EPISODE 965

[INTRODUCTION]

[0:00:00.3] JM: The container orchestration wars ended in 2016 with Kubernetes being the most popular open source tool for deploying and managing infrastructure. Since that time, most large enterprises have been implementing a “platform strategy” based around Kubernetes. A platform strategy is a plan for creating a consistent experience for software engineers working throughout an enterprise.

At most companies, a software engineer should be thinking about business logic, whether that logic is related to banking, insurance, oil and gas, or e-commerce. Today, engineers at many enterprises need to think about continuous delivery, application deployment, security policy management and other deeply technical problems that have nothing to do with the business that they are actually working at.

Kubernetes is a foundational open source building block that allows enterprises to base the rest of their infrastructure decisions around. Kubernetes has made it much more viable for enterprises to pursue a platform strategy. With widespread adoption of Kubernetes, there's a business opportunity for companies that can offer other platform solutions that build on top of Kubernetes. A service mesh is one such tool.

A service mesh provides networking and security features for all the services in an organization. The service mesh category is a large business opportunity, because it sits on the critical path of every network request that goes through an enterprise. Service mesh is a potential insertion point for lots of other products, including logging agents, distributed tracing, network packet scanning, security policy management and A/B testing.

The potential for business expansion from the basic idea of a service mesh is why so many businesses are entering the service mesh category today, from cloud providers to API gateways. There's lots of potential for upsells and cross-sells and enterprise sales expansion in the service mesh category.

Buoyant was one of the first companies to work on a service mesh tool with the Linkerd open source project that was based on the experience from Twitter. William Morgan is the CEO of Buoyant and he has been on the show several times before for some great discussions of modern application development and service proxying, service mesh. William returns to the show to discuss the competitive dynamics of the service mesh market.

A quick announcement, we are hiring a content writer and a operations lead for Software Engineering Daily. If you are interested in either of these roles and you like Software Engineering Daily, send me an e-mail, jeff@softwareengineeringdaily.com. Both of these roles are part-time positions working closely with myself and Erica. If you like the idea of that, then you can send me an e-mail. I'd love to hear from you.

[SPONSOR MESSAGE]

[0:03:01.7] JM: If you want to extract value from your data, it can be difficult, especially for non-technical, non-analyst users. As software builders, you have this unique opportunity to unlock the value of your data to users through your product, or your service.

Jaspersoft offers embeddable reports, dashboards and data visualizations that developers love. Give your users intuitive access to data in the ideal place for them to take action within your application. To check out a sample application with embedded analytics, go to softwareengineeringdaily.com/jaspersoft. You can find out how easy it is to embed reporting and analytics into your application.

Jaspersoft is great for admin dashboards, or for helping your customers make data-driven decisions within your product, because it's not just your company that wants analytics, it's also your customers. In a recent episode of Software Engineering Daily with TIBCO, we talked about visualizing data inside apps based on modern front-end libraries, like React, Angular and Vue.js. If you want to check out that episode, it's available on softwareengineeringdaily.com. You can also check out Jaspersoft for yourself by going to softwareengineeringdaily.com/jaspersoft and finding out about embedded analytics.

Thanks to TIBCO for being a sponsor of Software Engineering Daily.

[INTERVIEW]

[0:04:39.9] JM: William Morgan, welcome back to Software Engineering Daily.

[0:04:41.8] WM: Thanks, Jeff. It's great to be back.

[0:04:43.3] JM: I think this is your fourth or fifth appearance. You're closing in on the other highly numbered people. Over the last three years, we've talked several times about service proxies and service mesh. For many companies, the service mesh has gone from a curiosity to something that they're actively experimenting with. A growing number of companies, service mesh is actually playing a significant role in their infrastructure.

There are people who have actually deployed some service mesh and are really getting a lot of value out of it. For the companies that are beginning to experiment with it, beginning to roll it out, as they try to roll out a service mesh and make use of it, what are the biggest barriers to leveraging a service mesh and making it into something useful?

[0:05:37.0] WM: Yeah, yeah. That's a great question. You're right that it is happening and it's happening in a lot of companies. I think we've been privileged in the Linkerd world to have been exposed to it for the past couple years. Linkerd was the very first service mesh. As part of that, we've been in production in various companies for over three years.

We tend to take a pretty hands-on approach when people are adopting it, so we've seen what happens. There's a bunch of challenges, honestly. I think I even gave a KubeCon talk two years ago that was titled How to Get a Service Mesh into Production Without Getting Fired. Because there's a lot of ways for it to go wrong.

I think one of the challenges is just like any piece of technology that's relatively new when you add it into an existing system. Whenever something goes wrong in that system, that's the first – the service mesh is the first thing that anyone will blame, right? Something's not working now, well what do we just change? Oh, well. William added the service mesh three weeks ago, so this is probably Linkerd's fault.

As a result of that, we've actually built a lot of features into Linkerd around introspection and debunk ability of the mesh itself, because we need to give our adopters the ability to have this slightly defensive posture where they can say, “Oh, look. I actually understand what's happening here. I understand all the components. It's not the service mesh, it's actually the code over here has a bug, or something like that.” That's one challenge to adopting it. It’s just it’s new technology and you get blowback when things go wrong.

I think another challenge which is a little more specific to the service mesh itself is that the space is very confusing. Right now, there's a lot of different projects, there's a lot of information and misinformation that's out there. You get into these situations where people don't even know what the set of projects are, or whether this is the most appropriate one, or what the state of this thing is. Is it ready for production? Is it new? Is it somewhere in between?

There, I think the challenge is navigating that landscape and understanding what the state of these projects are today and where they're going to be in six to 12 months from now, or four to five years from now.

[0:07:59.6] JM: From talking to you, I have trouble understanding to what degree you think the choice of service mesh even matters. There's some set of functionality that you need out of a service mesh. Why does it matter which service mesh we choose?

[0:08:21.0] WM: Yeah. That's a good question. It is complicated by the fact that this landscape is so complex. There are projects to pick on network service mesh for a minute. There are projects like network service mesh which are great projects, but that are not anything like the other projects calling themselves service meshes and then operate on totally different part of the stack.

I think it matters, because these projects have very different surface areas. I'm obviously the most familiar with Linkerd. Linkerd takes a very particular approach, where we are quite Kubernetes-centric. Our goal is to make it so that when you run Linkerd, it imposes the smallest possible operational burden on top of Kubernetes itself. You've already adopted Kubernetes, it's a huge surface area. It's like a bunch of operational complexity you've just taken in. When you add Linkerd, it shouldn't double that. It shouldn't even add 5% more burden. It should be as small as possible.

