EPISODE 461

[INTRODUCTION]

[0:00:00.3] JM: Event-driven architecture. Each component of application logic emits events, which other parts of the application respond to. We've examined this pattern in previous shows that focus on pub-sub messaging, event sourcing and CQRS.

In today's show, we examine the intersection of event-driven architecture and serverless architecture. Serverless applications can be built by functions as a service like AWS Lambda, together with backend as a service tools like DynamoDB and Auth0.

Functions as a service give you cheap, flexible, scalable compute. Backend as a service tools give you robust, fault-tolerant tools for managing state. By combining these sets of tools, we can build applications without thinking about specific servers that are managing large portions of our application logic. This is great, because managing servers and doing load balancing and scaling is painful.

With this shift in architecture, we also have to change how data flows through our applications. Danilo Poccia is the author of *AWS Lambda in Action*, a book about building event-driven serverless applications. He also works at Amazon web services as a developer evangelist.

In his book, he discusses the connection between serverless architecture and event-driven architecture and we explore that here as well. We start by reviewing the evolution of the runtime unit, from physical machines to virtual machines, to containers, and now to functions as a service.

Then we dive into what it means for an application to be event-driven. We explore how to architect and scale a serverless architecture, and we finish by discussing the future of serverless, how IoT and edge computing and on-premise architectures will take advantage of this new technology.

[SPONSOR MESSAGE]

[0:02:05.8] 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 into 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 e-mail login like Slack, or phone login like WhatsApp.

Getting started is easy. You just grab the Auth0 STK 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/sedailly. 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 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]

[0:03:55.9] JM: Danilo Poccia is a technical evangelist at Amazon Web Services. He's also the author of *AWS Lambda in Action*. Danilo, welcome to Software Engineering Daily.

[0:04:05.6] DP: Hi, Jeff. Thank you for inviting me.

[0:04:07.6] JM: It's great to have you, because we are going through this transformation where cloud computing is going from the runtime unit, from the container becoming the runtime unit of serverless or functions as a service.

It seems like we've gone from virtual machines to containers, and now we're going to functions as a service. Describe this evolution as you see it. Why are we evolving in this direction?

[0:04:34.5] DP: Well, what I've seen is that a lot of us, they want to go faster. They want to gain more speed. I think this is what is behind this move. With virtual machines, everything you do in a matter of minutes. We've contain this in a matter of seconds. With functions, it's even faster. You enter into the milliseconds range.

This is really the – depends where you want to spend your time. What I see or interest. If you want to do the fine-tuning of the kernel of the operating system, then probably you need virtual machines. If you still want to manage a class server application components, then containers with a management platform they are great of course. But with functions, you can really focus on the code that you write and what you want to build.

[0:05:17.6] JM: With virtual machines and containers, we have a state-full execution environment. We've got memory, we've got a file system, it feels like classic computing. How does our thinking about applications need to change given the fact that we're going to serverless, and in serverless we have a lack of state, because we're not addressing an actual box, we're just throwing a function up into an anonymous blob of compute somewhere.

[0:05:45.1] DP: Yeah. Personally I think, stateless is good. It's a architectural pattern. It's something that I would suggest even to people that is working with different architecture. It's good for scaling and for availability, because it's much easier to manage the scaling and the availability of something that doesn't need to keep a state.

A state should be normally managed for a distributed architecture differently. Each service of for example a distributed architecture should keep its own state inside only if needed. Then you should take that since the architecture is distributed. Things are different than with the

monolithic application. You should think of things like eventual consistency, so data will slowly converge into a final state.

Also you should build all the interface that you'd assign if possible as either important, so that if the same data comes again then you get the same results. The importance and the eventual consistency are probably two ways to overcome this limitation.

[0:06:43.9] JM: When we're using serverless, that means that we don't need to care so much about load balancing or auto-scaling groups or capacity planning. That's one of the set of advantages that people have said in previous episodes that is really valuable about serverless.

When we're talking about functions as a service, it may be hard for some people to understand, because it's like a paradigm shift where it's not pure upside, it's not pure downside. It's just a different paradigm and you have different advantages and disadvantages. Help people understand, what are the upsides and the downsides of moving to serverless.

[0:07:23.4] DP: Well, the upsides I think as I said, you can follow on what's on the features that you want to build, not the boarding stuff. You don't pay for idle and you pay very little for small workloads. This is frustrating experimentations, so you can build a prototype and wait. If you're a startup, or even if you're a large company you can quickly build a prototype, test it and you don't have the impact, or the economic impact of building something if it doesn't work. Running experiments is really the only way to build something new, to innovate.

Also I think there's a big advantage in security, because security is part of what you build. It's forced to give permissions to Lambda functions. You have to think about permissions, and this is different from when you start creating something locally on your laptop.

