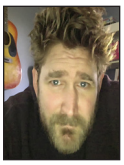
iVoyeur
Tests and Metrics

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

I realized this morning that I haven't been to a change management meeting in years. I imagine that many people who make software for a living haven't been to one in a decade or more. Continuous integration (CI) and automated testing killed change management for the modern software engineering shop, but those of us who fix things were left only with configuration management, which was not quite enough to keep us out of change meetings.

As I write this in the brief lull between AutomaCon, which finished last week, and LISA15, which will arrive before I'm ready for it, I can't help but muse a little bit on "infrastructure as code" (IaC)—the somewhat ungainly offspring of configuration management and continuous integration that has killed change management for me and increasing numbers of other operations folks.

Setting aside for a moment the somewhat utopian notion of abstracting away all of our ugly pipes and wires into software, IaC is a good thing because it gives most of IT a common interface to make changes—namely, the deployment pipeline. Software engineers make changes to files that represent applications, and commit them to Git, which calls out to a CI tool to run some tests on it, and if they pass, a process or person deploys it to production. Now operations folks do pretty much the same thing, making changes to files that represent servers or routers or whatever, and committing them to Git and etc.

All of this rests on the foundation of continuous deployment, and continuous deployment rests on the foundation of tests. But testing infrastructure as code is a pretty new endeavor; it's just not something your typical sysadmin or operations person has much experience with. We're starting to see a few tools pop up, most notably Serverspec [1], but we're still going to need to become skilled in choosing and crafting good tests.

One thing I've noticed in my ongoing developer anthropology is that a lot of software engineers had and continue to have the same problem with choosing monitoring metrics. It's just not something the typical software engineer has a lot of experience with (which is tragic, but that's beside the point). And from that observation follows another: it turns out that choosing good tests and choosing good metrics are similar endeavors, and in this article, I'd like to explore some of those parallels with you.

Our Deployment Pipeline

Librato is a prolific engineering shop. We range between 40 and 60 deployments per day. In fact, as I write this, so far today we've deployed code 40 times—12 of which were production changes (the others targeted for various staging environments). I can see all of these deployments in our corporate chatroom, because we use chatbots to push code into production. In fact, most of our interaction with the services we maintain is abstracted behind chatbots in one way or another. So when someone merges some code into a production repo, I can see it in group-chat:

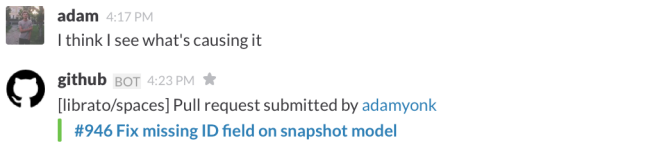


Figure 1: Chat transcript of a GitHub PR

And not only can I see the pull request (PR), I can see whether the proposed change passes or fails its unit tests:

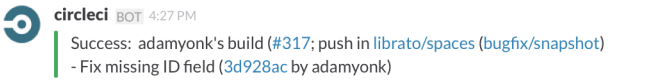


Figure 2: Chat transcript of a passed unit test

And then I can watch with bated breath as the developer then deploys the change into production.

Figure 3: Chat transcript of a deployment

How Often Do Good Testers Test?

I'll pick on our alerting service because it's written in Go and, since it uses Go's built-in testing framework [2], is easy for a knuckle-dragger like myself to inspect with grep. Let's see how many test files there are:

```
find . | grep _test.go | wc -l
```

That returns 44 individual, test-laden files. Roughly one for every other .go-suffixed file in this repository, each one named for the unit it tests. Ergo, for foo.go, we find foo_test.go about half the time. Lightly poking into the files that don't have an associated test file, I find mostly type definitions and other data-structure-related code (not the sort of thing you normally test directly).

How about actual test functions?

```
grep -ri 'func Test.*(*testing.T' . | wc -l
```

This yields 172 individual tests. About a 4-1 ratio of total functions to test functions. So about 25% of the functions we create are tests.

What about by sheer volume of code?

```
find . | grep _test.go | while read i; do egrep -v '({|})' ${i} |
grep '[a-z][A-Z]'; done  | wc -l
```

Gives me close to 2400 lines of code devoted to tests. In fact, test-related code makes up almost half of this repository measured by lines. So OK, we test a lot, but then all of us who work in continuous integration shops do nowadays.

Change-control meetings are intended to protect healthy production environments from human error by instituting a layer of peer review. Whether this works or not is debatable, but it is unquestionably slow and drains productivity. Long release cycles allow more time for development and production branches to diverge. The classical change-control methodology, therefore, by slowing down the release cycle, tends to foster larger, more substantial (and therefore more error-prone) changes.

Relying on unit tests to protect us from human error instead allows us to make smaller, simpler, safer changes more often. We can spend as much time creating tests as we might otherwise spend on halting productivity to create change proposals and argue about them in a weekly meeting.

