EPISODE 511

[INTRODUCTION]

[0:00:00.5] JM: The ways that applications can fail are numerous. Disks fail all the time, servers overheat, network connections get flaky and you assume that you're prepared for such a scenario, because you've replicated your servers, you have the database backed up, your core application is spread across multiple availability zones, but are you really sure that your system is resilient? The only way to prove that your system is resilient to failure is to experience that failure, to simulate that failure and to make swift responsiveness to that failure an integral part of your software.

Chaos Engineering is the practice of routinely testing your system's resilience by inducing controlled failures. Netflix was the first company to discuss Chaos Engineering widely, but more and more companies are starting to work it into their systems and finding it tremendously useful. By inducing failures in a system, you can discover unknown dependencies, single points of failure and problematic state conditions that can cause data corruption.

Kolton Andrus worked on Chaos Engineering at Netflix and Amazon where he designed systems that would test system resiliency through routine failures. Since then, he founded Gremlin, which is a company that provides Chaos Engineering as a service. In a previous episode, Kolton and I discussed why Chaos Engineering is useful and he told some awesome war stories about working at Amazon and Netflix.

In this show, we explore how to build a Chaos Engineering service, which involves standing up these Gremlin containers that institute controlled failures. To find the previous episode I record with Kolton as well as other supplementary materials described in this episode, you can download the Software Engineering Daily app for iOS or android. These apps have all 650 of our episodes in a searchable format. We have recommendations and categories and related links and discussions around the episodes. It's all free and it's also open source. If you're interested in getting involved in our-open source community, we have lots of people working on the projects and we do our best to be friendly and inviting to new people coming in looking for their first open-source project, whether they're looking for a big or a small contribution, we've

Greenfield projects, we've got very small tasks for people to work on. We've got all kinds of stuff, and you can find that project at softwareengineeringdaily.com, which I'll link to the GitHub repo or you can go to github.com/softwareengineeringdaily. We've also got a Slack channel, and I would love to see you there. So thanks for listening and I hope you check out the app. I hope you enjoy this episode.

[SPONSOR MESSAGE]

[0:02:57.1] JM: DigitalOcean is a reliable, easy-to-use cloud provider. I've used DigitalOcean for years whenever I want to get an application off the ground quickly, and I've always loved the focus on user experience, the great documentation and the simple user-interface. More and more people are finding out about DigitalOcean and realizing that DigitalOcean is perfect for their application workloads.

This year, DigitalOcean is making that even easier with new node types. A $15 flexible droplet that can mix and match different configurations of CPU and RAM to get the perfect amount of resources for your application. There are also CPU-optimized droplets, perfect for highly active front-end servers or CICD workloads, and running on the cloud can get expensive, which is why DigitalOcean makes it easy to choose the right size instance, and the prices on standard instances have gone down too. You can check out all their new deals by going to do.co/sedaily, and as a bonus to our listeners, you will get $100 in credit to use over 60 days. That's a lot of money to experiment with. You can make $100 go pretty far on DigitalOcean. You can use the credit for hosting or infrastructure and that includes load balancers, object storage. DigitalOcean Spaces is a great new product that provides object storage and, of course, computation.

Get your free $100 credit at do.co/sedaily, and thanks to DigitalOcean for being a sponsor. The cofounder of DigitalOcean, Moisey Uretsky, was one of the first people I interviewed and his interview is really inspirational for me, so I've always thought of DigitalOcean is a pretty inspirational company. So, thank you, DigitalOcean.

[INTERVIEW]

[0:05:03.0] JM: Kolton Andrus is the CEO of Gremlin. Kolton, welcome back to Software Engineering Daily.

[0:05:08.9] KA: Thanks. It's my pleasure to be here. Thanks for having me on again.

[0:05:12.2] JM: Yes, it's great to have you. The last time that you came on, we mostly discussed the motivations for why companies do Chaos Engineering, the philosophy around Chaos Engineering, but we didn't talk as much about how to implement it. We didn't talk about how to build a system for Chaos Engineering, but you've been working on Gremlin, which is a company that provides Chaos Engineering. Before you started Gremlin, you had been at Amazon and Netflix, and if people want to hear about those experiences, they can listen to your other episode that we did.

So you've seen the internal chaos systems that these companies, but the average company is not like Amazon or Netflix. So when you are designing Gremlin and you were trying to make this service work for the average company, were you able to port the same design principles from Chaos Engineering and Amazon and Netflix to the public service?

[0:06:12.3] KA: I think I've seen some things that translated very well from Amazon and Netflix and some things that were completely different. A couple quick examples; we didn't have to worry as much about security when we built an internal tool. We were able to delegate some of that to existing processes and existing teams and we could always hide behind, "Well, it's an internal tool. No one else should have access to it."

The types of failure modes that we saw though, those translated well. The types of outages that Amazon and Netflix experiences and the types of things that we saw often are the same types of things that most of our customers see, so there's a lot of overlap in that regard.

[0:06:52.6] JM: Let's say I have a company with healthy infrastructure and I want to create chaos in it. I want to do the kinds of tests that anybody who is implementing Chaos Engineering would want to do. I want to attack it. What's the deployment model for getting an attack into my application?

[0:07:12.3] KA: Yeah, and maybe it was my mistake for not saying our key tenants when you asked me the last question, but there're three things we really focus in on at Gremlin; safety, security and simplicity. So the installation, this falls under our simplicity bucket. We want to make it as easy as possible for engineers to do the right thing. I think I said that last time. That was a lesson we learned at Amazon and Netflix. If you want engineers to do the right thing, make it easy.

