Client-Transparent Migrations

Pavan Adharapurapu details how to approach large-scale migrations while optimizing for user experience.

MP: Hello and welcome to the Google SRE podcast, or as we affectionately like to call it, the Prodcast. I am your host for today, MP, and along with me is Viv.

Viv: Hi! Happy Episode 5.

MP: And with us today is Pavan. Pavan, why don't you go ahead and introduce yourself?

Pavan: Thanks, MP and Viv. I'm so excited to be on this podcast. Hello, to all the listeners. My name is Pavan Adharapurapu, and I'm a software engineering tech lead and manager in the API platform group here at Google, where I've been working for the past 9+ years. The API platform where I work is part of the backend that processes Google web API calls, or REST API calls. When you use, say, the Google Calendar app on your mobile phone, or create a YouTube playlist on your browser, or upload a Google photo, they all result in API calls to application servers running in Google Cloud.

The API platform is a layer of those application servers and provides functionality that is common to all Google web APIs, such as authentication, rate limiting, data translation between JSON and internal protocol buffers, building telemetry, and so on. In the recent past, I led a successful multi-year migration project that moved hundreds of Google web APIs to a newer and better API platform. Most importantly, this was done in an invisible manner to the millions of applications

and users that consume these APIs. As part of this project, I learned a lot about the challenges involved, common solution approaches, and most importantly, the mindset one needs to have for performing complex backend migrations.

MP: Yeah, so that's a problem that comes up in a lot of different forms for both SRE and developers: that we want to change something really significant about our system, but we don't actually want customers or users to notice anything has changed. We want it to be transparent to clients. What does "client-transparent" really mean? Does it just come down to byte-for-byte identical responses between an old service and a new service, or is there more to it?

Pavan: Yes, client transparency, as you mentioned, is definitely in walls matching responses, but it is not the same as matching responses. It's more than that. So first, let's define client transparency. Client transparency is a property of a migration that comes about by the source and target system behaving the same way for all user-observable aspects, for all the existing traffic. These qualifiers are important. We don't care if two systems do not match in behavior for non-user-observable aspects, or for requests that are not part of the current production traffic set.

So what are the user-observable aspects? Having a compatible response for every existing request is definitely one of the aspects, but there are two other aspects that make up client transparency: namely, compatible state changes and comparable latency.

MP: Can you talk more about state change and how that plays into this? What does that really mean?

Pavan: Sure. Yeah. So the compatible state change property requires that the state change affected by the new system must match the old system for a given request. For example, one could imagine two systems that both reject and update requests if it has invalid credentials. Further assume that the errors returned by both systems are exactly the same. So we have matching responses, but still the migration between these two systems could be not client-transparent. For example, suppose the first system mutates the resource

before authenticating the request, while, say, the second one authenticates it after mutation. Then even though both the responses are the same, the final state— the final state of the resource— differs between the two systems. This is a contrived example, but you get the idea why state change should match between the two systems.

One other thing is that the state— when we talk about the state, it is all pieces of the state that is accessible to the user. So that certainly includes the underlying resource state. Every service has an underlying resource that it manages, but it also includes other parts of the state— like billing state, rate limiting state, telemetry state— because these are all accessible to the users. For example, billing state is accessible when the user gets an invoice at the end of the month. The telemetry state is visible through charts and dashboards and Google Cloud Console. So we need to ensure that all of the state is updated compatibly between both the systems.

The last component that I mentioned that makes up client transparency is latency. It has to be comparable because a significantly deteriorated latency for the new system will definitely be noticeable by the user, and you couldn't really call it client-transparent migration.

Viv: Okay, so you've mentioned three components here. Are these the top three components, or is this all we need to consider? And how do we know that that is the appropriate amount to consider for client transparency?

Pavan: Great question, Viv. Indeed, how do we know that these three components that I mentioned— namely, compatible responses, compatible state changes, and comparable latency— are not only necessary, but also sufficient conditions for client transparency? Well, that follows from the fact that every real world software can be modeled as a finite state machine with non-zero propagation delay. For a given input, the only things that can be measured for such a finite state machine are the output, the state transition, and propagation delay. And therefore it's enough for us to just make sure that these three components are matching between the two systems and client transparency follows from that.

Viv: Interesting. Makes sense. Although I am sure it is much easier said than done.

Pavan: Mm-hmm (positive). Indeed.

MP: It all comes back to modeling things as finite state machines.

Pavan: Indeed.

