EPISODE 360

[INTRODUCTION]

[0:00:00.0] JM: Containers make it easier for engineers to deploy software. Orchestration systems like Kubernetes make it easier to manage and scale the different containers that contain services. The popular container infrastructure powered by Kubernetes is often called cloud native. On Software Engineering Daily, we've been exploring cloud native software to get a complete picture of the problems in this space and the projects that are being worked on as solutions.

One area of interest; how should services communicate with each other? What should be standardized? How can you easily identify problems and avoid cascading failures? One solution is the service mesh, a tool that allows services to communicate with each other more safely and effectively.

William Morgan was an engineer at Twitter, he was helping to scale the company in the early days when the company was dealing with lots of outages, fail whales. He was on the show previously to discuss his experience scaling Twitter, and in today's episode we go into the company that he is running today; Buoyant, where he works on building a service mesh called Linkerd.

Software Engineering Daily is looking for sponsors for Q3. If your company has a product or a service, or if you're hiring, Software Engineering Daily reaches 23,000 engineers listening daily and we're looking for interesting sponsors who have a message that they want to get out to engineers. Send me an email, jeff@softwareengineeringdaily.com. I'd love to hear from you. Thanks for listening to the show.

[SPONSOR MESSAGE]

[0:01:50.0] JM: Artificial intelligence is dramatically evolving the way that our world works, and to make AI easier and faster, we need new kinds of hardware and software, which is why Intel acquired Nervana Systems and its platform for deep learning.

Intel Nervana is hiring engineers to help develop a full stack for AI from chip design to software frameworks. Go to softwareengineeringdaily.com/intel to apply for an opening on the team. To learn more about the company, check out the interviews that I've conducted with its engineers. Those are also available at softwareengineeringdaily.com/intel. Come build the future with Intel Nervana. Go to softwareengineeringdaily.com/intel to apply now.

[INTERVIEW]

[0:02:46.1] JM: William Morgan is the CEO of Buoyant and, recently, he was on Software Engineering Daily to talk about scaling Twitter, and he's now back on to discuss Linkerd. William, welcome back to Software Engineering Daily.

[0:03:00.5] WM: Thank you very much, Jeff. It's a pleasure to be back to one of my favorite podcasts.

[0:03:04.2] JM: We're going to talk about Linkerd, but first we should talk about what Linkerd is more abstractly, which is this idea of a service mesh. What is a service mesh?

[0:03:13.8] WM: Service mesh is a dedicated infrastructure layer that handles service-to-service communicating, handles kind of the operational aspect of that. If you think of a big micro-service or multi-service application, you might have service A talking to service B, talking to C, talking to D. What the service mesh handles is things like retries, timeouts, deadlines, circuit breaking, the kind of operational stuff, as well as providing instrumentation and things like distributed tracing, and ideally control over the communication as well.

[0:03:45.8] JM: As an engineer, when I write a service call from my service to some other service within my company, what is the interface that I want to have to that other service?

[0:03:59.6] WM: Ideally, what you want is a service doesn't know anything about the underlying infrastructure. It doesn't know anything about service mesh, it doesn't know anything about Linkerd. Ideally, it doesn't even know kind of anything about whether it's running in Docker, in Kubernetes, or any of that stuff. What you want the ideal situation is service A only knows that,

A; there's a service B, and to talk to it, I like say, "Connect to B." Now, in practice, it's not always 100% possible to totally decouple the application from that.

[0:04:33.4] JM: Aha. The architecture for a service mesh is that I've got all of these different services that I'm running throughout my company and each of those services has some part of itself dedicated to the service mesh functionality, and then you also have this other centralized component of the service mesh that is aggregating and doing other things that serve a purpose for the overall applications. Give us an architectural breakdown of the different components of the service mesh.

[0:05:10.8] WM: Sure, and I'll give you a little bit of historical context too, because I think, usually, helpful for putting this in perspective. Maybe I'll start with that actually, because the service mesh idea is a new idea and it can feel like, "Oh, this is another thing that we have to add to our stack. We're already adding all these other things, and now there's this whole thing that I have to learn about and understand." I think, in reality, what the service mesh is, it's not really a new thing. It's a moving functionality that used to exist somewhere else into a separate layer.

If you think of the way that applications evolved overtime, in 15 years ago, we had kind of this idea of the three-tier application where you'd have some web serving layer, like nginx or Apache, and then you'd have your application in the middle and you'd have a database at the end, and you'd have a very limited form of communication between these things.

The web server would talk to the application, the application would talk to the database, and that was kind of it. The web server Apache had very very sophisticated control over how it would pass request to the application. The application would have a client-side library for talking to the database that had very specific logic for how to manage the communication. That's kind of the state-of-the-art in circa 2000.

What happened overtime is you advance to forward a couple of years and you have bigger companies, like Google and Facebook and Netflix, they start decomposing that monolithic application. They break it down into lots of different things. Nowadays, we call it microservices. At the time, we went through this at Twitter too. We didn't have the word microservices, so we

called it SOA and kind of — We knew that wasn't a great word for it, but that was the word that we had.

What happened there was now you have much more service-to-service communication, and so what we started doing and what all these companies is that they had these kind of fat client libraries. Twitter had Finagle, Netflix had something called Hystrics, Google had Stubby, which was a format protocol, but also a set of libraries around that.

This was kind of the very beginning sort of service mesh idea. We have this kind of consistent model for managing communication between services. Does that make sense?