[0:08:11.4] JM: As we have to make this paradigm shift, maybe you don't want to call downsides, but it seems like there are some shifts in thinking that we're going to need to make. Because we're not deploying our applications. You're not writing a big Node.js application and deploying it, the it just stays up and running for a long time. It's a different world. How does it contrast, what are the paradigm shifts that we need to make as we start to build in a world of functions as a service?

[0:08:42.3] DP: As you said, it's a new approach. You'll need to understand it so that you use probably a different mindset. It's a distributed architecture, so everything runs inside functions. You can control for example the order of delivery of messages, add some in some components. You have to think that the availability is managed by the platform, but you still need to program the logic flow, so what are the relevant events that should trigger your function.

I think the downside is really thinking in a different way. That's why for example, in my book I have a long tutorial spanning across multiple chapter of building a simple media sharing application. Where you create a user, you validate the user sending an e-mail to test the e-mail address. Then if the user is okay, you can start uploading pictures and so on. This is a tutorial that it's not really focusing only on the technical side of it, but really in thinking in the right way.

[0:09:42.2] JM: Yeah, it's like a different set of constraints. I think they're in many ways very healthy constraints. Sometimes constraints can really help you go in the right direction. You mentioned security advantages. The lack of servers means that we don't have to worry about our servers getting logged into and misappropriate it.

The serverless functions as a service and the other platform as a service, things that we might use to wire our application together, they have a very specific set of functionality. If our units of functionality have more narrow scope, they can't be told to do things that we don't want them to.

These are some security benefits. What are the other security implications of functions as a service?

[0:10:28.7] DP: Well, the operating system and the programming framework, so the runtime you want to use Node.js, Java or Python. They are taken care by the provider for by AWS, in the case of AWS Lambda, so you don't need to do the patching of the operating system for example. You only need to look at patching your code and the dependencies, the libraries, the modules that you bring with your code.

The narrow scope that you mentioned, the fact that each function is usually covering only a small narrow scope can help you embrace the principle of least privilege. This is a security principle that test that every module in your application should only have the minimum amount of privilege that they need to perform the action they're supposed to do.

For example, if you build a customer service application where in some way sometimes you need to refund money to your users, you don't need to give the permission to give the refund to everything that is running inside the customer service application, but only in a specific function there will be the refund money function.

[0:11:32.6] JM: Your book focuses on event-driven applications in much of the narrative. When we're talking about event-driven applications on this podcast, sometimes people don't really know what that means, like what is an event. Because most people are building these imperative applications, where you're making calls, explicit calls to a function and it feels like you're writing the script for what's going on.

An event-driven application feels like more of a responsive, reactive application. I want to help disambiguate what this means for people. What is an event?

[0:12:17.1] DP: If you look at the definition, an event is like something that happens or something that takes place. In the sense of computer application, an event is telling you – the list of events is telling you the history of what is – what happens. Like telling you the history of everything that happens inside your application.

For example, a file has been uploaded by your users or some part of the application as a database table changing some values. Each event by itself is interesting, because it's immutable, because it's telling you that something happened. It's not something that is going to change. In computer, in distributed application immutable data is always easier to manager than something where you can have concurrent access.

[0:13:01.8] JM: What does that translate to in terms of an application? What is an event-driven application?

[0:13:09.0] DP: It's an application where the business logic executes and responds to the events that are received. As you said, it's a reactive architecture, because you are reacting to an event. In some ways, bringing the reactive programming approach that normally we use in the frontend to build the user interfaces into the backend, I would say finally.

What I like to think is that the event is like the cause of something, and then the logic that is executed is the effect. If we start thinking of this cause-effect, so event and logic, they are the cause the effect, is much easier for our minds, for our brains to understand how an event-driven application work, because cause-effect is something that we are used to looking to anything since we are a child.

[0:13:55.7] JM: Tell me some of the implications of that change to a cause and effect driven application. Why is that useful for a serverless environment?

[0:14:06.8] DP: It's not useful – at single function level maybe, but it's useful when you start to design an architecture that's more complex than a single function, because you start to think, you look at the – for example, the data of your application you start thinking, "What happens if something happens, like a new file has been uploaded or a new user is inserted in the database, or a new item is inserted in the shopping cart of my commerce application?"

Then you start to think, "Okay, if this happens then I should do that." In this way, you start thinking of course in an event-driven architecture, but also you started to link the logic of your application with the flow of the data.

[SPONSOR MESSAGE]

[0:14:55.0] DP: This episode of Software Engineering Daily is sponsored by Datadog, a monitoring platform for dynamic infrastructure and applications. Datadog integrates seamlessly with more than 200 technologies including AWS Lambda. After just a few lines of instrumentation, you can get insights into your serverless applications with powerful dashboards and machine learning-based alerts.

