EPISODE 819

[INTRODUCTION]

[00:00:00] JM: Twilio is a communications infrastructure company with thousands of internal services and thousands of requests per second. Each request generates logs, metrics and distributed traces, which can be used to troubleshoot failures and improve the latency.

Since Twilio is used for two-factor authentication and text message relaying, Twilio is critical infrastructure for most of the applications that use it. The service must remain highly available even in times of peak application traffic or outages at a particular cloud provider.

When he was at Twilio, James Burns worked on platform infrastructure and observability. James was at Twilio from 2014 to 2017, a time in which the company experienced rapid scalability. His work encompassed site reliability, monitoring, cost management and incident response.

He also led chaos engineering exercises called Game Days, in which the company deliberately caused infrastructure to fail in order to ensure the reliability of failover systems and to discover problematic dependencies.

James joins the show to talk about his time at Twilio and his perspectives on how to instrument and observe complex applications. Full disclosure, James now works at LightStep, which is a sponsor of Software Engineering Daily.

[SPONSOR MESSAGE]

[00:01:35] JM: When I talk to web developers about building and deploying websites, I keep hearing excitement about Netlify. Netlify is a modern way to build and manage fast modern websites that run without the need for addressable web servers. Netlify is serverless. Netlify lets you deploy sites directly from git to a worldwide application delivery network for the fastest possible performance.

Netlify's built-in continuous deployment automatically builds and deploys your site, or your application, whenever you push to your git repository. You can even attach deploy previews to your poll requests and turn each branch into its own staging site. Use modern frontend tools and site generators, like React, and Gatsby, or Vue, and Nuxt.

For the backend, Netlify can automatically deploy AWS Lambda functions right alongside the rest of your code. Simply set up a folder and drop in your functions. Everything else is automatic, and there's so much more. There's automatic forms, identity management and tools to manage and transform large images and media.

Go to Netlify.com/sedaily to learn more about Netlify and support Software Engineering Daily. It's a great way to deploy your newest application, or an old application. So go to netlify.com/sedaily and see what the Netlify team is building. Also, you can check out our episode that we did with the Netlify CEO and founder; Matt Billman. That was a really enjoyable episode. I'm happy to have Netlify as a supporter of Software Engineering Daily.

[INTERVIEW]

[00:03:29] JM: James Burns, welcome to Software Engineering Daily.

[00:03:31] JB: Hi! Thanks, Jeff. Glad to be here.

[00:03:33] JM: You were an engineer at Twilio for three years when the company was going through significant scaling. Tell me about your time scaling Twilio.

[00:03:42] JB: It was an interesting process. When I first joined, there were only three people that were on call for all Twilio. So that was at that point still about 500 services. So the on call process particularly was [inaudible 00:03:56], and what would happen would be you'd be paged for a system that you may have seen before or not, probably once or twice a day. Then being on call once every four, five weeks.

So I learned a lot through that process about how all the systems work and how in general a large scale cloud provider worked, but I learned it the hard way, I think.

[00:04:23] JM: How did Twilio get to a point where there were three engineers doing the on call duties for 500 services?

[00:04:29] JB: So it had grown out of originally there being three-based teams. So each one of those teams had to offer a tribute for every week to operate that team set of services, but also to cooperatively operate all the different services. So that was where they'd grown out, where they've grown to to that point. About a year after that through a lot of work on the platform team and through a lot of tooling and through also a lot of people work we were able to move to a team-based on call. With that team-based on call, teams would operate their own services, which still required some work to get that balanced out, because the team I was on in particular had – We handled 45 different services. Most of them that were core critical that we were then on call with about six people for.

It was mostly an outgrowth of a lot of early choices to build things like commutable infrastructure, but then just having gone through that growth process, building things that people could easily rollback, have invisibility, and then having a really strong on call culture, where if you are paged, you are there within 15 minutes, period, and that was the expectation. That was what was done. So you could get people on who, if you didn't know the answer, you could find somebody who would show up who would know the answer and get things done when they need to get done.

[00:05:55] JM: Twilio's growth was around the same time – If I have the timeline right in my head, about the same time that Uber was really taking off. And I done some coverage of Uber and them scaling their massive numbers of services and trying to maintain high-availability for very sensitive transactions, and it's really hard. This was kind of a period of growing pains for basically the cloud providers and all of the companies that were building or trying to build very reliable services on the cloud providers. I do feel like things have gotten better. If you wanted to build Twilio or Uber today, it would be so much easier. To what extent do you think that's true?

[00:06:37] JB: I think that's mostly true. I think a lot of it has happened because many of the frameworks that people used have become open source. So at that point, the only real example out there was Amazon itself. So people were trying to follow their patterns, and then the patterns

that Netflix had pioneered on top of Amazon. A lot of people are not Netflix or Amazon. They're not at that scale even if they aspire to be at that scale. So some of those patterns ended up being relatively heavy for what people need to do, and they were having to do it themselves. They're having to try and understand not only what the company was doing, but what the underlying principles were. What the tradeoffs were?

Now I think it's a lot clearer. There's been a whole bunch more published, like the Google SRE book, for example, and then there's instantiations of this as open source, like Kubernetes, like all the different distributed tracing products, like Jaeger and Zipkin, that give you not only the tools to do it better, but also the context, the understanding of the tradeoffs for building these large scale systems better.

