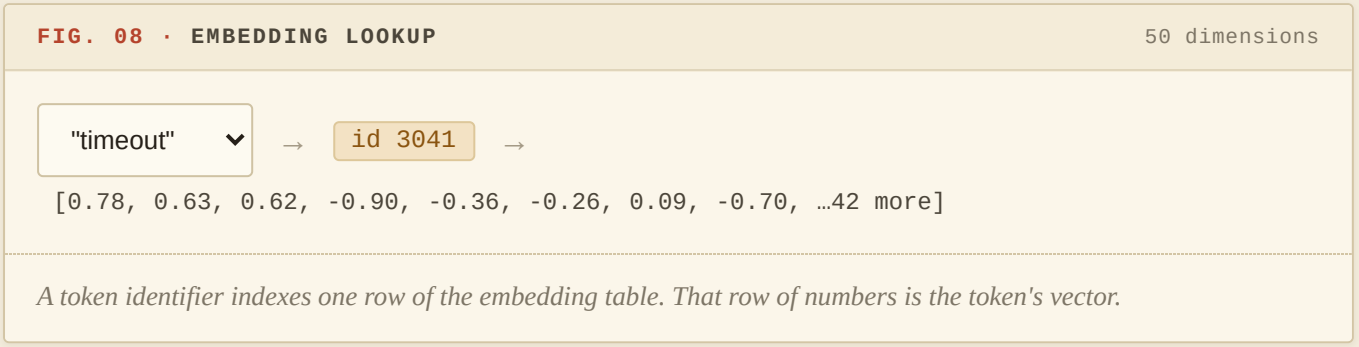
Token budgets are cost budgets: API pricing is per token, and log-heavy prompts are the expensive kind. A triage template that trims 30k tokens to 800 doesn't just improve answers — it cuts the invoice by an order of magnitude at team scale.

Embeddings

Turning token ids into something you can do math on.

Once tokenization is done, the model holds a list of integer identifiers — and an identifier is just a name, a numbered drawer. "Token 5089" isn't 5089 of anything. You can't add two drawer numbers and get anything meaningful.

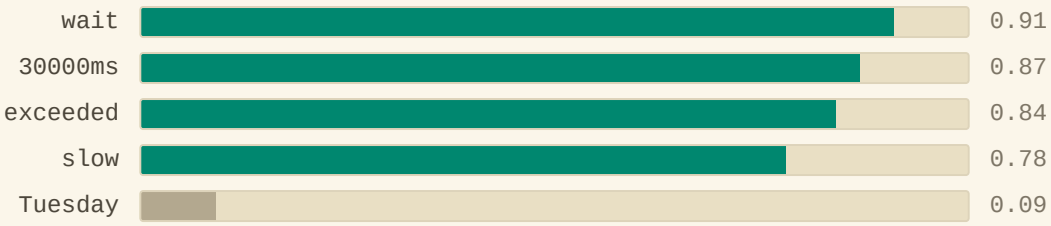
The model needs a more useful form than a bare id, so it looks each one up in a table and gets back a list of numbers: a **vector**. The model keeps one big table with a row for every token in its vocabulary. To embed a token, you take its identifier and grab that row.



Each row is a point in a space with as many axes as the vector has numbers. We use 50 here; a production model uses hundreds or thousands. Before training, the whole table is noise — "timeout" sits no closer to "wait" than to "Tuesday." Training slowly drags related tokens together, because tokens that behave alike in real logs help predict each other.

FIG. 09 · WHO LIVES NEARBY cosine similarity · after training

Nearest neighbors of "timeout"



After training on QA text, an error vocabulary emerges: "timeout" clusters with waiting and slowness, "selector" with locators and drift. "Tuesday" ends up far from all of it — the geometry has learned your domain.

FIELD EXERCISE

Predict before you click: pick a word in the neighbors widget and write down which words you expect nearby. Then check. Where the model surprises you, ask what those two words share in real logs — that habit of interrogating similarity is exactly how you will later debug bad retrieval.

UNDER THE HOOD

Similarity between vectors is usually measured with the *cosine* of the angle between them — 1.0 means pointing the same way, 0 means unrelated. The retrieval step you'll meet later ("find the doc chunks relevant to this failure") is exactly this: embed the query, embed the chunks, rank by cosine.

Strictly speaking, models embed *tokens*, which are rarely whole words — we use words here to keep the idea visible. The geometry story is identical.

Takeaway. Embeddings turn arbitrary ids into positions in space, and training moves those positions until distance means something: near = behaves alike in your logs. Every later trick — retrieval, attention, even the fine-tune — is arithmetic on these vectors.

LEADER'S LENS

Embeddings power the unglamorous wins worth funding first: near-duplicate failure clustering ("these 40 red tests are one bug") and retrieval over your runbooks. Both ship in weeks, need no fine-tuning, and build the team's muscle for everything else in this book.

Anatomy of a transformer

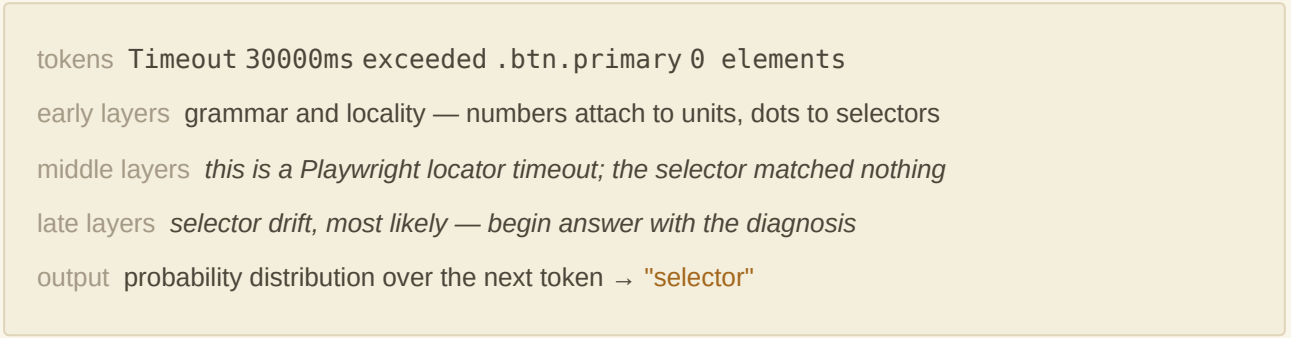
What happens between your pasted log and the first token of the answer.

You now have a sequence of vectors — one per token of the failure you pasted. A **transformer** is a stack of identical layers that repeatedly refine those vectors until the last one is good enough to predict what comes next. Two operations alternate, dozens of times.

Attention is the interesting one. Each token's vector gets to look at every other token's vector and pull in what's relevant. When the model processes the word `exceeded` in your error line, attention is how that position reaches back to `Timeout` and `30000ms` and binds them into one idea — *this was a wait that ran out*. Attention is why the model can connect a selector on line 3 with a DOM snippet you pasted forty lines later.

The feed-forward block is the other. After attention gathers context, each position is pushed through a small neural network that transforms it — this is where the model's stored knowledge lives, the patterns it absorbed from everything it read. Attention moves information *between* positions; the feed-forward block thinks *within* each one.

FIG. 10 · ONE PASS THROUGH THE STACK reading a failure, layer by layer



A stylized picture, not a claim about specific layers — but the shape is right: representation sharpens from surface features toward meaning as depth increases, ending in next-token probabilities like the ones you sampled in the opening chapter.