[0:07:21.6] JM: Yeah, absolutely. I used something like this when I was at Amazon, when I was an engineer at Amazon, there was a service that did this. Back then, I didn't know if it was a service mesh or a service proxy, or I didn't even know these terms, but the interaction you have with it as an engineer at one of these big companies, whether it's Amazon or Twitter or Netflix, is you don't interact with it kind of. It's a transparent thing that gives you some insights into how your service is performing, and it handles things that every service would want, like failover, and load balancing, and circuit-breaking.

[0:08:00.0] WM: Yeah, that's right. The service mesh concept today is really the same thing, except implementation-wise, rather than using these fat client libraries, we pulled that out typically as a separate proxy layer. You'll have these runtime proxies, and this is what Linkerd is. It's just a little proxy, and you'll run those alongside application code rather than having libraries attached to it. It's the same functionality. The reason to do it as a proxy rather than library is because it's easier for polyglot applications.

We have things like Kubernetes and Docker now that make it very easy to co-deploy stuff so that the deployment cost of running additional runtime components is much less than before. We have now the flexibility of building this later separate, totally decoupled from the application. That's really the service mesh. It's not really a new thing that we're introducing, it's just a moving of functionality from the application out to kind of the underlying infrastructure.

[0:08:56.7] JM: To clarify the difference, the deployment process for these fat clients would be you deploy your service on a server or on a VM somewhere and you include the library that is necessary to have access to this service mesh functionality, whether you're talking about Hystrics or Twitter's Finagle. That's in the old days. But today, since everybody is deploying via Docker containers, many people on Kubernetes, you'd much rather have this functionality in a separate container because it gives you a devoted area that is partitioned, that is it's devoted to the different things that you would want out of a service mesh or a service, or proxy. Whatever you want to call it.

[0:09:46.1] WM: Right. That's exactly right. It used to a compiled time binding where you'd use a library, and this isn't a bad idea. Many companies still work this way today and it makes a lot of sense, but it does constrain the set of languages that you can use because every new in language that you introduce, you'd have to port this library to, and that can become difficult overtime.

By separating it out as a separate proxy, a separate runtime component, now you've totally decoupled it from the application. If you have a nice polyglot system, that's fine. You don't have to maintain this across every possible language that you're using.

I wouldn't say this is like a moral, good thing to do. It's not like, "Oh, this is what God intended." It's more a reflection of the fact that not only are we now in a world where polyglot micro-services systems are much easier to do, but we're in a world where the deployment cost of running kind of these co-process models is substantially lower than it's ever been. We've got those two factors that kind of shift the equation for the cost or the relative value of running something as a sidecar.

[0:10:57.6] JM: In Kubernetes, if I'm correct about how the architectural model works, you've got a pod for each instance of your service, and in each of those instances you've got the container that is doing your application stuff, or maybe you've got multiple containers that are doing your application stuff for a specific instance of your service, and then you've got other containers that might be doing other stuff. One example might be a container for Linkerd. Is at the right architectural model?

[0:11:29.5] WM: Yeah, that's exactly right. One of the nice things about Kubernetes is that the pod model makes it very easy to deploy these sidecar processes. Linkerd is often deployed as a sidecar in Kubernetes. There's another deployment mechanism called the demon set in Kubernetes that distribute to one per host. That's also an option. Every Linkerd instance is statelessness and is independent, so we don't actually — It doesn't care. Where you choose to deploy kind of depends on the specifics of your application.

[0:11:59.8] JM: I'm writing a service that does X and my application code is in a container and some in the pod and then there's also alongside it the Linkerd container that is the proxy. What is the communication pattern between my application code and the Linkerd proxy?

[0:12:22.1] WM: Yeah, really good question, because it kind of depends. At least today, it's a little protocol specific. What you want is every application, every instance of service A, rather than talking to service B directly, rather than doing a DNS lookup and kind of relying on layer three, layer four transport to open up a TCP connection, you want it to talk through its local Linkerd instance and just pretend that's the destination. Linkerd acts as a little proxy.

if you're using HTTP, you can often set a HTTP proxy environment variable and just kind of have magically things work without you having to do configure changes. If you're doing something like GRPC, then it's a little more difficult. You actually have to make a config change to point it to a Linkerd instance.

I think, in the future, this will become much easier. If you look at the Istio project for example, which is a really good example to service mesh, they have some layer three, layer four IP tables magic effectively to do a lot of these automatically for you. I think, in the future, this integration point will be much easier. Right now, it's a little bit protocol specific.

[0:13:29.4] JM: The communication between different services goes through Linkerd. My application instance is in a container and it's talking to the Linkerd instance that's also in the pod along with this container, and then that Linkerd communication point is going to reach out to the Linkerd communication endpoint on the downstream service. Oftentimes, a service call is going to have multiple services that it needs to hit in order to fully fulfill the request.

Now that you've talked about how my application is going to talk to Linkerd, explain how Linkerd brokers the communication between two different services.

[0:14:17.9] WM: You're exactly right. Service A, that instance will talk through its local Linkerd instance, which will proxy the request to the destination Linkerd instance, which will then in turn proxy it to the destination service. We've actually introduced Linkerd on both sides. We're kind of simulating not only the client but also the server side of that hub.

From an architecture diagram, suddenly that looks a little scary because you've introduced two additional hops to every service-to-service call. What happens in practice is that the way that Linkerd does things like load-balancing and circuit-breaking actually can improve your tail latencies. Even though you're introducing these two hops, you've actually made things faster. At least you've made your tail latencies lower which are usually the things that you really care about.