Start a free trial today and Datadog will send you a free t-shirt. Visit softwareengineeringdaily.com/datadog to get started and get that free t-shirt. Again, that's softwareengineeringdaily.com/datadog.

[INTERVIEW CONTINUED]

[0:15:42.2] JM: There's a typical pattern for event-driven computation and you've talked about this in some your presentations that I've seen where for example, a frontend app like my mobile application – let's say my media sharing, or my media consumption application, I've got a video player application on my phone and it makes a call to a serverless function and the serverless function makes a call to a database like DynamoDB, and then that change to the database, like let's say I'm playing a video and so DynamoDB registers that I have just played a video and that changes to a database trigger – sorry, that change results in a database trigger, and that trigger leads to another serverless event.

Here we have a cause and effect – a series of cause and effect relationships. User requests a video, database updates, database causes an event, change to the database triggers another event, that event propagates to another serverless function and we can start to see how we can have a chain of cause and effect leading to broader functionality.

Maybe you could give a more full-fledged example, since you spend a lot of time talking about this, where we might want to use this pattern?

[0:17:07.0] DP: There's lots of use cases, a very basic one similar to what you described is when you create a new user, so you can have create a user function that rides the new user data on the database. It can come from a mobile application, or it can come from a web application. This triggers another function, so the fact that you insert the user data on the database, you can trigger another function that can verify the data, can for example depending on the content send an e-mail to verify the e-mail address or a personalized welcome e-mail based on the interests that you flagged when you signed up.

This is a very simple use case where you chain function, a database trigger and another function. A more interesting use cases I've seen is, for example in data processing, so maybe

you receive a stream of data that at a point can be processed as an event or grouped in micro-batches. After you insert the data point, for example it is coming from I don't know, an IoT sensor, you can have a function that just trigger it and is looking for the last 10 minutes of data points sent by the send sensor and can update some statistics that can give you the minimum, the maximum, the average or something like that on the data report that provide the sensor.

Moving forward, you can build even more interesting architecture that where the Lambda function that this trigger can look at the set of data and do anomaly detection. Understand for example if a sensor that is deployed in the field is working correctly or not.

[0:18:35.4] JM: Right. How would you contrast – maybe we can take one of those examples and contrast how the architecture might work with a monolithic or different deployment unit. Like if we were deploying to a virtual machine or a container, what would make that architecture more difficult to manage?

[0:18:58.0] DP: You mean, more difficult to manage in a serverless use case?

[0:19:01.4] JM: No, no. In the serverless use case, why is it advantageous? Why is it better that we're deploying to this cause and effect serverless composition world?

[0:19:12.0] DP: Because it's completely not relative to the architecture, because when you create a function, a serverless function, it just sits there. There's no way to call it if you don't link this function to an event. It can be an API call, it can be an update on the database as I said. You need to think of this trigger. This brings you to thinking in a different way.

[0:19:33.9] JM: Right. I guess, you don't have to stuff – Sorry, go ahead.

[0:19:37.6] DP: No, I was thinking that this is at single function level. If you start to have lots of functions, then it's good to design the overall architecture how all the flows goes, but you also need to have everything inside a single description of all your function, of all your event-driven architecture in a single repository, such as a single template file.

That's why there are tools that came from the open source like the serverless framework or terraform. As in AWS we released an open source, the serverless application model. It's an open source specification that extends and simplifies cloud formation for serverless use cases. You have a single template file that describes your functions how they correlate with all the repositories that you have, just like S3, DynamoDB or other databases. Then you can use this template with user's code to recreate the overall architecture.

[0:20:33.1] JM: Awesome. Yeah, you're describing how we start to manage this world, because people who are a little afraid of this might think, "Oh, we've got five or 10 different serverless functions that are all wired together, and these are replacing my monolith. I'm very comfortable with my monolith that has all of these functions in one place."

Talk more about how we manage all of these functions, how we compose them together and keep track of what's going on when we're moving out of the world where we've got all of our functions defined in a single node application that we can just deploy to a container and we can easily understand. How do we keep track of what's going on?

[0:21:20.1] DP: Well, from a particular point of view, you should have this central template that can help you keep track of what functions you're using, how they relate one to another and to the data repositories that you have.

You can use serverless framework, AWS, terraform. This is probably something that I would strongly suggest. Don't work creating by hand one function at a time, because you can lose track of what you're building. I think the sensor repository helps. Then you should have the right mindset, as we said at the beginning you should start thinking a different way.

