EPISODE 800

[INTRODUCTION]

[00:00:00] JM: Uber's infrastructure supports millions of writers and billions of dollars in transactions. Uber has high throughput and high availability requirements because users depend on the service for their day-to-day transportation. When Uber was going through hypergrowth in 2015, the number of services was growing rapidly as was the load across those services. Using a cloud provider was a risky option, because the costs could potentially grow out of control.

Uber made a decision early on to invest in physical hardware in order to keep costs at a reasonable level. In the last three years, Uber's infrastructure has stabilized. The platform engineering team has built systems for monitoring, deployment and service proxying. Developing and maintaining microservices within Uber has become easier.

Prashant Varanasi and Achal Shah are engineers who have been with Uber for more than 3 years. They work on Uber's platform engineering team and their focus is on the service proxy layer right now, which is a sidecar that runs alongside Uber's services providing features such as load balancing, service discovery and rate limiting.

Prashant and Achal join the show to talk about Uber infrastructure, microservices and the architecture of a service proxy. We also talk in detail about the benefits of using Go for critical systems infrastructure as well as some techniques for profiling and debugging in Go.

Before we get started, we have a few events coming up. We have a conversation with Haseeb Qureshi at Cloudflare on April 3rd, 2019. You can find out about that by going to softwareengineeringdaily.com/meetup. We also have a hackathon for FindCollabs, the company that I'm building. The FindCollabs hackathon can be found by going to softwareengineeringdaily.com/hackathon. It's both a virtual hackathon and an in-person hackathon. We're meeting at App Academy on April 6th, 2019, and you can sign up. We're going to have some food. We're going to hang out. We're going to hack on some interesting projects. It's for anybody who has a project they want to hack on, whether it's an art project, or an open source software project, or something relating to music. It is for technically-minded people, just like the listeners of Software Engineering Daily. You can find out more about FindCollabs by going to findcollabs.com or findcollabs.com/hackathon. The hackathon has a $5,000 price purse. So you're not hacking for nothing.

With that, let's get on with today's episode.

[SPONSOR MESSAGE]

[00:02:53] JM: Today's show is sponsored by Datadog, a monitoring and analytics platform that integrates with more than 250 technologies, including AWS, Kubernetes and Lambda. Datadog unites metrics, traces and logs in one platform so that you can get full visibility into your infrastructure in your applications.

Check out new features like trace search and analytics for rapid insights into high-cardinality data, and Watchdog, an auto-detection engine that alerts you to perform its anomalies your applications. Datadog makes it easy for teams to monitor every layer of their stack in one place. But don't take our word for it. You can start a free trial today and Datadog will send you a t-shirt for free at softwareengineeringdaily.com/datadog. To get that t-shirt and your free Datadog trial, go to softwareengineeringdaily.com/datadog.

[INTERVIEW]

[00:03:58] JM: Prashant and Achal, guys, welcome to Software Engineering Daily.

[00:04:02] PV: Thank you for having us.

[00:04:03] AS: Yeah, we're glad to be here.

[00:04:04] JM: You're both engineers at Uber, and I'm looking forward to talking about several different infrastructure topics within Uber. Three years ago, we did a show with Matt Ranney, who is a senior engineer at Uber, and that show has always stuck out for me, because around that time, Uber was developing systems to deal with the rapid scalability and it was indeed rapid. Perhaps the most rapid scalability that any software engineering organization has faced.

So you've both been at Uber for three years. You are around in that time. How has the infrastructure changed since then?

[00:04:43] AS: Yeah. So things have definitely changed quite a bit since you did that interview with Matt. I think when Matt spoke to you, that was really at the peak of the crazy growth in computing needs at Uber. So we were really focused on the short-term. Every month, there would be a new system that needed to be rapidly fixed so that it could survive just the next 3 or 6 months of growth.

Since then, I think through a combination of rebuilding systems just with elbow grease and good engineering, and a little bit more growth in infrastructure hiring at Uber, we've managed to retrench a lot of those systems and things are much more stable now. That gives us the leeway to think one or three or five years ahead and really build systems with a longer time horizon.

[00:05:37] JM: One reflection I had from watching Uber around that time and talking to some engineers who work there was there was really a psychological exercise that I think needed to be done because if you're going through that level of hypergrowth, your inbox is overflowing. You're suffering from alert fatigue. It's very hard to think longer term, let alone even shorter term. It's just very stressful.

I saw a talk from Achal where he said going through hypergrowth early on, "Everything was on fire," which is consistent with what Matt Ranney was saying around that time. Do you have any reflections on how to psychologically adjust as an engineer in that kind of experience?

[00:06:24] AS: I think the best thing was having a team of people around that I trusted and that I was lucky enough to work with. Having that team really always helping each other set priorities and remember that some stuff was going to break every day or every week, and the important thing was keeping the business moving and focusing on the various breakages and the problems that we're going to bring the whole business down. So if we got paged on and off for side systems or things that were kind of optional, that was okay and that was just part of the cost of doing business.

