EPISODE 735

[INTRODUCTION]

[00:00:00] JM: Software products are distributed across more and more servers as they grow. With the proliferation of cloud providers like AWS, these large infrastructure deployments have become much easier to create, and with the maturity of Kubernetes, these distributed applications are more reliable. Developers and operators can use a service mesh to manage the interactions between services across this distributed application.

A service mesh is a layer across a distributed microservices application that consists of several service proxy sidecars. Each service proxy sidecar runs alongside a service within that cluster and there's also a central control plane component of these service mesh, and the central control plane communicates with those sidecar proxies. A service mesh has many uses. Every request and response within the application gets routed through the service proxy that's associated with that service and that can improve observability, it can improve traffic control to different instances of the service and it can also introduce circuit breaking in case of an instance failure.

The central control plane can be used to manage network policy throughout the entire system. So if you want to impose certain security constraints, certain authorization patterns, you can do that with this centralized service mesh control plane, and we've done shows about each of the different components of a service mesh system including different types of service proxies, like Envoy. We've done a show on Kong that's coming up, and also we've talked a little bit about nginx. We've also talked about the service meshes built on top of these proxies. Linkerd, which is made by the startup Buoyant, was the first service mesh product to come to market and it has the most production use at least in terms of public success stories with customers like Expedia and Monzo Bank. Istio is a more recent service mesh which uses the Envoy service proxy.

Istio came out of Google and it's also supported by IBM, and this sets up a classic competition between a startup, Buoyant, and the large encumbrance that are trying to change the mindshare and introduce their competing product against that startup with a bit of a head start. It's an interesting competitive story, the service mesh landscape, and William Morgan is the

CEO of Buoyant. He joins the show today to talk about the use cases and the adaption of the service mesh. He also talks about the business landscape of the service mesh category, and more generally how to compete with giant cloud providers.

I want to mention that we have a survey up and we would really love your feedback. You can go to softwareengineeringdaily.com/survey and give us your feedback. Please tell us what we're doing wrong, and also if you feel like, tell us what we're doing right. We would love to grow the listenership and we would also love to make for a better listening experience for you. If you also want to sign up for our newsletter, you can do that by going to softwareengineeringdaily.com and clicking newsletter in the upper right. We have a weekly newsletter that goes out with some episodes and some content that we've been reading.

So with that, let's get on with this episode.

[SPONSOR MESSAGE]

[00:03:49] JM: Managed cloud services save developers time and effort. Why would you build your own logging platform, or CMS, or authentication service yourself when a managed tool or API can solve the problem for you? But how do you find the right services to integrate? How do you learn to stich them together? How do you manage credentials within your teams or your products?

Manifold makes your life easier by providing a single workflow to organize your services, connect your integrations and share them with your team. You can discover the best services for your projects in the manifold marketplace or bring your own and manage them all in one dashboard. With services covering authentication, messaging, monitoring, CMS and more, Manifold will keep you on the cutting-edge so you can focus on building your project rather than focusing on problems that have already been solved. I'm a fan of Manifold because it pushes the developer to a higher level of abstraction, which I think can be really productive for allowing you to build and leverage your creativity faster.

Once you have the services that you need, you can delivery your configuration to any environment, you can deploy on any cloud, and Manifold is completely free to use. If you head

over to manifold.co/sedaily, you will get a coupon code for $10, which you can use to try out any service on the Manifold marketplace.

Thanks to Manifold for being a sponsor of Software Engineering Daily, and check out manifold.co/sedaily. Get your $10 credit, shop around, look for cool services that you can use in your next product, or project. There is a lot of stuff there, and $10 can take you a long way to trying a lot of different services. Go to manifold.co/sedaily and shop around for tools to be creative.

Thanks again to Manifold.

[INTERVIEW]

[00:06:03] JM: William Morgan, you're the CEO at Buoyant. Welcome to Software Engineering Daily.

[00:06:08] WM: Thanks, Jeff. It is a pleasure to be back.

[00:06:10] JM: Indeed! We've done several shows over the last few years. You were very early to seeing the use of a service mesh or a service proxy layer from your time at Twitter. People can go back and listen to those previous episodes. We should do a brief refresher on the purposes of a service mesh before we get into the contemporary subjects. Explain what a service mesh is.

[00:06:34] WM: Sure. A service mesh is a layer of infrastructure that at its heart is managing the communication that happens between microservices. The idea is that you've deployed a lot of microservices, and typically this is done in a cloud native stack. So everything is containerized with Docker and you've got a container orchestrator like Kubernetes that's managing this pool of hardware and deploying the container for you. The service mesh sits in between each of the services, and as the services communicate, so A will talk to B, talk to C, talk to D, talk to a database. These chains can get quit long. Service mesh will handle a set of things for you. It will handle things, a set of reliability features. It will handle a set of security features and then they'll

handle a set of observability of feasibility features for how the traffic flows between those services.

[00:07:34] JM: So let's say there's some kind of high-level request to the surface of my application. Take me through the lifecycle of a request without a service mesh versus with a service mesh.

[00:07:46] WM: Sure. In both cases, a request will come in. Let's say this is an API that you're serving, and I used to work at Twitter, so I'm very familiar with the Twitter API. So you're using cellphone and you're like, "Okay, show me the latest tweets," and your Twitter client on your phone will make a call to trigger API and that will come in through – There's a bunch of stuff that happens at the edge to actually get that traffic. But then once it's inside the Twitter kind of application on the server side, it will start going through a series of basically RPC calls which can be done in a variety of ways, a very common pattern. This is just to use HTTP, although if the newer fancier world is moving a bit to GRPC.