That strategy has a bunch of implications for the sorts of features that we want Linkerd to have, the sorts of features we don't want it to have. It's not always the most appropriate service mesh. Of course, I'd like to say Linkerd is the best for any possible use case. I guess, there's too much engineer in me to really be able to say statements like that.

There are plenty of reasons why Linkerd wouldn't be the right choice for you, especially if you are not adopting a really Kubernetes-centric lifestyle. I think understanding what the benefits and strengths of each of these projects is and where they – we'll give you a sense of which one suits your particular use case the best. A lot of them have very similar value prompts, right? You're getting the same, at least at the broad – from a broad view, you're getting reliability features and security features and observability features, but the details being quite different between different projects.

[0:10:21.0] JM: Did you say Kubernetes-centric lifestyle?

[0:10:24.4] WM: Yeah, that's right.

[0:10:25.7] JM: That's a good one.

[0:10:26.6] WM: It's not just a container orchestrator. It's a lifestyle.

[0:10:29.4] JM: The surface area that this thing should occupy, that it should – the service mesh or the service proxy construct, it seems like you think it should be pretty small. It should just be this thing that's very easy to add to your infrastructure. Then of course as a company, that raises the question if all I'm providing is this tiny little thing that gets inserted into the infrastructure, where is the money being made?

[0:11:02.7] WM: Yeah. Well, so that's also a good question. I guess I'd say it's not so much at the service mesh is tiny from a line of code standard, although certainly, we try and minimize that. It's more from the operational standpoint. You can have things that are quite complex under the hood, but that aren't easy to operate, although it takes a lot of work to get there.

Yeah. I wear two hats. I think we talked about this last time. I wear the Linkerd hat, where I'm happy to be a maintainer, at least nominally. I wear the Buoyant hat, where I'm the CEO and we're a startup and obviously, we're venture backed. There has to be some strategy presumably, around Buoyant making money. Even from the Linkerd perspective, right? The reality of open source is that you want to have a really healthy economic engine behind the project that continues to invest in it, right?

Very few people are doing open source nights and weekends these days, right? There are companies behind these projects. If you care about the sustainability of the project, then you want there to be a healthy ecosystem of companies that are supporting it and nourishing it. For us, we finally have an answer to this question that I have had to dance around for the past couple of years and that we have just a few days ago, we've launched our SaaS offering, at least to private beta. It's called dive. Dive.co. If you go to the website today, you will see a wait list to sign up for dive.

The way we've partitioned this and I'm really happy with the way this worked out, because I think it's going to be really good for Linkerd, as well as obviously for Buoyant and for all of our adopters and users, is that Linkerd is extremely good at solving basically computer problems, right? Distributed systems, problems how do I have my services talk to each other in a way that's secure and reliable and that I can make sense of as an operator?

Dive solves a related, but very different class of problem, which is the problems of people and process. How do we as human beings on a team of engineers and some of us are in platform roles, some of us are in developer or service owner roles, how do we work together to accomplish a shared purpose, which is deploying code to production over and over again, so we can iterate on the feature set in a way that is viable and safe? Those are both parts of the same puzzle. We need to have the computers behaving, as well as the people behaving, but they're two very different domains.

[0:13:34.3] JM: Take me through the adoption of Linkerd as the service mesh, service proxy that we all know and love, through the adoption of dive. What is that user journey, or customer journey?

[0:13:55.2] WM: Yeah. In both cases, the sorts of problems that we want to solve are problems that SREs and platform owners have, right? These are folks who are adopting. In our case, we're adopting Kubernetes. They're building this internal platform for the rest of their team. They're picking a CICD system. They're picking a code repo. They're picking Kubernetes, or building up basically all the little bits and pieces of this internal platform.

They're thrust into this complicated situation, because what all of the technologies ultimately enable is building microservices, right? It's not super cool to talk about microservices this week. It was very cool several weeks ago and it'll be cool in the future, but right now it's not cool. That's really what – if you look at what Kubernetes and Docker and the service mesh all enable, it's all making it really easy to run a whole lot of services, right? You can agree or disagree, whether that's a good thing. I wrote a very long article about this recently on servicemesh.io. I wrote my Meshafesto, which is –

[0:15:05.2] JM: Did you really? I missed this.

[0:15:06.7] WM: Yeah. Well, it just came out last week. I scrambled to get it out before KubeCon. I tried to lay out everything. Okay, I'm going to back away from this tangent in a minute, I promise. I tried to lay out everything that I knew about the service mesh, including not just what it is and how it works, but why it makes sense and what else has changed in the world to make it a plausible thing, so you can get a lot more theories –

[0:15:28.6] JM: Wait. Why don't we go there right now? Let's come back to dive. Give me the Meshafesto compressed, as a daily audio version.

[0:15:36.9] WM: Yeah, okay. Yeah. The service mesh works by adding a whole lot of proxies everywhere, right? It's nice doing that. It's really nice in the sense that it turns out that's a great way to add functionality to a system without having to change the code, right? It turns out it's bad in the sense that we're running a whole lot of proxies, right? There's an operational burden to doing that. What's made this a feasible approach is basically the adoption of containers and Kubernetes, where now it doesn't matter what language the services are written in. I can just wrap everything up in a container.

The deploy time burden of deploying thousands or tens of thousands of user space proxies next to every application is handled for us by Kubernetes, right? That's not how things were five years ago, or 10 years ago. Imagine doing this with puppet scripts or whatever, which is how I grew up. It would have been a non-starter, right? I think what Kubernetes and Docker and the service mesh enable is this adoption of microservices.

I relate this to my experience at Twitter, because I was there 2010 to circa 2015 when Twitter went through this giant transformation from monolith to microservices. The amount of time and energy that Twitter had to spend on that transformation was staggering. It was a lot of engineers and a lot of time. It worked at the end, which is amazing, right? Especially since so few things worked at Twitter at that point in time. You can contrast that.

Twitter had to do that, because the monolith was not scaling anymore. It took a lot of time and an energy, but you can contrast that with startups today, where out of the gate, they're running 50 microservices and there's five engineers. That's a feasible thing for them to do. You may or may not agree that's the right thing to do, but it's feasible for them to operate that –

[0:17:29.0] JM: They're doing it.