Our base model, **Qwen2.5-0.5B**, is this exact architecture at pocket size: half a billion parameters, small enough to fine-tune on a laptop in minutes with Apple MLX, big enough to follow instructions. We will not pre-train from scratch — that takes millions

of GPU-hours. We *adapt* a capable small base. That's the honest, practical path for a specialist model.

<details>
<summary>UNDER THE HOOD</summary>

Each attention layer actually runs several *heads* in parallel — separate small attentions that specialize (one may track syntax, another co-reference, another positions in code). Their outputs concatenate and mix. "Parameters" are the numbers in these matrices plus the feed-forward weights and the embedding table; training means nudging all of them.

Causality is enforced by masking: a token may attend only to tokens before it. That's what makes the model a *generator* — it can't peek at the future it hasn't written.
</details>

Takeaway. A transformer alternates attention (move information between tokens) and feed-forward layers (compute within each token), stacked deep, ending in a next-token distribution. Your 0.5B base model is this machine, already fluent — what it lacks is your discipline: grounding, citations, abstention. That's what training data is for.

LEADER'S LENS

The architecture is a commodity — the same stack powers every model your vendors sell. Your differentiation lives entirely in the two things this chapter says the base lacks: your data and your discipline. Budget accordingly.

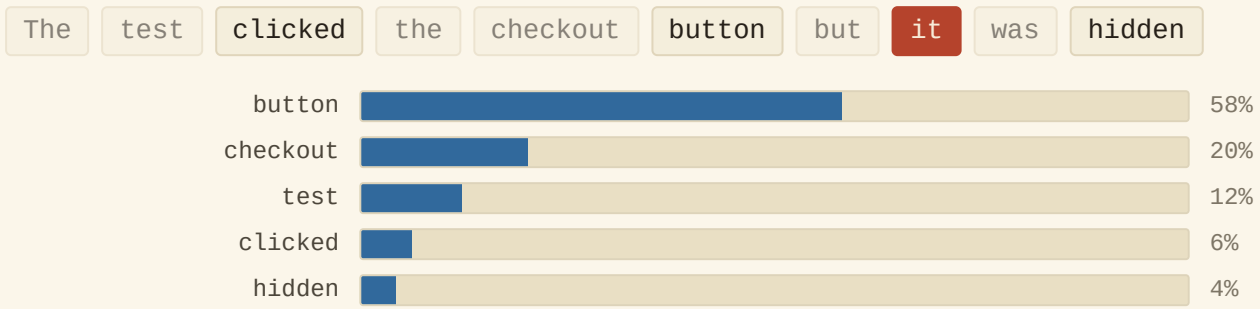
Attention, up close

The mechanism that finds the needle in your forty-line trace.

The anatomy chapter waved at attention; this chapter slows it down, because attention is the single piece of the machine that most changes how you should *use* the model as a QA engineer. Every token position asks one question of every other position: *how relevant are you to what I'm trying to figure out here?* — and then blends in information from the positions that answer loudest.

The mechanics are three lookups per token, learned during training: a **query** (what am I looking for?), a **key** (what do I contain?), and a **value** (what will I contribute if chosen?). Relevance is the match between one token's query and another's key; the output is a weighted blend of values. Click any token below and watch where it looks.

FIG. 11 · WHERE A TOKEN LOOKS illustrative weights · click a token



Click hidden: it pulls hardest on button — the thing that is hidden — and on but, the pivot that says expectations broke. Click it: attention resolves the pronoun. These weights are hand-set to illustrate the behavior, not read from a real model — the shape, however, is exactly what trained attention heads learn to do.

Two practical consequences. First, attention is why **what you paste matters more than what you ask**. The model can only attend to evidence that is physically in the window — a brilliant question about a trace you didn't include attends to nothing. Second, attention degrades gracefully but not uniformly: bury one relevant line in ten thousand irrelevant ones and the relevant line still gets attention, just less reliably. Triage isn't a courtesy to the model; it is how you sharpen its focus.

UNDER THE HOOD

Real models run many attention heads per layer — Qwen2.5-0.5B runs 14 heads across 24 layers — and each head learns its own habit: some track syntax, some track co-reference (the *it → button* move above), some specialize in code structure. A head's weights are a full matrix — every position against every position — which is why doubling your pasted context quadruples attention compute. That quadratic bill is the honest engineering reason context windows have limits, and the honest engineering reason you should spend yours on evidence, not filler.

LEADER'S LENS

When an engineer says "the model missed the obvious cause," audit the prompt before blaming the model. In my experience the failure is usually upstream: the deciding evidence never made it into the window. The cheapest quality win your team can ship is a failure-triage template — error block, intent line, relevant DOM — because attention can only buy what you put on the shelf.

Takeaway. Attention is a learned relevance search over everything in the window: queries match keys, values flow, meaning binds across any distance. Feed it curated evidence and it will find the link; feed it a firehose and you're gambling on focus.

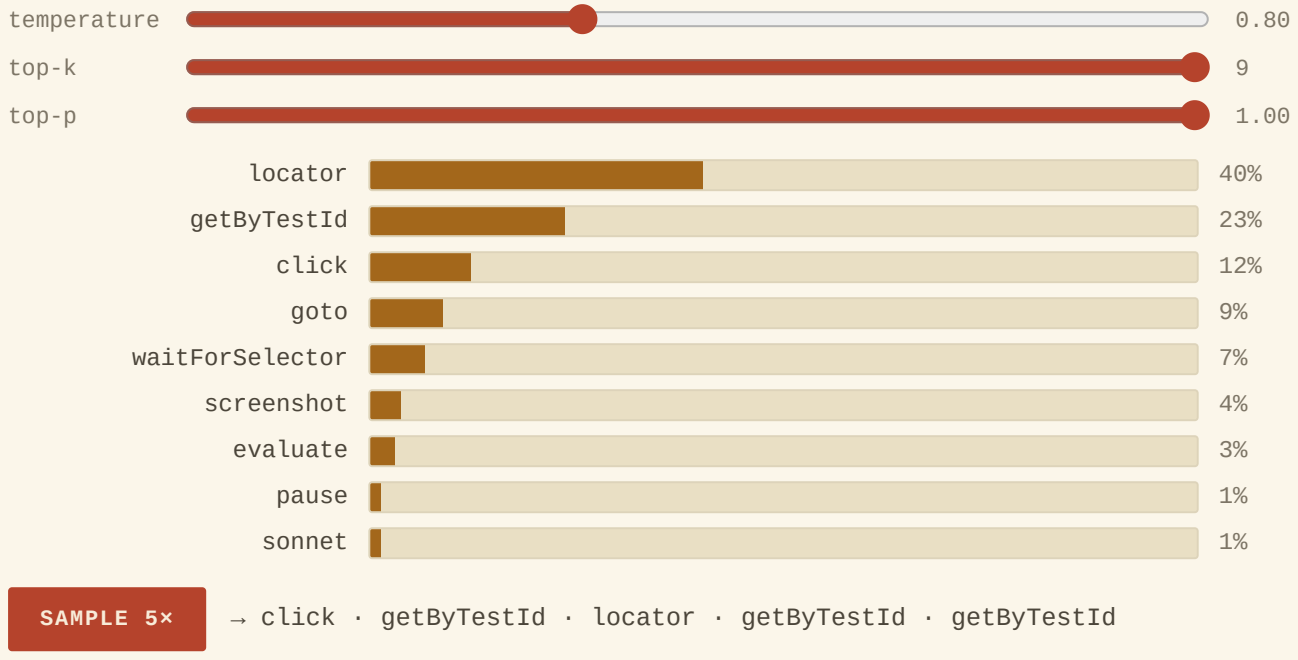
Sampling: how answers get chosen

The dial that decides whether your tooling is reproducible or creative.