You'll have some service that will take that request from the client and it will make a call to another service. That service will make a third call, a fourth call, and the entire time the client is waiting. So my phone is waiting for a response, and if that response doesn't come back in some number of second to milliseconds, then my client will retry and eventually it will fail. Internally you'll have that same behavior too, where service A talks to service B and it will say, "Okay, I need a response. If B doesn't respond in time, then that's considered a failure and maybe retry or maybe you don't."

Your question was, with a service mesh versus without a service mesh. So in both cases, that communication has to happen. The big difference is with the service mesh, the way it works is you insert these little proxies in between each of those calls. The proxies act as these kind of out of process network stacks. The reason why we do that rather than building a client library is because it allows us to insert that logic across our application without having to worry about what language the service is written in or what framework it's using or anything like that. In fact, in these bigger applications, you'll often have services that are written in different languages. So the service mesh gives you this uniform layer of reliability and visibility and security and all the features independent of how the application itself is constructed.

[00:09:56] JM: If you have lots of services and several instance for each service, your service mesh can route intelligently. It can fight the right instance to server request. It can do the circuit breaking pattern. Explain how the service mesh has the data and the instrumentation to do all these complex routing.

[00:10:18] WM: Yeah. A lot of the kind of goal of the service mesh is to insert these proxies in a way where the application doesn't really care or doesn't really know. The application thinks that service A is just talking to service B. Through either IP table's magic or through some other techniques, we wire that communication through the proxies and the proxies are configured to understand what it takes to actually deliver that request reliably. A talking to B, it will go through a proxy. The service mesh proxy will measure how long it takes to get there. It will kind of know what the destination is. So it will integrate in a system like Kubernetes. It understands enough of the Kubernetes API to say, "Okay, service A wants to talk to service B. B is these 50 pods that are on these different IP address and port pairs and I'm going to pick one to send it to based on a bunch of logic around how reliable things are or maybe how fast they are. If there's TLS involved, then I'm going to encrypt on the side and then I'm going to decrypt on the other side."

It gets that information either from the metadata of the request or from what it knows around about the environment, so it understands enough about Kubernetes to be able to do that routing in the way that makes sense.

[00:11:34] JM: You've been working on Buoyant for three years, four years?

[00:11:39] WM: About three and a half.

[00:11:41] JM: How is your perspective on what a service mesh or a service proxy, this service layer that we're talking about, how has that evolved since you started the company and you've seen more customer use cases, you've seen the evolution of different infrastructure patterns. What's changed?

[00:11:58] WM: Oh my gosh! So much has changed. When we started working on Linkerd, which is our primary project. It's a CNCF project. I think it's the only CNCF service mesh,

because Envoy is now considered a service proxy, which is slightly different, also in the CNCF. But when we first started working on Linkerd, the term service mesh didn't exist. In fact we didn't even know what to call this thing. We had seen this pattern from our work at Twitter and we knew that it was a good pattern, but it didn't have a name and so we were calling it like an application router or maybe we call it an RPC proxy. You'd have to kind of explain to people what that was and what that meant.

At some point we had the brilliant marketing idea to just give it a new name. So we didn't have to keep differentiating it from like HA proxies and things like that. So we were like, "Well, it's kind of meshing all these services. So we'll call it a service mesh." That term really caught on.

Linkerd, kind of by virtue of being widely adopted, spread this term service mesh and the model caught on. Now fast-forward a couple of years, the service mesh, not there's like 20 different service mesh projects and they all have very, very different implementations and different goals, but the core model is the same, which is that we want to extract that logic out of the application and we want to deal with it at the infrastructure level. There's a couple of key things that have happened as part of the kind of ecosystem around it, especially in Kubernetes and Docker that have made this model suddenly very, very relevant.

[00:13:30] JM: So there is this very niche world of backend infrastructure, analysts and journalists this world that I am in, where the topic of a service mesh is like a way to start a conversation. It's like a water cooler or a cocktail party conversation. The idea of service mesh, like do you need one or not? Is this a giant platform business or not? It's become a conversation starter, which is kind of weird for people like me who I'm not writing software. I don't have any privilege and understanding of what is or what is not something that will become an infrastructure trend.

There are companies who have made very good use of a service mesh, like you have companies like Monzo Bank who we had on the show. You have PayPal, Expedia, and then there are people who say, "You don't need a service mesh if your most companies. It's just going to add you extra complexity. This is the microservices industrial complex." The service mesh debate seem to have this strange political edge to them sometimes. Am I imagining this or do you see this same kind of fervent debate happening?

[00:14:42] WM: I don't think this is anything that is unique to the service mesh. I think anytime there is a new piece of technology that's being introduced, you always have these debates of like, "Is this really necessary? Gosh! We've got all these other machinery and now you want us to add like this other layer of complexity? I just wrapped my head around Kubernetes. I just wrapped my head around Docker. I just wrapped my head around microservices and now you're telling me there's this other thing?"

I think it's a very natural reaction, and I think if you rewind history and you go back 5 years, or 10 years, or 20 years, you'll find the same debate happening about pretty much every technology. In some cases, those technologies went on to transform the world and in some cases they really were nonsense and not that useful. In retrospect, everything is clear, but it's a common reaction.