[0:17:29.6] WM: Right. They're doing it either way. There's a really cool tweet from the Monzo folks recently. Monzo is a startup bank in the UK, where they showed off their topology of 1,500 services and the Internet lost their minds about why would you ever do this? To me, I was like, it's a little extreme, but it's not that crazy. They have a 150 engineers. That's a 10-to-1 microservice to engineer ratio. It's possible, whether or not you think it's a good idea. It is made possible by things like containers and container orchestrators.

[0:17:58.9] JM: And Linkerd.

[0:17:59.9] WM: And Linkerd. Well, in their case not Linkerd, but you know.

[0:18:02.6] JM: I thought Monzo used Linkerd.

[0:18:03.7] WM: In the olden days it did. Yeah. That was back when we were on the JVM. Yeah. As I scaled the JVM hits these performance limits and which is why we rewrote Linkerd off of the JVM. That's all to say the reason why the service mesh makes sense today is because you have these tools, like Kubernetes that allow you to co-deploy a whole lot of user space proxies in a way that's not totally insane. You have mechanisms for doing the IP tables wiring, so that all traffi c goes through there. It unlocks the abilities for these sidecar-based design patterns.

At the same time because you have so many microservices being built, because again, they're being unlocked by containers and container orchestrators, the need for the service mesh is also really increasing, right? Because you have all these developer teams and you've done all this great work to unblock them, so they're doing deploys on their own schedule. You've got service owner teams and they deploy their services whenever they need, and so you've got all these microservices.

Now at the platform level, there's a lot of functionality that you need, that you don't really want to incur a dependency on the developers for. If you want to adopt TLS throughout your entire application, your choices are well, I can ask every developer team and you might have hundreds of those, to add TLS to their application. Well, that's going to be painful. You're going to fight with the product manager, right? Or we can just do it with the service mesh. We can add Linkerd and Linkerd will do TLS, mutual TLS for us. In fact, we do it out. We turn it on by default. All of those factors are not only enabling the service measure, making it a feasible approach, but they're also driving the need for it.

[0:19:38.5] JM: Okay. I get the bull case for service mesh. It's out of the core of my codebase. It's going to give me some benefi ts. It's a nice abstraction layer away from the critical path of stuff that I'm actually manipulating, but the bear case is that it adds complexity to my infrastructure. I'm going to be spending time implementing this thing that I could be spending writing business logic. Are there any other negative sides to it? What are the biggest critiques of this abstraction?

[0:20:16.4] WM: Yeah. Definitely it adds complexity to your infrastructure. As much as we try and minimize it, it's not zero. It adds latency to everything. Right now, every call is not just going from A to B, it's going through to proxy hops, to user space proxy hops, each of which is introducing latency. It's a whole new concept for people to think about. There's definitely a cost that you're paying. With any layer of abstraction, you pay a computational cost to do this and you pay a operational cost. We try and minimize both of those in Linkerd as much as we can.

The benefit that you're getting, right? The benefit that you're getting is really a benefit that the platform team gets more than the developer team, which is the things that they are fundamentally responsible for, like what is the performance of the application as a whole, which services are healthy and which ones are unhealthy, regardless of who wrote them or what frameworks they used, or how they were deployed? All of that, the ownership of those solutions gets placed in the hands of the platform team.

In servicemesh.io, one of the statements I make which I really like is that the service mesh is less of a solution to a technical problem as it is a solution to a socio-technical problem. I'd like to see things through the lens of human beings, which is also my dive as a human focused, I guess. It is a solution for the platform engineers that gives them things that they could have in other ways, right? You could ask the developers to do all this stuff, but you give it to them in a way that decouples them from that dependency.

[SPONSOR MESSAGE]

[0:21:57.6] JM: As a programmer, you think in objects. With MongoDB, so does your database. MongoDB is the most popular document-based database built for modern application developers in the cloud era.

Millions of developers use MongoDB to power the world's most innovative products and services, from cryptocurrency to online gaming, IoT and more. Try out MongoDB today with Atlas, the global cloud database service that runs on AWS, Azure and Google Cloud. Configure, deploy and connect to your database in just a few minutes.

Check it out at MongoDB.com/Atlas. That's MongoDB.com/Atlas. Thank you to MongoDB for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[0:22:53.2] JM: Okay. We have talked about service mesh at length in other episodes and you've revisited this with your service mesh manifesto.

[0:23:04.3] WM: Meshafesto.

[0:23:05.0] JM: Meshafesto. Before we go into dive and start talking about that product, is there anything else you want to talk about, like ground floor service mesh ideas before we get into dive?

[0:23:17.7] WM: Well, I still remember the first question you asked me the previous time that we met, which is in Barcelona and I was super jet-lagged and I was not really prepared for this question, but you threw it at me and I had to stumble my way through it was is the service mesh a winner-takes-all market? The answer that I gave, which I actually do believe, so luckily I managed to stumble to the right place that it's actually not. It's not a winner-takes-all market.

I think as much as I would like to say that Linkerd is going to dominate the universe, I think the reality is there's going to be multiple service mesh implementations that are out there and they're going to have specific benefits and drawbacks. I think that's okay. I think that's fine. I think service mesh concept is really valuable and I think it's important for there to be implementations like Linkerd out there that are lightning fast and really easy to install.

We need to be able to say, “Oh, you want to try service mesh here, it'll take you 30 seconds to install Linkerd.” Linkerd doesn't have to be the be-all end-all of all service meshes. I think it would be bad for Linkerd, for us to try and do that anyways. I think that's a little surprising to people who are coming from the Kubernetes community where they've seen Kubernetes take over the universe and they're like, “Okay. Clearly this is a war and only one will survive.”

[0:24:32.9] JM: There were a lot of casualties in that war.

[0:24:34.4] WM: Yeah, there were.

[0:24:35.5] JM: It was bloody. Reporting from the frontlines, it was bloody.

[0:24:38.9] WM: I don't think it's going to be that way for the service mesh and I think that's fine.

[0:24:42.4] JM: Okay. Dive. You instrumented your infrastructure with a service mesh, you got your proxies set up, you got your control plane, you got mutual TLS, you're cruising along. What do you want next?

[0:24:57.6] WM: Yeah. Once again, it's replaying the experience that we had at Twitter, which was once we had gotten that stuff working and Twitter didn't have a service mesh as we know it, but it had an equivalent with Finagle, the big challenge that we had was we had done all this work to decouple the engineering teams, so that they didn't have to synchronize. We didn't have to have deploy Tuesdays where everyone got into the campfire room and we all tried to merge to the branches and to the monolith, but there were still these synchronization points that we needed that we didn't really have a mechanism for doing, right? Because services have relationships, right?