For this reason, I created an open source tool that is available on my website. It's called Serverless by Design. The website is sbdserverlessbydesign.danilo.net. If you go on this website, you can really graphically design as an event-driven architecture, so you can create your repository, your object, your functions, then you can connect one repository to a function that is triggered if something is changed, for example in a database or you get an API call. The idea behind this tool is to really, because I want to give everybody a tool so that they can quickly prototype, quickly design and start thinking in this way.

[0:22:36.9] JM: It's a great tool. I've seen it. It's like the UML that we always wish we had. I think the problem with UML has always been that it's – you take a look at it at a UML diagram and it sometimes is confusing what's going on. You've got maybe triangles, or boxes or you've got a diagram that – you've got a little symbol that looks like a database and that's very easy to understand. But when you start to get into components with richer functionality, it's unclear what each UML diagram it's supposed to represent.

If you're doing it with – let's say your entire application is architected in AWS, AWS has those nice little symbols for each of the different services. If you're making a UML diagram that's basically AWS services wired together, it can be easier to understand what each of these module or units it because it's just an AWS service.

[0:23:42.7] DP: Yeah, it's typically more colorful than a UML diagram. Apart from that, I think when you create a user interface for developers, you need to think that developers they know what they want to build. If you start to put too many features, so if you reflect all the possibilities, all the configuration possibilities that you have in the user interface, then you make the user interface too complex to use and the developers start just using the text files as usual.

What I did is to get something very easy to use. I take a lot of assumptions when you create these diagrams so that I don't need to ask 10 questions every time you deploy something graphically on the architecture. I think that's why it's interesting. This is something that you should always think when you create a user interface for developers that it should be easier to use than the text files.

[0:24:34.0] JM: When I'm building a traditional monolithic application, I am oftentimes building that application out of libraries that already exist. I'm depending on certain libraries. You can end up with complex dependency graphs where I build one system that depends on another, and then I build another system on top of that that depends on the previous system. When we're breaking down our applications into serverless functions, how does the notion of dependencies and dependency graphs change?

[0:25:14.2] DP: At some level you should think similarly to what you would do with microservices, even if this is of course a little bit more extreme, more similar components, so the same guidelines work. Then inside each function, you can bring your own dependencies. I usually suggest to only bring the dependency you really need so that you can keep the function smaller.

But every function can bring use on libraries, can be binary so with AWS Lambda, for example you can binary. You have an example that is on the website that I created for my book where I use OpenCV, there is the open computer and vision library, it's a C++ library that I compiled statically and brought together with the Lambda function and then I use it through their Javascript and their Python and points. This is from a technical point of view.

Then from more an architectural point of view, what I did for example with the two that I created with serverless by design is really to map an architecture into a network model with nodes and edges, so a formal mathematical network. That was an idea that it was with me since last year, it was I think at Serverlessconf in London, and I had this idea that every event-driven architecture could be mapped one-to-one to a network, where the nodes are the functions and the data repositories that you have. Then the interaction between the notes can be averted graph, so averted network.

[0:26:38.4] JM: When we're architecting our applications, how do the backend and the frontend responsibilities look in an event-driven application? How can we start to think about architecting what we would traditionally call the frontend and the backend?

[0:26:56.2] DP: My suggestion is to create a standard interface between these two words, so that they can communicate through a stable interface. Usually you would put an API gateway, and API gateway can map API calls into the execution of one or more functions, and you can start using the API gateway with mock implementations and then you can start, so that the frontend team that can be working on – and not even mobile app or a web, or hybrid application can work on the mock implementation and start creating the client.

Then the backend team can implement the integration between the API gateway and the Lambda functions, for example one by one, so that you can implement the right functionality, and then you can integrate the two to have a test.

[0:27:42.7] JM: For people who haven't built these kinds of applications, explain in little more detail what an API gateway is and how you're interacting with it as a developer.

[0:27:51.7] DP: The API gateway is a place where you can design a web API usually a REST API, but you can even go beyond the full trust model. You give a web interface to the functions that you implement in the backend.

Normally you want to model a good API, you want to create a good API model because that's like a contract between the backend and the frontend and you don't want to change it if it's not really a requirement. You should spend some time and define the right API interface. For example, you can have a – you should look for what resources you need to manipulate from the frontend, or you probably want to create, I don't know, users.

Then these users can have a shopping cart, and then they can put items in the shopping cart. These will be the three resources that you model. Then on top of these resources, you should be at HTTP interface so that you can create a new user, update a user, delete a user, add content to a shopping cart and so on.

Usually with the REST model and you use the HTTP method to do so. So that like HTTP get – it's used to get a result. HTTP post and put are used to create a new object, a new resource or to what data, one that already exist. In this way you create an HTTP interface that is like a contract, then you can start implementing on the thought sides in parallel.

