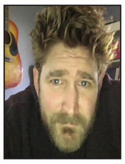
iVoyeur
We Don't Need Another Hero

DAVE JOSEPHSEN



Dave Josephsen is the sometime book-authoring developer evangelist at Librato.com. His continuing mission: to help engineers worldwide close the feedback loop. dave-usenix@skeptech.org

Someone recently asked me this question: "What's the first thing that comes to your mind when you hear the word 'DevOps'?"

A loaded question, I agree, and of course I lied, and made up something about the "first way." I mean really, if you want a possibly embarrassing answer to a loaded question, you really should confront me face-to-face with it in a public place. If the asker of that question had done so, I would have had to answer honestly that the first thing I think about when I hear the word "DevOps" is Brent from The Phoenix Project novel [1].

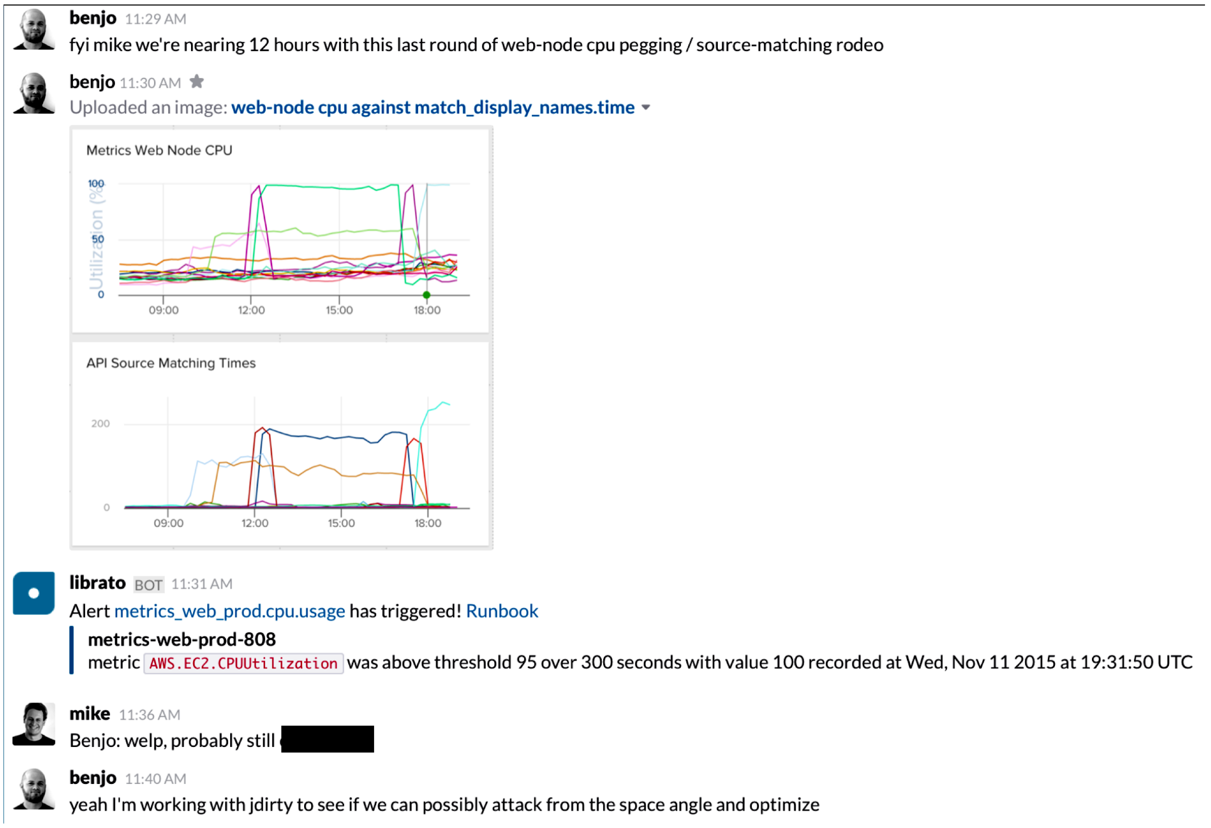
If you haven't read it, let me explain: Brent is probably you. The one person who knows how all the stuff actually works, and who everyone depends on to fix things when they go sideways. Brent is a hero. And because the book is about DevOps, and DevOps abhors constraints and local optimization (in other words, because DevOps hates heroes), Brent is basically a huge organizational problem.

