EPISODE 807

[INTRODUCTION]

[00:00:00] JM: On Amazon Web Services, there are many ways to run an application on a single node. The first compute option on AWS was the EC2 virtual server instance, but EC2 is a large abstraction compared to what many people need for their nodes, which is a container with the smaller set of resources to work with.

Containers can be run within a managed cluster like ECS or EKS. Containers can be run on their own, such as in an AWS Fargate instance, or simply as a Docker container running without a container orchestration tool.

Beyond the option of these explicit container instances, users can run their application as a serverless function as a service such as AWS Lambda. Function as a service abstract away the container and let the developer operate at a higher level, while also providing some cost savings. Developers use these different compute options for different reasons.

Deepak Singh is the director of compute services at Amazon Web Services, and he joins the show to discuss the use cases and the tradeoffs of these options. Deepak also discusses how these tools are useful internally to AWS. ECS and Lambda are high-level APIs that are used to build even higher level services, such as AWS Batch, which is a service for performing batch processing over large datasets.

Before we get started, this is the last week of the FindColalbs hackathon. This is the hackathon for the new company that I'm building, FindCollabs. You can find out more about this hackathon by going to findcollabs.com/hackathon, or softwareengineeringdaily.com/hackathon. The first place prize is $4,000 cash along with some SEDaily hoodies and an appearance on Software Engineering Daily, and the second place prize is similar, but a thousand dollars. There's also other prizes, and it's a great place to find and collaborate with other people. You can go to findcollabs.com to check out what FindCollabs is. It's a place to find collaborators and build projects, and you can go to findcollabs.com/hackathon to find out more about the hackathon.

Thanks for listening.

[SPONSOR MESSAGE]

[00:02:34] JM: Testing a mobile app is not easy. I know this from experience working on the SEDaily mobile application. We have an iOS client and an Android client and we get bug reports all the time from users that are on operating systems that we did not test. People have old iPhones. There are a thousand different versions of Android. With such a fragmented ecosystem, it's easy for a bug to occur in a system that you didn't test.

Bitbar is a platform for mobile app testing. If you've struggled to get to continuous delivery in your mobile application, check out bitbar.com/sedaily and get a free month of mobile app testing. Bitbar tests your app on real devices, no emulators, no virtual environments. Bitbar has real Android and iOS devices, and the Bitbar testing tools integrate with Jenkins, Travis CI and other continuous integration tools.

Check out bitbar.com/sedaily and get a free month of unlimited mobile app testing. Bitbar also has an automated test bot, which is great for exploratory testing without writing a single line of code. You have a mobile app that your customers depend on, and you need to test the target devices before your application updates rollout.

Go to bitbar.com/sedaily and find out more about Bitbar. You get a free month of mobile application testing on real devices, and that's pretty useful. So you can get that deal by going to bitbar.com/sedaily. Get real testing on real devices. Get help from the automated test bots that you have some exploratory testing without writing any code, and thanks to Bitbar.

You can check out bitbar.com/sedaily to get that free month and to support Software Engineering Daily.

[INTERVIEW]

[00:04:43] JM: Deepak Singh, welcome to back to Software Engineering Daily.

[00:04:46] DS: Thanks for having me back.

[00:04:48] JM: I want to start y talking about the ideas that we explored in the last episode just a little bit, specifically on the abstraction of compute, because we just did a show about the abstraction of storage with Kevin Miller, and I think pairing those two discussions of storage and compute both in a cloud and a single node context will make for some good lessons around computer science.

So if we just think from the point of view of a single node that we want to run, a single application node. There are many ways where we can run a single application node in the cloud. Describe some of those ways.

[00:05:32] DS: Yeah. Great question. I used to have a slide that I used to show I think in the first slide of every talk that I used to give back in the day, and it literally said – I show two pictures. One was of a server, like a physical server, and one was of "my server", and it's an EC2 run instances. Quite in those days, that was the simplest way. This is now nine years ago, something like that. Simplest way to launch an instance, which was EC2 run instances and one small – Something like that, which is at the end of it, you had one machine, one virtual machine with a network that you could do things on. It's actually the first thing I ever did on EC2 back in the day, now, 2006, when EC2 first came out.

Since then, the ways that you can run a single compute entity has evolved. You can still use the EC2 CLI, you can use APIs. You can spin up something like AWS Fargate, which is almost like a custom built – You could use it as a custom built instance. Give it some CPU. Give it some memory. Give it a network, and at the end of it you can put code on it and run.

But I still think that the EC2 run instances call with a T3 small, or T3 micro, I guess now, is the simplest way to spin up a single machine if what you want to do is spin up the machine, then log in to it later, and then use it as a machine that happens to be on the other side of a network.

But I think our customers have gotten little more sophisticated. They want to declare intent and they want to be able to, at the end of it, have the machine run and do things. From an AWS context, there's many ways to do that. You could use something as simple as Elastic Beanstalk,

where you decide that you have an app, a 3-their app that's running on one node. You put it behind. You actually don't do anything. You don't give it that intent and it spins up an app.