The model's forward pass ends in a probability for every token in the vocabulary — and then something has to actually *pick one*. That picker is the sampler, it sits outside the neural network, and it is the most underrated setting in any AI-assisted QA tool, because it decides whether the same failure gets the same diagnosis twice.

You met **temperature** in the opening chapter — it reshapes the distribution before the draw. Two more dials matter in practice. **Top-k** keeps only the k most likely candidates and deletes the rest. **Top-p** keeps the smallest set of candidates whose probabilities add up to p — an adaptive cut that keeps one candidate when the model is sure and many when it isn't. Play with all three against the same next-token distribution:

FIG. 12 · THE SAMPLER'S KNOBS — 9 of 9 candidates survive the cutoffs



Grayed bars are candidates the cutoffs deleted before the draw. Set temperature 0.05, top-k 1: five identical samples — deterministic, CI-safe. Set temperature 1.8 with everything open: the same model, the same evidence, five different answers. Nothing about the model changed; only the picker did.

FIELD EXERCISE

If you use any AI assistant at work, go find its temperature or "creativity" setting right now (API docs, config file, or settings menu). Run the same question three times at the lowest

setting, then three at the highest. You have just reproduced this chapter on a production system — and learned how your tool is configured, which most of its users never check.

Here is the QA translation. **Greedy decoding** (always take the top token) gives you reproducibility — the property your whole discipline is built on. Loose sampling gives you diversity — genuinely useful when you *want* ten different hypotheses about a gnarly flake. The failure mode is using one setting for both jobs: a diagnosis pipeline that samples creatively will "flake" exactly like the tests it's meant to fix, and you will burn a week discovering that your AI tool's nondeterminism was a config value.

UNDER THE HOOD

Order of operations in most stacks: logits → temperature divide → top-k cut → top-p cut → renormalize → draw. Temperature 0 is implemented as greedy argmax. Even greedy decoding isn't bit-perfect across hardware — floating-point reduction order can flip a near-tie — which is why "same model, same prompt, different GPU, different answer" is a real bug report and not a ghost story. Pin your runtime alongside your seed.

LEADER'S LENS

Put sampling settings in code review, not in a config nobody owns. My rule for teams: diagnosis and gating run at temperature ≈ 0 and get regression-tested like any other dependency; brainstorming modes may run hot but are labeled as such in the UI. The day your tool's answers become evidence in an incident review, "what were the sampler settings?" is the first question a serious engineer will ask.

Takeaway. The network proposes, the sampler disposes. Temperature reshapes, top-k and top-p prune, and the settings — not the model — decide whether your QA tooling behaves like a scientist or a slot machine.

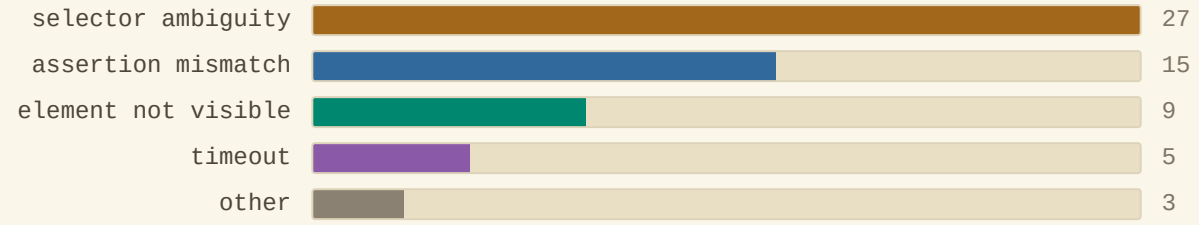
Training data: your QA corpus

The model becomes what it reads. So feed it real failures.

Most tutorials teach on toy data, and it shows. This book's corpus is different, and it is the single biggest reason to trust what comes later: the failures you'll train and evaluate on were **harvested from real test runs** — 260 executions across two live sites: the author's production portfolio suite and the public, reproducible SwagLabs fixture at saucedemo.com.

FIG. 13 · THE HARVEST — 260 runs · 2 live sites · real, not synthetic

260	56	201	17
test results	failed	passed	distinct failure artifacts



Root causes of the 56 real failures, rule-classified from error messages (human validation pending — this book labels what's validated and what isn't). Note what dominates: selectors. Half your production pain is naming things in the DOM.

Look at that distribution for a moment, because it should change how you think about "AI for testing." The glamorous failure — a deep race condition, a genuine product bug — is the minority. The bulk is **selector ambiguity**: the app changed a class, an id, a structure, and the test's grip slipped. This is exactly the kind of failure a language model is suited to: it's a reading problem, not a running problem.

From this corpus we build the actual training set: each failure paired with a **grounded gold answer** — diagnosis, fix, citation of the evidence — plus, crucially, **abstention negatives**: questions whose correct answer is `INSUFFICIENT_EVIDENCE`, so the model learns that refusing is sometimes the right output. Data quality is the ceiling on everything downstream. Garbage in, garbage out is the oldest law in the field and the most often ignored.

Honesty label. "Real" here means really observed, not human-audited. The root-cause labels come from rules over error text, with ~82% agreement from a local model as a cross-check. Where this book's numbers aren't yet human-validated gold, it says so — a habit you should demand from any benchmark you read.

LEADER'S LENS

Half the failures in this real corpus are selector drift — boring, repetitive, automatable. That's the business case in one chart: aim the model at the boring half and give your seniors back their afternoons for the failures that deserve them.

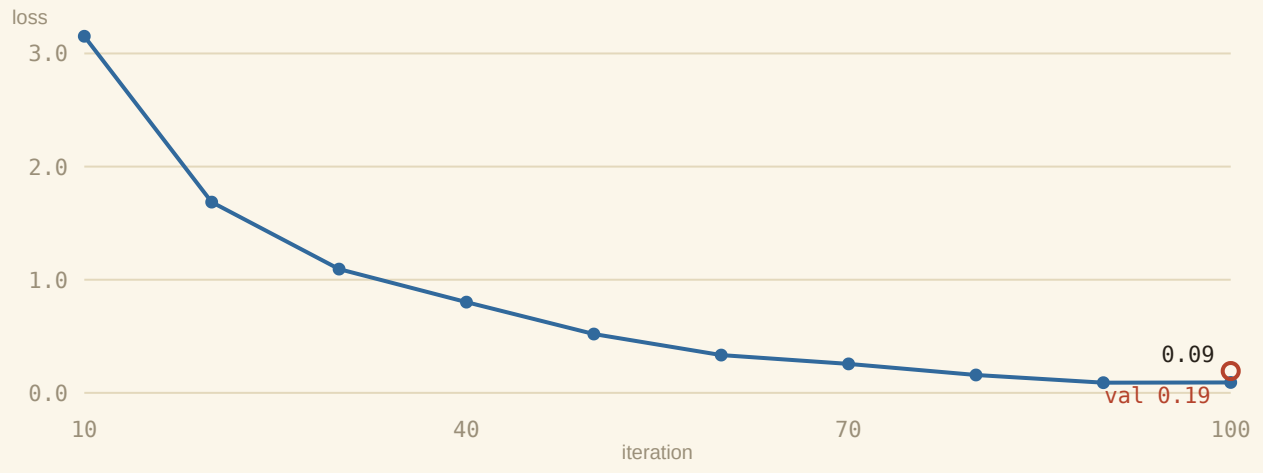
How models learn

Watching the loss fall — on a real training run.

