EPISODE 456

[INTRODUCTION]

[0:00:00.4] JM: The Netflix API is accessed by developers at Netflix who build for over 1,000 device types. TVs, smartphones, VR headsets, laptops, if it has a screen, it can probably run Netflix. On each of these different devices, the Netflix experience is different. Different screen sizes mean that there is variable space to display the content.

When you open up Netflix you want to efficiently browse through movies. The frontend engineers who are building different experiences for different device types need to make different requests to the backend to fetch the right amount of data. This was the engineering problem that Vasanth Asokan and his team at Netflix was tasked with solving. How do you enable lots of different frontend engineers to get whatever they need from the backend?

This problem led to the development of a "serverless-like platform" within Netflix, which Vasanth wrote about in a few popular articles on Medium, which are linked to in the show notes for this episodes. This platform enables frontend developers to write and deploy backend scripts to fetch data decoupling the responsibilities of frontend engineers and backend engineers.

The tight coupling of frontend and backend engineering was problematic to the development velocity at Netflix, and if you know anything about Netflix, tis culture, you know that development velocity is the height of importance.

We've done many shows about Netflix engineering covering topics like data engineering, user interface design, performance monitoring, Falcor. If you want to find these old episodes, you can download the Software Engineering Daily app for iOS and for android. With these apps we're building a new way to consume content about software engineering, and they're open sourced at github.com/softwareengineeringdaily. If you're looking for an open source project to get involved with, we would love to get your help.

[SPONSOR MESSAGE]

[0:02:04.5] JM: Auth0 makes authentication easy. As a developer, you love building things that are fun, and authentication is not fun. Authentication is a pain. It can take hours to implement and even once you have authentication, you have to keep all of your authentication code up-to-date.

Auth0 is the easiest and fastest way to implement real-world authentication and authorization architectures in to your apps and APIs. Allow your users to login however you want, regular username and password, Facebook, Twitter, enterprise identity providers, like AD and Office 365, or you can just let them login without passwords using an email login like Slack, or phone login, like WhatsApp.

Getting started is easy, you just grab the Auth0 SDK for any platform that you need and you add a few lines of code to your project. Whether you're building a mobile app, a website, or an API, they all need authentication.

Sign up for Auth0, that's the number 0, and get a free plan or try the enterprise plan for 21 days at auth0.io/sedaily. That's A-U-T-H.io/sedaily. There's no credit card required, and Auth0 is trusted by developers at Atlassian and Mozilla and the Wall Street Journal and many other companies who use authentication.

Simplify your authentication today and try it out at A-U-T-H.io/sedaily. Stop struggling with authentication and get back to building your core features with Auth0.

[INTERVIEW]

[0:03:54.0] JM: Vasanth Asokan is an engineer at Netflix. Vasanth, welcome to Software Engineering Daily.

[0:03:59.4] VA: Thank you for having me here, Jeff.

[0:04:01.1] JM: You're on the show to discuss this article that you wrote, actually a couple articles, about Netflix's serverless-like platform. The Netflix API is accessed by developers who are building for over 1,000 device types; phones, TVs, virtual reality. You've got all these

different people who want to build frontend interfaces into the backend system that provides the contract for, “Here is a video. Go ahead and play that video.”

What are the requirements of that API between the person who’s writing a consumer, like a VR headset or a TV or a phone, the contract between that and the server that is delivering the video?

[0:04:54.1] VA: That’s a great question. I think it gives us opportunity to go into the background for how the Netflix API was and how it evolved. It started out being a traditional RESTful API. Given the tremendous growth Netflix experienced and its proliferation to a number of device types overtime, at one point the API team realized that a one size fits all model could no longer scale.

Even as developers, let’s be honest, we’ve seen a number of public APIs and we’ve written a code that consumes those APIs. When was the last time that we saw an API that was just perfect for our needs? There was always this little bit of information that you need on top of what it gave you. You had to make an extra call. The payload wasn’t quite formatted the way you wanted it. It was just not scaling in terms of the proliferation of client developers and device types.

Also, the other aspect is an API written to handle the needs of a UI experience on, let’s say, a powerful device that’s working in the US, working off of megabits per second connection, it simply does not provide the same ergonomics for, let’s say, a device that is underpowered and working in a part of the world that could be on, say, a kilobits per second connection.

Even if such an API could be designed, it’s a very slow innovation model. So both consumers have to go to the centralized API team and ask for a change when they want to change, and then the centralized team has to implement it, it has to make its way before everybody gets the benefits.

Netflix was optimized for innovation velocity. We have dozens of A-B tests going on simultaneously at any given time, and the centralized API model does not lend itself well towards each client developer optimizing the API experience.

Those were the requirements that sort of drove the API innovation within Netflix, and the central goal was how can we enable a wide array of different APIs to evolve and be active simultaneously? Today's Netflix API, it's quite unique born out of these requirements. It's an experience-based API. Consumers customize the API with the help of server-side code and that allows them to deliver the experience they need to their devices and customers.