[00:15:31] JM: It does seem like most of the time these technologies do end up being useful enough to have a large market. Like you were just saying, there are products that have come out and then they're not that useful. I guess you could talk about Betamax or something, but in the software world, can you think of an example of something where it just ended up not being useful enough for a big enough market to build a business?

[00:15:56] WM: Oh, yeah. I mean, I think the history of software is littered with those ideas possibly more than the other way around. I think you could look at something like the semantic web, for example. That was a huge topic of thought leadership for a very, very long time, and it was backed by the same people who had invented the worldwide web. It wasn't just made up by a bunch of randos. It was like serious thought went into this and it never really went anywhere. You could make similar arguments to – Although maybe they took off a little bit more than Semantic Web did. You could look at things like SOAP, RPC or you could look at some of the like more dense web specs around – Gosh! There were all these WS-* specifications that became very, very intense and had a lot of thought put into them but never really caught on. Just because the service mesh is popular, it doesn't mean that's going to last. But I do think it is and for other reasons, just not popularity-based ones.

[00:16:54] JM: What kinds of companies really need a service mesh, or is it something where you would just say, "I'm standing up a Kubernetes cluster anyway. I might as well have a service mesh if I could just one-click it." Who is it for?

[00:17:08] WM: So it really is less about the company. It's more about the technology stack. In particular, if you are adapting microservices. We've been talking about microservices for 10, 15 years as an industry. Maybe not 15 years. I don't know. What has changed in that time is basically the rise of this cloud native stack, so the rise of something like Docker and Kubernets has really reduced the cost of adopting microservices. I'm going to go on a little microservices tangent here, because this is really kind of fundamentally what the service mesh enables.

So when I was at Twitter, and this was starting almost years ago, so 2010. This was the tail end of the kind of monolithic Ruby on Rails infrastructure. It was falling over. There were all sorts of problems. I have all sorts of hilarious stories about the man problems we had.

[00:18:03] JM: We've retold those on previous episodes.

[00:18:04] WM: Yeah, exactly. Please refer back to previous episodes. I'm going not going to retell them. I probably forgotten them all by now. You've got all the good ones on tape. But that investment over the next five years at Twitter, into Twitter's massive microservices architecture, was insanely complicated and insanely expensive. There are so many engineering hours, engineering years that were poured into that transformation.

Now, as a startup, you roll up to the table with Docker and Kubernetes and you already have something that's better than what Twitter probably will ever be able to achieve. So what those things have done is they've dramatically lowered the cost to adapting microservices.

We have startups who we talk to who are five developers and have 35 microservices, or even more, 100 microservices, because once you are using – Once you have one service deployed in Kubernetes, deploying another one is kind of the same and deploying 10 more is kind of the same, except for once you actually have those things running, now life gets a little more complicated, and that's where the service mesh comes in.

[SPONSOR MESSAGE]

[00:19:18] JM: Simple Contacts lets you renew your contact lens prescription online easily. For many years I have ordered contact lenses on websites with low quality user experience. I have ordered contact manually from my optometrist, which requires me physically driving to the optometrist's office to pick them up. It doesn't make any sense.

Contact lenses are expensive. It's a high-margin product that is cheap to produce. Why is it so hard to order them over the internet? Well, now it's actually easy to order contact lenses on the internet. Go to simplecontacts.com/sedaily and use promo code SEDAILY to get $20 off of your contacts. I used Simple Contacts and I was amazed that it actually works. The website is pretty good. I uploaded my prescription. I ordered my lenses and then they arrived quickly in a box. If you have contact lenses, there's no reason not to try it out and save $20 on your next batch of lenses. Why not? Go to simplecontacts.com/sedaily and use the promo code SEDAILY.

Thank you to Simple Contacts for being a sponsor of Software Engineering Daily and for making a product that I'm just getting use going forward. It's pretty much the way that I would want to order contacts, and it took a really long time to finally get here, but I'm glad it is here. So go to simplecontacts.com/sedaily if you use contact lenses.

Also, as a quick disclaimer, Simple Contacts is not a replacement for your periodic full eye exam. Simple Contacts only tests that your current prescription still helps you see 20/20 and helps you renew that prescription. They don’t write completely new prescriptions or examine eye health.

[INTERVIEW CONTINUED]

[00:21:22] JM: So there are some large companies using Linkerd in addition to Monzo Bank, and Expedia, and PayPal. There are several others. When you talk to these companies about service mesh, how do they describe what is valuable about it?

[00:21:39] WM: Yeah. I think the value propositions are usually – They fall into these three categories. One is purely around the visibility or observability layer. What the service mesh can

give you, because all these proxies are running at kind of layer 5 through layer 7 in the OSI networking model, they're able to give you metrics around success rates and request volumes and latencies. These are kind of like what we call the golden metrics. These are the things that you really care about, because if the success rate goes down, well, someone's waking up to fix it at 3 AM.

Now there's a lot of other metrics that are application specific that the service mesh can't help you with. There you're kind of, you know, you've got to [inaudible 00:22:18] application. That's always been the case. But in terms of having a uniform layer of observability across the entire stack, service mesh is a huge win. So that's one bucket. There's a bucket around security features certainly in terms of both encryption everywhere. Service mesh can, in Linkerd, is used heavily for this purpose, initiate a TLS connection and then terminate it on the other end. You have encryption and authentication in between different services without having to do a whole lot of work.

In fact in the latest version of Linkerd, we ship we a CA, ship with a certificate authority that will do the key distribution for you. So you literally don't have to do anything to get validated TLS certs spread everywhere in your application. Finally, the third bucket is the reliability features, things like retries and circuit breaking like you mentioned before.