Similarly, I think a service that is near and dear to my heart is AWS Bath. The Java for AWS Batch is to spin up just the right amount of compute that you need for the work that you have in hand, and you could say, "I want to know more than one machine," and you could basically say, "Here's my job," and as long as the job – You put the job into a queue, and as long as there's work in the queue, it will keep spinning up that one machine to do that work and then it will shut it down. So if you have no work to do, you don't have a machine.

If you have work to do, you have one machine, and that model I think is pretty powerful, because you're not thinking about the machine now. You're thinking about the work, and you've just put a constraint on that work that says, "Don't run it on more than one machine." I think that's a very powerful paradigm that I think more and more customers are asking us to look at in many different ways in the context of batch computing. That's AWS Batch.

[00:08:26] JM: So I completely agree with you about the idea of people just want to have a way to get their applications to run on the cloud, and they don't necessarily want to address individual server instances. They may not want to even address individual functions. However, if you're building something like AWS Batch, somebody is going to have to be addressing servers. Somebody is going to have to be addressing functions. So I do want to explore a little bit more the different ways that we can address these particular servers, these particular functions and the tradeoffs that we're making along this spectrum.

So whether we're talking about running an EC2 instance, AWS Fargate container, a Docker container running on an EC2 instance, an AWS Lambda function. Give me the spectrum of options for running a single node and the tradeoffs that we're making along that spectrum.

[00:09:30] DS: So I just gave – Just before this conversation, gave a talk where I talked about the AWS shared responsibility model. One very simple way to think about, I'll use the context of EC2, because that's the best starting point to have that discussion, which is with EC2, AWS is responsible for the physical infrastructure, the physical machine, the network. Everything up to a user space, the operating system that your applications run in, the application itself, the

permissions on what that instance is allowed to do, or your responsibility as a customer. You have to set the security groups. You have to decide whether you want to allow ingress, which we see it goes in. Those are things that a customer sets up. But we are responsible for everything underneath that.

The next stage above that – And I'll use databases as an example. You want to run a database. You can run it on EC2 yourself, or take the container example. You can take a Docker container and run it on the EC2 instance. You are still responsible for the lifecycle and lifetime and the patching of that instance. You are responsible for landing the Docker container on to that instance, either you log in to that instance and do a Docker pull and a Docker run. You pull a Docker run and then pull the instance from some registry somewhere. But how you do it, when you do it, everything is your responsibility. AWS is still at the, "We will give you an instance and we will guarantee that underneath the hood we are doing the best we can to give you a high-quality experience."

Actually can take that one step further now, that we do bare metal instances. You could entirely be responsible for running your own hypervisor on a bare metal instance, because you can install and run it. But I think what we found is our customers feel every category. There are customers who want to run their own hypervisor on a bare metal instance. Most customers, historically, have wanted to run their applications on an EC2 instance. So they are responsible for selecting the operating system. Do I want to run Red Hat? Do I want to run Amazon Linux 2, or do I want to run Ubuntu and the applications inside it?

But more and more and more they're starting to move towards a world where they want us to take on more and more of that responsibility. If shared responsibility is lined in a box, that line keeps going up where everything underneath is what is AWSs responsibility, and what's on top is a customer's responsibility.

The next stage is running something with ECS or EKS, or actually the next stage is running your own Kubernetes, where you now have a scheduler making those decisions for you, but you are still responsible for the box that it's running it on. With ECS and EKS, the running of the scheduling system is no longer your responsibility. You are still responsible and have ownership of the nodes that your containers run on, but the whole management infrastructure is no longer

your responsibility. It can get automated away quite a bit, especially if you start using things like auto scaling, where if a node fails, it gets spun up again. You are no longer worrying about, "Oh, I now manually need to start another node," because auto scaling has done that for you. We've automated – The orchestration of your machine is now handled by something else.

The next step of that is Fargate. The machine goes away. All you're responsible for now is your container. The patching of the host operating system, not your responsibility. You don't need to think about it. AWS is making that decision for you. You are still responsible for your container itself. If you're running an operating system inside your container, you're responsible for that. You are responsible for – If you pick up a malicious container from somewhere, that's on you. We have now prevented it from breaking out into the machine and infecting somebody else. But the malicious container is still yours.

Then the ultimate step – Well, for now, in that shared responsibility model, is Lambda where all you're responsible for is the code that you bring. Not the container, just the code. At that point of time, AWS has said, "We are taking all the responsibility of scaling, concurrency, encapsulation, everything. Your only responsibility is, "Here's a function. I'm going to run it, make it run."

I think our customers actually enjoy the fact that they have the full spectrum of responsibilities, because even within the same application or the same company, you may have all of those. They may be running a database by themselves, because there are specialized needs. On EC2, a big chunk of their application might be running inside Lambda functions, and a number of services within their application might be running inside something like ECS, or Fargate, or EKS.

So the fact that you have the flexibility allows customers to reason about their application based on the needs at the time. I do think in an ideal case more and more customers want to run as much for "serverless" as possible where most of the responsibilities take on AWS. You are making a tradeoff. You're assuming that we are going – The behavior of the system. AWS will handle it correctly. Having to scale, how to scale, all of that you're leaving to AWS. So I think that's the tradeoff you're making.

