Eval-Driven Development

Build a real eval suite from zero with Evalite — deterministic checks, LLM-as-judge, and the data flywheel that turns user feedback into a system that improves itself. The unit tests for probabilistic software.

Roger Stringer · rogerstringer.com

June 17, 2026

Contents

Evals Are the Unit Tests of AI

If you've shipped traditional software, you have a reflex: change the code, run the tests, trust the green. That reflex is the first thing AI breaks, and the most painful.

Traditional software is deterministic. The same input gives the same output, so a test is a clean assertion — `expect(total).toBe(42)` — and it passes or fails. An LLM is probabilistic. The same prompt can give a different answer twice in a row, a slightly reworded input can swing the result, and "correct" is often a matter of degree rather than a boolean. `expect(reply).toBe(…)` is useless when there's no single right string.

Evals are the unit tests for that world. An eval runs your AI system over a set of inputs and *scores* each output — not pass/fail, but "how good, on a scale." Did the classifier pick the right category? Is the drafted reply actually helpful? Did the summary stay faithful to the source? You get back a number, and a number is something you can watch move. That's the whole game: turning "it feels better" into "it scored 0.82, up from 0.74."

Throughout this guide we'll build evals for one running example: **Triage**, a support assistant that takes a customer email and returns two things — a category (`billing`, `bug`, `feature`, `other`) and a drafted reply. It's a good teacher because it has both kinds of output: the category is checkable the old deterministic way, and the reply needs judgment. Most real AI features are exactly this mix.

"Eval-driven development" is the posture, not just the tooling. The way test-driven development lets tests shape the code, eval-driven development lets evals shape the system: you write the eval for the behaviour you want, then change prompts, models, and logic until the score climbs. The eval is the steering wheel. The rest of this guide is learning to drive — the three kinds of eval, where the data comes from, how to score honestly, and how to wire it all into a loop that improves itself. We'll use Evalite (https://evalite.dev), the local-first runner I maintain, but the ideas carry to any tool.

First, why you can't just skip this and eyeball it.

The Demo-to-Production Gap

Getting an impressive AI demo is easy. Getting a system you can *change* without fear is the hard part, and the gap between them is where most AI projects quietly stall.

Here's the trap. You build Triage, it works on the three emails you tried, you ship it. A week later you tweak the prompt to fix a billing misclassification. Did that tweak break the bug-vs-feature distinction? You try a couple of emails, they look fine, you ship again. You've just changed a system where *any* edit can affect *any* output, and your verification was "I tried a few and it felt right."

The problem is that feeling fine and being fine are different measurements, and people are demonstrably bad at telling them apart while iterating. You remember the cases you checked; you don't see the ten you didn't. A prompt change that fixes one category and silently degrades another reads as a pure win, because you only looked where you expected the change. Vibes-based iteration doesn't just fail to catch regressions — it actively hides them behind the confidence of a good demo.

It gets worse with the big moves. The most valuable changes you'll make — swapping to a cheaper model, restructuring the prompt, adding a step — are exactly the ones most likely to shift behaviour across the whole system. With no way to measure, you can't make those changes safely, so you stop making them. The system ossifies around the prompt that happened to demo well, and "we can't touch it, it might break" becomes the team's relationship with its own AI.

An eval suite turns that around. With one, a prompt tweak isn't an act of faith — you run the suite, watch billing go from 0.71 to 0.89 *and* bug hold steady at 0.93, and ship with evidence. A model swap isn't terrifying — it's a number that either holds or doesn't. The suite is what lets you keep changing the system after the demo, which is the only way it ever gets good.

That's the case for doing this. Now let's set up the tool and write the first one.

Setting Up Evalite

Let's get an eval running before we theorize any more. Evalite is local-first and built on Vitest, so if you've written a test you'll be at home. Install it alongside the scorer library we'll lean on later:

```
pnpm add -D evalite vitest
pnpm add autoevals          # ready-made LLM-as-judge scorers, for later
```

An eval lives in a `*.eval.ts` file. Here's the smallest real one for Triage — just the classifier, scored by whether it returned the right category:

```
// triage.eval.ts
import { evalite, createScorer } from "evalite";
import { classify } from "./triage";

const CorrectCategory = createScorer({
  name: "Correct Category",
  scorer: ({ output, expected }) => (output === expected ? 1 : 0),
});

evalite("Ticket classification", {
  data: async () => [
    { input: "My card was charged twice this month", expected: "billing" },
    { input: "The export button does nothing on Safari", expected: "bug" },
    { input: "Could you add dark mode?", expected: "feature" },
  ],
  task: async (input) => classify(input),
  scorers: [CorrectCategory],
});
```

Three parts, and they show up in every eval you'll ever write: **`data`** is the set of cases (an input and, usually, the expected answer), **`task`** is the thing you're testing — your real code — and **`scorers`** are how each output earns a number.

