EPISODE 512

[INTRODUCTION]

[0:00:00.3] JM: How do you build resilient failure-tested systems? Redundancy backups and testing are all important, but there's also an increasing trend towards chaos engineering; the technique of inducing controlled failures in order to prove that a system is fault-tolerant in the way that you expect.

In last week's episode with Kollton Andrus, we discussed one way to build chaos engineering as a routine part of testing a distributed system. Kolton discussed his company Gremlin, which injects failures by spinning up a Gremlin container and having that container induce network failures, memory errors and filled-up disks.

In this episode, we explore another insertion point for testing controlled failures. This time from the point of view of LinkedIn. LinkedIn is a social network for working professionals. As LinkedIn has grown, the increased number of services has led to more interdependency between those services.

The more dependencies a given service has, the more partial failure cases there are. It's not to say that there's anything wrong with having a lot of service dependencies, but this is just the way we build modern applications and it suggests that we should try to test the failures that can emerge from all these dependencies.

Bhaskaran Devaraj and Shao Li are engineers at LinkedIn, and they're working on a project called Water Bea, that has the goal of making the infrastructure at LinkedIn more resilient. They particularly want to do this in ways where not all of the service owners have to even be aware that resilience testing is being added. They want to add it in a way that if you're a service owner, you can just easily test that your service is resilient to downstream dependencies. We'll get into what that means.

LinkedIn's backend systems consist of a large distributed application with thousands of microservices. They're communicating with each other over a proxy called Rest.li. Rest.li is this

proxy that standardizes communications between services. You get this at a lot of really big tech companies like Amazon, or Netflix, or Google, where you have this standardized way of communicating between services that assist with routing and AB testing and circuit breaking and other aspects of service-to-service communication.

This proxy can also be used for executing controlled failures, because as services are communicating with each other over this proxy, you can create a controlled failure by simply telling your proxy not to send traffic to downstream services. You can just make a call to a service and if it has the failure testing scenario switch on, then it's just not going to respond. It's going to pretend like it's down without actually shutting down.

This is a pretty ingenious way of implementing chaos engineering. If it sounds confusing, don't worry, we will explain it in more detail. In this episode, Bhaskaran and Shao describe their approach to resilience engineering at LinkedIn, including the engineering projects and the cultural changes that are required to build a resilient software architecture.

[SPONSOR MESSAGE]

[0:03:26.1] 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 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. You're building something that's really cool and very much needed in the world. Thank you OutSystems.

[INTERVIEW]

[0:05:17.8] JM: Bhaskaran Devaraj and Shao Li are engineers at LinkedIn. Guys, welcome to Software Engineering Daily.

[0:05:23.4] SL: Thank you.

[0:05:24.0] BD: Hi, Jeff.

[0:05:25.0] JM: Today we're talking about resilience engineering and how you have implemented some resilience at LinkedIn. First off, let's just talk about that term. How do you define the term resiliency?

[0:05:39.3] BD: I can take that. This is Bhasky. The way we look at resilience is like any software systems that can actually withstand any changes to its environment either downstream or hardware, anything that could fail. Actually, that's going back to its request in a graceful way. That's how we see this against.

To give an example, one of our payment systems it has different gateways to get payment information for people who pay on LinkedIn. If one of our gateway is actually broken, the system automatically allows the request to honor the one, which wasn't the case before. That's example of even if things go down, we actually find an ultimate path to get that transaction done.

[0:06:22.1] JM: Yeah. There are other kinds of patterns that we could get into regarding resilience, things like the circuit breaking pattern. I think we should talk a little bit about culture. Why is company culture important to the idea of resilient systems and how can a company culture encourage resilient systems?

[0:06:43.0] BD: Actually, I've been with LinkedIn for a while. When I say a while, it's like 9 years. I have seen the company get better at what we do in terms of what we offer for our customers. As a result of that, we build really good complex systems. Over time, we started delivering what the customers needed, but do not pay a lot of attention of how resilient some of the systems we build.

