EPISODE 473

[INTRODUCTION]

[0:00:00.8] JM: Functions as a service are deployable functions that run without an addressable server. Functions as a service scale without any work by the developer. When you deploy a function as a service to a cloud provider, the cloud provider will take care of running that function whenever it is called. You don't have to worry about spinning up a new machine and monitoring that machine and spinning the machine down once it becomes idle. You just tell the cloud provider that you want to run a function and the cloud provider executes it and returns the result.

Functions as a service can be more cost effective than running virtual machines or containerized infrastructure because you're letting the cloud provider decide where to schedule your function and you're giving the cloud provider flexibility on when to schedule the function. The developer experience for deploying a serverless function can feel mysterious. You send a blob of code into the cloud, later on you send a request to call that code in the cloud. The result of that execution of that code gets sent back down to you.

What is happening in between? Why am I able to execute this code despite not deploying it to a server? Rodric Rabbah is the principal researcher and technical lead in serverless computing at IBM. He helped design OpenWhisk, the open source functions as a service platform that IBM has deployed and operationalized as IBM cloud functions. Rodric joins the show to explain how to build a platform for functions as a service, otherwise known as serverless functions.

When a user deploys a function to OpenWhisk, that function gets stored in a database as a blob of text waiting to be called. When the user makes a call to that function, OpenWhisk takes it out of the database and queues the function in Kafka. Eventually that function gets scheduled on to a container for execution. Once the function has executed, OpenWhisk stores the result in a database and sends that result to the user. When you execute a function, the time spent scheduling that function as a service and loading it onto a container is known as the cold start problem.

The steps of executing a serverless function take time but the resource savings are significant. Your code is just stored as a blob of text in a database, rather than sitting in memory on a server, waiting to execute. In his research for building OpenWhisk, Rodric wrote about some of the tradeoffs for users who build applications with serverless functions and those tradeoffs exist along what Rodric calls the serverless trilemma.

In today's episode, we discuss why people are using function service. We discuss the architecture of OpenWhisk and we talk about the unsolved challenges of building a serverless platform. Full disclosure, IBM is a sponsor of Software Engineering Daily.

[SPONSOR MESSAGE]

[0:03:10.0] JM: Blockchains are a fundamental breakthrough in computer science and Consensus Academy is the place to learn about block chains. The Consensus Academy Developer Program is a free highly selective and carefully designed 10 week, online curriculum where you will immerse yourself in blockchain development and earn a Consensus Blockchain Certification.

By completing the program, you will be eligible for immediate hire by Consensus, a leader in the blockchain space, focused on the Etherium platform. If you want to learn about blockchains and become a developer for blockchain technology, check out the Consensus Academy by going to softwareengineeringdaily.com/blockchain.

The graduation ceremony is a spectacular, all expenses paid trip to Dubai where you will meet block chain developers working on real world solutions. Dubai has announced ambitious plans to become the first city to run on blockchains by 2020.

If you like the idea of immersing yourself in blockchain technology, graduating, and going to Dubai, checkout softwareengineeringdaily.com/blockchain. Build your new career on the blockchain with the Consensus Academy Developer Program. Applications are open from now until July 1st, so you want to apply soon.

For more details and to apply now, go to softwareengineeringdaily.com/blockchain. To learn more about Consensus and Etherium, please visit Consensus.net and sign up for the Consensus weekly news letter for everything you need to know about the blockchain space. Thanks to Consensus for being a sponsor of Software Engineering Daily.

[INTERVIEW]

[0:05:09.8] JM: Rodric Rabbah is a principle researcher and technical lead in serverless computing with IBM research. Rodric, welcome to Software Engineering Daily.

[0:05:17.7] RR: Thanks Jeff, I'm excited to do this.

[0:05:19.0] JM: In previous episodes about serverless, or functions as a service, we've talked about how the architecture of applications can change to take advantage of functions as a service and you work at IBM, you work on OpenWhisk, which is an open source version of a serverless platform. So you're well versed in this topic.

The application model that we've used for years involves not only functions but also objects and state and databases. We all want to take advantage of this new serverless idea where serverless idea where we can just have stateless computation that we deploy as a function to an opaque blob of compute in the cloud somewhere and if people aren't familiar with this, we've done lots of previous shows where you can get the basics on serverless.

So I'm going to assume that the listeners have some level of understanding about serverless architecture and what serverless actually means. What are the application architectures that are evolving to take advantage of these functions as a service?

[0:06:23.7] RR: Yeah, that's a great question. So function as a service when it really came out and as this evolved has proliferated this myth that functions are stateless. Where in fact, they're not really stateless. That's a fact of reality. When you write code today, when you're going to run it in the cloud, you're going to be interacting with API's, you're going to get your IO from somewhere as an external source. That inherently implies some notion of state. You're setting

up database connections, you're going to be reading objects from a database. You're going to be using other API's to enrich the functions and the capability that you're computing over.

So even though platform providers say functions should be stateless, it's really a myth and to do anything real, you end up as a developer having to involve a lot of state management within your function itself. That creates added complexity so that, while functions and service has been attractive, it's putting to burden on a developer to manage a lot of the state and number of the things that you eluded to.

Where I see a frontier in serverless computing really being pushed forward, and this is not just unique to IBM but I think in general for all the platform providers looking in the space, at about the same time is looking at the state management and how you manage state complexity for applications because to build interesting applications, you're going to not only have functions that have to interact with the outside world, but they have to persist some notion of state in terms of the computation that as it evolves over time.

You see this for where serverless is used today in terms of CICD, which is continuous integration and delivery, you see it in workflow orchestration, you see it conversation services and chat bots. There's a number of these use cases where you might have applications that you can build out a function as a service.

But you need a little bit more and where we're looking to take this from apache OpenWhisk and also its hosted version in the IBM cloud, called IBM cloud functions, is to deliver next tier of value proposition with respect to state management.

[0:08:24.5] JM: Let's dive in to that a little bit more. Why is state – why do people talk about state being hard to manage within serverless functions?

[0:08:35.2] RR: Sure. Yeah, if you look at how typically you compute in a traditional classic model, right? If you harken back to divine norm and classical model of computing, you don't just talk about the CPU, right? The classical model is there is a CPU and a memory. A lot of what you do in programming is really about bringing data into the CPU computing on it and storing it back to memory.

