EPISODE 1407

[INTRODUCTION]

[00:00:00] KP: If you're working on a proof of concept which you hope will help you eventually raise funding, it's fine to take a few shortcuts. Use the tech stack you know the best. Don't fall in love with your code. And when you start to experience growing pains, hopefully, you'll have the time to thoughtfully and carefully identify the bottlenecks and limits of your tech stack applied to the specific industry problem you are solving.

Another great strategy is to simply copy the tech stack of a larger company with the confidence that what works for a bigger company will likely work for you. But if you're a company like Uber, there's no larger company to copy. Worst still, in comparison to most businesses, even a few minutes of downtime is pretty damaging for Uber. To successfully deliver a solution like theirs, one must identify bottlenecks and growing pains in advance, find solutions, and deliver the plan in a way that's invisible to customers.

In this episode, I speak with Uday Kiran Medissetty, Principal Engineer at Uber, about steps taken in their core state machine design.

[INTERVIEW]

[00:01:06] KP: Uday, welcome to Software Engineering Daily.

[00:01:10] UKM: Hey, Kyle, good morning. Thank you.

[00:01:13] KP: To kick things off, can you tell listeners about how you got your start in software?

[00:01:19] UKM: So I got started in software maybe like 15 years ago. I think software is one of the few things that can have a broad impact around the world, and it's scalable, right? Like when you build something, you can immediately see the impact of that in any part of the world. And there's only a few things in the world that has this kind of broad impact. And the reason why I'm at Uber is also the same thing, because it's one of those things where through software you

can fundamentally change one of the basic human needs with respect to transportation, with respect to getting access to transportation, access to the things that you need wherever you are. And I think that kind of magic can be delivered through software.

[00:02:06] KP: Absolutely. Well, it makes sense then that you'd be attracted to Uber. What in particular though made you want to go work at Uber?

[00:02:13] UKM: So when I was first trying to figure out where – I think, five years ago, when I'm trying to figure out where do I go next, I was trying to figure out what are some fundamental human needs where the opportunity size is so big? And I was also particularly interested in solving climate change. And transportation is one of the huge contributor to climate change. And what I felt was by accelerating the transition to shared mobility to a fleet of vehicles who can transport people by using things like Uber Pool where you can transport more number of people with fewer number of trips. And by transitioning this fleet quickly to electric. Then we can completely change how people commute around the world. And over time we also went into different kind of verticals like Eats and other things.

But when I joined, we didn't have Uber Eats. So all we had was Uber, the rides part of the business. And I think the total opportunity size of transforming this segment around the world was so massive. So that fascinated me a lot. And after I joined, yeah, it was a roller coaster ride since then.

[00:03:28] KP: Yeah, there's something about uber that's interesting and how deceiving the product is. Anyone with a smartphone can install it and very quickly set it up and very quickly have a product brought to their home or a driver come to get them to take them somewhere. And it all feels very seamless. But it has to be the case that Uber's facing unique scaling challenges behind the scenes. Broadly speaking, what's the technology stack look like?

[00:03:54] UKM: Yeah, absolutely. I think the point that you mentioned like for a regular consumer, it might seem very deceiving, right? You just click a button, a car is coming. And in fact, when I was I was joining Uber at that point, like one of my relatives said, "Oh, the app is working fine. Why do they need you? Why do they need to solve for?" But only once you get like the simplicity, the complexity is all about creating a simple interface and hiding all the complexity

of making things happen in the real world. And Uber is one of those tricky scenarios where we are trying to make things happen in real world with users who we don't have full control over, and we are trying to make things happen, right? And that's where the tricky part comes. And like the riders might not show up on time. There might be traffic. There might be rain. And all sorts of things we need to account for to deliver that seamless experience every single time.

And with respect to the code technologies that we live in under tech stack, when I started six years ago, primarily most of our backend was in Python and Node. As and over time we transitioned to Go and Java-based services. Most of our data stores is key value-based. We were using Cassandra. And now we have an in-house storage system called **[inaudible 00:05:16]**, which kind of provides similar guarantees for. So that's kind of our main language and storage choices that we have so far. And we can go into more details through the interview.

