EPISODE 1376

[INTRODUCTION]

[00:00:00] KP: The Internet is a layer cake of technologies and protocols. At a fundamental level, the internet runs on the TCP/IP protocol. It's a packet-based system. When your browser requests a file from a web server or an MP3 file from a podcast feed, that server chomps up the file into tiny pieces known as packets and puts them on the network labeled with your machine's address as the destination. That system obviously works incredibly well for receiving a file from a web server. And if some of the packets arrive out of order, that's not a problem. If one is lost, it can be sent again.

There are no guarantees in a packet-based system. No direct connection. If a flood of new packets show up, the system can slow down and you may experience a lag in response time. This can be annoying when visiting a blog that is slow to load, but it's not really a ruined experience.

Streaming video on the other hand does not degrade elegantly in this situation. No consumer wants to have the experience interrupted with a spinning wheel. Traffic can be spiky and unpredictable especially around live events. All this is to say, the stakes are high for building a scalable, efficient streaming video solution.

Amit Mishra is a member of the team at FOX, which is responsible for building platform to live stream content across the FOX properties. In this episode, we discuss some of the technical milestones in delivering this platform and why Golang was the right choice.

[INTERVIEW]

[00:01:27] KP: Amit, welcome to Software Engineering Daily.

[00:01:30] AM: Thank you.

[00:01:30] KP: So can you tell me a little bit about where your career journey in software began?

[00:01:35] AM: Yeah, sure. I have been in software industry almost for 17 plus years and most of my career has been into media and entertainment industry. I have been working on mostly backend space for various clients and playback kind of scenarios.

[00:01:53] KP: What's some of the technology that you've worked with throughout your career?

[00:01:55] AM: Yeah. Some of the technologies that I have been working on, I started my career as a Java developer. From there, I continued on working on those things like almost a decade. And then slowly I moved into PHP space a little bit because of requirements from the product point of view. Then I again went back to Java, because that was my kind of like original love. And then I moved to Golang space. And since then, I've never went back to any other technology space.

[00:02:29] KP: So it sounds like you had some success migrating from Java, Node.js, to Golang. That's a path other technology groups might be thinking of following. Is it always the case that you think someone working in Java and/or Node.js should probably migrate to Golang?

[00:02:43] AM: To be honest, no. The choices should depend on what exactly you need. So it's around 2016, I think. Actually, 2015, I would say. We were looking to rewrite couple of things in our platform. And we used to be a big Java shop. And at that time we had recently migrated everything to PHP-based system. But then we started having you know lots of performance issues. And then quickly we also migrated so many things to Node.js system, and then we had the similar kind of issues.

Then one of our colleague had a very good experience with Golang, and then he kind of suggested that to POC. And then we thought, "Okay, let's try this out also," right? And this is where we kind of like started writing like a small POC, which was greatly received by all the devs, especially most of them coming back from Java and Node.js background, but they all loved the simplicity of Golang. And me, also as a software engineer, loved that whole idea of

simplicity. So we kind of stick around it and then we kind of like move forward with that idea at the organizational level. So, yeah, my answer would be like choose based on what you need. And for all use cases, I think Golang was the choice most of the time.

[00:03:57] KP: Well, to hear that you have one developer working in maybe Java, showing appreciation for another language is promising. A lot of people are dogmatic in that regard. But at face value, it seems like maybe they just appreciate the syntax, or the data structures, or the design of the code, something like that, the elegance of how to write it. But it seems to me there's something more you're looking for in a migration to Golang. What are some of the features beyond the ones that a developer would appreciate that will motivate the move?

[00:04:25] AM: Yeah. So as a developer, when I say simplicity, like simplicity not only for the syntax, also the speed of developing your software itself. So one of the thing we found with Golang was like our speed to deliver the software increased like 100 times. Whenever we used to write a Node.js-based services or the Java-based services, the development time for those used to be like, let's say, for a simple service, used to be like more than one sprint or two weeks. But with Golang, that simplified a lot, and that whole thing went down to two to three days. And that was exact same thing, exact same business logic, but the kind of speed Golang provided in the development process was incredible.

And not only that, like the performance itself, the Golang itself was like so lightweight. Without doing lots of tuning at the language level, we were able to get the kind of performance we wanted to have by default, which was like great sell for us, because in this case, without having very lots of experience in a particular language, we were still able to build a performant software. So that was the great thing to do. And that is what like kind of started giving us the confidence in this direction.