That model, while it has been very simple conceptually has enabled a lot of innovation and computing, serverless computing today talks about the compute resource and not about the memory. Our first task is to basically expand the scope of serverless to say, you're not only dealing with the compute resource, you also have to deal with the memory resource and in fact, I started calling this the new kind of computer is the cloud computer.

But, if you look at it in that way, there is a resource for compute and a resource for memory. I don't think it takes a lot of imagination to look at the complexity people have had to face in terms of the programming models that have evolved, the way you write programs, the way you optimize programs in the classical sense that now has to be transplanted and applied in this new cloud computer model. You know, in the imperative sense, you have constraints about or you have concerns about how you have to manage the memory.

When do you bring in the data from the memory to the processor to compute on it? How long do you keep it near the processors that you have optimized performance because you don't want to fetch things from memory always because that's an expensive operation. That's why memory architectures have developed with caching layers for example. But a lot of that complexity in nontraditional architecture is mostly taken care for you. Whether it's the programming language, run time, the architecture is doing a lot of that for you. Transplant that to the serverless model or the cloud computer. You don't have that; we're building the stack from the bottom up and we're building essentially with serverless functions and service, just to compute management.

That means a lot of complexity falls on you. So where's your data? You have to now manage that yourself. How long do you keep your database connections open? You have to manage that yourself. Are you co-locating your compute and your data? You have to manage that yourself, I think that gives you an idea of the kind of complexity that, you know, in the cloud native environment you're facing and why you have to shift a burden to the provider or sort of the provider of the platform itself.

Because these are going to be hard problems and with cloud computing, you have less and less visibility just because of how nascent the field is in terms of being able to do deep performance analysis, deep analysis about where your hot spots are with respect to the platform. So you just

don't even have the kind of visibility that you might when you're running code on your machine. I think that's part of the complexity and part of the challenges that we're facing.

[0:11:18.5] JM: Okay, so when I talked to the previous people who I've had on the show to talk about serverless, the way that we frame the discussion is, serverless is this thing where you've got companies who have giant data centers and the way that they have been doing their cloud computing, they've been offering servers as a service. You can provision a server on demand, it spins up a server and you've got a big blob of compute that you can do whatever you want with and you address it like a server.

The idea of serverless is on these big cloud computing clusters, they have left over space that people have not provisioned as addressable servers. So they can say, "Okay, well we've got this left over space and we're just going to aggregate all of that together as this opaque blob of compute that is not durable and we're just going to address functions to it and have those functions execute on that blob of compute."

Because it is this shifting blob of compute with variable SLA's, you're taking all kinds of different machines and just putting them together in this serverless blob of compute and you're scheduling that blob of compute out as functions. Because of that, because it has variable SLA's, you don't want to ascribe at any amount of durability, that's why people say, okay, well we don't really want to treat this estate full because to treat it as being stateful would be to say that this thing, if it fails, we're going to lose some component of state.

But you're saying that the statefulness is actually going to be inevitable, we have to figure out how to imbibe that blob of compute with a sense of state, am I encapsulating your arguments correctly?

[0:13:10.8] RR: Yeah, I think that's correct. If I could comment on a couple of things. Even when you run servers today, a lot of work loads tend to be idle for large portions of time. From a platform provider, you want to be able to reuse that capacity and actually use it for something else. That's one aspect of serverless computing with functions. I can aggregate all of the excess capacity as a provider that I have in my datacenter and now I make it available to others to run their workloads. So I shift some of the excess capacity and I can reuse it.

That's beneficial for me as a provider, but what it also does is it brings in this programming model, which is really rich empowering in that it frees you from a developer's perspective of having to manage the servers in the first place. Now you can shift the operations burden from managing the servers to managing your code. It's not that there are no ops, you're trading one burden for another. But when you do this and when you shift into this model, it's inherent in the kinds of applications that you're going to build that there is going to be some aspect of state.

If you don't take that into account and you don't make that first class, things that are attractive about this model will lose their value because it becomes expensive to, for example, reestablish database connections because your compute was afemoral and you've lost it. If you're trying to use this model for doing things like machine learning or inference in that context, you might be loading very large expensive models and if every time your function resources are deallocated, you have to bring in those resources again from a database or wherever you're loading your model from. That's an expensive operation. You're going to want to optimize that somehow.

Who's burden is that? Should that be your burden as a programmer and does the platform give you enough capability to be able to do that, or does the platform itself has to inherently give you some of that capability that says, "I'm aware of your state, I'm going to help you manage it," because it's easier for me to do it and just like I'm able to aggregate resources for a compute as a service, I can do the same kinds of things with memory as a service. Now I can take advantage of locality in the data center, I can take advantage of knowing what state you're persisting, what state you're not persisting.

You know, thick applications tend to mostly state list but there is some aspect of the program that if you take it, you can partition it where state is localized to very small pieces and because of that, then you can take advantage of that and optimize very specifically. This is not really a new observation. If you look at the history of computing with stream programming models, high performance computing, a lot of this has been played over and over again. It's just that now we have to transplant that context, and apply it in serverless.

[SPONSOR MESSAGE]

[0:15:56.6] JM: This episode of Software Engineering Daily is sponsored by Datadog. We know that monitoring can be a challenge. With so many services, apps and containers to track, it's harder than ever to understand application performance and to troubleshoot issues. Built by engineers for engineers, Datadog is a platform that's specifically design to provide full stack observability for modern applications. Datadog helps dev and ops teams easily see across all their servers, containers, apps and services to monitor performance and make a data driven decisions.

To get a free T-shirt and start your free trial of Datadog today visit softwareengineeringdaily.com/Datadog to get started. Datadog integrates seamlessly to gather metrics and events for more than 200 technologies including AWS, chef, Docker and redis. With built in dashboards, algorithmic alerts and end to end request tracing, Datadog helps teams monitor every layer of their stack in one place. But don't take our word for it, start a free trial today and data dog will send you a free T-shirt. Visit softwareengineeringdaily.com/Datadog to get started.