[00:07:06] PV: Yeah, I agree with Achal. I think it's really a matter of prioritization and kind of accepting that you're not going to be able to fix everything about the current system and you need to focus on what really is important to keeping the business going, and then using the other cycles to come up with a solution for the longer term. So one example is when you're on call, you're really, really focused on solving the immediate needs. Anything that's affecting the business gets your full attention. But when you have a larger team, the other people on your team, they can help you fix some of the less important issues, but it's almost better for them to focus on the longer term solutions and make progress towards fixing these problems in a more systematic way rather than patching individual issues with the current architecture. So sometimes you do need to focus on the longer term architecture rather than the immediate needs as long as those immediate needs aren't affecting the business.

[00:08:04] JM: Okay. Anyway, getting to infrastructure, why was it so hard to scale? I mean, we have "auto-scaling infrastructure" from cloud providers. Why didn't that solve all your problems?

[00:08:18] AS: That's a really good question. Part of the answer is that Uber as a business was large enough and growing fast enough that how much money we spent on infrastructure actually mattered. So if you look at, for example, some auto-scaling cloud thing, like Google Cloud Functions of AWS Lambdas, per unit of compute, they're actually fairly expensive. Uber made a decision pretty early on to build the core of our platform on hardware that we owned in data centers or our colos that we rent for multiyear periods of time.

So that meant that we weren't able to use things like Lambda or Cloud Functions when they first came out. I would also say the core of the problem around when Prashant and I joined was not usually burstiness, and certainly not burstiness on a day over day level. The core of the problem was usually just baseline growth.

So you might go from needing a hundred servers to solve some problem today, to needing 200 six months later, and needing 500 a year later at a steady state. That's stressed a lot of our early designs. One early thing that started falling over all over the place was the database. It's very difficult to horizontally scale most databases. So even if you can add more compute power to the application logic, it got very difficult to scale the databases to the volume of data, the number of connections and the number of queries that we were trying to run against them.

[00:10:06] JM: Another element of the complexity that I didn't understand until saw Achal say it in a talk that he was giving, was that from the outside, Uber looks like it's one giant business, but it's actually several hundred small local businesses across the world. Different geographies have different versions of Uber, and thinking about the complexities of sharding the business as well as sharding the infrastructure really makes my head spin.

[00:10:42] AS: It is certainly complicated. I think, luckily, things are somewhat more rationalized than they were when I originally gave that talk. But it's still true that the general managers of Uber's business in different cities from a technical perspective have a lot of control over exactly what toggles and settings are enabled and configured for their city, especially for the business layer of Uber's software. That means that you can exercise very different code paths when handling traffic from different geographies.

Particularly for monitoring, that can be unpleasant, because you might find out that the code path that has a bug in it is only exercised in cities that are 12 hours different from where you live, and so all the bugs there are surfaced in the middle of the night for you. Luckily though from an infrastructure scaling perspective, our infrastructure is homogenous and is generally not concerned with that level of sharding.

[00:11:51] JM: Let's get to talking about the services within Uber. So another reminiscence from that discussion with Matt Ranney was the number of services that Uber has running, which is not unfathomable for a large company. Many large companies have lots and lots of services. What principles do you have around architecting services within Uber?

[00:12:17] PV: I think a lot of our microservice architecture has evolved from some of the problems we've ran into with the monolithic architecture. So the monolithic architecture coupled a lot of things, coupled deployments. So if you wanted to ship a new feature, you had to wait until the monolith was deployed. It coupled reliability, so if something low tier was crashing for some reason, you are bringing down the monolith, which was affecting the business and more business critical functionality.

So we've evolved from there, and that's where I think that's why we have a lot of these services, is we want to isolate them as much as possible. So that means that when something less critical to the business is down, it shouldn't affect the rest of the business. So part of it could just be, "Is this important enough that it can be coupled with some other important functionality? If not, let's avoid the risk. Move it into a separate microservice." With that, you also get the ability to deploy and move a little faster too.

So it allowed us to move faster in maybe the less critical microservices, whereas the critical microservices could be deployed a little bit more slowly instead of deploying over, say, a couple of hours if it's something really important. You might even spend a week doing that deploy. So that kind of ability to separate how you deploy, how you monitor, how quickly you need to react to an issue in that service. Things like that have helped us decide how to separate our microservices.

[00:13:51] AS: I think that's absolutely true on a technical level. I think another aspect of Uber's growth that was really important was that we were hiring an onboarding new engineers and new teams and even new offices around the world relatively quickly. So one other big benefit of microservices is that it lets Uber engineering as a whole organization set some pretty lose guidelines and let individual teams decide how they want to do release management and how they want to feature flags and code style and a bunch of other things that often end up being controversial when you have large groups of engineers. This let us optimize for having individual teams maximize their own feature velocity and their own development. We went into that knowing that that would come at the cost of uniformity across the whole stack.