By doing that, by having Linkerd on both sides of both the client and the server side, that allows us to really decouple the transport communication from what the application is speaking. A very common use case for this is to have Linkerd initiate and terminate TLS on both sides of the node. There's many production users of Linkerd in an environment like Kuberntes who just want to have TLS between nodes, but they don't want the application to do it, so you can offload that work to Linkerd.

One of the things we're going to get to in the near term, which I'm really excited about, is protocol upgrade. We will be able to take something like HTTP 1 one and translate that to HTTP 2 between the Linkerd instances, and there's all sorts of nice reasons why you might want to do.

Basically, what happens is you've decoupled with Linkerd with a service mesh, you've decoupled the transport from what the application really knows.

[SPONSOR MESSAGE]

[0:16:04.0] JM: For more than 30 years, DNS has been one of the fundamental protocols of the internet. Yet, despite its accepted importance, it has never quite gotten the due that it deserves.

Today's dynamic applications, hybrid clouds and volatile internet, demand that you rethink the strategic value and importance of your DNS choices.

Oracle Dyn provides DNS that is as dynamic and intelligent as your applications. Dyn DNS gets your users to the right cloud service, the right CDN, or the right datacenter using intelligent response to steer traffic based on business policies as well as real time internet conditions, like the security and the performance of the network path.

Dyn maps all internet pathways every 24 seconds via more than 500 million traceroutes. This is the equivalent of seven light years of distance, or 1.7 billion times around the circumference of the earth. With over 10 years of experience supporting the likes of Netflix, Twitter, Zappos, Etsy, and Salesforce, Dyn can scale to meet the demand of the largest web applications.

Get started with a free 30-day trial for your application by going to dyn.com/sedaily. After the free trial, Dyn's developer plans start at just $7 a month for world-class DNS. Rethink DNS, go to dyn.com/sedaily to learn more and get your free trial of Dyn DNS.

[INTERVIEW CONTINUED]

[0:18:04.5] JM: You talked about this term tail latency. Explain why that is such an important concept, and reiterate why Linkerd has an impact on that tail latency.

[0:18:18.3] WM: Sure, when we measure the behavior of a system that's responding to requests, if there's any kind of variability, it's very rare for us in this world of like multithreaded, multi-hosted, multitenant software to have a system that performs — It takes the same amount of time for every single request.

Typically, what happens is you have a distribution, you have a probability distribution, or a histogram over the latencies. You can actually draw that out. Usually, there's kind of a peak in the middle of like, "Here's how much time it normally takes," but then there'll be this long tail at the end. By talk latencies, we mean the request that took much much longer than most the others. When we characterize a performance of a distributed system, we'll often talk about the P-95, or the P-99, or the P-39s, these are all talking about the percentiles if you measure how

far out you are in the tail of that distribution, you're talking about the percentile. These are like the really bad requests.

I'll give you an example, if you have a system that most of the time takes 50 milliseconds to respond to a request, but every once in a while it will take 500 milliseconds, or 5 seconds to responds to a request, then you really want to know about that. Those are your tail latencies, and those are the latencies in a distributed system that you really want to control for and that you want to monitor. Those are kind of the scary parts of the system. Those are where things start breaking down where you have queues that are backing up, or where you have a garbage collector that's rearing its head, or where you have some kind of lock intention.

Monitoring the tail latencies is really one of the most critical parts kind of monitoring and operating a distributed system. Okay, that's tail latencies. So far so good?

[0:20:03.3] JM: Yes.

[0:20:04.7] WM: What Linkerd can do is because of the The way that we're doing load-balancing, the way that the service mesh operates is we're doing stuff at the request level often. That means that in contrast to like a TCP proxy where we'd be kind of sending bites, we'd open up a TCP connection and send bites from here to there. We're operating at the level request, so we'll proxy a request. As part of doing that, we'll measure the latency of an individual instance. We'll say, "Okay, this instance is really fast. This instance is really slow." It will start shifting traffic towards the instances that we expect to be faster.

That's where we're able to reduce the tail latencies. By being intelligent about which instances get the request, we can reduce tail latencies even at the expense of introducing additional hops to this process.

[0:20:53.1] JM: I can understand how when you've got two services, two service instances communicating with each other, those two service instances, the communication pattern allows you to measure the latency of each of those requests. In order to have insights about where are the other places where you could send requests between, you need to have some information that's being propagated among the different Linkerd sidecars so that they can be smart about

rerouting and doing the load-balancing. You need this centralized point of information that all of the different little Linkerd sidecars are communicating with. Describe that point of centralization.

[0:21:43.6] WM: We actually don't have a point of centralization at all for this kind of behavior. There's kind of a philosophical, somewhat philosophical debate. In the interest of simplicity, we decided that every Linkerd instance is going to be stateless and it's going to be independent of the other instances. They don't communicate, they don't share load-balancing information, which is good and bad. It's bad and that an instance can only observe, can only make decisions based on traffic that is observed. When a new instance comes online, it kind of has a very naïve view of the world. It doesn't have a way of collecting information from the other more senior instances.

The good thing, and the reason to this beyond simplicity, is the fact that different instances might actually be exposed to very very different latencies. You might have something in a different rack, or you might have something in a different datacenter, or you might just have some weird situation on this one machine. It's very difficult to reliably share latency information between instances that might be running in very different locations.