[INTERVIEW CONTINUED]

[0:17:22.7] JM: I want to jump ahead to talking about OpenWhisk because we've done a bunch of shows about serverless computing but we've not done shows about how to actually build a serverless platform. You just gave a great motivation for why companies that have a large datacenter with excess capacity would want to have a serverless component of their datacenter where the amount of resources that are in that serverless cluster are going up and down and external users are accessing that. Or maybe even internal users?

You know, members of the company who want to use resources opportunistically. Maybe they've got some batch jobs they want to run whenever there's enough compute and a serverless platform give them granularity for running that. I think that's one of the motivations for OpenWhisk, that's one of the motivations for having an open source platform for anybody to deploy their own serverless systems. That's what OpenWhisk is, it's a server side of a serverless platform. I want to get into talking about that. So let's say I define an action. OpenWhisk action is essentially a serverless function.

If I would just want this OpenWhisk action to just simply emit “Hello world”. I write it as a small java script application, I invoke it using OpenWhisk, if we were talking about other serverless platforms, I would have very little idea about what would occur behind the scenes. I would just deploy this code and it would execute and I would get a response. But since OpenWhisk is open source, you can tell me. So describe what happens end to end when I deploy that “hello world” java script application on OpenWhisk and execute it.

[0:19:09.7] RR: Sure, that’s a great point and you know, part of the open in OpenWhisk is that it’s basically open source. It’s free for you to use and deploy on your own, modify, et cetera. I think that’s part of an important platform because — I’ll answer your question in a second, but I wanted to make one additional point. Because we think this is early days for serverless and functions as a service, having an open ecosystem where we can push the frontier for this model of computing is also important.

As you pointed out, because it’s open, you can go and inspect the guts and see exactly what’s happening and from the time you create the action basically, you start typically by developing the function on your browser or your local file system, using your favorite IDE, and then you make an API call to OpenWhisk, or the hosted version of it like any IBM cloud functions for example that makes a copy of your function that then gets hosted on the server side so this is the open side of things.

That function is there, there’s no resources provision just because you created the function. It’s only at the time when you actually invoke the function that then something happens. That something essentially translates to the system having to – at the end, provision a container to inject your code into it and then send it any parameters, run the function inside that container and get the parameters out and also get the logs out.

What happens in that flow is that there’s essentially an API controller, which accepts all API requests, whether it’s traditional create or delete operations or what we call activations. That’s our terminology for running an act or invoking it. When the controller receives a request that says “invoke an action”, it posts internally a message over a message and the current implementation we use Kafka for that which is proven for durability and also latency of communicating messages across the control plane.

The controller poster request who – the message says, “I have a request, somebody would like to execute this function.” That message is picked up by one of invoker machines, which are slaves in the system that understand how to talk to containers and receive requests, essentially try to decode them as fast as possible, provision a resource, and run them. The flow is basically controller receives the message, it goes through a load balancer to the message bus, which essentially queues the message for a particular invoker to run the action. Eventually the invoker picks it up, runs it inside of a container and then gets the results, creates the activation record et cetera.

You can see this if you run open with locally, you can see it in the logs, we have very detailed logs which help you flow through the entire execution, you can also use that to measure how long you’re spending in any individual step and you can also observe that yourself if you’re trying to use OpenWhisk for running other kinds of actions which maybe we’ll get into a little bit later with respect to compositions.

Not just individual actions but compositions of actions together. There’s slightly different behavior that happens in that context which the system has to take care of. But again, because the implementation’s open, you can take a look and exactly see what happens in that context as well.

[0:22:20.2] JM: Yeah, I do want to talk about the composition a little bit later but about how OpenWhisk works as you just described, let’s go through it a little more slowly. This architecture was described quite clearly in an article called *Uncovering the Magic About Serverless* and I’ll put that in the show notes by a guy named Markus Thömmes, he describes it quite clearly as some diagrams, I found this really useful.

My function, I developed this function and I’m going to deploy it to a serverless cluster, it’s just a hello world function and the first step is it h its engine X, which is a reverse proxy and load balancer and then it hits the controller and the controller is written in Scala, it’s going to do some orchestration between different components of this serverless framework.

Describe what the controller does.

[0:23:15.4] RR: The controller has largely two pieces, one is handling all the API request and the other is a load balancer, which is going to make a decision about where to run particular activations and let me explain that in a little bit later.

For the first aspect, which is just handling API requests, for a CRUD operations, which are create read update delete of actions or other kinds of assets that open with support including packages, triggers and rules. It interfaces to the database so that it's creating its record for the functions that you want to run or the meta data that says, "I have an event." When that event happens, create a trigger and a rule that causes an action to run.

The second aspect is the load balancer, which is given a request to run a particular action, which might come in by directly trying to invoke an action or because you receive a trigger that says, "When this trigger happens, execute the following action." It has to make a decision about where to run this particular action. The reason there's a load balancer in the controller is because we are very sensitive to latency and because spinning up a container for us to run the action in downstream is expensive. I mean, it can take somewhere from 500 milliseconds or even a second or more.

You want to be able to take advantage of resources you're already allocated to run that particular function for that particular user. So if I have a container that I've provisioned in the past, the load balancer is aware of that and wants to write that request to that same container. The controller is doing both aspects, it's handling all the API requests and it's also trying to do the load balancing which is a way of maximizing utilization in a system and improving the locality of where we run individual actions.

[0:24:55.8] JM: This is – what other serverless platforms might call an API gateway basically? It scales up and down, depending on how many people are requesting actions from the serverless platform?

[0:25:12.2] RR: The API itself, the controller itself is a finite entity in terms of its maximum capacity it can handle. If the controller needs, if the system detects that the controller would need access capacity then that has the provision itself, next controller. Typically you might

deploy many controllers behind the same engine X and then have some load balancing that engine X is doing for you.

It would be an interesting proposition to say, can you implement the controller itself as essentially a serverless system because you want that kind of elasticity you eluded to. Because it's the controller in a typical deployment would be heavily used and essentially up all the time servicing requests, you essentially cross over the point where it makes sense to run something as a function as a service versus a dedicated resource. So when we deploy the controller as OpenWhisk and even IBM cloud functions, we actually deploy it as a container and it's managed as a single monolithic microservice.

