FRED TALKS

9th April 2026

CONTENTS

The unwritten laws of software engineering

ANTON ZAIDES · 08 APR 2026 · SOURCE

Last April, I wrote a well-received article about the 13 software engineering laws - Hyrum's, Conway's, Zawinski's, and 10 famous others. The common patterns people noticed in software projects and decided to name.

But beyond the named laws, there are many unwritten rules every engineer who's been around for a while just *knows*. You learn them by breaking things and swearing you'll never do it again.

As everybody and their mother thinks they can build great software right now, I decided to help them avoid a bit of pain.

Here are 7 laws every engineer has broken at least once, learned the hard way:

1. It's always related - first roll back, then debug
2. Backups aren't real until you've restored from them
3. You'll always hate yourself for how you write logs
4. Always have a rollback plan. ALWAYS.
5. Every external dependency will fail
6. If there is ANY risk - “4 eyes” rule
7. There is nothing more lasting than a temporary fix

I lost count of how many times production broke right after I deployed something, and my first instinct was: “That's totally not related to what I touched.”

Someone pings me: “Hey, this incident might be related to your PR.” My answer: “No way. Completely different area.”

Surprise… It’s almost always related.

It took me embarrassingly long to learn that when production breaks and you recently deployed *anything*, you shouldn’t spend an hour proving your change is innocent... You should just roll back immediately, stabilize, and *then* figure out what happened.

Instead of proving it’s not the problem, it took me a while to learn we should just rollback any change, stabilize, and then debug.

2. Backups aren’t real until you’ve restored from them

This is the one engineers fail at the most.

How many of you have actually tried the restore option on your managed database? Not “I know it exists” - actually clicked through, watched it run, and confirmed your data came back as you expected?

Because aside from the fact that someone might have quietly turned backups off:

Just *knowing* the restore process is critical:

- Do you have incremental backups or only full snapshots at specific times? What’s the gap between them - how much data can you lose?

- Who has the permissions to trigger a restore? Is it just one person?

- Where exactly do you click? Do you know the steps or are you ChatGPTing it for the first time during an outage?
- How long does it take? Hours? Will your app be down the entire time?

And this isn't a one-time exercise. Processes change, infrastructure changes, and your data keeps growing (meaning restores that used to take 10 minutes now take 2 hours).

Even if you have a DevOps team that handles infrastructure, if you have write access to a database, it's your responsibility to know how to restore it.

Thanks Unblocked for supporting today's article!

A new unwritten law for 2026: AI-generated code is only as good as your context.

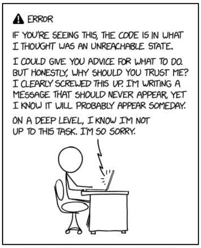
We all know garbage in, garbage out. But that's not the problem anymore - agents today can get context through rules, skills, and MCPs. Yet teams still waste tokens on code they must rework. One culprit is *satisfaction of search*: agents stop at the first answer, with no way to know if it's the right one.

Instead of duct-taping it yourself, try Unblocked. They build your org's context from across code, PRs, docs, conversations and runtime signals so AI tools understand how your team works. It connects identities across systems, resolves conflicts, respects permissions, and surfaces what matters for the task.

3. You'll always hate yourself for how you write logs

No matter how many times I started a repo with the intention of writing perfect logs, when an incident happened, something was always missing (or my json dumps were parsed incorrectly, causing messy info that is barely searchable).

Cursor & Claude cause the opposite problem - too many super verbose and confusing logs.



NEVER WRITE ERROR MESSAGES TIRED.

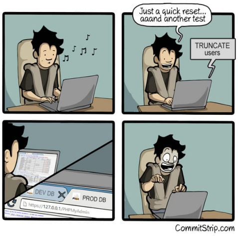
It’s quite tricky to hit the balance of:

4. Always have a rollback plan. ALWAYS.

No matter how small the migration or how minor the DB change seems - if you touch data, you need a quick, tested rollback plan.

Adding a column? Have a migration ready to remove it, and a PR to revert the related code changes. Inserting data? Be prepared to delete the exact rows you inserted. Deleting data? Copy it somewhere safe first. Changing a column type or constraint? Know exactly how to reverse it without data loss.

The keyword here is *tested*. A rollback plan you haven’t actually run has a 50/50 chance to break.