[00:05:38] KP: That speed up is obviously hugely appealing. I'm not sure if I have the core insight yet. What really is the root cause of such an increase in delivery time?

[00:05:47] AM: Yeah. So this is not against any language, just a disclaimer, right? And since I have been a Java developer almost more than 12 years. So I really appreciate Java as a technology. And I still code on Java. But one of the problem with Java or even the Node.js is like

it these kind of things comes with lots of baggage surrounding their ecosystem. And because of that, onboarding process for a particular engineer becomes very hard. Similarly, whenever you are writing a program itself, compiling itself is very, very slow. And this is where like you get stuck into that loop where you are debugging particular issue. But because of the language framework, you are stuck in those like couple of hours, right? So these small things count. Where in Golang, these things are very simplified, right? Like your syntax are so simple that you don't need to rely on end-to-end deployment in order to find any things, right?

Inherently, whenever you are writing any program in Golang and kind of like building it and running it, that process itself takes couple of seconds compared to Java or Node.js, where it takes like couple of minutes, right? So those small, small things counts a lot during a development process itself. So that's one of the thing.

Another thing was like the performance point of view, like I wouldn't bring up like we ran like little like three simple POC. We did a comparative study with Java. We wrote a similar service in Java. We wrote a similar service in Node.js. And we wrote the similar services in Golang. That was one of our account service. And when we run the load test, we found like Golang link service was like much more faster than Java. And this is coming from like the devs who did not have any experience on Golang. It was just simple modification, simple translation from one language to another language without doing – And all the three platforms were running on default settings. No tuning there. So that's another thing which came into that. And then we just rolled with that in production and it just saved us so much cost and energy in that direction.

[00:07:48] KP: Let's just go ahead and label it a rumor, because I've only heard this from one person. But they had done some work and Go and had some very early success, and their complaint was that it wasn't a rich ecosystem of libraries. So if they needed some connection to this or that, they might have to write it themselves. What was your experience like?

[00:08:08] AM: So that was definitely an issue for us when we started, honestly. But it was not a blocker, because kind of things we wanted to have, those were already part of the ecosystem. One of the thing we wanted to do using Golang was we wanted to move towards a microservices kind of architecture. In Node.js, we already had those kind of things by the way.

All our services, microservices were written on Node.js and in Java. Those were written through the Spring Boot.

So when you come from Java Spring Boot kind of ecosystem and suddenly write the services in Golang, then you are definitely going to miss those features. Like there are like production-ready endpoints in Spring Boot. You don't need to worry about for health checks and all those things. Those are already part of the ecosystem. But in Golang, you got to write all those things on your own. But I think those things are so simple. It's kind of accepted, I would say, drawback in a sense, because you get out of it so much that all those kind of things you don't miss end of the day. So that's how our experience was.

Like we also missed couple of things in Golang. But then we slowly moved and then we opted in for like various libraries. And honestly, Go community is so strong and they're so helpful. I got so much help from the community whenever we had an issue. So that kind of like simplified lots of things for us. I would give an example of circuit breaker. Circuit breaker already had a library, which we just opted in and then it just ran for us. It worked for us. There are lots of other simple examples which we got help from the Golang as a community.

[00:09:40] KP: Well, in comparison to a framework like Spring and the whole Spring ecosystem, there's a big maturity there. So you get a lot of things. It occurs to maybe that sort of is a monolith framework and maybe all those advantages don't pay equity in a microservice world. Would you agree with that?

[00:09:56] AM: Yeah, I would agree with you. Spring itself is a like very old framework and then a Spring boot kind of like more of the microservice towards your framework, right? So it has lots of features. So as a dev, you get lots of things out of it. But think about it like it comes with lots of its own baggage, because you have so many packages, so many jars as part of this whole ecosystem, which takes time to compile and make it bill ready, or production-ready service. But in case of Golang, all those things are not available, but writing those things are so simple that you don't miss those things. And the thing is that you can write those things in the way you like it. You don't have to rely on mercy of a particular framework. You have to stick with those kind of things even though you don't want those. So that's one of the thing, like Go kind of provided us the opportunity to choose and go based on what we needed.