At some point, actually after a certain scale we realized, "Oh, we cannot continue in the way we are doing in building software." We realized, "Let's think about how even –" if you have enough complex systems that things will always fail. We can plan for it, but there's only so much you can do. You have to put a break on how we build systems and actually think holistically from in a study standpoint, from a developer standpoint, even from a product standpoint we needed to get together and understand how to build a software system that is more resilient.

It's really talking about culture. When we started this project, there were some – at least it was not real, but it's mostly now our heads – I mean, we started this, it has powered our developers as one when we say like your system is not resilient. Or are you ever thinking we can provide a score of how resilient their systems are and how are they going to respond.

Actually, we were surprised to see people like, "Oh, we didn't know that we could fail with these many failure points. Let's fix that." It was surprising to see how people took this more in a welcoming way than we initially thought.

[0:08:13.7] JM: Let's get into talking about actual engineering, and then we can talk about some of the decisions that were made around making the LinkedIn architecture more resilient. Let's start off with just an overview of the LinkedIn software architecture. Let's say I go to linkedin.com and my browser requests a bunch of information that has to come from linkedin.com. It's going to all these different backend services at LinkedIn. Explain what's happening in more detail. What is the interaction between my desktop web client and LinkedIn?

[0:08:48.5] SL: For end user, it looks very simple. You tap linkedin.com and the whole website will render for you. Actually behind the scene, there's a lot of things happening. First of all, we are drill load balancing the user to the closest data center that we want to serve the user. Once the request getting to our data center for – it also depends on what platform the user is using to make their request.

For example, if you're using the desktop client browser, we're actually going to send a JavaScript version of the web application to the browser first. The browser will load this application and the application itself, the JavaScript applications itself then start to pull the data from the backend to start rendering the actual content of the LinkedIn system.

Obviously, there's a lot of details happening, technical details here including how we route user to different data center or how we –

[0:09:49.0] BD: You can jump in back once you collect your thoughts again, it's okay. What I was going to say is if you're logged in or logged out, your experience we want to optimize that. If you are first time logging in, you will be – you're routed to the closest data center. If you're already logged in, you will have a default data center primary and a secondary data center.

Essentially we want to optimize for speed. Once you hit the data center, you will be – the request goes to frontend, which [inaudible 0:10:16.2] gathers all the information, they create the results and sends it back to the browser in a very asynchronous way.

[0:10:22.2] JM: Okay. That makes sense. Now the restful service interaction at LinkedIn is managed by a library called Rest.li. It's important for us to discuss Rest.li in order to get into some of the resilience engineering solutions that were built by you guys. Explain what Rest.li is.

[0:10:43.8] SL: Okay. Rest.li is actually a little bit more than just a library. It is actually a framework which includes the resource discovery protocol that we call dynamic discovery and request to response protocol we called R2, and also includes the backup compatible schema for request between services and also the load balancing between load balancing algorithm for

requests, so it is a whole protocol for – that includes together the whole Microsoft architecture of the whole thing.

[0:11:17.6] JM: Okay. What are the problems that Rest.li solves?

[0:11:21.3] SL: Obviously, the main problem Rest.li solve is to – how do you – microservices architecture, how do you find which server is serving which kind of resource. When the resource is available or when a new server is online to serve that new type of resource, how do you notify the other consumers of that service, or of that resource of the new server available? Or when the server goes down, how do you notify those consumers as well? These resource discovery and request response is the main problem that Rest.li try to solve.

[0:11:59.6] JM: Is this a standardized layer that exists on all of the different services that are deployed at LinkedIn?

[0:12:08.2] SL: Yes, it is. This is actually one of the main advantage that we have. We can touch on it later for when we build our resilience framework, because we have this very standardized Rest.li framework that almost all LinkedIn service are built based on this. We have a centerpiece that we can put the resilience framework into this, so all these other services will get benefit from it.

[0:12:35.3] JM: This is a pattern that I've seen when interviewing people from places like Twitter, or Facebook, or Google where they have – I've heard it be called a service proxy in many context. In other places, in the Kubernetes world, you have the service proxies and then you might have a service mesh that does some centralized things. Here, we're really talking about this layer that sits on all of the different services that people deploy.