We made a decision early on to just say, "Okay, these things are going to be stateless, they're going to be independent, and there's a little bit of non-suboptimal behavior associated with that," most of the time it actually results in a better situation and easier to reason about.

[0:23:03.8] JM: Certainly feels right from distributed systems standpoint. The centralization is always the point where the headache centralizes. I guess I still don't quite understand how the metrics and the optimal routing information gets aggregated so that you can make smart macro decisions if you don't have this point centralization.

[0:23:32.6] WM: Oh, that's a really good question. What we do, what every Linkerd instance does is it measures the latencies and the success rates and everything else at the traffic that it seems and it reports that. You can talk to an individual instance. You can say, "Hey, tell me everything you've seen over the past minute." That's good. Everything is highly instrumented. You're right, at that point, you actually want to aggravate some of that data.

There's a variety of options for doing this. We have a little project called Linkerd-vix, which is just a prepackaged Grafana Dashboard and Prometheus that knows how to identify where all the Linkerd instances are and kind of automatically extract metrics from them. Once you have Linkerd running, for example, in your Kubernetes cluster, you can install Linkerd-viz and you can get an automatic top level service metrics dashboard, which his really powerful because you have things like success rates and latencies.

Those are the first order characteristics of service behavior that can be reported automatically for you independent of your application. The application doesn't know anything about this. The services are written in different languages or whatever. As long as the traffic is going through Linkerd, ou can actually get a top level view of everything that's happening in your cluster. That's actually a very common use case for the service mesh.

[0:24:50.5] JM: Do the Linkerd instances need to know about everything that's going on the cluster, or does every Linkerd instance just need to know about the services that are downstream from this individual service that we're talking about?

[0:25:07.8] WM: Every instance only knows about the traffic that it sees. It's fairly — I don't want to say naïve, but it's a fairly regimented view.

[0:25:19.3] JM: Let's talk about some of the different functionalities that we can get out of having this service mesh. What are the canonical examples of functionality that we can get out of this? You mentioned that this could really reduce tail latency. I know there's many other features that we get out of having this robust communication layer. Talk about some of these.

[0:25:44.7] WM: Sure. The service mesh — I'd say the real goals of service mesh is we want to give you a name and we want to give you a handle and we want to give you something you can you can think about and reason about and control that's reflect of the service-to-service communication inside your application. The real goal is to make that a first-class citizen of your environment. You can kind of — It's not just about putting — It's not just about taking code out of your application and moving it into a proxy. It's about turning all this behavior, which is really critical to the runtime performance of your system and turning it into something that you have a name for and that you can visualize and that you can control.

I'll go through a couple of those examples. We just talked about really good one, which is the top line service metrics. Once you have the service mesh installed, then you have traffic going through Linkerd. You get success rates and you get latencies, and those are two things that you want to alert on immediately. If the site is going down — If it's 3 AM and your CPU usage is increasing, do you want to wake up? I don't know. Maybe, maybe not, unclear. If it's 3 AM and your success rate is dropping. Oh, yes, you definitely want to wake up. Metrics is one thing.

In addition to metrics, there'd distributed tracking. Linkerd can automatically emit Zipkin traces, so you get distributed tracing. We say distributed tracing for free, but the for free is kind of in quotes because you actually have to do a little more work. You actually do have to pay a little bit for that. That's kind of on the visualization side.

There's the baseline reliability feature. Things we've talked about a little bit earlier on in the podcast, retries, and timeouts, and deadlines, and circuit-breaking, that's often the big driver for running a service mesh, because getting that code write is actually quite difficult. There's a big interplay, complex interplay between the way the load-balancing works and the way the retries work and the way that circuit-breaking works. You can kind of just rely on Linkerd for doing that stuff. That's reliability.

Then number four on the list is control. A big part of what the service mesh gives you is the ability to change the way this traffic is working at runtime. You can do things like change routing policy. We want to shift — I have service A talk to service B. What is service B? What does that mean? Does it mean this mean this one that's running in staging, or does it mean this one that's running in production? Does it mean the one that's running on this datacenter? Does it mean the one that's running on that datacenter? Does it mean this version or does it mean that version?

When service A talks to service B, the meaning of that — What service B, it actually wants to talk to is something that can be encoded in the service mesh and something that could be modified on the fly, which gives us ways of doing things like cross-state or datacenter failover, gives us ways of doing blue-green deploys, gives us ways of doing kind of ad hoc staging environments, or canaring environments. That's a type of control.

Then coming up a little bit later this year, we'll have things like security policy there as well, where you can say, "Okay, service A is not allowed to talk to service C. It can only talk to service B." Those are all things that can be changed on the fly in the service mesh, and it's totally decoupled from the application, which is I think what you really want.

[0:29:08.6] JM: Some of these things, it's very clear to me why we would want these — Assuming we're going to have a service mesh, it's clear to me we would want, for example, the gathering of information about distributed tracing, we would want that information gathering to take place in the service mesh. That makes complete sense to me.

Setting policies around how stuff gets routed, that's like a topic that's I've heard other people say, like, "Oh, you want to do this in DNS," for example. I think this gets at — Maybe it's something I am confused about, but you talked about this in a Cloudcast episode that I listened to where you were saying — Cloudcast is a great podcast, by the way, for anybody who doesn't listen to it. You mentioned the fact that the CNCF, the Cloud Native Computing Foundation, which is the Linux Foundation of — It's actually a subset of the Linux Foundation, where it's actually this place where it hosts discussions and oversight and governance for these different open-source projects, but the CNCF does not bless a specific stack. What I mean there is in this Kubernetes environment, this cloud native world, there is so much opportunity for different infrastructure technologies because it's such a revolution in how we build our applications that there are different services and technologies that have overlapping functionality.