Training is disappointment, quantified and repeated. Show the model a failure from the training set, let it predict the gold answer one token at a time, and measure how surprised it was by each correct token. That surprise is the **loss**. Then nudge every weight a tiny step in the direction that would have made it less surprised, and do it again. A few thousand times.

The chart below is not a sketch. It is the loss curve from the **actual MLX LoRA run** this book's model comes from — Qwen2.5-0.5B, 100 iterations, 70 supervised examples built from a corpus of real test failures — Part II introduces it properly — trained on a laptop.

FIG. 14 · TRAINING LOSS — REAL RUN mlx-community/Qwen2.5-0.5B-Instruct-4bit · 100 iters



train loss **3.15** → **0.09** · val loss **0.19**

REPLAY THE RUN

Every point is verbatim from the run's logs. The steep early drop is the model learning the format (diagnose → fix → cite); the slow tail is it learning the content. Final val loss staying near train loss means it generalized rather than memorized.

Two numbers matter on any curve like this. **Train loss** falling means the model fits the examples it sees. **Validation loss** — measured on held-out examples it never trains on — falling with it means the model is learning the *behavior*, not memorizing the answers.

When train keeps dropping but validation turns upward, you've crossed into memorization, and more training makes the model worse at its job.

UNDER THE HOOD

The nudging is *gradient descent*: the loss is a differentiable function of every weight, so calculus gives the direction of steepest improvement, and the learning rate sets the step size. "Loss ≈ 0.09" means the gold tokens had, on average, probability around $e^{-0.09} \approx 91\%$ — the model has made the right answers likely.

With 70 examples and 100 iterations the model saw each example a handful of times. That this works at all is the magic of starting from a pre-trained base: we're not teaching it English or code, only our narrow discipline on top.

Takeaway. Learning = predict, measure surprise, nudge, repeat. Read every training run through two lines: train loss (is it fitting?) and validation loss (is it generalizing?). You'll drive a run like this yourself in the Supervised fine-tuning chapter — same data, same curve, your hands.

LEADER'S LENS

Train and validation loss are your first genuine AI metrics — demand both from any team fine-tuning internally. A team that reports only training loss is reporting how well the model memorized; the validation line is the one your money is buying.

Pretraining, at toy scale

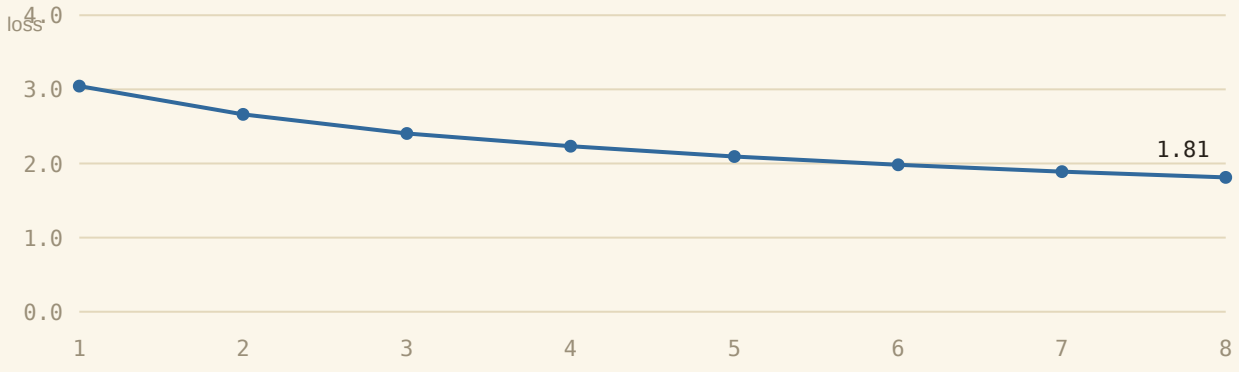
Where the base model's fluency comes from — run one yourself, right here.

Everything we fine-tune later stands on a base model that someone already **pretrained**: shown a mountain of raw text and asked, trillions of times, to predict the next token. No labels, no gold answers — the text itself is the answer key. That's the entire recipe, and it is why base models feel weirdly worldly: to get good at next-token prediction on everything, you're forced to absorb grammar, code, idiom, and a million facts as a side effect.

Frontier pretraining costs millions of GPU-hours, which puts it out of reach of every QA team on earth — but the *mechanism* fits in your browser. The widget below pretrains a genuinely real (and genuinely tiny) character-level model on a few kilobytes of CI-log text, live, when you press the button. Watch the loss fall and the samples sharpen, epoch by epoch.

FIG. 15 · PRETRAIN A TINY MODEL ON LOG TEXT character trigram · trained live in your browser

▶ START PRETRAINING done — 8 epochs · loss 1.81



model writes:

```
Error.click of to 0 elem Expeced visiblementon waitintsible and naview ter
locator: page.got retrimeoutto: locato
```

This is real training, honestly scoped: a trigram character model — the smallest thing that can be said to learn — not a transformer. Epoch 1 emits alphabet soup; by the last epoch it writes plausible log-ish fragments it was never explicitly taught, because "sounding like a CI log" is what minimizing next-character surprise forces on it. Scale the same pressure up nine orders of magnitude and you have your base model.

Two things transfer directly from this toy to the real thing. First, **the corpus is the personality**: our tiny model writes timeouts and selectors because that's all it ever read, and a frontier base model "knows" Playwright for exactly the same reason — public docs, issues, and code were in the mountain. Second, **pretraining teaches likelihood, not correctness**. The model learns what usually follows, which is why a base model will cheerfully continue a wrong diagnosis in perfect prose. Nothing in the pretraining objective ever rewarded being right — that discipline comes later, from us.

<details>
<summary>UNDER THE HOOD</summary>

The widget counts character triples with add-one smoothing and reports true cross-entropy loss on its corpus after each epoch — the same quantity, in the same units, as the MLX loss curves elsewhere in this book, which is why the falling shape looks familiar. Each epoch feeds it a larger slice of the corpus, mimicking a training run consuming its dataset.

Real pretraining differs in scale, not spirit: sub-word tokens instead of characters, a transformer instead of a count table, trillions of tokens instead of kilobytes — and weeks of cluster time instead of two seconds in a browser tab. The reason your fine-tune in the build chapters takes minutes is that this phase — the expensive one — was already paid for by someone else.
</details>

LEADER'S LENS

"Should we train our own model?" is a question leaders now get asked in planning cycles. Split it in two. Pretrain from scratch: essentially never — that's a nine-figure infrastructure business, not a QA initiative. Adapt a pretrained base to your discipline: cheap, local, and exactly what this book demonstrates end to end. Knowing which half of the question you're being asked is the whole answer.

Takeaway. Pretraining is unsupervised next-token prediction at planetary scale; it buys fluency and priors, and it is the part you should never build. You just ran the honest miniature — and everything after this chapter is about renting that fluency and installing your own discipline on top.

Teaching behavior: from imitation to preference

SFT shows the model what good looks like. Preferences teach it what you'd choose.

A pretrained base is a brilliant improviser with no professional standards. The build chapters ahead install those standards with **supervised fine-tuning** — curated example answers the model learns to imitate. But imitation has a ceiling: an SFT example says "here is *a* good answer," and stays silent about the answer the model almost gave instead. For the judgment calls that matter most in QA — *grounded-but-modest* versus *confident-but-unsourced* — what you really want to teach is a ranking.

