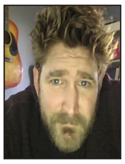
iVoyeur
How Do I Even KPI?

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

As I write this I'm on a plane back from "DevOps Days Toronto," at which I had a marvelous time. Probably the highlight of the trip for me was the "Open Space" on choosing effective KPIs (Key Performance Indicators). If you haven't been at a conference that does Open Spaces, they're very much like BoFs, except that they happen during the conference (not at lunch or after hours), and the selection process is more formal.

Honestly, I used to think they were kind of silly and suspected they were merely a means of making up for a lack of presenter content, but having spent the last year and a half traveling a lot more to various conferences, I've increasingly come to value them. The format really manages to give you a good feel for what everyone is dealing with in a specific problem domain (especially if you can manage to attend a few of them in different parts of the country).

The Open Space on the topic of choosing KPIs began with a question from the developer-turned-architect who had initially proposed the KPI Open Space. He'd just been put in charge of figuring out how to stabilize the efforts of 68 different development teams (!), and by stabilize, he meant that their product was behaving erratically, and they were beginning to have large blocking outages.

It sounded like his teams were all working on different parts of a single, large microservices architecture, which had grown large enough that the individual development efforts for each service were growing apart and becoming siloed. Because he was known to be a talented engineer who'd contributed to many of the services individually, the business had decided to "DevOps" him—i.e., snap him off from his current team so that he could focus on making the entire system work together better. He was eager to help but was having a hard time figuring out how to begin. He knew he wanted to get some data that would give him a good feel for where the problems were, but his question was, *what* specifically he should measure: "How do I choose some KPIs from scratch?"

It is a (usually) unwritten rule in programmer forums not to ask the room to do your homework for you. I'm not sure whether this applies to Open Spaces, but the architect's question certainly flirts with that line. In an Open-Space setting, however, I actually prefer this kind of discussion to the shallower and more uninformative "what is everyone using for X?" sort of question that typifies the Open-Space experience. In fact I think it's fair to say that when someone commits an oversharing faux pas in an environment like this, it relaxes everyone else, and puts us all in the mood to overshare a little bit ourselves.

Anyway, it quickly became apparent that many people in the room were having exactly the same pragmatic problem of not knowing where to begin with choosing metrics to measure. The first suggestion he got was to implement a policy that mandated filling out a form that included information like what KPIs should be measured before every deploy to production. This suggestion was accompanied by a lengthy, and very opinionated, anecdote that at some point segued into a full-bore anti-continuous delivery rant.

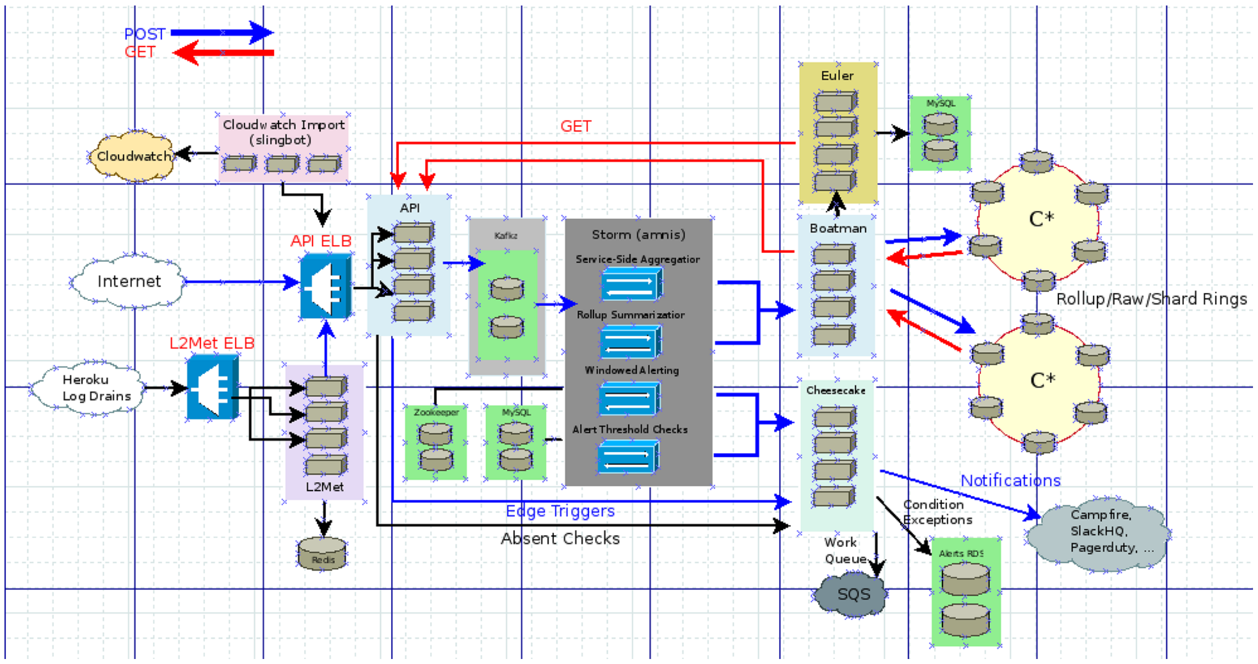


Figure 1: The prototypical (I hope) architecture diagram

"Best Open Space ever," I thought to myself as the room launched itself into a 40-minute long sanctimonious DevOps shame-splaining party. In the end, though, we were nowhere nearer to helping out with the original question (although we had a lively and entertaining discussion about the nature of DevOps versus "what the business actually needs").