You mentioned that you thought it was a really good way of doing business that the CNC — I shouldn't say business, doing nonprofit foundation governance that the CNCF doesn't bless a certain stack. It doesn't say, "Oh, yeah. This is like the lamp stack of distributed systems. This is the way things should be done." It just says, "Here's a bunch of stuff, and we are kind of like keeping our eye on all of it, and we can talk to you about it, and we can help broker relationships."

Between the different technologies, Linkerd being one of them, Kubernetes is another, Prometheus is another, those three have fairly, I think, disjoint sets of functionality, but there are other things that have some overlapping functionality. Explain why you think — I think this is a

really interesting area of discussion. I guess, first, talk about what it's like being in this kind of market where there are so many different players with overlapping goals and it seems like it's hard to isolate what your specific technology, what the bounds of it should be.

[0:31:52.7] WM: Yeah, that's definitely true. This is one of the most — One of the VCs like to call it frothy. This is one of the frothiest areas, which I think is a sign, means it's a sign that there's a lot of invitation happening in this space. The world of how do we run big distributed systems in a way that's scalable and safe and reliable and that doesn't wake people up at 3 AM for silly reasons, that's a very unsolved problem. The world is just starting to scratch the surface of what that looks like.

I think the fact that it's so early in that space and yet everyone has to do this, it means that there's a lot of solutions and people are still figuring stuff out. I think a lot of the — There's going to be overlap. There's going to be overlap. There's going to be, I think, evolution of all these things. What Linkerd looks like today, what the service mesh looks like today, I think will be quite different from how it will look in two or three years ago.

Linkerd is a user space proxy right now. Is that really the end goal? I don't know. Maybe, maybe not. Right now, we're using containers and we're running them in VMs and they're running the VMs on some hardware that someone else owns. Is that the end goal of all these stuff? I don't know. I think the world is still figuring this out. I definitely don't speak for the CNCF, but what I like about the CNCF is that I think they recognize the fact that there's a certain amount of editorial aspect to being a part of the CNCF. They don't accept any old projects. There are things that they care about, and there's value to having the CNCF logo on your project, because it means that you've past their editorial kind of criteria.

They also know, they also recognize that the surest way to kill a technology is to have a foundation or to have a consortium or to have a standards body talk about it. That's the surest way to slow down the pace of innovation and to make sure that nothing interesting or important is actually going to happen in the project. I think we've seen — The software industry have seen that happen again and again. If you look at things like enterprise, Service Bus and CORBA. Probably going back, there's even more examples. You just don't want that. The CNCF is, I

think, doing a pretty good job of balancing its role of being an editor, but not being a standards body or a consortium of companies that making technical decisions by committee.

[SPONSOR MESSAGE]

[0:34:37.0] JM: Your application sits on layers of dynamic infrastructure and supporting services. Datadog brings you visibility into every part of your infrastructure, plus, APM for monitoring your application's performance. Dashboarding, collaboration tools, and alerts let you develop your own workflow for observability and incident response. Datadog integrates seamlessly with all of your apps and systems; from Slack, to Amazon web services, so you can get visibility in minutes.

Go to softwareengineeringdaily.com/datadog to get started with Datadog and get a free t-shirt. With observability, distributed tracing, and customizable visualizations, Datadog is loved and trusted by thousands of enterprises including Salesforce, PagerDuty, and Zendesk. If you haven't tried Datadog at your company or on your side project, go to softwareengineeringdaily.com/datadog to support Software Engineering Daily and get a free t-shirt.

Our deepest thanks to Datadog for being a new sponsor of Software Engineering Daily, it is only with the help of sponsors like you that this show is successful. Thanks again.

[INTERVIEW CONTINUED]

[0:36:01.1] JM: It does seem important in this space to have some differentiation and to know even if you don't know exactly what you are going to do, maybe you at least know what you are not going to do. For example, Linkerd can say, "We are not a container orchestration system." Maybe you have intentions to laterally move into that. You do not. Okay. Do you have a good idea for what the bounds of Linkerd are? Do you know what you don't want to do?

[0:36:34.0] WM: Yes. The bounds of Linkerd are services-to-service communication. Anything around that is stuff that we want to tackle because we believe that not really a critical part to

how modern applications perform at runtime, but it's a totally raw undiscovered area that really is in need of some real innovation.

That stuff that we do want to do, stuff that is around, like deployments and orchestration and talking to raw hardware, or even layer three, layer four stuff. It's not really what we're interested in doing. It's not core to the mission, and I think we all want to live in a world of like the Unix philosophy where you have these tools that kind of have very isolated areas that are — Not isolated, but well-defined areas of functionality.

[0:37:31.0] JM: It's been interesting to see Kubernetes up as this super popular project that outstrips almost any — I think it outstrips any other open-source technology in terms of how fast it's risen and how much velocity and acceleration it has behind it. When I talk to people about the idea of this service- to-service communication problem, it's not like everybody agrees this is something that should be solved. There seems to be some convergence around the idea of having a sidecar that sits next to your application.

Why doesn't Kubernetes itself standardize on something? Where is the subjectivity in this problem?

[0:38:20.1] WM: I think Kubernetes has done a pretty good job of drawing the line or drawing the boundaries for, "This is the stuff that we care about and this is the stuff that we don't care about." Kubernetes goes on the network level. They kind of go to the point where, "Okay, we have layer three, layer four in place, and we have DNS," because any kind of early-stage bootstrapping, you should probably have DNS. That's as far as they've wanted to go.