And that was one of the reason. Like when we opted in for the Golang-based software, one of the reason, like most of the dev started asking us like, "Hey, what are the framework we are going to use? Java has a Spring boot, right? They have already been built for writing all these service layers. In Golang, what kind of framework we are going to use? And we consciously chose to write our own framework internally because we kind of like analyzed or researched lots of frameworks available using Golang to write the services. But those had so many different features, which we never needed. And those also had the similar kind of issues with Java might introduce into a system. So we kind of like rewrote everything, which we needed at that point of time, and then we continued it. Like, okay, at that point of time, we just needed to write a simple Rest endpoint. We did not want it to have any kind of other functionality as part of that whole thing. So we just wrote simple handler for that supported by some kind of data store, right? I don't need to think about supporting 10 different data store as part of my service, right? It should have just one support for one data store. Let's run with that, right? Why should it include 10 different data stores and libraries unnecessarily?

Similarly, I would say, circuit breaker, as I said, right? In a couple of services, we did not need any circuit breaker functionality. Then why the service would have those kind of dependency at compile time, right? So we had the opportunity to basically not to go with those and just rely on that default Go language feature to write those kind of things.

[00:12:11] KP: Well, in my experience any group that's doing a serious Java project at scale is eventually going to have some challenge related to the JVM doing garbage collection. It's going to do that full pause, scan, all that, and require some fine tuning. Does that get better, worse or the same in a migration to Go?

[00:12:29] AM: It becomes much better in case of Go, honestly, because I remember struggling with Java to come up with those settings all the time and find that sweet spot. But once you find that sweet spot, you will not have any issue. But reaching to till that state is a big thing from the product point of view. But in Golang, we never had those kind of things. Like we just ran with whatever came out of box. Slowly we learned more things. If we were able to write a good Go program, it was already memory-optimized. We did not have to worry about any of the garbage collection kind of issues in this case.

Interestingly, recently, we had an issue. But then later it was found that it was our code. It was not a language problem. It was our code, which had these excessive memory utilization problems. And Golang provided us the way to optimize those. And then we went back to our normal situation.

[00:13:18] KP: Well, these all seem like good indicators that Go is a useful tool for your team to be leveraging. But you're not going to just do a full rewrite of the whole system. How do you look for the right opportunities to introduce a rewrite?

[00:13:30] AM: That's a very interesting question, because this is what we have been doing in the last year so. So, yeah, I mean, most of our platform was either on Java or the Node.js. like I would say like since we were migrating from Java to Node.js, so most of our platform was in Node.js, right? But then when we started having issue with Node.js, then we made a conscious choice to migrate to Golang. But it was not an easy task for us. So the kind of approach we started following was like implementing a simple Go proxy on top of each of these services. And honestly, like just writing the proxy service on top of existing Node.js legacy service gave us the performance boost. And that, we don't know how did it happen. Honestly, like we were also amazed like why this is happening, right? How come Go is able to optimize these things, which is just take same request and passing the response from the underlying services. But the way Go was able to manage their resources, that was incredible. So that's what we did. Like we did it in couple of stages to be honest, right? Like let's say we had a service which we wanted to migrate, we just started writing like simple proxy on top of it. Once we had the Golang-based proxy in the background, we started basically swiping small, small component with Golang and started doing the comparative testing. And once we got that confidence based on our comparative testing, we kind of like went through and removed the whole proxy layer, because now our whole service was already on Golang and we were able to publish it out. But just swiping the whole service in one shot from legacy to a new microservice is not doable, or is not practical. Just because the kind of space I work on, like on live streaming space, it's not possible. We have the events every week, right? So kind of risk is very high if we do those kind of experiment at that moment.

[00:15:25] KP: What are some of the challenges in doing live streaming?

[00:15:28] AM: Yeah. So one of the biggest challenges in live streaming is like it's live. You do not get the second chance. Once you have that moment passed, then that's it. You are done. You lost – Like if something goes wrong during live streaming, then you already lost your users. And not only you lost those users for that moment, it might impact your image also, lost that user forever. So that's a very kind of like unique challenge to handle. And other challenges are like the business challenges, where you got to deal with the number of concurrent users dynamically, because you cannot predict that how many users are going to join at that moment when the game or a particular event is happening on, right? You can predict little bit, but honestly it's never correct, right? There is always slight chances where on a particular game there is moment happen when everyone got interested and then suddenly you have the flood of traffic, which you cannot control. So managing those kind of expectation at that moment is very difficult. And this is where like most of our challenges comes into the picture.

[00:16:36] KP: So in that regard, I mean, you could, I guess, fall back on what the cloud offers natively. There are some you know scalability solutions built into all the major services. But I'm not sure they're built for every single use case. Where do you guys fall with that? Do you lean into a cloud provider or do you have to do something custom?