We have Debian and RPM packages for installing on Linux distributions or a container, and in the end it's like three lines and it takes more than five minutes to get up and running and running an experiment, we've done our job poorly. Most people are able to do it in a couple of minutes.

[0:07:54.8] JM: So the deployment can either be done directly into the application, the operating system that is running my production application, or I can have like a container that contains the code of the attacking system, the chaos system.

[0:08:13.8] KA: Yeah, the containers. So containers are obviously very important. We've seen a lot of our potential customers and early customers exploring Kubernetes and starting to move toward container architectures. So we wanted containers to be first class citizens. So not only can you deploy Gremlin as a container, but we allow you to either — You can attack the root node from that container or you can attack adjacent containers.

We can do things like you've got a node that's running 20 containers, but there's a specific application you want to target by container label. So within Gremlin, you can deploy Gremlin on to that host either via the packages or the container and then you can target either the entire host or just a subset of containers on that host.

[0:09:00.1] JM: Help me understand the deployment model a little bit better, like am I deploying like my Chaos Engineering to this container that is going to administer all the attacks or do I ever want to like, let's say, I have got the service that provides — If you send a request to, it responds with a cat picture. It's a server that does. Would I ever want to deploy the actual Gremlin attack code to that server or do I really want to segregate it in this separate attack container that administers the attacks remotely from another container?

[0:09:34.7] KA: Yeah, I think that's just a question of your architecture. One of the things that matters there is this concept of the blast radius. We always want to run the smallest experiment that will teach us something. So if we can target a subset of our infrastructure in the non-container world, kind of the level of smallest granularity, is the host, and so we cause failures at the host level, but in the container world, we can only run it against that smaller subset of that application.

There's a bit of a question there. The other thing that makes me think of is when you're talking about where do you run a failure exercise, that's a deep question. One of the things that happens is your infrastructure might fail. You might have a top of rack switch or a network device fail, but that isn't necessarily where you would want to run that chaos experiment. That might indirectly affect too many things or it might be too difficult to get on that router and cause the failure the way it happened.

So one thing I found myself explaining a couple of times I think is super useful, is when we're dealing with a failure across the network bound — So I have a service and it serves cat pictures and that's my —I'm making a network cop. Let's say those cat pictures are stored in S3. Instead of going and breaking S3 itself, we simulate S3 failing from that service. So similarly, instead of failing network devices or third-party services themselves, we simulate that failure by running on that service, on the cat picture service.

[0:11:05.8] JM: I see. Let's walk through some of the typical chaos tests that I might want to run. You just alluded to one of them, but if were talking more specifically about the types of failure domains we want to test, the most simple one is resource failures. So we've got some basic resources we can work with, CPU, memory, I/O, disk. Let's talk about those. What are some real-world scenarios that could cause me to run out of CPU? Before we talk about the actual failure testing, when could this actually happen where I would run out of CPU?

[0:11:43.2] KA: Yeah. So part of that is understanding for a service that you own and operate, what is your resource that you're bound by? Traditionally a service is going to be bound by one thing, the most expensive thing, and so when it's under a real load or duress, that's the dimension that's going to be pinched or is going to be most competed for. You could imagine —

And akin to CPU, I would throw out threads, because we've seen a lot where, "Okay. I'm thread bound, and so I can't do any more work, because I'm rejecting requests. I just don't have any more capacity in my system."

It could be that you have a very computationally expensive process. Think of like lambda functions that do a lot of calculation. In that case, if you're running that and you hit an unexpected load or a more expensive code path, then all of a sudden all of the processes on your service are going to be maxed out. You're going to be rejecting load or things are going to slow down because there's just not enough time to get things through.

Disk is a really easy one. We actually wrote a blog post because we got burned by this. We had our service up and running, we thought we had log rotation happening correctly, and then we hit a point where our services failed and we dug into it and that wasn't working the way we expected. We hadn't yet wrote the disk Gremlin at that point, but we knew we needed to, and so we went and wrote that Gremlin and then we went back and we failure tested it. We did another game day where we ran running out of disk to make sure we caught it. We were alerted, that we heard about it when we could and that we were taking the appropriate remediation action.

[0:13:19.8] JM: Did that occur — You had a service where you were just storing the logs directly to the disk that was attached to that service or did you have a remote logging server where you were shuttling those logs to and that logging server failed?

[0:13:35.7] KA: Yes. So we were keeping it simple. It was just local on disk storage of logs, and even though we back some of them up in other places, the local copies. The service had long — Was running for a long period of time, and then you could still hit the point where you filled up disk.

[0:13:51.7] JM: So in order to test some of these failure domains, you have to create attack vectors that disrupt the specific resource that you are targeting. You just gave examples for how CPU and disk could be exhausted. How do you test those?

[0:14:08.7] KA: Yeah. So then once you know that this is something your system has encountered or you've identified that that's your resource bound that you care about, then you

want to go — It's almost the same as any other chaos experiment. You want to go you — You have your hypothesis. You thought about what could go wrong.

The second step before you even jump to running the experiment is thinking about how you think your system will behave. How do you measure this failure? In the case of disc, do you have a metric that tracks available disk space? In the case of CPU, we all have CPU or a load average metric that we're looking at. Then you theorize how you think that failure will impact your system.

Then the next step is to go run that experiment, and so then that's — I mention the concept of the blast radius, but, really, one of the things we want to do is we want to run the smallest experiment that will teach us something. If we run it for a single host, will that show us how it behaves?

