**EPISODE 1049**

[INTRODUCTION]

**[0:00:00.3] JM:** NGINX is a web server that's used as a load balancer and API gateway, reverse proxy and other purposes. Core applications servers such as Ruby on Rails servers are often supported by NGINX, which handles routing the user requests between the different application server instances. This model of routing and load balancing between different application instances has matured over the last 10 years due to an increase in the number of servers and an increase in the variety of services.

A pattern called service mesh has grown in popularity and it's used to embed routing infrastructure closer to the individual services by giving them a sidecar proxy. The application side cars are connected to each other in this proxy system and requests between any two services are routed through a proxy. These different proxies are managed by a central control plane, which manages the policies of the different proxies.

Alan Murphy works at NGINX and he joins the show to give a brief history of NGINX and how the product has evolved from a reverse proxy and edge routing tool to a service mesh. Alan has worked in the world of load balancing and routing for more than a decade, having been at F5 Networks for many years before F5 acquired NGINX. We also discussed the business motivations behind the merger of those two companies.

Full disclosure, NGINX is a sponsor of Software Engineering Daily.

[SPONSOR MESSAGE]

**[0:01:37.7] JM:** When I'm building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust.

 

Whether you are a new company building your first product like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals. Go to softwareengineeringdaily.com/g2i to learn more about what G2i has to offer. We've also done several shows with the people who run G2i, Gabe Greenberg and the rest of his team. These are engineers who know about the React ecosystem, about the mobile ecosystem, about GraphQL, React Native. They know their stuff and they run a great organization.

In my personal experience, G2i has linked me up with experienced engineers that can fit my budget and the G2i staff are friendly and easy to work with. They know how product development works. They can help you find the perfect engineer for your stack and you can go to softwareengineeringdaily.com/g2i to learn more about G2i. Thank you to G2i for being a great supporter of Software Engineering Daily, both as listeners and also as people who have contributed code that have helped me out in my projects.

If you want to get some additional help for your engineering projects, go to softwareengineeringdaily.com/g2i.

[INTERVIEW]

**[0:03:27.4] JM:** Alan Murphy, welcome to the show.

**[0:03:29.2] AM:** Thanks, Jeff. Thanks for having me.

**[0:03:30.8] JM:** You were with F5 beginning back in 2005. This was about 15 years ago. Describe the load balancer market when you started at the company.

**[0:03:42.0] AM:** It was all hardware. I mean, it was big, big iron, big volume. The bigger the better. Our customers back then were fell into two camps. They were either co-locating at a data center, or they were running their own. Either way, the customers who were buying F5 hardware were aggregating tons of data through those boxes. We dealt with a lot of really high-volume clients.

Then a lot of more geographically distributed technology, like Global DNS to be able to distribute traffic between multiple data centers in colo, but it was all just massive ingress single point traffic.

**[0:04:16.1] JM:** When was there this industry shift from physical load balancing infrastructure toward in augmentation of the software based load balancing?

**[0:04:29.3] AM:** I think the big shift came – I guess the first indication that a shift was coming was when colos and data center companies started offering more segmented and granular services, companies like Rackspace is a great example. They did a great job at changing the market away from single point service, single point entry to more virtualized and distributed services. There was definitely a movement at the data center level back then, or maybe around 2010, 2009, something like that.

The big shift no question was cloud and it was the reliability, the stability and the cost and accessibility associated with cloud. That immediately flipped the switch of conversation from hardware to software and from single point aggregate traffic management to distributed traffic management.

**[0:05:14.5] JM:** Did that cause any other downstream effects in the market, like as we went from physical load balancing to software-based load balancing?

**[0:05:24.1] AM:** Sure. I think it solved too big, right? I should say introduced two big changes to the way companies handle their application infrastructure. One was that the more you software, if you used that as a verb, the more you softwared, the more granular they could start managing their traffic. Instead of just having this front door service that we used to call the bigger hardware aggregation point, instead of having a front door service, they could start managing traffic, managing that door, so that first step gate closer and closer to the app, so further down the app stack. I think that was a huge downstream change that moving to software enabled, that virtualizing traffic management itself enabled.

Then I think the second one was the change of ownership. Before, there were typically one or two teams that were responsible for all of the traffic management functionality, because they

handled it all. That shift to software and that disaggregation of the application stack, of the services of the traffic management brought new teams into managing that traffic. Today, we're talking with DevOps down in the weeds, day-to-day development engineers who have to deal with traffic management. Huge changes downstream.

**[0:06:31.8] JM:** The shift to software-defined networking, was this all managed by commodity infrastructure, or were there still specialized types of servers that needed to be used to host the load balancing and networking infrastructure as things move to the cloud?

**[0:06:51.2] AM:** I think it came down to what type of traffic was being managed. I think commodity, whether it's hardware software, commodity for your run-of-the-mill HTTP traffic can absolutely be handled by a number of different services, in different form factors, in different locations and environments. When we talk about very specialized traffic management, or the function of that traffic management, you start to see less and less commoditization. If we're talking about very application-aware traffic management for what we now call east-west traffic or service-to-service traffic for microservices, there's still a level of awareness and understanding in the environment that moves it out of the commodity requirement, or commodity bucket.