Run it:

```
pnpm evalite watch
```

Evalite runs every case through `task`, scores each one, and opens a dashboard in your browser. You get the score per case, the average, and — the part that makes it stick — a side-by-side of input, expected, and actual output for every case, so when something scores low you can see *why* without adding a single `console.log`. In watch mode it re-runs as you edit, so tightening the prompt becomes a tight loop: change, watch the number move.

That's the whole setup. No cloud account, no dashboards to provision, no rate limits on how often you run — it's a dev dependency and a file. Now let's look properly at the kind of scorer we just wrote, because deterministic scorers are where every good suite begins.

Deterministic Evals

Deterministic evals are the ones that don't need an LLM to judge — a plain function looks at the output and returns a score. They're fast, free, and completely reliable, which makes them the foundation of every suite. Start here, always, and lean on them as hard as the problem allows.

The `CorrectCategory` scorer from the last chapter is the archetype: the output either equals the expected category or it doesn't. But deterministic scorers go far beyond exact-match. For Triage's *drafted reply*, plenty is checkable without any judgment:

```
import { createScorer } from "evalite";

// The reply must never leak an unfilled template placeholder.
const NoPlaceholders = createScorer({
  name: "No Placeholders",
  scorer: ({ output }) => (/\[[A-Z_]+\]/.test(output.reply) ? 0 : 1),
});

// Policy: an auto-draft must never promise a refund.
const NoRefundPromise = createScorer({
  name: "No Refund Promise",
  scorer: ({ output }) =>
    /\b(refund(ed)?|money back)\b/i.test(output.reply) ? 0 : 1,
});

// Shape: the model must return one of our known categories.
const ValidCategory = createScorer({
  name: "Valid Category",
  scorer: ({ output }) =>
    ["billing", "bug", "feature", "other"].includes(output.category) ? 1 : 0,
});
```

Notice the range of what's now testable for free: format (is it valid JSON, does it match the schema), bounds (is the reply between 20 and 200 words), safety and policy (no refunds, no banned phrases), and structure (one of the allowed enums). None of these need a model to judge — and every one is a real way your system can embarrass you in production.

The instinct people get wrong is reaching for an LLM judge to check things a regex could. "Is the category valid?" is not a job for another language model; it's an `includes`. Spend your deterministic budget generously, because these scorers are the cheap, certain backbone — they run in milliseconds, cost nothing, and never flake. Every output your system produces should clear a wall of deterministic checks *before* you spend a token asking an LLM the harder, fuzzier questions.

But some questions really are fuzzy — "is this reply actually helpful?" can't be a regex. That's where the second kind comes in.

LLM-as-a-Judge

Some qualities resist a regex. "Is this reply helpful?" "Does the summary stay faithful to the ticket?" "Is the tone right?" For these you reach for an LLM-as-a-judge: you hand the output (and often a reference) to another model and ask it to score.

The quick win is a ready-made judge. `autoevals` ships several; `Factuality` checks an output against an expected answer:

```
import { Factuality } from "autoevals";
// Scores how factually consistent `output` is with `expected`, via an LLM.
// Just add it to your scorers array; needs an OPENAI_API_KEY.
```

When the built-ins don't fit, write your own — a judge is just a scorer that calls a model. Here's one grading whether Triage's reply actually addresses the customer's problem:

```
import { createScorer } from "evalite";
import { generateText } from "ai";
import { openai } from "@ai-sdk/openai";

const Helpful = createScorer({
  name: "Helpful",
  scorer: async ({ input, output }) => {
    const { text } = await generateText({
      model: openai("gpt-4o-mini"),
      prompt:
        `A customer wrote:\n"""${input}"""\n\n` +
        `Support replied:\n"""${output.reply}"""\n\n` +
        `Does the reply directly address the customer's problem? ` +
        `Answer 0 (ignores it) to 1 (fully addresses it). Reply with only the number.`,
    });
    return Number(text.trim());
  },
});
```