I was deeply hurt by this plot device on my first reading. In fact, if I hadn't been trapped on a plane with nothing else to read, I probably would not have finished *The Phoenix Project* because of it, which would have been my loss. It's just hard to wrap your head around how a hero like Brent could be bad for the IT organization as a whole (especially when I relate so strongly to him). As a result, I also had a hard time wrapping my head around the endgame. Sometimes it seems like all anybody talks about is "improvement-kata" and "getting there." What it looks like day-to-day once you've arrived is something you almost never hear about.

I write this in the still-warm afterglow of LISA15, where I gave a talk about (surprise) metrics and monitoring. In that talk, I had two big bones to pick. The first was to attempt to fill that gap, basically to show off a bit of what the DevOps endgame looks like for operations folks like ours, who still work to solve real problems day to day. The second was to make the point that DevOps is mostly still getting "monitoring" wrong, because rather than working monitoring in to the the rest of the improvement processes they practice, DevOps seems intent on treating monitoring as a "heroic" discipline. Creating an ever-increasing litany of new, specialized monitoring categories, which in turn silos the resulting telemetry data in ways that limit its potential to the rest of the organization.

During the talk, I shared several chat transcripts with diagrams, like the one in Figure 1. Each of these represented a real problem in several wildly varying contexts that our engineering teams had encountered in the weeks leading up to LISA. My thinking at the time was that presenting a breadth of different problems instead of a depth of one would better illustrate the point that, since our monitoring data was *not* being siloed, it was therefore more useful than this sort of data is at other shops. It was frustrating to me, however, that I couldn't dive as deeply into the actual problems as I would have liked to in the time I had on stage.

So in this issue, I'd like to choose just one of these and dive a little more deeply into it, so you can really get a solid feel for how our engineers are using the telemetry system in the context of detecting and tracking imbalances in the system that will eventually lead to catastrophic outages if left unchecked. It's a pervasive belief today that metrics are not yet useful for very



benjo 11:29 AM
fyi mike we're nearing 12 hours with this last round of web-node cpu pegging / source-matching rodeo

benjo 11:30 AM
Uploaded an image: web-node cpu against match_display_names.time

librato BOT 11:31 AM
Alert metrics_web_prod.cpu.usage has triggered! Runbook
metrics-web-prod-808
metric `AWS.EC2.CPUUtilization` was above threshold 95 over 300 seconds with value 100 recorded at Wed, Nov 11 2015 at 19:31:50 UTC

mike 11:36 AM
Benjo: welp, probably still [redacted]

benjo 11:40 AM
yeah I'm working with jdirty to see if we can possibly attack from the space angle and optimize

Figure 1: Some internal chat where Ben notices a CPU spike, posts those graphs to the chat, and involves engineers from other areas

early problem detection of this sort—that advanced aberrant detection algorithms need to be developed to help in this regard. I'm going to use the problem depicted in Figure 1, which is also the first problem I presented in my LISA talk.

Let's talk for a second about how these graphs even exist, much less exist in the same system. The first graph is the age-old OPS CPU graph. We get ours by way of collectd, and I'm sure you have them too. The second graph was put in place by the data engineering team because they needed to quantify the amount of time we lose looking up metric and source names in our various index DBs. I couldn't cover in the time I had at LISA what that means, but here I can spare the space, so let's digress into that for just a few paragraphs.

One big problem with designing time-series databases is that the pattern of reads and writes is very different. If you think about writing time-series measurements into a database, you'll see that we're writing into columns. That is, if you think about it like a spreadsheet, you have a column for every second of lapsed time, and then a row for each metric name. We can only write one column at a time, because the future hasn't happened yet, but we can write to multiple rows in that column at once because we have many different metrics to track, and all of those numbers are coming in at once.

But when you read from a time-series data store, you read *rows*. That is to say, you're never interested in a search that gives you a single data point for *every* metric you're tracking at a single point in time. You always want 60 or 90 seconds' worth of data for one metric. So you have to read out rows.