[0:07:27.4] JM: "Developers are only responsible for the adaptor code that they write." That's a quote from your article. The ideal experience of a developer is that you want to just have an API that regardless of what system they're developing a client interface for, they can plug into Netflix's backend in a seamless fashion and the minimal amount of code has to be written on the frontend.

Explain what an adaptor is and what the experience of a developer who's writing an adaptor is.

[0:08:01.4] VA: Of course. The adaptor is the piece of code that runs server-side and it talks to the central API.

[0:08:08.9] JM: Sorry? It runs server-side or it runs client-side?

[0:08:11.1] VA: The adaptor is a piece of code that runs server-side.

[0:08:13.3] JM: I see.

[0:08:14.0] VA: Our entire platform is built as server-side infrastructure. The developers, our client developers, they're UI engineers typically, device software developers and they write code that executes on the device.

The most interesting thing about the platform is how can you let someone who's a client developer who's a UI engineer operate a piece of software that runs server-side with the least amount of effort and complexity? Those were actually core motivations for a platform. The adaptor that we refer to in the article is a piece of code that actually runs server-side. I'll get a little bit more into the details of what form these adaptors take and so on.

Ultimately, the goal of these adaptors is to take in a device request and do the necessary processing required to satisfy that request and send back a response that is custom built for the device that requested it.

[0:09:11.8] JM: You've got developers that are building against this Netflix API and the experience of building against that Netflix API is similar to the serverless function as a service platform. We've explored this on a bunch of different previous episodes, so if listeners are totally unfamiliar with the term serverless, they can check out one of those previous episodes. Let's assume they know what a serverless platform is.

Describe the analogy that you're drawing here between the Netflix API and the function as a service style platform.

[0:09:48.5] VA: The analogy is pretty high level. The goal of serverless platforms off the shelf, commercial serverless platforms, is to raise the abstraction levels to the point where one doesn't have to worry about physical hardware instances, provisioning them and maintaining them, and users write functions or pieces of code that gets scheduled on to instances that are managed and provided by the third part.

In the Netflix API model, developers kept the exact same abstraction load. They actually don't know anything about what hardware instances actually executed their code. They do almost no operations of any sort around the underlying instances. Do keep in mind the implementation, the Netflix implementation is bespoke, architecturally speaking.

Organizationally too, we are set up in a way that a central platform team operates the underlying infrastructure and takes away some of the concerns that you would still have to solve if you were using an artificial serverless platform. Even off the shelf platforms cannot and will not take away certain operational concerns. In that case, because we have a central team, some of those things are alleviated and taken care of. The analogy is very high level, but it's very close when it comes to just the developer experience of operating a serverless platform.

[0:11:16.3] JM: How do you define that term serverless?

[0:11:20.4] VA: The accepted and common definition of serverless out in the industry is infrastructure that is stateless. It's compute infrastructure. It is very ephemeral and typically built per use. The infrastructure that we refer to is fully managed by the third party and developers do not directly concern themselves with the management of the underlying hardware instances.

Sort of the implicit in this whole idea is that is efficient resource utilization, because when the serverless instances are not being used or maybe underutilized, the third party is free to sort of take out the backend instances, tear it down, reuse it for somebody else. There're sort of efficiencies of scale and efficient resource utilization, which allows you to sort of get lower cost per CPU cycle, so as to speak. That's the very accepted definition of serverless in the industry.

[0:12:21.3] JM: Agreed. What was the motivation for what you built with the serverless-like platform at Netflix? Explain exactly what you were building.

[0:12:33.7] VA: In our case, the traditional selling points for commercial serverless platforms, they were not actually our goals. We were not trying to optimize resource utilization or build client teams. All of our client teams are internal. Some of them are external, but we weren't trying to optimize their cost. What we were trying to do was raise the abstraction level.

Like I mentioned earlier, a lot of the engineers that are wanting to customize the APIs, their UI engineers, they write client software. They're not micro-service developers. They do not want to be on-call or they do not want to operate a piece of machinery that is running server-side in a stateful or a longer manner.

What we wanted to do was raise the abstraction level for them, and that was fundamentally the single biggest goal for why we came up with a platform like that.

[0:13:28.8] JM: You want to remove their responsibility to spin up a cloud instance and make sure that that cloud instance stays up, scale it up, scale it down. You wanted them to have more of a Twilio-like experience where they're making an API call and that's it, and the person who's responsible for the API management will take care of all the heavy lifting for all the people who are consuming this API.

[0:14:03.0] VA: That's absolutely right.

[0:14:04.9] JM: What are the advantages and disadvantages of the serverless style world that you wanted to move to?