Three things to get right, because a careless judge is worse than none — it hands you a confident number that means nothing:

- **Ask for a scale, with anchors.** "0 to 1" alone invites mush. Tell the judge what 0 means and what 1 means, like the prompt above. Specific anchors make the score reproducible.
- **Use a capable model, and know it has biases.** Judges favour longer answers, their own house style, and the first option in a comparison. Keep judge prompts tight, and blind them to things that shouldn't matter where you can.
- **Treat a judge as a smoke alarm, not a verdict.** A low judge score is a strong signal a human should look — not proof of failure. Sample the disagreements and check the judge against your own opinion now and then; if it's wrong a lot, fix the judge before you trust it.

And mind the bill. Every judged case is a model call, so a 200-case suite with two judges is 400 calls *per run* — fine occasionally, ruinous on every file save. That's exactly the tension the "local, CI, and daily" chapter resolves. For now: deterministic scorers for everything you can, a judge for the genuinely fuzzy rest.

Human Evaluation

There's a third kind of eval, and it's the one people skip because it doesn't automate: a human looks at the output and judges it. For some qualities there's no substitute — subtle tone, brand voice, whether a long answer is genuinely *good* rather than merely correct, the kind of taste you can't fully specify and a judge model only approximates.

Early on, before you have much data, human evaluation is often your *only* real signal. You don't yet know what good looks like well enough to encode it, so you look. The mistake is treating that as unstructured — "I'll just read some outputs." Make it a real eval: a fixed set of inputs, a clear rubric (what's a 1, what's a 5), and a record of the scores over time, so "I read some and they seemed better" becomes a tracked number like everything else.

The expensive-but-honest version is sitting down with a batch and grading against the rubric. Evalite helps here — because the dashboard shows input, output, and expected side by side, it doubles as a review surface; you can read every case for a run without building anything. Do this at milestones, not constantly.

The scalable version is to *capture* human judgment from the people already using your system: your users. A thumbs-up / thumbs-down on each Triage reply is a human eval, collected for free, at a volume you could never produce by hand. You're not asking users to grade on a rubric — you're letting their one-bit reaction stand in for the judgment you'd otherwise pay for. The trick is to record it with enough context (the input, the output, the verdict) that each downvote can later *become* a test case.

Which is the real point of this chapter, and the hinge for the rest of the guide: human evaluation isn't a separate track that stays manual forever. It's the source of your best eval data. The reply a user downvotes today is the case you add to your deterministic-and-judge suite tomorrow, so the system never makes that mistake again. That pipeline — human judgment becoming automated evals — is the flywheel we build a few chapters from now. First, where the rest of the data comes from.

Building a Representative Dataset

Your eval suite is exactly as good as the data you run it on. A perfect scorer over a lazy dataset just tells you your system works great on the three easy cases you happened to write down. The dataset *is* the eval; the scorers only read it.

The first principle is **representativeness**: your cases should look like the traffic your system actually sees, in the same proportions. If a third of real support emails are billing, a third of your eval set should be billing. A dataset that's 90% tidy, well-written questions gives you a reassuring score and a system that falls over on the messy reality of a real inbox. Pull from production the moment you can — one real input is worth ten invented ones.

The second is **edge cases on purpose**. The average case isn't where systems break; the edges are. For Triage that's the email that's two categories at once (a bug report that's really a refund demand), the angry all-caps one, the empty one, the one in another language, the one that's just "hi." Collect these deliberately — each encodes a way your system can fail that the happy path will never reveal.

A few practical rules:

- **Start small and real.** Twenty genuine, varied cases beat two hundred synthetic ones. You can write a useful suite in an afternoon.
- **Grow it from failures.** Every production miss becomes a case (that's the flywheel). Your dataset should get *harder* over time, not stay frozen at launch difficulty.
- **Keep it honest.** Don't quietly drop the cases your system flunks to make the number look better — those are the most valuable rows you own. A dataset you've curated to pass is a dataset that's lying to you.
- **Hold expectations where you can.** Deterministic scorers need an `expected`; judged qualities may only have the input. Both are fine — just know which cases carry a ground truth and which lean on a judge.