This is usually okay, because in TSDB land, you write a lot more often than you read, and columnar writes are pretty efficient. But reads are exquisitely painful. Think about it—as time progresses, the rows keep getting longer and longer. Soon you need to search through increasingly gigantic rows in order to isolate the set of data you need and then extract it. So you wind up spinning cycles in linearly increasing seek time to find and retrieve your data as the rows grow. Not good.

One way we get around the long-row read problem is with a rotating row-key. Imagine that, instead of the row being the name of the metric, it was a number that changes every six hours. That way, we create a new row for each metric every six hours, and we store it as a name/number tuple and our seek time never goes above a certain predictable value.

"Predictable" is something of a keyword there, because really what row keys buy you is a heap of different kinds of very important predictability. With row keys, we know, for example, that our rows will also always be a predictable size (in bytes), which is the size of a measurement times the number of measurements in a row-key interval. From there we can extrapolate how much data we'll need to put on the wire for client reads, and how much storage and processing power our data tier will require. The point is, we can make really important decisions by relying on the predictability that row keys provide.

But here's the problem (well, *one* problem) with running a shared storage back-end for a multi-tenant system: users have the power to name things. So if you think back to how we're now storing our data as name/measurement tuples, we no longer have a predictably sized data structure because end-users can create *really* big names if they want to.

So when you use a row key to buy predictable computational quantities in the data tier, you also often have to pay some taxes in the form of lookup-tables, or indexes if you prefer. In this example, we're going to need two indexes, one to keep track of where in our storage tier a given six-hour block of measurements is stored (because our rows are named after rotating numbers now, so we need some way to actually find the right data when a user asks for 60-minutes of metric:foo), and another to map user-generated variable-length names into either hashes or some other unique identifier (so we know what to even search for in the first index when a user asks for 60 minutes of metric:foo).

Okay, now we know pretty much everything we need to reexamine the first problem I shared in my LISA presentation on metrics. Ben, the Ops guy in that conversation, is tracking what he considers to be a resource allocation problem. The symptom Ben is reacting to here is high CPU utilization, which is actually something of a rarity for us, but it's also why I included it here (everybody can relate to a good-ole CPU utilization graph). Our problems are often CPU bound—that is, a problem in something we've built will often result in high CPU utilization, but it's rare for us to discover it by way of a CPU graph.

The second graph is the one we can now fully understand given our short discussion about time-series data stores above: briefly, the second graph is timing how long it takes us to perform an index lookup on a metric. You'll recall that I said rotating row keys come at the cost of index tables. Well, you can think of the second graph here as quantifying the cost of our row-key taxes. Literally, when we want to retrieve 60 seconds of metric:foo, how long does it take us to convert "metric:foo" into "ID12345" so we can use the ID to retrieve the data from the data storage tier?

If you're asking yourself why we don't cache this stuff, the answer is we do. So this problem—the one Ben and Mike are discussing in chat—is extraordinary for that reason alone. It never happens except in the event of a cache-miss. And for it to be this bad, and progress for this length of time, an awful lot of recurrent individual cache misses need to happen, and that strongly implies that we're encountering a new end-user behavior here. Either these services don't work like we think they do (not the case here), or an end-user is doing something to generate an inordinate amount of cache-missing index lookups. Problems like this one are, as you can probably imagine, very interesting to us. They inevitably teach us something we didn't already know about our systems, by showing us either a gap in our understanding or a path through our system that we didn't anticipate.

I think it's fair to say that many engineers believe that being good at Web-operations engineering, or really any kind of high-availability engineering, means building solid infrastructure and reacting quickly and effectively to blocking-outages as they occur. But in my opinion, being really excellent at operations engineering is 99% about being interested in problems like this one: problems that are non-blocking, that are not currently causing anything close to a catastrophic outage. Annoyances really, but exactly the right sort of annoyance. The annoying little imbalances in the system that teach you something you didn't know, or hadn't anticipated about the thing your organization created. Problems that you, by definition, can't actually fix by yourself.