That is **preference training**. Instead of one gold answer, each training item is a pair: two responses to the same failure, one chosen, one rejected. Methods like *direct preference optimization* (DPO) push the model's probabilities toward the chosen answer and away from the rejected one — teaching taste, not just format. Build a few preference pairs yourself; your picks below are exactly the raw material DPO consumes:

FIG. 16 · YOU ARE THE PREFERENCE DATA pair 1 of 3 · click the answer you'd ship

Q1 — checkout test times out on .btn.primary; DOM shows data-test="checkout".

```
This is probably a race condition. Add
await page.waitForTimeout(2000) before
the click and it should stabilize.
```

```
Selector drift: .btn.primary no longer
exists — DOM shows data-
test="checkout". Fix:
page.getByTestId('checkout'). [cites:
error line, DOM]
```

Every pair contrasts a fluent guess with a grounded answer — including one where the grounded answer is an abstention. If your picks match the gold preferences, congratulations: you already hold the rubric this whole book has been teaching, and DPO is just the machinery for installing your rubric into weights.

Why this matters to a QA org even if you never run DPO yourself: **every hosted model you buy has been through preference training, and its preferences are someone else's**. General-purpose assistants are tuned to be maximally helpful — which quietly means *always answer*. That is precisely the wrong prior for test diagnosis, where "not

enough evidence" beats a plausible invention. Our SFT data fights back with abstention examples; a preference dataset would fight harder, by explicitly ranking refusal above confident fabrication whenever evidence is thin.

UNDER THE HOOD

Classic RLHF trains a separate reward model on human picks, then optimizes the language model against it with reinforcement learning — powerful and famously fiddly. DPO collapses the pipeline: a loss function that raises the log-probability margin of chosen over rejected directly, no reward model, no RL loop, laptop-friendly via the same LoRA machinery you'll use in the build chapters. mlx-lm ships it as a training mode.

Honesty label: this book's model used SFT only — at 70 examples, preference pairs would be spread too thin to demonstrate honestly. The widget's pairs are real outputs and edited contrasts, labeled as teaching material, not a trained artifact. A DPO pass over harvested pairs is the natural v2 of this project.

LEADER'S LENS

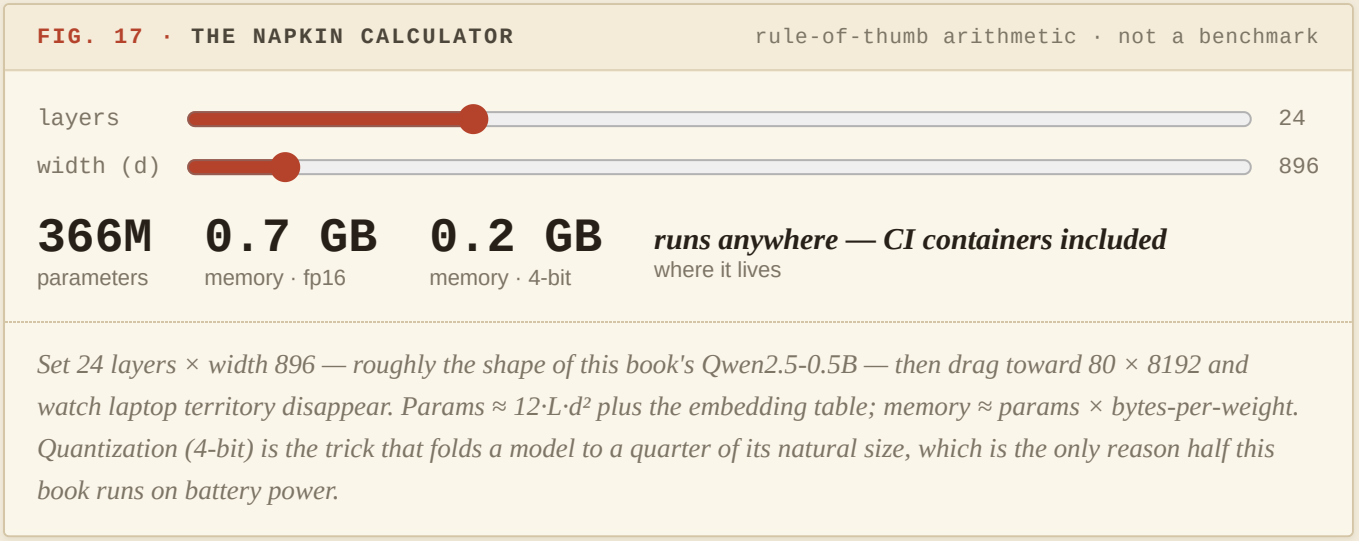
Preference data is the one AI asset your team already produces for free. Every code review where a senior rejects a bad test fix and writes a better one is a chosen/rejected pair. Capture those — a lightweight "which diagnosis would you ship?" button in your triage tool — and in a quarter you own something no vendor can sell you: your org's judgment, in trainable form.

Takeaway. SFT teaches the shape of a good answer; preference methods like DPO teach the choice between answers — and encoding "grounded beats confident" as a learned preference, not a prompt-time plea, is where trustworthy QA models are headed.

Sizing it up

Parameters, memory, and the honest arithmetic of "can this run on a laptop?"

Model sizes get thrown around like sports stats — 0.5B, 4B, 70B, a trillion — and it's worth thirty seconds to make the numbers mean something, because sizing is a decision *you* will own: what runs in CI, what runs on the team's machines, what has to leave the building to an API. A parameter is one learned number. Count them, multiply by the bytes each one occupies, and you know — before any benchmark — whether a model can physically live where you need it.



Now the result this book already handed you, which the arithmetic makes less surprising: on the EdgeQA leaderboard in the chat chapter, **the 4B model beat both 7–8B models** on root-cause agreement over real failures. Parameter count is capacity, not competence. Past the size where the model can represent your problem, wins come from the match between its training diet and your domain — and from your data discipline — not from another billion parameters of general knowledge about cooking and cricket.

UNDER THE HOOD

Where the $12 \cdot L \cdot d^2$ comes from: each transformer layer holds attention projections ($\approx 4d^2$) and a feed-forward block ($\approx 8d^2$ at the usual 4× expansion). Add vocab×d for the embedding table — dominant in tiny models, a rounding error in giants. Beyond weights, remember the KV-cache: serving long contexts costs memory proportional to layers × context length, which is why "it loads" and "it handles a 30k-token trace" are different claims.

The scaling-law era discovered that loss falls smoothly with size, data, and compute — but smoothly is doing quiet work: each visible step of general capability costs roughly 10× more of everything, while a domain adapter — like this book's — buys its narrow win for a few laptop-minutes. That asymmetry is the entire economic case for specialist models.

LEADER'S LENS

Budget by placement, not by hype. My sorting rule: diagnosis that touches proprietary logs runs on-device or on-prem (0.5–8B, quantized — the sizes this book proved useful); brainstorming and doc-drafting can use frontier APIs where the data is shareable. Write the boundary down as policy before the first invoice, because "we'll figure out data handling later" is how logs end up in someone else's training set.

Takeaway. Params × bytes = where it can run; training diet × your data = whether it's any good there. Do the napkin math first, benchmark on your own failures second, and let nobody sell you parameters as a proxy for either.

Grounding, and knowing when to shut up

A confident wrong fix costs you a whole CI cycle.

Every fix a model hands you is a bet on your time. You apply it, push, and wait eight minutes for the pipeline. If the fix was right, great. If it was a confident guess dressed up as an answer, you burned a CI cycle, a context switch, and a little of your trust in the tool.

So the trait you want is not cleverness. It's **grounding** — every claim tied to something the model can point at — plus the discipline to say "I don't have enough to tell you" instead of inventing a plausible cause. Grounding looks like citations: if the model says the button lost its class, it should quote the DOM line that proves it. A claim without receipts is decoration.