The payoff of a good dataset is leverage: once it's representative and edge-heavy, every change you make is tested against the real shape of the problem, not a flattering sample. That's when the number on the dashboard starts to mean something. Which raises the next question — how do you turn a pile of per-case scores into a number you can actually trust?

Scoring Strategies

You've got multiple scorers and a dataset of cases. Now the question is how to turn all those per-case, per-scorer numbers into something you can glance at and trust — and how not to fool yourself doing it.

Averages hide things. A suite that scores 0.85 overall sounds healthy until you slice it: maybe billing is 0.98 and bug is 0.55, and the mean is quietly drowning a category that's failing half the time. Always read scores *by dimension* — per category, per scorer, per slice of input — not just the headline average. The dashboard makes this easy; use it. The number you ship on should be the worst important slice, not the comforting mean.

Not every scorer deserves equal weight. "Valid category" being wrong is a hard failure — the output is unusable. "Reply slightly too long" is a nitpick. If you collapse scorers into one number, weight them to reflect that; better, keep the critical ones as separate pass/fail gates and let the soft ones be the gradient. Don't let a great helpfulness score paper over an invalid-category rate.

Thresholds turn scores into decisions. A score is just information until you decide what to do with it. Set explicit bars: deterministic safety scorers (no refund promises, valid category) must be 1.0 — anything less blocks the change. Judged qualities get a floor you don't regress below ("helpfulness stays above 0.8"). Now the suite answers a real question — *can I ship this?* — instead of emitting a vibe in numeric form.

Watch deltas, not absolutes. The absolute score of a judged eval is noisy and a little arbitrary — 0.82 helpfulness doesn't mean much on its own. The *change* is what's real: 0.82 down from 0.88 after a prompt edit is a regression worth investigating, even though 0.82 sounds fine. Eval-driven development is mostly watching deltas as you change things, which is why running the same suite repeatedly matters more than any single run's number.

The honest summary: a good score is one you've tried to break. Slice it, weight it, gate it, and track its movement — and be most suspicious of the comfortable average no one's pulled apart. Next, how to run all this without going broke or grinding to a crawl.

Local, CI, and Daily Runs

Evals cost two things: time and tokens. Deterministic scorers are nearly free of both; LLM judges are not. Run everything on every keystroke and you'll either wait forever or burn a fortune — so the practical art is matching cadence to cost. Three tiers.

Local, on every change — the fast set. While you're iterating on a prompt, you want a sub-second loop. Run only the deterministic scorers and a small, hand-picked slice of cases — ten or twenty that cover the main shapes. No judges, no full dataset. This is `evalite watch` running while you work: change the prompt, see the deterministic score move instantly, no token spend. It won't catch everything, but it catches the obvious breaks the moment you make them.

CI, on every PR — the gate. When a change is up for review, run the full deterministic suite plus judges over a representative subset, and *gate the merge on it*. This is where the thresholds from the last chapter earn their keep: safety scorers must be 1.0, helpfulness must not regress, or the check goes red. It costs some tokens per PR — a fine price for never merging a silent regression. Because Evalite sits on Vitest, it drops into the same CI step as your normal tests.

Daily (or pre-release) — the full sweep. Once a day, or before a release, run the whole dataset through every scorer including the expensive judges. This is your comprehensive health check — the run that catches slow drift and the rare-category regression a subset would miss. Schedule it, post the result where the team sees it, and treat a drop as a real signal.

The split solves the tension that makes people abandon evals: "too slow to run constantly, too important to run never." You run the cheap thing constantly and the expensive thing deliberately. And there's a cost lever worth knowing — judge with a smaller, cheaper model on the frequent runs and a stronger judge on the daily sweep, trading a little judge accuracy for a lot of budget on the runs that happen most.

That's the operational backbone. Now the part that makes a suite get *better* on its own.

The Data Flywheel

Here's the mechanism that separates an AI system that slowly improves from one that slowly rots: the data flywheel. It's the loop where *using* the system generates the data that *improves* the system, and it runs on the eval suite you've now got.

The loop, concretely, on Triage:

1. A customer emails; Triage classifies it and drafts a reply.
2. The reply is wrong — it reads a refund demand as a feature request — and the support agent (or the customer) thumbs it down.
3. That downvote, captured with its input and output, lands in a review queue.
4. You turn it into an eval case: the email as `input`, `billing` as the `expected` category, added to the dataset.
5. You change the system — prompt, examples, logic — until the suite passes that case *and* all the others.
6. You ship. The next time a similar email arrives, Triage gets it right.