Then if we're talking about really specialized traffic for service providers for financial institutions that have their own way of formatting traffic, of passing traffic, but also traffic reliability, I think there's still absolutely a need for specialized components, whether again those are hardware, software, on-prem, cloud, doesn't matter. Yeah, the more specific we'll get towards the application need, the more we'll still need specialized, or very, very fine-tuned application delivery software or hardware.

**[0:08:01.4] JM:** I do want to eventually get to a discussion of modern software to find traffic networking and specifically service mesh conversation. Since you've worked in this area for so long and you've interacted with a lot of different client bases, you just mentioned that different verticals might have different needs for the hardware, perhaps the software that they use to manage their networking infrastructure. Financial services, for example. Tell me more about how different verticals might have different needs for their networking infrastructure.

**[0:08:44.3] AM:** Sure. We definitely see it, again, if we're talking hardware, software. It's a really interesting space to zero in on what those differences are. I'll talk about a couple examples. Financial, FinTech, FinServe, those companies are very, very dependent on insanely high reliability. It makes sense, right? We need low latency, high-volume transactions for things like economic trading. We need the ability to embed security controls into the application language itself, from the application traffic language.

FinTech, very, very focused on something like low-latency, high-availability. If you look at service provider, 4G historic service provider, also very concerned with low latency, but massive, massive volumes of very large types of traffic. We're talking about over the top providers, companies like Netflix who need to distribute their content over a service provider network. Those companies, those service providers are very, very focused on high-resiliency, high-availability, but for much, much bigger or larger traffic patterns than you'd see in FinTech, for example.

5G brings a whole different scope to that, because now we're talking about pushing services closer to the edge, containerization; still that high volume, but now the ability to carve out that traffic in a different way than we would in a traditional infrastructure. I think healthcare is another very interesting one, because security, regulations, government, control over that data or rather government control over how that data is used in access is paramount. We also talk a lot about remote working when we're in healthcare, prior to everything that's going on today. We talk a lot about big, big software components that need to reside on-site, but they need to communicate back to a centralized secure cloud environment.

Then we're talking about regulations and how to keep data sandboxed and highly secure. It's really, really interesting that these larger verticals do have these different traffic necessities that require either specialized components, or at least very, very application-aware components.

**[0:10:42.7] JM:** F5 now is the owner of NGINX. We're going to be talking some about NGINX today. You've spent considerable time working at F5, as well as NGINX. Both of these companies are heavily used products to define the networking infrastructure for different companies. F5's historical application has been the hardware load balancer, but it's moved into a lot of different software services that are required for edge networking, DNS infrastructure.

NGINX has been used as an edge proxy for a long time caching infrastructure, reverse proxying, load balancing. I'd like to understand the acquisition from your point of view. When F5 acquired NGINX, why did that acquisition make sense?

**[0:11:46.6] AM:** Yeah. I mean, as you mentioned, I was at the NGINX side when that acquisition happened. From the very first moment, I was super excited. It made complete sense to me. When I left F5, I'd been there for about 10 years as you mentioned. I left in 2015. The F5 culture at the time was very much focused on helping customers support their software, their drive to software. It was an exciting time at F5, but at the same time there was also this internal understanding that hardware is still an important factor. It absolutely drives a lot of the customer requests and we still have those verticals that we talked about, like service provider in 2015.

When I left to move to NGINX, it was a great opportunity to focus on as you mentioned, the exact opposite side of that coin, which is 100% software, but also 100% DevOps and application focused deployments. All of our customers at NGINX were focused on application level, reverse proxying, embedding that reverse proxy functionality as literally close to the app as you can get today. We'll talk about side cars in a few minutes in that regard. It was very much focused on software-only, get the packet to the app and then let's figure out how to handle it as soon as it gets into our environment in the software space.

When the acquisition happened, even with that four-year gap between leaving F5 and then rejoining, it made perfect sense to me because it melded the best of those two worlds, but also filled two gaps where F5 is predominantly hardware front door service with a drive to software. NGINX was virtually 100% software, but starting to deal with higher level networking order that things like containerized networks bring in, so that being able to put those two together of hardware network, deep history understanding, software application delivery, deep industry understanding bring those together. It just filled both gaps so eloquently, so perfectly, made perfect sense now. I was super excited to come back into F5 with that software mentality.

**[0:13:48.5] JM:** When that acquisition happened, you helped build out some of the strategies for integrating with cloud providers. This is an interesting strategic task to build, because you have these gigantic cloud infrastructure providers that are to some extent providing commodity services. They have their specialized services that they've built as well, but there was this huge

reorientation, well this orientation, reorientation in the industry that's still going on for how companies that are not cloud providers, companies that are not these gigantic infrastructure providers integrate with the cloud providers. Tell me about defining the strategy for F5 to integrate with the cloud providers.

**[0:14:39.7] AM:** I can speak first to the NGINX strategy and the work that we did there. In my role when I first joined NGINX was to help design some of that from the technology perspective. What are the technology roadblocks? What are the hurdles we need to get around? What do we need to provide to those cloud providers from the NGINX space? We were very fortunate, because we had such a huge market awareness and breadth of deployment at NGINX. Pretty much everybody in the world is using NGINX. That in itself was driving questions back to us of, “Hey, we're moving our infrastructure into AWS, for example, or GCP. We want to take that same functionality of NGINX into that environment.”