[00:23:08] JM: Yeah. I did a couple of shows recently, one on SPIFE, which SPIFE is a workload identification system, and open policy agent, which is for managing the policies throughout your organization. It was kind of unexpected, but in these shows it seemed like the service mesh was a great way to deploy and manage policies. Is that consistent with what you're seeing? How are you seeing people managing networking policies and workload authentication, workload identification across Kubernetes clusters?

[00:23:44] WM: Yeah. It's definitely – The service mesh is an extremely convenient way of deploying kind of the core component that you need for policy, which is enforcement. Because we've done all these work already to kind of wire up the proxies so that traffic is going through them automatically. When A talks to B, when service A talks to service B, by default the service mesh is just proxying and doing its best, but the service mesh could say no. It could say, "No. A is not allowed to talk to B based on some policy."

How you define that policy, how you describe it, how you iterate on it, all that stuff is kind of complicated, a complicated set of concern that's outside of the service mesh's sphere of influence, but how you enforce it and who's actually saying, "Yes, this request is allowed," or "No, it's not allowed," that's 100% what the service mesh is positioned to do. So you often find those things going hand-in-hand.

In fact, there's a huge – I'm sure you touched on this in those podcasts as well, but there's a huge shift that's happening in the world of kind of enterprise security right now where everything used to be done at the perimeter. Once you're inside the perimeter, you're like in the like soft underbelly of the beast and everything was allowed. People are moving away from that model to a world where even when you're inside, even if you manage to get inside there, everything is super locked down.

[00:25:04] JM: Yeah, zero trust networking. When you are dealing with these different enterprises that are adapting service mesh, are you seeing them with one giant Kubernetes cluster or a series of smaller Kubernetes clusters? How many Kubernetes clusters are we talking about at these large companies and are they managing all of these clusters with one service mesh? How does that look?

[00:25:30] WM: Yeah, the world is still figuring this out. It's rare to see only one Kubernetes cluster. Typically, at a minimum, you have one as kind of a staging environment, one is a prod environment. You may have several as kind of developer environment as well. That's not even including the Minikube or Docker for Mac Kubernetes mode that you'd run on your laptop. But the topic of federation and how that works and how that interacts with the service mesh, it's still very early and my suspicion is that's going to be a huge topic of development and discussion in the upcoming year.

[00:26:05] JM: Right. Do these multiple clusters get managed with one service mesh?

[00:26:10] WM: That's also kind of a topic of some discussion. I think our model so far is to keep the service mesh on a per cluster basis, or possibly at least the Linker is it keeps a service mesh on a per cluster basis and even possibly on a per name space basis. You can actually run

Linkerd in a single name space and there are some users for whom that is a big plus and then to deal with cross-cluster communication at a separate layer. That kind of fits in with how we've seen most folks adopting Kubernetes, but it's far from set in stone at this point.

[00:26:47] JM: So what about monitoring? How does a service mesh fit in with a monitoring stack, like you've got Prometheus and Datadog and all these other tools you might be using? How can a service mesh be useful in this stack?

[00:27:05] WM: Yeah. So it's really designed to complement those systems and not to replace them. Linkerd, for example, the 2.x branch ships with a little Prometheus installation as part of the control plane, and that powers a bunch of the UI and it powers a bunch of the CLI. So you can actually run these CLI commands that are kind of equivalent to something on the top for looking at processes. We have this equivalent, which is top for your services, like show me all the services that are running sorted by request volume, or sorted by success rate or something like that, and then there's Grafana Dashboards and things like that. But the way that we do it is we try and we make it very easy for you to get those metrics, but they're not supposed to stay in Linkerd for the long term. Linkerd will expire those metrics after some number of hours and you're supposed to extract them and put them into a real long-term time series database, whether that's your own Prometheus cluster that you're running, whether it's Datadog, whether it's something else.

What the service mesh is very good at doing, what Linkerd is very good at doing, is getting those golden metrics per service, certainly, and giving those to you on a uniformed basis across your cluster. That's not the end of the metric story. There's a lot more you have to do. Actually, this is a great time to have that question, because we're about to release Linkerd 2.1. In 2.1 we have something that I haven't seen anyone else do before, which is we can give you per path or per route metrics. It's not just service A has as success rate of 89%. It's, "Oh, this call to service A/user/ –" Whatever, get or something. It's not a great example. Is that 100%? But a post to / users, well, that's failing 12% of the time. So we can break things out per route. That's a huge step forward for platforms owners and for service owners who have to operate these services.

[00:28:59] JM: Yeah, that's kind of like a distributed trace sort of thing out of the box.

[00:29:04] WM: Yeah, that's right. That's right. There are a lot of similarities. Now, we don't do the full [inaudible 00:29:08], and in fact nothing precludes you from doing it as normal, but we can give you a lot of the value without you having to do any work. In fact we can even draw the application topology for you. We can show you a map of A talks to B, and B talks to C, and C talks to X and Y, and B and C talk together, and we can generate that map on a real-time basis, because the proxies see all that data and they report it to the control plane and the control plane can aggregate it. That's typically something that you have had to implement distributed tracing to get, but we can draw that picture for you without you having to do any work.

[00:29:41] JM: That's pretty useful. So you started with the first version of Linkerd, and then you built Conduit, which was a new service mesh that was written in Rust, and then Conduit got integrated into Linkerd 2.0. Now you're on the cusp of releasing 2.1. Take me through the product evolution.