[00:14:51] JM: How do services at Uber communicate with each other?

[00:14:55] PV: We've actually gone with a model that's similar to something like Envoy that you see now, but we actually had this model probably for 4, 5 years now at Uber, where rather than have microservices talk directly to the other microservices, they go through a load balancing proxy that's typically deployed on the host. So every service will make a request to say, "HTTP, local host, port 10,000," and port 10,000 would mean I want to talk to the user service. Then the load balancer would have all of the logic for service discovery, health checking and some sort of load balancing strategy. That would all happen out of process, and that actually let us evolve our microservice architecture for different languages and frameworks without having to build all of the logic of our service discovery and load balancing into every language.

[00:15:51] AS: Of course, in the year since we started that system, it's evolved significantly. At this point, I think it looks a little bit more like Envoy, or some service mesh-like things. So you no longer address services by port number. They're all addressed by name, and there's kind of a sophisticated service discovery system that's tied into deployment and you can delegate routing decisions to other processes and there's a bunch of complexity like that that's not used by most services, but does enable some fairly complicated use cases.

[00:16:29] PV: One of the advantages of doing this kind of routing out of processes similar to Envoy. We have a observability consistently across all of our services. So I think some of our most frequently used metrics actually come out of our routing stack, because it emits successes, latency numbers, all sorts of useful information consistently for a microservice communication.

We've found that the out of process load balancing model has really helped us give us a consistent view into what's happening, but at the same time giving us freedom to implement the microservices in the appropriate language for the functionality.

[00:17:12] JM: So what I understand about the service proxy model, at least from my conversations with different people who have used Envoy in this architecture, is you have your services that run often in containers, and then you've got a separate container that you schedule next to this main service container, and that separate container is what's called a sidecar container. The sidecar container is holding the service proxy, which will often be something like Envoy. The responsibilities of that service proxy include service discovery, routing between different services. It can give you a lot of analytics. It's basically things that you want attached to every service so that we don't have to fold that logic into the core service that's sitting in that core service container. How does that align with your architecture?

[00:18:10] PV: It is very similar. So we've done the same thing where exactly as you said. We don't have to build all those functionalities into the core service, but it's happening outside a process. The one difference is, currently, we actually have a single deployment of the proxy on the host rather than having one per service container, and that's actually something we're looking to change overtime.

So initially, we built it that way for – Just because we thought we could get the same functionality of out of process load balancing analytics without having to pay the cost of an extra container for every single service container. So we're able to share the resources, which is connections to the service discovery platform were kind of coalesce for all of the services on the host. So in terms of resource usage, we think this was a bit more of an efficiency win, I guess. But overtime, we're actually moving more towards the side car model mostly for other reasons that we haven't actually discussed yet, which is things like security. It's great to be able to provide security in a consistent way similar to the load balancing without having to build that into the core service. But once you're doing security, you really need the identity of the service and you don't want to have to deal with multi-tenancy, especially from a security perspective. It just complicates the architecture and can also reduce some of the security benefits that you can get. So we're actually evolving our architecture to more of the proxy sidecar per container.

[00:19:45] JM: Interesting. So why wouldn't you able to just do security policy management with that model where you have the service proxy singularly sitting on the host?

[00:19:58] PM: Yeah, it's partially an integration issue as well, which is now if you have a service proxy that's global for the host, it now needs to be told which security policies to use for which service. Actually, one of the biggest problems we've had is you can't easily identify the service that's making a connection to you. Typically, we use TCP for connections between the service and the proxy. That gives you very little knowledge of who is the remote service that's talking to you. What security policies apply to that service? So what tokens can you add in? What authorization checks can you add? There's a bunch of questions around that that are harder with TCP.

[00:20:41] AS: I think part of the problem is that the point of doing this authentication in the proxy is so that you don't have to bake this kind of logic into the service. If every service needs to have some truly secure way of verifying identity and connecting to the proxy, you've solved most of the problem of having services connect directly to each other correctly.

In a sidecar model, I think one of the benefits that we're looking for is having the connection between the service process and the proxy be private to those two processes. So you don't have to do a complicated security handshake there, and you could have all the security logic live in the connections between the proxies.

[SPONSOR MESSAGE]

[00:21:35] JM: Deploying to the cloud should be simple. You shouldn't feel locked in and your cloud provider should offer you customer support 24/7, because you might be up in the middle of the night trying to figure out why your application is having errors, and your cloud provider's support team should be there to help you.

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

At linode.com/sedaily, you can get started with two gigabytes of RAM and 50 gigabytes of SSD for only $10. There are also plans for cheaper and for more money. Linode makes it easy to deploy and scale your application with high uptime and simplicity. Features like backups and node balancers give you additional tooling when you need it.