Each turn of that loop, the dataset gets a little more representative of reality and a little harder, and the system gets a little better at exactly the things it was getting wrong. The failures aren't embarrassments to bury — they're the highest-quality data you have, because they're real and they're the cases you actually got wrong.

The critical, skippable step is #4 — turning the failure into a *permanent* case. Fixing the one email in production and moving on teaches the system nothing; the same class of mistake recurs forever. Promoting it to an eval case is what makes the fix stick: you didn't patch an instance, you closed a gap, and the suite now guards it for good. It's the same instinct as the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide's Solve loop and the learn-loop in Context Engineering (https://rogerstringer.com/guides/context-engineering-the-craft-of-agents-md) — fix the system, not the symptom.

The infrastructure this needs is modest and worth building early: a feedback control on every output, and a path from "thumbs down" to "row in the dataset" with low enough friction that you actually walk it. Get that pipeline running and your eval suite stops being a thing you maintain and becomes a thing that grows itself — sharper every week, fed by the only data that truly matters, which is your real users hitting your real edges.

Evals as a Regression Net

Once your suite is representative and gated, it quietly becomes the most valuable thing you own: a regression net that lets you change anything without fear. This deserves its own chapter because it inverts how scary the big moves feel.

The changes that most improve an AI system are the ones that most terrify teams without evals: swapping the model, rewriting the prompt, adding or removing a step. Each can shift behaviour across the entire system in ways no spot-check will find. With a suite, each becomes a measurement instead of a leap.

Model swaps are the headline case. A new model drops, cheaper and supposedly better. Without evals, adopting it is a gamble you'll spend weeks nervously hand-validating. With evals, you change one line, run the suite, and read the truth: helpfulness holds, classification ticks up, but the no-refund-promise scorer drops because the new model is chattier about refunds — so you tighten that part of the prompt and re-run. A multi-week act of faith becomes an afternoon with a number at the end. The model-cost economics the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide talks about only become *capturable* when you can safely chase cheaper models, and the suite is what makes it safe.

Prompt changes are the everyday case. Every prompt edit is a potential regression somewhere you're not looking; the suite looks everywhere at once. Fix billing classification, run the suite, confirm bug and feature didn't move, ship. The net catches the silent trade-off that vibes-based iteration hides.

Provider and dependency drift is the invisible case. Models change under you even when you don't touch them — a provider updates a model behind the same name, and your behaviour shifts with no commit to blame. The daily full-suite run is what surfaces that drift, turning "users say it got worse and we don't know why" into a dated drop on a chart you can correlate.

The deeper point is the one from the first chapter, now cashed out: evals are what let you *keep changing the system after it ships*. Without them, every AI system trends toward frozen — too risky to touch. With them, change is cheap and measured, and a system you can safely change is the only kind that gets better over time. The net isn't there to slow you down. It's what lets you move fast without breaking things you can't see.

Eval-Driven Development in Practice

Let's run the whole loop on one real change to Triage, the way you'd actually do it — evals in the driver's seat from the start.

The change: support wants Triage to stop drafting replies that promise things it can't, after one too many "we'll have that fixed by Friday" auto-replies. New rule: replies may acknowledge and route, never commit to timelines or fixes.

1. Write the eval first. Eval-driven, like test-driven — the check comes before the change. A deterministic scorer for the hard rule, plus a few real cases that currently fail:

```
const NoCommitments = createScorer({
  name: "No Commitments",
  scorer: ({ output }) =>
    /\b(by (monday|tuesday|friday|tomorrow)|we'?ll (fix|have|release)|will be (fixed|
done))\b/i
      .test(output.reply) ? 0 : 1,
});
```

Add it to the suite and run. It scores low — the current prompt cheerfully makes promises. Good: a failing eval that captures exactly the behaviour you want to change. That red number is your target.

2. Change the system to move the number. Edit the prompt — add the rule, maybe two examples of acknowledge-don't-commit. `evalite watch` re-runs on save; you watch `No Commitments` climb toward 1.0 in real time. This is the tight loop the whole guide has been building to: change, watch the score, repeat.

