EPISODE 03

[0:00:01.3] JM: Servers in a datacenter fail sometimes. Sometimes the entire datacenter has a power outage. Bugs in an application can make it into production. Human operators make mistakes and cause data to be deleted. Failure is unavoidable. We make backups and replicate our servers so that when a failure occurs, we can quickly respond to it without making the user feel much pain. How can we test that our response will work before an actual catastrophe occurs?

Kolton Andrus is CEO of Gremlin, a company that works on failure injection as a service. Gremlin is based on ideas around planned failure that Kolton learned from his years at Amazon and Netflix. We ended up talking as much about the culture of Netflix and Amazon as we did about how and why to build failure injection.

It's always nice to share war stories with other people who have worked at Amazon, because the culture is so distinct. If you want to know more Amazon's culture, check out the episode tomorrow with Brad Stone, author of *The Everything Store.*

[SPONSOR MESSAGE]

[0:01:10.3] JM: Good customer relationships define the success of your business. Zendesk helps you build better mobile apps and retain users. With Zendesk mobile SDKs, you can bring native in-app support to your app quickly and easily. If a user discovers a bug in your app, that user can view help content and start a conversation with your support team without leaving your app.

The conversations go into Zendesk and can automatically include information about the user's app information, device information, usage history, and more. Best of all, this is included with Zendesk for no extra charge. Use the out of the box iOS UI to get up and running quickly, or build your own UI and work with the SDK API providers. Keep your customers happy with Zendesk.

Software Engineering Daily listeners can use promo code sedaily for $177 off. Thanks to Zendesk for supporting Software Engineering Daily, and you can check out zendesk.com/sedaily to support Software Engineering Daily and get $177 off your Zendesk.

[INTERVIEW]

[0:02:34.6] JM: Kolton Andrus is the founder and CEO of Gremlin. Kolton, welcome to Software Engineering Daily.

[0:02:39.6] KA: Thank you. My pleasure to be here. I appreciate the opportunity.

[0:02:43.0] JM: I appreciate you coming on. Netflix originally built a system called Chaos Monkey to make their software fail randomly. Why would we want to inject random failures into our software?

[0:02:58.0] KA: That's a great question. We should almost break that into two parts. Why do we want to inject failures and why do we want to inject failures randomly?

If you want to talk about the random bit, I think the value of random failure injection is that it forces you to be prepared. Similar to an outage that could occur at any time, if you're randomly causing yourself pain, it could happen at any point, and you need to be prepared.

[0:03:23.8] JM: What are the kinds of random failures that we might want to inject in our software systems?

[0:03:29.3] KA: Chaos Monkey is best known for rebooting host. It's a nice small level of granularity. You're moving to the cloud. You can't trust that your instances will be there. By forcing those to randomly terminate, you're forced to handle the natural state of a cloud-deployed application.

One of the side effects of that is that it tends to make your services stateless, because you can't rely on storing state on the box, because it could go away at any point. You have to move that state elsewhere.

[0:03:59.9] JM: It almost reminds me of the fact that early days of cloud systems — I guess this is still true, is servers fail all the time. They get replaced, and so that's what's happening at the hardware level. You're simulating that at the infrastructure as code level by just making these servers fail and restarting them.

[0:04:23.8] KA: Yeah. I think, additionally, as your service matures, you tend to do things like autoscaling, where you're going to add and remove instances in accordance with your traffic, or your influx of request you need to serve. For that reason, even on a happy case, servers can be coming and going throughout the day.

[0:04:43.3] JM: How did this random failure injection — Maybe we're just talking about the servers restarting. How did that kind of failure injection influenced the Netflix software development culture?

[0:04:56.3] KA: Yeah. I think one thing that's great about the Netflix culture is that when they made this decision that everyone had to participate, that they were going to break things on a regular basis, it became everyone's problem. This is something that we could argue engineers need to address and be ready for anyway, but by making it an important part of the engineering process, it becomes a first class citizen. It says, "Hey, we care a lot about resilience and ensuring that our software handles failures correctly. You need to be ready to handle that."

[0:05:29.5] JM: What are some of the more sophisticated types of failure injection that we might add to our system?

[0:05:35.2] KA: There's the good question. I honestly think rebooting a host is like your bare minimum. It is the basic attack that you can cause. More interesting failures — We have focused on a set of a handful of key failures upfront, things like, "Let's max out CPU. Let's take up a bunch of memory, force a garbage collection. Let's consume all the bandwidth to our discs and see what happens when we get stalled waiting for IO." That's an interesting class. I call those resource attacks. I think those teach you interesting bits about your system.

I think the most valuable class of attacks are the network attacks. We're all moving to distributed systems, microservice architectures. We're making a lot of network calls, and the network is fundamentally unreliable.

In that case, being able to simulate or emulate failures like a dependent savior slows down. It usually responds in a hundred milliseconds, but now we're going to make it respond in a second and a half. How do we handle that? What if that dependency just goes away? We want to black hole all traffic to a dependency and see how our service handles it. Can we gracefully degrade? Can we continue serving critical traffic, or are we done? Are we going to fall on our faces? Then, there're other variations of network traffic; packet loss, or corruption, or other things that you can inject.