MP: So one thing I'm curious about: I think it's a fairly well-known phenomenon that when you have a large enough API with enough possible ways to use it, someone somewhere is going to start using your API in an unsupported or undocumented way. How are those sorts of things handled when trying to do client-transparent migrations?

Pavan: You touch upon the main reason for complexity in migrations, MP. Indeed, a system with a wide surface area and with a large and diverse usage will have some clients depending on implementation traits of the system. There is even a name for this. It's called the Hyrum's Law. It's shortened as, "the implementation is the interface for large systems." It is important to point out that client transparency requires sequencing of behavior not just for documented features, but also for accidental features, implementation idiosyncrasies, and bugs as well. Since, as the Hyrum's Law says, there will be users who would inadvertently come to depend upon them.

MP: There's always someone where the saying "that's not a bug, it's a feature" is actually true for that user.

Pavan: Yes, it is. Yeah.

Viv: So is that a small percentage of users? I guess I'm curious: when you're accounting for the entire set of people who use your service, if you are trying to do this client-transparent migration, are these bugs and idiosyncrasies just for a few people? How do we decide that that matters and that we want to account for all of that since I know all this can be very expensive and already pretty complex?

Pavan: Yeah, so how many users depend on these non-main line features? It is really a statistical property of your system and usage. During our client-transparent migrations we noticed that a majority of the users indeed use the system as intended— as designed for the mainline scenarios. However, there is a non-trivial portion of the users who depend on these other behaviors of the system: the ones that are not designed intentionally, but are just properties of the underlying tech stack or some bugs. And the interesting thing is that we see that different users use different— depend on different bugs and different accidental behavior, so it's kind of spread out. And that's the property we see for many other migrations, as well.

The important thing is that those users have to be cared for. You cannot just ignore those use cases just because they're a small percentage of it, because what is an off-label usage of your system can be the basis of a mission-critical workflow for that user. So when it comes to client-transparent migrations, the behavior of the current system is always correct. That's what you assume. It's the spec, and that's what you have to ensure is carried over to the new system, as well.

Viv: Makes sense. I like it. It's like no user left behind.

MP: Well, I'm wondering, is that an absolute? Like, if you had one user customer that was relying on some sort of idiosyncrasy and porting that functionality over to the new system... and hypothetically introduced a massive complexity increase? You had confidence that they were the only one doing it. Would you ever consider asking the customer to change what they're doing so that it doesn't have this kind of ripple effect? Is there a balancing act there?

Pavan: That's a great question, MP. So, indeed, when we say that client transparency is an extremely important criteria, is it absolute? Like, is it a 100.00%, or are there cases where, you know, we could make the case for reaching out to the customer and politely requesting them to change their code to not depend on some undocumented feature or a bug in the system?

So first thing is that there should be an extremely high barrier for doing that. Why

do we have that? It's because making even small changes to production applications can be a huge cost for the customers, and if the customer uses a service through mobile apps/smart TV apps, then they may not even be in control of the update life cycle. It would be the end user.

And another important point is that customer applications which have been running for a long time in the field are generally part of very important workflows for the application users. So making even small changes requires a lot of testing to ensure that these important use cases are not broken. So the barrier should be extremely high, but if in the extreme case as you mentioned, MP, where it's just one user and where creating client transparency causes a huge amount of complexity on the service side in the new system, then maybe you could make a case for reaching out to the customer. You know, giving them enough time, providing enough documentation to make the change that you think would be necessary to not port that buggy behavior or accidental behavior into the new system.

MP: So it sounds like it is that usual complexity trade-off, that it's either going to be complexity somewhere inside of the new service, or it's going to be that complexity would get pushed back off to the customer, and there needs to be a really substantial difference in— like in my head, I'm sort of saying it needs to be orders of magnitude easier for the customer to make the change than for the migration to be transparent for that use case, is sort of what I'm thinking in my head.

Pavan: Yes, exactly. It's almost like you have to have this mindset that you will never burden the customer, and the exception has to be so clear that it proves that rule.

MP: What are some of the other difficulties? What are some of the other things you have to look out for when you are trying to ensure you have a client-transparent backend migration?

Pavan: Yeah, so there are four major challenges we found during our migrations. The first one is knowing the complete set of disparities between the two

systems. Generally, it's not too difficult to figure out the major disparities at the well-known feature set level. For example, one system could support a particular type of authentication, whereas the other system doesn't. It's pretty easy to find that out, but it's much harder to do so for bugs and implementation traits.