3. Check you didn't break anything else. Here's the moment that justifies the entire practice. `No Commitments` is now 1.0 — but the suite shows `Helpful` dropped from 0.86 to 0.71. Suppressing commitments made the replies curt and useless. Vibes-based iteration would have shipped the win and never seen the cost; the suite caught the trade-off the instant you made it. You adjust — teach the prompt to be warm and concrete *without* committing — and re-run until both hold: `No Commitments` at 1.0, `Helpful` back to 0.85.

4. Gate and ship. Push the PR. CI runs the full suite, the new scorer and the old ones all clear their thresholds, the check goes green, and you merge with evidence rather than hope. The new case lives in the dataset forever — the regression net just got one notch finer.

5. Close the loop. A week later a reply slips through with "we'll sort this out shortly." Your regex missed that phrasing. Don't just fix the prompt — add the case, widen the scorer, re-run. The flywheel turns; the suite gets sharper.

That's eval-driven development end to end: the eval defined the goal, drove the change, caught the side effect, gated the merge, and absorbed the next failure. The system got better and you have the numbers to prove it — which, in a probabilistic world, is the only kind of "better" that means anything.

Building Your Eval Suite

You don't need a perfect suite to start — you need a running one, this week, that you grow. Here's the path, in order of leverage.

- **Pick one AI feature and write three cases.** Not a framework, not a plan — three real inputs with their expected outputs, in a `*.eval.ts` file. `pnpm add -D evalite`, write the file, `pnpm evalite watch`. You'll have a number on the board in twenty minutes.
- **Wall it with deterministic scorers.** Before any LLM judge, encode everything checkable for free: valid shape, bounds, format, the safety and policy rules you can never violate. This is the cheap, certain backbone, and most teams under-invest in it.
- **Add one judge for the fuzzy thing that matters most.** Helpfulness, faithfulness, tone — whatever your feature lives or dies on. One good judge with anchored scoring beats five vague ones. Calibrate it against your own opinion before you trust it.
- **Make it representative, then make it hard.** Pull real inputs in place of invented ones, then deliberately add the edge cases — the ambiguous, the angry, the empty, the multilingual. Twenty real cases beat two hundred synthetic.
- **Wire the cadence.** Fast deterministic set on every change, full suite gated in CI, the whole thing including judges on a daily sweep. Set thresholds so the suite answers "can I ship?"
- **Build the flywheel.** A feedback control on every output, and a low-friction path from "thumbs down" to "new eval case." This is what makes the suite grow itself instead of decaying.

The mindset underneath all of it: in deterministic software you write tests to prove the code works; in probabilistic software you write evals to *steer* a system you can't fully predict. The eval is the steering wheel, the score is the road, and eval-driven development is just keeping your hands on the wheel as you change things — which, with a system that can shift under you at any moment, is the difference between driving and being driven.

Let the machine generate; you decide what "good" means and measure relentlessly toward it. That measurement is the scarce skill now — anyone can wire up an LLM, but the person who can *prove* theirs is getting better is the one whose system actually does.

If you want the surrounding discipline: the 70/30 Engineer (https://rogerstringer.com/guides/the-70-30-engineer) guide for verifying AI work in general, Context Engineering (https://rogerstringer.com/guides/context-engineering-the-craft-of-agents-md) for steering the agents that write your code, and Evalite (https://evalite.dev) itself when you're ready to go deep. Now go put a number on something that's only ever been a vibe.

About Roger

I'm Roger Stringer — I build things, break them, and write up what I learned so you don't have to learn it the hard way. These Field Guides come straight out of that work.

Working on something bigger? I take on a handful of fractional CTO (https://rogerstringer.com/guides/the-fractional-cto-field-guide) engagements — helping founders and teams set technical direction, build AI-powered workflows, and actually ship the hard parts. If you're wrestling with the kind of problem this guide covers and want someone in your corner who's done it before, that's exactly what I help with. Drop me a line. (mailto:roger.stringer@hey.com)

And if a guide helped, got something wrong, or you just want to compare notes, I'd love to hear from you:

- **Email** — roger.stringer@hey.com (mailto:roger.stringer@hey.com)
- **X** — @freekrai (https://x.com/freekrai)
- **GitHub** — github.com/freekrai (https://github.com/freekrai)
- **LinkedIn** — linkedin.com/in/rogerstringer (https://www.linkedin.com/in/rogerstringer)

New guides go up as I hit problems worth documenting — follow along wherever suits you.