On paper, it was a literal transition. You're running NGINX software on-prem, pick it up and move it wherever you want, still works today. We had so many customers who followed that model. At the same time, we also had customers as you mentioned, who were coming to us and saying, we're either trying to build more intelligence into the cloud platform. We don't want to do just a lift and shift. Or we want to consume some of the tried-and-true NGINX resources through a consumption-based model in the cloud.

We had as you would expect, we had a really, really strong marketplace offerings and all the cloud providers, so someone could purchase NGINX in a consumption-based model. That was fine. We also spent a tremendous amount of time working with those providers on what value can NGINX help bring them. Today, some of the largest cloud platform tools incorporate NGINX. A lot of people use NGINX without realizing it’s NGINX because we have that flexibility. A lot of that work is what influenced some of those conversations and some of those architecture designs, so that we could provide what we do really well, which is extremely high speed reverse proxying at the application level, bring that technology into some of those more commoditized cloud components that you mentioned. We still see that drive today. We still work hand in hand with customers who want to use that core technology in cloud spaces, such as service mesh, for example, as we'll talk about in a few minutes.

**[0:16:39.5] JM:** Yeah, let's go ahead and get into that. For people who are unaware, describe what service mesh is. Describe what the service measure architecture is.

**[0:16:49.1] AM:** Yeah. The goal of a service mesh, if you think about a Kubernetes cluster, for example, so it's a good analogy. If we think about a Kubernetes cluster as its own data center, you still have the same traffic requirements within that cluster that you would have 15 years ago when you were building out application services within a data center. You still need to handle traffic as it comes in to the front door somewhere. Once it's into the environment, you still need to decide where it goes based on tried-and-true load balancing, high availability methods and algorithms.
Then when the traffic arrives at a particular application, oftentimes that application will need to talk to another application. You have to deal with what we call east-west traffic, so traffic between services or applications. In a Kubernetes clustered environment, all of those requirements are still there. Kubernetes does an excellent job of providing the core transport networking that we need to be able to allow those services to talk to each other, to allow ingress traffic to come in.

What a service mesh does is it brings a few extra higher-order level of authority to that application traffic, so that you can define security policies between services on a east-west traffic, you can orchestrate new policies to very easily get pushed out to application services, you have visibility components so you can start monitoring all that east-west traffic.

For me, a service mesh is really bringing that higher order logic to Kubernetes, or a clustered, container clustered environment to give you all of the traffic management tools that you need anywhere you deploy an app to give you that same level of granularity down at the service to service level, the east-west traffic level within clusters.

[SPONSOR MESSAGE]

**[0:18:37.8] JM:** This episode of Software Engineering Daily is sponsored by Datadog. Datadog integrates seamlessly with more than 200 technologies, including Kubernetes and Docker, so you can monitor your entire container cluster in one place. Datadog’s new live container view

provides insights into your containers health, resource consumption and deployment in real-time.

Filter to a specific Docker image or drill down by Kubernetes service to get fine-grained visibility into your container infrastructure. Start monitoring your container workload today with a 14-day free trial and Datadog will send you a free t-shirt.

Go to softwareengineeringdaily.com/datadog to try it out. That's softwareengineeringdaily.com/datadog to try it out and get a free t-shirt. Thank you, Datadog.

[INTERVIEW CONTINUED]

**[0:19:32.4] JM:** The service mesh model has a sidecar proxy, a sidecar container that gets paired with any of the applications, any of the services that are across that service mesh. Why is it advantageous to have all of your communications from these different services going through a sidecar proxy?

**[0:19:56.0] AM:** Yeah. If you think about that same metaphor of a cluster as a data center and you double click or zoom way, way down, you still have the need to get traffic from one application service to the other. In our case, let's say it's from one container to another container, in a typical Kubernetes environment or any containerized environment, because the core network transport is provided for us, Kubernetes for example, does a really good job of giving us that network connectivity through network overlays and technologies that they provide and also does a really good job of letting us know when one service is down or when a service needs to be scaled, because it's overloaded for example.

Beyond that, the services are basically allowed to communicate with each other as they see fit. If you think to very, very specialized deployments, let's talk about healthcare, because we mentioned them earlier, there may be very strict security policies in place that say, even though services are running in the same cluster, they should never be able to talk to each other without very, very strict security requirements. Those security requirements may be MTLS, so mutual TLS between services. Those security requirements may be very, very rigorous access lists or access policies.

You need a way to apply those policies to that service-to-service traffic and that's what a sidecar does. It resides in the data plane, so it's there between the actual traffic going from one service to the other. By having a sidecar there as a proxy, you have a way now to gate any traffic that's going into that containerized application. No traffic can ever get to the actual app, unless it goes through the sidecar.

The benefit there is twofold; one is that you have a single point for policy enforcement, security enforcement, that's the sidecar. The second benefit is that you no longer have to code any of that into the app stack or the app layer. Your application developer doesn't need to know what an ACL is, they don't need to worry about SSL exchange, or having the right client certificate. It takes that burden off the app and puts it back in the sidecar. That's the real value of a sidecar and why it's so important in east-west traffic management, specifically in a service mesh.
**[0:22:06.2] JM:** How does the configuration for the sidecar proxies get configured and deployed out to the sidecar proxies?

