Editor: **Robert Blumen**
SalesForce
robert@robertblumen.com

Tammy Bütow on Chaos Engineering

Edaena Salinas

From the Editor

In Episode 325 of Software Engineering Radio, host Edaena Salinas talks with guest Tammy Bütow of Gremlin, a provider of failure injection as a service, about chaos engineering. Previously, Bütow worked at Dropbox on cyber liability engineering and at DigitalOcean on cloud infrastructure as a service. She also cofounded Girl Geek Academy. The excerpt here covers Tammy's background in failure testing, the nature of failure injection, what we can learn from chaos experiments, how to work through failure scenarios to harden your infrastructure, and their value in training operations staff. Portions not included here go into more depth about the hands-on training Tammy conducts, the use of monitoring tools, and how chaos tooling is evolving to look more like other areas of software development. To hear the full interview, visit http://www.se-radio.net or access our archives via RSS at http://feeds.feedburner.com/se-radio.

Edaena Salinas: Let's look at motivation, because some of the worst outages lead to the introduction of chaos engineering. You worked at Dropbox. Can you talk about its worst outage?

Tammy Bütow: The worst outage at Dropbox is publicly available on the blog [https://blogs.dropbox.com/tech/2014/01/outage-post-mortem]. It was in 2014 and went for multiple days. It was related to the databases. The vice president of engineering wrote a full review of what had happened, and explained the action items to make sure that it didn't happen again.

One of those action items was injecting failure to ensure that everything was more reliable. The idea of injecting failure is similar to a flu shot, where you inject a bit of harm, but it makes you stronger. You are able to withstand it because you have injected it frequently. A lot of people learn it the hard way.

The purpose of injecting failure is to learn about what could go wrong, correct?

Yeah. Infrastructure, whether on the premises or in the cloud, is going to fail in some way: hardware failure, power failure, firmware failure, kernel issues, or issues with your own tooling. There are many things that can go wrong.

We wish that cloud infrastructure was 100 percent available and worked perfectly, but that's not true. We need to build our infrastructure knowing that things will break. It's much better if you control the injection of failure, observe it, monitor it, learn from it, and use that to make your infrastructure more reliable.

You learn so much from doing chaos-engineering experiments. Do something simple, like shut down a server. That tests your automated self-healing. Regularly shut down a database replica—to test your clone process, instead of only testing when a replica [fails on its own]. Maybe that happens once every few months, but if you test it once a week or even more frequently, that's so much better.

How you go about it depends on how big your infrastructure is. How

0740-7459/18/$33.00 © 2018 IEEE

frequently you're going to be injecting failure and what kind of failure you inject depend on whether you have tens of thousands of servers or hundreds of servers. But everybody can learn from failure injection.

How does testing differ from chaos engineering?

I started to work at a bank after university. In a bank you have to do disaster recovery tests once a quarter to hold your banking license. It's a compliance requirement. If you can't prove that you can fail over your entire bank, then you will have problems with the regulators, and you will be shut down.

That was where I was introduced to the idea of live-scale failover. You fail over everything. You must test that your entire team can operate, even if your office wasn't available and it was over a weekend. You go to a different building for two days and fail over everything to make sure that, if something went wrong at your headquarters, you could keep the bank running.

That was my introduction. Later, when I was working as an engineer, whenever I built something it was always too slow. I [realized that I] couldn't rely on the infrastructure underneath it for it to work well. That sent me further and further backward.

I spent a lot of time working with the database, and even further back, to hardware, networking, security. It is very helpful to have a good understanding from the very top level—JavaScript—all the way back down to the power within the datacenter or your cloud infrastructure. Going through every single level, that's where you can think about failure injection across the whole stack.

What's the role of size in deciding whether to do chaos engineering?

Often you have a lot of machines, but you don't have that many people. That's why you need to do a lot of automation work, and you need to really understand the failure modes.

There are engineers responsible for maintaining the fleet, reliability, availability, and durability. But not all of them have been there for many years. Some of them have only been there a few months, some for one year, and a few longer than that. And somebody who had been there for a few years may have just left with a lot of knowledge.

With large-scale infrastructure, there's a lot of knowledge that needs to be transferred throughout the team. And with large scale, you usually don't have many more engineers than you do for small-scale infrastructure, but you have a lot more that you need to do.

You can use failure injection as on-call training. Run a game day having 10 to 15 engineers in a room together, talking through failure injection scenarios. You can whiteboard it. Say, "If we injected failure at this point, then we would expect this and this to happen," and then, "We expect this and this to be the downstream impacts, and the cascading failure could look like this." And then you could say, "We feel pretty confident that we could inject this failure and everything would be all right."

Your hypothesis is, "We can shut down this replica for this database, and the clone should kick off. Everything should be fine, and within this period of time I would be back to having a primary with two replicas." That's a scenario. Within the game day you would inject that, and then see how it goes.

There will be other scenarios where you say, "We're not yet ready to run through that because we need to fix other parts of our infrastructure to make it more reliable so that it can withstand that failure." That's what you're going to learn when you do these exercises.