Go to linode.com/sedaily to support Software Engineering Daily and get your application deployed to Linode. That's L-I-N-O-D-E.com/sedaily. Thank you, Linode, for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:23:04] JM: I think this is a good time to discuss what you guys actually work on. So the last few shows that I've done with Uber engineers have been around things that I would call platform engineering. So we did a show about M3, which is the logging service, or logging and monitoring service. This is something that's going to be on – Every service will adapt it. Then we did a show about Peloton, the scheduler that every service gets scheduled on to. What do you guys work on? Are you working on individual services or platform engineering type things?

[00:23:41] PV: So both Achal and I have been platform engineers for the past few years. So I actually worked very heavily on the service proxy that we were talking about earlier, and Achal was a big part of that proxy as well. Then we kind of worked on closely related areas, but I work more on basically the service discover load balancing, thinking about the sidecar service mesh solution, bringing some of the benefits of what's going on today with Istio and Envoy outside of Uber. Bringing that to Uber today.

So that's what I tend to focus on. The reason this kind of overlaps well with Achal is all of these infrastructure is written in Go. So when I first started Uber, Go was pretty immature. So I help build some of the core networking libraries in Go and then we started building this proxy in Go and then the service discovery platform is now in Go as well. So Achal and I have kind of worked on these Go infrastructure platforms.

[00:24:47] AS: Then I joined the networking team to work with Prashant and actually help build that proxy and then have since gone on to work on our Go platforms and core libraries team. So that team's job is to build the service frameworks, and many, but not all, of the core libraries that go into almost every Go service at Uber.

[00:25:10] JM: We'll definitely get into Go and Go performance, because I've seen some great talks from I think both of you that regard Go. But since we're on the subject of this service mesh, service proxy layer, it's a curious discussion, because this is an emerging technology and it's hard for me at least as an outsider to really evaluate the state of thing and I guess make predictions about things.

I guess in your research of the service mesh and service proxy area, what has come to mind as a vision for how this is going to unfold. Do you think that Istio is going to be this very rich ecosystem that sits kind of like separate from the Kubernetes ecosystem and Istio is the service mesh that everybody should place a bet on, or do you think it's like a multi-mesh world, where some people want Linkerd for some reasons and some people want Istio for some reasons.

What's the importance of what the different proxies are? Should everybody just be using Envoy? I know that's a big bag of questions, but maybe you could just give me your diagnosis of the service mesh situation.

[00:26:26] AS: I think my perspective on that is a little bit zoomed out. I think my sense is that Kubernetes and this source of container scheduling systems are definitely the direction that the industry is moving in. So every cloud platform has a strong motivation to have a bulletproof way of addressing other containers by business purpose. In our case, we typically call that a service name. So it'll be like the user service, and the trip service, and something like that. But every company has their own way of mapping business functionality to some string.

So some service mesh-like thing I think is a critical part of Google and Amazon and Microsoft having a viable cloud container platform. So there will continue to be work in innovation in that field. My view as mostly libraries and frameworks person for the last year or two is that the right thing to do at this point is to bind very, very loosely to those things.

So you don't want kind of notions that are very, very particular to Istio or to Linkerd sipping really far into your business logic. You want to try and keep those at the edges of your logic so you can change them as necessary.

[00:27:50] PV: Yeah. I think we've looked at Istio, Envoy, of course. Right now it is still a little bit too early to make a bet that it's all going to be Istio or it's all going to be Envoy, especially at a larger scale. So we've looked at the sidecar and service mesh model. We think it definitely solves a lot of problems especially when you're dealing with a very diverse microservice architecture. For us at Uber, that's important, because we have 3,000 services written in different languages and frameworks and it's important for us to be able to rollout features to all of these services without having to rewrite all of them and without having to write libraries in platform features in every one of these languages as well.

We see service mesh as a great way to solve some of those problems. But at the same time, there are still some concerns around – Performance is one example. If we keep adding proxies between services, how much CPU are we spending just on proxying versus the actual business logic? I think as this evolves, we'll have more data and a better idea of where it's going. But right now, like Achal said, we don't want to tie ourselves to any one technology, but support this model of service mesh. At the same time, keeping in mind that maybe service mesh won't solve all the problems. Maybe there are some other ways of solving these problems while still getting some of the performance benefits of peer-to-peer.

[00:29:24] JM: To get to what you guys do, I worked at Amazon for about 8 months and one thing that's cool about working on one of these big companies, like in Amazon or an Uber, is that you come into work and you have an email in your inbox and it's like, "Your service has been upgraded, or your service proxy has been upgraded and all you need to do is install this dependency and then now you've got access to it," and it's a very magical feeling, which comes from just having a platform engineering team and a company that is 10-years-old or older often times, or slightly younger.

I'd love to know about the process of updating the platform that engineers are working off of, because, of course, most people who are writing service logic within Uber, I assume, don't really know much about the service proxy layer. All they know is they wake up one day and their services are more reliable. So can you tell me a little bit about that deployment process and the process of rolling out new platform features to people?