What Makes Good Tests Good?

The operative word there is RELY. Our tests can't protect the production environment if they aren't meaningful. In creating them, we generally need to be both procedural and selective. We need to select test criteria that we can genuinely rely on to help us ship quickly and safely.

GOOD TESTS ADD CONTEXT AND ENCOURAGE COOPERATION

If we make our tests too difficult, obtrusive, or meaningless, or if we try to enforce things like coding style that everyone hasn't already agreed to, people will just work around them. Self-defeating behavior like this is more likely to emerge when we sequester test creation to a particular team. Tests should mostly enforce the expected operational parameters of the things we create. Everyone should craft them, because they help us all reason about what we expect from the things we build. Tests that we didn't write should give us insight into new code-bases rather than encourage adversarial relationships between engineers.

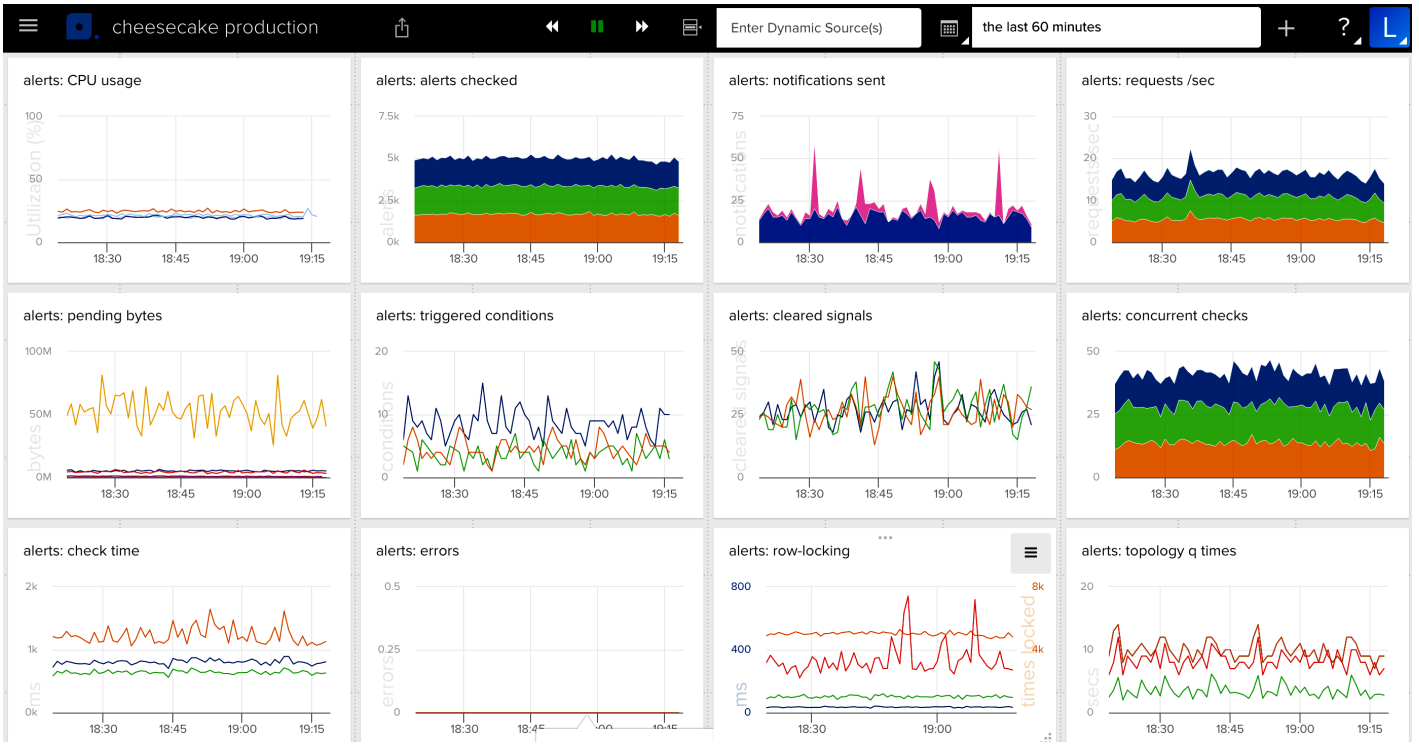


Figure 4: The alerting service dashboard

GOOD TESTS IMPROVE OUR DESIGN

Creating and maintaining good tests requires that we reason about correctness when we design and create software, thereby making us cognizant of our own expectations and assumptions. Choosing good test parameters means thoroughly understanding not only what we've created, but also the difference between what we've created and what we set out to create in the first place. Testable code is well-designed code, and poorly designed code is usually hard to test.