To boil that down, when we go and run a failure tests for the first time, we're going to start with a single host in dev or staging and we're going to go ahead and we're going to max out at CPU or we're going to max out at disk space, and that'll show us how a single host handles it, which isn't enough on its own, but it gives us a glimpse. Then we're going to — If it fails, we're done. We found a failure. We found something we didn't handle well.

Then if it succeeds, then we're going to scale it up, then will run it for two hosts or five hosts or 10 hosts, and we may see different failures as we start to grow that failure scope. We may see different impacts to the system.

[SPONSOR MESSAGE]

[0:15:49.3] JM: The octopus, a sea creature known for its intelligence and flexibility. Octopus Deploy, a friendly deployment automation tool for deploying applications like .NET apps, Java apps and more. Ask any developer and they'll tell you that it's never fun pushing code at 5 p.m. on a Friday and then crossing your fingers hoping for the best. We've all been there. We've all done that, and that's where Octopus Deploy comes into the picture.

[INTERVIEW CONTINUED]

[0:17:21.3] JM: Another constraint is memory. So when I'm running a bunch of applications on my laptop, I have this happen, I run out of memory, I'm trying to edit a podcast, I'm trying to play a 3D game, I'm trying to run 40 chrome tabs. It's just too much stuff. On a server, I've seen systems run out of memory, because the system is not able to do garbage collection quickly enough. Like I've worked with Java in some server environments where the garbage collection is not configured properly and the system just gets exhausted. What are some typical reasons that an application can unexpectedly run out of memory and how do you simulate the memory failure?

[0:18:03.6] KA: Yeah. You've nailed them. It's, I've got a memory leak, I'm not garbage collecting correctly. Java processes can have some trouble with those things. You go and you — Really, we write a process that just holds on to as much memory as possible, and then what happens is when the operating system, when the application, when the virtual machine tries to hold onto more memory and it's unable to, we see the kind of behaviors that a system might run into. Things might get pushed in the swab or it might block or it might fail and terminate, and that's exactly the types of things we're trying to uncover.

[0:18:38.7] JM: Can you describe what is actually going on? If I administer an attack that seeks to exhaust memory, what are you doing under the covers?

[0:18:49.3] KA: Yeah. So I mean the memory wants a fun one, or the CPU want to talk about, because I think those are kind of your entry level failure experiments. If you were going to write

this yourself and you want to take up a lot of CPU resources, you're going to do a lot, for example, floating-point operations in a tight loop across a number of threads and you're just going to try to hold on to as much resources as you can.

For memory, you're going to write a bunch of random strings into memory and just hold on to that memory. Now, there're some subtleties. Linux is really good about reclaiming unused memory. So keeping it active, keeping it paged, making sure it's in the right place. There are some subtleties to make sure you get that to behave correctly.

[0:19:30.1] JM: Can you dive a little deeper? I'd love to know about some of those subtleties.

[0:19:33.5] KA: It's really those ones that I mentioned. The Linux operating system is really good about, "Oh! You wrote this memory, but you never touched it again, or you're not coming back for it, or it's in cache." So even though you think you're using that memory, it's still available to other processes, and so it's not really creating that pressure that you need it too, because Linus is just like, "Look. I know what you're trying to do here, and it's a sieving you bad code." It's like, "No. You want this memory, but you're not really using it. I'm going to set it over here."

I mean a counter example, when I wrote this type of failure mode in Amazon, I chose to do it in a JVM, because the JVM is really good at just taking a bunch of memory and not letting anyone else touch it, and that was like a simplifying assumption. We didn't do that this time in Gremlin. We had to just dig a little deeper and make the operating system behave the way we wanted it to. Yeah, those are the kind of trade-offs.

[0:20:30.0] JM: It's more faithful. For that instance, if you really want to exhaust the memory of Linux, you've got to write all these strings to memory and then maybe you have to keep reading them so that Linux actually thinks that you care about the strings.

[0:20:45.1] KA: Yeah.

[0:20:47.5] JM: I see. When you talk about these things that are going to exhaust CPU or exhaust memory, it seems like if you do it wrong, you could do a test where you're trying to disrupt the CPU and you end up disrupting the memory, like by virtue of — I'm a little bit rocky

on this, but I think like memory is partly — The heap is the variables the you're keeping in memory and then the stack is the operations that you're throwing on. So it seems like if you're trying to disrupt CPU with a bunch of floating-point operations, those operations take up some space. So can you accidentally do a memory test when you're disrupting the CPU?

[0:21:30.1] KA: Actually, what we ran into was the other way around. Some of the disk in the memory ones, writing all those files, reading all those files, if it's unbounded, those can be very CPU-intensive. So our experiments have a set up phase for memory and disk where it allocates everything before it waits for you to go see the full effect on the system. So during that ramp up space, we actually see a pretty heavy increase in CPU while we're getting set up.

[0:22:00.1] JM: Does that matter? Does it matter if you accidentally break the system because of CPU when you're trying to break it because of memory or do you have a way to report that? Like if somebody does a CPU test and destroys their server and their conclusion is that, "Oh, this is because of CPU," when actually it's because of memory. They could make erroneous fixes to that. So would I be able to determine where my breakage is?

[0:22:27.1] KA: Yeah. Actually, you've touched on a greater point there, which is does Gremlin tell me how my system fails, and it's not something we're in the business of doing. We think that you have dashboards and alerts, monitors and ways in which you already use and understand how your system behaves. In addition, it's really hard to come up with a, "Yeah. This system gracefully degrade it and handled this odd circumstance correctly," because those are often based on business outcomes. Was your customer able to do what they wanted?