[0:14:14.8] VA: Let's take advantages to start with. From the perspective of a developer, and in my opinion, abstraction and velocity are two big advantages for why you want to use a serverless solution. How can a developer or a development team spend more time focusing on their business goals and their business-oriented software and less time sort of on the nuts and bolts of the missionary?

For many use cases, the other benefit of serverless is the idea of efficient resource utilization and does cost savings and precise billing based on active use. If you're a startup with a great new idea, you are likely more interested in testing and quickly evolving your idea. Basically, development velocity and running lean rather than hiring a team that is very skilled at operating server-side infrastructure. That's one of the biggest advantages.

From the perspective of the serverless platform provider, it's really efficiencies of scale. Allowing yourself to treat your physical hardware as a sort of an elastic pool into which you do find brain scheduling of compute work, allows you to better utilize what you have. It opens your business up to a wider pool of developers who come to such a platform for the benefits that we previously stated. These are advantages for both the consumers as well as the providers of a serverless solution.

When we look at disadvantages, and if you are just focusing on operational disadvantages, at a high level the main one is that because you've raised the level of abstraction, you've also taken away some of the control. It isn't generally true for all things, but the implications of loss of control could be significant depending on your precise use case. This is not to say that a good balance cannot be reached. In my mind, it seems possible for serverless solutions to sort of follow the principles of progressive disclosure. That's an interaction design which implies you provide enough details for the user to get started with an experience, but if the user desires more control or more data, there is a way to get to it. Because you've built your solution up in

layers, it's possible to peel sort of the layers of the onion and go into a deeper level if you so wish.

I don't think it is possible for serverless platforms to do this and break open their abstraction if users want more control. Developers can then dial themselves up or down the abstraction chain depending on their particular need.

I was going to mention, one of the disadvantage, which we've seen in our experience, this isn't so much a disadvantage coming from the platform itself, but it's more around the design patterns that we saw for such serverless applications. Because serverless is such a natural fit for breaking up large monolithic applications into fine grain ones, what you typically see is like a dimensionality increase. What used to be one big application could be now 10 or 20 smaller ones, and what we have seen is that that increase can stress many of today's conventional workflows and manageability aspects, and so it's really a tradeoff. Would you rather spend more time managing hardware instances or do you want to take on new and possibly different operational tasks, because now you have like 20 of these things?

[SPONSOR MESSAGE]

[0:18:06.0] JM: Amazon Redshift powers the analytics of your business and intermix.io powers the analytics of your Redshift. Your dashboards are loading slowly, your queries are getting stuck, your business intelligence tools are chocking on data. The problem could be with how you are managing your Redshift cluster.

Intermix.io gives you the tools that you need to analyze your Amazon Redshift performance and improve the tool chain of everyone downstreamed from your data warehouse. The team at Intermix has seen so many Redshift clusters that they are confidence that they can solve whatever performance issues you are having.

Go to intermix.io/sedaily to get a 30-day free trial of Intermix. Intermix.io gives you performance analytics for Amazon Redshift. Intermix collects all your Redshift logs and makes it easy to figure what's wrong so that you can take action all in a nice intuitive dashboard.

The alternative is doing that yourself, running a bunch of scripts to get your diagnostic data and then figuring out how to visualize and manage it. What a nightmare and a waste of time.

Intermix is used by Postmates, Typeform, Udemy and other data teams who need insights into their Redshift cluster.

Go to intermix.io/sedaily to try out your free 30-day trial of Intermix and get your Redshift cluster under better analytics.

Thanks to Intermix for being a new sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[0:19:51.6] JM: Right. That seems like an interesting one to discuss, because if you've got a monolithic backend service that is going to serve the right video to whatever API adapter is requesting it, maybe it's a VR device, maybe it's a mobile phone in a low bandwidth environment. Let's say in the initial non-serverless world, you've got some big monolithic API and it just vends all those different types of video from its monolith and then you say, "Okay. In order to do this right we need to break it up into different microservices that will then — This one vends the low bandwidth version of the video. This one vends the mobile version of the video," and so on. That turns into a lot of different services that you're managing and it can get much more granular than that, then you start to get into a world where you have dependency management and you have dependencies of services upon each other. We often think about dependency management as a matter of single computer managing different packages, but this is a sense in which dependency management becomes also a distributed systems problem. You've got different services that depend on one another and they're getting pretty granular. It sounds like that leads to some operational burdens that are not trivial.

[0:21:22.5] VA: You're absolutely right. The way we think about it is you've taken a big picture and sort of broken it up into little jigsaw puzzle pieces, right? Operationally, you need to build back the big picture, and that's where I think the opportunities lies. I don't think this is an end state. I don't think this is how it is going to be going forward. It's an exciting opportunity for

providers to sort of reassemble compositionally the big picture post the fact, but still allow you to independently operate these fine granular bits when you need to.