The second challenge is testing two systems for parity. How do we prove that the migration between two systems is going to be client-transparent? The regular unit and integration testing cannot cover all the bugs and accidental features of the system.

The third challenge we found was that it is hard to know when you broke the customer, because not all compatibility issues can be detected server-side. Let's say the new system adds some innocuous white space, which is allowed by many specs. For example, JSON allows white space, and let's say a not-so-well-written customer application stops working because of it. Without client-side telemetry, which is generally not available for a public service with diverse clients, you cannot know that you broke this client at the server side. And this is one of the core challenges of ensuring client-transparent backend migration.

And the final challenge, which we touched upon, is that we could end up bloating the new system with parity fixes— especially the old system idiosyncrasies— and this could reduce the maintainability of the new system for all users, current and new.

MP: None of those challenges sound very easily surmountable. It sounds like a lot of the, "it's really hard to know what you don't know."

Pavan: Indeed. Yeah. Unknown and knowns. Right.

MP: Is there a methodology for these migrations that we can take kind of generally to help make sure we meet all of those challenges?

Pavan: Yes, definitely. There is an engineering approach you could use to get control over the disparities between two systems and true client transparency.

But before I go there, before I outline the approach that we used successfully, let me take a step back and discuss how we should approach a complex migration in the first place. The first thing to do is to avoid a large-scale complex migration in the first place if possible. We should only—yes, it's a...

Viv: Don't do it!

Pavan: —it's a surprising thing to say, but I think this is very important. We should only attempt one if the long term benefits to users and not just a service owner significantly outweigh the cost to the user. Migrations are fertile grounds for all sorts of compatibility issues. It's not to be undertaken lightly. One must do cost benefit calculations, get it reviewed, signed off by, you know, your team and leadership chain.

Now, once you decide to migrate, embrace the risk. Do not second guess. Next, approach migrations with a spirit of constructive pessimism. We hope for the best, but we always plan for the worst. Then, before you start migration, you should ensure to prevent any backsliding from happening. That is, make sure no new usage is allowed on the old system— otherwise, it would be like pumping water out of a pool that is being simultaneously filled by water. Not the most efficient way to migrate. Next, aim and plan for client transparency as an overarching constraint for the migration. Anything you do as part of the migration should be invisible to the user, and this cannot be emphasized more. Finally, have a plan be ready in case the migration becomes intractable or fails. There's no point in continuing with the migration that has caused a lot of issues to the users. It's better to just switch over to another plan that continues to serve the users without causing any more issues.

MP: The last one there sounds like it's trying to warn about the sunk cost fallacy a little bit?

Pavan: Indeed. It is immaterial that you spent a lot of effort, a lot of hard work, in taking the migration up to a certain point. That cost is meaningless if in going further you're going to inconvenience your users even more, right? So you should be careful of the sunk cost fallacy, as you said. What matters is from this point

onwards, what is good for your users, for your customers? You always have to follow that path.

Viv: So perhaps just a note to continually evaluate whether or not what you're doing is in the best interest of users as you proceed.

Pavan: Indeed. Yes.

MP: So now, once you've decided to move forward with the migration, how do we try to guarantee— especially when we're putting client transparency as one of our overarching requirements for the success of the migration— what's the strategy there?

Pavan: Yes, so the strategy has the following broad steps. To ensure client transparency in your migrations, first of all, we need to enumerate all elements of the user-observable state. As we mentioned before, it could include more than just the resource state. It could include things like billing state, telemetry, et cetera. It is important that we identify all user-observable state because if you don't know what can start behaving differently, you cannot track it or prevent it from going bad.

MP: What about just, like, server logs? Do those count as state?

Pavan: Server logs count as state only if they are exposed to the user in some fashion. If some pipeline consumes those logs to generate some reporting that is exposed to the user, then yes. It is part of the user-observable state, and you have to make sure that it's updated consistently between the old system as well as the new system.

Viv: That sounds really hard.

MP: Yeah. Like, would this go all the way down to error messages?

Pavan: Well, it's the same thing again. Does the user see the error message in one form or the other? If the error messages are just used for internal debugging by the service developers, then no— you don't have to care about it. But if it is—

let's say you scrub those error messages, filter them, and maybe you display it in some console and the customer uses it regularly for doing something important, then guess what? It fits the definition of user-observable state, and you have to make sure that it's in the similar format and similar structure as the old system.