[0:06:59.5] JM: Can you clarify why the network level is being classified here as more unreliable than other levels? Because we have all of these lawyers to a typical architecture these days; you've got the host level, you've got the container level, you've got the container orchestration level, you've got the cloud service provider level. I realize I'm mixing what I'm referring to as a level. Why the focus on the network?

[0:07:29.2] KA: I think that's just — I don't know. Maybe that's a traditional approach. If you had a datacenter, if you had a lot of hardware, if you had a lot of redundancy, you still have to communicate across the internet to other people, to other dependencies. That's the glory of the internet. You can talk to anybody. When you go out across a public network, you're no longer in control.

We can kind of discuss our debate. If you're on a cloud provider, the degree of reliability of the network is going to be somewhat influenced by them, but weakest link principle, we're going to hit a point where we're going out across the open internet and anything could happen.

[0:08:07.1] JM: If a company decides they're going to start implementing this failure injection and they start instituting these failures, how can they know whether something has broken as a result of a failure injection or because of something organic?

[0:08:22.8] KA: That comes back to this point about should you randomly failure test, or should you do it more explicitly? I'm actually a fan, especially in the beginning of purposefully causing failure, scheduling it, planning out an exercise. Thinking of it like an experiment that you're going to run, where you're going to learn something about the system. You're going to cause the failure, but you're also going to know what failure you caused. You could revert it or clean it up. You don't have to root cause it if things go wrong.

If you set up a test and exercise like this, then you know when it started, you know when it ended, you know what you think should happen. Like any good experiment, you should have a hypothesis of how you think that failure will impact your system. I think it's important to have things like success criteria. When things go well, what do we expect this failure to behave like? Most of the time, we don't expect it to impact customers, or we're going to gracefully degrade and continue on. Business is normal.

Then, a setup of abort conditions, "Hey, if we're ever impacting customers, if we're running this failure test and we've ever cause an impact beyond what we intended, we're going to halt it. We're going to clean it up. We're going to revert it."

I think that kind of an experiment driven approach lets you know when and where so that you can be looking for the how and the why.

[0:09:47.8] JM: If I hear you correctly, during a failure injection, if a failure occurs — A failure will occur. You may not know if in that time window that failure is an organic failure, or an injected failure, but it really doesn't matter. All that matters is that in this time window, you're looking at failures that occur. You're correlating them with other things that are going on and you're looking for solutions to those correlated failures.

[0:10:15.8] KA: That's a good clarifying question. I guess I would say if another failure occurs during your exercise, then as part of your understanding of that experiment, you're going to realize that that wasn't from the failure that you injected. That's still a great opportunity to learn and to adapt to that failure. It might even be a reason to stop your other failure exercise, because something else is going on. Two variables at once changing kind of a situation.

[0:10:44.0] JM: At Etsy, they do these GameDay exercises. I know this isn't unique to Etsy. In a GameDay exercise, they prepare for the failure. They will induce, and then they induce it. Is this the strategy that you're suggesting?

[0:10:57.6] KA: Yeah. I'm a big fan of the GameDay, or Dropbox calls it a DIRT, a disaster recovery testing exercise. This planned out thoughtful exercise, where you're going to cause a failure, you're going to train — It has a side effect. It's more than just being able to learn about your software. When you run a GameDay, you have an opportunity to train your team to help them to prepare for real outages. To do it during business hours, when the caffeine has kicked in instead of 2 in the morning when the pages goes off.

[SPONSOR MESSAGE]

[0:11:36.4] JM: Indeed Prime flips the typical model of job search and makes it easy to apply to multiple jobs and get multiple offers. Indeed Prime simplifies your job search and helps you land that ideal software engineering position. Candidates get immediate exposure to the best tech companies with just one simple application to Indeed Prime.

Companies on Indeed Prime's exclusive platform will message candidates with salary and equity upfront. If you're an engineer, you just get messaged by these companies, and the average software developer gets five employer contacts and an average salary offer of $125,000. If you're an average software developer on this platform, you will get five contacts and that average salary offer of $125,000.

Indeed Prime is a 100% free for candidates. There are no strings attached, and you get a signing bonus when you're hired. You get $2,000 to say thanks for using Indeed Prime, but if you are a Software Engineering Daily listener, you can sign up with indeed.com/sedaily, you can go to that URL, and you will get $5,000 instead. If you go to indeed.com/sedaily, it would support Software Engineering Daily and you would be able to be eligible for that $5,000 bonus instead of the normal $2,000 bonus on Indeed Prime.

Thanks to Indeed Prime for being a new sponsor of Software Engineering Daily and for representing a new way to get hired as an engineer and have a little more leverage, a little more optionality, and a little more ease of use.

[INTERVIEW CONTINUED]

[0:13:23.9] JM: It seems like at a company as big as Netflix, or Amazon, or Uber, there are so many naturally occurring failures that they wouldn't need to inject even more. Why is not the case?