[00:30:02] WM: Yeah, so now you're getting into kind of the sorted history.

[00:30:05] JM: That's all right. It's all sorted. Everyone's history is sorted in one way or another.

[00:30:09] WM: Yeah. This is a little bit of how the sausage got made. So the initial version of Linkerd, what we call the Linkerd 1.x branch, and by the way we continued to maintain this actively and we invest a lot of time and energy in 1.x even though we have 2.x as well. 1.x was built on top of the Twitter stack directly. We're very familiar with that stack. It was production grade. So we built Linkerd 1.x on top of Scala, and Netty, and Finagle and this ecosystem that Twitter had really advanced of having very high-performance, high-throughput network operations on the JVM.

That got us a lot of adoption. In fact, many of our biggest adopters are on the 1.x branch. 2.x is much more recent. But the JVM is very good at scaling up. What it is not good at is scaling down. In this world where we are distributing lots and lots of proxies everywhere and you might have 50 or 100 or even more proxies running on a single node, it kind of sucks to run 100 JVMs on a single node. As much work as we do into squeezing it down, and we can actually get it quite small. It's still painful to do that.

Starting some time last year, 2017, we were investing heavily in what was at the time a very experimental idea, which was, “Okay, can we get off of the JVM entirely and can we build a networking stack that is in native code?” The language we chose to do that was Rust, and Rust was a bit of a risky choice, because unlike the JVM, it did not have, certainly not at that time, a fully fleshed out networking stack that we could rely on. So we had to invest pretty heavily in building out some of those core networking components, things like tokio, and tower, and h2. These are all core networking libraries that Buoyant and that Linkerd has spent a lot of time and energy on.

We call that Conduit, because Linkerd had a very good reputation of being this production grade system and we’re all nervous of just releasing a new thing in Linkerd that had not gone through the extensive vetting in production that Linkerd had. But then fast-forward maybe not quite a year, but almost a year, and this thing, this idea of using Rust and then on the control plane, we’re using Go, was all shaping up extremely well. It was a risky choice, but the bet was really paying off. We find ourselves in a point where everything we were doing with Conduit we knew had to be the future of Linkerd as well. So we merged the Conduit codebase into Linkerd and a month or two later that became the first version of the 2.x branch. So that was Linkerd 2.0. So 2.0 now is off the JVM entirely. Still gives you the same and it still has the same value props around visibility and security and performance and reliability and all those things. But a very new codebase and a much better performance as well as much better resource utilization.

[00:33:08] JM: A service mesh is something that is deep in the stack. This is a low-level piece of infrastructure. How do you test these new service mesh versions as you build them and make sure that they’re production ready? What’s the testing and release process?

[00:33:24] WM: We just throw it out there and see what happens.

[00:33:27] JM: Okay. Just like a podcast.

[00:33:29] WM: It’s open source. Hey! No. we have a pretty rigorous set of tests that we run. One of the advantages of being a CNCF project is that there’s a community cluster that we rely on quite heavily to run these test suites. Every time a commit gets merged into master, it runs through a pretty rigorous test both kind of integration and functional testing, but also

performance testing. So Linkerd is open source, but there are many companies who build their core infrastructure on top of it. So just throwing it over the wall would not feel right at all.

[00:34:04] JM: So in the last year, what's the hardest engineering problem you've had to solve? This is a question I've started to ask more and more people. It's a very broad question, but tell me the hardest thing you've had to solve.

[00:34:16] WM: Yeah, that's an interesting – That's a really interesting question. That's probably a question for someone who writes codes, because these days all I write are emails. Sometimes emails describing code. But I'll tell you what I do know, which is one of the things I think has been tricky to tackle, but that has been really awesome, is how is the interplay between memory management and network performance at the proxy level.

So in the JVM, in the 1.x branch of Linkerd, we can rely on the JVM which has a very sophisticated garbage collector. We can kind of allocate memory. You have to be a little careful about how you do it, but you can allocate memory as part of the request and then not worry too much about it, because you know the garbage collector is very finely tuned and it will scoop up that memory afterwards. That makes writing the code a little bit easier. But what it means is that every once in a while the garbage collector rears its ugly head and it says, "Okay, I need to spend a couple tens of milliseconds or a hundred milliseconds or something collecting all these garbage."

When that happens, your network performance goes down. We characterize this performance in terms of things like tail latencies, where you say if you look at the P99 – We talk about the P99 of the proxy, and that's the latency that happens, the worst 1% of latency that happens. Typically that P99, when that number is high, it's because the garbage collector reared its ugly head.

So now in the Rust world, this is where things get interesting, because Rust is purely manually managed memory. Rather than having a garbage collector – And even Go, which has very good performance, still has a garbage collector and still works largely by growing the heap. So there's a bunch of reasons why Go was a great choice for us in the control plane, but we didn't want to write the proxies in it.

In Rust, there's no garbage collector at all. You never have to worry about that. But instead you're paying the price because you're doing manual memory allocation. A lot of the work that we did in this network, in kind of the underlying network libraries, was how do you – At kind of the programming level, how do you get – You want kind of the best of both worlds. On the one hand you never want to incur a cost for allocating and de-allocating memory except for kind of the minimum, absolutely minimum cost that you can.