Well, I'm a caller, you're a callee, and so you doing something on your team, to your service actually will have an effect on me, depending on the nature of that relationship. We didn't really have mechanisms for dealing with that. A lot of what you did as a Twitter engineer was you interacted with other engineers in this service-centric manner, where you were saying okay, so you're a client of mine. How much traffic are you going to send me and when are you going to deploy? Actually, please don't do that right now, because my service is dying. Can you just hold off on making changes, because we're trying to do this? I don't even know who's sending traffic to my service right now. I don't even know that. Where's this traffic coming from?

Suddenly, we have 3X traffic to our service. I have to send an e-mail to everyone in engineering be like, “Hey, did someone change something recently?” All of those experiences, they're not technology problems, right? They're problems of how are the humans in this process communicating with each other? We didn't really have a good solution to that. We had sketches of solutions. By the time we had gotten to solve these computer problems, we were left in the I guess, scarier realm of well now, we've got the human problems left and we all wanted the same thing, right? We all had the same goal, which was we wanted to ship features for Twitter and make it better and iterate in a way that was fast, but we found ourselves stumbling, because of these communication issues that didn't have great solutions.

[0:27:00.0] JM: Dive, you showed me a demo of it, is a really nice looking management panel that allows for setting of SLO service level objectives. It provides a event history of all the changes that have occurred to your services, does a bunch of other stuff that I want to know about. Why do I want another management panel? Don't I have a million management panels and communication tools already? Why do I want another tool?

[0:27:32.6] WM: Yeah. Calling it a management panel makes it sound horrible. I would never call it that.

[0:27:36.8] JM: I'm sorry.

[0:27:38.1] WM: I call it Facebook for services. Facebook for microservices. Doesn't that sound exciting?

[0:27:41.6] JM: Yes.

[0:27:42.5] WM: Yeah. Yeah, that's something I want to use. It's like Uber for HTTP calls. That sounds exciting. Well, I don't call it that. The reason why, yeah, no one wants to look at another thing for sure, but what we've noticed is that every company that starts adopting microservices ends up building something like this in-house. It is really shocking –

[0:28:05.7] JM: UI panel from managing their services.

[0:28:07.8] WM: It's less about managing my services and it's more about what even – what services are even out there? I don't even know what's running in my infrastructure right now. Even the very act of giving you a list of services that are running and where they're running and what the – who owns this, like what team is, what team owns this thing and when was the last time it was deployed and what was that code, even the act of collating that information doesn't happen at a lot of companies.

You'll end up with companies that have wiki pages that are always out of date, or someone came around and made a spreadsheet once. By the time they finished compiling the spreadsheet, it was totally out of date. The thing that we've done and the thing that is made possible, that has made dive possible is Kubernetes and the service mesh.

Now that those things are in place, we actually can determine the majority of the stuff fully automatically. We can inspect Kubernetes, we can inspect the relationship between the deployments in Kubernetes. We can inspect the health of each of these endpoints and we can compile that.

What dive does fundamentally it sounds a little weird, but it's translating all of that computer stuff into social constructs, because a service when you're running microservices, a service is fundamentally a I guess, I'll call an organizational construct. It's a bunch of deployments on various Kubernetes cluster, sure, but it’s also a code repo, right? It's a team who owns this ideally.

Often, you'd have services that are un-owned. It's a readme and it's a run book and it's some operational knowledge. All that stuff just giving you a Facebook page for that thing. I'm going to keep calling this Facebook, until the legal department shuts me down. Just giving you –

[0:29:49.9] JM: Why not Twitter?

[0:29:53.2] WM: Yeah. I guess, we can call –

[0:29:53.4] JM: I guess Twitter for microservices is not – that doesn't have good recollections for –

[0:29:58.2] WM: It seems a little incestuous. Yeah. Just giving you a homepage for a service that someone else in the company can look at, and that's why it's extremely valuable. That's why SLOs are such a natural part of this, because I don't just want this to be well, here's readme.

[0:30:12.3] JM: Explain what SLO is.

[0:30:13.2] WM: Yeah. SLO is a service level objective. There's a whole lot of SRE theory around this. Basically, what it allows you to do is to express the reliability expectation of the service. What does it mean for this service to be healthy and what does it mean for it not to be healthy, right?

Because it's different for different services, right? In some cases, you may have a service – if it's up, that's great. If it goes down, it's not a three-alarm emergency. It's an autocomplete thing. Yeah, we should fix it, but no one's going to suffer. We're not going to drop revenue. No one's going to die if we don't have autocomplete, but there are other services that are like, “Oh, this is the database. If this thing goes down, everything goes down.” You have different reliability expectations.

An SLO is basically a way of formalizing that and saying, “Hey, this is what you can expect from this service,” right? It's a communications mechanism. There's cool things you can do with SLO, such as concept of error budgets, which give you a way of balancing feature velocity versus reliability and having a way of pushing back and saying no, we can't deploy a features right now, because we're out of our error budget. Fundamentally, they are a communication mechanism for saying here's the expectations of reliability for this service.

[0:31:33.0] JM: The thing that I think is cool about this product is I think when the service mesh wars, or starting to heat up, or people were starting to think about –

[0:31:43.2] WM: The service mesh, we should call them something else. Struggles.

[0:31:45.6] JM: The struggles. Yeah, okay. Fine, fine. All right. Enough, enough of the wartime commentary.

[0:31:50.1] WM: Service mesh, co-evolution.

[0:31:53.8] JM: Service mesh markets. Service mesh yeah, category. As the service mesh category was developing, I think people had an intuitive understanding that this was a high-value software category, because this is something that you were putting, you were attaching to all of your services. Stretching across your entire infrastructure. Clearly, there was some value there.

I think, even you intuitively knew that there was going to be some value there, but you didn't really know what is the value. What are you going to gain from instrumenting all your services? Obviously, you know you have this day one experience of circuit breaking and telemetry and TLS, whatever. Just the fact that you now would have this piece of infrastructure that's sitting everywhere across all your services, you knew there was some value there. This is the first foray into what else can we do now that all of our infrastructure is instrumented with this uniform layer?

[0:33:02.4] WM: Yeah. Yeah, that's right. That's right. I gave this incredibly inspirational talk at ServiceMeshCon on Monday.