[0:13:38.9] KA: That's going to be based upon the type of failures you're seeing. There might be enough random host going down or network blips that cause you to feel that pain and address it. When it comes to how do you handle one of your dependencies getting slow or one of your dependencies failing, those types of things may not be happening regularly. When they do happen, you want to prevent that outage.

If we're talking of moving to three nines and four nines kind of territory, you don't have a lot of time to be down. A good part of every incident involves just getting everyone involved, looking at the problem, diagnosing it, and then mitigating it.

If you've proactively gone out and tested that you can lose one of your dependencies and that business will go on as normal, you've not only mitigated the actual impact, but you've mitigated the time it takes to figure it out and to resolve it.

[0:14:36.4] JM: I recently was listening to a software engineering radio episode, that's a different podcast, with the John Allspaw from Etsy, and he was encouraging people to do it in production. Why should you do failure injection in production?

[0:14:58.9] KA: Another great question. I'm a fan of John's, and I think he's correct here as well. Production is what matters. Production is what serves your customer traffic. If you've tuned and you've tested things in staging, that may not apply on production. Think of things like your timeouts, your queues, the amount of throttling you're going to do. Those interactions between services.

In a happy case, or in a test or staged environment where you don't have full production traffic, you're just not going to be able to prepare for the scale that that failure may happen. The happy case tuning just may no save you. It may not protect you well enough when you're doing a production incident.

There's another quote I love, it's by James Hamilton, who's a distinguished AWS engineer. He came from Microsoft Research. He wrote a paper that I kind of consider one of my ops-bible papers on designing and deploying internet scale systems. In there, he essentially said that those unwilling to test in production aren't confident that their service will continue operating through failures. If untested, the recovery mechanisms won't work when called upon.

I think that second bit is also important. We build these mitigations, so Netflix has the ability to fail out of regions. If you haven't tested that you can fail out of a region on the fly, in production, in a short period of time, during an incident, then you may exasperate or lengthen the incident, or that tool just may provide no value to you in protecting you.

[0:16:37.0] JM: You have a lot of these quotes around failure and chaos on the gremlininc.com website, so those are from Nassim Nicholas Taleb, for example, and I think you just mentioned another. What are you favorite quotes about failure and crisis drills?

[0:17:01.2] KA: Yeah, I love that James Hamilton quote. I love Nicholas Naassm Taleb's anti-fragile view of the world. One of my favorite analogies to draw is that of the vaccine. When we think of something being fragile, we think of it breaking or not handling change well.

Typically, we think of the opposite of that as robust, or resilient, things that just handle change well. Anti-fragile kind of proposes, "We want something better than that. We want a system that doesn't just go back to being normal. We want a system that actually improves in the face of failure." This translates to organisms, to people, to ecosystems, to societies, things that adapt and grow stronger.

The vaccine analogy, while counterintuitive, fits that same bill. We're going to inject ourselves with something harmful in order to build an immunity. I think that translates very well into the

distributed system's world. We're going to go and inject some harmful behaviors into our system to build immunities to them so that when they occur, they're non-events.

[0:18:08.9] JM: You're building failure injection as a service. Why does that need to exist? Why don't people just import the library — There's an open source set of libraries around failure injection that Netflix has. Why build failure injection as a service?

[0:18:29.3] KA: Yeah. The crux there, it's not that hard to break stuff. I actually think it is hard, there's some subtlety to breaking things precisely as you want. There's more to it than just that. If you're just going to go out and start breaking things, that might be a place to start, but you're not going to learn as much as if you do it purposefully.

If you want to run failure injection and production, for example, you want to make sure that it's safe to run, that if something goes wrong, you're able to revert that impact, or clean it up. You want to ensure that it's secure. If you're building a system that could potentially break everything, obviously, that would be a huge target if somebody wanted to come cause you problems.

You want things like auditing and a history. You want — I'm a big believer that good developer tools make it easy for people to do the right things. You want a simple UI that guides people through it, helps them understand what's going to happen. You want a control plan that ensures that if your service breaks, or if anything unexpected happens, you stop the attack and you clean it up.

I think of Chaos Monkey, and I think some of these open source tools. They're that bottommost building block. They can break some things. A lot of those haven't even thought how do they revert the impact, or clean themselves up. Then, when you start layering in all of these other aspects that you want in a service, you can see where — It's more than just an open source project you plug in.

[0:19:58.9] JM: It's curious how these different SAS tools, many of them start off looking like things that are just small features that you want as part of your cloud provider. I think of continuous integration tools, for example. People had built entire businesses around continuous

integration tools. In the early days of continuous integration, we might have thought, “Oh, this is just something that AWS will bolt-on,” but it turns out to be such a rich ecosystem. There’s a rich variety of things that people want out of a continuous integration tool. Why is that the case with failure injection? Why is this a rich enough problem that it is worth breaking off into an entire company, rather than this is going to be something that’s a bolt-on AWS, or Google Cloud?

[0:20:46.7] KA: Yeah. I thought a bit about that. I think on one hand, cloud providers already failed more than they want to. I don’t know if they want to be culpable for purposely causing failures for breaking things. There might be kind of a third party audit role at play there.