On the other hand, you don't want to end up with this really complicated spaghetti code of callbacks upon callbacks and kind of this tortious logic that will be difficult to maintain. So Rust gives you the ability to build what they zero-cost abstractions. You can say, "Hey, I have – This is a future." A future, if you're familiar with JavaScript or Finagle, it's way of expressing kind of a pending computation, a computation that may or may not have happened or a call that may or may not have returned. So you can build in the proxies. We've gotten to the point where we can build using futures and other kind of zero-cost abstractions, ways of allocating when a request comes in, we allocate all the memory that we need for that request. When that request terminates, we de-allocate all that memory. We do it all incredibly quickly. By doing it on a per request basis, that means that our latency distribution is extremely sharp. It means we don't have these long tail latencies, because we don't have a garbage collector. At the same time, the code is kind of – It doesn't end up being this crazy mess of spaghetti callbacks. It is this zero cost, very immediate, "I just delete the future and everything else happens." That's probably the hardest challenge that I'm aware of that we've had to tackle over the past year.

[00:37:49] JM: I feel like memory management is at risk of becoming like a lost art. Is it hard to find people who can program Rust code?

[00:37:59] WM: I'd say yes and no. On the one hand, it is a complicated language. It requires some study and some thought to really master it. I'll contrast that with Go. Like I said, we use Go for the control plane, and the barrier to Go, what I like about Go, is that the barrier to entry is quite low certainly relative to something like Rust. So we have a lot of external contributions that happen on the control plane, because it's easy for people to pick up Go.

On the Rust side, that barrier to entry is difficult, but it's also the kind of language that people get really excited about, because it's so cool. It also has a really welcoming and really accepting

community, which is rare I think in a very hardcore systems language. You expect the community around that to be like kind of the worst, angriest people. But actually that Rust community is super friendly, super welcoming and really motivated by this idea of lowering the barrier to entry for a systems programming. It has been, I guess, medium difficult.

[00:38:59] JM: I know you're probably spending most of your time in your inbox or maybe on an Asana Board these days, or on LinkedIn perhaps. What are you doing in terms of broad company scaling? How had Buoyant evolved in the last year and what are the some of the challenges there on that company scaling front that you've had to solve?

[00:39:18] WM: For us, I think we're a little atypical in that kind of the – You have this vision of the startup being like always in this hyper-scaling mode and like we've got to add 50 people, now 500, now we're at 5 million. For us, open source can be dealt with. I think the realization we've had is that open source can be done very efficiently. We don't have to have everyone in San Francisco. We don't have to have these lengthy kind of product processes. Because it's an open source project, we really have a lot of advantages. Also, because we do a lot of – We do invest a lot of our time and energy in the community around Linkerd, it really is a community project. Buoyant is lean and mean, which is the way that we like it, and we just do our best to keep the community around Linkerd in a really healthy and engaged shape.

[00:40:11] JM: Amen to that. I can support your position on very small efficient companies, because we only have like two or three people here.

[00:40:20] WM: Yeah. Life is a lot easier.

[00:40:23] JM: It is.

[00:40:23] WM: When that communication burden is not that high.

[00:40:29] JM: That's right. In terms of the competitive landscape, Google came out with the Istio service mesh. That was about a year ago. I guess it was a Google open source project. I think IBM and some other companies were also involved. What was interesting to me is that it seemed like even though Istio came out of Google, Linkerd was still, I guess, the vendor of

choice, because you had just been working on it longer. I think when people go to find a service mesh to purchase, they want one that's battle tested, because this is something they're going to insert in the core of their infrastructure. Did Istio affect your product strategy or your sales process at all?

[00:41:11] WM: It affected it in the sense that it was extremely helpful to have a company like Google enter the service mesh space and say, "Hey, we have a service mesh project too." Because as a startup, you can only make so much noise. This is basically called market validation. Istio coming in to the market was a huge validation of the fact that the service mesh was a real pattern and a real model that had value. There's only so much we could say as Buoyant.

So it was very helpful, and I think in response to that, it's also why you've seen the 17 other service mesh projects that have cropped up. People are realizing that this is a real thing. There are some real value here, but that said, it's still the tip of the iceberg. The companies that can adapt the service mesh today are still very much the ones who are kind of on the bleeding edge of this cloud native ecosystem. So that's a long, long roadmap ahead of us in getting the rest of the world to get the value of the service mesh.

[00:42:13] JM: Yeah. More recently, AWS came out with their App Mesh, which is a service mesh product. Is that also a validation moment or is that anymore of a concern?

[00:42:25] WM: I guess you could consider it further validation. I mean, we kind of – It's an additional 5% validation I guess. Are there concerns? I don't know if they – Would I call them concerns? Well, honestly, I haven't really looked at App Mesh very much. Istio, we're quite familiar with mostly because it's been around longer.

No, I wouldn't call them concerns. I think the goal for all these projects is to provide the service mesh value props to the user and it's really a question of, "Well, which tradeoffs are you making and how tightly integrated are you with Google Cloud, or with AWS, or whatever else?" But at the end of the day, at least in my mind, if you can get people using something like Linkerd and getting the value out of it, then you've done a good thing for the world.

[SPONSOR MESSAGE]

[00:43:22] JM: How do you know what it's like to use your product? You're the creator of your product. So it's very hard to put yourself in the shoes of the average user. You can talk to your users. You can also mine and analyze data, but really understanding that experience is hard. Trying to put yourself in the shoes of your user is hard.

FullStory allows you to record and reproduce real user experiences on your site. You can finally know your user's experience by seeing what they see on their side of the screen. FullStory is instant replay for your website. It's the power to support customers without the back and forth, to troubleshoot bugs in your software without guessing. It allows you drive product engagement by seeing literally what works and what doesn't for actual users on your site.