You might be tempted to call it preventive maintenance, but the big difference between the problem we're looking at here and preventative maintenance, in my opinion, is the fact that Ben can't solve this problem alone. In fact, I think probably the most important aspect of this issue—certainly the thing that made me want to share it with the world—is that Ben the operations hero can't solve it by himself. These issues are *so* important, they are the bubbling spring from which catastrophic torrents are born (especially in distributed systems). Like Muhammad Ali said, it's always the punch you didn't see coming that knocks you out, and this one certainly would have eventually been a knockout punch if Ben hadn't seen it coming. The problem is, you need not just interdisciplinary know-how, and a toolchain to work problems like this one, but a culture that encourages cooperation over heroism.

First, Ben needs to know enough about the system and the monitoring data it generates to even discover that this problem is there, that it's worth working on, and what's causing it. Then he has to track it—quantify its occurrence long-term over weeks and months. He needs to understand the nuances of the system from which it has emerged well enough to even know whom to work together with. And, finally, he needs the interpersonal skills to get other engineers involved, as he's done here, as well as the toolchain to allow him to easily share the data he's looking at.

In the snippet, he's giving Mike, a data engineering guy, a heads up about it. We can tell they've spoken about it before, and the eventual fix will certainly be a data-engineering endeavor. At the very bottom, you'll see that Ben is also working with Jason on some UI tweaks (our UI is called "spaces").

So if you're a system administrator, or otherwise consider yourself operations or SRE, try to put yourself in Ben's shoes. You're looking at a CPU spike that correlates to the amount of time it takes us to run an index lookup on a metric name. I think many of us wouldn't have made it that far. Past versions of myself certainly wouldn't have made it that far, because I wouldn't have even had access to the data that would have allowed me to discover that correlation. I certainly wouldn't have had the domain knowledge about our product to know what that correlation meant.

Anyway, rather than opening a ticket and throwing it over the fence, Ben personally gets data engineering involved, helps them help him further understand the problem, and then proceeds to update them periodically as he sees it reoccur. Together they establish a pattern of behavior and narrow it to a handful of customers (that's the little black box that was redacted from the chat transcript).

I personally have never made it that far. I've never been able to create a strong enough rapport with engineers outside my own discipline to be able to work together to understand an issue like this one. I think my own super-hero-thinking is a huge contributing factor in this unfortunate truth. Any problem I couldn't solve by myself with open source software just wasn't worth my time and should be thrown over this or that fence. Made someone else's problem. And every time I pulled that rip cord, I gave up any chance I had of learning about how the things my organization cared about actually worked.

But Ben never stops. In fact, looking at the transcript, he takes it one step further even than *that*. Well, he reasons, if customers use our UI to retrieve data from the data tier, then really it's the *UI* that's triggering all the index-churning. Therefore, it's possible that some light UI optimizations might help alleviate part of this index lookup churn. Maybe index lookups would be faster if we batched them, or maybe we can pre-fetch them based on user behavior? I don't know if those particular optimizations are what Ben had in mind when he roped in the UI team, but as an engineer they occur to me as distinct possibilities.

And therein lies the thing about this conversation that *really* fascinates me. I've been calling this stuff "domain knowledge" and saying that it's "interdisciplinary," but really none of these notions are *so* abstract or complex that I can't understand them with a few minutes of consideration. It only took me a few hundred words to bring you fully up to speed with respect to index timing and what that means in our TSDB, and of course we all know, or can infer, that a user-interface might be able to batch-query things to avoid a multitude of individual requests, so why, in the 20 years I've been doing operations work, have I never stepped over that line to work together with other departments toward a common solution to problems like this one?

One answer is certainly that my monitoring data has always been trapped in silos. I didn't have a means of sharing engineering data with other teams. Just being able to see what the data engineering team has chosen to measure inside a newly constructed service teaches me as an operations engineer an awful lot about that service. It also provides an opportunity for me to formulate questions that I can ask those engineers when I see them at the coffee pot or the bar after-hours. Those conversations build rapport, and rapport is exactly what you're seeing there between Ben and Mike.

Monitoring data shouldn't belong to anyone, and it certainly shouldn't be a magical contraption reserved for a select few heroes who have bothered to understand how it works. I really hope this little play-by-play helps illustrate what I mean by that, and also what I mean when I say we don't need another hero, we just need (as always) to let the data be free.

Resource

[1] Gene Kim, Kevin Behr, and George Spafford, *The Phoenix Project*: http://itrevolution.com/books/phoenix-project-devops-book/.