So while we try very, very careful and we do a lot of work to validate that the failures we inject behave the way we expect them to, there's still a bit of the exercise left to the reader to understand how that impacted the system, and it's really a tool there for you to better learn your systems, to dig in and understand those trade-offs. It might be that when your disk is full, you're seeing a lot of CPU spike as well, because things are writing heavily to it. That may be close to reality. It may not be.

[0:23:31.4] JM: So then why would it matter if I used Gremlin versus just some macro load testing tool? Some blunt force trauma load testing tool? If I just need to just have monitoring

instrumentation in order to know what the problem is going to be in any case, why would I use Gremlin over this blunt force load testing system?

[0:23:52.9] KA: Yeah. That's where — You can run a lot of this by hand. You could write some scripts, and a lot of the open-source projects are kind of those V1 implementations that they work to cause a failure.

What we found is it's the safety and the security that really worked together, plus the web interface to really make it a product and not just the library. So an example of that, the safety features, every Gremlin attack is built to be revertible, and so there's an undo button. There's a stop button, and if it any point things aren't working the way you expected or if anything happens with our control plane, then that attack can be halted. It can be reverted. You can failsafe to steady state where things are no longer impacted.

Having that kind of guarantee gives you a lot of peace of mind. The example I gave, you could run, for example, IP table's commands or TC commands or Linux commands that impact network traffic, and if you're running them by hand on a remote host and you make a mistake, you could block off your SSH access to that host or you could make that host unreachable for anyone, and at that point, if that host was critical and you needed to restore it to service, you're out of luck.

So the safety feature is super important, and then security, just in terms of running this ultimately in production, at scale, you want to have some auditing. You want to have — At Gremlin, we have a security engineer on staff. He's a super smart fellow, ex-RSA. We audit and pen test our API, our client, our service, our own infrastructure to make sure we're doing things well. We're using lease permissions and we're being very careful about what the client does.

[0:25:37.7] JM: Yeah. It makes a lot of sense. So let's get back to these attack types. Another category of attacks, we're not talking about necessarily the host. We could talk about the network, and network attacks, the simplest kind of network attack is the black hole. You could have the network drop all of my traffic between service A and service B. This is something that you would want to simulate, because this could definitely happen if there's a network partition. If

all of a sudden the network gets flaky and service A can't reach service B, but how do you simulate that? Give me a description for the simulation process.

[0:26:19.6] KA: Yeah, and I would say, it's more than just network partitions. Again, back to our conversation earlier, that service could get slow for a variety of reasons. Maybe they've got database issues. Maybe they've got memory or CPU issues. But to us, across the network bound, it either looks like they got slower or they didn't respond. So in some regards, especially in a micro-service architecture, every time you're crossing a network bound, you can simulate that service failing or degrading by just delaying its traffic or black holing them.

So what we do is, on the host that happens. So your service A calling service B. We would run an experiment on service A. We would correctly identify the network traffic that goes to service B. That's kind of our blast radius or our failure scope that we want to apply there and we're going to make sure there we're only breaking traffic to service B. Then the first step might be to just black hole it, just drop all traffic to B, to and from B. That shows us what happens on service A when it can't talk to service B. That's usually — It's binary. We handle it well or we don't. We can degrade and we can give like a fall back or a cached response or we fall on our faces.

Then the second step is, "Okay. Now we now we know how the hard case behaves. What if it's the gray area, where it's just degraded or it's slower?" That's where we would say, "Start with 100 ms delay. Add 100 ms to every one of those requests. How did things happen? Then add 500 ms, add a second, and you slowly increase that. What that helps you do is it shows you the point at which you start timing out. If you set your timeouts correctly, it shows you the point at which your application becomes unusable to your dependencies, and it really helps you fine-tune those values as you try to figure out that balance between waiting as long as possible.

An example, we were doing failure testing on ourselves again against Dynamo, and there's this trade-off where we have these right operations or these reporting operations that take a long time. We want to wait as long as possible for them. There are these really short read operation that we don't want to wait any time for. So making sure that we've got those timeouts and those two classes tune correctly is a really good example in my opinion.

[0:28:36.3] JM: Yeah. Well, your colleague wrote about that Dynamo DB test. You did a game day where you are testing the resilience of Dynamo DB, which you use for much of your data management. Describe this game day event when you were giving Dynamo DB all that you could.

[0:28:53.7] KA: Yeah, and I'll just give a little call out to Phil there. He was the first engineer that joined our team and he's a big fan of Software Engineering Daily. So he'll loved that we talked about his blog post.

[0:29:02.3] JM: Oh, really? Oh, okay!

[0:29:05.8] KA: Yeah. For us, Dynamo — Yeah, we store a lot of data in Dynamo and it's our source of truth, and so knowing that we handle it correctly was very important. We ran a game day earlier this summer where most of our resource tests and most of our other tests went smoothly and we got Dynamo and things just fell apart, and we found a lot of things that we had to go fix and make better. The way our UI handled some of those timeouts or some of those failures was not what we wanted it to be, and so we went and improve those. We found this timeout issue where the default timeout in Dynamo is very high and, again, X-Netflix, X-Amazon, we should know better, mea culpa. There's a lot of trade-offs of building software. But going in and really protecting ourselves against those timeouts, but at the same time knowing that a good portion of the application would be unusable if we don't wait as long as we can, really, just balancing those.

What it came down to was those read and write operations and reporting operations that behave in completely different call patterns and building in timeouts that were representative of those classes so that we could balance those concerns.