[0:26:08.4] JM: Yeah, the next step is the controller takes the function and hands it off to CouchDB. If I pass my hello world application, if I send a request at my hello world application to OpenWhisk, the controller is going to hand that to CouchDB, the entire function gets stored. I guess this was something that was kind of like surprising to me. But it was just like, you just store the function in a database and then when somebody calls it, you take it out of the database and you're eventually going to load it into a Docker container.

Why CouchDB? CouchDB is this system of record that stores all of the functions that people are going to invoke against OpenWhisk? Why do you use CouchDB?

[0:26:50.5] RR: Right. Some of this is a historical and that when we started OpenWhisk, we weren't quite sure how the eventual schema was going to end up and one of the advantage of NoSQL is you don't have the CouchDB as an example of NoSQL database. You don't have to lock into a particular kind of schema and then as we were evolving the implementation very rapidly, it sort of freed us from having to manage a large aspect of that and worry about schema migrations and things of that sort.

So it fit in that model. It also allowed us to have indexing functions that we could push against the database so we co-locate the system of records and the actions themselves with map and reduce functions that we wanted to apply against those functions to compute various kinds of indices.

So the kind of meta data that we typically store inside of CouchDB is small enough and we wanted additional flexibility with respect to the schema to where it fits. Now, you can argue for other meta data like the triggers and rules, which are really relational by definition that we should have used a different data store for that and I think that's something that's within the interest of the community where we'd like to be able to plug in different databases that we use for different entity management.

The nice thing is that because OpenWhisk is part of — the code is on GitHub, we've had people ask these kinds of questions, "Can I plug in my own database?" We had factored the interfaces so that you would be able to bring in your own implementation of a database and use it or swap one out for the other. The details matter and it might not be as easy as I make it sound but, you know, initially, it made sense to use NoSQL database plus an object store, ultimately for storing the large code blobs.

The way I look at it really is you want a mirror of your local file system in the cloud, you know, whether you use Couch or S3 or object store, that's really what you're trying to do. I want to mirror my local development environment in the cloud. It's the way I organize the files, the way I organize my projects, et cetera. Then, when you want to run, I want to be able to figure out what you're going to run and then bring it into the resources of revision to run that code.

CouchDB is an implementation detail, it has some advantages for us today but it's also a very interesting question about what – now that OpenWhisk has matured and in terms of our operational experience of running OpenWhisk in the IBM cloud, starting over from scratch, might we use different strategies and might we use different data stores? I think that that's a valid question, and it makes sense four different kinds of entities that we manage.

[0:29:25.8] JM: So after I've gotten my function stored in CouchDB, that means that whenever I want to call that function or whenever I want to call my hello world function, I can again just ping the OpenWhisk platform and my request is going to get load balanced across one of these controllers and it's going to fetch that function from the database because I stored it there earlier, and it's going to schedule it on this fleet of Docker containers that are called invokers and before it gets scheduled on to those Docker containers, it gets queued up in Kafka.

Why do you need this buffer, this Kafka buffer for scheduling functions against this fleet of invokers?

[0:30:14.1] RR: Sure. The main reason is reliability. You know, you're in a distributed system, that works can partition, system components might fail and you want to essentially try to provide a notion of some reliability to the end users.

Once we've accepted a request, we try to move it into basically a more persistent data store or medium and we use Kafka for that. We defer all the reliability to Kafka and now once the request is in Kafka, if the controller goes down and, you know, it comes back at some point in the future or if all of the invokers in the system go down and then come back in the future. They can pick up from where they started. It's providing a notion of reliability and, you know, communication and distributor system is now zero latency, you need some mechanism for queueing, especially as load increases.

Sometimes you might not have enough capacity in the system. What do you do? You have to be able to handle back pressure somewhere while you either charge provision new resources or wait for capacity to be available. So Kafka gives us not only the reliability then but also the queueing aspect to handle more load where we might not have enough capacity in the system.

You might be indirectly touching on an interesting property of serverless, which is servers level agreements. Typically when you talk about service level agreements, you know, you say, my up time is 99.9% of the time. You know, I have three nine's, a four nine's or five nine's. But in serverless, I think that's not as meaningful and it's a little bit more challenging to come up with an SLA for serverless.

I think the kind of guarantee you want is, how much or what percentage at a time are you going to process my request within say, one millisecond, right? Because if our uptime is a hundred percent but we queue your request for minutes at a time for a function that might execute for a hundred milliseconds, that's not an interesting SLA. Rather, we want to say, with four nine's we can start executing your function within one millisecond. Now, that's the kind of property that I think you need to build interesting applications, especially when you might be latency sensitive.

[0:32:17.5] JM: All right, after my functioning gets buffered up in Kafka, eventually it's going to get scheduled against these fleet of invokers, these Docker containers. Is there anything special going on at the layer of invokers? Like what's going on there? Is it — Do you use a scheduler like Kubernetes or Mesos or did you write your own scheduler?

[0:32:40.1] RR: Right, we wrote our own scheduler and the reason for that is latency again. If you try to look at what's happening in the invoker, when it receives a request, we wrote our own container pool and management because we needed to be able to take advantage of locality and we also wanted to be able to provide sort of continuing to resources with as little latency as possible.

Typically, if you try to spin up a new container for every function that you're going to run, you know, as I eluded to earlier, a Docker run can take anywhere from 500 milliseconds and up and under load, that time goes up. I've seen times over a second for spinning up a new container. You know, for Kubernetes, I've seen some numbers that say they can spin up a pod in sub seconds, but the latency for that is you know, I think in general doesn't work.

What we do is — what the invoker is doing is that it try to create as few containers as possible for individual functions that it's seen in the past. It wants to cash containers that it's used in the past and reuse them. The way we do that is by pausing and unpausing containers. If you think of the lifecycle management of a container, it goes through a Docker run, you run the function and then there's a Docker pause immediately after your function has returned. We do that because we don't want the functions to essentially continue executing, it has to do also with billing and metering.

Once your container is paused, it's essentially not consuming resources, we can keep it around for a while and if we get a new request to run that same function from the same user again. We can just unpause that container and then start executing again. The Docker pause and unpause, we actually use something lower than Docker, we use Run C and the latency for doing that is on the order of 10 milliseconds. Whereas a Docker run, let's say on average is 500 milliseconds.