If I'm billing a service at LinkedIn, it does billing and payment. Just processes billing and payments. I don't want to think about routing and load balancing and circuit breaking and metrics and monitoring. These are things that I want bundled into my service, and you might want to do this at the service proxy level. Do you call it a service proxy? Is that the right terminology here?

[0:13:32.0] SL: At LinkedIn, we actually didn't refer this as service proxy, but luckily Rest.li protocol or Rest.li framework at LinkedIn cover all the things that you mentioned above, including the metric, including the load balancing, including how I register my service so other a few hundred service at LinkedIn can easily discover me as a billing service that is available through the LinkedIn data center.

[0:13:58.5] BD: Truly, in-line with what Shao mentioned, this is something that you get for free. If you're a developer, you're coming in both application at LinkedIn, you focus on the business logic and what developer for what you just said, like if you're building a billing system, you just have to focus on building the billing system. The rest of it, you get for free, like everything is part of the package when you do a build on the product, if that makes any sense.

[0:14:21.1] JM: Of course. What are the other kinds of service standardization that I'm going to get out of having a service that I deploy with Rest.li?

[0:14:32.0] SL: Yes. I can take that. I think if we would deploy a service with Rest.li, it is actually the most important benefit that will come with it is the metric part. As a developer, you don't really need to worry about any type of service metric that what is my latency, what is my response error rates and the QPS and all that. Those all come free with the Rest.li framework.

There is a technical stack called filter chain in the Rest.li framework, in the Rest.li client. One of the filter is responsible to emit these metrics to our metrics clocking system. If you build a service within Rest.li framework, it's elaborating Rest.li framework, all of these things are going to come for free. I think that's the main benefit of having to build on that on top of Rest.li.

Prepare your team for disaster by proactively testing failure scenarios. Max out CPU, black hole or slow down network traffic to a dependency, terminate processes and hosts. Each of these shows how your system reacts, allowing you to harden things before a production incident.

Checkout Gremlin and get a free demo by going to gremlin.com/sedaily. That's gremlin.com/sedaily to get your free demo of how Gremlin can help you prepare with resilience as a service.

[INTERVIEW CONTINUED]

[0:16:33.9] JM: What about things that are more complex and subjective? It might vary from service to service. For example, I want to do load balancing and AB testing in different ways on different services. How do I configure things to work differently if I'm using Rest.li? What are the knobs that I can tune and how do I tune them?

[0:16:56.8] SL: That's a great question. I believe LinkedIn, when the service infrastructure team, when they build this Rest.li framework, they already have that in mind. Also this framework has been evolved over the last many years. It actually has a lot of knobs that you can tune. Just give you one example that you mentioned, what if I want to do load balancing differently, just for example I have a very high QPS backend service, it should – we want to – normally, what we do is round-robin the request to different server that provides response.

In this specific case, because of the high QPS and low latency expectation it will benefit a lot from in-memory cache that we can have. That means, it will benefit a lot if we can route the request from the single user, because that will constantly have the same response. If we can route the request of a same user to the same host, that will have a lot of benefit in terms of service response latency.

We can actually configure in this Rest.li framework load balancing algorithm that instead of using round robin, we use member ID based on the member ID, we route to specific host. That is just one example of many, many knobs that we can tune in Rest.li protocol. There is other things like how do we tune a load balancing algorithm, so that this load balancing algorithm can adapt to different type of service profile.

For example, if a service is taking thousands or even tens of thousands of requests per second, you definitely want the load balancing algorithm to act fast in terms of a second, or even millisecond before you lose many user request if one server goes down. If a server is just taking a few QPS then you might have more leaning, or you might have more tolerance in terms of how does the server, how can the server respond, how long does this already need to get back to the request.

[0:19:15.7] JM: As LinkedIn has grown, the increased number of services has led to more interdependency among those different services. This is a common thing that happens at companies. You get dependencies. You get dependencies among services. What are the problems that are created when you have services that are dependent on one another?