[00:30:35] AS: Yeah. I think I would say that Uber is still quite a bit smaller and less mature than an Amazon or a Google or a Microsoft in this space, but probably the best example of this is actually the routing kind of host sidecar that Prashant and I worked on that we're talking about. It was a replacement of the previous system we used, which was written in JavaScript and was just having problems scaling to the request volume and the number of services that we had built. By 2016, it was getting pretty creaky, and that's when we staffed up this project to replace it.

I think that was one of the better upgrades and rollouts that Prashant and I have had. We ended up building something that was compatible kind of at the network layer with the previous system. We rolled it out gradually and moved tiers of traffic over to the new system. So we started with some of the least critical stuff. Then we moved on a handful of the most critical things to make sure the new system handled our most demanding users, and then moved over the remainder of the traffic.

For the most part, there were no code changes required and it was pretty much as you described. We send emails to people just notifying them that were moving their traffic over and that they shouldn't see any problems. In fact, they should see a pretty significant performance and reliability boost. That's more or less what happened I will say that's the ideal case. It's what we aimed for, but we don't always hit the mark there.

[00:32:27] JM: So I guess it's just a matter of updating the service or I guess the container that's running on the host if you're updating like a service proxy. You just get the writes to that host and then you just rollout an update and probably gradually move the traffic over to that updated service proxy.

You mentioned that you did an upgrade from a JavaScript-based proxy to the Go-based proxy. I guess it's a good time to get into Go. So could you contrast JavaScript and Go from the standpoint of building a proxying layer like that?

[00:33:05] PV: Yeah. So I think Go really is made for solving problems like this. It really did make it a lot easier, versus JavaScript, it was designed for the browser initially. So we ran into some interesting issues with JavaScript. So one example is back when we were deploying this JavaScript proxy, we'd have trouble with the V8 garbage collector if the heap size grew to over 1.5 gigabytes. I think there was like a threshold after which the garbage collector behavior just rapidly declined and we were just better off restarting the process. It wasn't really intended for that large scale heap size, at least back then.

JavaScript was of course single-threaded, and I don't think we had service workers or anything back then either. That also is a little bit limiting, because now any CPU bound operation you do in your thread is blocking everything else in the process. There's nothing else you can do. So we often found that the proxies typically made up of two separate kind of functionalities. One is more control. So getting control messages about service discovery, where things are running, etc. Then there's the routing.

What we found is any slowdowns in the control pieces could affect the routing just because it was single-threaded and blocking the event loop. Those kinds of issues were resolved by using

Go, because Go kind of takes concurrency and parallelism as first-class concepts in the language. So that made it a lot easier to build a system where you could have a background thread that was interacting with our service discover platform. It receives updates. Maybe the update is huge, so it ends up blocking that specific thread for a while, but it doesn't affect the routing. The routing is still happening in some other threads. So there was no kind of spikes and latency when we're processing large amounts of control data.

It was probably easier to deploy Go, of course, because it's static. So it's much nicer to just build a single binary and just ship that binary to host rather than having to kind of have some sort of packaging and then distribution of the package. A little bit of additional complexity that you can avoid by doing that. In general, Go tends to be a lot more efficient for some CPU-bound operations just because it is a statically compiled language. So we did some good performance wins just by writing some of the logic in Go versus the JavaScript implementation.

[00:35:46] AS: Part of that also because, at least for us, performance in Go was easier to reason about and it was easier to flag performance problems in code review. There are definitely a lot of really, really expert JavaScript programmers at Uber, and they have produced some spectacularly efficient JavaScript. But it tends to be finicky and a little bit error-prone even for them. So our feeling was that as a project that was going to live on for the long-term at Uber and going to need to survive handoff between multiple engineers as people come on and off this particular project. Go was a little bit less finicky performance-wise.

[00:36:32] JM: Yeah. As you mentioned, you have the V8 garbage collector, which I don't think is going to be as performant. Go is garbage collector, right?

[00:36:42] PV: Yes. Go is garbage collected.

[00:36:44] JM: Okay. But Go has a little bit more – I mean, Go as a language was completely architected to be very performant from day one and to run as a backend system. So imagine the garbage collector is a little bit more tuned for this kind of thing. But in addition, you have the concurrency model. I believe the concurrency model of Node, Node.js, the JavaScript framework that you were using, is more of this event loop, where it's essentially just roundrobining through the different threads rather than actually having threads running concurrently. So you have like better performance out of the box from Go.

But that said, when you did deploy this Go proxy layer, eventually you wanted better performance. You want to be able to profile your Go service and have better performance. That's where some of the talks that I've seen from – I can't remember which of you gave it, but a talk that I watched about performance in Go and improving performance, which was really insightful. Maybe we could just go through the process of profiling and improving a Go program from your point of view.

[00:37:58] PV: Yeah, sure. I think you might be referring to a couple of the talks I've done on profiling Go. So, of course, those talks we'll get into a little bit more detail. But in general, what we really liked about Go is it's built in. Profiling is completely built into the compiler, the frameworks. So you can just hit a running process and take a live profile and find out what's using up your CPU. So it's so easy to get started.