[0:21:57.9] JM: What kinds of services makes sense to be decomposed into serverless-like services? Could you give me some — Because we discussed the API that vends different types of videos as a certain example. What are some other examples of things that seem like good candidates to be broken down for this platform that you've built?

[0:22:18.7] VA: Talking about serverless solutions outside Netflix, even processing is the other big use case, where this makes total sense. You have one little event which has to trigger some computation, some processing and maybe some state storage into a database somewhere. Use cases like that are like really good fits. Events tend to be bursty or varying in nature and scale overtime and serverless solution is a great fit for that.

If you're talking about the Netflix serverless platform, it was purely built to handle stateless request response handlers, route handlers. Also, given the nature of a micro-service-based service oriented architecture, to render a single response back to the client often involves making dozens and dozens of calls to lots of different backends and assembling all of it together and sending back one response to the client.

What we built in the Netflix platform is very much oriented towards this. How can we make it easy to compose these dozens of calls in an easy, resilient and performant manner? Those were kind of the main goals for what kinds of services we can operate on the platform.

[0:23:37.7] JM: That sounds almost like a workflow orchestration system that you were trying to build. You're trying to make it very easy to wire together scripts and run those scripts economically.

[0:23:50.5] VA: That is accurate, though I would say workflows sort of implies slightly more heavier processing than what ends up going on here. A workflow typically involves taking data from somewhere, doing some heavy lift; number crunching or processing the data and then saving the results.

In our case, the data itself tends to be pre-computered by various tiers all over Netflix. That could be a tier that has pre-computed your subscriber information. That could be a tier that has pre-computed sort of you're A-B test profile. It's really about getting these pieces of data and assembling them. I would say it's a more data composition type workflow rather than any heavy batch style workflow.

[0:24:40.8] JM: The way you described it in the article is the developer experience is to write scripts that contain application logic. What is an example of a script?

[0:24:51.4] VA: All of our scripts are primarily request response handlers. They take a particular device request, do preprocessing on the request, maybe you decorate it with some state about who's making the request. Then there's a whole bunch of downstream data collection involved, which then gets processed by the script, assembled into a payload and rendered as a response back to the device.

Other than that, this is what it sends back to the device, but other than that the script may also do some sort of data collection or publishing metrics which could be useful for business use cases. As a trivial example, like how often did a customer click on their bookmarks? There's some sort of metric collection and data processing that's going on.

[0:25:39.6] JM: We had this show recently about video infrastructure, and this show is all about how when you upload a video to YouTube, for example, often times that's going to be rendered into a bunch of different versions and it's transcoded into these different versions that will run at different bitrates depending on what the client who requested it. If the client is requesting it over a T1 connection and they're watching it on their smart TV, maybe you render it with the highest bitrate available. If they're requesting it on a smartphone in the middle of nowhere on low bandwidth, then you want the lowest bitrate quality video possible.

There is this huge combinatorial explosion of the different — Not huge, but it's a sizable combinatorial explosion of the different videos that can be requested here, and I just want to clarify what exactly we're talking about here, because we're talking about this API provider that provides for these different clients.

Is that what we're talking about? We're talking about like these different request types, these different clients who are requesting client-side experience or different video types or what exactly.

[0:27:02.4] VA: Oh, that's a really good question and probably useful clarification for the rest of our talk. We are actually not talking about the API request response handlers that actually send the bits of the video. Just like how you described for YouTube, Netflix does something very similar, and this is probably documented within our tech blog in great detail. All of the videos transcoded ahead of time for various profiles, device profiles, and the act of pressing play on a particular video gives you one particular stream for one of the Netflix CDNs.

That's a completely different request response, sort of that. The API platform that I'm referring to and the scripts that we are discussing here, they are what gets executed as you browse the Netflix UI. Starting from the time that you power on the Netflix app within your device, how the app gets all of the data it needs to boot itself up and then render a UI experience that is tailored for the customer that requested it and the device that requested it.

These are the API involved in giving back what we call the discover data, the data that allows you as a user to sort of discover the title that you want to watch up to the point that you press play. This is not to say that we are not involved after you press play. There is a whole bunch of scripts that are still involved, but they aren't actually sending back the bits of the video. They may be involved in, let's say, saving of bookmarks. Up to what point have you watched the movie? They may be involved in tracking back the customer quality of experience. What bitrate did we send? What was the actual UI performance as you used the various pieces of the UI? The search experience, and so on. The scripts that we are referring to here, they help power that experience.

[0:29:06.5] JM: All right. Are they doing things like getting recommendations and the custom feeds or simpler operations? You mentioned bookmarking something. I guess bookmarking something is not — How much more complex is a request like bookmarking than a request like gathering recommendations? Are both of those things fits for a script that would run on this serverless-like platform?

[0:29:32.8] VA: That's a good question. To answer that example specifically, both of them go through this API platform that we are discussing.

[0:29:40.9] JM: Okay.