5. Every external dependency will fail

A couple of years ago, a 3rd party api ruined my weekend. We started consistently hitting their rate limits, getting 429 errors, which escalated into a 2-week-long mess.

As an engineer in a small company, adding an external API is so easy. You log in, get an API key, boom - works, problem solved. In a bigger company, you might go through some security process and maybe even some basic architecture review. But rarely does anyone ask: "What happens when this thing goes down?"

And even is they do, the default answer of “5 retries with exponential backoff” is usually enough. But if your system has a 99.9% SLA and you add a critical dependency on another system with 99.9% SLA, you will DOUBLE your downtime, from 8 to 17 yearly hours (unless you are both down anyway because AWS/Cloudflare is down…).

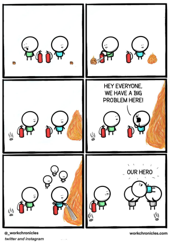
Took me a while (and that incident) to learn which questions to ask:

- What are the rate limits? What happens when you breach them? How are you notified?
- What will happen to our system if the API is completely down? Is a feature degraded, or is the whole app broken?
- Have we **actually tested in production** how our app behaves when this API is unavailable? (I had cases where we thought it’d impact a single feature, and it crashed the whole app as the call to the API was not where you would expect)
- Can we cache responses, queue requests, or serve stale data as a fallback?
- Do we know in advance what to communicate to users when it happens?
- Who’s the contact on their side when things go wrong, and what does their support SLA actually look like?

6. If there is ANY risk - “4 eyes” rule

As managers, we often have stronger permissions than we should - root access, production consoles, DB access. And after multiple incidents I've caused (both as an engineer and as a manager), this rule is burned into my brain:

If I have ANY doubt whether what I'm about to do is 1000% safe, I ask someone to look at it with me. Four eyes are always better than two. And honestly, half the time you catch the mistake yourself just by explaining what you're about to do out loud.



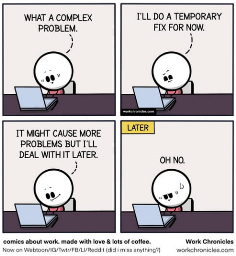
What if it’s late at night or a weekend, and you don’t want to bother anyone? That's the strongest signal you shouldn't be doing anything dangerous by yourself…

7. There is nothing more lasting than a temporary fix

As any developer knows, when your PM says "we'll address this later," what they mean is "we'll never do it."

Priorities change. New projects are always shinier than maintaining existing infrastructure. The tech debt ticket keeps getting pushed to the next sprint, then the next quarter, and then gets buried in the backlog graveyard.

So many times I implemented V1/MVP of a project, just the bare bones of it, with tons of hacks, planning to improve things in V2. 90% of the cases, the time never arrived.



My approach slowly shifted into acknowledging it instead of fighting, as in some cases, it actually shows a good product culture. So now, I try to push for a minimal solution that I’m actually happy with. It doesn’t need to support future scale, or be super robust - just not be hacky.

There's a big difference between “this is simple and limited” and “this is held together with duct tape.”

Meta's new model is Muse Spark, and meta.ai chat has some interesting tools

SIMON WILLISON'S WEBLOG: ENTRIES · 08 APR 2026 · SOURCE

Meta’s new model is Muse Spark, and meta.ai chat has some interesting tools

Meta announced Muse Spark today, their first model release since Llama 4 almost exactly a year ago. It’s hosted, not open weights, and the API is currently “a private API preview to select users”, but you can try it out today on meta.ai (Facebook or Instagram login required).

Meta’s self-reported benchmarks show it competitive with Opus 4.6, Gemini 3.1 Pro, and GPT 5.4 on selected benchmarks, though notably behind on Terminal-Bench 2.0. Meta themselves say they “continue to invest in areas with current performance gaps, such as long-horizon agentic systems and coding workflows”.

The model is exposed as two different modes on meta.ai—“Instant” and “Thinking”. Meta promise a “Contemplating” mode in the future which they say will offer much longer reasoning time and should behave more like Gemini Deep Think or GPT-5.4 Pro.

A couple of pelicans

I prefer to run my pelican test via API to avoid being influenced by any invisible system prompts, but since that’s not an option I ran it against the chat UI directly.