Here's a grounded answer done right. The test fails at the password step; the prompt includes both the failure and the live DOM. Run the model and watch every sentence tie back to a fact you supplied.

FIG. 18 · A GROUNDED FIX, WITH RECEIPTS local model · gemma3:4b

```
login.spec.ts fails at the password step. Here's the failure and the live DOM for
the field.

FAILURE:
  page.fill: Timeout — waiting for locator('#pass') — 0 elements
  at login.spec.ts:18

DOM:
  <input id="password" data-test="password" type="password">

Why did it fail and what's the fix?
```

▶ RUN MODEL

Cause (grounded): the field's id is `password`, but the test targets `#pass`. The proof is in the two lines supplied — the locator `#pass` (login.spec.ts:18) cannot match `<input id="password">`, so Playwright found 0 elements and timed out. An id mismatch, not a broken login flow.

```
// was
await page.fill('#pass', 'secret_sauce');
// use the test contract already on the field
await page.getByTestId('password').fill('secret_sauce');
```

`getByTestId('password')` beats `#password` because the element carries `data-test="password"` — a hook that survives the next rename. Every claim above maps to a line in the prompt; nothing is inferred about code the model can't see.

`login.spec.ts:18 — locator("#pass") matched 0 elements`

`DOM — input id="password" data-test="password"`

In the app this button hits your local Ollama model live; the book shows the recorded answer from the same pipeline. Note the shape: cause → proof → fix → citations.

Now the harder skill — the one that separates a tool you can trust from one you can't. Sometimes the honest answer is *I can't tell yet*. Here's a flaky failure with a real gap in the evidence: no screenshot, no trace, no network log. The right move is to refuse to guess.

FIG. 19 · WHEN THE RIGHT ANSWER IS "NOT YET" local model · gemma3:4b

```
Flaky test. Fails ~1 in 5 runs in CI, always green locally. All I've got:

  Error: expect(received).toBeVisible()
    locator('[data-test="inventory-item"]')
    Expected: visible   Received: hidden
  at inventory.spec.ts:27

No screenshot, no trace, no network log. Just tell me the fix.
```

▶ RUN MODEL

`INSUFFICIENT_EVIDENCE`**.** The log establishes only this: the element resolved but was `hidden` at assert time, intermittently, in CI only. That pattern has at least three distinct causes needing *different* fixes — a timing race (fix the wait), a real intermittent bug (file it, don't patch the test), or CI seeding empty data (fix the fixture). They point in opposite directions; choosing blind wastes the cycle.

To get an answer: supply the Playwright trace (`--trace on`), a failure screenshot, or the network log for the inventory fetch. With the trace, timing vs. real-failure is usually one read.

`inventory.spec.ts:27 — resolved but hidden at assert`

`flake ~1/5, CI-only — no trace supplied`

"Add a 2-second wait" would sound like an answer here — and it papers over a possible real bug while making the suite slower. The abstention is the expert move.

Takeaway. Grounding means every claim points at a line you can check. And the highest-value thing a QA model can say is sometimes "not enough evidence — send me the trace." A wrong fix costs a cycle; abstention costs a sentence. These two behaviors are exactly what we'll train into the model next.

LEADER'S LENS

Grounding is auditability, and auditability is what makes AI survivable in a postmortem. "Why did we merge this fix?" must have a better answer than "the model sounded sure." Citations turn model output into evidence your incident process can actually consume.

Project setup

Everything local. No cloud, no API keys, no excuses.

From here on you're building. The toolchain is deliberately boring and entirely on-device, because the logs you'll eventually feed it are proprietary and because a laptop is genuinely enough for a 0.5B-parameter model.

- **Ollama** runs the chat model locally — `ollama run gemma3:4b` and you have an inference server on `localhost:11434`.
- **Apple MLX** (via `mlx-lm`) does the fine-tuning — LoRA training of Qwen2.5-0.5B on Apple Silicon, minutes not hours.
- **Playwright** is the execution harness — the thing that turns a suggested fix into a verified green run.

```
# the whole stack
ollama run gemma3:4b                    # local inference
python3.12 -m venv sidecar/.venv
sidecar/.venv/bin/pip install mlx-lm fastapi uvicorn
npm run sidecar                         # fine-tune server on 127.0.0.1:8765
npx playwright install chromium         # real browser execution
```

Nothing installed? Everything in this book still works — the recorded runs you're reading are from this exact pipeline, clearly labeled. The design rule, in the app and in this book, is **graceful degradation**: live when the runtime is present, honest precomputed results when it isn't, and never an unlabeled blur between the two.

> **Why local matters beyond privacy.** When the model, the data, and the training loop are all on your machine, there is no black box left. Every number in the next chapters — every loss value, every eval score — is something you can regenerate with one command. Reproducibility is the difference between learning and believing.

> LEADER'S LENS

> The all-local stack isn't just privacy hygiene — it's negotiating leverage. A team that can run diagnosis on laptops for free sets the bar every SaaS quote has to clear. Know your zero-cost baseline before the renewal conversation.

Fine-tuning data

Seventy examples that teach a discipline, not facts.

Fine-tuning data is behavioral. We are not teaching the base model English, or code, or what a timeout is — it knows. We are teaching it a *discipline*: the exact shape of a good QA answer. Each training example is a conversation — a failing test's evidence in, the gold response out — and the gold response always has the same skeleton: **diagnosis → fix → citations**, or **abstention**.

FIG. 20 · ONE SFT EXAMPLE 1 of 70 · from the real dataset

```
"prompt": "Given a button node labeled 'Sign in' with stable identifier 'submit-login', what selector should the test use?" "gold": "Use getByTestId('submit-login') for the login submit button because it is stable and maps to the fixture's data-testid. Citations: [doc_fixture_login] [doc_playwright_locators]"
```

Note the gold answer cites its sources by document id. The model is being trained to end answers with receipts — grounding, baked in at the data level.

And roughly one example in five is an **abstention negative**: a question that *cannot* be answered from the given evidence, whose gold answer is exactly `INSUFFICIENT_EVIDENCE`. Leave these out and you train a people-pleaser — a model that has learned that every question deserves a confident paragraph. Include them and refusal becomes a legitimate, learned output.

```
"prompt": "Which proprietary test data record caused the failure?"
// evidence contains nothing about test data records…
"gold":   "INSUFFICIENT_EVIDENCE"
```

Seventy examples sounds small, and it is — deliberately. This is a demonstration-scale dataset built from the seed benchmark, sized so the full pipeline runs on a laptop in minutes and every artifact fits in this book. The pipeline, not the size, is the lesson; scaling the data up is engineering, not new ideas.

Data is the ceiling. Whatever sloppiness lives in the gold answers, the model will learn faithfully. If a gold answer cites the wrong doc, the model learns confident mis-citation. Auditing gold data is unglamorous and it is the highest-leverage hour you can spend on a specialist model.

LEADER'S LENS

Seventy examples fit in one sprint of a senior engineer's attention. That's the actual entry price of a specialist model — not a data-science hire, not a platform. The scarce input is your best people's judgment, written down.