So it's a very big performance gap and this is also part of the reason why we don't use Kubernetes or Mesos or a container management system at that level. Because the way those systems are architected, I think today, they're not quite there for being able to do that kind of low latency operations for functions as a service.

That said, there's a lot of effort in the space and a lot of interest in being able to reuse some of these course container orchestration systems for that purpose. So I would expect that in the future, things will change. But until they do, we've had to sort of fill that gap and that's part of why the invoker, essentially is a custom bespoke container management system that we built as part of OpenWhisk.

[SPONSOR MESSAGE]

[0:35:17.7] JM: Indeed Prime flips the typical model of job search and makes it easy to apply to multiple jobs and get multiple offers. Indeed Prime, simplifies your job search and helps you land that ideal software engineering position from companies like Facebook or Uber or Dropbox. Candidates get immediate exposure to top companies with just one simple application to indeed Prime. The companies on Prime's exclusive platform, message the candidates with salary and equity upfront. Indeed Prime is 100% free for candidates. There are no strings attached.

Sign up now and help support Software Engineering Daily by going to indeed.com/sedaily, if you're looking for a job and want a simpler job search experience. You can also put money in your pocket by referring your friends and colleagues. Refer Software Engineering to the platform and get $200 when they get contacted by a company and $2,000 when they accept a job through Prime. You can learn more about this at indeed.com/prime/referral for the Indeed referral program.

Thanks to Indeed for being a continued sponsor of Software Engineering Daily. If I ever leave the podcasting world and need to find a job once again, Indeed Prime will be my first stop

[INTERVIEW CONTINUED]

[0:36:58.1] JM: Yeah so one of the parts of the serverless that is important to recognize is that when I request my function to be called, when I request my hello world function to be called it has to get scheduled and deployed to a container and this leads to what is called the cold start problem, because you have to spend the time spinning up an actual container. Contrast that with the pre-serverless world where you've got a container or a server that is already sitting there and all you have to do is send an HTTP request to it and you get a response and it's a low latency.

This is one of the tradeoffs of the serverless world is because the tradeoff is you actually have to – you're going to have to have somebody spinning up this opaque blob of compute and you have the cold start problem and you're talking about, okay we are going to have this cold start problem but if somebody is going to call hello world one time, the cold start problem okay kind of an avoidable. We just have it.

But if they are going to call hello world and then a minute later they are going to call again and then three seconds later they are going to call it again and they're going to call it continuously for three hours and then they are going to stop calling it, you would much rather spin up a container and keep that container up so that you can serve more hello world requests.

So help me understand to what degree does the user tune how much they want their container to stay around? Do I have to tell OpenWhisk, "Hey I am calling hello world and I'm going to call it again for the next three hours pretty frequently. So keep my container going." Or is OpenWhisk does it intelligently figure out that it's going to be called frequently?

[0:38:54.3] RR: Right, so I think that is a great question. So OpenWhisk today and the way it works, when you run the function as you noted it will spin up a new container. Now it behooves this to keep that container around for as long as possible in the pause state because then when we need that resource again, we can just un-pause it. The load balancer knows where that container is and tries to go to it. So we get some of the effect that way.

Another aspect for the system in trying to mitigate this cold start latency is trying to keep what we call stem cell containers around for a bit. A stem cell container is essentially a container that is ready to run. We don't have to do a Docker run on it, which saves us 500 milliseconds or

more but doesn't have any user code initialize or specializing it. So you pay the cost of loading the user code into it, which can be expensive but if you are loading a large zip file for example. But in most cases, is in the order of a few milliseconds.

So that's another aspect of the capabilities of a system to try to mitigate code latencies but that is largely where things stopped today and as an end user we have seen and we get this question a lot, "How can I keep my container around and how long will my container stay around" So there are tricks and they are largely just tricks or hacks that people do today to keep their containers around by constantly pinging them.

So you can ping them by firing an alarm, which generates a trigger that then causes the action to run. The action says, "Oh I'm just taking a warm up trigger, do nothing." So that keeps the container alive in the system but that is a silly way of doing it. One, it's putting undue burden on the programmer to do this and two, now we're using resources to wake up a container that essentially does nothing. So it's the kind of capability that you want to shift to the platform that either tries to do a predictive analysis or tries to give the user some capability that says, "Here's a distribution of my load and try to use that to keep my containers around as much as possible."

This is I think a complicated problem and you know there is probably good queuing theory models to explore this a bit. In some ways, it also goes back to my server's level agreement kind of model. You know I might have finite resources, so I don't want to keep your resources around if you are not really using them. But if you are an important customer or you are very latency sensitive say chat bot or a conversation service, hey a three second latency spin up a container is basically dead in the water. That kills the user experience.

So there is work that has to happen in a platform to make the latency very low, to deliver that user experience that you would get with a dedicated resource and I think that is part of the challenge and the next frontier in computing in this model and all is not loss in that at least with OpenWhisk. We have the notion of sequences, which are actions that are going to run in essentially a pipeline manner, a producer-consumer manner. We can use that. We don't do this today just to be clear, but we could use that as a way of predicting what actions you are going to run because we essentially have a history.

When you start executing action, a particular action, we know you're going to also execute these other actions. So we'll have to go ahead and provision resources to cut the latency and that is a form of speculation that the platform could do. So I'd like to see a model where the platform is doing more of this rather than the end user doing this but there is also a hybrid model and history tells us the hybrid probably works well where the user might hint at what their load looks like and the platform uses that in combination with it speculation techniques and it's prediction techniques to mitigate the cold start latencies.

Does that answer the question of where you are trying to get at?

[0:42:39.7] JM: Yes, it does and I want to talk about some of the other economical problems with serverless and especially some of the stuff that you've addressed in your research. We've been talking about how OpenWhisk as a specific serverless platform is architected. Now I'd like to talk about the usage of that kind of platform. We have done a bunch of shows about event driven architectures and how event driven architecture relates to serverless applications because you have these events and one event triggers a serverless function and then the serverless function might cause another event trigger and so you get this cause and effect chain of different serverless functions being applied to one another and that's how you get – you start to develop rich functionality out of these fairly narrow functions as a service.