[00:16:53] AM: We use AWS as our cloud provider here. And we depend on them for lots of things. But for a couple of things we cannot rely on. Like because whenever there is any live streaming going on and if there is a flood of traffic coming in, then depending on your system might not auto scale fast enough in order to support it. For those kind of things, we came up with our own ideas like implementing our own caching kind of system and then implementing some kind of like fail-forward scenarios at our end. Those kind of things, like we had to come up with our own things. So yes, we depend on lots of things on AWS. But at the same time, we had to come up with our own ways to handle these kind of things.

[00:17:33] KP: What's unique about your situation that maybe couldn't be covered in the standard cases that Amazon had prepared for?

[00:17:40] AM: Yeah. I mean, one of the scenario I would say, like let's say app crash kind of scenario. Like let's say you have some kind of live event going on. Let's say Super Bowl, right?

You have all the users logged in and suddenly your app crashed. Of course the app should not crash, but things happens. And when it happens, like all these users are going to retry the stream. And as a user, we are going to click or retry multiple times. And I don't even know like as a user how many times I'm going to click until my stream is going to load, because it's such an important game. I want to watch it at any cost before I can try another provider.

So in this case, you can think about like amount of traffic a one user might be sending, unexpected traffic. So now when you multiply this traffic between millions of concurrent users, then this becomes totally unmanageable traffic, right? For this kind of scenario, we had to come up with an innovative way of like implementing various patterns like single flight kind of pattern, or throttling at some level of the service, level, right? Or if things are going out of control, then how do we ensure that those are not client-initiated retried? All those retried are controlled by us, not by clients, right? So those kind of things we kind of like came up with, where AWS couldn't help us, but we had to implement on our own.

[00:18:58] KP: So it makes total sense to me that you cannot predict these spikes in traffic. They're random noise in many ways, unexpected things. But is it so hard that it's not worth trying? Is there any ML or anything going on in the background trying to give you a prediction that's useful?

[00:19:14] AM: Yeah. Like we have those analytics always running in, right? And we always have those numbers. And based on those numbers, we are ready with event. Like we always scale in advance so that expected amount of the traffic can be handled. But these things can happen internally also, right? Like where one of your service for some reason is not performing the way you are expecting due to some unexpected reason. And that can basically create pressure on other microservices. So those can be your internal reason, but it's still unmanageable. So even though we had auto scaled our system enough in advance, but still you have those kind of things happen. But since those are internal issue, it doesn't mean that it should ruin your client experience or user's experience, right? Because we got to have our stream or the things playing without any problem. So those are couple of things like we have to keep in mind whenever we are like implementing these things.

[00:20:09] KP: It seems like there's more than one, for lack of a better phrase, plan of attack you can take here. There's maybe caching you could do. There's having a pool of warm servers to pick up as you need them without cold start. Maybe there's other techniques I'm not aware of. Is there one clear winning strategy you're chasing? Or are all these things on the agenda?

[00:20:31] AM: Most of the time, we rely on caching. But at the same time it's a combination of different, different strategy based on the traffic or dynamic nature of the event we are expecting. So let's say there is an event. I will give an example of Super Bowl itself, right? During Super Bowl, our goal is to have this uninterrupted experience to users at any cost. During Super Bowl kind of scenario, like sometime your event is so important that other features kind of like might be little bit less important at that moment because your streaming with Super Bowl is more important.

For those kind of scenario, we have our own defcon con mode kind of implementation where like we run on normal mode all the time, right? But let's say if there's an infrastructure failure or anything happen, at that moment in time, how do we ensure that we can still serve the Super Bowl or the event itself? For that, we have like something called like defcon zero, where we can just feature flag it and it just basically redirects our traffic to a different region itself and we can run from there. And even if that reason also, let's say different reason also goes down, then we have our internal failure scenario implemented in this case where we can also rely on some different kind of infrastructure where we can start serving the stream from. So that's one of the thing for like most important event we do. Other than that, as I said, like we rely on mostly like caching and I would say fail for what kind of scenarios we have implemented.

[00:22:03] KP: It makes total sense to me that you would have strong operational procedures like what you described. This happens, you have pre-planned the execution of what to do, because these are such important events that you shouldn't be thinking on your feet, right? You need to have a pre-calculated solution almost. With that in mind, maybe I'm curious if you see it a different way, but I guess my question was going to be do you have to run drills or war games around this kind of thing? Because you don't want to learn during an actual emergency.