This, I believe, is a critical prerequisite for client transparency. It's as simple as saying if you don't know what can break the user, then you can never guarantee client transparency. If you only focus on the most important state like the resource state in the database— if that's all you care about, that's all you focus on— then you would be surprised by customer tickets that says, "Hey, my dashboards are behaving funkily", or "my billing is off," or "my requests are getting throttled much more aggressively than before what's happening." Right? So you will eventually discover all the user-observable state one way or the other. It's better that you sit down and then you enumerate it as part of your migration planning.

So let's say you enumerated all the user-observable states. The second thing you do is you quantify the disparity between both the systems. This is the hardest part of ensuring client transparency. We need to quantify compatibility issues between the two systems. If you cannot measure it, you cannot control it. The only practical way that I know of doing it is through what we call as production traffic replay, or sometimes also known as "dark launch." This is where production traffic to the old system is simultaneously run through the new system, and the responses and state changes are compared between both the systems. And I'm happy to go into more detail later on it.

And moving on with the different steps for ensuring client transparency, the next thing we do is that we limit blast radius of potential compatibility issues during traffic cutover. Regardless of how much effort you put in, it's hard to guarantee that two systems are 100% compatible. There are always unknown and knowns, and so we have to ensure that any damage they do is limited, and we generally do it through a gradual rollout strategy.

The next step is detecting incompatibility issues during cutover. So for this we use monitoring for issues that can be detectors server-side, such as errors that

happen only on the new system. For issues that cannot be tracked server-side— remember we discussed that not everything, not all compatibility issues, can be detected at the server side— we closely monitor all customer support channels for tickets or complaints from the customers. Once again, we should never, ever depend on the customer for identifying incompatibilities, and we track customer channels only as a precaution and only for learning about unknown and knowns. And when we do detect issues, we need to mitigate, and that we do through a fast rollback to previous state or by moving all traffic to the original system.

Finally, we need to correct any issues detected during rollout, and as part of that, do a postmortem, review the overall migration plan to see how this was missed, and if everything looks good, proceed.

MP: A lot of this sounds like fairly standard SRE philosophy in terms of limiting blast radius to being able to detect problems, mitigating problems, doing appropriate postmortem review after the fact. So, like, the two things that really stick out are this state enumeration and this production— this dark launch thing. And we've talked about the state enumeration a fair bit. We've talked about how that's actually very core to the idea of client transparency.

I'm curious more about this production traffic replay, and I can imagine a lot of ways that that is not— it's not just as simple as sending requests to the new service. I have a suspicion that since we've been talking about state, I have a feeling that state changes are going to be kind of complicated once you get— like, if you don't have idempotent service, replaying traffic is not always gonna be a good thing to do.

Pavan: Yeah, that's a great point. Yeah, MP, yes, that is definitely a key engineering consideration in any production traffic replay system. Before I go into it, let me quickly touch upon why we need a production traffic replay system that takes the production traffic and replays it through the new system that you're trying to migrate to. So we already mentioned that large systems, they have a lot of accidental behavior. So you cannot prove the client transparency property by analyzing the system in vacuum, right? You cannot just go through the source code of the system. It could be too huge, or even if you could go, it's very hard to

prove that they have a compatible behavior. So the only way to do it is to actually run the existing production traffic to the old system on the new system, and actually check that their responses, state changes, and latency match.

The basic idea is this. We continuously sample production traffic to the old system. We capture the user request, the response from the old system, and the state change from the old system. We then later replay the same user requests through the new system, and once again capture the response and state change from the new system and compare them. If there are no diffs, then it's safe to migrate between the two systems. When there are no diffs for all of the production traffic, right? But if there are any diffs, we treat the new system until the diffs disappear regarding that state. So we should ensure that no state change happens due to replay traffic. It's very important. So how do we compare the state changes between the old system and the new system while ensuring that no state actually changes by the replay traffic? That goes into one of the important engineering considerations in building off a production traffic review system.

So how we do that is by first isolating the layer that changes between the old and new systems. For example, the database generally remains the same between both the systems. You don't rewrite everything from the ground up, like everything from top to bottom, right? So for the layers that don't change, we capture the messages that cross this layer boundary for the old system, and we replay it for the new system. So for example, when the old system tries to read from the database, and the database returns a certain value, we capture that. And when we replay it in the new system, we replay that captured database value, as well. And similarly, when the old system writes to the database, we capture that, and we compare it with the captured value from the new system. So that way we never change the database— we just use the captured value for replaying and for comparison, and that's how we ensure that we compare state changes without really changing the state on the new system.