It's almost like each function is a service can be the size of a helper method and you have these helper method function as a services that you want to tie together to create rich functionality and if I understand your literature correctly, this is what you call composition. You want to compose different serverless functions together to create richer functionalities. Is that what composition is?

[0:44:01.9] RR: Yeah, absolutely. It's saying that functions are the building blocks and applications are made up of many functions that have to communicate and orchestrate a data flow between them and composition is essentially the mechanism and the programming approach that we have taken to enable this kind of model on top of functions as service and serverless computing.

[0:44:24.0] JM: Now in a monolithic application, I've deployed my giant application to a container or a server somewhere and the upside of that is that all of these different methods are sitting in the same machine. So if I've got a method, if I hit an API end point and that API end point calls a method, which calls another method, which calls another method, it's not really a big deal. All of these methods are sitting on the same machine.

So it's just a single machine and it's unlikely to have distributed systems style problems. But when we break those methods into their own functions and you get these different functions and different helper methods that are sitting in different machines, this becomes a distributed systems problem. All of the latency issues that we articulated with the cold start problem, these become a real issue because if you've got – like if I need to call one API end point that hits a helper function as a service that needs to have a cold start, which hits another function as a service that needs to have a cold start, which to another function of a service that needs to have a cold start, this is going to compound the latency which can be a real issue.

Am I articulating one of the problems that you wanted to tackle with your studies of composition?

[0:45:45.0] RR: Absolutely. I mean, this goes back to a lot of the points that you're right on with and there's two things that play here. One is this programming model where you try to bring up — you try to deconstruct your large models like application into functions and then this driver applications of composition functions is in some ways good software engineering but if you look at it from the context of parallel programming, or distributed programming it's like the holy grail in that serverless is forcing you, by construction, you must decompose your application to small functions.

But if you take that to the extreme, then you are going to pay very heavy cost with respect to performance, given where the systems are today. So you have to try to recoup some of the monolithic style of the application, but without trying to compromise essentially the good software engineering principles that you might have applied by decomposing code into individual functions. Maybe you wrote different functions and different languages because it just suites the problem that you are trying to solve better and so with Composer, which is something that we have released either of IBM research as a technology preview, it works with

OpenWhisk. We like to get users to use it to give us feedback and evaluate how we're doing in the space.

But part of Composer is to try to recover this aspect of the monolith by saying, the applications might be individual functions at a cloud functions or inline functions that you write in place. But we wanted to come down to representation of that composition in a way that then we can build tooling on top of it like compilers that do optimizations on that composition. So an example that fits in the context of what we are talking about here is fusion. I have a function that then talks to another function that then talks to another function rather than running them as three separate containers as part of OpenWhisk we fuse them together into now a single larger function and we run it in one single container.

The advantage of that now is that you don't have the three cold start. You might have only one. The communication between the functions now doesn't have to go through the system or the system of records. It can be very local, and this is an example not only of the state management you want to get at but also the kind of compiler optimizations you might want because there's performance advantages to it. Informal experiments that we've measured, you can get 30X performance gains for even simple functions that you fuse in this way.

So the idea is really to with Composer is to expose the functions themselves as the individual building blocks, expose the communication between the functions. That's the data flow between them, and third is expose to state that individual functions might want to persist across activations and with these three primitives sort of exposes building blocks. Then I imagine how we can build tooling on top of it to do a compiler optimizations to do automatic state management to do a runtime optimization and this is really the frontier that we are pushing from my group in IBM research.

[0:48:47.8] JM: Can you give an example of when a developer would want to use fusion?

[0:48:53.3] RR: So the idea is not necessarily for the developer to say, "Hey I want to fuse two functions together," but rather the developer says, "I have two functions. I wrote them in a way that is natural to me." It's the systems job or the compilers job to figure out when it should fuse them together. But given the state of things today, we've suggested to users, "Hey, for

performance reasons you should essentially apply manual fusion, which essentially takes all of the functions and it in lines them together into a single larger function."

You'd want to do this if you are building for example a chat bot or your latency sensitive and latency of a system might be too expensive for you to tolerate if you have too many cold starts for an infrequent use case and this is a real use case where we have actually suggested to users, "Hey I think for your sake here you should just manually fuse all the used functions together." So it typically happens that if you have functions that are written in the same language like all Node.js code or all Python code and you're really concerned about latency and performance and your load is such that is very infrequent to when you might suffer from a lot of cold starts, you'll want to apply manual fusion.

Our goal however is to say, "Really you shouldn't be optimizing things at that level manually. It should be part of the automatic tooling that the platform provides that IBM cloud functions provides, for example, to do that optimization for you." And then we'd like to be able to broaden it so that you are not just restricted to fusing functions that are in the same language, but also how do you fuse languages across languages because I might have written an application in this model where I have some functions and Node.js, some functions in Python, some functions in Java. You typically can't do that very easily yourself. You'd have to increase the complexity of your code.

So that's part of what we'd like to automate and mechanize and provide that as a platform capability rather than shift the road into the end user.

[0:50:43.7] JM: There's a term you define called client side scheduling. Can you explain what that means?

[0:50:49.8] RR: So client side scheduling in the context of — So this came up in the context of exploring compositions. So let me give a simple example where you have a function that you want to execute conditionally if the result of some other function is true or false. Where do you do that orchestration that says, "Run the function, look at its result. If it's true then invoke the second function and if it's false then do nothing."

There is a scheduler that you have to run somewhere or you can think of it as a workflow executor. We've called it a scheduler because essentially it's scheduling the execution of your application. You can imagine that you would run this scheduler in a client like a browser. So you can write a Node.js application that runs in your browser that says, "Use the OpenWhisk API to invoke the first action," get the result and in the Node.js code that you are running say in your browser, the JavaScript code that you are writing in your browser writer, you're doing the conditional checks and that explicitly doing the invokes of the functions.

So by client side we mean it's a composition that runs outside of the serverless system itself. So the composition itself is not a serverless function. It has to run somewhere else. But by doing that, you can free yourself with a lot of the capability, the limitations rather that the serverless platform imposes on you. Like the client scheduler might be a long running orchestration that might last many tens of minutes. Typical function that runs in the serverless setting has to finish within five minutes and may not consume more memory than allowed. So if you want to break out of those limits, either you have to do something different within the platform itself or you have to run outside of the serverless platform.