[0:19:40.1] SL: There is actually a technical term to describe this. It is called dependency hell. As the name suggests, it is really a nasty situation where you have a handler of service that interdependent on each other. To make it even worse, sometimes they're so killer dependency. They are across [inaudible 0:19:59.9] dependency for one service itself, so there is definitely a lot of problems created by this.

Obviously, because self – all the different pieces are moving, it is really hard to track where the problem is coming from because of all the moving pieces. Also there is the probability of the whole system fail is really high, because if you depend on one-hander service, or one-hander different pieces. Even if those one-hander dependencies are 99.99% available, or reliable, the overall probability of the whole system being available is actually only – is the overall ability of the whole system is actually pretty low. Identify those dependencies and they try to reduce the number of critical dependencies is really important.

[0:20:56.7] JM: This conversation that we're having about Rest.li and services and the sprawl of having tons of services and interdependencies, this is related to the discussion of resilience engineering, because when you have this sprawl of tons of different services and there is dependencies and whatnot, you get a more brittle architecture. It's an inevitable trend, but nonetheless because your architecture becomes more brittle, it's going to have more failures, unless you actively take precautions to make it resilient.

I think this is the motivation behind the project that you two started at LinkedIn called Water Bear. What is Water Bear? What were the design goals of Water Bear?

[0:21:45.8] BD: I can take that as much again too. Essentially about a year and a half ago when the Shao NS team started this coding, the capability of Rest.li in general, like what Shao is mentioning about all these things we have, all these nuts and bolts and gears that we can move within Rest.li, like they're actually untapped. We also to your point, like LinkedIn has grown to a point, like they have all these complex system that we build, like anything that fails would actually impact on stuff we totally did not expect.

We have this problem, we also had this possibility of making an impact with this Rest.li system we already have. Nobody has taken advantage of it and we also notice that this is a common problem, not just with LinkedIn, but in the industry where things get complex and there are too many moving parts as you mentioned.

His teams are tuning some of the downstream time odds essentially trying to get to a point, like if your downstream is not responding within a certain time, like you actually respond back either with an error or a graceful degradation. That's how it started.

We had a little – Shao and I were talking, like how can we do this in more holistic way? How can we enable our developers to actually make this a practice of how they build software? That's when I was watching actually. I discouraged how she'll award this creature goal, targeted at Water Bear. What that creature is all about is how resilient it is, essentially can throw this organism or microorganism in the space.

You can survive that, you can throw it in the wild canal. You can see the panel. We essentially wanted to build a system – or it's a system, in the same time also creating a culture around the system, how we can build resilient system software to be more prepared for different moving pieces and things could actually break.

[0:23:34.1] JM: Indeed. This is a philosophy, or is it a package? Is it a set of software? What exactly are we talking about when we're talking about this Water Bear idea of becoming more resilient?

[0:23:49.1] BD: The actual engineering group that we are building and we have built some of it is called resilient engineering. Water Bear, it started as a concept of how can we build more resilient software and it actually has a bunch of tools that we have built, which I will talk about if he has a chance.

It also includes education of how we can build better software. Essentially, it's a concept and everything that comes over it basically like tools and education and more exposure and evangelization and all that.

[0:24:24.7] JM: Right. I think we could start with the high-level goal of wanting to promote chaos engineering. Chaos engineering is another higher level idea, which is where your application should be resilient to random failures essentially. It's been several years since Netflix started talking about the Simian Army and randomly killing instances, randomly killing data centers and making sure that your overall application health is okay, even when those things happen.

This is another high-level idea, but I think against the type of applications that we would want to build into our infrastructure, if we wanted to be more resilient. Let's talk about that on the application level. If we wanted to simulate application failure on the LinkedIn architecture, if we wanted to simulate things like a data-center dying or a service dying, what would we do?