Viv: So that sounds like it could get pretty complex. Do you have a list of four or five challenges or things to watch out for while you're doing that?

Pavan: Yeah. So in building and using a production traffic replay system over the years, we've learned, you know, a lot of things. First thing is that we should ensure that the production system is not affected due to sampling and interception and capturing, right? It's like the equivalent of the Hippocratic Oath: do no harm. You don't disrupt the existing traffic on the existing system in your attempts to create a pathway for a safe migration.

The second (we already discussed) is not changing state for the replay traffic. As MP mentioned, not only are all systems not idempotent, it's also that we don't know what are the repercussions downstream from changing the state. There could be multiple other systems which could be reading from it, so we should never touch the state as part of replay.

Next, we should ensure that all privacy principles are followed, even when we replay, and we diff the two responses.

Next, we need to be able to make sure that we sample all kinds of traffic. There should be no gaps. If there are any gaps in sampling— certain classes of traffic— then you could have incompatibilities hiding in there that will bite you during the actual migration.

Then there are some practical difficulties, like knowing how much to sample, how long to sample, so that we cover all customers. There could be customers who make infrequent requests like once every week or even once every month. So, how long do you sample?

Finally, you know, there's some timing issues between when you capture and when you replay. For example, our tokens expire within a window, so if you replay it after that window, you might detect spurious diffs. There can also be flukes. You know, network is— you know, they could have flakiness and all sorts of things that could result in issues that could take a long time to analyze.

Regardless of all these practical issues, production traffic replay is exceptionally effective in quantifying the disparity between two systems and guaranteeing that

migration will be client-transparent with a very large probability.

MP: So is production traffic replay enough to— so you build your production traffic replay system and you get to the zero diff point. Is that sufficient to convince yourself that the migration will be client-transparent? Are there any blind spots that production traffic replay has?

Pavan: So reaching diff zero takes you a long way towards client transparency, but there are things that you could still do outside of production traffic replay to provide more confidence.

For example, production traffic replay, as good as it is in ensuring that responses and state changes are compatible, is not so great at proving that they have comparable latency, because it's hard to mimic [the] production load profile when doing sampling. So you may want to use load tests for that. And similarly, you may want to use integration tests, which we already said are not sufficient for proving client transparency, but they're great for establishing a baseline level of compatibility before you employ production traffic replay for flushing out the remaining compatibilities.

And you could also use other techniques like error fallback. Many of the incompatibilities result in the new system throwing errors where the old system processed. So you could also have some sort of a reactive mechanisms where whenever there's an error on the new system, you forward it to the old system for processing, for second-chance processing, when it happens on the request flow before the state changes happen. But yes, we should keep an open mind and try to use other mechanisms in addition to production traffic replay to beef up the competence in our migrations.

MP: So up until this point we've spent a lot of time talking about validating the new system and making sure that the behavior of our new system is correct for our users and customers. But how do we actually approach moving traffic? Which is really the true validation, is that you move all the traffic over and no one complains.

Pavan: Yes. So the cutover phase, the, you know— or in migrations, the rollout is generally called traffic cutover— that is definitely exciting, but also a phase where you also have a lot of tension. Because that is when all this hard work you did in validating the compatibility of the two systems is put to test.

First thing to note is— first thing to realize is that unlike new product or feature launches, the blast radius for migration cutover is quantitatively and qualitatively larger. It's quantitatively larger because you have a large amount of existing traffic. There is no organically increasing traffic that starts at zero, like for new product launches. It's already there. It's qualitatively larger because this is established traffic. This could be from a customer who's been sending traffic for the last ten years. Imagine all the use cases, all the workflows that this traffic is powering. And so request for request, the blast radius is huge for migrations. It's definitely more painful for the user when some disruptions happen.

So with that said, what are the goals for the cutover? They remain the same as for new launches, which is: we would like to ensure a low mean time between failures, a low blast radius or impact, and a very low mean time to repair. So, how do we ensure low mean time between failures, as well as low blast radius? We primarily depend on fast and easy incremental ramp-up and ramp-down of diversion traffic. Fast means multi-hour, not multi-day. If it's multi-day, then it's hard to track the health of the incremental ramp up. If it's multi-hour, it's sufficiently gradual, and it also enables, you know, tracking of the various health metrics.