But, of course, when you're building something extremely high-performant, like something like a proxy layer, you don't want to just build it and then profile it later on. So we took a performance first – I'd say like performance first mindset while building this. So every time we were adding any logic, we were writing benchmarks. We had a bunch of like – We know what slows down Go. Go tends to be slowed down by allocating a fair bit. So Go does give you enough control that you can write code in a way such that objects are allocated on the stack rather than on the heap. You can run the Go compiler with flags to tell you why something is on the heap. So when something is on the heap and why it's on the heap, use that to figure out whether there's a better way of riding it so it's allocated on the stack.

We wrote like integration test kind of benchmarks so that we knew end-to-end performance of proxying was some certain number of request a second or request a second per call, and we'd make sure we were watching that number closely as we were building this proxy. Because you want to know that you're not slowing down the performance overtime. You want to make sure you're maintaining that bar that you've set, and that means you need to monitor that from the get go.

So I think it's more of a mindset of caring about the performance from the beginning, then building something and then worrying about the performance, because there are a lot of design changes and API changes you need to do sometimes for performance reasons, and that gets harder to do once you've already built the system. So I think it's better to think about it from the beginning, and I think that's actually where something like Zap is a great example of a performance first kind of mindset. I'll let Achal talk about the API of Zap.

[00:40:20] AS: Actually, as we were building this proxy, of course, it's a piece of infrastructure software. So we're going to be running tens of thousands of instances of this thing across Uber's fleet, and we need to know if any one of those proxies is encountering problems. That usually means we use something like M3 to collect metrics, and we also need to emit logs, just regular logs that get written out to some text file and then slurped up into some log ingestion system, like Elk, or Loggly, or Papertrail or whatever.

As we started to add some metrics and some logs to this proxy, all of a sudden all of our performance benchmark started failing. We looked into it and it's basically because all of these logging libraries were allocating a tremendous number of small objects on the heap. So we looked at our needs and started designing a logging API with the sole purpose of remaining fast and keeping data off the heap. What came out of that is Zap. It's Uber's most popular open source Go code, at least as far as I know, measured by GitHub stars at least. I think the primary insight there is that, like many places, we log in JSON now and that it's possible to write JSON without constantly emitting or constantly allocating maps and pointers.

[00:41:58] PV: One other thing we did do while building out this proxy and other Go infrastructure, the profiling tools that were built in were great, but sometimes they're a little bit hard to digest the output. So we also built some tools to make it a little bit easier to understand what exactly the profiling data was telling us. So that's where Go Torch, which is a tool that takes the profiling data and visualize it as a flame graph, we built that to better understand this data. That's actually been so popular now that the Go team has built this into pprof. So now it can natively emit flame graphs, which are much easier to visualize and understand than the profiling output that was present earlier.

[00:42:47] JM: Flame graphs are useful because they give you kind of a trace of how different services are being called and how long those different services are spending in sub-calls, right?

[00:43:02] AS: I think you could use a flame graph for that. In this particular context, we're not talking about benchmarking calls between services. What we mean more is within a single Go program, if I'm looking at the stats coming out of this proxy and I notice that yesterday was using a 10th of a CPU, and today it's using half a CPU and I want to know why. What I would do is I would make a request or start running some traffic through the proxy, and I'd use this flame graphing tool, which will tell me how long the program is spending in each function call. If I compared that with the flame graph from the day before, maybe I would be able to notice that, for example, some logging function went from being super cheap yesterday to suddenly being really expensive, and then I can track down the changes that we've made in the logging library and figure out where we introduce this performance regression.

[00:44:04] JM: Understood. In that talk, there was one part where you spoke about having to look at assembly code to optimize it. When would a Go programmer need to look at assembly code for the purposes of optimization?

[00:44:20] PV: Yeah. So I love deep diving into everything that's running in my process. So there are time when I'll often even dig into the assembly. The Go profiling tools make it extremely easy to do that as well. You can ask it to print out a function and you see, "Oh! There's a line of code there that's using up more time than I would expect."

Of course, a line of code actually could be one, could be 10, could even be more than that in terms of instructions. Sometimes it's nice to look at the instructions to get a better understanding of why that line of code is slow. So that's when I'll tend to dig in. But at the same time, it isn't something I would recommend that most people do. It doesn't really give you a huge benefit to the amount of kind of complexity and cost of understanding what's going on.

So I always tell people, "If you care about nanoseconds, then maybe looking at the assembly is a good idea and you might find some wins. If we're talking about milliseconds, then no, it's definitely not worth it." So I I'll only ever dig in if it's something that's run millions of times in a second and it's actually worth optimizing, then I'd say, "Yeah, it's worth looking at the assembly and understanding what's happening."

[00:45:41] AS: I'll also say that despite reading a lot of assembly in writing this proxy, we never stooped to the level of writing Go assembly, which you can also do, but is rarely a good idea.