[0:29:42.1] VA: That said, the nature of work tends to be very similar. Part of getting to understand why that is the case is that recommendations aren't done as the user is actually using the UI. A lot of things are pre-computed. They've already been pre-computed. It's more a data gathering exercise when you actually open up the app. Your recommendations are been pre-computed ahead of time. So it's not so much the script goes and actually computes the recommendation. It goes, fetches the pre-computed recommendations.

[0:30:14.5] JM: Of course.

[0:30:14.8] VA: If you think about it, then that isn't that different from a bookmark.

[0:30:19.0] JM: Right. That's just like a database query. It's very straightforward.

[0:30:22.5] VA: Exactly.

[0:30:23.7] JM: You mentioned that the serverless type of requests are great for bursty workloads. It seems like these types of requests would be more predictable. You know at any given time you've got people that are going to — People are going to log on. You're going to have people that are requesting bookmarks. You're going to have people that are requesting their recommendations. Why is it more economical to frame this as a serverless style platform than to just have an EC2 instance that's standing there able to do all the bookmarking? Explain the economics of moving to a serverless-like platform.

[0:31:09.9] VA: That's a good question. One that we kind of partly address in older question, our goals were actually not the economics. Our goals were really around the development —

[0:31:19.9] JM: That's right. The ergonomics.

[0:31:22.2] VA: And the ergonomics and the abstractions. Keep in mind, these are UI engineers. It's very hard for them to understand.

[0:31:28.6] JM: Oh, yes. Of course.

[0:31:29.7] VA: It's not that they're not skilled. Of course, anybody can pick up those skills. It's more a skillset; what are they good at and what would they actually like to focus on is the data gathering exercise, and this can actually get quite complicated.

On the surface, a Netflix UI experience may appear simple. If you look at sort of the data graph that is needed to render a single page, it's really complicated and goes through at least five or six different service tiers with a huge fan out on each one of those tiers.

It's really oriented towards solving that user case. Yeah, we weren't driven by the economics of it. We were more driven by the developer experience of it.

[0:32:13.7] JM: That said, as the person who is maintaining the backend for this, does your serverless request processing system, does it look like an AWS Lambda or does it just present a contract to the frontend engineer that feels like an AWS Lambda?

[0:32:37.1] VA: That's a really good question. Specifically, talking about our current iteration platform, it provides an experience that is not similar to AWS Lambda. The key thing about Lambda and other solutions is that server instances are lazily provisioned. Let's say particular script is not getting any requests. The first time a request arrives, an instance is actually spun up. There might be some cold start delay and so on and so forth. We don't have that. We have instances pre-provisioned and ready to take request.

We do auto scale them. We do scale them up and down during the day depending on the traffic profile. That said, it's very different from how the actual scheduling happens in Lambda. What we do present is the exact same contract that Lambda provides to the developers.

[0:33:29.1] JM: This script that we're talking about here, this is like a script that takes a request for a user's recommendations or takes a request for bookmarking a video to watch later and

then it hands back a response to the developer. How much code goes into that script? What is the experience of the frontend developer who is writing those scripts?

[0:34:00.6] VA: It varies. Some scripts we've seen are really lightweight, and literally a few [inaudible 0:34:06.3] that are chained together. Overtime, we have also noticed that some scripts have gotten quite big. They have code that expands easily a couple of dozen packages. Have deep inheritance hierarchies. If you look at the call graph within a particular indication or the stack trays, it can be [inaudible 0:34:24.7]. Some are complex enough to have dead logs and memory leaks and all that kind of stuff.

It's kind of an inside job, but some of them have become microservices in their own right. They're no longer tiny little scrubs, but really big, almost fat applications. That's actually speaking to one of the advantages of serverless, because it's optimized for developer velocity, the code can very quickly evolved. What started out as something very simple, in a matter of weeks or months, can quickly become a big piece of software. That's what the serverless platform gives you. You don't have to decide either ahead of time or on an ongoing basis, should I run this on an M3 to Excel? Should I upgrade myself to enforce? Users don't. The platform provisions and accounts for the actual resources that you consume.

[SPONSOR MESSAGE]

[0:35:29.7] JM: Dice helps you accelerate your tech career. Whether you're actively looking for a job of you need insights to grow in your current role, Dice has the resources that you need. Dice's mobile app is the fastest and easiest way to get ahead. Search thousands of tech jobs, from software engineering, to UI, to UX, to product management.

Discover your worth with Dice's salary predictor based on your unique skillset. Uncover new opportunities with Dice's new career-pathing tool, which can you give insights about the best types of roles to transition to and the skills that you'll need to get there.

Manage your tech career and download the Dice Careers App on android or iOS today. You can check out dice.com/sedaily and support Software Engineering Daily. That way you can find out

more about Dice and their products and their services by going to dice.com/sedaily. Thanks to Dice for being a continued sponsor, and let's get back to this episode.