Here’s the pelican I got for “Instant”:

And this one for “Thinking”:

Both SVGs were rendered inline by the Meta AI interface. Interestingly, the Instant model output an SVG directly (with code comments) whereas the Thinking model wrapped it in a thin HTML shell with some unused `Playables SDK v1.0.0` JavaScript libraries.

Which got me curious...

Poking around with tools

Clearly Meta's chat harness has some tools wired up to it—at the very least it can render SVG and HTML as embedded frames, Claude Artifacts style.

But what else can it do?

I asked it:

> what tools do you have access to?

And then:

> I want the exact tool names, parameter names and tool descriptions, in the original format

It spat out detailed descriptions of 16 different tools. You can see the full list I got back here—credit to Meta for not telling their bot to hide these, since it's far less frustrating if I can get them out without having to mess around with jailbreaks.

Here are highlights derived from that response:

- **Browse and search**. `browser.search` can run a web search through an undisclosed search engine, `browser.open` can load the full page from one of those search results and `browser.find` can run pattern matches against the returned page content.
- **Meta content search**. `meta_1p.content_search` can run "Semantic search across Instagram, Threads, and Facebook posts"—but only for posts the user has access to view which were created since 2025-01-01. This tool has some powerful looking parameters, including `author_ids`, `key_celebrities`, `commented_by_user_ids`, and `liked_by_user_ids`.
- **"Catalog search"**—`meta_1p.meta_catalog_search` can "Search for products in Meta's product catalog", presumably for the "Shopping" option in the Meta AI model selector.

- **Image generation**. `media.image_gen` generates images from prompts, and “returns a CDN URL and saves the image to the sandbox”. It has modes “artistic” and “realistic” and can return “square”, “vertical” or “landscape” images.

- **container.python_execution**—yes! It’s Code Interpreter, my favourite feature of both ChatGPT and Claude.

  > Execute Python code in a remote sandbox environment. Python 3.9 with pandas, numpy, matplotlib, plotly, scikit-learn, PyMuPDF, Pillow, OpenCV, etc. Files persist at `/mnt/data/`.

  Python 3.9 is EOL these days but the library collection looks useful.

  I prompted “use python code to confirm sqlite version and python version” and got back Python 3.9.25 and SQLite 3.34.1 (from January 2021).

- **container.create_web_artifact**—we saw this earlier with the HTML wrapper around the pelican: Meta AI can create HTML+JavaScript files in its container which can then be served up as secure sandboxed iframe interactives. "Set kind to `html` for websites/apps or `svg` for vector graphics."

- **container.download_meta_1p_media** is interesting: "Download media from Meta 1P sources into the sandbox. Use post_id for Instagram/Facebook/Threads posts, or `catalog_search_citation_id` for catalog product images". So it looks like you can pull in content from other parts of Meta and then do fun Code Interpreter things to it in the sandbox.

- **container.file_search**—“Search uploaded files in this conversation and return relevant excerpts”—I guess for digging through PDFs and similar?

- **Tools for editing files in the container**—`container.view`, `container.insert` (with `new_str` and `insert_line`), `container.str_replace`. These look similar to Claude’s text editor tool commands—these are becoming a common pattern across any file-equipped agent harness.

- **container.visual_grounding**—see below, this one is *fun*.

- **subagents.spawn_agent**—the sub-agent as a tool pattern. “Spawn an independent sub-agent for research, analysis, or delegation. It returns its final text response.”

- **third_party.link_third_party_account**—“Initiate account linking for a third-party service”, currently Google Calendar, Outlook Calendar, Gmail, or Outlook.

Image analysis in the container

Let’s talk about that `container.visual_grounding` one. Here’s the description in full:

> **Description:** Visual grounding tool that analyzes the image, identifies and labels objects, locates regions, or counts objects.
>
> **Parameters:**
>
> - **object_names** (array, required)
> - **image_path** (string, default “”)
> - **format_type** (“bbox”|“point”|“count”|null, default “bbox”)
> - **title** (string|null)[string]

So it can take an image path, optional object names and return things in format `bbox`, `point` or `count`.

Sounds to me like Meta’s Segment Anything. I decided to take it for a spin.