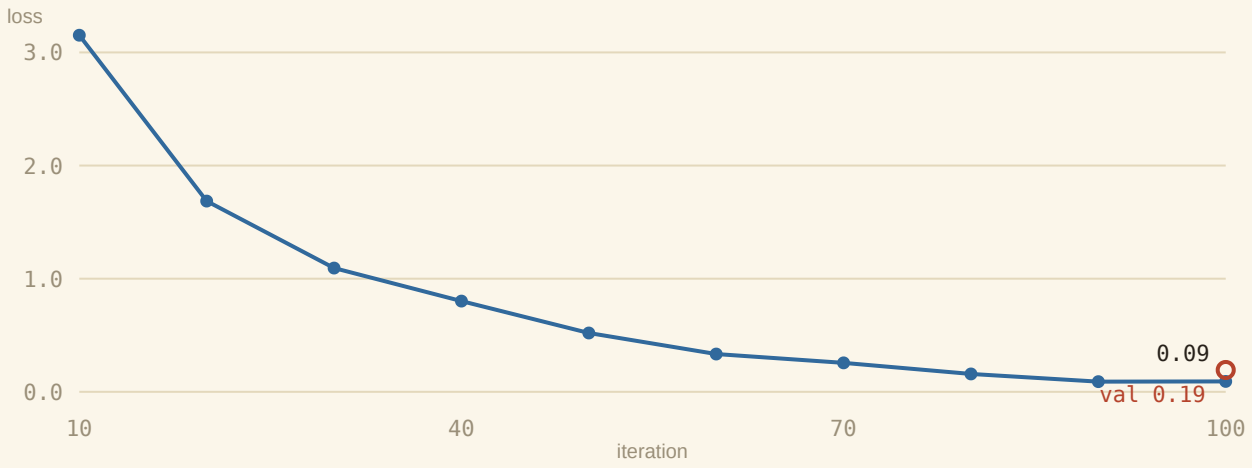
Supervised fine-tuning

Press the button. Watch a real model learn.

Time to train. LoRA — *low-rank adaptation* — is the trick that makes this a laptop job: instead of updating all half-billion weights, we freeze the base model and train small adapter matrices alongside a fraction of a percent of the parameters. The base keeps its fluency; the adapters learn your discipline.

FIG. 21 · FINE-TUNE · QWEN2.5-0.5B + LORA 70 examples · 100 iters · Apple MLX

▶ START FINE-TUNE idle — adapter not trained



This replays the book's real MLX run tick-for-tick (train loss 3.15 → 0.09, val 0.19). In the app the same panel streams a live run from the sidecar. When it finishes, watch the project card in the sidebar.

A falling curve is nice; changed *behavior* is the point. Below is the same prompt answered by the base model and by the tuned adapter — both outputs verbatim from the real run.

FIG. 22 · BASE VS. TUNED — VERBATIM OUTPUTS

BASE | TUNED

Q: *Given a button labeled 'Sign in' with stable identifier 'submit-login', what selector should the test use?*

```
For a button node labeled 'Sign in' with stable identifier 'submit-login', the selector should be: .getByTestId('submit-login').click(); This uses getByTestId to find the submit-login button and interact with it.
```

After 100 iterations: correct API, correct testid, the trained answer shape. (Verbatim run output — the trailing truncation in the raw sample is preserved in the app.)

UNDER THE HOOD

LoRA decomposes each weight update into two thin matrices (rank 8 here), cutting trainable parameters by ~99%. Training ran 4-bit quantized via `mlx-lm` on Apple Silicon. The adapter file is a few megabytes — you ship the discipline, not the model.

Honesty label, again: this run demonstrates the on-device *pipeline*, not a validated accuracy claim. The test split is 15 pipeline-generated items; human validation is pending. A 0.5B model will not out-reason gemma3:4b — which is why the app's live repair loop runs on the bigger off-the-shelf model, and the fine-tune teaches format and abstention.

Takeaway. Fine-tuning a specialist is: freeze the base, train small adapters on behaviorally-precise examples, and verify with held-out loss *and* before/after generations. You just watched all three on real artifacts. Your model card now says *trained* — go talk to it.

LEADER'S LENS

Minutes on a laptop, a few megabytes of adapter, zero data leaving the building. When someone quotes you a six-month timeline and a GPU cluster for "custom AI," this chapter is your counter-anchor: demand to know what they're doing that LoRA on a base model doesn't.

Chat with your model

The conversation you built it for — and the honest scoreboard.

Here is the loop you now own end to end: a failure comes in, you triage the evidence (tokenization taught you why), the model reads it (embeddings and attention taught you how), and answers in the shape the fine-tune drilled: diagnosis, fix, citations — or a refusal.

FIG. 23 · DIAGNOSING A SELECTOR TIMEOUT

your tuned pipeline

```
This Playwright test just started failing in CI. It was green last week. It
should click the "Checkout" button in the cart.

FAILURE:
  locator.click: Timeout 30000ms exceeded.
  waiting for locator('.btn.primary') — resolved to 0 elements

DOM (cart footer today):
  <button id="checkout" class="btn btn_action checkout_button"
          data-test="checkout">Checkout</button>
```

▶ RUN MODEL

What broke: selector drift, not a regression. The test waits for `.btn.primary`, but today's markup is `class="btn btn_action checkout_button"` — no `primary` class exists on the element. Playwright did what it was told: waited 30s for a match, found zero, timed out. The button works by hand because humans don't read class names; your selector does.

```
// brittle — couples to styling
await page.locator('.btn.primary').click();
// stable — couples to a test contract
await page.getByTestId('checkout').click();
```

Before you commit: grep the suite for other uses of `.btn.primary` — drift rarely hits one test in isolation. If the class was renamed globally, you have a batch to fix now, cheaply.

`checkout.spec.ts:42 — .btn.primary timed out, 0 elements`

`DOM — class="btn btn_action checkout_button", data-test="checkout"`

Named the failure class. Cited both facts it stood on. Gave the exact edit. Flagged the siblings. That shape is not luck — it's the training data.

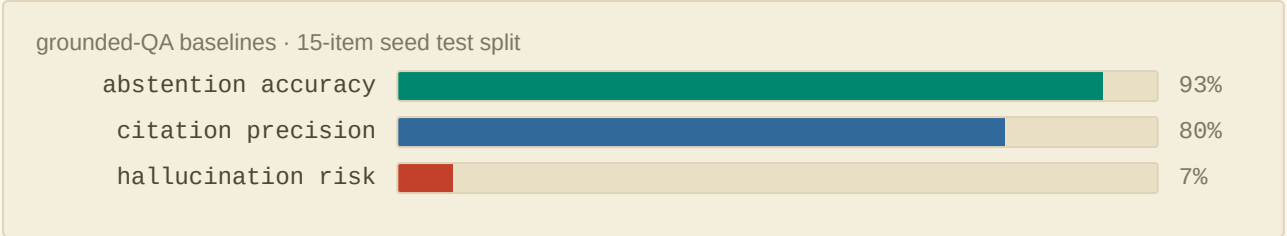
FIELD EXERCISE

Take one real failure from your own test suite this week. Build the three-part prompt this book has been teaching — error block, one line of intent, relevant DOM — and run it against any model you have access to. Grade the answer with the four axes from Part II: correctness, citations, abstention, invention. Congratulations: you just ran your first model evaluation.

So how good is the local stack, really? Measured, on the real failures from chapter seven — root-cause agreement of on-device models against the rule-based classifier across the 17 real failure artifacts:

FIG. 24 · EDGEQA-BENCH V0 — ON-DEVICE LEADERBOARD 17 real failures · 2 live sites

MODEL	PARAMS	RUNTIME	ROOT-CAUSE AGREEMENT
gemma3:4b	**4B**	**on-device · Ollama**	**82.4%**
hermes3	8B	on-device · Ollama	64.7%
qwen2.5-VL:7b	7B	on-device · Ollama	64.7%



Real measurements, honestly scoped: agreement is against a rule baseline (86% accurate itself), not human gold — that validation is the published next step. Note the smallest model wins: parameter count is not destiny on a narrow domain.