[00:07:47] JM: So the infrastructure at Twilio, it obviously changed while you were there. So you saw some of these trends modifying the industry from a firsthand perspective. How did the infrastructure at Twilio changed while you were there?

[00:08:02] JB: I think that one of the biggest changes was to actually invest in platform explicitly. There's a process and a thought, a general approach, that the idea is that people – Everybody can be an expert at everything, like the really hardcore devops kind of approach, where people are going to be able to manage their services all the way from the kernel, all the way up. That ends up being pretty tough to scale.

So one of the changes was to invest a lot more heavily in a platform, and then have that platform team build out the tools necessary to do things like scaling, canaries, different kinds of CICD architectures and let the application developers be more focused on building the applications. They were still responsible for operating them, but they were operating them in the context of a set of tools that we were building specifically for their use cases. I think that made a big difference.

There were also some changes that were made to the build process and to the way that we deployed software, and some of those were still in flight when I left. I think that that's what really even increased the velocity more than where it was.

[00:09:17] JM: The platform engineering role is an emergent role. Obviously, it was adapted many years ago by certain companies. But it really has become more of a pattern, and a platform engineering team will do things, as you said, like they'll figure out, "Here's the continuous delivery model that we're roughly going to follow throughout the company. Here's the logging patterns we're going to follow roughly throughout the company, the distributed tracing patterns, the, perhaps, service mesh patterns." What do you think should be the responsibilities of a platform engineering team?

[00:09:51] JB: I mean, I think that that's a decent list. It's also, in addition to those, so metrics, login, distributed tracing, service discovery and software deployment, including common software deployment patterns, like auto-scaling, canary deploys, canary analysis probably would also fall under that. I think it's also that they aggressively reduce the cost to follow those patterns, because one of the things, one of the hard lessons I learned on the platform team is that migrations are expensive. They are amazingly expensive, and the more you can optimize for migrations being cheap, the more likely they're going to take less than months of time, or years of time. There were a couple of migrations that did end up taking years of my time there, and I tried to learn from that experience.

[00:10:47] JM: What's an example of a migration?

[00:10:49] JB: So, one, migrations was from one major operating system version to another. Another migration was from EC2-Classic to VPC.

[00:10:58] JM: Let's take that second one. Why was that hard?

[00:11:00] JB: Well, at first there was no networking between classic and VPC. So people had come up with all kinds of different there. There are a bunch of articles out where they were like creating IPsec VPNs.

[00:11:12] JM: VPC, by the way, virtual private cloud. EC2 is just the raw EC2 instances?

[00:11:17] JB: Right, and EC2 had a single address space for all instances everywhere, well, within that region. So you needed to – In order to migrate traffic live and not just have a big

bang cutover, which nobody would be willing to accept the risk of that. You need to create a way for your services on both sides of that divide to communicate.

[00:11:38] JM: Just as a preface, why would you want to migrate from EC2 to VPC?

[00:11:41] JB: EC2-Classic was being decommissioned. All the new features were in VPC. The security model for VPC was a lot better. There were certain mistakes that you can make in configuring EC2-Classic that would let random other people in the region connect to your instance, which is not something you generally want to do. The networking, the reliability, the sorts of things you could do in VPC were significantly better than what was in EC2-Classic.

[00:12:10] JM: Okay, sorry. Sorry to interrupt you. I just wanted to give a preface to people who are a little unfamiliar with these terms, but please tell me more about the migration.

[00:12:16] JB: So, originally, there was this idea that, "Oh! We're going to just follow the other patterns that were out there," which was to create, encrypt the networks between all your machines everywhere. I'd have to look up which article it was, but one of the major cloud companies had done this or applications on the cloud.

We weren't quite ready to do that. So, eventually, this thing called ClassicLink came along that allowed you to connect classic instances to VPC instances. But even with that, there was a big question about, "Well, so how do we do this?"

So, originally, there was this large plan of, "Here's how we're going to do it." You would know the Gantt Chart, the Waterfall Migration, for moving over, and it's like, "Well, this service is going to move first. Then this service is going to move next. Then this service." and it was built on top of like our understanding of the dependency diagram, which of course was pretty dynamic as most things are in a large service-oriented architecture microservices.

What I ended up having already gone through the OS migration, and there was another one in between there. I raised the flag, I'm like, "This is a big thing that we're asking people to do," and we're asking them to do it on a very specific schedule, and the risk that any kind of problem is going to delay everyone was pretty high. So at that point, I started advocating pretty strongly to

figure out how we could absolutely, in every way, minimize the amount of effort it took people to do this migration. So what we ended up doing was we ended up changing the way we did service discovery so that you could control whether your traffic was going just to classic instances, just to VPC or both. Then we made it so that using that tool and making the default be to go to both that people could launch their services and VPC and launch those services in EC2 and have it just work from the perspective of their consumers and from the perspective of people that they were consuming as downstream services.

So that process made it so that people could move when they are ready. So it wasn't something that had to be on sprint schedules for the next six months. It was, "Hey, you've got a couple hours. Go try and launch some stuff and VPC. Hey! Does it all work? Well, then you migrate it. Good. We're done," and that made a big difference.