FullStory is offering a free one month trial at fullstory.com/sedaily for Software Engineering Daily listeners. This free trial doubles the regular 14-day trial available from fullstory.com. Go to fullstory.com/sedaily to get this one month trial and it allows you to test the search and session replay rom FullStory. You can also try out FullStory's mini integrations with Gira, Bugsnag, Trello, Intercom. It's a fully integrated system. FullStory's value will become clear the second that you find a user who failed to convert because of some obscure bug. You'll be able to see precisely what errors occurred as well as the stack traces, the browser configurations, the geo, the IP, other useful details that are necessary not only to fix the bug, but to scope how many other people were impacted by that bug.

Get to know your users with FullStory. Go to fullstory.com/sedaily to activate your free one month trial. Thank you to FullStory.

[INTERVIEW CONTINUED]

[00:45:45] JM: Do you have a perspective on how much – So like if somebody's already on AWS, to what degree are they just defaulting to the AWS Services? Because the sense that I get when I talk to companies that are making these purchasing decisions, for example a purchasing decision between Elasticsearch hosted on AWS versus going with Elastic's enterprise version. When a customer is making this buying decision for some enterprise product

that's going to be at the core of their infrastructure, they are going to look at the AWS option, but it seems like they look quite seriously outside of AWS as well. I feel there might have been overblown concern about, "Oh, AWS has every single option available. How can anyone else in the market compete?" It does seem like people are able to develop their own independent businesses that are superior or at least highly competitive with the AWS product.

What has it been like kind of positioning yourself as an independent software company to the side of the cloud providers and to what degree do you feel like the cloud providers have this really strong advantage of already having a market channel?

[00:47:03] WM: I think if Linkerd were a commercial product, I would have a very different opinion, because then everyone using – Every person who was using some AWS solution and not using Linkerd was like less money into my pocket. Because it's an open source project, I think my attitude is a little different.

I do think people are increasingly aware, certainly the companies that we talk to are increasingly aware of vendor lock-in and increasingly kind of less likely to just make a decision based on, "Oh, well AWS has it and we're already paying a bunch of money. So let's just do it." That might have been the case 5, 10 years ago, but I don't think it's the case now.

Then I guess the third thing is technology choices, and I think this has been very heartening to see, but the technology choices are increasingly being made by engineers, kind of the boots on the ground engineers. It's not the CTO mandating, "Hey, we have to use App Mesh because we have an AWS strategy," or something. Engineers on the ground are saying, "Hey, we're looking at these three options, and on these, Linkerd is the best and therefore we're going to use Linkerd."

That's been great to see, because it means that as long as Linkerd is the best project, we're like – Well, best is a loaded word. As long as Linkerd kind of meets the needs in kind of the most immediate way, let's put it that way, then they'll adopt Linkerd. That's kind of how it should be.

I guess, finally, I think the cloud providers certainly color the nature of their projects. If you look at something like Istio, Istio in my mind at least is very much designed to be something that

Google will run for you. It's got every feature ever known to men, every feature checkbox, but very complex to operate. It's very large, takes a lot of memory. There's a big kind of cognitive burden to understanding all the new APIs. I think that makes sense as something that GCP would run for you, because then that burden isn't really there. You just hear about that feature checkbox.

The tradeoff for Linkerd are very different. We do want people to adopt it. We want people to operate it. So the cognitive overhead and kind of the resource consumption of both the proxies and the control plane are very much top of mind concerns for us. So it's a very different focus for the project.

[00:49:27] JM: Yeah. We had a few shows a while ago about the open source business model discussion, and this was around the time of the whole commons clause, Redis Labs question. What's been your experience building a business around open source and what was your reaction to that commons clause discussion?

[00:49:51] WM: I can understand why you would want to do something like that. As a startup operator, I guess I very naturally have that perspective. I think if our business model or selling Linkerd+ or Linkerd enterprise or something, and that's kind of – That's probably a direction I would be thinking about too.

For us right now, Linkerd is not a product that we are going to sell. I think it's really important to us for Linkerd to be a first-class community project for there to be a healthy, engaged community around it. Buoyant does sell support for it, but it's not a different version of Linkerd. We support the open source. So we've kind of skirted a lot of these issues and that our goal, the software that we do want to sell is really not the open source software.

[00:50:42] JM: Do you think the support business is big enough to be a long-term community or do you think you'll end up developing additional products in addition to the support business?

[00:50:55] WM: I think the support business is extremely healthy for us and I think it's necessary for Linkerd adaption, but I don't think it's the long-term, certainly not the long-term business model for us. I think we have to do it. It's good for the project. Yeah, I think it would be

difficult to build a really scalable business purely on top of support. The example that everyone gives is like the counter-example or the exception that proves a rule is Red Hat, and I think there's a bunch of interesting questions around why they were able to do it. It's not what we want to do. I do want to continue providing support, because it's healthy and it's good, but it's not the long-term goal.

[00:51:30] JM: I think it's a great strategy, because the thing is if service mesh is a thing you want, if it's not the semantic web of Kubernetes, then there's going to be a lot of stuff built on top of it. If you do support in the early days, you get a ton of customer interaction and you don't set yourself down a path of building products over eagerly. You just kind of build this steady support contract revenue and then you can kind of build products opportunistically.