Believe it or not, I do make an effort to keep my big mouth shut during the Open Spaces I attend (I rarely succeed). In this case, however, since no one else had offered any constructive advice, I ventured to share what has worked for me in the past. And since it was well received, and the problem seemed so prevalent, I figured it might make a nice *;login:* article this month, so I'll share it with you too.

I'm sure I've said before in this column that good metrics test systems hypotheses. They capture the operational limitations we've learned about the things we build. When I say they test systems hypotheses, I mean that when we think about the systems we build, and how they should act in certain situations (e.g., given 50,000 connections, this round-robin-based load balancer should send 25k to server A, and 25k to server B), good metrics confirm our valid assumptions and discredit our biases. They teach us about how the things we build actually work.

By this yardstick the classic CPU/memory/network triumvirate is mediocre at best. You *may* have a meaningful hypothesis about how much RAM or CPU a process should use, and you *may* learn something about your system (or more likely the underlying interpreter or OS, or garbage collector) if your assumption isn't borne out in practice, but metrics that measure things like how long a particular database call takes, or count the total number of worker threads, or queue elements, reflect assumptions that make for a more meaningful understanding of the system you're dealing with.

Not only do experienced engineers understand that building a system is not the same thing as understanding it, they can pretty quickly intuit how well a system they didn't build is understood by the team running it. The evidence is everywhere: in how deeply we can test our code, in how specifically we monitor them, in how precisely we can derive our capacity plans, and even in how repeatably we can deploy them.

The architect who asked this question was an experienced engineer. He knew that these teams didn't understand what they'd constructed, and therefore no amount of asking them to fill out a form listing their KPIs was going to give him the insight he needed to make it work better. He had to get his own hooks in, but the question was where?

Whenever I'm put in charge of a large and churning wad of software that I didn't write, I draw a picture of it, and that picture inevitably comes out looking something like Figure 1. In fact, this is one of the actual pictures I drew when I was first hired on and trying to wrap my head around how Librato's microservices architecture works in practice.

Measure the Space between the Services

Normally, we'd focus our attention on the boxes, and in the end we do want to know, in depth, how each of these services works so we can derive some metrics that are key indicators of how well