[0:25:19.5] SL: As I mentioned, there is this beautiful Rest.li framework that we can build on top of it. That's what actually this project started. Basically we call it a disruptor into the Rest.li filter chain. As a result, we can actually accurately fail the request and for one particular request, or for one particular user, or for one particular section of user.

One thing that I want to point out here is we start different from the chaos engineering approach, which is by randomly killing services in production and hope for force the application to be more resilient. We're actually coming from another approach whereas we want to accurately measure and understand the impact if a service is dying, or if a service is not responding properly, how is that issue bubble up to the user?

As I mentioned, at this point if we want to test when we try to render a homepage, let's say, a user go to linkedin.com. Once the user send that request and that request will be served by a few hundred different down streams, and we'll want to test if one of the downstream, how well the user see. Is the user going to see oops page, or 500, or 404? Or is the user going to see a gracefully degraded experience?

The current tools and framework that we'd put in place is able to allow us to accurately test that for that user, or for that single request without impact basically the rest of the world. That is really the power of combining the Rest.li framework and the targeting system that we have at LinkedIn.

[0:27:11.6] JM: This is dramatically different than the approach of killing a service, or killing a data center. This is more failure simulation. If I understand it correctly, you actually do it at the client level. You don't kill any services, you go to a client and you essentially say that the client is going to filter out certain responses from certain services. Is that correct?

[0:27:40.7] SL: That is correct. As I mentioned in Rest.li, there are client part, there is server part. At the start, we start at the client side, because we feel this will accurately simulate the experience. As mentioned, there is this filter chain. We add this disruptor filter at the end of the chain, which is right before the request go out. We basically just intercept that outgoing request to the downstream and respond with a failure. That's for the whole service stack including the metric, including all the monitoring stack. They all view this intercepted request as a normal failure response from the downstream service.

[0:28:23.2] BD: Just to jump in and add a little bit color to what Shao is saying, Jeff is the question about this is different from the chaos engineering. What Shao mentioned is yes, it is different and the approach is the link out, he didn't mention the name, but that's the tool that we're calling. The link out is a tool that we provide the developers to actually build resilient software.

The second step to that, which we haven't got to is chaos engineering, where we actually – will be at some point once we have – once we're building this momentum behind building resilient software and services, we haven't decided when we will start actually doing what you are saying

to go around in production and start killing nodes randomly and see how it responds. I would call this a little bit, like in our conservative approach to how we want to make sure that they're not having any service disruption, or user impact when any of these happens.

[0:29:22.3] JM: Sure. It makes sense. For people who – actually for myself, I should say, what's the difference? What's the difference between killing a server and just ignoring all of the requests from the server? In practice, it seems like those would basically be the same thing.

[0:29:40.0] SL: Yes. I think the most important thing here we're trying to do is to gain confidence that our system is able to handle the situation where the server fail without actually impact the user. As I mentioned, ignore the response from the user, from the downstream server, this approach we're actually able to accurately target for which request, for whose request we want to ignore or we want to simulate this failure.

If we actually kill the server, there is no way we can protect the real user experience from being impact by this. I think long-term, yes that is definitely our goal. The system should be able to handle all these random killing gracefully. Right now to capture there, to build a confidence that we have the ability to do that. We are starting with a more graceful, less interrupted way.

[0:30:35.3] BD: Yeah, so essentially getting back to the question to what device is it depends on how much impact you want to take to test this system. Whether you want to do it for a single user and understand all your downstream dependencies, versus like you actually kill a server and realize they're not just the service, there are 10 other services or a hundred other services got impacted because of a server gone. This goes back to the point, we have built such a complicated system, conflict system that at this point we want to take this more measured approach.

[0:31:05.4] JM: When you look at the architecture of the company holistically and you say, "Okay, we want to make it so that any service can fail." We're talking about the entire service failing and going offline, not just one of the instances. If we want to make it so that any of our services can fail, but the overall application health will be intact, whose responsibility is it to be able to deal with those failures?