And I'm talking about incremental ramp-up, not the entire rollout. So even within the gradual rollout, it should be very slow and very small in the beginning, and it can be accelerated somewhat after reaching 5%. We generally notice that it's very rare to see new issues after crossing 5%. Still, you should not move more than, let's say, 15% maximum traffic between every step, and in the beginning we generally start at something very small, like 0.001% or less. Definitely less than your error budget for your service. And leave it there for a week, because with enough time, even a small sampling ratio results in a lot of the customer traffic being eventually sampled. And then we gradually increase the increments, and

after we reach 5%, we go 15% every three days.

One important point is that the diverted traffic must be randomly selected. This is so that every customer sees a gradual rollout from their point of view. If it's not random— for example, if you hash by IP address or some sort of ID— then what happens is that the customers whose traffic is migrated at the end of your rollout suddenly see a large shift in traffic from the old system to new system, and in case of any incompatibilities, suddenly a large part of their traffic is getting affected, and random selection avoids that.

And finally, as part of the careful rollout, the way you guarantee a very low mean time to repair is through fast rollback. It's your big red button. You should be able to roll back to either the previous state, the previous split between the old and new system, or maybe rolling back 100% to the old system. You should be able to do it very fast— like in seconds or minutes. And so it's important to build such a fast rollback mechanism before you even start migration. As they say, in case of doubt, roll back, and then investigate. And that's how you ensure that not only is the damage limited, but also you get your customer traffic healthy very soon after you detect something suspicious.

MP: Yeah, [a] migration plan should always come packaged with the rollback plan.

Pavan: Indeed. Yes, it should have both an entry and exit criteria for different rollout stages so that there's a clear signal that we are good to go on to the next stage or when we have to roll back from that stage.

MP: So this is all a lot to think about and consider when trying to migrate a service, particularly the aspect— the client transparency— specifically, there's a lot to account for there. Are there situations where client transparency shouldn't be the goal?

Pavan: The answer to that is, surprisingly, yes— but only for very few and specific scenarios. So you should not give priority to client transparency for security and privacy issues. If that's part of migration— you realize that there is a security

vulnerability in the old system that could be exploited and that could hurt your customers— then it's important that you don't silently port that security issue to the new system, because in the long term it causes more cost to your customers. So it's better to inform your customers about it and then plug that gap.

And there is a case where you need not aim for client transparency, and that is for a service that has gone past its support window. Although in such cases, it's good to have some courtesy advance notices to your customer about the impending migration.

But otherwise, client transparency is a very strong requirement for all migrations, and ignoring that can be a very frustrating experience for all the parties involved.

Viv: Yeah, for sure. Especially the more complex a service is. Yes.

Pavan: That is true. A lot of the things that we discussed either do not apply, or get diluted for services that are either simple or do not have diverse traffic. For example, we migrated a service that had a very large amount of traffic, but it was one of the easiest services to migrate. And the reason was, all of the traffic was coming from one client that was owned by Google itself, so we knew exactly the pattern of requests that would be coming to the service. And so it wasn't the case of an open-ended set of clients, you know, sending all sorts of patterns. It was a very simple and a uniform pattern. So it was very easy. So a lot of these things, a lot of these challenges in client transfer migrations, really come about for systems with large surface area and large and diverse usage.

Viv: Cool.

MP: Anything else you'd like to share with us today, Pavan, before we wrap the episode up?

Pavan: Yes. The one thing I would like to reiterate is how critically important client transparency is for any backend migration. The customer point of view should be given primacy over the backend system view. What is an edge case to the backend system could be a mission-critical use case for the customer, and

ignoring that could lead to much customer frustration. So I know we repeated this many times, but it's really, really important to always think from the customer perspective and ensure that all your migrations have client transparency as a P0 constraint and also plan for it.

Viv: Thank you so much.

MP: Yeah. Thank you so much for your time today. It was really great talking to you about all of this. I learned a lot.

Viv: I also learned a lot.

Pavan: Thank you, MP, and thank you, Viv. It was a pleasure to discuss the knowledge and the techniques that we learned as part of these complex migrations that we have done over the past, and I hope they are useful to the listeners who are planning their own migrations in the future. And good luck migrations to all those.

MP: I know I'm definitely gonna be taking things I learned here into my day-to-day practice.

Viv: Me too.

Pavan: Great.

MP: Well, thank you.

Pavan: Thank you. Thank you, everyone.