[0:30:15.1] JM: Yeah. Well, I'll give a shout out to Phil as well, because I enjoyed reading his article, and it did make me think about something that is kind of, I guess, hard to address with Gremlin, which is — Well, I mean — Or, really, with any service that I'm aware of, which is the fact that we're increasingly dependent on these managed services, which is fantastic. Of course, I love managed services. I love the fact that they let us build so quickly, but auditing something like Dynamo DB even to the degree that you did is fairly opaque. It's very hard to —

These services aren't going to give you all the introspection that you would ultimately want out of them. If you wanted to debug like EC2, well, you're going to hit certain limitations for what sort of logs you're going to get around that. Fortunately, EC2 is a very battle tested service, but for less battle tested services, you're going to have more difficulties and equally difficult logging problems. So what advice do you have for people who are all in on managed services who perhaps don't have a whole lot of trust in them and are afraid that they might be the brittle weak point in their infrastructure?

[0:31:36.2] KA: If you phrase it like that, I say your instinct is correct. You should be cautiously — Not even cautiously optimistic. You should be suspicious of all your managed services in a good way, because if those fail, it's still your responsibility.

This is one thing I've heard a bit, and that's why I call it out. If S3 goes down tomorrow and it brings down my application, it's still on me. I chose to use S3. I could've built redundancy in. So I need to handle that failure correctly. I think this is why Chaos Engineering is becoming more important. Not only have we seen this increasing complexity in our systems in general as we move to micro-services and we start shifting everything apart, but now our infrastructure is no longer in our control. Our database is no longer in our control, or this other area is no longer in our control.

So the only — Well, one of the most proactive ways to really go and test that is to go failure test, to go cause it to break and see what really happens, because then you can make these trade-offs. An example that I may have used before — But gracefully degrading, if you have a service and your customers — The ones I always use that Netflix is the recommendation service. If you rely on the recommendation service to tell someone how much you think they'd like a movie, that's great, but if that service fails, then give them a cache value. Give them an empty list. Don't recommend. Don't tell them how well you'd recommend it, but still let them stream or still let them do their work.

So it's those kind of trade-offs where once you see how it fails and you see that you don't handle it well, you can start to say, "Okay. Well, maybe in this case, that's game over." Like DRM and Netflix was one where it was like, "No. We got to have DRMs. So if that goes wrong, we have to be real careful."

But these other things that are nice to have, Expedia spoke about this at Qcon about like their ads. Their ads are important. They like to make money, but if it's a choice between booking your hotel or showing an ad, book the hotel and let the ad just go away.

[0:33:45.0] JM: Indeed. Let's come back to the network discussion. There are some less deterministic problems that you want to simulate, problems that are less deterministic than just creating a — Well, I know you said it wasn't just network partition, but problems that are different from just dropping all of the traffic. So these less deterministic problems that you want to create with your attack system. Like you would want to create some packet loss, you would want to create some latency, because often times this is how services "fail" is with incomplete failure. They start to drop packets, they start to cause latency and may be if you've got a request, that it's going to jump through 18 different services. It's not a big deal if you get a little latency sometimes, but maybe there's some amount of latency that you add in one of those services and all of a sudden it creates the effect of the overall request failing. So how do you implement that as a set of attacks where you drop random packets or you introduce latency? What's a safe way to do that?

[0:34:52.1] KA: Yeah. A safe way is an interesting question, because if you're dropping packets, I mean there're two things that happen, right? If you're using TCP or a reliable protocol overall, those packets will be retransmitted. They'll still attempt to get there. You're really just adding more latency, but it's more of a jittered latency. With latency itself, you're just delaying. So it's still relatively safe. It'll get there, but it's those knock-on effects that are the not.

So from the implementing failure perspective, those are fairly straightforward. We go into the network, we drop packets based on the rules or based on the matching or we delay packets or we introduce packet loss. Certainly, there are things you can do with like distributions in terms of how random you want to make it and what that looks like. But it's that knock-on effect to your application of, "Yeah —" In fact, the exact example you described. I have request, it actually hits 16 different services. If all of those services got a little slower or if one service got much slower, do I hit some overall boundary where the request fails?

[0:35:57.9] JM: So another form of network chaos could be the inability to access DNS servers. I know this is another feature that you've built into your system. Can you explain what happens when an application cannot access DNS servers?

[0:36:13.8] KA: Oh, man! Most applications, it's not real pretty. It's funny. We wrote this after Dyn got DDoS-ed last fall. And so a lot of people were unable to resolve DNS and we thought, "Okay. This is the perfect example of what you'd want to do with Gremlin." So there's kind of two ways I see this playing out. On one hand, if you are using discovery and you're addressing everything by an IP address, you're probably relatively immune to this type of failure. There's still a big question about whether your customers can get to you if they can't resolve DNS, but there's not much you can do about that other than have a different DNS provider or have multiple routes.

Then if you don't use a discovery or your discovery is based upon DNS name and your services can't talk to each other, well that's where things melt down, because if that's one of those base building blocks you've used and you expect it to be a reliable and expect it to be there and it goes away, the house of cards kind of crumbles.

[SPONSOR MESSAGE]

[0:37:19.7] JM: Your company needs to build a new app, but you don't have the spare engineering resources. There are some technical people in your company who have time to build apps, but they're not engineers. They don't know JavaScript or iOS or android, that's where OutSystems comes in. OutSystems is a platform for building low code apps. As an enterprise grows, it needs more and more apps to support different types of customers and internal employee use cases.