**[0:22:17.4] AM:** Every service mesh is going to handle it differently. A lot of it comes down to what tool is being used for the actual sidecar proxy in the data plane. Our world, because we have NGINX, I mean, NGINX is such a tried-and-true, fast, efficient reverse proxy, we're able to translate whatever that policy format is and I'll talk about that in a second, we're able to translate that policy into configuration language that NGINX understands.

For example, if we need to rotate the certificate, so we want to rotate certificates between services every hour or every 24 hours, or whatever the business policy is there, that certificate rotation can be transparently passed to an NGINX sidecar and that NGINX sidecar consumes it with zero interruption. It just consumes the new certificate and it's just great. From this point forward, this is the new client certificate I'll use for egress traffic, this is the new certificate I'll used to validate ingress traffic.

It depends on what the service mesh is. For us, it's NGINX being able to deploy that policy in NGINX language. The interaction with the administrator happens through the control plane and that happens through standard Kubernetes deployment methodologies through config maps, through any tool that the administrator is already using to manage application deployments

through Kubernetes. It's the same way that you pass a new policy to the sidecar. All we need to know is where do we inject that sidecar, so which application service do we need to attach to and what's the policy that needs to be applied to that traffic. Those two things are done through standard Kubernetes administration nomenclature.

**[0:23:56.4] JM:** There could be different proxies used for these different sidecars. Why would somebody use NGINX as the choice of proxy when they could use – there are other things like Envoy? What are the advantages of using NGINX as the service proxy?

**[0:24:16.2] AM:** To some extent, it's the same value prop you would have if somebody said, "Why should I use NGINX Plus, or NGINX in my data center instead of HA proxy, or another open source tool that's there? To a certain extent, it's that same argument. It's that NGINX and NGINX Plus are market tried-and-true. We have the most deployed reverse proxy deployments of any other application in the world fronting very, very critical high-volume sites and we have for 15 years.

A lot of it is just – it's trust. It's that we know how to build an extremely efficient reverse proxy. We know how to embed some extremely sophisticated routing mechanism, security policies. We can do all that today with NGINX. That's the first answer.

The second answer is that we were designed from the ground up to do all that in software only as a containerized appliance, or containerized function. Running as a sidecar to us is just as native as running on bare metal on an 88 core del box in front of the data center. It's just as native as running as an AWS AMI, just as native is running your own VM. We have that market support, we have that market trust, but we also have the design pedigree of knowing how to write really efficient software in a tiny, tiny footprint.

We're very fortunate that from day one, we can look back in hindsight and say some of the decisions that were made in the very first open source NGINX still affect why we are so agile and so performant in those containerized environments today.

**[0:25:56.3] JM:** A service mesh has the control plane and the data plane. These NGINX side cars in this case would be the data plane. If I was building my service mesh out of NGINX, what would be my control plane?

**[0:26:14.3] AM:** If you were starting from scratch today, typically a service mesh providers, whether you're choosing one that's more of a smaller, lightweight service mesh, or you're choosing one of the larger, more full-featured service meshes like it's steel, for example. Typically, they go hand in hand in the control plane and the data plane. They don't have to by any means. There are service missions available today that support a plug-and-play model for data plane.

From the NGINX perspective, some of the work that we're doing in the service mesh space is focused on being able to take advantage of some of the underlying features that we already have available to us in NGINX through the data plane. That means that we natively understand and we can natively handle some of the critical components of a data plane and a service mesh, such as SSL key handling, the ability to reload, dynamically configure SSL certs without interruption, the ability to completely change configuration spec with zero interruption to the traffic that's currently going through that sidecar.

Because we can take advantage of those functions at the data plane level with NGINX today, then we have the ability to write some of that specialized policy management two-lane in the control plane that you would need. Again, I'll stick to MTLS is a great example. Having services that understand how NGINX consumes SSL Certificates and how it can populate those certificates without interruption, if you can build that level of logic into the control plane because you have that tightly coupled NGINX world, for example, then you get performance advantages, you get management advantages at the control plane level, because you're able to understand what the data plane expects.

From us, it's very, very critical to have a very homogeneous control plane and data plane, because we can take advantage of so much that's already there in NGINX. When we talk about control plane functionality, we're talking about how does the admin interact with the mesh, how does the mesh inject sidecars, how does it manage and consume policy and push that policy down to the sidecar for consumption, that's where we can start talking about really, really cool

opportunities to have an NGINX control plane, NGINX data plane, very, very tightly coupled, very, very efficient and small, small footprint. That's our thinking in NGINX world about whether you should use a data plane, sidecar off the shelf, or use one that you can truly understand and integrate with natively.

**[0:28:47.8] JM:** In this service mesh model, when you have a sidecar running next to all of the services in your infrastructure, does that get expensive either from a resource consumption standpoint, or from a cost perspective?

**[0:29:04.9] AM:** Generally no. I mean, it's always depends on the type of traffic, the type of scale. In a consumption model, there are pros and cons in general. The pro is that you can only use what you need. The con is that it's not always price optimized for a huge amount of consumption. As with any good technology answer, it depends, right? Generally, no. I mean, we're very fortunate that containerization has this concept and structure of a sidecar already. We have tooling available to us in these containerized environments, like IP tables that make the deployment and just the raw functionality of a sidecar very, very inexpensive to operate.