I think, in general, because this is a new space, perhaps down the road, that that will make sense for them. Perhaps it will just be core to the way we develop software in 10 or 15 years. In the end term, not very many people are doing this. It’s tricky. You do want to get it right. The cost of being wrong can be high, and so there’s a value in getting help and expertise in this journey.

[0:21:31.0] JM: I was at this RedPoint event recently. You might have been there too. I don’t know if you were, but it was called *The Journey to Cloud Native,* and there was a panel of people including Craig from Kubernetes, or now he’s working on Heptio, I think. They were talking about what cloud services can be built that will not be subsumed by a cloud provider? We’re talking about a number of things, but kind of the struggles of building a business that is separate from a cloud provider.

Also, does it have to be open source? What parts of it are open source? What are the pricing models? What are the sales models? What are these big questions that you are asking yourself as you’re building the foundation for this SAS business that — As far as I know, I think it’s based on the open source tools.

[0:22:29.5] KA: Right. Actually, we’re not based on any open source.

[0:22:33.2] JM: Okay. All right.

[0:22:33.6] KA: We've had this debate a few times. Gremlin has been built from the ground up, we're not reusing Netflix's pieces. In fact, I would argue we're doing it a little bit better for some of the reasons I stated earlier.

We have a Linux compiled binary that causes the bad behavior that runs on any Linux distribution, and then a service, and a website, and the SAS offering that ties together. We've debated whether we want to open source the client, and at what point that makes sense. The difficulty or a young a company where you'd like to be around in a couple of years, means you need to make some money and be profitable. If we give away one of the core components of our business upfront, it's kind of a one way decision. In that point, we've debated that we think that, perhaps down the road, we may open source our client, but we're waiting to see when and if that's the right decision.

[0:23:30.1] JM: Can you talk about some of the differences between the open source tools, such as Chaos Monkey and the work that you've done internally at Gremlin?

[0:23:40.2] KA: Yeah. I think Chaos Monkey is most known for randomly rebooting host. There's a few other of the Simian Army, Kong, and Gorilla, that allow you to simulate zone and region failures. Then, there's a whole other host of monkeys that I don't think a lot of people know about. There's Latency Monkey. There are some other monkeys in there that muck with network traffic or do other things.

What we've built is — We've tried to build the superset of failures that we think are interesting. As I outlined earlier, a set of resource Gremlins, a set of network Gremlins, a set of reboot Gremlins. Right after the S3 outage, we made sure that we could properly emulate not being able to talk to S3 and have that built in as a building block in our system. We were confident after the DNS outage last fall. We went and wrote a DNS Gremlin. How does your system behave when you can't access DNS, or resolve certain host names?"

There's kind of the breadthwise in the number of attacks, and that library is going to grow overtime as we find things that are valuable for our customers to inject. I think, depth-wise — We talked a little bit about some of the Chaos Monkey scripts, maybe not Chaos Monkey itself, but some of the other monkeys, are kind of one-liners. They break something. They do

something bad. In my opinion, and respectfully, because they laid some very ground work here, they're kind of naïve solutions. They haven't been thought through real deep in not just how to cause the failure, but how to revert the failure.

One of the things that we think is very important is that all of the Gremlin attacks are reversible. In fact, if anything goes wrong, we'll automatically reverse it. That kind of safety guarantee is missing from something like Chaos Monkey.

[0:25:26.2] JM: Thinking about it more, you actually have a different set of incentives than the cloud providers, because you would theoretically want to build failure injection systems for iOS, Kafka, Linux, MySQL, Windows, all of these different systems which may or may not be in the purview of a cloud provider.

[0:25:48.3] KA: Yeah, everything fails, and so in the end, we want to ensure that all of our customer phasing systems are resilient, and that involves testing it at various layers.

[0:25:58.9] JM: What's the integration process for a customer that wants to onboard will Gremlin?

[0:26:04.2] KA: Yeah. Installation-wise, we do common Linux installation processes, RPM, Debian. We have a bash script. We use a public private key similar to AWS, so everything is authenticated and you're only granting access to the clients and the host that you expect to. We have the Web UI. We do single sign-on. In general, someone can come in, signup. Right now, we're in closed beta, and so you would email me and get a code, or I would send you an invite link and then you can log in to the UI. You install Gremlin on the host you want to cause failure on. From the UI, you can create those attacks, you can manage them, you can halt them, you can see the history, things like that.

[0:26:52.3] JM: You mentioned some liability that a cloud provider might have for breaking services. As a service, do you incur any legal risk yourself since this is the center of your business?

[0:27:06.1] KA: It's a good question. Right now, certainly, for our evaluation license, we have this little line in our legal document that says, "Warranty**:** This software is built to break things. It might break things in an unexpected way. We're not responsible for that."