[00:22:30] AM: Oh, yeah, definitely. Yeah. Before any big event, we have months of drills running. In fact, like for 2023 Super Bowl, we have already started preparing for. So each and

every big event before that we basically prepare our mock events where we run these trails. And we, on purpose, bring down certain part of our infrastructure and see how we are behaving. can we still serve the stream or not? So there are lots of things goes in background in order to deliver the final version of the event itself. So yes, I agree. I totally agree with you in this case. Like, yes, we do have like lots of drills happens over the period of time. And those things, the frequency of these things increases. We are more closer to the events, to be honest. But the way things are organized, we have very little chance to react also. So this is where like we have to be ready in very much advance.

[00:23:21] KP: Could we talk a little bit about the playback platform? I don't know that listeners will be aware of it. Perhaps you could start with what the solution does.

[00:23:29] AM: Yeah. Basically, our playback platform, like recently we wrote that whole platform into Golang. And this playback platform is nothing but simply like how do we ensure that we deliver the end-to-end playback experience to the users? So as part of this platform, you basically deliver the streaming playback URL. Of course, you have to deliver some kind of analytics from the clients. And also the most important piece is like analytic metadata, because whenever there is any live streaming going on, there are ad markers, right? Based on those ad markers you want to deliver the right ad to the user. And during that point of time, since those markers are so dynamic in nature, going back to the back end in a dynamic manner at the same time and getting the result and serving the right ad becomes very crucial especially from the revenue point of view. So that's one of the critical thing it does.

Apart from that, playback platform also handles like various challenges. The challenges like game extension. Because any live event, all the programs have some schedule. So that's how we also do that here in our case. Like we have everything scheduled in advance. But in case of live event, if the game goes over time, your schedule kind of disturbs. In this case, you have to basically, manually, from the content operator point of view, go back and reschedule everything and then just to extend for couple of minutes. And also you don't know how long that game will be extended. Like you can predict, "Okay, it will be 15 minutes, or 20 minutes, or 30 minutes." But you cannot predict exact moment, "Okay, this is when this is going to end. And this is when my next program boundary should restart," right?

So coming up with that kind of prediction and then extending those kind of events right at the moment is riskier also from the digital streaming point of view because, in your backend, it resets lots of things. You cannot rely on the caching for this kind of scenarios because it's going to create thundering hard kind of problems, right? So this is one of the main feature like our playback platform needs to support in general. Other than that, like standard expectation from any platform where it should be scalable to millions of concurrent users and it should be resilient enough and fault tolerant enough so that we can support things the way they should be.

[00:25:43] KP: Well, any stream processing system at some point has to manage the debate, do we want to run with batch mode calculations, or be totally event-driven, or some sort of hybrid? Where do you fall on system design?

[00:25:57] AM: It's kind of a hybrid I would say. So what we do, like most of our contents are like in live streaming, as I said, like dynamic. But we can still calculate couple of things in advance. When I say advance, not like in advance couple of hours before, but at least couple of minutes before. And this is where we use – Like I would give an example of latency, right? Whenever any broadcast event is going on. So there will be some latency from broadcast to digital streaming, right? And that is the thing we kind of like use in our benefit. There is latency. So like before the streaming is going to propagate into different clients, before that itself, we get all those requests into our system and that kind of like helps us to cache the data. So we use that drawback in our benefit in order to kind of like serve the dynamic traffic at scale. Not only that, like we also have done some analysis like that, okay, these kind of metadata are going to be available in advance. So we are kind of ready with those. And these kind of like metadata are not going to be available. So how do we basically create those buckets for dynamic data so that even in case of failure it impacts us very less from the future point of view without impacting the streaming?

[00:27:09] KP: It seems to me you have a tremendous potential to generate volumes of data even greater than what you're transmitting if you really wanted to have full grain tracking of events and things, you can't possibly store everything. How do you approach what to keep, what to throw away, what to aggregate and that sort of thing?

[00:27:26] AM: From data point of view, we try to collect as much as data we could. And honestly, like for this particular playback framework, we do not store any data at our end. Like it's all read only from our end. So it's all from like what clients are requesting from. But being ready for that kind of data, that response is kind of tricky, because each request is basically a unique client request. Because as a user, when you come to a stream, you are a unique user. You will have your own location or DMA and own IP addresses. And based on that, what kind of stream we are going to serve is a different thing. And these things becomes more complicated when the subscription scenario comes in the picture, because as a user, you may or may not have the subscription or the entitlements to play a particular event. So these things becomes more dynamic in this nature.