Once in place, once a policy is defined for injection for example, then it just is what it is as we say. When we talk about footprint and what we're adding to that service as far as containers, again very, very fortunate that NGINX is so highly optimized to run in that environment. If we go to the extreme of making the most efficient container that we can as a sidecar, we can get NGINX down to a tiny, tiny runtime footprint, so very, very inexpensive for compute resources, virtually no storage is necessary in that environment.

As long as we really focus on optimization, which we do in the NGINX service mesh world, when we focus on making the most efficient sidecar, we can absolutely remove the majority of that cost burden. Again, just like anything in life and technology, there are always times when you might be paying some small percentage sacrifice for that sidecar, but you're still getting in the huge bonuses of full security, no service can get, or no traffic can get to that service. It's a balancing act.

**[0:30:51.3] JM:** The other approach to service mesh that has been tried is this embedded library approach, where somebody installs a library in their application, rather than having this

entirely separate sidecar container running alongside their service. Do you have any architectural perspective on the embedded library approach of service mesh, versus the sidecar approach?

**[0:31:23.5] AM:** I'm personally a much more fan of the sidecar approach. I'll give you a real-world example. We are working with a customer a few weeks ago who brought to us that exact use case. No joke, it's exactly what he said. He said that the downside to that for them is that one, their application developers have to consume that library. The application has to understand it, they have to test for it, so they're QE, they're in testing, had the factor in, all of the functionality of that library that was embedded in their application.

The downside beyond just having that functionality different in that workflow difference was that they didn't control that library. It was actually built by another team. If they happened to introduce something different in their application that then broke that library interaction, that was only discovered during QE or E-to-E testing. They then had to go back to the team that owned that library and say, “We have this problem. Can you help us? Can you help us debug it? Can you tell us why it's not working? Why is our intercommunication no longer functioning?”

For them, it was a huge burden. They were very much looking for the sidecar model, because they wanted to offload all of that interaction. In their case, it didn't necessarily mean that they would take ownership of things like security policy. What it did mean is that that particular team who were all frontline engineers, that team no longer had to factor in another component of their application. All they had to do is say, when we get a packet, we then handle that packet.

To me, it was such a really good eye-opening use case where the value of the sidecar is that we can offload all of that, not just from the functional perspective, but also just to get it out of the minds of the developer. Developers don't need to add it to their test. All they need to do is say, it's your responsibility to get me a packet. Once I get a packet, I'll take it from there. That was a huge burden off their shoulders and the singular exact reason they were looking at a service mesh.

**[0:33:23.3] JM:** If you have that sidecar and all the network requests are going through the sidecar, is there significant latency that's added if all of your network requests have to go through this additional hop to container sidecar?

**[0:33:40.4] AM:** Again, it always depends. Again, it's very similar to asking that question. If I put a load balancer in front of my data center, am I going to slow things down? Over time and almost always, the answer is generally no, because you inherit the ability to understand the traffic and do things differently when with the traffic then the application itself may need to process. SSL offloading is the quintessential example for any architecture, whether it's data center front door, cloud, micro-service, ingress, east-west.

If you can offload SSL processing from the application, then the traffic does have an extra hop. What you're doing with that traffic is taking a huge amount of CPU and computational load off of the application. Even if there's a tiny 5% latency introduced with a reverse proxy for example, you're going to get a 20% benefit on computational speed downstream to that application, because the application is no longer having to deal with SSL handshakes, decrypting SSL traffic, feeding that traffic and then doing an SSL destruction.

You're taking that huge computational load off the application, moving it to the sidecar, moving it to a tool like NGINX, which is designed to very efficiently handle SSL, so you're getting a much greater benefit. The question of latency always comes down to testing, environment, what do you need the reverse proxy to do, but generally no. It's 100% worth any trade-off you might see in the networking side for an extra hop.

**[0:35:12.9] JM:** The proxies can be scripted. You can have NGINX proxies run scripts that are written in Ngin script or in Lua. Is there a use case where scripting in the sidecar proxy for a service mesh would be useful?

**[0:35:34.1] AM:** Yeah, absolutely. I mean, you named two of the options in JS and then Lua. Both of those tools are designed for traffic management scripting. For example, if you needed to look inside a packet and ask what was the source address for this packet, we can do that natively in NGINX. If you need to go one level lower and say, “Well, I want to look at what the actual payload is. I want to understand more characteristics of the traffic to get into some of

those specialized deployments," those are excellent use cases for NJS, for Lua, again whether you're talking sidecar or front door of a data center or cloud.

Then NGINX itself has a modular architecture. If you need to either go deeper into a particular function or performance is 100% paramount, NGINX fully supports customers writing their own modules in C, compiling those to run them as a dynamically loaded module with NGINX. That's a third option where customers can add very, very specialized functionality with high, high performance. Sure, absolutely. There's always a need to be able to do more with the traffic than just a simple proxy in, proxy out and that's where NJS, Lua, or even custom modules will allow you to do that with an NGINX sidecar.