That being said, I've kind of outlined a few of the things we do for safety's sake. I used to joke at Amazon and at Netflix. A little backstory. At Amazon, we also built a failure injection service. It actually came out about the same time as Chaos Monkey, but because it's Amazon, we didn't speak publicly about it. I always joked, I never want to be a part of a SevOne incident call, where my tool broke everything. Trying to build in those guardrails and those safety steps to help people run this safely without breaking everything is key.

[0:27:57.7] JM: You and I share the history of having worked at Amazon. I only worked there for eight months, and I was more of a spectator. I committed very little code. I'm not necessarily proud of that, but I enjoyed my time at Amazon. I left with my own accord.

You worked there for about four years, I think. Then, you worked at Netflix for two and a half years. I'd love to talk a little bit about that. What was your experience at Amazon?

[0:28:26.4] KA: I enjoyed my time at Amazon and I wouldn't trade it for anything. It taught me to be a better engineer. I got to work on some great projects. I had some great teams and some great team members. Amazon is a great place, especially, I think, for a younger engineer, because they really beat into you, these principles.

[0:28:46.0] JM: Basic training.

[0:28:47.1] KA: Yeah. It's a little bit like boot camp, and maybe that's why I referred to my time at Amazon as my four-year tour of duty, because when my four years and a week were up and my StockVest was done, it was time to move on.

[0:29:01.6] JM: That's how I'm starting to feel when I look at these different companies. You can look at Facebook, or Google, or Amazon, and think about Amazon, is that it is a regime. It is hard work and it prepares you to work for yourself. I'm not sure if you could say the same thing

about Google, of Facebook where you just have these lavish lunches, and benefits, and everything. Do you really want that out of a corporate job?

[0:29:31.1] KA: It's my own personal experience, but I look back on some of the hard times that I went through. I spent a year and a half at Amazon managing the retail website latency program, and it was our job to make sure Amazon was faster than everyone else, to measure it, to make improvements across the company.

Being a manager at Amazon was hard. I didn't enjoy it, but it taught me a lot and it forced me to grow. Now, especially, as I look back at CEO of my own company, of all the things I've had to do over the past year, that experience has been invaluable in preparing me for what lied ahead.

[0:30:06.9] JM: I just interviewed Brad Stone yesterday. He wrote the book, *The Everything Store,* which is about Amazon. I don't know if you've read that. Have you read that book?

[0:30:16.0] KA: I have not.

[0:30:16.7] JM: Oh! It's awesome. I recommend that. We were just talking about what makes Amazon different — I've spent eight months there, but I still think about this all the time. What do you think makes that company so different?

[0:30:32.1] KA: I think Jeff is a very smart individual and has great vision. I think that he's tapped in to hiring smart people that want to work hard, that want to prove themselves. We used to joke that most of the — It feels like most of Amazon is interns and entry-level engineers with a just a few senior people around to guide them and keep them on track.

That ability to go there when you're young, when you're hungry, when they're a big name, when you want to make an impact, you're really excited. People work really hard. I think that's a part of it. I think the other part of it is the value system. I actually think that the value systems at Amazon and Netflix are great, and there's a lot of their leadership principles that I identify with and that I want to be part of my company.

Customer focus; we care about our customers and customers being successful more than anything else. I think that's supper important. Bias for action; we like to get things one. Vocally self-critical; we're okay to say we screwed up, and we're not perfect, and we can do better. Some of those principles, I think, really help frame things and set the context well.

[0:31:50.5] JM: Between the Amazon management principles and the Netflix culture of slide share, where is the difference? What's the difference in the value sets of the two companies?

[0:32:02.8] KA: There's a lot of difference. I think it's the superset of the two that you find the happy medium. The things I love in Netflix; freedom and responsibility. Very seldomly that I have people telling me what I needed to be doing. When I came to people and I suggested — One of the things, when I came to Netflix, I thought, "Hey, cool. They've got failure testing down. I'm just going to come be an engineer on the Edge team and I'm not going to necessarily worry about that."

What I found was Chaos Monkey was being run all the time, but Latency Monkey and some of the other monkeys weren't being run. It's because they were causing too much impact. They were causing unintended impact when those exercises were being run.

I had the opportunity to come in and build another failure injection system on top of Chaos Monkey, kind of the V2, and it does application layer failure testing. You can kind of arbitrarily inject failure delay into your application, and you could distribute it across the SOA. You can do nitty things like scope it so that you're only going to break an individual request, or an individual customer, and then grow that scope so that in the end you're running for 100% of production traffic.

I tell that story because I wasn't on the Chaos Team when I built that. When I first went to my managers and I was like, "This is what we need to help us test Hystrix, which is a circuit breaker library and to make sure that our fallbacks work and we gracefully degrade.

They kind of gave me this look, "We've got the monkeys, and what are you doing? But, okay, let's see where this goes." Because they were willing to trust me and listen to me even though they didn't necessarily agree with me, had the opportunity to build this system, and it was very

successful internally. It helped us improve our service, the Edge service reliability from three nines to four nines. It cut our operational burden year over year by 20% both years I was there. They had a great outcome, but to be there, to get to that point, you have to trust your engineers. You have to give them freedom to think and to go try things, and to fail, and to see where it goes.