So we call anything that you do to do this orchestration outside of the serverless platform as the client side scheduling and it is a way of executing your composition but using the API of a serverless framework and then you logically doing all of the things that you have to do to keep navigating the execution.

[0:52:51.4] JM: You mentioned this project called Composer. Can you explain a little more detail what Composer is?

[0:52:57.7] RR: Sure. So Composer is a library that we implemented currently in Node.js. It allows you to describe how you want to compose functions together and impart data flow and control flow. So an example I gave earlier where you want to do conditional execution of an action, you would express that in Composer with what we call a Combinator. So we do name of library, for example, is called Composer. You do a composer.if and that's a function call where you give it up to three parameters.

The first is the condition expression or the cloud function you want to run. That's going to give you whether you're going to go down the event branch or the else branch and then either a function or a cloud function or inline function that you would run for the event branch and then the else branch and Composer essentially gives you a library of control for the constructs to do things like tri-catch, to do an if than else, to do a loop, to do things like map and reduce or parallel and join kinds of compositions.

And the reason we took this library based approach is we didn't want to invent a new language but we also didn't want to change the semantics of how you might write an if than else in an imperative style saying, JavaScript. In JavaScript you just start writing, if A then B else C. We needed to extract from that sort of the three things that I was eluding to before, which functions are you going to run, what's the control flow, what's the state?

So by using this library of creating compositions, we force you into a style that helps us identify the control flow and the data flow and the state and to make it easier for you to develop these compositions, we also built a tool called the shell and what the shell allows you to do is visualize your composition as you are developing it. So you can write your code in your favorite editor, you point a shell to monitor that file and as you're building out your compositions about using more and more combinators, you actually have a visual representation of your control flow as you go along, and that helps you in debugging your code and you can sanity check that you've actually constructed an application that sort of fits with your mental model and your block level diagram that you might have.

[0:55:17.4] JM: Great. So, I read this paper that you wrote where you explored something called The Serverless Trilema and this was a way of presenting the tradeoffs of a serverless application. I thought this was useful because it presents a framing that developers could start to look at and say, "Okay, well these are the fundamental," you know, it's almost the time space fundamental tradeoff in regular computing. You have these three tradeoffs where functions should be black boxes.

Function composition should obey a substitution principle with respect to synchronous invocation and invocations should not be double build. So there is going to be tradeoff between these three areas. So I want to go through these and define them and start to flesh out what

your theories are behind the serverless Trilema. So first of all, the one corner of the triangle is that functions should be black boxes. What does that mean?

[0:56:16.7] RR: Right, so we think, or we believe rather that some of the fundamental value propositions in serverless is that you can write functions in just about any language that you choose and because of that, it empowers you to use the right language for a particular problem that you are solving. If you are going to support more than one language, at some point this polyglot, essentially polyglot functions then you might want to look at the functions themselves as a black box. You can't look inside, you can't see what they're doing, you don't know what languages they are built in and if you are going to treat them that way then you also can't necessarily modify them.

So that fits in the context of composition because if you want to do something special in the context of composition to satisfy some of the other constraints or you simply want to support the ability to compose functions that might be written in any one of these languages, you have to either buy into this notion that functions are black box, I don't know what they're doing, I cannot modify them or you have to say, "I don't care about this property because I care about some other properties," which I guess we'll get into and that's a tradeoff that you make. So strictly speaking, black box just means the serverless platform shall support functions written in any language or presented as strictly as binaries. They are unmodifiable, I can't inspect them, I can't modify them.

[0:57:41.1] JM: You touched on the serverless substitution principle; what is the serverless substitution principle?

[0:57:47.6] RR: So by substitution we mean that if you take two functions and you compose them together, we'd like the composition itself to be a function. It has to behave as a function. I have to be able to call it as a function. I should not be able to distinguish that I've called a composition from individual actions, and in OpenWhisk we had an example of this even from very early days because we thought this was a fundamental property even when we were building OpenWhisk with sequences.

So when you use Apache OpenWhisk today, you can create individual actions or you can create sequences, which is a primitive composition that essentially forms a pipeline. The way you construct, the way you execute the sequence is the same way you execute the action. If you were to look at an activation record for a sequence versus the activation record of an individual action, they look the same. They both have a result and they both have a start time and an end time and more over, I can take that sequence and now further compose it say with other sequences or other actions.

So it goes to just layered model of programming where you start with a building block functions for us and you want to build libraries out of them and you want those libraries to further compose into applications and essentially to continue increasing modularity and reuse, everything should have the same shape and in this context, everything should really look like a function. So substitution really goes to functions are the building blocks, compositions are the functions should also behave as functions. So I can always substitute a composition for a function or a function for a substitution.

[0:59:23.7] JM: All right, so the final corner of the triangle is the double billing issue and you talked about how if there is a remote scheduler for your composition of functions then you have a double billing problem because you're paying for the scheduler as well as the functions that you're code is getting scheduled onto. So what is the double billing problem? Why is that important to this tradeoff?

[0:59:51.9] RR: So the double billing applies really if you're going to try to run the composition inside the serverless framework. To the client side scheduling that you asked about earlier, if I am running the scheduler essentially that says, compose function A with function B, on my laptop then I'm not necessarily subject to double billing in that the serverless platform is not billing me for that scheduler. It is running on my laptop, it's outside that system but I have to keep my laptop on. Now the composition that I have running in my laptop can't be reused. So other functions can't really call unto my laptop to use it.

So if you want to bring this scheduler into the serverless model then now your subject and you have to play by the rules as serverless model applies on you, which means that you can't run forever, right? So typically actions will run for no more than five minutes. At that time, your

resources are freed up whether you are done or not. So if you want to compose two functions that themselves each take five minutes, there's just no way for you to execute that code without running the scheduler at the same time as your other functions and then having to possibly re-spawn yourself so that you can see if the execution is finished and now what you've done is you've written code that is basically paying for the idle time or the waiting time of the scheduler in the serverless framework.