[0:33:09.1] JM: Was that your own conference?

[0:33:11.0] WM: No. It wasn't my own conference, but we were there with a good number of – good amount of Linkerd content. The talk that I gave was basically does the service mesh matter, right? My assertion was that it matters yes, but it matters in the same way that plumbing matters, right? Plumbing is great, it allows us to build cities and to live together and in a way that's hygienic and people aren't getting sick all the time. Once you have that, then there's all sorts of great economic and cultural benefits from living in cities.

The pipes and the tubes or whatever, that's not the important part. That's cool too a very small audience of people who really end up pipes and tubes, but plumbing is important not because of what it is, but because of what it enables. I think it's exactly the same for the service mesh, which is look, I get caught up in the details and I'm like, "Oh, look. We've got especially weighted moving average, load balancing algorithms and whatever." That stuff doesn't really matter. That's the tubes and the plumbing. What is interesting to me and what I think is a real value of the service mesh is not what it is or how it works, but what it unlocks.

That's what's so exciting to me about dive is now that we have Kubernetes and the service mesh and obviously, we're living in the very bleeding edge of the software adoption cycle here. It's not we have these things in any global sense. Now that those are things that are there and are starting to see adoption, where I'm most excited about and this has been the mission for Buoyant the entire time is what does this unlock and how do we start solving the problems that I actually care the most about, which are the problems that human beings have when trying to accomplish some shared purpose together?

I'm a weirdo in the infrastructure world and that what I care about is the people aspect, right? This feels very natural to me, but I think it seems a little strange if you're coming at it from the what's the ideal load balancing algorithm perspective.

[0:35:06.0] JM: Well, if you asked a lot of the microservices theorists in the world, they'll tell you that microservices is a answer to a human problem. You break up the thing into microservices, because you need that for managing a number of people you have, allocating the number of people you have to the right number of services.

[0:35:25.1] WM: Yeah, that's right. That's right. You get things like Conway's law, which are explicitly relating software composition and organizational composition. I'm not the only one in the world who understands this. It's just the part of it that I think is really, really interesting.

[0:35:39.0] JM: As a side note, is there any risk that service mesh is one of these things that we're going down a blind alley as an entire industry, like chatbots? We all got really hyped about chatbots. Then it turned out like no, chatbots were just a Silicon Valley fever dream. No, the world is not really going to adopt chatbot. I mean, they are adopting chatbots, but it's happening much slower. What if service mesh happens much slower and then there's some other pivotal change to the infrastructure world that makes service mesh look like a total foray, or a blind alley? Is there any risk to this thing happening?

[0:36:16.7] WM: I don't think so. I don't think so. I think that's a great question to ask ourselves at any point in time, because as you point out, we do as an industry go down these blind spots, especially when there's the fuel of the cheap calories of VC funding, which I say that –

[0:36:35.5] JM: Having [inaudible 0:36:35.5].

[0:36:36.4] WM: Yes. Having feasted heavily on this utility. Actually arguably, that's great for us, because we can explore those things really quickly. You’re going to be like, “All right, that didn't work out.” Everyone knows the deal what you're getting into. No one expects us to be like, success guaranteed, except for Buoyant and Linkerd where success is guaranteed, or your money back. Oh, no. I shouldn't say that. For Linkerd maybe.

The reason why I don't think that's the case for the service mesh is it's actually the same reason that I think it's not the case for container orchestrators and for all that stuff, which is I think it's fundamentally tied to what are the demands that we are placing on our software, right? I think those demands are changing, right?

10, 20 years, 10 years ago let's say, it was okay for software to be down for 24 hours for maintenance, right? Maybe it was 15 years ago. I don't know. Things weren't moving that fast and the Internet was only so big, there weren't that many people on it relative to now. All of those factors are changing, right? Today in the modern world, the Internet is not getting any smaller. It's getting a lot bigger, a lot faster. The software is not allowed to be down. It needs to be up 24/7. The pace of feature iteration is really, really fast.

All of those demands have had some pretty transformative results in the industry. That's part of I believe, why even moving onto the cloud, right? Is fundamentally I think due to those demands. Once you're in that world, it turns out that the way you're architecting software has to be pretty different. That's why you end up with things microservices, right? Then once you're in that world, well okay, all the deploy in operational components start getting really complicated and therefore, you have containers and container orchestrators. I think it's all tied to these fundamental demands that we’re placing, changes in the demands that we’re replacing on our software. I think the service mesh is in service of those demands.

By that metric, I don't think it's a mistake. Certainly, what we've seen with Linkerd and even what we saw with the prototype forms of the service mesh, like companies like Twitter and Facebook and Google early on, is that it provides real value. It's actually making people's lives easier and it's allowing them to do stuff in a way that they couldn't do before.

[0:39:04.6] JM: Now it doesn't surprise me that a service mesh can provide value at a company like Twitter, or Facebook, or Google, or Uber. These companies have fleets of really good engineers. They have central platform teams. I know that insurance companies and banks have their own platform teams these days. I know John Deere Tractor probably has a platform team as well and all these companies that have a lot of money to spend that are becoming software companies, they all are developing this platform-based strategy, where they platform teams. They have some determined strategy for adopting new technology and propagating that technology through the organization.

I do wonder, is there a difference in the quality of platform engineering between Twitter and a insurance company? Is the insurance company, I know they can set up something called a platform team. Do they actually have the wherewithal and the technical skill to get a service mesh and a Kubernetes installation deployed, such that it's actually providing them value?

[0:40:18.2] WM: I think the answer is probably yes. I think this is a really important question. My co-founder, Oliver Gould, hates the term ‘democratization’, but I'm going to use it with apologies to him. I think a lot of what these technologies do or can do is democratize access to the ability to write software that's reliable and secure.

I think that's really important, because if we don't have that, then only the companies that have access to really, really good software engineering talent, which are going to be Facebooks of the world, not the John Deeres of the world. Well sorry, I don't know anyone in John Deere, so not the – you know who I –

[0:41:02.2] JM: I don’t either. By the way, I would love to do a show with John Deere.

[0:41:03.9] WM: Yeah. John Deere is probably awesome.

[0:41:05.1] JM: I've heard they're crushing it. I mean –

[0:41:06.9] WM: You know who I hate though who I'm totally going to shit-talk is Equifax.

[0:41:09.7] JM: There you go. Okay. Equifax we need to give better software, we needed – Do we really want to democratize? We just want Equifax to die at this point, right?