[0:34:13.7] JM: Okay. I think that's such a salient point, because I'm thinking back to my time at Amazon — I reflect on this occasionally. Basically, what I did at Amazon, I didn't work at all on the work I was assigned. Instead, I worked on the these side projects, and I tried to get a little bit of traction from other teams that might be able to support these side projects so I could spin up my own business idea at Amazon.

In retrospect, my execution was really bad. Maybe it would have worked, for example, if I would have been doing my work that I was actually signed in addition to the side projects. Maybe if I was a little bit more diplomatic about how I engaged the other teams that I was trying to — Or other stakeholders that I was trying to get buy-in from. It was an interesting lesson in diplomacy, and it sounds like at Netflix, when you were going off into teams that you didn't have a signed purview into, you were more successful.

What were the lessons you learned? Because I'm sure there's people out there listening who want to execute on some sort of thing that is outside of their assigned purview. What are the tips for having the right bureaucratic lever-pulling and diplomacy?

[0:35:29.9] KA: Sure. That's hard. I'll just say upfront. It's hard to get right. I think one thing I learned at Amazon from being a manager is kind of learn what the other side of the coin is like you learn the same as we learn in a lot of areas of life that no one really knows what they're doing, and the rules are all kind of made up and they're all just kind of doing what they think is best, or what someone told them to do.

Once you understand that, you're a little bit more comfortable going to a director, or a VP, or a manager of another team and just saying, "Hey, I want to sit down for 30 minutes and talk about what's going on here."

You go into that with a diplomatic approach. You talk about their problems, “How can I help solve your problems? How can I make your life easier?” If you can set up a mutually beneficial situation like that, people are often happy to participate. Especially, once they start to see the value and you’ve kind of convinced them that the idea is sound, then they’re willing to start investing some time and resources and start supporting you.

[0:36:34.6] JM: Yeah, I completely agree with that, because, I think, in retrospect, I took a very — I had to internalize an adversarial relationship between me and the big corporation. I don’t know where I got that narrative from, but that’s not the reality, and that was a mistake to believe. If you go into a situation where you’re like, “Oh, I’m going to try to build something inside of this company and try to lever the things that have been built before me, but it’s an adversarial thing. That’s a complete mistake.” If you look for the synergies and try to make win-win stuff, you can get pretty far.

[0:37:11.1] KA: Yeah. I’m reminded of a pulp fiction quote that’s probably not appropriate here, but as engineers —

[0:37:16.2] JM: No. It’s definitely appropriate. Whatever it is, it’s appropriate.

[0:37:19.9] KA: As engineers, we have to be willing to set aside our pride sometimes. That’s something that I’ve been able to see this past year being CEO. I wrote a lot of code in the first year. I’m writing a lot less code now. As I get away from the engineer in me, I’m able to be — I keep my passion, but not get so irate or worked up about little details, or pedantic things.

I think keeping your — It’s that keeping your eye on the prize. Keeping that overall vision of what you’re trying to accomplish so that you can stomach of the bumps along the way and you can just kind of swallow some of the jabs or the unpleasantries.

[0:38:01.6] JM: Dig in to that a little bit more, because I think people have internalized, or some people have internalized, “Oh, you want the Steve Jobs mentality, where you’re zooming in on every particular detail and every particular mistake that somebody makes. If they make a mistake and they ship it, you’re going to bear down on them and make them stay up for five

days until they get this thing exactly right and shipped out the door." It sounds like you're taking less of that approach, perhaps.

[0:38:31.4] KA: Yeah, it's interesting, because that makes me think of two things. On one hand, one criticism I have of Netflix's culture is they're big on freedom and responsibility, but the responsibility side is a little lacking. The ultimate responsibility is that you're asked to leave. The intermediate ways in which you hold someone responsible if they're having a difficult time or not making mistakes, is a little less clear.

The flip side, the other side to that is — In general, I want to hire a team and work with engineers that I would trust, that I respect. I want to treat them the way I want to be treated. That may not work for everybody, but my personality type is much more, "I'm worried about the little mistakes. If I screw up, I'm going to take it hard, because I'm my own biggest critique and I'm the one who's really going to feel bad and work hard to mitigate that."

If someone's already — If that's already the approach they take, going in and yelling at them, or telling them that they need to work all night, isn't going to solve your problem. In fact, if someone's behind the ball and you tell 'em, "Hey! You need to work twice as much." I think that's probably a net loss there.

I'm a fan of working a reasonable amount of time and having downtime. I think that's work-life balance and the ability to decompress and relax and not get too caught up in things.

[SPONSOR MESSAGE]

[0:39:59.3] JM: You are building a data-intensive application. Maybe it involves data visualization, a recommendation engine, or multiple data sources. These applications often require data warehousing, glue code, lots of iteration, and lots of frustration.

The Exaptive Studio is a rapid application development studio optimized for data projects. It minimizes the code required to build data-rich web applications and maximizes your time spent on your expertise. Go to exaptive.com/sedaily to get a free account today. That's exaptive.com/sedaily.