So not only are you paying for the execution of the individual actions that you've run but you've paid for the scheduler that you are running as a function in a serverless platform and it's not just paying in terms of cost, you are paying in terms of time, which means you have to be subject to this time limit that the system imposes on you and you have to do some extra things to deal with corner cases where the things that you're composing run for more than five minutes.

So we use the term double billing because I think it's easy to understand in the context of I'm running the orchestration as a function, I'm going to pay for that. I'm running the actions as a function, I'm going to pay for those and so now I am double billed because I paid for two things and from serverless, one of the tenants of serverless is don't pay for idle time and the scheduler's role is really to just sit there and wait between executions that say "is it finished?" If one finished then I make the next decision about what to execute next and execute it.

So double billing really refers to schedulers running inside the serverless framework and having to deal with either the time limits that the system imposes on you or the metering that it does in terms of the dollars you pay for the scheduler itself. So together, double billing, the black box and the substitution principle provide three constraints that as a provider of the serverless framework, you have to choose among or do something special to support all three.

[1:02:43.0] JM: Now let's bring this full circle. *The Serverless Trilema* was this paper that you wrote about constraints that a serverless platform operator is going to need to abide by. We started this conversation talking about OpenWhisk, which is a serverless platform that you've contributed to heavily. Explain how the ideas that you explored in that paper, *The Serverless Trilema*, how did those affect the development of OpenWhisk?

[1:03:12.2] RR: Yeah, so a lot of those ideas came out of our work with supporting sequences in Apache OpenWhisk and as I eluded to, sequences and the way they evolved, we started running them as – just to remind the audience, the sequence is essentially an action that runs, which was going to individually call actions A, B and C say for a composition of those three actions. We started by having a version of sequences, which is self-ran as an action and then we realize that while we were able to satisfy the substitution principle and the black box principle, we weren't satisfying the double billing. You are paying for the function, which was a sequence and individual actions.

So as we tried to figure out how we best implement sequences in a way that also avoids double billing as I eluded to, it's not just a cost. It is a time limit issue, we changed the sequence implementation essentially internalize it into the controller. So while we are talking about the controller earlier I left out that there's special schedulers within the controller to understand the kinds of actions that you are running. So a particular kind is the primitive action or primitive function but also a sequence and so if you are running an execution that's a sequence or a composition that is a sequence, the controller understands that and then it schedules it internally.

Because of that, we are able to provide essentially a Trilema satisfying implementation of sequences and that's why sequences in OpenWhisk today are not doubled billed. They're actually the most efficient way of running a composition. If you just need a straight line code they support the black box because you can compose functions that are polyglot and essentially solve the double billing and the substitution principle. So a lot of our thinking in the space really evolved by actually having to build something that provides a pleasant programming experience in this context and doing it in the context of OpenWhisk.

Now where we stop with sequences or where OpenWhisk stops with sequences today is you can't do control flow. I can't do an if then else, right? And sometimes you might want to do an event condition action. I want to short circuit a sequence and I might want to do air handling. My sequence aired out, I want to try catch. I might want to flow parameters from one action at the head of the sequence all the way to the tail of the sequence but without really having all the individual actions in between to see that data because it might be a secret that I want to manage and I want to sort of keep separation of secrets from the rest of our data flow.

So when you start looking at going from the simple control and data path that sequences give you to building larger and more rich applications with different kinds of computational graphs, you need something new and so this is where we started building Composer and as we built composer we also worked on user experience and so Composer and the shell I think go hand in hand for that.

It is available on NPM, you just go download it and use it and as we were doing this for composition, we realized that again we have this tradeoff between the three axis of the serverless Trilema to solve and what we've done with composer today is we've put out an implementation that supports black box composition and to some extent, we are working on addressing the double billing issue and some of the changes that we're contributing from IBM research back to Apache OpenWhisk will address these substitution principle as well.

So just like what we have done in sequences before, we'd like to bring this full capability of compositions that are Trilema satisfying to the framework and this is what sets I think Apache OpenWhisk and our work with it from IBM in that context. We can make this changes and one of the things our paper shows is that the platform has to do something special if you want to satisfy all three aspects. Otherwise you have to break one of them. There's just no way around it and if you look at some of the other models in the space, you can really cast and see which of the three constraints they break.

[1:07:13.1] JM: All right, well I'd love to wrap up by just talking a little bit about your research and what else you're concerned with around the serverless area. Because I haven't seen many papers written about serverless other than *The Serverless Trilema* and I think a few others. What are the areas of research around serverless that you would like to examine?

[1:07:33.5] RR: There is so many and this is I think what's exciting and actually a good place for us from IBM research to drive some of these and the framework to look at this and I think the way to imagine how we're approaching this is you look at serverless as a new kind of cloud computer. Composer gives us essentially a foundational model for how we want to program this cloud computer and now we have to build a stack of middle ware tooling programming client's etcetera on top of it to make the user experience pleasant, to deliver run time performance

that's good to keep presenting essentially various value propositions and you can look at things that we're doing in the context of analyzing the functions themselves to know whether to actually state list or not.

Looking at compositions of stateless functions, have you composed functions that are well formed? Are schemas aligned properly? What are the issues around security in the context of compositions? Can you do things like taint analysis to say, "Hey look your composition is leaking sensitive data. You did not intent to do this perhaps. Can you provide approval guarantees in the space that hey, your composition is sound or not sound? Can you do things with respect to runtime optimizations?"

Because I have the composition, I know where you are going to go in the space of your computation, can I use that to inform your framework about what resources to pre-provision and how to optimize your code for cold starts and warm starts? Can I do things like fusion? Can I do function re-ordering? So there's a lot of things that fall into context of analysis, compiler optimizations, runtime optimizations and tooling and we're really looking at the full gamut of these and taking a very vertical approach from the bottom up in terms of trying to deliver a serverless computing programming model in the space that really tries to target sort of cloud native by construction with a lot of interesting value propositions.

[1:09:31.0] JM: Okay Rodric, well it's been great talking to you. I am fascinated with OpenWhisk and it's really been great to get a look inside of how a serverless platform actually works. Thanks for coming on the show.

[1:09:42.1] RR: Thank you Jeff, this was great. I really enjoyed it.

[END OF INTERVIEW]

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]