I think that's, to their credit, they haven't wanted to make it into like a PAS, a platform as a service, where you have everything in there, like "Here's how you do CICD," all the way down only down to, "Here's a runtime that — Here's how you deploy — Here's how you write code. Here's the IDE." They've wanted to tackle one very specific problem here, which is a scheduling orchestration problem. I think that's to their credit, and I think that attitude means that innovation can take place in other parts of the stack too.

[0:39:17.8] JM: Do we have a public understanding of how Google does service proxying internally?

[0:39:23.6] WM: That's an interesting question. I don't know. I believe I've heard that the majority of traffic at Google is run through a service proxy. I don't believe it's one that's been open-sourced, but I may have misheard that. I would ask someone from Google. I don't know. I understand Twitter pretty well, which did not move into the proxy world or even into the container world. Yeah, I don't know what happened at Google. I'm sure it's something amazing though.

[0:39:53.3] JM: I'm sure it's amazing too. A quote from an article that I'll put in the show notes about; service meshes. This is a really good read that you wrote that's on the Linkerd, or on the Buoyant I/O websites, sorry, "The service mesh must be designed to safeguard against the many opportunities for small localized failures to scale into system-wide catastrophic failures."

Now, this is kind of the idea of cascading failures where like, "Oops! This minor exception that was thrown cascaded into a fail whale."

What are the catastrophes that a Linkerd or some other service proxy or service mesh can help prevent?

[0:40:41.6] WM: This is one of the funny things about distributed systems is we've done all these work to have these like super decoupled, decentralized things, and yet if the communication between services is done naïvely, we've actually made an incredibly fragile system. The reason I know this is because we went through this process that Twitter where we failed in every possible way that there was to fail and kind of slowly worked our way out of that swamp.

One of the most common ways for something to fail in a distributed system is if one component start slowing down, and they could slow down for any number of reasons. Maybe the garbage collector is going on, or maybe someone's running the vacuum cleaner nearby, and that's interfering with the electrons. I don't know. That's not a real example. There's are any number of ways for an individual system to slow down.

If the service you're talking to is slowing down, it will start hitting a timeout. The caller, I'm service A talking to service B, if B doesn't respond in 500 milliseconds, well, I'm going to retry, because maybe I hit some crappy instance, so I'm going to retry. Still taking 500 milliseconds, so I'm going to retry again. What am I doing? I'm actually adding load into the system, and the more load I add, the slower it's going to get. What happens? B starts slowing down. Am I A? Yeah, I'm A talking to B. I should have started with B talking to C.

Let's say C is slowing down, and so B is adding more load on to C, which is compounding the problem. Furthermore, B is starting to slowdown too because it's just waiting. It's got this incoming request from A, and it's saying, "Hey, I got to talk to C, but C is not responding. I'm still retrying. Hold. Hold on. I'm going to retry. I'm going to get it this time guys." B starts slowing down, and then A starts slowing down. Pretty soon, this one isolated little failure here cascades to the extent where you have a side-wide outage.

I think the core problem is that load and latency are kind of really intertwined in a distributed system, and most of the naïve approaches to managing this ad more load in the case of latency, and so you end up with this horrible situations.

[0:42:57.2] JM: That just knows we're almost up against time. Do you have an extra 10 minutes, 10 or 15 minutes?

[0:43:00.5] WM: I sure do.

[0:43:01.6] JM: Okay, great. You mentioned like the vacuum cleaner thing. I did a show recently with somebody who was at Google for like 10 or 11 years. This guy; John Looney, and he worked on work on Google infrastructure and he talked about this incident where the postmortem was basically that cosmic rays had flipped some bits somewhere and it caused a cascading failure and which is like down for the system, it's like cosmic ray.

That is a tail event, but if you're Google, you hit the tails. You hit every tail.

[0:43:40.5] WM: Right. I'm impressed they were able to trace that to cosmic rays.

[0:43:46.6] JM: Yeah. It could be something. I've heard — I hear when they're building satellite systems, they really need to watch out for those cosmic rays, because they don't have the atmosphere, I guess.

[0:43:57.9] WM: Yes, but satellites are in space. Like the Google datacenter, it's not — I suspect it was a programmer somewhere who wrote a bug and then decided to blame to a cosmic ray. That's the true story of what happened.

[0:44:16.5] JM: I don't know.

[0:44:17.7] WM: No, I have no idea.

[0:44:18.8] JM: That is probably the true story. Yeah.

[0:44:20.5] WM: I have no idea.

[0:44:20.9] JM: Yeah, who knows. Yeah. We talked — You mentioned Prometheus earlier, and that's this monitoring system for distributed systems. We talked kind of about where Linkerd ends, and it's for service-to-service communication, but it helps with these insights about how the overall system is working. It can help with distribute tracing or monitoring. What is the interaction between Linkerd and whatever the monitoring system is? Is it like the monitoring system is pinging the different Linkerd instances and aggregating information? How exactly that does that work?

[0:45:01.1] WM: We have a whole plug-in models where you can kind of do whatever you want to do. Most of the plugins like the Prometheus plugin will pull the Linkerd instances, so Prometheus will go around and it will talk to each Linkerd instance and it will say, "Hey, what are you seeing? Okay, what are you seeing? What are you seeing?" It will aggregate all that stuff together.