The Exaptive Studio provides a visual environment for using back end algorithmic and frontend component. Use the open source technologies you already use, but without having to modify the code, unless you want to, of course. Access a k-means clustering algorithm without knowing R, or use complex visualizations even if you don't know D3.

Spend your energy on the part that you know well and less time on the other stuff. Build faster and create better. Go to exaptive.com/sedaily for a free account. Thanks to Exaptive for being a new sponsor of Software Engineering Daily. It's a pleasure to have you onboard as a new sponsor.

[INTERVIEW CONTINUED]

[0:41:24.3] JM: Particularly, if you're hiring people for a big corporation like Netflix, these are people who have options to go work at startups, and if they want to be a workaholic that stays up five days in a row, they're going to be much better compensated for that sort of overwork at a startup where their impact is going to have a commensurate output in their salary. Would you agree with that?

[0:41:48.1] KA: In part. I think you can go to a startup and work hard and not make as much money as you make at a big company. I think that's the — If you live in the Valley, Netflix, and Google, and Facebook —

[0:42:00.4] JM: Sorry. I should have said expected value.

[0:42:03.2] KA: Expected value. Maybe that's a better way to put it.

[0:42:05.9] JM: Maybe. I guess it's a gamble. You're right, it's a gamble. A place like Netflix though, you can put in those five-day straight marathon sessions and you're still going to have a fairly fixed upside. I would suggest that if you're going to work like crazy, you should at least be working with uncapped upside.

[0:42:26.1] KA: Yeah. I agree with that. In principle, it's funny, because that's a little bit the anti-fragile principle, is you want to bound it downside and then unbound it upside. You want to be in situations in general where the worst thing that happens, you know about it, and you're okay with it. The best thing that happens is either way better than the worst, or could go to the moon. Startups feel like that sometimes.

When we started Gremlin, the downside was, "Hey, we might waste a year of our time, maybe two. We might not go anywhere. We might not make any money. If things die, we'll go back to Netflix, we'll go back to another company. Our lives won't be over."

The upside is, "Hey, if this takes off, if this goes somewhere, then we get to be in at the ground level and hopefully that's worth it in the long run."

[0:43:17.8] JM: Do you think engineers in general underestimate the amount of bounded downside? By that, I mean — I feel like if it were more internalized that, look, as a software engineer, basically, your career is super durable and the only world in which software engineers can't get a job is some world that is almost unfathomable from our current state of being. Why not take maximal risk with your career?

[0:43:56.3] KA: I agree with that statement that you made, in general. I want to knock on wood after you say that, because, then, tech falls apart and we're in the middle of the next bubble and we looked back on this article and we're just like, "Oh, man! Why did we say that?"

I do think that in terms of understanding that risk proposition, you need to think about it. I hadn't thought about it deeply. I always think about it kind of in the short term. If you're thinking calculus, it's like a local minimum, a local maximum. Sometimes you don't step back and think about, "Hey, what's my five-year plan? What's my 10-year plan? What's the worst that could happen? What's the best that could happen? What risks am I willing to take along the way?"

[0:44:41.9] JM: Maybe, what's the expected value over that entire framework that you've created?

[0:44:48.4] KA: Yeah. For people thinking about doing a startup, or people that are interested, one; feel free to email me. I'm happy to share with you my pain and my joy. I think even if we don't make any money out of Gremlin, I've learned an immense amount from my time already. The things you learn being a CEO, the things you don't want to learn; HR, benefits, hiring, recruiting, and I think a lot of engineers deal with recruiting. All of these things that you've learned about running a business, about selling to customers, about contract negotiation, about all these things, they give you more context. They make you more mature. They help you better understand not just our industry, but the world as a whole. I think that's where the time is value spent.

[0:45:37.1] JM: Yeah, I can definitely agree with that. I want to talk a little bit more about failure injection in Gremlin, and so on. Since we're talking about some big companies, we talked about Netflix and Amazon, it sounds like they have failure testing in some sense. What about people at Google and Facebook, or maybe other big companies that you've talked? How do they do chaos testing?

[0:46:01.0] KA: I know that Google runs these large scale disaster recovery tests. I think they're a little bit higher level than what we're really discussing here, because they're more business continuity. How can people do work? How can you continue to operate as a business if you lose one of your areas? I know that from Chaos Community Day last year, that they're working on some tooling similar to this inside. Some ways to programmatically let people inject failures and better understand their services.

Facebook came to Chaos Community Day two years ago and they spoke a little bit, but I don't really know what they're doing there. My impression is, is that they care a lot about reliability, but I don't know that they've really embraced this failure testing approach.

[0:46:53.3] JM: Do you think that makes sense for a company like Facebook though? Because I can imagine, Facebook — They're like, "All right. The core thing that battle test is you can log on to the facebook.com, you can look at your newsfeed, basically, maybe Messenger." A lot of these other things — The graceful degradation is kind of built in to the fact that it's not super important. It's like, "Okay. Does the trending news — Is the trending news server up?" "Is the

comment — If you make a comment, does that appear quickly?" Maybe these things are less important.