[00:05:28] KP: Absolutely. One of the key aspects of that technology stack is the core state machine. I know what a state machine is from broadly in software engineering. Can you describe what Uber's core state machine is?

[00:05:43] UKM: Yeah, absolutely. So Uber's core state machine and the platform that handles the orchestration for all of the ongoing orders, jobs, and active user sessions, we call it inside uber the fulfillment stack. And what that means at the high level, when a consumer clicks, get a ride, or get food, we capture that user's intent and then we fulfill it by matching it with the right set of providers. And for this end-to-end lifecycle to happen, we have a set of entities, whether it is orders, jobs, job offers, user sessions. Each and every entity is modeled as a state machine. And in a state machine in a regular sense, like we have a trigger. And then the state machine is in a particular state. It will react to that trigger and figure out which are the possible transitions. And in our case, we have – Let's say when a driver begins a trip, we need to make sure the entities corresponding to the job. And for that particular session, go to the right state appropriately.

And if you think about the basic example, like an UberX. In that case, we have a transportation job entity that can comprise of two waypoints. And a waypoint represents a location and a set of tasks that can be performed at that location. And in a transportation job entity, imagine there's a

pickup waypoint and a drop off waypoint. And imagine if you had multiple destinations. At that point, along with pick up and drop off, we might have certain via waypoints in between.

And on the similar side, for a provider or for a driver or delivery person who is on active session, they are doing more than one trip at the same time, right? And for them, they have to see all of the waypoints that they have to perform in a chronological order. And all of this orchestration is handled by the fulfillment stack. Making sure, "Okay, when you request an order, creating the right kind of jobs, right kind of waypoints. Assigning it to the right kind of providers. Making sure we generate the plan for that provider with the right kind of waypoints and order them in a chronological way. Adding the right set of tasks in those waypoints." If you have alcohol delivery, at that point, for that drop of waypoint, we might also need to add a signature collection task. We might need to add a proof of ID task. And those are the things that needs to be orchestrated in these state machines.

[00:08:13] KP: We've bumped into a similar problem. You've described it in an elegantly simple way. But there's so much to it. Even just to say, "Oh, there's some waypoints along the way." There must be mapping services that are keeping updated about detours. And just so many services to kind of fit together. Do you work with like a microservice architecture from that point of view? or is it some monolith? What's Uber's general way of looking at how you connect all these services together?

[00:08:39] UKM: Yeah. So Uber is completely into microservice architecture. And we have a ton of microservices across the company each handling a unique logical functionality. And for the fulfillment stack itself, we have the microservices that handle these core state machines. Previously we used to have just two microservices. One for handling all the jobs and one for handling all the supply state machines. And in the new architecture we have for every entity, we have a state machine. And we can deploy that in a modular fashion, whether it's in a single service or multiple services. And along with this, we have a bunch of uh services that consume the events from these state machines and expose geospatial indexing and expose some rules engine and expose some metrics, expose some search interface. So these are inside the fulfillment stack. Outside of this we interface with microservices, from fares, pricing, matching, maps, which handle their part of the whole puzzle. And ensure, we fit together all of these things to deliver the end-to-end experience.

[00:09:55] KP: So the state of all of those entities, I'm sure there are some interesting challenges about how you store and persist that that we'll get into. There's also the logic of it. Like a driver who already has two or three passengers can't possibly have more if the vehicle doesn't have seats, that kind of thing. Is that part of the state machine service? Or is the service a little bit lower level than that logic?

[00:10:19] UKM: So how we try to organize our functionality within Uber is we think of a layered architecture. The functionality – The business logic that is purely around transforming the particular presentation layer, like even for a rider app. We have API interfaces. We have mobile interfaces. We have web interfaces. And each of them have their own presentation surface, which tries to take the data from all the platforms. Transform it to that particular user flow. And then we go get into the product layer. In that layer, the functionality is how do we orchestrate across various platforms? For example, if I want to do a pickup trip. I need to orchestrate across the risk. Do some risk checks. I need to go to payment. Do some payment checks. Need to go to some other platform. Do some other checks. And then we create the trip required.