We have a Statsd plugin, which I think is a push model. We'll support whatever. The basic kind of division of labor is at Linkerd will instrument everything and will report it, but we don't do

anything with aggregation. What you do with that data is kind of up to you. If you don't get it fast enough, sorry, it's disappeared. It's stored in memory in Linkerd and you need to read it once a minute otherwise it goes bye-bye.

[0:45:46.0] JM: I've done a couple shows recently with people who are working on serverless on top of Kubernetes applications. These are Cubeless and Fission. What's clear whether it's — Whether people are going to be doing serverless on top of Kubernetes or they're going to use AWS Lambda or whatever, it seems to be a pretty good opportunity, and people talk about serverless — When I've talked to these serverless on Kubernetes guys, the way they say is serverless is really useful for these glue code sort of things where you just have a very — Well, I don't want to put words in their mouth. You can listen to that those episodes if you're interested.

I should just ask, what is the interaction pattern that you see evolving between the serverless technologies, like AWS Lambda and Kubernetes, and how might that impact with the direction that Linkerd goes in?

[0:46:45.9] WM: I think in a lot of ways, serverless is a pretty natural evolution for both Kubernetes and for Linkerd. As the cost of deployment goes down, it becomes easier and easier, and this is what Kubernetes has given us, is now when you deploy something, there's a set of APIs and we have a container for kind of managing the runtime stuff, so you just like type some commands and, poof, it's deployed. That's a huge difference from where we were 5 or 10 years ago.

As a cost of that deployment goes down, it's easier and easier to move into a world where — Think of it as it's almost like auto-scaling where the base state, rather than always having the service running, the base state is zero and you just spin it up on demand and then you shut it down. All the same problems that you have around service, service communication exist in the serverless world, you actually haven't — If anything, you have it more. A service will spin up — Sorry. A function will spin up, and then it needs to talk to another function, which needs to talk to another function.

Not only is the deployment aspect, or the instantiation aspect, there's also the communication aspect. I think it's a pretty natural extension of the work that's happening in Kubernetes and also in Linkerd.

[0:48:01.5] JM: As long as I'm using a homegrown open-source serverless on Kubernetes technology, like Fission or Cubeless, I think I can specify what sorts of sidecars are going to be spun up along with the serverless function that I've deployed. That might not be so easy if I'm using one of these opaque things, like AWS Lambda or Google cloud functions.

[0:48:31.2] WM: Yeah, that's probably right. That's probably right. This is a really interesting area for us, so I think we'll be investing in it pretty heavily coming in the next couple of months and years even.

[0:48:43.6] JM: Man, talk about frothiness. That's going to get frothy and quite interesting.

[0:48:48.2] WM: Yeah, if you thought that Kubernetes an service mesh and all that stuff was confusing enough, yeah, it's going to be worse. That's going to get worse before it gets better, but it will get better in the end. I think the pattern that we're seeing, and I just compare it back to where we were — Again, I refer to my Twitter experience because that's kind of the big fixed point in my mental landscape.

Man, when we started doing this stuff in 2010, nothing — There was nothing there. We built a whole — What I would argue is the first cloud native architecture at Twitter, even though it wasn't in the cloud, without containers, without an orchestrator until we introduced one later. Without even the word microservices. It was really painful, and if we did that today, we'd have so much tooling and we'd have so many thing we could rely on. It would have been so much easier to do that.

The world is improving. Well, the world of software is improving. The rest of the world is going to help.

[0:49:43.9] JM: Right. Yeah. We need the Kubernetes of presidential candidates or something. Kubernetes 2020.

[0:49:56.2] WM: I wonder. Can you run Kubernetes in your nuclear bunker? Maybe that's the next startup fad.

[0:50:04.0] JM: You talked about the trauma of Twitter early days, probably it was the same at Netflix or Amazon or any of these giant companies that grew up in the growing pains between pre-cloud native and post-cloud native. That's probably growing up in the depression where these people who grew up during the depression and they are forever grateful for — I guess — No, I guess the people — You know what they say about the children of the depression is they're just forever traumatized, but they're probably also grateful.

[0:50:34.1] WM: Right. You always have to eat everything on your plate, because if you don't, you're wasting food. Maybe I'm going to end up like that, where you have to use 100% of your CPU at all times. Otherwise, it's like a moral failing on your part. You've like wasted CPU. In my day, we barely had any CPU at all. We had one for the whole family and we just had to share it.

[0:50:56.2] JM: Yes. I want to come back to these different overlapping projects in the CNCF area, because it's this interesting thing where you've got to do — The Diplomacy here must be so interesting, because you've got all these different projects and some of them kind of maybe have dependencies on each other, or there's coopetition or you don't want to make some breaking change against an update that's coming. Do you have any — Can you give some perspective on what the diplomacy is like in this space?

[0:51:30.2] WM: It's not that bad. It's not that bad, because it's not a zero-sum game. Every month, every year, there's more cool stuff to do and there's more opportunity. You're always rubbing elbows with someone. Is that good or bad? You're rubbing shoulders. There's always a little bit of overlap in what you're doing. Of course, you think your thing is better than theirs or maybe it's not. I don't know. There's so much opportunity and it's increasing from day to day, that there really is not a lot of infighting. At least there hasn't been in my experience.

[0:52:05.5] JM: That's great to hear.

[0:52:07.0] WM: Then, again, I'm small fry. I'm not like Docker versus Kubernetes versus Mesos versus whatever. I'm just little old me over here. Maybe I don't get exposed to that.