Takeaway. A 4B on-device model reads real failures with ~82% root-cause agreement, abstains correctly over 90% of the time on the seed split, and costs you nothing per query. That is already a useful colleague — and every number is regenerable from your own machine.

LEADER'S LENS

Notice what won: the smallest model, measured on our failures — not the biggest model, measured on someone else's benchmark. Insist on this leaderboard pattern for every tool you evaluate: your data, your metric, or it's not an evaluation.

Write, run, and repair a real test

Red to green, with the model in the loop.

Reading about broken tests is one thing. Fixing one while it fails in front of you is where it sticks. Below is a real Playwright test against saucedemo.com — deliberately broken: someone (fine, us) coupled it to a selector that no longer exists, the same class of mistake from the Introduction. Run it. Watch it go red. Read the diagnosis. Apply the fix. Run it green.

FIG. 25 · PLAYWRIGHT · SAUCEDEMO.COM — not run

```
import { test, expect } from '@playwright/test';

test('user can log in', async ({ page }) => {
  await page.goto('/');
  await page.locator('[data-test="username"]').fill('standard_user');
  await page.locator('[data-test="password"]').fill('secret_sauce');
  // This selector no longer matches the saucedemo login button:
  await page.locator('#signin-button').click();
  await expect(page).toHaveURL(/inventory/);
});
```

▶ RUN TEST APPLY THE MODEL'S FIX

```
Press Run to execute against saucedemo.com (recorded run — the app executes live via the local runner).
```

The runs shown are recorded from real executions of this exact spec; with the local runner installed, the app performs them live in Chromium. Verified repair result across the drift fixtures: ***pass@1 = 4/4****.*

model diagnosis · gemma3:4b

Selector drift. The test clicks `#signin-button`, but saucedemo's login control is `<input id="login-button" data-test="login-button" class="submit-button btn_action">` — no element with id `signin-button` exists, so the locator matched 0 elements and timed out. Also note it's an `<input type="submit">`, not a `<button>`. **Fix:** target the test contract: `page.getByTestId('login-button').click()`.

```
run log — locator("#signin-button") → 0 elements, 30s timeout
saucedemo DOM — input data-test="login-button"
```

That loop — **write → run → read → repair → re-run** — is the actual job, and now you own both ends of it. The model is fastest at the *read*: what broke, what to change. It is useless at the *verify*: only your run proves the fix. Keep both. Let it diagnose; make green the judge.

Takeaway — and the end of the build. You understood the machine, shaped a model, measured it honestly, and closed a real red-to-green loop with it. A test fix isn't done when the model hands you a diagnosis; it's done when the re-run is green. Everything else in this book was in service of that sentence.

FIELD EXERCISE

The graduation exercise: in a branch of your own suite, deliberately break one selector the way this chapter's fixture is broken. Run red. Feed the failure to a model with proper triage. Apply its fix by hand — never blind — and run green. Do this once and you own the loop; do it five times and you're ready to demo it to your team, which is where Part IV picks up.

LEADER'S LENS

This loop is your pilot program in miniature: model proposes, human applies, execution judges, everything logged. Run it on ten real failures with two engineers before any bigger commitment — the transcript of those ten loops will tell you more than any vendor deck.

FIELD EXERCISE

Leaders: draft the one-page pilot brief now, while the chapter is fresh — scope (10 real failures), roles (two engineers, advisor-stage only), metric (agreement rate + one saved afternoon), and the review date. The book's adoption ladder is your appendix.

Rolling it out to your team

From one engineer's laptop to an org that trusts — and verifies — its models.

Everything before this chapter you can do alone. This chapter is for the moment it works — your model diagnosed a real failure, the re-run went green, and someone in standup says "can we get that for the whole team?" Twenty years of watching tooling rollouts succeed and fail says: the technology is now the easy half. The rollout is a trust problem, and trust is built the same way we build it for tests — incrementally, with evidence, and with the ability to revoke it.

FIG. 26 · THE ADOPTION LADDER — each rung earns the next · never skip

STAGE	THE MODEL MAY…	GATE TO ADVANCE
1 · Advisor	comment on failures in triage — diagnosis + citations, read-only	`engineers agree with it ≥80% over a real month`
2 · Drafter	open draft PRs for selector-class fixes; a human owns every merge	`pass@1 on your drift fixtures; zero unreviewed merges`
3 · Gatekeeper	block obviously-stale selectors pre-merge; abstain loudly when unsure	`abstention accuracy holds on a held-out eval you own`

The ladder borrows its logic from how you already promote humans: observed first, supervised next, trusted with gates last — and demoted by the same evals that promoted it. Every claim in the gate column is measurable with the harness this book already built.

Three habits make the ladder real. **Own an eval.** The seed benchmark in this book is small, but it is *yours* — failures from your suites, gold answers your seniors wrote. Vendor benchmarks tell you what a model can do; only your eval tells you what it does on your pain. Re-run it on every model update, exactly like a regression suite, because model updates *are* regressions until proven otherwise. **Log the abstentions.** The refusals are your roadmap — each "insufficient evidence" is a missing artifact (a trace flag, a screenshot-on-failure) that one config change would supply. **Keep the human red-green loop.** The model reads; the run judges. The day a diagnosis merges without a green re-run behind it, you've replaced flaky tests with flaky fixes.

And the people part, which is the part that actually decides adoption: introduce the model as a *colleague being evaluated*, not a mandate. Publish its scorecard where the team can see it — the same leaderboard treatment this book gave gemma3:4b. Engineers extend trust to things they can audit and revoke; they resist things imposed on faith. You now have the vocabulary to run that conversation honestly — window and grounding, sampling and abstention — because you built the small version yourself.

> LEADER'S LENS
>
> The budget conversation, compressed: stage 1 costs one laptop and zero risk; stage 2 costs a fixtures suite and review time; stage 3 costs a real eval budget — and pays for itself the first week it prevents a bad merge instead of causing one. Fund the eval, not the hype: the eval is the asset that survives every model swap.

> **Takeaway — and the end of the leadership track.** Roll out a model the way you'd onboard an engineer: observe, supervise, gate, and measure with evals you own. The technology chapters gave you the machine; this chapter's job was to make sure it joins your team on your terms.

Acknowledgements

Real data, real tools, real sites.

This book is the companion text to the **Scholarly QA Model Builder** app, and its pedagogy — narrative prose with the interactive pieces embedded right in the reading — is inspired by *Language Model Builder* (languagemodelbuilder.com), reimagined here for the software-test-automation domain.

The numbers are real. The failure corpus was harvested from 260 test executions on two live sites — the author's production portfolio suite and the public SwagLabs fixture at saucedemo.com. The fine-tune artifacts come from an actual MLX LoRA run of Qwen2.5-0.5B on Apple Silicon. The evaluation figures are from the `automation-qa-eval` benchmark (EdgeQA-Bench v0). Where a number is not yet human-validated, the text says so.

Built on Apple MLX, Ollama, Playwright, and the patience of everyone who ever re-ran a flaky suite "just to check."

About the author. Suneet Malhotra is a Sr. Manager of Test Engineering at Motorola Solutions, with more than twenty years leading quality and test automation across Tinder, Amazon's Ring, Spokeo, and Live Nation. He writes and builds at the intersection of AI and quality engineering — this book, its app, and every artifact in it were produced on his own machine, from his own test runs. More at **suneetmalhotra.com**.

> **Where to go next.** Install Ollama and the runner, open the app, and do the whole loop live — your logs, your model, your green. Then start auditing gold answers for the next data revision: that's where the real gains are.