And once we get into the business platform layer, that's where we have the state machines. And we try to minimize that logic inside the state machines to the fundamental things that are needed for the shape machine to function. Like let's say if I have a constraint that a rider cannot be on two trips at the same time. This functionality, like that is essential part of that shape machine. The rider should not become active until – If they're only on a trip, we cannot accept one more trip. So that check should be in the place where you have the state machine logic. But a lot of other functionality that does cross platform orchestration, that does presentation orchestration, those are in layers above. So that way we try to make sure that you know we have a good reason behind any logic that goes into each layer.

[00:12:05] KP: When updating the state, especially in like an Uber Pool situation, where you have a driver with a device, and I presume you're still in communication with all of the riders' devices as well. And they're all probably on a shaky connection depending on where you live. They might report differences. Do you face any consensus style problems in building the state machine?

[00:12:27] UKM: Yeah. Ultimately, the state, especially – That's a really good point, right? Most of Uber's users are using mobile devices. And sometimes they might be in choppy networks where they might not have the up-to-date state. But ultimately, the source of truth is the backend. And if they might – A single participant in the system, they might have a stale view. But the moment they have a good – Like their network becomes active again, they will fetch the latest state from the server and then they show the right information. In fact, maybe a driver has started the trip. But a rider's phone is currently inactive. And the moment they get the network, they see that, oh, the trip is active again. Like the rider is on trip. So that's when the riders' device will get updated.

And at a particular instant, given client interface might be stale. But the moment the network becomes active, they fetch the latest state from the server. And the state machine and the storage backed by that is this ultimate source of truth of what is the state for a given user. And in case if one of the users who's on a stale state performs any action, that action will be invalid, right? Because maybe the server has already progressed to a different state and then that request will error out. And then by that time, their interface will be updated and they'll know, "Okay, this is why this request could not be performed."

[00:13:56] KP: So all of this service has to be provided in real time. You don't have any other choice really. How does that decision impact the way you build the system?

[00:14:04] UKM: Yeah. So that is a unique asset, because like we have millions of concurrent users who are interacting with our system, doing various kinds of operations, whether it's going online, going offline. The backend generating offers for particular trips. The drivers accepting offers, completing trips. The riders requesting and cancelling trips. All of these are real-time operations. We don't have the flexibility of taking this transaction and then letting you know later about what was the result of that. It's a real time operation. The user is waiting on the response from the server. And we need to return the response to them whether it is valid or invalid within a reasonable period of time, because the user is waiting on that particular action to complete.

So that means the systems that we built, they need to handle high concurrency. They need to be real time. Their latency has to be acceptable. The user should not – At least the person perceived latency from the user point of view, they should not have a laggy experience. And that

makes it really tricky. And the system has to be transactionally correct, right? If I begin a trip, we have to be very accurate with respect to at what point the trip has started and what was the timestamp, what was the location? And let's say if we did not record this transaction correctly, then the entire fairs will get messed up. The rider experience will be messed up. So there's a lot of implications if we mess it up. And that means once a transaction has happened, we need to make sure that it is consistent from then onwards.

[00:15:46] KP: In my career, I've been in a couple of situations where I had to recommend a technology to adopt, like a database or something like that. And I get healthy pushback where people will say, "How do you know this is going to scale for our business?" And a trick I've like to use is what I call the big fish strategy. Find a company much larger like Uber or Netflix and say, "They're using the technology. They're 10 times bigger than you. Surely, if there's a bug in it, they're going to find it first." But there aren't any companies ten times bigger than Uber. Do you have to collaborate with partners to get on to new technology or to do bug prioritization, things like that? I guess are there any challenges being a leader or at the forefront of pushing the limits on technology?

[00:16:31] UKM: Yeah. I mean, absolutely. I think that's also an advantage, right? Like if we are at the forefront of adopting new things and taking new things, we also will get that level of support from the partners or from those teams. And like, I think, at least in the last six years, no matter which technology choice that we chose, we would collaborate very closely with that open source team, or whether the third-party provider, or whether the company. And they would give us the right kind of support because they would also know that if we can solve their use case, that would be a good proof point for their system as well. So I know we have many instances like that where we collaborated super closely with that particular platform team even from other companies, and then made sure that we can customize their software to scale to our requirements.