[0:41:16.8] WM: Well, so what I would like to do is make it so that even companies like Equifax can build software reliably and securely, right? They stop having those incidents.

[0:41:26.3] JM: You're not vindictive.

[0:41:27.4] WM: No. I would like to make things better for them. I think that Kubernetes and Linkerd can play a role in that, right? One of the things –

[0:41:33.8] JM: It's funny, because literally the last interview I made an Apache Struts joke, which is the downfall of Equifax.

[0:41:41.1] WM: Oh, was it Apache Struts?

[0:41:42.8] JM: I think so. Yeah.

[0:41:44.2] WM: I heard it was because they didn't have mutual TLS in between their services.

[0:41:47.1] JM: Well, I thought it was – Okay. Maybe. It might have been.

[0:41:49.8] WM: No, I don't know. It's definitely not that.

[0:41:53.9] JM: Oh, I get the joke.

[0:41:55.7] WM: Yeah. I think security is a great example of this. Having a secured system is really, really hard, right? There are things you can do to make it easier. For example with Linkerd, we do mutual TLS between all services, right? Which is great. Now you get encryption and you get authentication of the identity of the service and so on. We actually turn it on by default. Zero confi g on by default, mutual TLS. Part of the reason why we walked down that path –

[0:42:24.7] JM: Falling into success.

[0:42:26.0] WM: Yeah, exactly. Is to the extent that security is complicated or hard, you have to confi gure some stuff, it's not going to be used. If we have the opportunity to do that, we can make Linkerd provide as much as it can, can provide security by default. It's a lot more to a security story than just adding a service mesh just adding mutual TLS, of course. The parts that we can control, I think it's really important for us to make that basically zero work, right? We talked about this last time? I forget, because I've told the story to many people.

When I when I'm using my web browser and I'm surfing cat pictures on Reddit or whatever, I get the green lock icon, right? I've got a secure connection to Reddit and I've validated that it's Reddit and all that stuff just for my cat pictures. I don't have to do any work to get that. I've done literally zero work to do that, other than type in reddit.com.

It should be the same way for my services. Why should my services be any different, right? They're passing credit card numbers around, or health data. Having that security by default, I think it's not just cool to talk about. I think it's really important, because there are going to be companies of the future, the Equifaxes of the future that are going to be built, that aren't necessarily going to have access to the top tier, very expensive talent that companies like Facebook and Twitter have, but it's important for them to be able to build secure and reliable software. Because if they don't, we're all going to suffer, right?

The thing that blew my mind about Equifax was that the information of my wife and my friends was leaked and they didn't even have a choice. They weren't even Equifax customers. It's just like – it's gone and leaked, right? We've got to do what we can to prevent that. We've got to be able to, I'm going to use that word again, democratize as much as we can, reliability and security in software. Otherwise, we're going to all end up in a world that's just more Equifax.

[SPONSOR MESSAGE]

[0:44:22.6] JM: Being on-call is hard, but having the right tools for the job can make it easier. When you wake up in the middle of the night to troubleshoot the database, you should be able to have the database monitoring information right in front of you. When you're out to dinner and your phone buzzes because your entire application is down, you should be able to easily find out who pushed code most recently, so that you can contact them and find out how to troubleshoot the issue.

VictorOps is a collaborative incident response tool. VictorOps brings your monitoring data and your collaboration tools into one place, so that you can fix issues more quickly and reduce the pain of on-call. Go to victorops/sedaily and get a free t-shirt when you try out VictorOps. It's not just any t-shirt, it's an on-call shirt. When you're on-call, your tools should make the experience as good as possible. These tools include a comfortable t-shirt. If you visit victorops.com/sedaily and try out VictorOps, you can get that comfortable t-shirt.

VictorOps integrates with all of your services; Slack, Splunk, CloudWatch, Datadog, New Relic. Over time, VictorOps improves and delivers more value to you through machine learning. If you want to hear about how VictorOps works, you can listen to our episode with Chris Riley. VictorOps is a collaborative incident response tool. You can learn more about it, as well as get a free t-shirt when you check it out at victorops.com/sedaily.

Thanks for listening and thanks to VictorOps for being a sponsor.

[INTERVIEW CONTINUED]

[0:46:13.6] JM: The TLS example, it is worth repeatedly bringing up, because as I have talked to the other service mesh people, what they repeatedly say is yes, enterprises do want a service mesh. Yes, they are excited about all the things, the routing and the telemetry and A/B testing and the green-blue deployments and whatever. What they really want the most is TLS. The way to get that is well, one way to get it is through a service mesh.

I think the brightest vision of what that looks like from where we stand today is you also have some way of – I mean, if you look at the spiffy and the spire projects, you have some way of giving application identities to your different services, so that the security operator can hand out credentials, can hand out security credentials to those different applications and understand the security posture of the organization through the service mesh management plane. Is that one of the things that dive is going to do, like deal with this security layer?

[0:47:32.1] WM: Probably not. Probably not. Dive is really focused on the problems of people and process. I think there's interesting security stuff we can do with dive, but they're probably not going to be in the realm of well, I need to tie together service identity across all these different types of infrastructure that we have, because we've accumulated these strata of technologies over time. To me, that's a problem that's best solved by open source, that's best solved by things that are running in your cluster. It doesn't seem like a dive thing to me.

[0:48:05.0] JM: Do you have to do anything fancy to – so I talked to the Tetrate people recently. This is one thing they said they're really focused on is building infrastructure to allow the management of this TLS credentialing stuff. Do you want some management planner? There's a Linkerd management plan also. Is that someplace where you can insert the security management policy stuff?

[0:48:33.4] WM: Yes. Yeah, yeah. Linkerd has a control plane and control plane has a dashboard and so on and so forth. That's all built into – that all runs on your cluster a little namespace. That's potentially a place where you would start configuring how identity, how service identities is provisioned for all your services.

Yeah, so that is certainly something that can happen in an open source land and certainly something that – Linkerd today, basically what we do is the control plane ships with little certificate authority and that certificate authority issues certs to each of the data plane proxies and rotates some every N hours and ties them to Kubernetes service accounts. We can make that work with spiffy. It's on the roadmap, just haven't gotten to it quite yet. To me, yeah, all that stuff is important and it needs to be managed. I would rather have that happen in open source land, than have it be a commercial product.