[00:14:43] JM: There you're showing a lot of subtle empathy for the developers throughout the organization. Do you have any other tips for developing – This seems pretty important for a platform engineering team, because the platform engineering team is trying to create sane defaults for everybody across the organization. But if you don't have a good sense of what is going on across the organization, you can't craft those same defaults in a truly, I guess, democratic fashion.

[00:15:11] JB: That is one of the key hard parts, is you are asking them to give up some set of control that they might already have, they might want to have. So you need to really understand what their use cases are, and especially use cases that you – In some case, might not want them to do. You need to understand what they're doing and why they're doing it and what you can do to help them out. That will make a big difference in how they look at the platform team either as an ally or a blocker to them doing their real work and application.

[00:15:48] JM: The word observability started getting used in the public engineering lexicon I guess around the time when you were at Twilio. So we've had these things, like monitoring, and logging for a pretty long time. Why is there this newer term observability? Does that that represent anything significant other than new marketing materials?

[00:16:11] JB: I think there's been an evolution. Originally, a lot of the checks, our checklist-based tests from outside, like the Nagios approach. I think there's been a realization that applications are their own spaces of complexity, which is our way of saying it. But like the inside of an application has a whole bunch of different sorts of state, different ways that it is doing whatever that application does. In order to effectively manage that application, when it's in production, when it's not just sitting on your laptop and you can connect a debugger to it, though I don't think very many people do that anymore either. It needs to be observable. It needs to expose some of that state in a way that developers can understand it and other systems can understand it so that they can interact with it effectively.

[00:17:04] JM: So describe some of the different types of data that you would want your observability tools to capture.

[00:17:10] JB: I mean, your application should be exposing, at the very least, the units of work that that's doing. Ideally, in some kind of business relevant sense, like messages handled, or messages forwarded, or messages created. These kinds of basic metrics that let you know that it's actually still doing the work that you asked it to do.

Then there's other things, like distributed tracing, where it puts that work in context about the work in the system. So that allows you to put together more of this entire story or narrative about, in the end, the customer experience. Then you have things like logs, which give you some kind of forensics use. But one of the things I found is if you're looking for logs, if you're looking at logs in real-time, then you've probably missed something else in your observability checklist that would give you something that's usually more structured to understand what your application is doing.

[00:18:10] JM: In many cases, the platform engineering team is going to be the ones responsible for defining the observability tools, defining the observability workflows, understanding what the "line engineers" are going to be facing in their on call experiences, their debugging experiences, as well as their performance improvement experiences.

So what's the workflow that you want to present to the different engineers? If you're a platform engineer, you're sort of selecting these different tools or these different vendors and you want to

be able to empathize with the average developer. How can you, I guess, reduce the work to the simplest integration process or the simplest onboarding process? I guess, more generally, how should you select the tools that are going to be in the observability toolkit?

[00:19:07] JB: I think there are two main things to look at. There's custom integration, which you want to drive down to zero, or as close to zero as possible. I think that one line is like acceptable, and there's this exponential curve beyond one line of integration that reduces the change that someone's actually going to use your stuff.

Then there's accuracy. If you ask people to use tools that don't provide accurate understandings or summaries or whatever else on top of the data, then people are going to lose trust in those tools and they won't use them. So I would look for how can we solve this particular problem. So that goes back to an approach that I like to use, which is asking what questions do you need to be able to answer and how do you answer those questions in a particular tool.

So you end up finding when you want to ask a question, there're going to be tools that are better and worse for doing that, but you have to understand the kinds of questions that people want to ask, and then you look for tools that can answer those questions. Then you look at how to make those tools most easily available to the most people by driving down the integration cost.

[SPONSOR MESSAGE]

[00:20:35] JM: On Software Engineering Daily, we've had several shows about the future of technology education. We believe that boot camps are an efficient, cost-effective way to become trained for the tech industry whether you want to be a programmer, a data scientist, or a designer.

Flatiron School can teach you the skills you need to build a career that you will love. Flatiron School has immersive programming courses on JavaScript and Ruby, everything you need to become a full stack developer. If you're interested in becoming a data scientist, Flatiron School has courses on Python, SQL and machine learning.

You can learn in-person or online and you can find everything you need to get started by going to flatironschool.com/sedaily. Flatiron School has options to save money on the program, such as gender diversity scholarships and income share agreements. Flatiron School also helps the students who graduate find a job. Every graduate is paired with a dedicated career coach so that they can find a job or their money back.

The complete details are at flatironschool.com/terms. Flatiron School is a cost-effective way to start working in a tech industry. Learn more at flatironschool.com/sedaily.

[INTERVIEW CONTINUED]

[00:22:04] JM: One thing about that integration cost – So when I was an engineer at Amazon, my brief time at Amazon, one thing I remember is when I was getting my service up and running, the thing that I was working on. I magically got all the stuff out of the box. I magically, "Oh! Hey! The thing is logging," and "Whoa! I've got monitoring and these things." It was total magic to me when I was working there, because I was just working on some Java backend service and it wasn't like infrastructure. So I didn't really know how the infrastructure within Amazon worked.

It was only later that when I started doing some shows about the service mesh or the service proxy area that I realized, like at a highly performant organization, you often have this kind of thing that just gets like bundled into your application auto-magically. Thanks to the platform engineering team that is doing all these stuff for you.