[0:29:13.8] JM: When you start with a API gateway that is the central routing point for all of your application logic, let's say you start a company and everything is managed in that single API gateway, but then your company gets huge and you realize, "Oh, I've got all these different departments and all these different –" if you start with the book-selling company and eventually you branch into all these other things. Do you create more API gateways and you have a central API gateway that routes to another API gateway?

[0:29:47.6] DP: One side of this is how you manage this as an infrastructure. That's something that depends on what you're using. But generally speaking, each micro-service that you want to build should have their own interface, probably their own endpoint.

That normally means that you can create one API gateway that is managing multiple services, or one API gateway depending on implementation can manage a single service. But the important thing is that each service has their endpoint, because each service should be in charge of managing their own interface, their own contract with the consumer of that API.

[0:30:25.3] JM: Okay. I see. Okay. After you hit this API gateway, different requests are going to get routed in different directions. Let's say some of the routing logic hits a – function as a service and the function as a service triggers some small amount of processing, and then it goes to a database. There is lots of different backend databases that we can use to give us some stateful functionality.

I mean, I think the way that we look at these applications, a lot of the ones that you talk about is you treat the function as a service for doing this ad hoc processing and then you have these really rich databases that we have today. These databases have really nice APIs, then the databases can generate events that can – right after you write something, it's like the database calls back and you continue this cause and effect relationship.

Describe how we think about these different databases. What are the different databases that we use for – or maybe databases to specific a word, you could almost say data store, because Redis for example is not exactly a database. It's a in-memory storage solution. Describe some of the different ways that we're managing state, because we're not managing that state in a stateful function application where we're just reserving the function as a service for this transient processing. What are the stateful data storage solutions that we're using for different parts of our application?

[0:32:02.4] DP: Normally, there is relational ward and this SQL ward where you can have multiple kind of databases such as graph databases, document-oriented. It's really not different from a traditional application. You should really find the best data model for your application.

Normally what I suggest is start to look how much your data is structured or unstructured. If you a very structured data set, then you can go more towards a more rigid database, even relational.

Or if it's completely unstructured, you can have text media content, binary content, then maybe you can go to an object store like Amazon S3, for example. Then you can also integrate non-standard database with the serverless ward. For example, and for any database you have access to the transaction log, you can very quickly create a plugin that can trigger a Lambda function for example, or a serverless function in general if you have something happening in a transactional log of a database.

You can really choose the database you want. Some of those are already integrated and the choice between relational or NoSQL is really depending on what you need to build. If you don't have lots of relational dependencies across multiple tables, then I would go for a NoSQL, like DynamoDB because it's much easier to scale.

[0:33:27.1] JM: The devices that are accessing our services, like my iPhone for example – my iPhone's got a reasonable amount of space, it's got memory, it's got essentially disk storage. How much state do you want to keep on client devices? Or does that story change at all when we're talking about a serverless environment?

[0:33:50.3] DP: I don't think it's really changing. I'm a great advocate of putting all the logic and all the data that is possible on the client on the frontend, and only use the backend when there is a compelling reason. Normally, there are like four main reasons why you need to put data or something in the backend.

Either you need to share data across different devices or a different users, then of course you need some point in the backend. You need to backup data, because you can trust the robustness of the device. Or you need security, for example if your application is – you have some payments in your application, you can't trust probably the client to take track if you gave money to someone else or not, you need like a centralized payment service in the backend or a distributed ledger, such as a blockchain to take trace of that.

The reason that I find is usually if you need more computation of storage capabilities of what's inside the device. Even if now, smartphone are much more powerful than they were a few years ago. You still can go beyond those limits, especially if you start to do fancy things with probably artificial intelligence and much learning.

Also this is very important for less capable devices, like IoT wearable. If you go in that space, then probably the devices that maybe needs to run on batteries, they don't have the computational capabilities or the storage capacity that you need. Then you need the backend for that.

[SPONSOR MESSAGE]

[0:35:26.6] JM: Do you have a product that is sold to software engineers? Are you looking to hire software engineers? Become a sponsor of Software Engineering Daily and support the show while getting your company into the ears of 24,000 developers around the world.

Developers listen to Software Engineering Daily to find out about the latest strategies and tools for building software. Send me an e-mail to find out more. Jeff@softwareengineeringdaily.com.

The sponsors of Software Engineering Daily make this show possible, and I have enjoyed advertising for some of the brands that I personally love using in my software projects. If you're curious about becoming a sponsor, send me an e-mail, or e-mail your marketing director and tell them that they should send me an e-mail; jeff@softwareengineeringdaily.com.

Thanks as always for listening and supporting the show. Let's get on with the show.