[0:49:30.8] JM: What else do you need beyond what you just described, beyond the automatic certificate rolling and issuance? Do you need a manual thing for –

[0:49:43.7] WM: The story is simple in the Kubernetes world, because he can just do this all automatically. What gets a little more complicated is when you want to have identity outside of a single Kubernetes cluster, especially identity where you have code that you have running on VMs, or on Mesos, or in other environments. That identity needs to be shared across all those things. That's where projects like spiffy come into play. Yeah.

We've taken, we've done the very easy part with Linkerd, which is if you're running on Kubernetes, we will give you everything you need for that Kubernetes cluster for free on by default. The moment that you extend beyond that cluster, then life gets a lot more complicated.

[0:50:22.6] JM: What does application identity actually mean in that context?

[0:50:25.3] WM: In this context, it's distinct from something like user identity. It's not like, “Oh, Jeff is doing this particular thing.” This is the foo service and I'm the bar service and both sides know that you are foo and I am bar. Therefore, I can trust that you're allowed to call me, or maybe you're not allowed to call me. There's the whole policy aspect in there too, right? Or foo has several different versions and you're actually the foo from staging, but I'm the bar from production, so maybe that's allowed, or maybe it's not. Yeah.

[0:50:58.4] JM: Okay. This is not hard to do in the all-Kubernetes world. It's difficult to do in a world where you've got a Mesos cluster and a VMware thing and a bare metal box with no Kubernetes, or VMs or anything. I’ve got a heterogeneous infrastructure and you're trying to have this policy management where you don't want staging talking to production, for example. Linkerd, you've focused on the Kubernetes side of the market anyway, so it almost doesn't make sense for you to work on a application identity policy management system quite yet, I guess?

[0:51:47.2] WM: Well, I would very much like to leave that all to spiffy and to just work with spiffy, right? Part of the Linkerd philosophy is to work with existing projects, which is why we don't have ingress, because there's already a 100 Kubernetes ingress projects, each of which are good in different ways. For something like that, or even policy talking about how do you defi ne this policy. There's projects like OPA out there that have – basically have solutions for that.

[0:52:12.0] JM: Right. Open policy agent.

[0:52:13.2] WM: Right. Yup. Yup. I don't want to invent any of that in Linkerd. I want Linkerd to live alongside those projects. Yeah, in my mind, it's following the Kubernetes philosophy of having these building blocks.

[0:52:25.9] JM: Okay. We've been talking about what dive is not. Let's talk about what dive is.

[0:52:31.6] WM: Yeah. Yeah. I think policy is an interesting example to talk about, because dive does have a policy component, but its policy in terms of these service constructs and in terms of the teams and the humans who operate that. The example that I like to give is let's say that you are a service owner, right? You're deploying, you want to roll out a new version of code to your service, right? That's all part of a day's work. There's teams that are doing 30 deploys a day, but your services actually has a client that's out of SLO, right?

There's another team that owns another service and right now their service is out of SLO, which means they're struggling to get things back up. You doing a deployment right now, introduces risk into the system. You might not actually want to do that. What dive can do is allow you to define policies that say things like hey, if you have a client that's out of SLO, then before you're allowed to deploy, you need to get approval from some team member there, or something like that. You can imagine variants of this.

In that case, dive can actually send a little Slack notification, or an e-mail to an owner of that upstream service and say, "Hey, this dependency of yours wants to deploy. Are you cool with that? Yes, or no." That'll at least spark a conversation.

That's the policy you can start capturing that involves – it's not policy of is foo allowed to talk to bar? It's more along the lines of, "Hey, this change is happening and these people are potentially going to be affected and do you want to get an explicit buy-in from them or not?" Maybe that's approval, maybe it's just notification. I don't know. These are all organizational things. Those are the kinds of synchronization points that have to happen between engineers when they're operating and building these services.

[0:54:13.1] JM: What are they doing today? They're just sending a Slack message?

[0:54:15.2] WM: Yeah, yeah. That's right.

[0:54:16.6] JM: Or not even notifying, or –

[0:54:18.3] WM: Yeah. A lot of the time, they're just not doing it and then things break and you get yelled at afterwards and you have a post-mortem, or they have internal tools that they've built.

[0:54:28.8] JM: Doesn't this thing already take place through github? If I'm going to make a change that is going to affect something, isn't that usually centered around github?

[0:54:39.4] WM: Github does have this notion of code owners and things like that, where it's hey, if you're making a code change to the service, you need to get N code reviews before it can be merged in. That's really a code-centric thing. That's more about what code is allowed into the service and it's less about can I deploy this code to this environment given all these runtime dependencies? Or that information is not captured in github. In fact, it's not even really captured anywhere around the service mesh.

[0:55:10.1] JM: Define what a runtime dependency is.

[0:55:12.7] WM: In this sense, I mean, who's calling bar, right? That's often not captured in the code. It might be captured – or if it is, it's captured on foo side and not on bar side. It turns out that's a really important part of the runtime behavior of the system, is this runtime relationship. Even capturing that is difficult.

We had all these incidents at Twitter that were due to the fact that someone introduced a new dependency and no one told the team that was going to happen, or that things went awry because that relationship was not ever defined in the code, you could go around and write it in a spreadsheet or something, but that spreadsheet would be out of date as soon as someone changed something. The runtime relationship between services is a critical part of the applications behavior, but it's not formalized anywhere, right? It's not captured anywhere.

[0:56:00.7] JM: This is the service dependency chain. A depends on B, depends on C, depends on D.

[0:56:05.9] WM: Yup. Yup. Of course, that dependency is not just a binary thing, right? It’s this very complex here are the calls that are making, here's the volume that you can expect. Maybe you have rate limits on your side. I've got some retry policy on my side. It's not just I talk to Jeff, it's Jeff and I are having a conversation about this topic on this particular basis and so on.
If you look at very sophisticated organizations, they eventually get to the point where you can't even introduce a relationship without formalizing that contract. This is a problem and people solve it through these homegrown tools, which is fine. To me, that's a sign that dive has a – there's a market.

[0:56:48.9] JM: Do you have any specific examples? Does Netflix have one of these things? Google has one of these things? Facebook has one of these things where it's the mappings of the runtime service relationships?

[0:57:00.6] WM: There are some examples, but none that I think I'm allowed to talk about weekly.

[0:57:04.7] JM: Okay. Fair enough. Fair enough.

[0:57:06.5] WM: You can certainly see –

[0:57:07.6] JM: This is somebody's infrastructure for everyone.

[0:57:10.0] WM: Yeah. Yeah, that's right.