Do you need to build an app for inventory management? Does your bank need a simple mobile app for mobile banking transactions? Do you need an app for visualizing your customer data? OutSystems has everything that you need to build, release and update your apps without needing an expert engineer. If you are an engineer, you will be massively productive with OutSystems.

Find out how to get started with low code apps today at outsystems.com/sedaily. There are videos showing how to use the OutSystems development platform and testimonials from enterprises like FICO, Mercedes Benz and Safeway.

I love to see new people exposed to software engineering. That's exactly what OutSystems does. OutSystems enables you to quickly build web and mobile applications whether you are an engineer or not.

Check out how to build low code apps by going to outsystems.com/sedaily. Thank you to OutSystems for being a new sponsor of Software Engineering Daily, and you're building something that's really cool and very much needed in the world.

Thank you, OutSystems.

[INTERVIEW CONTINUED]

[0:39:12.1] JM: So what do I do to protect against this in real-life? Do I need to have some sort of backup DNS system?

[0:39:21.0] KA: Yeah. Big internet scale companies typically have multiple DNS providers and the ability to shift between them if in the case that something occurs. DNS itself, as a large scale, doesn't fail that often, and so you might just be comfortable with that risk. Some people thought, "Okay. Well, if my customers can get to us, it doesn't really matter if we're broken." I don't love that answer, but that is an answer.

With a lot of things, it's these set of trade-offs now. You now know for certain how your system behaves if these things occur, but that doesn't necessarily mean you have to solve it right away. It means now we can make the business case about the time and investment it would take to mitigate this problem and how likely we think it is to occur and how bad it is when it happens.

[0:40:09.2] JM: Let me ask you a question. this is not really related to what we're talking about here, but you must think about Chaos Engineering at every scale. The most chaotic scale I can imagine is something like Dyn where you have this thing that is supposed to just be a service,

but it turns out this is a key point in internet infrastructure globally. Do you think at all about too big to fail when it comes to things like AWS or — I mean, Dyn is a perfect example, but Dyn is a drop in the bucket compared to what would happen if AWS had some sort of global failure. I can't remember if we discussed this last time. I talked about this with somebody else recently about what would happen if AWS failed? I think one of the things — What they said was — One of the things is like if, luckily, if AWS fails for you, it's failing for everybody else. So everybody goes down. They were saying that as like a redeeming aspect. I'm like, "Okay. Well, I guess that's true. Everybody fails at once, so nobody gets an edge," but it's still like going to take down our power grid by some downstream effect.

Does that concern you at all or do you think we're moving to a place where we have less vulnerability to that kind of event? What are your thoughts on the too big to fail of internet providers?

[0:41:33.5] KA: Yeah. It's definitely an interesting thought. On one hand, I trust that the engineers at Amazon, the engineers at Azure or Google, that they're following availability practices. It's the bulkheading approach, that if you isolate things, you never want the whole ship to sink. It's okay if one bulkhead fills up with water. That's a lot better than the whole ship going down.

The costs of other that — We've had two or three major airline outages this year that have been hundred million dollar plus events. When S3 failed this year, it cost everyone other than Amazon $150 million. It has a demonstrable effect, not just on our leisure and on our education. but on our business, on our travel, on people's health.

Actually, personally, I'm very passionate about this. This is why I'm working on Gremlin. We want to make the internet more reliable. The internet today is probably two 9s territory, maybe three 9s of availability. We want it to be four or five or six, like our phone, like the telco infrastructure, like some of the other things that have really been hardened, because so much rides on them. It's so important.

[0:42:47.0] JM: Absolutely.

[0:42:48.5] KA: For fun, the other one I think about is self-driving cars. I lane-split on my motorcycle in the Bay Area, and for me it's like I can't wait for self-driving cars, but also we got to failure test those. What happens when, heaven forbid, a motorcyclist goes down in front of cars? Well, today it's a really nasty scenario, but in the future it could be awesome if the cars automatically adapt and move around them. Having driven in some early cars that do auto drive, self-driving things, they're not there yet, and so Chaos Engineering and like testing for those failures is super interesting to me.

[0:43:28.4] JM: Okay. This is so strange to me why you would — Not to get philosophical. Why you would ride a motorcycle? It's funny, because my first introduction to distributed systems was a distributed systems class I took where the professor drove a motorcycle and he's this brilliant distributed systems professor. I think he's at Cornell now, Lorenzo Alvisi. I'm sure if somebody listening, has taken lessons from him, and he's a brutal professor, but he spends all day lecturing about single points of failure and what goes wrong if you don't ensure every single thing in the distributed systems that you're working with are insured against Byzantine failures, and then he walks at a class and hops on a motorcycle. I'm like, "That is — You're not really practicing what you preach." That's what I always thought. I mean, he's wearing a helmet. That's great, but like, to me, there is just some like disconnection. Unless your belief is that, "You know what? We just live in a world of chaos. It's hopeless to try to ensure against everything. We might as well have a little fun and ride a motorcycle." Is that your philosophy? Is that what you ride a motorcycle?

[0:44:40.2] KA: My dad rode a motorcycle. I started riding from a young age. Impart, it's just culturally ingrained in me. I feel a little disingenuous. Sometimes I have the thought, I was like, "If I really believed in this Chaos Engineering stuff, I just lay my bike down here and see what happened." I'll do some real failure testing. Then the cost of being wrong there is really high.