[INTERVIEW CONTINUED]

[0:36:46.3] JM: I want to keep coming back to this high level example to make sure the listeners as well as myself understand what we're talking about here. Let's say Facebook comes out with a new device that is like smart glasses and it's got — You can watch Netflix on your glasses, and I'm the client developer that is responsible for creating the interface. I'm the Netflix developer that's responsible for creating the Netflix interface where I can watch things on my smart glasses. Is that a situation where I, as the client developer, I'm going to have to write a backend adaptor, because I'm going to have to write a specific way to request recommendations, for example. I'm going to have to use the new type of real estate on this new device in a creative new way, and because it's a creative new way, I'm going to need to write some backend logic to get the request going and I'm going to do that on this serverless-like API platform. Is that an accurate use case?

[0:37:51.7] VA: That isn't a very accurate use case, and one that illustrates quite a few variations. You already mentioned sort of screened real estate. It is not scalable to give you the exact same UI that you would get, let's say on what we call a 10-foot device, like your smart TV. There's a lot of real estate, you can see rows and rows of movies. There's just no way to put that on to a glass.

The Netflix client app that is developed for those glasses is not likely to make the exact same kind of request. What it is likely to do is maybe take the user's wise input and just play that title. It could be optimized for that. It could be optimized for a second screen experience. You're likely to have a smartphone and you have your glasses. Maybe you do some sort of discovery process on your smartphone. The actual playback happens on the glasses. You can see already quite a few variations to the use case, and that's why a single client app just doesn't work on all these different device profiles.

[0:38:58.3] JM: I love the development velocity there, because in a world without this backend adaptor system where the frontend developer is empowered to write their own custom

recommendation system call, the frontend developer would instead have to go over to the recommendations team and say, “Hey, can you modify the API and just like get this — Maybe then go version the API so I can get my smart glasses prototyped up and running,” and then you’ve got this communication barriers that you’ve got to go set up a meeting with that recommendations team and you’ve already got 30 meetings today. This is the kind of thing that slows down innovation, and this is one of the things that Netflix does so well that some people don’t really know about is this culture of continually turning the crank to get things innovating faster.

[0:39:54.5] VA: You’re absolutely correct. That is the single biggest motivation for this API platform, and it gets even more nuanced and hairy. We already talked about the glasses as being a device profile that’s very different. It’s not just going to the API team and getting a new API added or an API changed. What if you want to do a completely different sort of caching behavior? These are underpowered devices, maybe you want to cache the data longer compared to a TV.

Having this script adopter layer allows you to do quite a few things completely of your own, completely your own innovation. That would just not even be possible by going to a central API team.

[0:40:36.9] JM: Describe the deployment process. I’ve written my script to get the recommendations that can fit on to my glasses. I want to deploy that script so that I can quickly get up and running with my new glasses prototype system. How am I going to deploy that script and be able to call it?

[0:41:00.6] VA: The deployment experience starts with SDK that we provide to the developers. The software development contains various tools that allow them to both develop and deploy these scripts. The same tools can be used to query their deployed state, get inside, etc., etc.

Deployment itself is actually just a single command that they execute. In the case of our platform, it takes five minutes to execute after which the script is available instantly on all the API instances. It is possible to deploy it globally. Netflix is deployed in multiple AWS regions today. It is possible to deploy it globally within five minutes, or what we actually recommend the

developers do is go region by region in order to mitigate issues and catch issues earlier. They can also go to a particular region, and the deployment tools give you the controls to do either global or regional deployments.

[0:41:59.8] JM: Is there a CI pipeline for the scripts or anything like that? Do you need a CI pipeline?

[0:42:08.4] VA: Most definitely. There is a CI pipeline. We let the developers themselves design their own CI pipelines, and there's a good reason for this rather than just provide a canned one out of the box. Like we already spoke about, these serverless applications are kind of part of a bigger puzzle piece, and CI and deployments have to be part of larger workflows that include most often the client. The client teams are the best place to design their specific CI pipelines and optimize them.

In fact, the CI pipelines are very common and very successful, and they're so successful that we actually see sort of 10 times the deployment velocity in pre-prod than in prod. Literally, every [inaudible 0:42:54.2] gets a CI tested through this pipeline and deployed to various environments.

[0:43:01.3] JM: How much of Netflix's infrastructure could potentially go in this direction to the serverless-like system? Because you spent all these time building this platform and you've got the use case of — I guess that client API use case. I guess even that itself, that is such a general problem that you're solving. I guess that in and of itself would be worthwhile, but there's so many other things to Netflix. There are so many other components, backend, data, people who spend all their day doing data science, people who have these long-running analytics jobs, they've got Spark and Hadoop and they've got layers and layers of scheduling and infrastructure built around those offline data analytics. Is that at all related to what you're doing?