[00:17:25] KP: And can we do a deep dive on some of the requirements? We've talked about it being real time. But there's always going to have to be tradeoffs. We have the CAP theorem to face and things like that. Maybe you want some sort of transactions, or maybe that's not important. What are some of the core features you're looking for when seeking technology?

[00:17:42] UKM: Yeah. See, for our core, state machine stack, and for the core storage system. I think some of the requirements that we want to make sure is, one, with that, is availability, whether single zone or single region. Even if there's intermittent infrastructure failure, it has minimal impact on availability. Because, ultimately, this is one of this core tenant that has to be from Uber stack, because users are in the real world doing operations. Like, in fact, let's say if you're on the street trying to go somewhere, and if the system is down, like then you are standing there. So we need to guarantee at least four nines of availability. We need to have strong consistency. And we have operations on a single row, multi-row, multi-row, multi-table, and we need to make sure we provide strong consistency within a region and across regions, so that even if there's an application failover from one region to other region, we don't have any user perceived inconsistencies. Given we are in – And we are not providing one single vertical, right? Like we are building the world's largest super app. And that means we are doing transportation delivery and all sorts of use cases. And for every country, that means our programming model has to be – To have clean abstractions and simple programming models so that we can have good product velocity. We need to make sure we don't have any data loss whenever there's any infrastructure failures. We need to have support for secondary indices, change data capture. We need to have good latency across. We need to have good latency SLOs. We need to make sure our infrastructure is very efficient, because as we scale our system 10x, every cent that we save is really important, because we are in low-margin business. We need to make sure our system is elastic.

If you think about Uber's workload, it's not steady throughout the day throughout the week. It goes up and down at various points in time of the day and week of the day, and also month of the year. And sometimes December is probably busier than some other part of the year and so on, right? And from our maintenance point of view, we need to make sure that we have low operational overhead, because we are constantly adding new products, new features, new cities. So we need to make sure that we can operate the entire system in with low overhead.

[00:20:08] KP: And I know you've gone through a recent rebuild. Can you talk a little bit about the motivation to kick off a rebuild? Were there ceilings you were hitting? Or was there some other motivation?

[00:20:17] UKM: Yeah, absolutely. So this is one of those things which we invested more than two years in this. And even before we started creating the platform, we spent close to six months trying to understand what should be the architecture that we should invest for the next decade and what are the pain points that we have seen so far? So if you have to summarize some of the limitations that we were facing before. So one is around consistency. And when the previous system was built back in 2014, the entire architecture was built by trading of consistency for availability and latency. And then consistency we would achieve as a best-effort mechanism.

So for example, since we're using Cassandra as a storage, now, which is a NoSQL system, which doesn't guarantee consistency, which is more tailored for availability and horizontal scalability. Now, if let's say – But in our system we have multiple concurrent operations, right? A rider can cancel and a driver can begin trip at the same time. And at that point, how do you make sure you can handle these two transactions that are coming into the system, which are operating on the same set of objects? So to deal with concurrency, we used a framework called Ringpop, which allowed application level serialization.

So what would happen is, for a single job entity, for example, all of the update operations for that entity would be serialized to a single worker. And at that point, in that worker, because it was a Node.js application, it was a single-threaded execution environment, and we had a queue, a serial queue, and an in-memory lock on the object. So then that would make sure that at any point in time, we only have one update operation in flight for a given object. That way we avoided the concurrency issues by eliminating concurrency by having this queue. But this is a best-effort mechanism. Like, Ringpop, if you're doing an application deploy. At that point, we might have a split brain where two different workers might say that I own the same entity. At that point, we might end up doing updates from two different workers at the same time. And then that would end up with consistency issues. Or it could also happen if you're doing region failovers from one data center to other data center. At that point in time, since we are using asynchronous replication from Cassandra, then the other data center might not be the state that is not accurate. And then we might be overwriting to a wrong state. And that would lead into many complex issues. That would be super hard to debug.