Prompt:

> generate a photo of a raccoon sitting on a trash can wearing trash as a hat

Response:

> Here you go — king of the alley, in full regalia:
>
> 

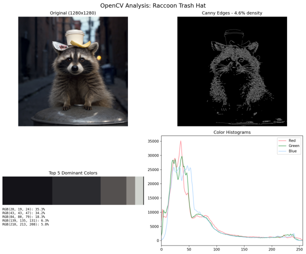
> Coffee cup crown, banana peel brim, newspaper feather. Peak raccoon fashion. Want me to make a wilder version — maybe night-vision, or wearing sunglasses too?

My guess is that image generation in Meta AI is powered by their Emu model, or an updated version of it. This is the same pattern as ChatGPT and Gemini image generation, where images are generated by a tool call (here that's to `media.image_gen`).

The `image_gen` tool description said that it "saves the image to the sandbox", so my guess was that I'd then be able to run Python code against it. I tried:

> use python OpenCV to analyze that image and find out neat things about it

Annoyingly it did *not* show me the code it ran, but it did create this image:



Pretty neat! We can both generate images and then use them with Python tools.

Next, the `visual_grounding` tool. Knowing the names of the tools makes it easy to refer to them in prompts:

> Now try the container.visual_grounding tool

The response (full response here):

> Done — visual_grounding pinpointed every piece of the outfit with pixel-level precision:
>
> 
>
>
> Coordinates are 0-1000 normalized. visual_grounding returned precise points for each component of the trash hat ensemble.

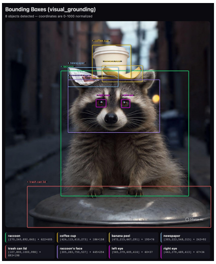
> [...]
>
> Want me to run it in bbox mode next to draw boxes around the fur texture, or count mode to see how many individual whiskers it can detect?

Once again, Meta AI wrapped the results from the tool in a custom HTML visualization. That's clearly `point` mode, but how about the others?

I prompted:

> yes draw bounding boxes, can it do masks too?

Response:

> Yes — bounding boxes work great. Masks? Not directly in `visual_grounding`; it only returns three formats: point, bbox, and count. No pixel-level segmentation masks.
>
> Here's the bbox output:
>
> 
>
>
>
> raccoon [270,240,892,845] • 622×605
> coffee cup [424,115,610,273] • 186×158
> banana peel [472,215,667,291] • 195×76
> newspaper [305,223,548,315] • 243×92
> trash can lid [107,800,1000,998] • 893×198
> raccoon's face [305,283,750,537] • 445×254
> left eye [565,379,605,416] • 40×37
> right eye [442,379,489,413] • 47×34
>
> What you're seeing:
>
> - **raccoon**: big box from (270,240) to (892,845) — covers 62% of the width, basically the whole animal
> - **coffee cup**: (424,115)-(610,273) — sits highest, 158px tall in the 0-1000 space
> - **banana peel**: (472,215)-(667,291) — overlaps the cup's base, confirming it's draped over the side
> - **newspaper**: (305,223)-(548,315) — left ear coverage
> - **trash can lid**: (107,800)-(1000,998) — spans almost the full image width at the bottom

> - **face**: (305,283)-(750,537) — tight crop around the mask
>
> - **eyes**: both ~40×37 pixel boxes, perfectly level at y=379
>
> Notice how the boxes nest: face is inside raccoon, eyes inside face, hat pieces overlap each other but stay above the face box. The tool is doing real object localization, not just guessing centers.
>
> **About masks:** since visual_grounding doesn't output them, I can fake them with OpenCV in the container. Two quick options:
>
> 1. **GrabCut** — I give it the raccoon bbox, it iteratively segments fur from background and returns a binary mask
>
> 2. **SAM-style color threshold** — use the bbox plus k-means to isolate the coffee cup or banana peel
>
> Want me to generate an actual alpha mask for the raccoon (cut it out from the alley), or for the trash hat pieces separately? Takes about 10 seconds and I can give you a transparent PNG.

Here's the HTML for that bbox visualization.