But to answer your question, from the data collection point of view, we send lots of data, we collect lots of data into our data analytics portion. But from the playback API point of view, we kind of ignore those requests, data collection request. So the traffic you're handling, correct me if I'm wrong, I think it's really the video data or maybe compressed video day that you're shipping to viewers. Is that right?

[00:28:35] AM: Yes, right. Yeah.

[00:28:37] KP: So there's some – I think watchers will be forgiving within a few seconds, right? I wouldn't even know if my neighbor was getting the super bowl three seconds – Maybe I'd know that because somebody cheered or something. But if I was a minute lagged, I'd probably be upset. You have a certain amount of wiggle room there, I guess, before users become upset. Is that something you optimize towards?

[00:28:58] AM: Yeah, definitely. That's one of the like our main focus, is like how do we reduce that latency between clients and even from the broadcasting point of view. Recently, we have been working on the latest features where this latency is going to decrease a lot. Like I think current latency is like around 30 seconds, 30 to 40 seconds. But once we release this new optimization, we are expecting our latency to be decreased to maybe 10 to 15 seconds. That is a big improvement industry-wise. And we are all kind of like very excited about it. But definitely, you're absolutely right. Even in testing or while supporting these events, whenever we are playing these events on different clients, we can definitely see that kind of lag where one of the

SRE team, they are watching the event that point of time and they suddenly cheer and we are like another room, "What happened there? We don't know." And after a few seconds we know, "Oh, wow! Something happened in the game and we just came to know about it," right? So those things, there are always scenarios where these kind of things happens and our goal is to reduce that kind of latency as much as we can.

[00:30:05] KP: So I definitely see your motivation to do that. But it seems like some of it's out of your control. Like you send the packets off from your servers. Eventually they go on to the backbone and find their way onto my home Internet provider to get to me. You don't own every hop. How can you account for that?

[00:30:22] AM: We cannot control those parts. Like there are a couple of things which we do at the player level itself, where we can help with the bandwidth issues or we basically deliver the right package size or the bit rate based on the Internet speed. But other than that, we have not done that much on that space honestly. Most of the optimization at the player level where we can opt in for.

[00:30:45] KP: What are some of the open challenges you guys are working on now?

[00:30:48] AM: One of the biggest challenges, as I said, we are working towards like preparing for the next Super Bowl, which is like prepping for at least five times of the traffic compared to this year. And since we are rewriting all those things into Golang, so we have learned on lots of things on the way. A couple of things we have learned that, okay, architecturally, we have the challenges. And how do we address on those things? One of the thing, as I explained to you previously, was like still serve millions of concurrent users. How do we aggregate or serve the API responses, which are user agnostic or the platform agnostic? Because, currently, everything is so unique. Caching those things becomes much harder. And because of that, your scaling capabilities are limited. So in order to handle those, we are looking forward for ways to implement our APIs in such a manner which are client entitlement agnostic. That's a pretty interesting challenge.

Once we can solve the responses, which are client agnostic and user agnostic, in that case, serving a standard live streams becomes much simpler, because after that we will have the

capabilities to cache couple of things at the CDN level itself. And after that, your scaling capabilities increases infinitely.

[00:31:59] KP: So in terms of what you're working on now and some challenges you might be facing in the future, what are you and your colleagues thinking about?

[00:32:05] AM: One of the challenges we have been working on was like a strategy for zero downtime while service migration. As I explained, one of the pattern we implemented from migrating from Node.js to Golang-based services. But even in that pattern we had the challenges. So we are still kind of figuring out, "Okay, how do we still ensure that we can migrate any of our legacy system without downtime?" Another strategy we have been working on basically are r multi-region strategy. I'm not sure if I told you that currently we use Elasticsearch as most of our data store capabilities. And until today – Actually, until last month, I think, AWS did not have any capability to replicate the Elasticsearch data to different region or different cluster. And because of that, we had very limited options when it came to some traffic from different reasons because you don't have your data store replication enabled on fly. AWS has recently announced they have the cross-clustering replication enabled. So this is one of the thing we are going to implement as part of our multi-region strategy. So pretty much interesting topic on this thing and looking forward to implement it.

[00:33:17] KP: Well, Amit, thanks again for coming on Software Engineering Daily.

[00:33:19] AM: Thank you so much. It was fun to talk with you.

[END]