Because I could imagine it being the responsibility of like, if I'm the service owner A and a hundred different services depend on service owner A, maybe service owner A should be responsible for figuring out the fail over cases for all of those downstream dependencies. Or maybe all 100 of those people who are requesting from service A should be responsible for the fail over. Who should be responsible for the fail over in the event that the service falls over entirely?

[0:32:05.5] SL: I think that is a really good question. At this point, we're actually one stop before that, which is try to identify and try to create a map for post-service provider and service consumer to understand who is depending on who. Most importantly, for service consumer to actually make assertion that this dependency I have is actually really critical. If this dependency failed, there is no way I can't deliver the value off this product that that is supposed to deliver.

I think after we create this map of – let's say, use your example, if this 100 upstream service consider this one downstream service as critical importance. There is no way I can get this piece of information if this one downstream has failed, then of course this one service they need to provide either they need to really make sure that all is available, they have high availability, or they provide another alternative solution for those 100 services if this one downstream dependency is for some reason not available.

Let's give another example, if 99 of those dependency says, "Okay, yes I depend on this downstream. But if it goes away, it doesn't really impact my ability to deliver my product value to the customers." Another one of those 100 dependency upstream says, "Oh, I have a core dependency. I critically depend on you." Then maybe it is this one upstream, this outlier's responsibility to really think about, "Okay, why am I critical dependent on this downstream, where other 99 service are not considered this way?"

We're trying to by creating this dependency, critical, non-critical dependency map and presented this to the service provider and consumer and provide basically a – provide a form for them to talk with each other and then we can star the conversation from there, who is responsible to it is, when this kind of mismatch of SLA, or mismatch of importance in concept happens.

[0:34:25.2] JM: Are you two doing resilience engineering fulltime at LinkedIn? Is that your title at this point?

[0:34:32.2] BD: No.

[0:34:34.2] JM: Okay. It's just like a side interest group sort of thing?

[0:34:38.1] BD: Like I said, we have realized the potential and the impact we can have in this area. We also recognize that how critical this mindset and tooling around resilience engineering is for LinkedIn overall in the last year. We have a concept called virtual team, where we borrow time from different team members who are really interested in the space. They sort of having exploring options, build a tool and actually it's almost like a venture kind of a bed, like a incubator within the team.

This year they have actually really delivered in terms of the value that we want to see. Next year we're going to fund it to be an actual team. Right now there is an almost zero fulltime resource allocated for it, but we're going to do more next year.

[0:35:31.6] JM: Do you have a platform engineering team that you could be a part of?

[0:35:36.5] BD: We do have a presentation infrastructure, so that will be equivalent, right?

[0:35:41.1] SL: Yes.

[0:35:41.7] BD: Yeah. We do have a team who takes care of the Rest.li and the frameworks on this, but since [inaudible 0:35:48.7] this is where that focus around side up mentality comes in, and that's where we're going to – I mean, we're going to explore different options along the leadership. Right now, the impact the team has created, like we want to continue with the momentum we already gained in this part.

[SPONSOR MESSAGE]

[0:36:10.3] JM: If you are building a product for software engineers, or you are hiring software engineers, Software Engineering Daily is accepting sponsorships for 2018. Send me an e-mail jeff@softwareengineeringdaily.com, if you're interested.

With 23,000 people listening Monday through Friday and the content being fairly selective for a technical listener, Software Engineering Daily is a great way to reach top engineers. I know that the listeners of Software Engineering Daily are great engineers, because I talk to them all the time. I hear from CTOs, CEOs, Directors of engineering who listen to the show regularly. I also hear about many newer, hungry software engineers who are looking to level up quickly and prove themselves.

To find out more about sponsoring the show, you can send me an e-mail or tell your marketing director to send me an e-mail, jeff@softwareengineeringdaily.com. If you're a listener to the show, thank you so much for supporting it through your audienceship. That is quite enough, but if you're interested in taking your support of the show to the next level, then look at sponsoring the show through your company.

Send me an e-mail at jeff@softwareengineeringdaily.com. Thank you.