[00:45:55] JM: When you modify Go assembly, does that mean like you're modifying it for every single instruction or do you like make a custom instruction with like fewer assembly lines?

[00:46:05] AS: So the assembly we're talking about is actually like a platform independent assembly that's part of the Go compiler tool chain.

[00:46:12] JM: Oh, okay.

[00:46:12] AS: So what you can do if you chose is you can implement a function in assembly instead of implementing it in Go. So it's not like we're patching like the X86 assembly that the Go compiler producer. It's basically a choice you make when you write the code. You can write it in Go, or you can write the function in assembly.

[00:46:36] JM: Is that like a bytecode language? I don't know too much about the compilation path of Go.

[00:46:42] AS: It's kind of similar to a bytecode language, but it is a little bit lower level than that. So it does look a lot closer to something like X86 or ARM assembly, but it tries to be platform independent in terms of you're not dealing with the physical registers on the hardware, but they have some virtual register idea, virtual program counter and so on. They have a couple of instructions, which might actually map to one or two real instructions on a specific platform.

Basically, there are some cases where you need to write Go assembly, especially when you're writing the Go compiler and the Go platform itself, and they wanted to avoid having too much platform-specific assembly. So that's where it ties in, especially when it's like switching stacks, when it's switching what function is running on a single physical thread when it's switching Go routines. There are some cases where you need to dig a little bit lower level than what the Go compiler, like what the Go language will let you do. That's where the Go assembly comes in handy. So it's still platform-agnostic, but you get to get a little bit lower level than what's –

[00:47:48] JM: So it sounds like the Go assembly language, is it like the last step before Go code is turned into a binary?

[00:48:00] AS: I can give you a high-level overview of how the build tool chain typically works in Go. I think for this particular explanation, it's easiest to use like C++ as a reference point. So I know not everyone is familiar with C or C++, but typically what you do to build a Go program is you type Go build, and that's a little bit like a standardize built-in make file that understands how Go projects are structured and how to tie together all the pieces of your project into a single platform specific kind of native binary.

So what Go build does is it uses a bunch of conventions that are built into the Go programming language to run like a C or C++ style compiler and linker under the hood. What those things dump out is a native binary. What that means is that it's specific to the platform and the process or architecture that it's going to run on, and that you can just copy that thing to a machine and run it. This is in contrast to a language like Python, or Node, where typically there's a Python interpreter or like nodes V8 runtime, and you have to copy the runtime to the target machine and all of your source code and then interpret the source code where it's going to run.

Does that make sense?

[00:49:38] JM: Yeah, it does.

[00:49:40] AS: So the assembly is one of the inputs into Go build. There's a convention of how you lay the files at and you can have a mix of Go files and a mix of assembly files, and the standard Go compiler tool chain knows how to knit those things together in the right way.

[SPONSOR MESSAGE]

[00:50:07] 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 that 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. So if you want to get some additional help for your engineering projects, go to softwareengineeringdaily.com/g2i.

[INTERVIEW CONTINUED]

[00:51:59] JM: Question about performance. There are some kinds of performance issues that only arise at the statistical tails. So things that can happen somewhat infrequently, but when they happen, they can be quite problematic. Could you describe the phenomenon of tail latency and how you've experienced it within Uber?

[00:52:20] PV: Yeah, we tend to monitor almost exclusively for tail latency for some of our infrastructure components like our proxy, because like you said, I mean, the P50 is not as relevant as the P99, especially when you're doing a huge number of request. The P99 will end up affecting like 60% of request just because of how many hops through the proxy a single user request can make.

[00:52:45] JM: Could you define those terms, P50 and P99? Some listeners may not –

[00:52:48] PV: Yeah. Of course. Yup! So while measuring our latencies, we tend to focus on not the median, which is the P50 or the 50th percentile, but rather than P99 or even higher, which is what the 99 percentile of the slowest request – Actually, I'll let Achal explain this. This is his [inaudible 00:53:08].

[00:53:09] AS: No. I was just going to say, so the P50 or the median, if you take all of the latencies and you sort them, it's the one in the middle. So 50% of requests are faster than that and 50% are slower. Then if people talk about a P99 or a P99.9, that's just picking something that's closer to the max of all the latencies. So the P99 means that 99% of things are slower and 1% are faster.

The P99.9 is the same thing, but even closer to the max. Then if you're being super, super picky, you measure the max. So for our infrastructure things, like we tend to focus on throughput per CPU core. So like how many request can I proxy in one second given one CPU core? Then some number close to the max for latency.

So for our service proxy, we might say I must be able to proxy 10,000 requests per second per core, and the overhead that the proxy introduces should be less than one millisecond at the P99 or the P99.9.

[00:54:24] JM: Understood. So what implications does that have for an engineer that's trying to discover and iron out tail latencies?