[0:57:12.0] JM: Enough said.

[0:57:12.4] WM: Yeah.

[0:57:12.9] JM: I mean, I look at it, I'm like, this looks like something that has been built at one of these gigantic organizations. This looks like something that people could use.

[0:57:19.9] WM: Yeah. These problems definitely exists and people are trying to solve them. Yeah.

[0:57:25.1] JM: As we wrap up, a little bit about business going to market stuff. The different service meshes all have a different market position. I look at App Mesh from AWS. Obviously, it's AWS. They didn't even need to have an open source thing. They're just like, "Here's our thing. It's proprietary. You're going to take it. You're going to like it. You're going to use it."

Kong has their north-south traffic manager that they can upsell into an east-west traffic manager. HashiCorp has a suite of products that have great reputation and they can cross-sell into console service mesh. These are all nice little entry points for all of these "service mesh products." How does your entry point compare? Because I assume there are plenty of organizations that have one of these other products that you're talking to and you're competing for the same land grab. Does that ever come up?

[0:58:33.7] WM: I think that's related to what we talked about a little bit earlier around the service mesh not having a single, not a winner-takes-all space. For me, what's important is not – like I said, not Linkerd dominating the universe, although that would be nice. What's important is that the service mesh model as a platform that other tooling can be built on top of takes off and it's a viable thing. That's what I think we're starting to see.

Regardless of the actual implementations of these projects, there being a service mesh there, just like there being a container orchestrator there that has an API that we can talk to, that we can use to understand what's actually happening on that cluster, that's the important part for Buoyant. That's what everything, all of our go-to market is predicated on.

In fact, if you look at the mission statement of Buoyant, it doesn't say anything about Linkerd, it doesn't say anything about the service mesh at all, right? It's all about hey, first of all, people are

– the way that software is being built is changing. Also, we need to provide mechanisms for the world to run software in a way that's reliable and secure, because if we don't, then we're going to end up in a world that we don't like. That's been the mission. The service mesh is a implementation detail by that view, right?

That's why things like dive I think are very much in-line with that vision, where it almost doesn't matter. The fact that there's Kubernetes and the fact that there's Linkerd under the hood, well that's great, but those are still implementation details. The real challenge that we want to solve and the real thing that we have to solve is how does a group of human beings, a group of engineers get together and accomplish a shared purpose? That's fundamentally a business problem. Maybe it's more general than that, but at least in this context it's a business problem. That's what I want to solve and that's what I think we can solve with dive.

[1:00:28.8] JM: Last question, so let's say there's somebody in the audience who is building a infrastructure product for developers, what are the most crucial lessons that you've learned about building infrastructure software, infrastructure companies over the last three, four years?

[1:00:48.5] WM: Yeah. Oh, man. Where to begin? We've learned so many lessons. I think the first is you need to have a clear idea of who your adopter is, right? Is it an SRE? Is it a developer? Is it someone who's writing business logic, or is it someone who's building a platform? For us, it's actually the platform owners. They have a particular relationship with the developers, but they're not actually building the business logic. Having a really crisp idea of who they are and what they care about.

I think the next lesson that we learned is not getting too caught up in the technology and doing everything you can to understand what are the actual problems that someone has to solve, because the reality is there are a large set of problems that are secondary. They're nice-to-haves, right? They're things that yes, I'd like to have cloud portability. Sounds good, right? Yeah, of course you want that, right?

There is also a set of problems that are if I don't solve this right now, I'm literally not going to have a job tomorrow. You need to understand, you need to find those problems and you need to solve those and as immediate in a directed way as possible. That's a big one for us.

Then the third one maybe is that at least in the open source world, it's almost the consumer world, in the sense that you are vying for people's attention, right? There's a lot of stuff out there. People have a limited amount of time in the day and unlimited amount of shits to give about some new thing that they've never heard of, right? You need to make sure that when they look at your website, when they try the thing, whatever. If they're going to give you 30 seconds, they need to get to like, “Oh, wow.” In 30 seconds. It needs to be clear to them that it's worth spending more time on there.

We've invested heavily. We learned this lesson a lot of ways in the transition from Linkerd 1.X to 2.X, where 1.X was like, “Here's a giant YAML file. It configure every possible option for Finagle, so go figure it out.” In 2.X, by contrast has an onboarding experience that we've really focused really hard on. We try and make it, so that you can install service mesh on Kubernetes in 60 seconds and get a dashboard that shows you a bunch of cool stuff that you didn't have before. That thing is really important from the product perspective, because product is what's communicating everything else.

I mean, your product has to communicate itself to the user. Say those first 30, 60 seconds, 5 minutes, whatever it is are really crucial for that. That's not natural for engineers, right? You think about the grand vision, you think about once we have this deployed everywhere, imagine all these cool things that we can do. You're not going to get to that point, unless you have the first 30, 60, 300 seconds really nailed down. Those are all lessons learned the hard way and hopefully incorporated mostly into Linkerd at this point.

[1:03:43.2] JM: Cool. Thanks, William. I'll see you at the next KubeCon.

[1:03:46.0] WM: Sounds good. Thank you.

[1:03:46.4] JM: If not before then.

[1:03:47.3] WM: Yeah. Sounds great. Thanks, Jeff.

[1:03:48.6] JM: Great.

[END OF INTERVIEW]

[1:03:58.6] JM: DigitalOcean is a simple, developer-friendly cloud platform. DigitalOcean is optimized to make managing and scaling applications easy, with an intuitive API, multiple storage options, integrated firewalls, load balancers and more. With predictable pricing and flexible configurations and world-class customer support, you'll get access to all the infrastructure services you need to grow.

DigitalOcean is simple. If you don't need the complexity of the complex cloud providers, try out DigitalOcean with their simple interface and their great customer support. Plus they've got 2,000 plus tutorials to help you stay up to date with the latest open source software and languages and frameworks.

You can get started on DigitalOcean for free at do.co/sedaily. One thing that makes DigitalOcean special is they're really interested in long-term developer productivity. I remember one particular example of this when I found a tutorial on DigitalOcean about how to get started on a different cloud provider. I thought that really stood for a sense of confidence and an attention to just getting developers off the ground faster. They've continued to do that with DigitalOcean today. All their services are easy to use and have simple interfaces.

Try it out at do.co/sedaily. That’s do.co/sedaily. You will get started for free, with some free credits. Thanks to DigitalOcean for being a sponsor of Software Engineering Daily.

[END]