[INTERVIEW CONTINUED]

[0:37:37.4] JM: As far as implementing chaos engineering, this is something that you guys are working on. You're working on a tool called fire drill, which is going to provide an automated way of triggering these infrastructure failures in production. Can you tell me about architecting fire drill, or how far along it is?

[0:38:00.5] SL: Sure. As Bhasky previously mentioned, a fire drill is the second phase, or second step that we're taking towards the whole resilience approach, linked out as accurately targeting to fail, simulate the failure for particular service. Once we build the confidence, we'd link out that this service can be filled out, or if this service goes away, other service can survive, then we can start using fire drill to actually go into production and start killing the servers that is running on these services.

As a project as a whole, fire drill is still in its early stage. We did have some modules that we already have and we tested doing production that is running. We are focusing, because most of the infrastructure level failure will manifest itself as either a slowdown, or host not available. We start at the network level. We're able to have the ability to slow down network to a host, or turn off the network for a particular host.

We're not planning to massively, to practice this practice fire drill in production at massive scale at this point. We're doing pilot testing with different services that we already gained confident with link out. In the midterm future, this is definitely going to be a goal that's similar to the Netflix approach you mentioned about. We want to build this automated system that trigger infrastructure failure in production and massive scale all day every day and collect data to make sure when these failure really happens, the user experience will not be impacted.

[0:39:51.1] JM: You did also mention the desire to build this service dependency graph. Once all the different engineers in the company see their service dependencies, this is going to be before – I believe before you roll out fire drill and start just killing instances mercilessly, how do you expect people to respond to – if they saw a company-wide graph of their service dependencies, how would you expect the engineers in those different teams to react to that information?

[0:40:26.3] BD: As engineers, I would start with like do you really respond well to data? That's one fairly subjective, because I would see also am I dependent on a 100 downstreams? That's not good. Definitely, that's a hesitation I mentioned before, like we had before showing this data, because we did map out some of our dependency graphs, or some of these really complex systems. We wondered how we can disclose this information.

Surprisingly enough, at least so far, I cannot say for other companies, but within LinkedIn I've seen really good positive reaction. Like, "Oh, I didn't know that I had so many dependencies. Can you provide me these toolings to identify my core dependencies was a non-core? How would I respond? What would I make sure of the user?" Essentially, it came it was a feedback to how we can build these tools better. That's one.

Second thing is again, I can speak from a LinkedIn's perspective that people definitely act like owners. When it comes to their services not being able to respond to failures and they hear about this initiative, they take ownership and definitely put that in their old map and discuss with **[0:41:29.6]** like how can we partner together to make this happen on LinkedIn? Honestly, we have not seen any really bad reactions so far. Knock on wood.

[0:41:39.2] JM: We're having this conversation near the end of 2017, but it's going to air in 2018. I'm actually going to air this after about – we're going to do a couple weeks of Kubernetes shows. Makes me curious, what's the deployable unit at LinkedIn? Are you deploying services in containers, or in VMs? Maybe you could just – I know we're up against time, but you could give an outline for the infrastructure at LinkedIn.

[0:42:07.4] SL: LinkedIn is definitely on the path to content analyzation and of as deployable selling key, we have – as I mentioned, we have thousands of we call [inaudible 0:42:17.3] product as deployables. They will be deployed into containers in production system.

[0:42:23.9] BD: Yeah, the frontend is play amber. The backends are JD nodes, if you're more interested in those kind of technology. From up your like content analyzation was as not – we are actually in a transition right now.

[0:42:36.6] JM: Are either of you involved in that transition?

[0:42:39.0] BD: We both are. Yes.

[0:42:41.1] JM: Okay. Can you give me a little – some tidbits about it? I mean, I'd love to know, because since we talked about Rest.li, like Rest.li is a service level module that's providing load balancing and routing and whatnot. Imagine some of the ways that you would have to build Rest.li, or some of the best practice around Rest.li would change if you are moving to fully containerized architecture?

[0:43:06.5] BD: From what I could say, that is not a real impact on how Rest.li has functioned, because of this now. I'm not sure how much of this is public yet. We call it LinkedIn platform

service. We are very confident that it's going to work, but I'm not sure if there's any block **[0:43:23.3]** that I can talk publicly about it.