So when you're the platform engineering and you're architecting this engineering experience for the other engineers that are building like the services that actually make up the business logic of the company, what are your selection of kind of deployment models or integration models? I see the service mesh sidecar approach is one. Perhaps you can also do like the way the finagle started out, where you have this – Like a Java library that you're importing and it auto-magically gives you all these things. What are the different modalities of deploying these observability tools to your service engineers?

[00:23:34] JB: That's a great question. I think the way that's evolved is really interesting. So I most familiar with this from the perspective of watching how Netflix has gone through this evolution in what they've shared externally. But, originally, there was – If you've got a single language especially, you just have a library and you package that library everywhere and it goes out an all your stuff works the right way, because everyone uses the same library.

But one of the interesting things that happens with that kind of what ends up being pretty tightly coupling is that there're some services that just don't get touched that often mostly because they just do their job. So when it comes up and they're like two weeks behind the current library, or four weeks, or six months, or two years, it starts becoming harder and harder to get changed out, because you have this tight coupling between the application and all the platform tools that you're building for that application. So that really limits the platform's ability to evolve.

So what people started looking at were things that were less tightly coupled, but like provided plugins for this kind of things. So people looked at GRPC, and GRPC is interesting and that it provides this full suite integrated in-application, but not quite as tightly coupled. But at the same time, it's still the same sort of problem where if you have major virtual changes or you want to make some change and all of that isn't driven on something in runtime, which is can be, but is relatively hard, then you still end up with something that's fairly tightly coupled across a whole bunch of systems.

So you end up with service mesh, which is a way to try and push a lot of the platform resilience logic outside of the application and have the application just be responsible for business logic. So have the service mesh handle retries, circuit breaking, all these different sorts of patterns to try and make it more resilient by default.

Then with that, you can – Especially with the sidecar model, you can re-launch those independent of the application even, but evolve that much more quickly as supposed to something that's either compiled in to the application or directly adjacent to it. That seems to have been the real success of service mesh, is to provide that additional agility and being able to push out platform improvements or improvements in resilience in general.

[00:26:15] JM: And that application model makes a lot of sense to me. Are there any downsides of that application model? We did a show with a company called Netify, that I think came out of Netflix, and there's this open source project called RSocket. Now they described – Well, the guy I interviewed, he described some issues that they encountered with this service mesh approach that led to a degradation in performance. Perhaps, this was very Netflix specific. But maybe I'm just wondering, because Twilio bear some similarities to Netflix.

Have you seen any deficiencies in the service mesh container sidecar model?

[00:26:59] JB: Yeah, definitely. One of my usual stories on distributed tracing is this. So what I saw in working through putting distributed tracing into the applications at Twilio was that the edge gateway for the API was made all kinds of request the authentication, because that's part of the core functionality of request needs to be authenticated. They were saying, "Look at this authentication service. It's just so slow. Look. It has a P99 of like three seconds, and it's just not reliable." So I knew some people on the authentication service team, and so I talked to them and they're like, "We're looking at our service and we were responding within 500 milliseconds no matter what."

So there became this question, "Well, what's really going on?" So I ended up putting together really a hacky way of synthesizing spans from HAProxy, which is what we're using for our service mesh. What I ended up seeing was that they're both right. The application on local hosts was experiencing delays up to three seconds getting to the local service mesh sidecar.

Once it got to the sidecar, it would be dispatched and that more or less matched up with what the application was seeing. But it's certainly possible for there to be issues on local host and that having to do with scheduling of threads, or networking, or whatever else. But that becomes invisible unless you have your service mesh instrumented in a way that you can compare those timeframes usually with distributed tracing.

[00:28:31] JM: Now, if I'm at an average company. Probably, those downsides of service mesh, maybe it's like latency issues, basically. From what I heard, it's kind of a performance issue. You can develop some performance issues because of this sidecar model. But I think for most

companies, probably that performance issue doesn't really matter, and like the sidecar model is going to be a pretty good fit. Would you agree with that?

[00:28:57] JB: I think it gets really tricky without distributed tracing, to be honest, because I think that you end up not being able to see where your connection actually went. So it becomes hard to correlate failures with endpoints, because you're connecting to the local host. So if that fails, if the request fails, you have to say, "Well, why did it fail?" Then like you're going through your service mesh logs, maybe.

What you really need is that kind of connection to where it ended up, and the easiest way to do that is with distributed tracing. It's really hard to do an aggregate with logs or metrics either. Metrics are interesting to see like failure rate or something like that, but to say, "Hey, this particular request, where did it go? Why did it go there and why did it fail?" allows you to do things about failing instances.

Another one of my common rants is that nobody's health checks actually do health checks. That's been true most places I've been, because the service mesh model requires some way of being aware of how healthy the downstreams are. I mean, their load balancers, their first proxies, whatever you want to call them. If they don't have an accurate view of health, then they will send a request to fail in applications, and you'll wonder, "Why is it all failing?" It's like, "Oh, look! They're sending request to an application that we know is unhealthy," but the health check didn't actually reflect the health.

So I think that that's one of the things where you still – Even if you're not really concerned with the performance, you still need a way to connect the dots between all the different things the request is going through.

[00:30:41] JM: Okay. We should take a step back here, because now we're talking about tracing, and you work at LightStep now. So of all the places you could have gone to after leaving Twilio, after working at Twilio for three years, why was the problem of distributed tracing interesting enough or important enough to change companies to LightStep?