You can also do chaos engineering at small scale, right?

Yeah. It's super-important to do it at small scale. One of the common mistakes running small-scale infrastructure is not having enough machines, putting everything on three or four servers, not spreading things out, not having redundancy, and not having backups. That's likely where you get into trouble. Because you don't yet have a ton of users, you don't need as many servers as you would to handle a large amount of traffic. But if you don't have backup and redundancy in place, and you haven't thought about how to fail over your services, then it's going to be impossible to do so [when you need to].

Small-scale infrastructures that haven't invested in building out those things can have much longer downtime and more risk of data loss because they haven't thought through the backup scenarios for their database infrastructure. Maybe they have not tested the restore process to see if they can restore their backups. They often do not have very good incident management in place to know that that "we're currently experiencing an issue; we need to do [steps] to fix it."

A lot of these things do not take a ton of time to set up, but require you to think about it in a different way. It often comes down to funding, because you must pay extra to purchase these instances.

But that's the exciting thing about cloud infrastructure now: it's becoming more and more affordable, and you're able to purchase smaller instances. You can spread out your services across a number of instances. You understand

that failure will happen, but you are building your infrastructure to handle that. It used to be much harder when you had to buy bare metal. You could only afford at most three servers, and then when they would fail you had to get them repaired. Now it's totally different with cloud infrastructure.

To what extent is this a lack-of-knowledge issue, and to what extent is it a cost issue?

It's not a significant cost. Servers are getting cheaper all the time. Look at the other side of the problem: if you pay this much for your infrastructure, what would happen if you were down for three days? That is pretty common. Even weeklong outages happen. These happen to some of the biggest companies and small companies as well.

What happens to your business if you're down for three days? Your customers are going to be unhappy, and you might lose them. If you invest in reliable infrastructure it might cost a bit more for your monthly bill, but you are going to have a more reliable service. Your customers will be happier because they're not going to experience downtime or data loss.

The nature of the product and project matters. If it's a photo app, maybe it's not that critical, and you're not going to lose customers; they're just going to be annoyed. But if you're dealing with a hospital or banking system, it's more critical. Is that right?

Even if it was a photo app, if you are down for a day or two, and somebody trusted their photos in there and cannot access their photos, then they might lose trust in your business. They may start to think, "Do I need to have a backup service to hold another set of my photos? If so, then why am I paying for this service?" They might worry that if it was down one day, does that mean that it's not reliable and that in the future it may lose data?

These are things people think about when they're looking at whether to use a service. It's so easy for people to create new businesses these days. Building more reliable services can help you acquire new customers and make money, because you're showing people they can trust your service.

Companies now have available infrastructure such as Amazon Web Services (AWS). Are companies more equipped to recover from failure if they leverage those systems?

Yeah, exactly. There was a big AWS S3 outage last year (that I was on call for). One region of S3 was impacted. There are several regions that you can use, or you can have a backup mechanism in place so that if that region goes down, then you have some type of failover ready. It's important that we think this way these days.

How do you teach chaos engineering?

This year I taught a chaos-engineering boot camp. What we do is to create a cluster for everyone, because often people say, "The only way to learn chaos engineering is on production." You can actually start chaos engineering first on a demo environment. Then you can also do it on staging. And then once you feel confident, you can do it on production.

But what I do for the chaos-engineering boot camp is create a demo environment. I spin up a Kubernetes cluster for everybody. I'll have three people working on a cluster together, and then I give them access to my GitHub repository Reaper, which has

SOFTWARE ENGINEERING RADIO

Visit http://www.se-radio.net to listen to these and other insightful hour-long podcasts.

RECENT EPISODES

- 330—Kevin Goldsmith and host Travis Kimmel converse about Conway's law.
- 328—Bruce Momjian talks with host Robert Blumen about SQL query planning and optimization in Postgres.
- 326—Dmitry Jemerov and Svetlana Isakova discuss the Kotlin programming language with host Matthew Farwell.

UPCOMING EPISODES

- Natalie Silvanovich shares her views on attack surface reduction with host Kim Carter.
- John Doran tells host Jeremy Jung how a salon-booking service provider fixed its broken development process.
- Saša Jurić goes in depth into the Elixir programming language with host Nate Black.

ABOUT THE AUTHOR

EDAENA SALINAS is a software engineer at Microsoft. Contact her at edaena@gmail.com.

chaos-engineering experiments. And I deploy a demo microservices app from Weaveworks, which is an e-commerce store. When you come into the workshop, you've got this demo environment, you can see the e-commerce site running, and then you can start to inject the failure. This teaches you a lot about running chaos-engineering experiments in a safe way.

IEEE Software (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; +1 714 821 8380; fax +1 714 821 4010. IEEE Computer Society headquarters: 2001 L St., Ste. 700, Washington, DC 20036. Subscribe to *IEEE Software* by visiting www.computer.org/software.

Postmaster: Send undelivered copies and address changes to *IEEE Software*, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854-4141. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8, Canada. Printed in the USA.