[0:43:25.8] JM: No problem. Okay, well then I guess just to wrap up, when you're thinking about the cultural changes – we touched on this a little bit earlier, but this was really emphasized in your blog post. What are the kinds of cultural changes that you're looking to get out of the Water Bear movement? I think one of the things that you mentioned was just having a culture where people are checking on graceful degradation. They're making sure their services have graceful degradation. How do you encourage or incentivize graceful failure and graceful degradation?

[0:44:01.8] BD: My vision, I'm sure Shao shares with this and other leadership folks as well, is to get to a state where people are actually proud of building a really resilient system. Almost we're talking about, oh yeah, is your product Water Bear certified? Why don't I take this from a concept are good to have to. This is basically built into how you build systems, almost like how you can check exceptions and almost how you do logging, these best practices of how to build good software, good craftsmanship. Today it's not. It's been received very well by the development counterpart. We want to get this from here to making this what we do normal.

[0:44:41.5] SL: Just to add a little bit on that, I think right now we deal with different tools, different framework in trying to change some of the process along with developments whole life cycle including how they test their code, or how they – do they need to manually think about or test their resilience.

Long-term I think the culture would really envision is developers should not really need to think about it as Bhasky mentioned. We should provide it as how you do logging. It's just there. Developers should just – if they are service is not able to graceful degradates if one of the dependency is expected to be – if one of the service is down, dependency is down and their services are expected to be graceful degradated and they're not. Make sure just get exception and they should just flood the system that deployment – the testing framework should block their deployment, or reject their commit.

End of the day, we want this just to build – we want the Water Bear to be holistic framework that's integrated as a whole into a whole LinkedIn ecosystem, so developer don't need to think too much about it. It's just there.

[0:45:57.6] JM: As hard as it is to deal with random failures in upstream systems and downstream systems, I think perhaps the worst kinds of failure to deal with are the partial failures. This is like when a service is half responding, or it's trying to restart while it's also responding to some requests, have you thought at all about in terms of implementing a chaos engineering system, how do you simulate partial failures?

[0:46:24.3] SL: That's a really good question. Again, I think we already considered – you are right. When the host is just die, it's actually not that bad for resilience consideration. For example, we had situations where downstream service will not die. It will just accept other connections and hold all the connections without responding and eventually cause all exhaustive – all the far handlers for old option request. That is actually causing a lot of problems.

Also there are early data in the pipeline problem when we kafka as our main data pipeline to send messages as messaging system. There are times when a message that is not compatible with schema in the system and it caused a whole lot of downstream service to just by processing the falling messages. These are really good problems to have, and as resilience engineering team, we are going to attack these problems and we have ideas and that we will be implementing slowly to solve these problems.

[0:47:33.3] BD: Yeah, I wanted to mention exactly what Shao has been saying, which is we have noticed that problem and you're totally right about it. It's not one of those easy things. This is 01. It's more about like it's in the gray area of that's working or not. That has been one of our biggest pain points to identify a system, either it's down and reroute or not.

We are right now collecting data around how to identify those systems on the client level and make corrective actions, but we are not – at this point I can confidently say that we can absolutely identify those and absolutely take corrective measures. We are in progress.

[0:48:11.6] JM: All right. Well, Bhasky and Shao thank you for coming on Software Engineering Daily. It's been great talking to you. It was great to read your blog post, which I'll put in the show notes. It's useful to anybody who's building chaos engineering, or just trying to build a more resilient organization. Thanks for coming on guys.

[0:48:29.5] BD: Thanks, Jeff. Fantastic questions.

[0:48:31.5] SL: Thank you.

[END OF INTERVIEW]

[0:48:35.1] 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-in to any one vendor or resource. You can continue to work with the tools that you already know, so just 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 offline. 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. The link is in the show notes. Thank you to Azure Container Service for being a sponsor of Software Engineering Daily.

[END]