**[0:36:50.2] JM:** The historic use case of NGINX was as a load balancer, or a reverse proxy. It might be sitting in front of a variety of different services and then you might have requests routed for NGINX to one of your different services. In the service mesh model, you're adding an NGINX proxy next to each of your services. In some sense, that's fulfilling the same role. I'm wondering if there were any significant re-architectures or changes to NGINX itself that needed to be implemented to satisfy that shift from the front, or that edge to being co-located next to the actual services.

**[0:37:42.9] AM:** The short answer is no and we're very, very lucky as I mentioned earlier, that some of the design decisions early on with NGINX give us impact today in designing that sidecar functionality. You were 100% correct in the tried and true architecture of NGINX as a reverse proxy, which means we take connections in and then we distribute them, we proxy them back to the application.

When you're talking about east-west or service-to-service traffic, you have that functionality, so one service will call the sidecar, in this case, NGINX of another service. That sidecar is functioning as the reverse proxy to the application. Then as soon as that application needs to talk back to another service, we think of that as egress traffic for east-west. What I mean by that is that if an app – when an application spawns a connection to another application that's running as a service, the communication is happening between the two sidecars in a service mesh environment.

An NGINX Plus instance, or an NGINX instance in that case is functioning both as the reverse proxy for ingress traffic to the service, but in some essence a forward proxy back out to the other service when new connections, or new traffic is initiated back out to that service. We're very, very fortunate that NGINX at its core understands that proxy means a packet comes in, a packet goes out. When we look at topology architectures, it doesn't care whether we would think of that as a reverse proxy, or someone else might think of it as a forward proxy. As long as we know where the traffic came from, where it needs to go, it is 100% bi-directional by design.

For that alone, there's no design considerations, no coding considerations we have to worry about when using NGINX as a sidecar. All of the feature functionality that are needed for that core east-west traffic management that you would expect like MTLS, the ability to add access control lists, whitelist, blacklist, service level authentication, all of that is built into NGINX. All of it is bi-directional. Very, very lucky that we had such a robust data plane to work with.

[SPONSOR MESSAGE]

**[0:39:54.8] JM:** Vettery makes it easier to find a job. If you are listening to this podcast, you are probably serious about software. You are continually learning and updating your skills, which means you are staying competitive in the job market.

Vettery is for people like you. Vettery is an online hiring marketplace that connects highly qualified workers with top companies. Workers and companies on the platform are vetted and this vetting process keeps the whole market high-quality. Access is exclusive and you can apply to find a job through Vettery by going to vettery.com/sedaily. That’s V-E-T-T-E-R-Y.com/sedaily.

Once you are accepted to Vettery, you have access to a modern hiring process. You can set preferences for location, experience level, salary requirements and other parameters, so that you only get job opportunities that appeal to you. If you have the right skills, you have access to a better hiring process, you have access to Vettery.

Check out vettery.com/sedaily and get a $300 signup bonus if you accept a job through Vettery. Vettery is changing the way that people hire and the way that people get hired. Check out

vettery.com/sedaily and get a $300 signup bonus if you accept a job through Vettery. Thanks to Vettery for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

**[0:41:31.2] JM:** What's the relationship between a service mesh and Kubernetes in more detail? I know you said you think of service mesh as a higher level configuration and management plane on top of Kubernetes. If I already have a Kubernetes cluster, what is a service mesh going to give me on top of that?

**[0:41:52.3] AM:** Yes. As you mentioned, if you have a Kubernetes cluster already and your apps are deployed and everything's working, you have the core tenants of service-to-service networking. You have a virtualized network, you have IP endpoints, you have a platform that understands what an IP endpoint is no longer responsive or is overloaded and needs to scale and you also have the ability to scale those services directly through Kubernetes.

Kubernetes gives you all of that day one networking, or table stakes networking that you need for any service and it conveys cluster to talk to another service, or to accept ingress traffic, or generate egress traffic. All of that, assuming that you have a functioning cluster and a network as part of that cluster, then you get all that.

What a service mesh brings is that policy level that you want to apply on top of those service-to-service communications or that east-west traffic. The three core tenets of a service mesh that we all live and breathe by are security/the ability to define policy at a security level, so set things like MTLS policy, set access control, set service level authentication. That's number one. Number two is orchestration, so the ability to force that that policy follow the application, even if Kubernetes decides to spin up a new set of services, the ability to orchestrate and attach that security policy to those new services. That's something that a service mesh brings to Kubernetes.

Then third is visibility. While Kubernetes does a great job of supporting traffic visibility within a cluster through its own tools, through support for standard tools like Prometheus, what it doesn't allow is that second level application awareness visibility. I need to be able to look in

Kubernetes and say service one is allowed to talk to service two. Service one is not allowed to talk to service three and I'm not seeing traffic go between service one and service three. That's the third core tenant of a service mesh is that visibility and being able to double click down and say, “Why am I seeing traffic, or why am I not seeing traffic between services based on the policy that I've applied?”

To summarize, Kubernetes brings you all of the core functionality you need. Service mesh brings you policy intelligence, policy orchestration, policy visibility for all of that east-west and service-to-service traffic.

**[0:44:17.6] JM:** Who should actually use a service mesh? I mean, if I'm a small company with just a few services, maybe it's not worth it. Maybe it's only worth it if I'm a gigantic company with a ton of services. What's the threshold of complexity that I cross where I actually should be using this and it's not just extraneous extra architectural stuff that I'm not really going to care about?