[00:54:34] AS: I think for us in Go in particular, that means the usual stuff if you're doing performance sensitive engineering, so it's being a little bit picky about data structures and algorithms. But the part of it that ended up getting really nitpicky is basically being extremely careful about when you let data escape to the heap and when you keep it on the stack. So that's where we ended up using Go Torch all the time. That's when we ended up looking at assembly all the time, and that's where we ended up writing libraries like Zap, whose purpose is basically to keep values on the stack and avoid heap allocations, which tend to be pretty expensive.

[00:55:15] JM: As we begin to wrap up, I think the topic I'd like to get both of your reflections on is this subject of writing performant platform engineering tools. I think this is a growing area in the industry. I think more and more companies are adapting some kind of platform engineering. So I'd love to just close by getting each of your individual pieces of wisdom on writing performant engineering tools.

[00:55:46] AS: I think that's a great question, and as infrastructure people, I know Rob who you talked to before would absolutely agree with this. Performance is a feature of infrastructure and it's an important feature. For us, at Uber, because of the language choices that we've made throughout the rest of our stack, we tend to do a lot of our more performance-sensitive infrastructury work in Go. That's because the build and deployment and management tools for Go are mature here.

Any gains we make in core libraries, the benefits then bubble up to all of the business layer services that are also written in Go. I think there's a ton of really interesting work going on in other languages. So C++ is still evolving rapidly. Rust is coming up really quickly. They're somewhat more esoteric languages, like Pony, or Crystal that I think are really interesting too. I try to restrain my interest in those things to my off hours and not inflict my crystal interest on Prashant during work hours.

[00:56:52] PV: I think Uber in general, and I'm sure any large high-scale kind of company will end up realizing how important performance is at all layers of the business. So to really provide a good user experience, you have to be thinking about performance. You have to think about what is the latency of this user interaction. That ends up pushing requirements all the way throughout the stack to make sure that you can respond to these requests quickly.

But a microservice architecture, you may have anywhere between 10 and 50 calls that happen for a single user request. For each of those 10 calls, they all need to do some common things. They might be serialization, logging, RPCs, security. So all of these need to happen very, very quickly to really respond to that user request in a reasonable amount of time. So you end up pushing up pretty high performance requirements for all of your services, but especially the common libraries that are used throughout all of these services. That's where I think the platforms do have to be really efficient. You don't want to spend all of your time serializing JSON when you have real business logic to do as well, and you don't want to have delay the user request by half a second, because you spent too much time serializing.

One thing we're focusing on here at Uber is distributed tracing. So Uber has open sourced Jaeger, which is our distributed tracing platform, but what Jaeger does solve is telling us where time is being spent in this microservice architecture. So that way you can at least – You know where to focus on and what services to look at to understand why performance is slow or what's happening.

I think Jaeger and the service mesh, which provides latency numbers into every single hub, these kind of platforms, they are all there to provide the introspection to what's happening. They have to give you the metrics to figure out, "Are my requests slow? If they are slow, where is the time being spent? Which service?" Then once you get to a service, "Why is this service slow?" There's a whole range of different problems there, and I think there are different ways of solving each of those problems, but you do need to look at all of these different areas to really build a performant user experience.

[00:59:26] AS: I just wanted to talk on one thing that I think we would be remised to not mention, which is that in the particular parts of infrastructure that Prashant and I work on, we solve all of these problems in Go. But we have a whole set of compatriots who build a bunch of mission-critical infrastructure in Java and they have all the same problems and they have tackled basically the same set of issues with very similar approaches. So I don't want to leave you with the impression that literally everything at Uber is written in Go.

[00:59:57] JM: All right. Well, thanks for the clarification. Prashant and Achal, guys, thanks for coming on Software Engineering Daily. It's been really fun talking to you.

[01:00:04] AS: Thanks, Jeff. We appreciate you having us on.

[01:00:06] PV: Thanks, Jeff. It was great talking to you.

[END OF INTERVIEW]

[01:00:11] JM: This podcast is brought to you by wix.com. Build your website quickly with Wix. Wix code unites design features with advanced code capabilities, so you can build data-driven websites and professional web apps very quickly. You can store and manage unlimited data, you can create hundreds of dynamic pages, you can add repeating layouts, make custom forms, call external APIs and take full control of your sites functionality using Wix Code APIs and your own JavaScript. You don't need HTML or CSS.

With Wix codes, built-in database and IDE, you've got one click deployment that instantly updates all the content on your site and everything is SEO friendly. What about security and hosting and maintenance? Wix has you covered, so you can spend more time focusing on yourself and your clients.

If you're not a developer, it's not a problem. There's plenty that you can do without writing a lot of code, although of course if you are a developer, then you can do much more. You can explore all the resources on the Wix Code's site to learn more about web development wherever you are in your developer career. You can discover video tutorials, articles, code snippets, API references and a lively forum where you can get advanced tips from Wix Code experts.

Check it out for yourself at wicks.com/sed. That's wix.com/sed. You can get 10% off your premium plan while developing a website quickly for the web. To get that 10% off the premium plan and support Software Engineering Daily, go to wix.com/sed and see what you can do with Wix Code today.

[END]