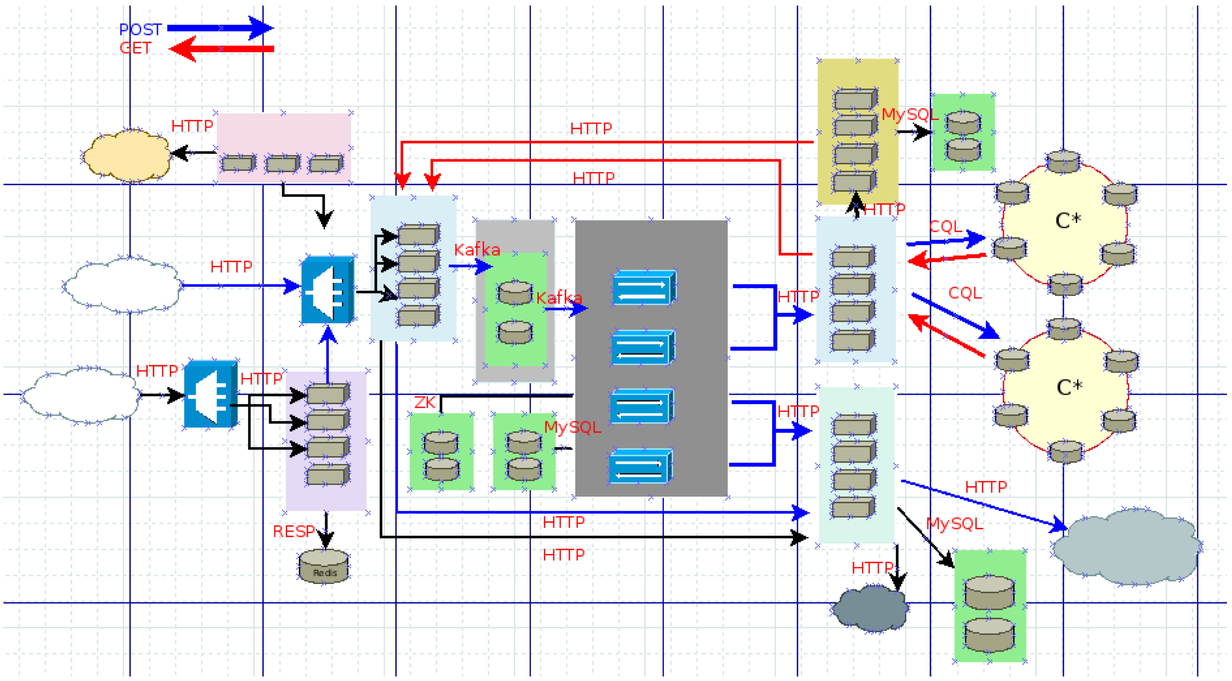


Figure 2: Figure 1, relabeled to accentuate the space between the services

they're doing what they should be doing. However, we're going to start by ignoring the boxes completely. In fact, I'm going to delete all of these box labels and replace them with letters, and in the place of all the service names, I'm going to label the lines. Above each line, I'm going to place a label that identifies the protocol each of those lines represents. This gives us Figure 2.

Check that out, our previously incomprehensible microservices architecture just became a handful of commodity network protocols. This, I can pretty easily wrap my head around. Every application is a balanced equation; it'll work fine as long as it is in balance, and eventually we'll root out all of the things that can throw it out of whack. But for now, the best way to detect when it's out of balance is by timing the interactions between its component parts—measuring the space between the services. Our strategy will be to figure out a way to time the interactions represented by each of these lines.

If I made that sound easy, it's not. Getting these numbers, which I collectively refer to as inter-service latency data, is going to require a lot of engineering know-how. In almost every case, you'll have to get into the source and add some instrumentation that wraps API or DB calls. Sometimes you'll be need to reconfigure a set of Web servers or proxies, and every once in a while, you'll need to write some glue-code or API-wrappers of your own.

You should wind up with a slew of numbers on the order of tens or hundreds of milliseconds. When something goes wrong with the application, these numbers will tell you *where* the problem is (in which service on which nodes). Note, this is not the same thing as telling you *what* the problem actually is, but we'll get to that in a minute.

Of course you'll need to actually put all of this data somewhere. That's the sort of thing I (and many other people) have written about at length, but it's worth mentioning here that you're going to need a scalable telemetry system to help you store and analyze all this stuff.

Extract Knowledge from Inter-Service Latency

Play around with these numbers as you get each of them up and running. Note the baseline values, and search for patterns of behavior, and things that strike you as odd. Do some service latencies rise and fall together? Do some appear dependent on others? Do they vary with the time of day or day of week? As you discover these patterns, talk to the engineers who run the services and see whether these patterns confirm their notions of how that service "should" work. It shouldn't take long before one of them squints at your data and says something like "huh." This is what scientific discovery sounds like. Dig into that service behavior with the help of the engineer who runs it, and you'll likely encounter a KPI or two.

When something goes wrong, look at the inter-service latency data and see how early you can identify things going sideways. The numbers tend to get big upstream of the services that are actually having trouble. Share your data with the engineers running those services, and dig into them together to figure out what went wrong; again, you'll likely encounter a KPI or two.

If that sounds kind of labor intensive and slow, it is. But before you know it you'll have several dozen extremely valuable KPIs. Until you get into the habit of choosing effective metrics, they take some time and effort to identify. Each KPI really is a manifestation of insight; each teaches you something you didn't know about the services you maintain. Each is a thing to be prized, shared, and talked about.

For Example

As you can probably imagine, we're pretty good at choosing effective metrics before we need them at Librato, but we still regularly encounter valuable metrics that we didn't anticipate. For example, we recently encountered a behavior in one of our newish services that we couldn't explain. Symptomatically, it was quite visible in our inter-service latency data as a latency spike between the service and a MySQL server.

When we dug into it, we found that there was a bug in the upstream API of a vendor that the service relied on. If we crafted the API request a certain way (the correct way), the API returned too many results (all of them, instead of the subset specified by the query), and we wound up over-taxing our own MySQL server writing this over-abundance of results back. But if we used a modified version of the broken-looking example from the upstream vendor's documentation, it worked fine.

We reported the bug and commented our code, but found that every engineer who came across this query had the irrepressible urge to fix this broken-looking API query, so we began tracking the number of results returned by this API query as a KPI for that service. Several months later, when the upstream vendor fixed their API, we had the opposite problem: we were getting 0 results back from that API (because our broken query, was in fact, now broken), but since we were already tracking that metric, we immediately saw what the problem was and were able to very rapidly push a fix for it.

Today, the engineers who were involved in that episode (myself included) tend to include KPIs like the number of results returned from interfaces they don't control as a matter of course. They probably don't even remember why. This is one of the many ways that going through the process of finding and relying on effective operational metrics changes the culture of engineering teams. It is a self-sustaining cycle: good data begets reliance on data, which begets better data.

KPIs that represent insight into the systems that we build give us a rock to stand on in the midst of uncertainty, and enable us to act quickly and decisively to protect the uptime of our services. Without them we don't really know how the things we build work. If you're in that boat, the place to start (IMO) is with inter-service latency data. Get it, and use it to work your way into insight.

Take it easy.