The other thing is around multi-entity rights. Like when we started, the number of multi-entity rights, we didn't have that much back in 2014, 2015. Like Uber Pool also was a new thing at that point. But if you think about some of the use cases that we have now where we have like a batch offer, a driver has to accept three trips, four trips at the same time. That is an operation on four different trips and a single supply entity, which is like five entities at the same time. And then we started exploring a pattern called saga pattern, which was doing application level transaction orchestration, which will make sure, "Okay, I do propose on all of these entities." And then once all of them accept, then I commit all of them.

Basically what we were doing at the time was we were doing a lot of what we used to think of like typical database level functionality at the application layer and trying to overcome the shortcomings of not having a storage system that would that would support both horizontal scalability and consistency. And because we had this layers of logic, we also were hitting with scalability concerns. How Ringpop used to function was it was basically Gaussi protocol. And it needs to understand who, and it was peer-to-peer **[inaudible 00:24:09]** protocol. And then if you had a number of – If you increase the number of workers in your application cluster, then the amount of work needed to make sure all of the workers are in sync with respect to the ownership information. That itself was taking a good amount of CPU. So then we are getting scalability limits with their architecture.

So these were some of the things. Beyond this, like essentially, at that point during Node.js. Now we're in Java and Go. The language itself, like almost all the platforms since then had moved to Go and Java. And this was one of the last platforms that was still in Node.js. And that was also causing issues for engineers, for other teams, because it's a tax not just on us but also every team, because now they need to support Node.js environment. And that also made it super complex.

[00:24:58] KP: So you have a service then that a lot of people rely on. How do you orchestrate the roll out of a new system and coordinate with all your different consumers?

[00:25:07] UKM: Yeah. So I think some – In software, it's sometimes more than coming up with the new architecture, coming up with the migration architecture is harder. If it's a green field system, it's always super easy. Like you don't have any constraints. You can come up with like

the cool architecture that you want. But the complexity is always about like we are supporting these 100 different products. They have these nuances. And they were built like this over the last few years. How do we – Without any user having any impact, how do we migrate to the new stack?

So we spent a lot of time trying to understand all of the product flows that were supported by the existing stack. What were the nuances in them? And what is the best rollout strategy? So we picked a city-by-city rollout strategy. Like we created a – So we had a roster of around 150, 200 features that were live that dependent on our platform. And then we did an intersection of that with all of the Uber operational cities. Then we kind of went from the cities with the least number of features and then cities with the – Two cities with the most number of features.

And even in that, we first picked one or two cities in each of those buckets so then we can get guarantees. Then before even we would roll out, we had set up complex shadowing mechanism so that we can shadow every request response from the primary stack and shadow stack and compare the differences. And that also is complex here, because if the requirements and guarantees provided by one stack and other stack are different, and since each – And the output of one operation affects the output of the next operation because the state machine on both sides. If one operation on the shadow stack fails, the second operation and every other subsequent operation will not catch up. So then it will affect how your shadow also. So all of those kind of nuances and how we try to set up the shadowing. But ultimately, we go to a point where we had a good shadowing system that would give us good insights. Then we went from least complexities to most complexities. Rolled out feature by feature. Verified that in shadow environment. And then rolled out few cities with those features. Verified that in production. And then it took close to a year plus for us to roll out all the cities.

[00:27:27] KP: So I believe you said the original system was Cassandra-based with the node kind of a cube system. What's the new technology based on?

[00:27:37] UKM: So I kind of went through the requirements, right? And when we were thinking about what should be the system for the next few years, we were exploring multiple options, whether we kind of move, do some update. Like still use NoSQL, but use some other technologies and maybe like not using pop and use something else. Or use sharded MySQL. Or

then the other option that you're exploring was NewSQL, and some kind of NewSQL way to reach this. Because this is one of those patterns that provides us both horizontal scalability and also ACID guarantees provided by SQL-kind of databases.