**[0:44:46.7] AM:** I think the first answer and working in this space and loving this technology and being super excited about what opportunity it brings us down the road for east-west and service-to-service traffic management, I admit I'm a little biased, but I do think that anybody who is dealing with containerized service-to-service communication or networking should be looking at service mesh, because of the policy, definition of the policy enforcement and the visibility that these tools bring.

I think anybody who wants to have any level of control, awareness, visibility over that service-to-service traffic, a service mesh is perfect for you. You mentioned complexity, any new tool especially in a highly complex environment like Kubernetes, any new tool is going to introduce some new level of complexity. In the service mesh world, I think it is 100% worth digging into that toolkit, because of the value you get from a mesh is just so powerful, regardless of what mesh you end up using, just the mesh technology itself.

I think first answer is anybody who has to deal with service-to-service, east-west traffic management, that Kubernetes or containerized environment. The second answer though is what size or what criteria company should hit first. I think it depends on who is responsible for

the day-to-day ownership of that east-west traffic management. If you're what we would call a younger, more DevOps focused company, or startup type company, typically that traffic management function is relegated to the engineers or developers, the DevOps who have to deal with building applications that need to talk to other applications. Those type groups are absolutely perfect for a smaller, very agile service mesh like the one that NGINX is building.

If you're talking about larger companies, like FinTech where more of a traditional net ops type of group might own the networking responsibilities within a cluster or cluster-to-cluster, which is typically what we would have seen 10 years ago with one team owning all of the networking in a data center, or all of the land-based networking, in those environments where a service mesh is on the table, those companies so far anyway tend to be looking at the larger orchestrated type of service mesh like F5's Aspen mesh for orchestrating and providing visibility to Istio.

When customers typically ask, which one should I look at? I say, it comes down to who's going to own it. If the developer owns their own cluster and they're responsible for their own east-west cluster traffic, a smaller service mesh like what NGINX is building seems to make more sense. If it's a net ops team and they need to adhere to a higher level networking security policies and network availability, something like Aspen Mesh and Istio typically makes more sense.

**[0:47:32.5] JM:** If I want to deploy a service mesh to a gigantic enterprise, what does that process look like? What's the gradual installation and onboarding process?

**[0:47:46.4] AM:** If we're talking about that second category of a company that has a more network-focused team who's responsible for rolling that out, they typically own what I would call cluster level up. When a developer and engineer, or a dev team requests a new cluster access, they might be requesting that cluster to develop their one app, so the mobile banking app for example, or a component of that mobile banking app. In a typical environment, that cluster environment would usually be owned by the networking team who also owns the responsibility, or the right to access to that cluster.

In that case when someone got a new cluster, it would come with service mesh policies. It would be configured to already be a part of the mesh. MTLS would already be a policy that could be applied to those applications. For a larger, again we'll talk FinTech for a second, a larger

FinTech company, if you're building out or starting to look at that larger platform service mesh, you're looking at cluster level integration, part of larger project level integration in single point management.

If you're in the startup world, deploying a service mesh is just ridiculously easy if you own that clustered environment. You simply deploy it as an additional Kubernetes application in essence and then you start applying policy directly within that cluster. Again, two different answers, larger company is going to push down that policy as part of a cluster admin rights level deployment. Smaller companies who own everything within a cluster, they're just going to deploy the mesh alongside their application in that cluster.

**[0:49:20.7] JM:** On that note, for the earlier, the newer companies that have the ability to just add it to their greenfield, brand-new application, is it worth it? I mean, that's some additional complexity, or are people actually doing that?

**[0:49:36.5] AM:** They are, absolutely and definitely worth it. Again, it's back to that level of policy definition, policy enforcement and visibility, where I do think it's worth it. If you think of a developer who gets a new cluster as a sandbox, for example and they start building their app, they are not focused on those network level security requirements typically. They're not focused on having to deal with SSL handshakes or SSL communication. They're just going to deploy and write the best possible application they can, knowing that Kubernetes is going to assign them an IP endpoint that's dynamic, that's defined by service name that can fluctuate. All of that happens outside of their application domain.

They just ask for an IP endpoint and they get one. What a service mesh brings though is the ability for them to apply more granular and secure policy to those IP endpoints that even though their application may not be aware of, or not need to be aware of, as an application administrator, whether that's the same person, whether that's a DevOps team that's just responsible for administering clusters, the mesh bringing that level of granularity is so important for things like compliance, for reliability, for assurance so they know without question which services are talking to which other services and which one should or should not be talking to those services.

The mesh is totally worth any initial complexity that might be addressed, because just like everything else in the containerized world, the Kubernetes world, everything can be CICD'd. As soon as the mesh is built and it's configured and there's a CICD pipeline around it, interacting with the mesh becomes no different than interacting with Kubernetes natively.

**[0:51:20.2] JM:** Now coming back to the very large enterprise, if you have a large enterprise with thousands of engineers, you're not going to be able to deploy one single service mesh to all of the different services across that enterprise. I mean, maybe you can, but I think it's more likely that you have a multitude of different service meshes. Maybe you have a service mesh per team, or maybe you have some federation model. I don't think I've actually seen that in practice, but do you have any perspective for what the different service meshes within a single organization, how many service meshes there are and whether those service meshes need to be able to communicate with each other in some special way?