Metrics Are Tests that Never Stop Running

There's another class of code in this alerting repository that's neither functional to the application nor related to unit tests. An example looks something like this:

```
metrics.Measure("outlet.poll.alerts.count", len(alerts))
```

This is instrumentation code, and grep counts a little over 200 lines of it in this repository. The idea behind instrumentation is to measure important aspects of the application from within. Instrumentation like this quantifies things like queue sizes, worker-thread counts, inter-service latency, and request types. These metrics are then exported to a centralized system that helps us visualize the inner workings of our applications. In fact, here's a screenshot of the dashboard (Figure 4) where the metrics from this alerting service wind up.

Unit testing is like the sign at the theme park that says we need to be this tall before we can deploy to production. Our metrics are more like the canary in the coal mine. They are tests that can follow our code into production. They help us continuously vet our assumptions about the changes we introduce. Like test-driven development, which uses carefully crafted unit tests to verify correctness, metrics-driven development uses well-chosen metrics to directly show us the effect of our changes.

What Makes Good Metrics Good?

At this point I could en masse copy/paste the section I just wrote about what makes good tests good, substituting the word metric for test. Our metrics are the primary means by which we understand the behavior of our applications in the wild, and so we need to rely on them arguably even more than on our tests.

Like our tests, our metrics also help us protect the production environment from human error. If they aren't meaningful, our continuous integration pipeline suffers.

GOOD TESTS ADD CONTEXT AND ENCOURAGE COOPERATION

Good metrics test systems hypotheses. They confirm our expectations about how the things we build perform in real life. Just like tests, everyone should be able to choose and work with their own metrics because they help us all reason about what we

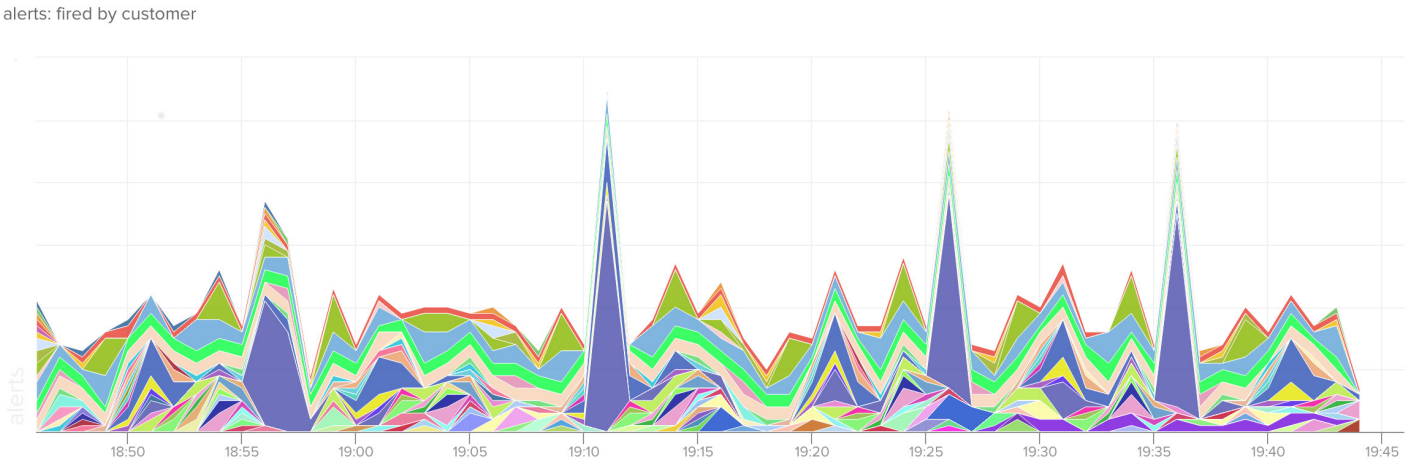


Figure 5: A well-chosen metric

expect from the things we build. Metrics can teach us about code bases that we aren't familiar with. Without any documentation whatsoever, I can infer many things from the metric in Figure 5 (e.g., this service sends alerts, the number of customers using it, the total and individual rates at which alerts are fired etc.).

GOOD METRICS MAKE GOOD CODEBASES

Choosing meaningful metrics also requires us to reason about correctness when we design and create software, but when we succeed, we gain ongoing operational insight that's invaluable to everyone, whether they're designing systems, regression testing, supporting infrastructure, or shipping features.

Good instrumentation is a sign of operational health. It keeps us cognizant of our own expectations and assumptions. Well-measured code is usually well-designed code, and poorly designed code is usually difficult to measure.

So even if you've never created a unit test and find the notion of Serverspec and IaC daunting, you can take comfort in the reality that being a diligent student of systems monitoring and reading excellently crafted columns like this one has prepared you for what is to come. No need to thank me, I'm already positively drowning in acclaim.

Take it easy.

Resources

[1] Serverspec: http://serverspec.org/.

[2] Testing in Go: https://golang.org/pkg/testing.