So we experimented with a bunch of NewSQL-based storage systems, CockroachDB, FoundationDB, Cloud Spanner. And ultimately we used Google Cloud Spanner, which was a new SQL kind of storage system, because it was a managed solution and it would give us a faster way to move to NewSQL. At least move the application to a NewSQL-based storage pattern. And we had **[inaudible 00:28:38]** layer so that the application developers don't have to worry about the nuance of storage system. But the application developers build in a NewSQL-kind of storage environment application built on top of that. And we used Cloud Spanner as the NewSQL backing device.

[00:28:57] KP: And do you have a good definition or just a working definition for the difference between NoSQL and NewSQL?

[00:29:04] UKM: Yeah, absolutely. So at least how we think about that is if you think about traditional SQL-based databases, their main USP was that you get ACID properties. And most of the e-commerce-based system, e-commerce systems, use that, right? Because if you're using any natural applications, you need to guarantee ACID compliance for any transaction that happens. So then SQL -based systems relied on relational tables. You would have strict data schema, and for storing all of your transactional data.

But they were hard to scale out if you have millions of users and then you would have to think about your own shard. Like then you have to go over to shard MySQL. You'll have to manage your sharding starting strategy. And like there's a bunch of raw issues with that, right? And then NoSQL-based databases emerged to solve this kind of scalability issue. But they confirmation on consistency, right? So you would focus on availabilities of consistency. And then a lot of key value stores like Cassandra, DynamoDB **[inaudible 00:30:04]** like MongoDB, columnar issues HBase. All of these systems kind of emerged that primarily focused on building internet scale applications that handle millions of users workload with good end-to-end latency. And that was the prime reason why it was the natural choice for even fulfillment style back in 2014. And in the last a few years or so, now we see a new trend with a NewSQL-kind of storages.

And what NewSQL-based systems provide is they provide both the ACID guarantees that SQL-based data stores used to provide and horizontal scalability that NoSQL storage systems provide. So that was a good – That was kind of what we needed in a new stack. So that's why we took the plunge to NewSQL-based storage.

[00:30:55] KP: And what made Spanner the ideal choice? There's a lot of options out there.

[00:30:59] UKM: Yeah, absolutely. So as I said, like after we went through all the requirements, we evaluated various options. We created benchmarks. And we benchmarked a bunch of solutions. during at that point in time, I think Spanner was the most scaled managed solution, because we didn't have any precedent within the company at that point in time of any system using NewSQL-based storage. So if we had to take any open source and if we have to protectionize within our environment, it would have delayed the overall migration effort. So we had to choose some managed solution. Because along with this, we also have to change our – We have to move from Node.js application stack. We have to move to a new programming model. We need to change our data model to support a bunch of new features that we needed to unlock in this year and in the upcoming years. So we have to decouple. Like we have to accelerate the application migration. So we needed something that provides a managed solution for NewSQL. And that's why spanner was – Like we ultimately settled on Spanner.

And within Spanner, they provide both a single region and multi-region. So we chose a multi-region configuration that guarantees five nines of availability and strong consistency. They also provide external consistency, which is strictest concurrency control guarantee for transactions. They also had other features like a point-in-time rates and bondage stillness rates. They did detection for deadlocks automatically and a bunch of other features that we felt were okay. We could build an application architecture around this. And that's why we went with Spanner. But one is, unlike most common scenarios in our case, we had our application stack running in Uber's operational regions and it would connect to Spanner that is deployed in JCP.

[00:32:58] KP: Are there any challenges around data center lag or anything like that?

[00:33:03] UKM: Yeah. So any cross data center request obviously it will add some additional latency. So we try to optimize at multiple layers at the networking layer. We work with the networking team from uber and Google networking teams to set up the identity connects, to set up the right redundancy. To make sure that we can have strong foundation there. And at the application layer, what we try to do was to reduce the number of round trips for a given user request. Let's say a driver accepts an offer. If that requires – If without optimizations, if that required a read of three different entities and an update to three different entities and begin transaction and in-transaction. So if we did a bunch of optimizations, we had sessions prepared ahead of time. So that when a request comes, we didn't have to prepare a session. So that would save one round trip.