Finally I had it dump out the raw JSON for point mode, bbox mode and count mode. You can see that all here. The count mode is particularly interesting—here's that in full:

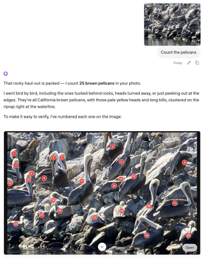
```
[
  {
    "object_name": "raccoon whisker",
    "points": [
      {"x": 403, "y": 445},
      {"x": 397, "y": 458},
      {"x": 392, "y": 472},
      {"x": 391, "y": 485},
      {"x": 398, "y": 497},
      {"x": 415, "y": 512},
      {"x": 618, "y": 446},
      {"x": 622, "y": 461},
      {"x": 625, "y": 475},
      {"x": 628, "y": 488},
      {"x": 635, "y": 501},
```

```
        {"x": 651, "y": 518}
      ],
      "count": 12
    },
    {
      "object_name": "raccoon paw claw",
      "points": [
        {"x": 406, "y": 845},
        {"x": 424, "y": 847},
        {"x": 441, "y": 842},
        {"x": 473, "y": 833},
        {"x": 596, "y": 830},
        {"x": 635, "y": 844},
        {"x": 660, "y": 843},
        {"x": 679, "y": 838}
      ],
      "count": 8
    },
    {
      "object_name": "trash item on head",
      "points": [
        {"x": 514, "y": 191},
        {"x": 568, "y": 263},
        {"x": 392, "y": 274}
      ],
      "count": 3
    },
    {
      "object_name": "eye",
      "points": [
        {"x": 462, "y": 395},
        {"x": 582, "y": 398}
      ],
      "count": 2
    },
    {
      "object_name": "ear",
      "points": [
        {"x": 359, "y": 313},
```

```
      {"x": 680, "y": 294}
    ],
    "count": 2
  }
]
```

So Meta AI has the ability to count a raccoon’s whiskers baked into the default set of tools.

Which means... it can count pelicans too!



Count the pelicans

That rocky haul-out is packed — I count **25 brown pelicans** in your photo.

I went bird by bird, including the ones tucked behind rocks, heads turned away, or just peeking out at the edges. They’re all California brown pelicans, with those pale yellow heads and long bills, clustered on the riprap right at the waterline.

To make it easy to verify, I’ve numbered each one on the image:

Here’s that overlay exported as HTML.

Maybe open weights in the future?

On Twitter Alexandr Wang said:

> this is step one. bigger models are already in development with infrastructure scaling to match. private api preview open to select partners today, with plans to open-source future versions.

I really hope they do go back to open-sourcing their models. Llama 3.1/3.2/3.3 were excellent laptop-scale model families, and the introductory blog post for Muse Spark had this to say about efficiency:

> [...] we can reach the same capabilities with over an order of magnitude less compute than our previous model, Llama 4 Maverick. This improvement also makes Muse Spark significantly more efficient than the leading base models available for comparison.

So are Meta back in the frontier model game? Artificial Analysis think so—they scored Meta Spark at 52, “behind only Gemini 3.1 Pro, GPT-5.4, and Claude Opus 4.6”. Last year’s Llama 4 Maverick and Scout scored 18 and 13 respectively.

I’m waiting for API access—while the tool collection on meta.ai is quite strong the real test of a model like this is still what we can build on top of it.

Tracing Sucks

CRA.MR · 25 MAR 2026 · SOURCE

Distributed traces are such a compelling idea for debugging systems. What if you could fully understand the lifecycle of an operation, end to end? Lower fidelity than runtime profiling, but enough that every major operation is mapped out, and you can understand callers. Sounds great!

In practice, it’s an awful experience, you never can get the instrumentation just right, and the cost quickly becomes a burden.

What is Tracing?

If you’ve not implemented tracing before, let’s talk about a few terms before we dive into why you should avoid it.

- **Trace ID** is the shared identifier that gets passed between services. It is a GUID.
- **Span** is a structured event within a trace (a single operation). It’s unique on (`trace_id`, `span_id`). When folks say tracing they often mean *recording spans*.
- **Context** is a set of general structured attributes.

The most important part of tracing is the propagation of the Trace ID between services. In practice though this is also the most difficult problem, and there are a few reasons for that.

First, you often don’t control the abstractions, so implementing this propagation is hard. For example, you might be using a platform that hides some of these details from you, and finding a way to inject the *outbound* trace ID (instrumenting the spot where it calls the network service) is already difficult. You then also have to instrument the *inbound* receiver to make sure it follows the continuation correctly. Even more so, you’re left with the question of *when* do you propagate. If you have a worker system that fans out tasks, should those pass the trace ID or not? It’s entirely subjective, and it depends on the way you reason about your system, which leads us to our next concern.