[00:52:04] WM: That's right. Yeah, that's right. That's right. If you look at my view of the service mesh, it's very similar to my view of Kubernetes and of Docker and kind of echoes what I said in the very beginning of this conversation, which is that all these things are enablers. They're enablers of microservices. These are all kind of technology choices that allow us to run and to build our software as lots and lots of services that interact. That transformation is a very profound transformation not just at the technology level, but at the organization level. That's where I think things get interesting for a company like Buoyant. What are the things that beyond just the core technology? What are the things that a company that is adapting this cloud native stack needs in order to kind of function at the organization level in a world where now the structure of engineering team is different, because we've got different teams supporting different services. As a result of that, the structure of how HR works is different, and the structure of how the finance department works is different. Once you adopt microservices, things change in the company, and that's where things get really interesting for Buoyant, the company, as supposed to Linkerd, the open source project.

[00:53:13] JM: So another changing trend is like people are going to lean more and more into functions as a service, other managed services on these cloud providers. Do you see an opportunity to instrument those with some kind of service mesh thing, or do you just expect to be a service mesh over the long-lived services that people are deploying themselves and that will be holly separate from your managed database or your functions as a service that are running on AWS?

[00:53:49] WM: I don't think it was an accident that Knative was built on top of something like Istio. I think what the service mesh provides are some fundamental components that any kind of function as a service style system is going to want to leverage. I think the two will go hand-in-hand.

[00:54:06] JM: Very interesting. Well, we're on the cusp of going to KubeCon. I think it's going to be gigantic. There's like 7,500 people that are going to KubeCon North America. What are you looking forward to at the conference?

[00:54:19] WM: Well, I find these conferences to be so energizing and so exciting, because as an open source project, you're interacting with people over the Slack channel or over email or something like that. To actually meet them in person really is a different kind of relationship. So every time we go to KubeCon, we'll have people come up to us and say, "Hey, we're using Linkerd in production. It's been really awesome," or they'll say, "We found this bug and we need to fix it," or whatever, but like just having a human being in front of you saying, "Hey, we're using your thing. We're building our job, our livelihoods. Our careers are being built on top of Linkerd. That's such an energizing conversation to have.

So really what I am excited about is the hallway track and talking to people who are using Linkerd in production to do their jobs, to build their careers and their livelihoods. The talks are nice, and I do attend them, but really it's the conversation with Linkerd adapters is what I'm looking forward to.

[00:55:20] JM: All right, last question, this is somewhat a self-interest. So my world is kind of understanding how developers adapt things and buy things and businesses around developer marketing. So I try to understand how are people making sales decisions. How are they making marketing decisions? What have you learned about developer marketing? I mean, you alluded to this earlier, that the engineers are making more and more of the buying process and probably it's shifted away from this CIO that is making a decision from on high. It's gotten pushed out the edges of the organization. How has that changed the actual sales strategy for a successful enterprise software company? How are you thinking about things today?

[00:56:08] WM: Well, I could only really talk about it in the context of open source, but certainly for open source adaption, the moment that anything has a whiff of marketing or sales in it, you immediately see people run away.

What's been most effective for us – And again, talking purely about open source adaption, is having kind of authentic engineer-to-engineer conversations and just being very clear and transparent about what's happening in the project and whether we can or cannot get to your feature request in the next 6 days, 6 weeks, 6 months, whatever it is.

The moment that there's like a marketing person trying to talk to you, every engineer is like, "Ah! I don't want that." They're going to try and sell me something. To the extent that what open source is really good at is – At least from my kind of high-level view, is it's really kind of a marketing tool. It's a way of getting your product into the brains of engineers, but you can't treat it the same way that you – You can't build up a team of like the GrowthHackers or kind of content marketer. Well, content marketers you can kind of do. You can't treat it the same way as you treat kind of standard lead-gen enterprise. That's been – I guess I wouldn't say that's been a big learning for us, because that felt very natural to us as a bunch of open source nerds, but it's really been driven home every time we do anything kind of that is marketing-y at Buoyant. We just have to be very careful and very clear about what the boundaries are between kind of the Buoyant marketing kind of treatment and the open source treatment.

[00:57:45] JM: William Morgan, thank you for coming back on Software Engineering Daily. Always a pleasure to talk about the latest in Linkerd and the service mesh.

[00:57:53] WM: Jeff, it's been a real pleasure. Let's keep doing it.

[00:57:55] JM: Absolutely.

[END OF INTERVIEW]

[00:58:01] JM: OpenShift is a Kubernetes platform from Red Hat. OpenShift takes the Kubernetes container orchestration system and adds features that let you build software more quickly. OpenShift includes service discovery, CI/CD built-in monitoring and health

management, and scalability. With OpenShift, you can avoid being locked into any of the particular large cloud providers. You can move your workloads easily between public and private cloud infrastructure as well as your own on-prim hardware.

OpenShift from Red Hat gives you Kubernetes without the complication. Security, log management, container networking, configuration management, you can focus on your application instead of complex Kubernetes issues.

OpenShift is open source technology built to enable everyone to launch their big ideas. Whether you're an engineer at a large enterprise, or a developer getting your startup off the ground, you can check out OpenShift from Red Hat by going to softwareengineeringdaily.com/redhat. That's softwareengineeringdaily.com/redhat.

I remember the earliest shows I did about Kubernetes and trying to understand its potential and what it was for, and I remember people saying that this is a platform for building platforms. So Kubernetes was not meant to be used from raw Kubernetes to have a platform as a service. It was meant as a lower level infrastructure piece to build platforms as a service on top of, which is why OpenShift came into manifestation.

So you can check it out by going to softwareengineeringdaily.com/redhat and find out about OpenShift.

[END]