[0:52:17.9] JM: Yeah. What was it like seeing that brouhaha among — I don't know about — I don't know about the Mesos side, I think, but certainly the Docker versus Kubernetes stuff. Really, that conflict boiled over for a little bit.

[0:52:33.7] WM: Yeah, that's right. Maybe that's a counter example to my fairly optimistic view in the world. I was not involved in that. I was just a bystander watching. I really don't have a lot to say.

[0:52:46.5] JM: You didn't take any lessons away from that or anything? Maybe don't fight or — Don't get caught in a conflict.

[0:52:53.8] WM: No, I think the only lesson I really took away from that is that in the open source world, having a community that really believes in you and that feels like you're a good citizen is an incredibly powerful thing. I think if the moment you start isolating your community — Now, I'm speaking with my CEO hat on rather than my open-source maintainer project hat, or open source project maintainer hat, I think it's always — Finding the line for an open source company like Buoyant, there's a line that you have to walk between — Obviously, you are a company and you have to make money at some point, otherwise why are you doing this at all.

You have to — At the same time, you want to have an open source, a thriving open source community that trust you, and that feels like you're doing the right thing for the project. I think the moment that you lose that, that's when things start turning south. When your community abandons you or the community doesn't trust you or it doesn't believe you or doesn't think that you've being honest about stuff, things that you're being overly commercial where you're doing something that's not that their benefit because it's going to make you rich, that's when things could start going south. That's maybe the one lesson I took away from that.

[0:54:10.3] JM: Yeah. It was all about the optics, because Docker bundled this thing that made it look like they were really trying to push their container orchestrator in the face of the growing

Kubernetes popularity, and all these people who are adopting Kubernetes were saying, “Why are you giving us bloatware in our Docker containers?”

Whether or not they were doing it because — I think what Docker, and this kind of makes sense, is like we want to have an all-in-one solution that’s really simple. In order to get that, we need to bundle this thing in, and that’s our vision. The perception from the community was that you’re trying to force us to use Docker Swarm.

[0:54:53.2] WM: Right. I think that’s right. This is now getting quite philosophical. It’s not like Google is this totally altruistic company [inaudible 0:55:02.6]. They’re doing Kubernetes for a reason.

[0:55:04.7] JM: Exactly!

[0:55:08.4] WM: I have a friend, Ben Sigelman who’s the CEO of this company, Lightstep, and he has a really a good quote that I like, which is that Kubernetes is a ship and it’s a ship that’s built just big enough to take all the money that you’re spending to Amazon and to kind of sail off with it, going somewhere else.

Every company here has a profit motive including my own, has a profit motive. No one is doing this out of charity. On the other hand, I am quite optimistic in general that I think there are ways of making very successful businesses on top of open source and keeping the community happy. I think if you have the right line between the commercial features and the open source features, I think you can accomplish that, and I think the world of open source, anyways, it’s becoming more comfortable with the fact that, “Gosh! Sometimes there are open source projects that are driven by companies and it’s still okay.”

I think if you look at open source — My first exposure to open source in the 90s, believe it or not, with Linux. This is a sign of how I’m old I am. When I was in high school, we would pass around a Linux distribution as a stack of floppies in a lunch bag, and that was a stack of 50 floppy disks, 3-1/2 inch floppy disks that was running slackware version 0.00 whatever. In that world, open source was kind of the antithesis. It was the opponent of a commercial venture. You

did open source because you didn't want to be like Microsoft. Now, the world is much more receptive to the idea that, gosh, you can kind of do both. You can have a project.

I think Kafka is a really good example of this. You can have Kafka, which is an amazing open source project. You have a company like Confluent behind it, and it's okay. In both cases, you got a good company and you got a good project. A really good company and a really good project.

[0:57:00.4] JM: Yeah.

[0:57:01.2] WM: That makes me quite optimistic.

[0:57:03.8] JM: From the whole rise of Kubernetes, just from a business case study point of view, it's like beautiful jiu-jitsu of, "Yeah, we're going to release this thing that lets you lift and shift your technology from Amazon to Google." It's like pretty incredible to watch. It's also pretty incredible to see how the world gets to reap the rewards of this battle of the Titans.

[0:57:29.6] WM: Right. Yeah, the result of all these is pretty good for the world as a whole.

[0:57:36.1] JM: Yeah. William, I want to thank you for coming back on the show. It's been a real pleasure both time, both episodes we've had, it's been really entertaining. I look forward to seeing how Buoyant evolves and how Linkerd evolves.

[0:57:49.6] WM: Great. Thank you very much for having me, Jeff. It's always a real pleasure to be on here. You don't mind me saying at the end that Buoyant is hiring actively. If you want to come work on cool open source infrastructure stuff, send me an email, william@buoyant.io

[0:58:04.5] JM: There you go.

[0:58:06.0] WM: Yeah. All right.

[0:58:07.0] JM: William@buoyant.io. Okay, great. We'll put that in the show notes.

[0:58:09.9] WM: Thank you, Jeff.

[END OF INTERVIEW]

[0:58:13.0] JM: Thanks to Symphono for sponsoring Software Engineering Daily. Symphono is a custom engineering shop where senior engineers tackle big tech challenges while learning from each other. Check it out at symphono.com/sedaily. That's symphono.com/sedaily.

Thanks again to Symphono for being a sponsor of Software Engineering Daily for almost a year now. Your continued support allows us to deliver this content to the listeners on a regular basis.

[END]