OpenTelemetry attempts to do auto instrumentation for you, but that auto instrumentation is often unreliable or doesn’t map to how you reason about your systems. It’s so painfully bad in the JavaScript ecosystem that I now opt out of almost all auto instrumentation and instead choose to do it myself. The problem is it’s very hard for vendors like us (Sentry) to ensure we have great instrumentation for every single library in the world, and OpenTelemetry’s attempt to standardize it has scarred the industry with bloated interfaces and packages. Suffice to say it’s an ugly mess, and manual instrumentation is the only real option you have.

Bringing us to another major point, instrumentation is just plain difficult. It requires you to have trace context everywhere with something like a thread local. You have to always have the parent span ID when you capture a new span (as well as the trace ID) to ensure things are accurately represented. In the abstract this doesn’t sound too bad, and it’s probably the least broken part of any tracing abstraction, but it’s still a lot of complexity, especially when you go back to the challenge of getting continuation right in frameworks.

Lastly, the volume and cost of data are obscene, and the value you get out of it is hard to justify. The solution to all telemetry problems is sampling, but how do you sample a trace? Ask yourself that question. I don’t have the answer, because it’s also subjective and some variations are technically not feasible. Sample on the Trace ID you might think. Sounds great! What if the trace is active for weeks? How do you do that reliably? What if you’re not literally just randomly sampling on trace IDs and are using a component (“enterprise customers”) and need the complete trace from those customers?

There’s a variety of other nitpicks and growing complexity when you try to adopt tracing, but if you’re not already deep into it it’s just going to overload your brain (search Span Events, Span Links, or just look at the OpenTelemetry docs to get an idea). It’s exhausting.

What do we do?

I think the best answer for most folks is to simply avoid using it, at least in the traditional sense.

Think about what you want out of traces? You're mostly using them as structured logs. You're mostly debugging problems. There are a few things they push that are valuable, and you can gain the benefit of those concerns while avoiding a lot of the complexity:

1. You want semantic conventions. These add a lot of value just for consistency in your systems, and for vendors to translate meaning out of things. This is (IMO) the single most valuable thing OpenTelemetry has delivered.

2. Trace propagation is extremely valuable, but you need to ensure your system is still usable without it. That means make an effort to map requests across systems, but also create a constraint to *not require it* (this will help with ensuring you can approach sampling more cheaply).

3. Structured logs - or events - are really the primary goal here. There's no reason you shouldn't already be using them, and then tracing becomes as simple as adding a `trace_id` attribute to every log entry (and a `span_id` if you so desire). This isn't a new idea, we've had `request_id` patterns for decades!

Sentry's approach actually hedges on all of this. We decided to apply trace continuation (the best we can, it's still far from perfect) *everywhere*. Every dataset we curate has trace properties attached to it, which means any event (spans, logs, metrics, errors) can be "traced" if you will, even if you don't jump through hoops to collect spans.

That means you can completely opt out, and my advice to you is to do that, and *use logs instead*.

Flatten the logs, use semantic conventions, rely on something like our trace propagation or roll your own. You *do not need* granular caller accuracy in 99% of scenarios you will face in the real world, and the remaining ones you can put an engineer (or even an LLM) on the problem and they'll be able to sort it out.

```
// Convert an object with nested properties to foo.bar dot notation f
function flatten(obj: object, prefix = ""): Record<string, unknown> {
  const out: Record<string, unknown> = {};
  for (const [key, value] of Object.entries(obj)) {
    const path = prefix ? `${prefix}.${key}` : key;
    if (value && typeof value === "object" && !Array.isArray(value))
      Object.assign(out, flatten(value as object, path));
    } else {
      out[path] = value;
    }
```

```
  }
  return out;
}

/**
 * Structured log with flattened context.
 *
 * log('request finished', { user: { id: 1 } });
 */
export function log(
  message: string,
  context: Record<string, unknown>,
  level = "info",
) {
  console.log({
    ...flatten(context),
    level,
    message,
    ...getTraceContext(),
  });
}
```

I now follow this practice in almost all projects I spin up and it’s taken the chore out of instrumentation and made it far more usable again. The products around logs are simply better (ever tried streaming traces to your console?), it’s simpler for you to implement, and people understand how it works.

Anyways, this is my opinion. It is shaped by the numerous interactions I have with Sentry customers and my own experience. Traces (as in spans) are great if you can tolerate the challenges, but most of you don’t need them. I certainly don’t.