For a given user request, we looked at the data dependencies and we used to coalesce the transaction. So we would not do a round trip to Spanner until we needed to store that state even within a transaction. So that way we reduced the number of round trips back and forth. And ultimately our benchmark was the final user level – Application level latency should be better than what we had before. And that way we were getting more transactional guarantees that the previous system was not able to handle without sacrificing the latency. And we were able to achieve that with all of the optimizations and **[inaudible 00:34:39]** even payload compression and all sorts of things to reduce even further.

[00:34:47] KP: At the point when you got it rolled out in the shadow state alongside your existing infrastructure, before you got into some of the performance results, did you have any expectation about what KPIs were going to be important or maybe what improvements you expected to see?

[00:35:03] UKM: At the application layer, one of the main things that we're monitoring was obviously the availability. Not just at Spanner server level, but from the application layer, which is creating and completing transactions. Because we have multiple hops from the application layer. We have to go through networking stack and through Google frontend, Google Spanner frontend, Spanner backend. We're looking at the application level availability from the client side that we monitor. We're looking at latency not just from – And latency from both of the Uber regions, because the networking paths from Uber west coast region and east region to Spanner

leader region is different, right? And the latency that we get from both of these regions is slightly different. So then we need to monitor the performance from both of the agents.

Obviously, we're looking at the – Let's say we begin trip operations. What was the error rate and latency at that layer, like, from the API that the mobile app used to call? Because, ultimately, that is the final API that we would look at. So all of the mobile APIs, we would look at the error rate availability before and after and also the error rate at the Spanner client layer from our application side and also the Spanner server level. And we would not roll out if we were not getting these – If our latency is worse than the previous one, or if the availability is worse than the previous one.

And I also mentioned the shadow stack. So they will also monitor at the property level for both for responses from both primary and shadow stack. What were the number of divergences? And then we would look at we would only roll out if we are confident that all the property divergences that need to take care of are covered and we had a high bar for that.

[00:36:55] KP: And as you've gotten past that kind of proof-of-concept phase and rolled this out more extensively, how's that process gone?

[00:37:02] UKM: Yeah. So I think that was probably the most challenging part of the whole project. And especially if you think about our stack – So let's say a user, a driver went online. And they are online for multiple – Like they could be online for multiple hours. And if they went online in the old stack, their state is stored in Cassandra. And if they went online in new stack, their state is stored in Spanner. And these are storage systems with different guarantees and different things, right? And we cannot have a storage sync across these two systems. So we needed to create a migration strategy where we can gradually roll out users and ongoing trips from one stack to the other stack. So we built an interceptor that would pin a user session or a trip to a particular stack. And then until the trip is completed or until their session is done, their request would be pinned to that stack. And once they went offline and they came back online, then their request would go to the new stack. So that way, in fact, like for some period of time for a given city, both the stacks would be operational and then you would see users and trips go down from one stack to the other stack over the next few minutes and hour. And this was kind of the migration setup that we did.

And then as I said before, then we tried to go from least complexities to most complexities. We did in fact even on the ground testing, we pinned some test riders and test drivers. Like when they were taking trips in the real world, we monitored each and every operation. That was like before we did the first city. Then we did the most simple one simple UberX city. We just had few features. Looked at all the metrics, all the error rates, all the end-to-end operations, all the bugs, all the contact rates from the from that city. Then we kind of progressed from that to from one city to cities with just UberX plus simple Uber Eats cities with UberX, plus airports. It is with UberX with airports and some other kind of features. So that we kind of gradually expanded the feature set. And once we covered a good bunch of small cities and big cities with different feature set, then we had a scale out phase where, okay, now we need to scale out 100 cities at once. And then we had to build a lot of custom tooling to measure the absorbability of each and every city at both application metrics and business metrics in the old stack and the new stack like an hour before the migration and hour after the migration and compared. And if we see any differences, then we would either decide if we need to roll the city back. Maybe we didn't fix some flow, and we missed some flow. And because of that, it's causing rider cancellations to go up. So we needed to roll that city back to the old stack. Debug that scenario and then roll it forward in the next batch. So we had – Essentially, every two weeks, we were rolling out some cities and going through this operational process until we finalized the final set of cities.