[00:31:02] JB: Well, there's actually a company in between, Stitch Fix. So I did the distributed tracing or started that migration at Twilio, and then was looking for what to do next. So I moved to Stitch Fix, which was different area, different concentration, different position, but pretty quickly got involved in their migration to have a standard observability approach, and saw that actually be significantly faster, mostly because it did end up through a fair amount of work being a single line integration. But I saw what a difference it could make to have pervasive distributed tracing and to – In both cases, both at Twilio and at Stitch Fix, I spent a lot of time working with application engineers, helping them understand the different perspectives that were being presented.

Because it's usually not just one thing. It's like, "Well, the application sees it this way. The service mesh sees it this way. The downstream application sees it this way," and getting people to think in that kind of perspective's mindset within the context of a trace and also within metrics as well. But having gone through that process twice, I got pretty motivated to try and make that experience better for people in general. I feel like a lot of application developers especially, or people who would call themselves application developers, they experience a lot of pain. It usually is that they don't understand that they need these tools until they're experiencing so much pain that's intolerable. I thought there's an opportunity by talking with people more widely by making this my job to help them understand earlier what the situation is, that even if you're building a monolith, it's probably distributed system, because you're probably deploying it into a cloud provider. At the very least, once you integrate a SaaS provider into your monolith, then it's also definitely distributed system.

To get people to that truth earlier, and then to show them that that doesn't have to mean that they have no idea what's going on, that there are these tools out there that help them understand what's actually happening that give them this perspective, that give them accurate information about what their codes actually doing. Because one of the other patterns I've seen is that there's this believe that because the code should be doing something, it is doing something. It's an interesting process to see that believe be challenged and people to realize that, "No, really. Even though I think the cloud is doing this, I need to instrument it in such a way that I can understand what it's actually doing."

So I was excited about the opportunity to do that work with LightStep and to spread this kind of message that you don't have to be in pain, that there are tools that can make it so that you can understand what your service is doing and that you can then build it for resilience. That was part of the whole story with also the chaos engineering, that by choosing to embrace failure on your terms, on your timeline, you can gain that kind of hand's on understanding about what your service looks like when it's not doing well. What your code actually does when things aren't working right? I've seen it transform the way that people do development. I've seen it give people that kind of basic confidence to be able to operate their service at scale, in production, and just not have to experience pain. There may be a little bit of pressure when something is going wrong, but they know they have the confidence, that they have the tools to figure it out.

[00:34:55] JM: We did a show with Ben Sigelman, who is the founder of LightStep, and he originally created distributed tracing at Google, or at least to some extent, he wrote a whitepaper, I believe, about it. What kind of blew me away about that conversation was just the depth of engineering problems that are inherent in distributed tracing. So like on the actual user side, you have to, first of all, decide at what rate am I sampling? How frequently am I sampling all of my application requests to see if there's some kind of problem or to actually get a reasonable view. How frequently do I need to sample it in order to get a true, honest view for how this thing is performing?

Then on the actual distributed tracing library side of things, you have, "How do I write a performant distributed tracing system that does not interfere with the actual services that I'm running?" Because if I'm sampling every single – Basically, every single hop, every single network hop, if that was implemented poorly, that's really going to degrade performance.

Then you have all of these traces that are created and you have to decide, "How am I going to store these things? When am I going to do garbage collection? When am I going to do compaction? How available do I want to make them for people to retrieve and see?" So there's really like a cornucopia of different engineering problems inherent in the distributed tracing space.

[00:36:31] JB: Yeah, definitely. That is one of the things particularly about LightStep, but has me excited not just about where we are right now, but where we will be. Because I think that the

satellite architecture that LightStep uses where we collect everything, or almost everything, everything that you can afford to send over the wire locally, which is going to be pretty much everything. Perhaps terabytes, perhaps petabytes a day in a structured fashion, and then put that in the satellites, which have a certain amount of recall and be able to ask questions with essentially for knowledge. Because you can take the information that comes in later and use that to re-understand the past so you can have a better than 20-20 hindsight, but at the data level, at the algorithm level, that allows you to make the right choices, to surface the right data as quickly as possible then to put that in front of engineers as a complete context.

So that was one of the things that was really interesting at Twilio using LightStep, is we drove a lot of chat ops around this, where it would notify Slack when there's an issue and it would give you your exemplar traces, and you could click on one and you could look at it and you'd be like, "Oh, look! It's broken in this particular way." That's evolved even further now here with the correlation's tool that I think is just released officially recently. Where it will do that kind of analysis in your real-time where you can say, "Hey, look. These two tags are correlated in this particular way," which will tell you things like, "All the request going to this AZ are failing. Maybe you should take all the stuff that isn't that availability zone offload balancer, and then it won't be filling anymore."

That's like just scraping the surface of what's possible with this kind of observability platform, where you have the ability to collect all these data because memory is cheap right now, and getting cheaper. So you can have a lot of data for a relatively short amount of time, but enough time for you to use the future to influence what you look at.

[00:38:40] JM: Can you describe this in more detail? Why is the – I mean, I think of it kind of is a garbage collection policy engineering problem. Why is the retention policy and storage policy of distributed tracer, why is that a deep engineering problem?