**[0:52:09.6] AM:** It's a great question and I'm glad to hear your example of federation is not something we've seen yet. What's interesting about this space, about anything to do with containerized networking right now is that it's still early days. It's very, very quickly evolving as we know. It's still at a point where we don't really talk to a lot of customers who have done full-blown production deployments of service meshes yet. That's either category, that's enterprise-wide mesh, like Aspen Mesh and Istio, or that's on a per cluster mesh. I'm using some of the smaller, more focused meshes.

We haven't really run into a lot of customers that are doing it at production. Now there was one company I worked with in Australia who did roll it out into production, they did it from the enterprise perspective, so they had standardized across the board on Istio and they said, "We're going to manage it for all of our engineers, unified service mesh. This is it. We'll give you deployment models on how to interact with this, but you never have to worry about it." I think that was a great model for them to choose.
Of the 100, 200, 300 customers that I've worked with over the past three years since service meshes even been remotely a conversation, they're really only – you can count on one hand how many people are doing it in production. That said, I think the nice thing about this space is that there's a model that's available and that works in any condition that a customer wants. To use your example, there are no technical limitations between rolling out an enterprise-wide

service mesh, configuring that mesh to support things like cluster-to-cluster communication, so abstracting MTLS policy, access control policy, service authentication policy, abstracting that up one level and handling it on cluster-to-cluster. That technology exists today. That can happen today.

There's no technical limitation at all to deploying something like that. But then, allowing engineers to deploy their own meshes and their own clusters and their own sandboxes, no restriction there. I think one of the cool things that we'll see over the next year, maybe 18 months or so, as these companies start to think about those things, we start to elevate the functional need of an ingress controller and some front door service that takes the burden on I'm going to talk to my enterprise mesh on the top, so cluster-to-cluster communication, adhere into that enterprise-wide mesh policy.

We're going to start to see that start to tap and maybe more at the ingress level. Then once ingress takes that traffic and it says this cluster is now allowed to accept this type of traffic, I'm then going to pass it down to my local mesh and say, now service-to-service traffic is on you. That mesh can be anything. Then when the mesh says, “Okay, I'm cool with this. I need to send traffic to another cluster,” it simply passes that back up to its ingress and say, “Now, this is your problem. Let me know when you get a response.”

That's what I think is so exciting about thinking of network service meshes and cluster-to-cluster service meshes, but still supporting that really low-level application stack side car model can absolutely co-exist.

**[0:55:13.8] JM:** Well Alan, it's been great talking to you about service mesh and NGINX. I just want to close off with one more question that's not really related to NGINX, but just this change to our industry that is a result of the virus, the COVID-19 virus is pervasive. I'm trying to ask the different people that I'm interviewing just how it is affecting their work and just get a little bit of a mosaic for how it's affecting the technology industry. I'd love to hear how COVID-19 has affected your work.

**[0:55:52.7] AM:** Sure. I mean, that's – you were spot-on. It's affected all of us in ways that we wouldn't have guessed two months ago, or three months ago, right? It's fascinating. I'll give you

my personal perspective from the corporate and then the very personal how it affects me day-to-day. NGINX prior to acquisition, but even today within F5, NGINX is a highly distributed company. We've had engineering offices globally. On my team, the service mesh team, we have engineers that are based in Ireland, that are based in the US, in Seattle, in San Francisco.

For two years, I was based in Australia and covered APCJ, but it never felt like I was remote or isolated, because that's just the culture of NGINX. We understand that innately as NGINX employees, it's a distributed a work force. We know that we can do our job anywhere. Honestly for two years, covering all of Asia, I did literally my job from the airport, or from a plane, or from customer offices every day. We have some of that.

It's like, yeah, this is a little extreme. On a corporate level, it's still the same. F5 is the exact same culture. Very, very distributed, very supportive. F5 has amazing technologies that enable remote working, like very solid, robust, SSL VPN tools and stuff that we all use every day within F5. We've had very little, thankfully very few functional barriers to this change in working.

On a personal level, it's fascinating. The service mesh team as I mentioned is distributed. Our stand-ups are done through Slack already, so we still do Slack stand-ups when we talk about things like sprint refinements. We still do those as a virtual team and a virtual conference room. None of that has changed. The biggest change for me is I have a very small apartment condo and see how it all – getting used to having these types of meetings, either from a very small desk, or from a bedroom, because it's that small, that's minor, minor stuff that is just becoming second nature at this point. Very, very fortunate in our space and the DevOps space in the engineering space that I think we're working very well in this environment, or at least as well as we can hope.

**[0:57:55.5] JM:** Okay, Alan. Well, thanks for coming on the show. It's been great talking.

**[0:57:57.8] AM:** Excellent. Thanks so much, Jeff, for having me. Great conversation. I mean, really, really cool technologies. Thanks.

[END OF INTERVIEW]

**[0:58:11.8] JM:** We're happy to have NGINX as a sponsor of Software Engineering Daily. NGINX is not only used for routing and load balancing, but also API management, API gateway, service mesh and other applications. To hear more about what Alan Murphy discussed around service mesh, visit NGINX.com/sedaily to see webinars and other resources that describe how NGINX works and how it can be used to solve problems in your infrastructure.

[END]