[0:43:56.9] VA: It's not. Definitely our API platform is very custom built for request response handling. It's not very suited for long-running bad scripts or analytics. It's also very sandboxed platform. Only specific dependencies are allowed. You cannot make network calls to any random thing out of the wild. The lifecycle is also very strictly controlled and you get access to

standard [inaudible 0:44:23.1]. You cannot, for instance, reboot yourself, which is something that certain use cases may need.

To answer your question, let's take a specific example. We use Cassandra as the data [inaudible 0:44:36.0] for a lot of Netflix applications. It would be impossible to run Cassandra on such a platform. There are so many other considerations that go into making what it does. There's a very specific topology to how the nodes are deployed and how they can discover and communicate with each other and so on. There's none of those provisions in our platform.

That said, request response use cases, definitely they can use such a platform. I would extend that a little further to say serverless is a really good fit for most of your typical microservices, things that are stateless.

[0:45:16.2] JM: The event sourcing example you've touched on a little bit earlier. I don't know if it's event sourcing or CQRS. I don't remember which one is which. The idea that you've got this event log and you want to be able to react to events by updating multiple data stores, like you want to update elastic search and you want to update your Mongo and several other materialized views in response to an event, and the thing to do that updating might be a serverless function. That's a great use case speaking about serverless.

Again, that is not really what we're talking about here. We're talking about just presenting a contract to the frontend developers that makes their life a lot easier. The recommendations and the bookmarking and these other things that a client-side developer wants to be able to impact server-side code with, they're going to want to compose these functions together. They're going to want to be able to build — Why don't you describe? What's the modularity and the composition story? What is the motivation for wanting to compose different services together for one of these API engineers?

[0:46:38.2] VA: Are you referring to the composition that is needed to give the final response to the device?

[0:46:43.8] JM: Yes.

[0:46:45.3] VA: Okay. The reason for composition is really an optimized UI experience for the client. Imagine, you bring out the app, all of those pieces of data come from very different systems. If you take any 10-foot device, for instance, there's a search widget at the top right and maybe your most recent searches have to be there ready for you to be able to click on it.

Likewise, you also get some sort of profile data. Out of my five different Netflix profiles, which am I currently on? I need to render that and maybe render a user thumbnail for the profile that's active. I also need to render all of the movies, and there's a certain ordering to them whereas the set of movies that I've added to my list, I want to see that upfront, and then there are all the recommendations that follow right below it.

You can see that even to render a single page, it goes to very different systems. That's where some of the composition needs arise and why a single script has to do the effective composition for one particular UI page literally.

[0:47:59.9] JM: Does this relate at all to Falcor? Because I know Falcor has been used at Netflix to do simplify the request response, kind of a scatter-gather approach to bringing a — What might otherwise take a bunch of different requests , you're able to do it in fewer requests.

[0:48:19.7] VA: It most certainly does. Falco is heavily used as an API layer within these scripts. There is a central sort of Falcor module that we published, and most of the times [inaudible 0:48:33.9] what they do is they take the Falcor library and then they implement a particular JSON graph for people that are familiar with Falcor, and the analogy is GraphQL from Facebook. There's a certain model that these users of these libraries build up using these core components. That data model is what is exposed in the response back to the device.

We definitely see a lot of these scripts use Falcor. Some of them don't. Transactional use cases, sometimes they prefer going with a lighter model where they directly do the underlying transactions themselves without using Falcor.

[0:49:19.1] JM: The process of managing dependencies can get complicated. You might need versioning on different versions of the API. Talk about that process of being able to manage different versions and allowing for modularity.

[0:49:38.8] VA: Definitely. Another element to how these scripts are authored today is that you can compose these scripts on top of each other. The piece of code that runs server-side, it can be made up of multiple bits, and then the bits can express dependencies on each other. The reason that we did this is in the earliest version of our platform, we only allowed sort of individual route handlers to be authored. This led to a lot of what we call copy pasta. A code is being copied from a previous implementation willy-nilly, taking it, applying it somewhere, making a slight change. You can imagine what sort of manageability this would have.

We did see a lot of desire for sharing code, lightweight pieces of code. That's why we, in the later version of the platform, we designed a first class sort of dependency system. There's also Conway's law. As teams grew and the Netflix UI needs grew, we saw teams being formed around some central functionality.

As a quick example is the Originals UI, Netflix is now very heavily into Originals. We have a very custom tailored UI experience around that. There's a central team that is totally in charge of understanding and optimizing that UI experience, and the code that they write is desired by many client device experiences.

What we enabled them to do is write a shared module, and then as producers of the shared module, teams can version and release updates to these shared modules.

The client developers, the consumers of the shared module then express dependencies on certain shared functionality. We have a versioning system that allows them to sort of express what startup binding they have with the shared module.