A lot of times I think of it, again, in terms of trade-offs, right? I enjoy riding a motorcycle. In the Bay, the difference between driving from San Jose to San Francisco during traffic on a motorcycle versus a car during rush hour is one hour, versus two hours. So that time savings is a real efficiency. I also mitigate the risk. I wear the boots, I wear the pants, I wear the jacket, I wear the gloves, I wear the helmet, but every now and then I'm like, "Wow! That was a little close or that felt a little risky. Why am I still riding a motorcycle?"

[0:45:34.5] JM: Okay. You've gotten some pretty large clients working with Gremlin. What have you learned about delivering chaos as a servicing? A company like Twilio, for example, that's a serious infrastructure company. How do you deliver chaos to that kind of client?

[0:45:52.9] KA: Yeah, you got to make it simple. That tenant about simplicity has really served us well, simple to install, simple to automate installation, simple to understand, simple to use. It really brings that barrier down.

A big part of what we're doing, especially this coming year, is just teaching the world about Chaos Engineering. Sharing what we've learned, best practices, how to do it properly. So that combination of make it easy for people to do the right thing, give them an undo button in case they make a mistake. So it's okay to make mistakes. It's okay that everyone's learning. Now, you can go out run these experiments and learn how things behave, and then just support, just teaching videos, podcasts, Slack channels, whatever we can do to really help people to understand this and to get started.

[0:46:44.0] JM: Do these companies typically have Chaos Engineering practices before they start working with you, or do they typically come to you and say, "Hey, we've been wanting to do this for a while, and now help us get started with Chaos Engineering."

[0:46:59.2] KA: What's been amazing to us, especially even this last month or two, many companies have come to us and said, "We're doing Chaos Engineering in 2018 with or without you. What do you have? How can you help us?"

I actually think it works best for these early adopter, early majority companies if they have a chaos champion inside, someone that really feels the pain. So I'm a big believer in skin in the game. Like you really see the value when you felt the pain and then you've solved it and it goes away. Then those people, once they are on board and they understand it and that they see the value, their coworkers are much more likely to listen to them than they are to me. So helping them to be successful and really empowering them, that it scales the education and it and it helps with the teams that are like, "Well, that sounds cool, but I don't know if we can do it, or I don't know if that's what we want to do now," and then they see one of their teammates do it, or

they see their on-call getting better, or they see that positive impact, and then they get excited and they're willing to dive in and do it themselves.

[0:48:02.5] JM: Okay. Just to recap for people. One way, if you want to do Chaos Engineering, is you take an attack container, you deploy an attack container on to your infrastructure and then you have this attack container do something, like launch a network attack or launch a CPU attack or launch a memory attack on some service that you're running, and then if you're using Gremlin, then containers also communicating back to your API. The Gremlin API, that is. So what's the communication between an attack container on my infrastructure and your API? What is your service do? What is it reading. What is it hearing back from the attack container?

[0:48:57.2] KA: Yeah. So the most important thing is that like that signal to halt or undo if things go wrong. So that's always the first thing it checks. It says, “I'm running this experiment. Am I still good to continue?” And as long as that's okay, then it falls into, “Here are maybe the logs from the box. Here's what I've done. Maybe some metrics.”

Actually, we’re very careful about what data we store and what data we send up. We try to be as succinct and as — To store as little as possible. Most of it is around how is this experiment progressing and sharing those logs.

[0:49:30.9] JM: So if somebody's using Gremlin, what they want is like a dashboard where they can see the outcomes of the attack and also like a control plane where they can disable attacks if something is going dangerously wrong or if they have proven that their system is brittle and so on.

[0:49:51.4] KA: Right. Gremlin gives you the dashboard for insight into the control plane, for creating and managing those experiments, for halting them, for seeing how the experiments themselves behave. Then it's coupled best with monitoring tool, whether that's Datadog or New Relic or your in-house one. You’re probably can going to have that dashboard up next to Gremlin. So you're seeing how your system is behaving and how your customers are being impacted while you're running that exercise.

Then the other thing I'd mentioned that's important about the control plane and the API is the ability to automate those experiments. I may have done this last time, but one of my soapboxes for Chaos Engineering, a lot of respect and homage to ChaosMonkey, but chaos doesn't need to be random. People shouldn't start by randomly breaking things. We think thoughtful planned experiments are a much more successful and simple way to start.

So this lifecycle is you run the experiment, you understand how your system behaves, you get it to the place you're happy and then you automate it, and whether that's via schedule or integrated into your continuous deployment pipeline, we give all those tools you know. As a dev first company, we wrote the CLI, and then the API, and then the web interface. So our API is really built to enable that kind of continuous testing over time, and that's how you prevent drifting back into failure, because things are always changing.

[0:51:16.2] JM: I was trying to think of some cases that will be hard to create attacks for, because the things that you've built, like testing, networking, like in a flaky network or drop packets or out of memory issues. These are — It's not trivial to build, for sure, but I can imagine much harder attacks that are still very common, but they're really hard to — Potentially hard. I couldn't think of an effective way.

For example, like cache invalidation. Like if I've got some system that does caching, my cache is going to get stale values sometimes and the data access layer might retrieve stale values and if this happens in your system and you're like, "Why is that happening? What is causing my system to do this?" At the same time, the issues that cause cache staleness might be so specific that — Well, I mean, I guess I'm curious how you think about that. I guess you could just make an attack that replicates cache staleness and you just assume that, "Oh! For some byzantine reason, your cache could end up being stale. Are you prepared for that?" Yeah, how would you think about an issue like cache staleness?