[00:38:55] JB: I probably don't know the very bottom basics of it, but one of the purely straightforward and easy economic issues is that you cannot afford to send all these data outside of your cloud in its raw form, and it's even a problem with login if you're doing that. The amount of data that's created, terabytes, petabytes, egressing that data out to some kind of processing in a SaaS of whatever else, and even if that SaaS is directly connected via VPC

peering or whatever else, it's just cost prohibitive. So you need to be able to keep that data local.

[00:39:41] JM: So it's the networking cost. That's the limiting reagent?

[00:39:45] JB: It is for a whole bunch of things. Yes. That it would be too expensive to move that data somewhere on the internet to process it, and it's relatively expensive to collect all that data at a high-resolution locally, but to persist it would be also, from a storage cost perspective, pretty prohibitive. So having this sort of in-memory pool that can be terabytes large, fairly easily, that the cost of memory, large instances, has really dropped significantly.

Then from that full view of the world, be able to construct what is meaningful or interesting and then just save that. So I think we shoot for at least the 1% reduction. But I think it ends up, in some cases, especially with heavy use, being significantly more than that. You don't want to afford to store petabytes an hour or data that you might not be interested in, because you don't need perfect recall forever. It would just be a bad set of tradeoffs.

So you want perfect recall for some short amount of time, and then you want to choose what you want to recall after that time. But the other part of it is that you want to be able to use the information you get later to help you understand which of those things you actually care about. So being able to play all the way within that recall window to be able to understand how change creates meaning. If there's a significant performance change, that changes the meaning of previous data, because it ends up being more interesting.

[SPONSOR MESSAGE]

[00:41:26] JM: Deploying to the cloud should be simple. You shouldn't feel locked-in and your cloud provider should offer you customer support 24 hours a day, seven days a week, because might be up in the middle of the night trying to figure out why your application is having errors, and your cloud provider's support team should be there to help you.

Linode is a simple, efficient cloud provider with excellent customer support, and today you can get $20 in free credit by going to linode.com/sedaily and signing up with code SEDaily 2019.

Linode has been offering hosting for 16 years, and the roots of the company are in its name. Linode gives you Linux nodes at an affordable price with security, high-availability and customer service.

You can get $20 in free credit by going to linode.com/sedaily, signing up with code SEDaily 2019, and get your application deployed to Linode. Linode makes it easy to deploy and scale those applications with high uptime. You've got features like backups and node balancers to give you additional tooling when you need it. Of course, you can get free credits of $20 by going to linode.com/sedaily and entering code SEDaily 2019.

Thanks for supporting Software Engineering Daily, and thanks to Linode.

[INTERVIEW CONTINUED]

[00:42:56] JM: What I think is kind of a long-term problem, but probably an inevitable problem, is the fact that eventually we're going to have these machine learning systems that behave kind of in a way that looks non-deterministic. We're going to have these – We're going to have a system of machine learning models and it's not going to be easy to tell from the code like through what order different machine learning models will be hopped through. The only way to have perhaps the source of truth – I mean, it would either be some kind of logging or some kind of distributed tracing. But I think this machine learning auditability problem is eventually going to become pretty difficult to solve. Then it becomes a really important – What your retention policy is on those – When you're rolling the perfect recollection in-memory trace data into something that you're going to store longer term, you're going to have to choose carefully.

[00:43:53] JB: Yeah. I think that like each individual part of a machine learning model is reproducible and that like it's running a finite set of steps that result in a certain score. But as those change overtime and you sort of need that kind of record of it, it was this thing at this time with these settings as part of your record of that transaction so that at worst case you can go back and reproduce and say, "Okay, I'm going to load all these models up at these versions with these settings and figure out why it made the wrong decisions." I think that that's definitely a thing, and that's part of why like annotating this information, these PaaS [inaudible 00:44:35], that give you that ability to contextualize that particular transaction is pretty important, because if

you don't include that information, then it really does become a total mystery how anything happened.

There're definitely tradeoffs between logs and traces, and you can actually put logs on traces. So there are some ways through that, but it's tricky too, because retaining logs indefinitely without some idea of their importance, especially in a post-GDPR world, is pretty fraud as well.

[00:45:08] JM: Can you go a little bit deeper on that logging versus tracing area? Because I feel like even if the platform engineer has handed me this perfect platform to work within, if I'm going back to my Amazon days when I was just a neophyte engineer working on a service and I kind of understand how Java works, but don't really understand anything else about anything in engineering, and then I've got this like buffet of different logging and observability tools. Chances are I'm going to use them in a suboptimal series of steps. So is there a set of best practices for how I can choose between logging and monitoring and distributed tracing for solving a given problem? I mean, is it intuitive for the kind of person that has all of these things available to them?

[00:45:59] JB: I don't think it tends to be intuitive, because I think that there's always the joke about print of debugging. Well, at least in the C days. Whatever print is in your current language now, I think a lot of people still treat logs that way as the record of the execution of their application. Usually, that's not structured most of the time. So it's just a log of human language describing what they thought they might want to know. Often, it isn't what they actually want to know.

So I found that logs are usually good for forensics. If there is some level of detail that you want to track about state across time at the service level and not necessarily the request or transaction level and that you need – You'll know that you need to look at it for forensics, for auditing, for whatever else within like seven or so days. Then you have [inaudible 00:46:54] log.