[INTERVIEW CONTINUED]

[0:36:28.3] JM: Yeah. What a great explanation of how to – those are great set of rules for how to decide what logic you should put on your client, what you should put on the server. We're going to need some notion of identity to do sessions for our serverless applications. How does identity management work in the world of serverless cause and effect event-driven applications?

[0:36:58.1] DP: it's in a way similar to what happens in microservices. Now in microservices, you still need to trace the identity and the authorization of the user across multiple services. On AWS, we have a service that can help you do that. It's called Amazon Cognito and you can use it to federate with other platforms, such as Twitter, Facebook, or with standard protocol such as SAML or OpenID Connect.

Or you can also manage your own user pool and the lifecycle of the users with Cognito. Or we can work with partners like Auth0 or – The idea is that you start getting an identity. This identity is mapped, you use it for authentication at first, then this authentication can be mapped at any component with different authorizations.

Then this can be managed depending on the service inside the logic of the function or in the service itself. We have, for example on AWS you can really fine-tune the level you access DynamoDB, DynamoDB table directly from a device, from a mobile application so that you can read or write without any server in between, because you can say only this user can access this table and this user can only access the table item – the items in this table, where user ID is in the hash key, so in the main key of the item.

It's quite complex. I'm not entering to too much detail, but yeah, it's that the services themselves can sometimes give you the feature to trace the identity and give permissions.

[0:38:31.6] JM: Okay. I don't want to go past your area of knowledge, but in that kind of world so like Cognito – is Cognito doing the session management and it's maintaining, "Okay, is this person's authorization token still fresh enough?"

[0:38:49.4] DP: Yeah, yeah. You can get JWT tokens from, for example, from within the user pools and they have a validity, and then they can give you the endpoint to manage their refresh of the session if you want. You can really build a session management platform and federate this with external identities if you want, or manage your own users.

[0:39:12.9] JM: Okay. That's cool. We've done a series of shows about companies that migrate to some sort of container management platform. Some of them use Kubernetes, some of them might use Amazon ECS, some of them might use Mesos.

But in any case, a lot of companies they migrated from a place where they were just using VMs, or they were just on real bare-metal servers to a place where they are managing things in containers and maybe they were managing them in the cloud, maybe they were managing them on premise.

Now, they're looking at this serverless world and they're thinking, "Well, I can save maybe – for some domains of my application 80%, 70%, 60%, 50%. I can save a ton of money by moving to serverless and I can help make my application easier to scale." What's the process of that migration? How is a company that has been building into this container world, or maybe they're still just on primary a monolithic situation. How do you begin to migrate your infrastructure to this serverless world with the appeal of the cost and scalability and all the stuff?

[0:40:34.6] DP: First of all, I would say that most customer that migrates to the cloud and specifically they want to use serverless is more to gain agility. It's not just the cost-saving. It's more the agility, the credibility to develop fast and especially to move what you develop in production faster.

When you have an architecture that you want to migrate to serverless, some of the initial thinking point are the same as when you want to migrate a monolith to microservices. You should start thinking what are the right boundaries where you want to peel the onion, like create an isolated component.

The idea is like for microservices, you should think really of business drivers, not of technical drivers, because business drivers would probably be the same across a longer amount of time, and you don't want to have your – the requirement change while you are developing something. You should look like, okay. But maybe the inventory, the root catalog, a user management, they should be separated by the overall application that I have.

Then technically speaking with a serverless architecture, you can start using the API gateway, so you can put an API gateway in front of everything. At the beginning it will be mostly a reverse proxy that can help you with advanced features such as throttling your authentication, but you can already put it in the front.

Then when you have a standard interface with the consumer of the application of the backend application; I'm talking about API, so you can start taking one component at a time. My suggestion is start at first with something easy to migrate, like something that hits maybe with low risk and low dependency, so that you can start understanding the tooling that you want to use, how that maps with your own internal process, because any company has different development pipelines that they want to build and use.

Then after you start being successful with your first component, then you can start to look for something that is maybe creating some problems right now that maybe these features that needs two years that you want to move in production, or you're not being able to and you can start looking into a more interesting use case for your next serverless experiment.

[0:42:54.3] JM: Indeed. Once we start to the serverless environment, how does our testing and debugging cycle work?

[0:43:03.7] DP: You can test locally if you want. There is lots of tools, and we released recently a tool that is called SAM Local. It's on Github that can help you to test locally your APIs, endpoints and your functions, for example.

I think that lots of people doesn't remember that we started to have a scale like a simplified testing environment, because in the past production the production environment was too expensive. But now with serverless, you can create a computer replica of the production environment with almost no cost, because of the test environment, unless you're doing performance test will be almost idle.