[0:52:35.6] KA: Yeah. Cache invalidation in particular, like does that happen because you're unable to hit the source of truth and update things? So is that essentially a network failure down the line? Is that happening because incorrect data was pushed? It could be earlier in your data pipeline. Somebody pushed — In fact, somebody pushed this wrong data and it percolates

through system. That's very hard to test for and it's very hard to do from a Chaos Engineering perspective.

So we've really thought about the core failures that could happen at the host level that are reproducible, that are fairly generic and that are value for everyone. I think when you get into some of those more specific cases like you're describing, that's where you want to move into like application layer, fault injection, and that's a bit of a trade-off where you're going to write some code. There's going to be some coupling, but you're going to be able to be more precise or prescriptive about a specific failure mode.

[0:53:30.5] JM: All right. what re the next Chaos Engineering attacks that you are planning to build?

[0:53:37.3] KA: Yeah. So we didn't talk about process killer or we didn't talk about time travel.

[0:53:43.2] JM: Oh! That's true. Those are the state attacks.

[0:53:46.0] KA: Then, of course, rebooting a host or container or killing a host or a container. We support those. Time was a fun one to build. If you change time on the box and you used time as something to synchronize your distributed system, you have to handle that. I remember a lot of fun getting that one worked out well.

[0:54:04.0] JM: This is like if you wanted to simulate Y2K.

[0:54:06.6] KA: It's so funny. I feel old, because I keep giving that example to customers. I'm like, "Imagine you could just test what happened with Y2K before it happened. Do you remember all the news, and there's so much excitement and it was like — It was a nonevent." Well, you just go in you're like, "Look. I set my clocks on that box to this time. We're good. Moving on." I'm sure many people did that, but obviously it helps there.

[0:54:30.0] JM: But time can be an issue. Amazon came out with some time service recently, right?

[0:54:35.2] KA: Well, and there's NTP, the network time protocol. In fact, when we built this, we had to build an option to break NTP, because otherwise NTP would just fix the time underneath you. So that was one of the subtleties. Yeah, being able to test leap seconds, being able to prepare for daylight savings, seeing what happens when your certificates expire or invalid. A lot of requests are signed. So what happens when those singings fail? There's a lot of interesting time ones, to be honest.

[0:55:06.6] JM: What was that time that Amazon revealed recently? Did you see that?

[0:55:11.4] KA: I don't have it offhand.

[0:55:13.3] JM: Okay. All right. I just remember seeing something and I was — It was something like an atomic clock service and I was like, “I don't even know why people would need this, but, okay Amazon, I'm sure it's useful for somebody.”

[0:55:24.5] KA: Well, there's this thing called clock skew that you may or may not be familiar with, but every CPU is a little bit different. So there's these microsecond, nanoseconds drifts in their clock time in the way they keep time. All computers essentially are always drifting a little bit, and you need something to unify them. That's what the network time protocols is for, or I imagine Amazon's time services is filling a similar need.

[0:55:50.5] JM: I'm just wondering why you would need Amazon’s thing instead if you have NTP, but maybe that's a question for Amazon.

Anyway. Okay, well this has been great. What's next for the company? I mean, you've gotten to launched. Last time we talked, you were preparing to launch. You were in kind of an alpha stage, I guess. I remember talking to you then, I just thought it must've been crazy to be getting — Like I can imagine, just an anxiety producing situation. Getting to launch with one of these infrastructure companies is so hard, and I was just mentioning before the show, like I can’t imagine what — There's no playbook for building a Chaos Engineering as a service company. I mean, did you learn anything about getting to launch or like shipping a product? Shipping a product is hard.

[0:56:46.3] KA: Yeah. there's so many things. I've have learned a lot about sales this year. I've learned a lot about marketing. I've doubled my team size. I've gone out and found people that are smarter than I and got them on my team. I'm really proud of our team. I think that's one of the keys, is if you have a good team of people you can trust and work with, you can figure it out. It's a little bit like the — What's the dilemma? It's the imposter syndrome. It actually turns out, for a lot of things, if you go out with a good team and you work hard, you can kind of figure out what works and make it happen.

Good advisors, good team members, that's really been helpful for me as I've been learning to be a CEO and learning to run a company, instead of just building a cool project and working on a fun software.

[0:57:37.6] JM: All right, Kolton. Well, it's been a pleasure talking to you, as usual, and I'm sure we'll do it again in future.

[0:57:42.4] KA: Yeah. Thank you so much for me on, Jeff. Always a pleasure.

[0:57:45.0] JM: Okay. All right. Well, I'll talk to you soon, Kolton.

[0:57:47.6] KA: All right.

[END OF INTERVIEW]

[0:57:51.5] JM: Azure Container Service simplifies the deployment, management and operations of Kubernetes. Eliminate the complicated planning and deployment of fully orchestrated containerized applications with Kubernetes. You can quickly provision clusters to be up and running in no time while simplifying your monitoring and cluster management through auto upgrades and a built-in operations console. Avoid being locked into any one vendor or resource. You can continue to work with the tools that you already know, such as Helm and move applications to any Kubernetes deployment.

Integrate with your choice of container registry, including Azure container registry. Also, quickly and efficiently scale to maximize your resource utilization without having to take your

applications off-line. Isolate your application from infrastructure failures and transparently scale the underlying infrastructure to meet growing demands, all while increasing the security, reliability and availability of critical business workloads with Azure.

Check out the Azure Container Service at aka.ms/acs. That's aka.ms/acs, and the link is in the show notes. Thank you to Azure Container Service for being a sponsor of Software Engineering Daily.

[END]