Metrics are for understanding aggregates, and aggregates by some different sorts of dimensions, which is where different kinds of tags come in for that. But you're looking usually accounts. Almost always, your interest in counts are derived on counts to understand how flow is working or how levels are relative to [inaudible 00:47:20] levels in the past.

Whereas distributed tracing is this sort of like long set of things that represent, often, a customer view, or a consumer view of what happened. So that when there's a bad experience at the overall level, you can see which components caused that to happen. That's basically impossible to do with logs or metrics, because they're not structured for that kind of, not necessarily long-term, but long-term relatively, narrative about what the experience is.

So if you're going to use logs, that helps a lot for them to be structured, so that you're not trying to read a million logs. But logs for real-time interaction is pretty tough. I think it's useful to think about all of these though as events, and they're just events with different structure. The components of a distributed trace as spans, so those are events that have a particular structure that describe start and end times, tags, what the particular thing was.

You could certainly log those, but is the construction of those into a trace that makes those interesting. Similarly, counters or settings on gauges for metrics are also events, and those events become interesting when you aggregate them together into counts and graphs and time series.

[00:48:43] JM: If I take you back to your Twilio days, Twilio is this thing that has to be super highly available, super resilient to tail events, because SMS delivery can be this life or death matter. So you have to be able to identity and handle outlier events. You need to be able to somehow have your observability tooling, or your platform engineering tooling, able to capture, "Okay, when did an SMS take an extra eight minutes to be delivered to somebody for some reason?"

Do you have any tips on instrumentation to be able to capture those outlier events, or are those easy to capture?

[00:49:26] JB: If you're instrumenting with a tool that has accuracy – I don't actually know what the technical term is. HdrHhistograms or something that it's been called at different times, there are a few different iterations that have happened. But you need to be –

[00:49:44] JM: Sorry. What is it, HDR?

[00:49:46] JB: Yes. I believe there're a few open source implementations of that, and there are various different approaches that people have taken. But what you need –

[00:49:56] JM: What is HDR stand for? Sorry.

[00:49:57] JB: I think it's high-definition something. It's referring to high-definition histograms, ones where you can get –

[00:50:04] JM: Oh, like FlameGraphs.

[00:50:06] JB: Yeah, bucketing. So that you can do bucketing that actually gives you Five9's latency. So that you know actually how far out your outliers are, because it's relatively easy to get into a situation where there are operations, they're done on the data that's gathered, which destroys data. So if you like go, "Oh! I've got all these different histograms, these different distributions or latency, I'm going to average those together."

Well, all the data is just gone at that point, because it's not that kind of data. It's not that kind of data structure. You need to be able to capture the distribution of times it takes to do something into a data structure that allows you to preserve a very wide set of things from millisecond, to seconds, to minutes even. That's something that goes to the – That I was alluding to in terms of accuracy of tools. It's pretty important to have tools that are actually giving you accurate information about what your tail latency is, what your tail air rate is. I've seen situations fairly regularly when we're using those actually for a log aggregation Twilio, where we would see canary deploys in our cloud providers. We would see that the Five9's latency go up and we'd be like, "Uh-oh! Once this roles out for real, there's going to be a problem," and four hours later, it rolls out everywhere and it's like, "Yup! It's just got 10 times slower, or started falling over," or whatever else. But being able to see that kind of tail latency for the SaaS that you're consuming, the cloud services you're consuming, is also very, very interesting.

[00:51:52] JM: Another way to capture these outlier vulnerabilities, or to at least raise your probability of catching them, is these game days or chaos engineering, and I know you're a fan of these things, because basically – If I understand game days correctly, it's kind of you take a

day where, “Okay, we’re not going to focus on building the new business logic and the new services and stuff. Basically, we’re going to spend a day beating up the infrastructure and causing random failures and see what comes up, see what kind of unexpected vulnerabilities emerge from the system.”

Tell me about your, I guess, preference for game days and why game days are important, or why chaos engineering is important to you.

[00:52:39] JB: Sure. I think a lot of it is that it’s the same sort of assumptions that what your code should be doing is what it actually is doing have to do with what happens when your providers or your downstreams or whoever else you’re related to have issues. So in our experience at Twilio doing that, we found that we put together the system. We thought it was really great. We’re like, “Yes, we’re ready to ship it. Let’s do a game day.” Usually a game day actually isn’t anywhere near a day. Probably most people – It’s pretty stressful. You don’t usually want to do it for more than a couple hours at once, because it is like the worst times of on call especially when you’re originally starting with the system that doesn’t have resilience and doesn’t have [inaudible 00:53:29].

But what it allows you to do by purposefully failing the things that you depend on to make your application work, you can see – Because you know that you created the failure, you can see whether you can observe the failure you knew you created, including its impact on business metrics. An astounding amount of the time you cannot actually see these things, because it wasn’t instrumented that way or wasn’t made observable that way. It’s made observable for the happy path. Once you’re off the happy path, it’s weeds, and ruts, and all kinds of a mess.

The chaos game day process, by purposely failing these things, you get to see what you can’t see but you know actually exist. You know that there’s impact. You know that your stuff is not working correctly, and you know that you can’t see it, and that tends to be very motivating for everyone including application engineers, because they know. It’s not some theoretical that’s going to happen in the future. It’s failing right now. They can’t see it. It probably will fail like this at some point in the future. Let’s fix it. Let’s make it observable.