I think this is something that you should leverage. You can test local, but do unit testing also in the same environment where you would do the production. When I say the same environment, I don't mean really that must be the same production website, but can be a complete replica, but using the same core services.

Then when you need to go beyond unit testing and you want to do some integration test of how everything works, you can use synthetic transactions, something that can test how the data flows across all these components that you have.

My suggestion, and it's something that where I've seen lots of customer going is testing production if possible, even if it's sends crazy. But a look for a small changes so that every time you do something in production with a small change, do IB testing in production, so that maybe you do like a blue-green deployment where you keep the new deployment a new version for a small subset of your users and then you can get some metrics, even business metrics to understand if the new deployment is working, and then you release it to all your users.

This is something that we are also looking at and we also pre-announce some new features for AWS Lambda in the Serverlessconf that we did in New York City, that was in New York City in October.

[0:44:55.6] JM: I want to talk about more far-flung concepts. The current model of compute is that we interact with a frontend from our smartphone, or our laptop, and then the backend execution happens on servers in the cloud.

But we're inching towards a world where there's going to be so many more devices where we could have processing occur. We could have smart earbuds, we've got smart glasses, we've got a watch, we've got a drone outside our window, we've got a smart car on the street. Maybe our shirt has a processor in it.

All of these computers can execute code. How will that change the current that we have where we see the frontend and the backend as these binary well-defined areas of compute?

[0:45:48.4] DP: I think we're going into something different now. That's a lot of discussion around on this edge-computing term with not a clear definition often happens. My idea is that the logic is going to get distributed more and more outside of traditional data centers, so you would be able to execute logic closer to the users.

In web architecture, also the content delivery network, the way content is distributed to the user is something that is also an interesting – it can be interesting to understand how we can put logic closer to the users.

We are looking at we can have people run Lambda functions in this way. We have a project that it's called – it's actually a service. Now it's available in production. It's called AWS Greengrass. It's an extension of AWS IoT platform where you can deploy a Lambda function around them on devices. We support x86 or ARM devices like – Last week I was in Cambridge in an AWS user group and we were running Lambda function on RS Verify.

It was quite fun, but it's also useful because we work with very large customers here, like for example NL. It's a public utility that is presenting in Italy, in Spain and South America and Eastern Europe. They are planning to use this framework to run logic in smart gateways that can use both for consumer and the consumer space and also in the industrial space.

[0:47:18.6] JM: Okay. Help me understand why I want to do that, because I think of serverless as being useful, because – so I make a call to AWS Lambda, and what I'm actually doing is I'm making a call to Amazon's massive reservoir of services – sorry, of servers, and Amazon is scheduling my function as a service onto whatever blob of compute is available.

The reason it's so cheap, but so reliable is because Amazon has extra resources that it can schedule small blobs of compute on to. Why would I want to have AWS Lambda running on my IoT device, on my Raspberry Pie?

[0:48:05.2] DP: Because you want to go beyond to limits; one is the latency that is due to the speed of light, so it can go faster than that. If your device is somewhere where you have on high latency on the internet and you want to have a faster response time to what's the – maybe other things that are connected to this client device, or you want to take autonomous decision at very low latency.

Also, if you want to go beyond this couples with disconnection, so maybe you're building on healthcare architecture and if you have a hospital with thousands of sensors and this hospital

gets disconnected from the internet, you still want to be able to take local decisions very quickly based on the value that the sensors are reporting to you.

For example, AWS Greengrass has also embedded in some of our product such as AWS Snowball. The snowball is like a storage device that you can order from our console. You can fill it with your data and ship it back to AWS to import data or do the opposite if you want to export data.

The Oregon State University, they're literally using this to collect data in the middle of the ocean. They have sensors, they collect data in the middle of the ocean on a boat. Since now, they have Greengrass inside, the snowball edge, they can start running Lambda function that can preprocess this data. Since the preprocess data is much smaller, they can upload this data much quicker to the cloud and then send the raw data that is maybe in the order of the terabytes shift back with the device. It's a way to overcome the limits of latency and possible disconnection from the internet.

[0:49:46.2] JM: Cool. Does this help me do something like most – Common pattern. Common pattern today is let's say I am like a financial trading company and I've been doing trading since the early 90s, and some of my architecture that I deployed to on-prem stuff over the years has been augmented by the cloud. Eventually I saw the usefulness of the cloud and I started moving some of my architecture to the cloud, but I've still got all of this on-prem resources.

Deciding what processing to schedule onto my on-prem resources versus my cloud resources maybe is not straightforward. Does this Greengrass model, does that help me more easily do that, because I can just say, "I want to schedule my jobs onto whatever Amazon Lambda device available, whether that's on my own machines, or in the cloud?"