[00:40:22] KP: And is the project – Do you consider it fully rolled out at this point? Las the old service been decommissioned?

[00:40:28] UKM: Yeah. And right now it is fully rolled out. The old service is fully decommissioned. And now we are 100% on the new stack. And now we are in the process of – Now that this first leg for us was to, "Okay, how do we move all of the existing products that was supported in the old one?" And as we were rolling out, we already had encountered some new features that only the new stack could support. So we had a period where if for any new features, they would only go in the new stack. And we would prioritize rolling out the cities in which we would need to experiment those new features. So that way we don't – Not every team plays a dual platform tags and like they don't need to implement in two stacks. They only implement in the new stack. A lot of new features that we could not build in the old one, now we are able to unlock. And over time uh the goal is we want to support many different verticals, not

just for transportation, both in transition mobility. Different fulfillment types, different verticals, as we go into grocery, alcohol and all kinds of retail. Like there's some new fulfillment flows there. And even on the mobility side, like you have seen Uber reserve. It's a completely new product that we launched last year that was fully built on the new stack. And a bunch of new features that were already built on the new stack that we could not have built in the old stack where we had to do all signs of crazy hacks to even build in the old one.

[00:41:56] KP: What is it about the new stack that makes it possible to build these features with greater ease?

[00:42:01] UKM: So I think along with all the things at the storage layer, we spent a lot of time at the application layer trying to create a modular programming model that will give us the flexibility to build different kinds of building blocks that can be used in different features. For example, in the previous one, we didn't have a good way to create new kind of tasks for a particular waypoint. Now we completely platformize that component. So then now, for a waypoint, we can attach different kinds of tasks and we can create different kind of task flows inside that waypoint. What that allowed is if you want to add pin verification task before you begin the trip, you can easily add that new task. And then once one featurity match that task as a component, now maybe risk team wants to add pin verification for some other scenario. So they can leverage the same building block in their particular feature. So we try to build as many building blocks as possible that can be leveraged across multiple different products. So then we have that leverage aspect. And then programming model helps us build these kinds of building blocks more and more. So then we create a repository of these building blocks that can be used in different mix and match scenarios and create different product experiences on top.

[00:43:22] KP: Well, with the platform delivered and the benefit of hindsight, do you have any closing thoughts on what the major advancements were? Is it obviously unlocking new features? Are there scalability and consistency wins as well? What are some of the main takeaways?

[00:43:37] UKM: So I think one is around consistency and like at the end infrastructure layer, the consistency and scalability. Now we have a platform that is horizontally scalable. In fact, it's a living, breathing thing. For throughout the day and throughout the week, the number of

storage instances we keep increasing and reducing. And it is completely auto scaled system, which is horizontally scalable to whatever scale that we need. And because we have strong consistency, it makes it really easy to reason about the system. And when some things go wrong, you know if something has happened. That fact is done. Like you don't have to guess, "Oh, did the system record this fact or not? Maybe there was some inconsistency because of which this operation did not happen." Now you can look at the system logs and you know for a fact that, okay, if this event is emitted, we know that the system will never be an inconsistent state. And at the application layer, we duplicated Node.js. We have Java. Now we use – We can leverage all of the new libraries and frameworks that Uber platforms build for Java services. That will also reduce the amount of operational overhead that we had in operating a Node.js service. Now we can leverage a breadth of knowledge across Uber in operating the service. And at the application layer, we completely revamped the data model. That helped us build a lot of new features. And a lot more are in the pipeline. So that's also a huge benefit. I think one more side effect that we got out of all of this was reduced infrastructure in spend, that at least at the application layer we reduced the number of application workers needed by a lot by significant percentage compared to the previous stack.

[00:45:33] KP: Very cool. Well, Uday, thank you so much for coming on Software Engineering Daily and sharing your work.

[00:45:38] UKM: Thank you so much. It was nice talking to you, Kyle. Have a good day.

[END]