Then once you get past that initial, we can't anything. Then you can start working on resilience, and resilience – Making the choice, making the choice of what tradeoff you want to make upfront. Do you want to slow the system down? Do you want to stop it? Do you want to raise alerts? Do you want to take some kind of fallback action? You can make all those kinds of decisions, and then test those. Test your resilience strategy.

One of the things I've also found with this is that an untested resilience strategy is like an untested backup. It basically doesn't exist. So unless you're regularly testing these resilience situations either through manual game day or through automated chaos engineering, your resilience story will stop working. Then when there is a failure, you go, "Oh! We developed resilience to this," and it's like, "Well, we haven't tested it in six months. So we made some code changes that actually broke our whole resilience strategy."

So game days eventually, I think, it tends to be higher stakes to do continual chaos engineering, but game days are relatively low cost a couple hours, even once a month, that eventually we got to do once a week to develop confidence, but in your ability to address these issues, develop confidence in your observability, develop confidence to address things on call. But I think the even more interesting thing than that is that they change the way that application developers look at developing applications.

They change the mindset so that when you develop an application, when you add a feature, the question immediately comes to mind, "How am I going to observe this? Because I know it's going to go into chaos game day. What resilience strategy, what tradeoff am I going to use here? Because I know what's going to happen. I know that failure is real, and I know that I need to make a choice in how I design my application, how I develop my code to make that application observable, to make it resilient."

[00:56:42] JM: One of these areas, the whole game day chaos engineering space that, is almost like the machine learning model example where there's so much depth to this problem. You think about, "Okay, what does a game day look like if I decide my continuous delivery service is like offline today?" For whatever reason, we've deferred continuous delivery to some provider, and let's just imagine that they had an outage, or like you're, "Okay, let's imagine my

logging service provider has an outage. Can we solve these problem without logging?" That's probably a game day level that you don't really want to do at most companies, at least today.

[00:57:22] JB: Well, it's funny, because I've seen that happen several times in the last couple of years, both of those examples, yes.

[00:57:27] JM: You've seen the continuous delivery. Oh my God!

[00:57:29] JB: Yeah. So you're left with the situation where you need continuous delivery to fix continuous delivery, and it's worth having a strategy for that, I personally think. Those were –

[00:57:43] JM: Circular dependency CI.

[00:57:46] JB: But people will do that, because people like live in the happy path, right? So you go, "Well, how do I build this?" It's like, "Well, for instances, why I'm going to build this, the instance in the same region that's having the outage to fix the outage," and you go, "Well, that's not going to work. So what do we do?"

It's funny, looking back at the different incidents and outages that I went through at Twilio. I think the worst ones were actually the ones which were like this, which it wasn't hard down. There was a pervasive problem, but we could not replace capacity because or some other outage. So if one thing goes wrong, we're going to be in a lot of trouble. So what are we going to do? Because the tooling is built around this. When that goes on for hours, it definitely makes an impression.

So I think that that's not actually unrealistic. To think about it, I mean, I think it's certainly hard to do practically. But for that particular example, it's something that it's worth giving some thought to, because if you have – And there have been some multi-hour outages with different cloud CICD solutions, you should have a plan to do something in that kind of situation.

[00:59:06] JM: To close this off, it's been a really great conversation, are there any gaps that you see in observability tooling that still exist today?

[00:59:14] JB: I mean, I think the biggest one is still cost of integration. I really like to see us have the expectation that all of our frameworks have basic integration for observability, distributed tracing, metrics, logs even. This should just be table stakes for any majorly or commonly used framework.

Right now the case is, “Yeah, you can create some add-ons, or you can buy a thing from this person or that person,” and they’re all going to do things a little bit differently. I’d really like to see us move towards observable by default applications, and I think that starts at the framework level. It’s a little bit hard to solve at the language level, because that’s a bit too general. But most frameworks have a particular use case that you can meaningfully create a set of default instrumentation, default observability around, that you can plug all kinds of different providers into to help you surface insight. But I think that expecting people to instrument observability themselves every time differently everywhere is a losing battle from migration’s standpoint. People won’t do things that aren’t core value and that are expensive. So I’d really like to see us move the ball for that kind of default observability.

[01:00:35] JM: James Burns, thanks for coming on the show. It’s been a real pleasure talking to you.

[01:00:38] JB: Thanks. It’s been great to be here.

[END OF INTERVIEW]

[01:00:44] JM: GoCD is a continuous delivery tool created by ThoughtWorks. It's open source and free to use, and GoCD has all the features you need for continuous delivery. Model your deployment pipelines without installing any plug-ins. Use the value stream map to visualize your end-to-end workflow, and if you use Kubernetes, GoCD is a natural fit to add continuous delivery to your project.

With GoCD running on Kubernetes, you define your build workflow and let GoCD provision and scale your infrastructure on-the-fly. GoCD agents use Kubernetes to scale as needed. Check out gocd.org/sedaily and learn about how you can get started. GoCD was built with the learnings of the ThoughtWorks engineering team who have talked about building the product in

previous episodes of Software Engineering Daily, and it's great to see the continued progress on GoCD with the new Kubernetes integrations. You can check it out for yourself at gocd.org/sedaily.

Thank you so much to ThoughtWorks for being a longtime sponsor of Software Engineering Daily. We are proud to have ThoughtWorks and GoCD as sponsors of the show.

[END]