Ultimately, to round it up, we have a fully dynamic system whereby a shared module can be updated by the producer at any time. A consumer, depending on what sort of binding he has expressed on the shared module, can pick up that update. This has led to some very actually quite complex composition patterns, just purely server-side. We are not talking about the data composition, but just one piece of script. All of the bits that go into executing it, where does that come from? We actually have a fully dynamic system there where they can bind themselves to different common [inaudible 0:52:13.2].

[0:52:14.4] JM: We've been focusing on what you've built from the perspective of the developer that is leveraging it. Tell us about building this product for — it's almost like an internal product for developers yourself. I know we're running up against time a little bit, but give us an overview of how this works on the backend.

[0:52:38.9] VA: Definitely. There is a central platform theme that is operating the missionary upon which all of these scripts execute. It started around five years ago, and actually it was a little bit before my time at Netflix started. I did live through the bulk of the maturation and the innovation on the developer experience set of things.

The platform was actually born while we were in the midst of rapid growth. Like I mentioned earlier, the central API team was just unable to scale towards all of these growth and all of the use cases that came out of that growth. Likewise, there were also numerous operational issues and innovation caps with the traditional architecture.

What we did is build a custom implementation that listens to activity by developers where they are asking for a particular piece of code to be deployed. Our infrastructure takes — It maintains a registry of what route handlers are currently known, what new rote handlers device developers wants to publish. It goes to a backing store, which is the underlying scripts that execute that particular route handler, stores it within fully managed API instances.

We use class loading technology. They're all Java applications, so we use class loading technology to load these scripts into isolated class loaders. When a request arrives, there's a routing layer that understands which backing piece of code is best suited to handle a particular request and it redirects that via a method called and some rapid code that gets executed. A particular device as a request is then serviced.

The job of the platform team is to; A, operate these really complex API instances that are cohosting and multi-tenanting completely different disparate script code, and their goal is to run it in a way where it's safe, performant and resilient to various manner of issues.

[0:54:51.2] JM: Can you tell me some of the takeaways from building this serverless-like platform? What were the lessons that you learned while constructing it?

[0:55:00.2] VA: Our biggest lessons I would say came from sort of the development philosophy. There was general love for how easy it is to deploy using this platform, and the biggest surprise is the scale of growth and the rate at which this platform was being used. When I joined, and this was around four years ago, I remember there were maybe 100 scripts total, which were updated once a week. Some were updated maybe two or three times a week. Activity was just starting to ramp up.

Fast-forward to five years later, we have thousands of scripts, maybe tens of thousands if you include the pre-prod environments. Especially if we take the pre-prod environments, they get updated sometimes a hundred times a day in aggregate.

There's a lot of change. There's a lot of activity at this layer. To me, that's the single biggest insight. Once you make it easy, you really see how well it's actually taken up in terms of adoption and usage. In terms of the learnings, our blog post highlights quite a few of them, but it goes back to what we were speaking earlier. If you take a big piece of code and you break it up and you allow independent developers to author those fine grain bits, what you end up having is nobody has the complete big picture. Everybody is very effective and optimized towards operating their little piece of code, but issues tend to be cross-cutting. How do you know what is actually going on when it comes to the big picture?

To me, the learnings, I would say, are more towards how can you give them the benefits of operating things in a very fine grained fashion, but extend the abstraction of serverless towards solving a lot of the day-to-day operational use cases. How do you recompose sort of the bigger picture and not be burdened by having to manage dozens and dozens of these tiny little things.

[0:57:05.8] JM: Okay, Vasanth. It's been great talking to you about the Netflix serverless-like platform. I will put both of the links to your article sin the show notes, and I really appreciate you coming on. It's been a pleasure.

[0:57:18.4] VA: Thank you, Jeffrey. It's been wonderful getting to discuss our implementation and our learnings with you. Thank you.

[END OF INTERVIEW]

[0:57:26.7] JM: Spring framework gives developers an environment for building cloud native projects. On December 4th through 7th, SpringOne Platform is coming to San Francisco. SpringOne Platform is a conference where developers congregate to explore the latest technologies in the Spring ecosystem and beyond.

Speakers at SpringOne Platform include Eric Brewer, who created the CAP Theorem; Vaughn Vernon, who writes extensively about domain-driven design, and many thought leaders in the Spring ecosystem. SpringOne Platform is the premier conference for those who build, deploy and run cloud native software.

Software Engineering Daily listeners can sign up with the discount code sedaily100 and receive $100 off of a SpringOne Platform conference pass while also supporting Software Engineering Daily. I will also be at SpringOne reporting on developments in the cloud native ecosystem. I would love to see you there and have a discussion with you. Join me, December 4th through 7th at the SpringOne Platform conference and use discount code sedaily100 for $100 off of your conference pass. That's sedaily100, all one word, for the promo code.

Thanks to Pivotal for organizing SpringOne Platform and for sponsoring Software Engineering Daily.

[END]