[0:47:31.2] KA: That's how we would like it to behave, and that's how it should behave. That doesn't mean that's how it actually behaves. What we find when we failure test is that we have these assumptions about how the system is going to behave and how it's going to fail. Most engineers, with complex systems in our heads, we usually get it wrong. There are subtleties, there are interactions that we didn't expect. They're knock-on effects.

Actually — I think the heart of that question is, "What's the cost of being wrong?" Facebook was down for about an hour last year, and it was estimated they lost $1.7 million in advertising revenue during that time. How many engineers and how many teams could be going out and proactively failure testing for that price?

I think you can extrapolate that to other industries. I was at Amazon, Black Friday was the most important day. We had incidents on Black Fridays, and they were very expensive incidents. Very expensive. In some regards, it's a little bit like buying insurance. You want to go invest that time and those people and that energy to be prepared upfront, because once it's done, it's done. You're not getting that money back. You're not getting back the impact to your brand, or your reputation. If the dollars and cents add up, which I think it definitely does for even most mid-sized and smaller companies, then you want to be doing this upfront.

[0:48:59.0] JM: Does chaos testing reduce the need for unit testing, or functional testing? Can you shift some engineering resources from one of those testing rules to chaos testing?

[0:49:10.4] KA: I think it's part of the same overall umbrella. One of the things we did at Netflix is we had — First of all, we had our failure injection system as part of a set of continuous integration, continuous deployment test. Quick story there — We wanted to test what are our critical services? What are the service that cannot fail? We went in the device lab, we ran Netflix on, Xbox's and Playstations, real customer devices. We spun it up, but then we would whitelist the critical services, and we would fail traffic to anything else. We would go through this process, "Can we login? Can we pick a movie? Can we stream? Does it work?" If that process

failed, we knew we'd introduced a new critical dependency, and we could go back and we could make a decision about that.

I think in terms of testing — Sorry. Do you want to —

[0:50:00.1] JM: I was going to say that's such an interesting point, because I was at an Uber meet up one time and I was talking to these Uber distributed tracing guys and they were talking about, "Yeah. Right now, we're working on figuring out what the critical path is through our system." I was like, "Oh, okay."

Talk more about that. Why is it important to understand the critical path in your system and why is that a hard problem?

[0:50:24.0] KA: Yeah. I'll just contrast it with what we did at Amazon, which was we all kind of knew what the most important services were, and we had a hand manage list. If you caused a website outage and you weren't on the list, you got on the list.

What it meant to be on that list was higher scrutiny, making sure that you were doing things well. If there was ever a large scale event, we paged everyone from those teams, or all of those on-calls to join immediately to cut down time to resolution.

You can debate — I guess the purist in me doesn't like that you should treat critical services different than noncritical services. Again, that's that engineering pedanticness. Ignore it. You can better prepare your critical services to fail. You can give them more of hardening, more work. Then, you should test that all of the noncritical services can go away without causing a problem. If you've done that, you've just simplified your architecture and what you need to keep in your head when things go wrong.

[0:51:25.0] JM: All right. I know we're nearing the end of our time. I know. It's been a really interesting conversation. You should come back on again, maybe when Gremlin comes out of beta, or something else, some other event. It's a really interesting conversation.

I guess, to close it off, why this problem? You could work on so many other problems. You've got a lot of experience. I'm sure you have other ideas. Why does this problem of failure injection inspire you as an engineer, as an entrepreneur?

[0:51:55.1] KA: Yeah, that's a great question. The little kid in me loves that my business card say, "Breaking things on purpose." I get to talk about — We go out and we break things. That's just a lot of fun.

I think, in my time at Amazon and my time at Netflix, I was also a call leader, or an incident commander. I joined every large scale incident. On one hand, it's kind of a rush. When you're the person in charge of fixing amazon.com, possibly, you've got some VPs, or SVPs on the call listening. It's both high pressure, but it's high value. You know that you're providing a service that everyone at the company cares about, because if the website is down, nobody is happy.

Because that's so important, that really just drew me into resilience engineering and high availability. I think it dovetails really well with the way software architecture is being done, and the way that we're moving things out and distributing it.

Maybe in 10 years, it's kind of boring and it's just how we do things. Right now, it's really exciting, it's new, and it provides a lot of value.

[0:53:02.7] JM: Yeah. I think your point there about Amazon — Again, harkening back to our earlier conversation. I guess, maybe, adrenalin. The adrenalin of Amazon is part of the unique lock in of working at that company.

[0:53:15.1] KA: Yeah. I think it factors in.

[0:53:17.1] JM: Kolton, thank you for coming on Software Engineering Daily, I really enjoyed our conversation.

[0:53:20.4] KA: My pleasure. Thank you for having me.

[END OF INTERVIEW]

[0:53:26.3] JM: Thanks to Symphono for sponsoring Software Engineering Daily. Symphono is a custom engineering shop where senior engineers tackle big tech challenges while learning from each other. Check it out at symphono.com/sedaily. That's symphono.com/sedaily. Thanks again, Symphono.

[END]