[0:50:45.6] DP: The advantage is that you have the same programming model, the same management interface as traditional Lambda functions, but then you can decide to deploy these functions on a device that is outside of the AWS data centers, the AWS regions.

To do that, you can do lots of different used cases and on just the two I mentioned with the hospital and the boat are so different from each other; that gives you an idea. But normally, the

two drivers that we see, our customers are trying to overcome are really latency. You want to do something at low latency, and this is especially important for some use cases.

Sometimes if you want to have the possibility to go – to continue to work even if you get disconnected from the internet; think of the IoT word for example, if you think of farming, like if you have sensors that you can use for defining, planning your irrigation system and stuff like that, you don't want to be dependent on who is providing the internet connection to you.

[0:51:49.8] JM: Okay. Let's talk a little bit about machine learning. How can we model machine learning pipelines in a way where we can utilize serverless architecture?

[0:52:00.6] DP: That's an interesting question. We are working with a few customers for that. The most [inaudible 0:52:04.5] approach now is to use traditional services on infrastructure to do the training of the model, where you can leverage for example GPU use, graphical processing units, or FBGAs to accelerate the training.

Then when you have the model that is trained, you can deploy the model inside a Lambda so we have customers doing that for example with tensorflow or with Apache MXNet. In this way, you can have the production side of your machine learning project when the consumer of the users of the model you train can get the benefit of the scalability and availability of tools like AWS Lambda and API gateway.

[0:52:46.4] JM: You touched on the benefits of moving to serverless in terms of how your development team might be able to change. You might be able to become more agile. How are development teams changing because of that serverless movement?

[0:53:02.2] DP: This is a great question, and it's something that I always – when I talk with a customer, try also to understand on their side and give my own suggestion. But it's something that where I have to say more on the learning side.

What I see is that with serverless, we have smaller teams that are more independent. There is more prototyping. If you have an idea, you can quickly build a prototype, a proof of concept and validate the idea or not. If it works, you really are on the right path to implement scalability,

availability, security, so it's much easier to move a prototype in production than from a traditional infrastructure.

This is probably the biggest change and it's something that we also have in [inaudible 0:53:44.8]. I know if you heard about the two-pizza team that we have, so the idea that a small team that are in charge of everything from the development to the deployment in production can give you the speed that you need when you start to grow.

It's better to have lots of smaller teams that a big team where if something bad happens, it's not clear who was in charge of that, and it's very easy to build internal dependencies that nobody is aware of.

[0:54:12.0] JM: Of course. All right, final question. You are doing lots of evangelism around serverless and you're talking to user groups, you're talking to customers, you're seeing the bleeding edge of how people are using this technology. What are you realizing about how people are using serverless and what are some of the cutting-edge use cases you're seeing? Give me a picture for what's on the cutting-edge of serverless and what are the kinds of changes to application development that we're going to see become popular in the next couple years?

[0:54:49.7] DP: It's always difficult to talk about the futures, because – but what I see is that the main reasons why the people is adapting serverless right now, like the speed that you needed then gives the possibility to build a prototype have very limited impact calls in your calls, unless you're going in large-scale production environment.

Those advantages would be there and are probably the one – the driver that is bringing people to adapt this new product of creating applications. We've seen customer create lots of different use cases, so from you know the Vogue Magazine, they created a photo vogue in Italy. It's a platform for photographers to exchange and promote their photographic work. It was faster in production, quicker to build and cheaper than what we are expecting.

Two, there are more conservative users like the driving license government agency in the UK, they are adapting an API-first approach and they are using the API gateway to split the onion, to

start to create standard interface, starter contract to interface different components and then modularize their architecture. It's really a value gate ward that we've seen.

[0:56:09.6] JM: All right. Well, Danilo thank you for making the time to come on Software Engineering Daily. It's been great talking to you. I recommend listeners check out your book *AWS Lambda in Action.*

Maybe we can do another show in a year, so when the world has turned on its head once again.

[0:56:26.8] DP: That would be awesome. Thank you.

[0:56:28.4] JM: All right. Great.

[END OF INTERVIEW]

[0:56:32.5] JM: Simplify continuous delivery with GoCD, the on-premise, open-source, continuous delivery tool by ThoughtWorks. With GoCD, you can easily model complex deployment workflows using pipelines and visualize them end to end with the value stream map. You get complete visibility into and control over your company's deployments.

At gocd.org/sedaily, find out how to bring continuous delivery to your teams. Say goodbye to deployment panic and hello to consistent predictable deliveries. Visit gocd.org/sedaily to learn more about GoCD. Commercial support and enterprise add-ons, including disaster recovery are available. Thanks to GoCD for being a continued sponsor of Software Engineering Daily.

[END]