[00:14:32] JM: When we think about the different kinds of workloads that we might potentially want to run on one of these compute mediums, like Fargate, or Lambda, or EC2, or whatever

we're running our stuff on, we could think in terms of the Kubernetes types of workloads, which are stateful, stateless, batch and daemon. If we were to map those different workload types; the batch, the daemon, the stateful and the stateless, to compute mediums on AWS, what is that mapping look like in your mind?

[00:15:14] DS: Well, you can run EKS and you get all of those at Kubernetes.

[00:15:16] JM: Sure, absolutely.

[00:15:17] DS: That's a good starting point. ECS is much the same. You can both mostly run any kind of application.

[00:15:23] JM: But, I mean, more abstract.

[00:15:24] DS: Yeah. Here's what we found when we launched ECS. We had two clusters of customers right in the beginning. There were people – Well, let's make it three. There were people who wanted something that made it easy for them to deploy code. That is the canonical container use case. I have code, I want them capsulated. I want them deployed on to a machine and have it execute there. It made it super easy. Container orchestration, in general, made that easy.

They're the class of customers that was, "I want something that allows me to run these stateless services." A lot of those customers are now using Fargate, because it's stateless by design. But we found these customers that are building these workflow engines, these batch computing engines on top of ECS. Essentially, if you're familiar with batch computing, batch jobs gets spread out across many machines. You need some engine that makes a decision on how they get spread out. You need an engine that decides how you divvy up the work. Most and very often, it reads it off a queue. You're putting a job into some kind of queuing system and it takes it off the queue and distributes on to a number of machines. Everybody was reinventing the wheel.

We stepped back and said, "Okay. What if we –" They were using things like Grid Engine, like Slurm. They were doing it on EC2, or they were trying to use ECS – ECS allows you to write

your own schedulers. They were writing custom schedulers on top of ECS to essentially achieve that. We said, “Okay. That seems unnecessary,” because everyone is reinventing the same wheel. What can we do? Can we do it in a way that takes advantage of the capabilities of the cloud? Not using something that was written for more static infrastructure. That’s what led to AWS Batch, which was, “Declarer intent, tell us how you want your work to be done. Do not even tell us what kind of machine you want to run it on. Just tell us how much CPU you need roughly, how much memory you need? What ratio you might want. Put your work. Give us the executable. Just package it up in a container. Put it in a queue and let the system take over from there.”

The system will scale depending on demand. It will scale down to zero when it’s done, which is I think one of the things that everyone wants to do. We have no work to do. Just spin everything down. Most importantly, the customer no longer have to decide what instance type to run on. The system made that decision for them.
Even though batch runs on ECS and there’s EC2 instances underneath the hood, from a customer standpoint, you abstract it away most of the decision making. All the customer now has to think about is, “I have an executable. I roughly know the CPU to memory profile of this executable needs to fit and not get home killed when it runs. By the way, I’m going to scale horizontally,” and AWS make the decision for me.

I think that epiphany of the fact that we can purpose build systems on top of, we happen to build it on ECS. We didn’t have to. We could have built it in many ways. Containers makes it super easy to build a system like that, but the fact that we could build a purpose built service for batch computing is what did that.

Databases are very similar. People have been running databases on EC2 forever, but then you are responsible for managing and batching your database. RDS takes it one step further. Aurora takes it a step even further where you no longer think about the database at all, and it’s a fully managed service. The ultimate one is DynamoDB with auto scaling, where you literally don’t have to think about anything. The system scales on its own. You do have to make some tradeoffs, because now you move to a different database architecture.

So I think the approach that we've taken is we will give you the raw, most fundamental primitive that you need to be able to run any kind of application. As we understand what tradeoffs people are willing to make, we want to abstract that away and put them into purpose built systems that allow people to do. Batch allows you to do one thing, batch computing. But it's very effective at doing that. It's a big enough use case that it's worth it, and you've done this with databases, we've done with the batch computing, and I think you'll see us do it for even more things in the future.

[SPONSOR MESSAGE]

[00:19:37] JM: This podcast is brought to you by wix.com. Build your website quickly with Wix. Wix code unites design features with advanced code capabilities, so you can build data-driven websites and professional web apps very quickly. You can store and manage unlimited data, you can create hundreds of dynamic pages, you can add repeating layouts, make custom forms, call external APIs and take full control of your sites functionality using Wix Code APIs and your own JavaScript. You don't need HTML or CSS.

With Wix codes, built-in database and IDE, you've got one click deployment that instantly updates all the content on your site and everything is SEO friendly. What about security and hosting and maintenance? Wix has you covered, so you can spend more time focusing on yourself and your clients.

If you're not a developer, it's not a problem. There's plenty that you can do without writing a lot of code, although of course if you are a developer, then you can do much more. You can explore all the resources on the Wix Code's site to learn more about web development wherever you are in your developer career. You can discover video tutorials, articles, code snippets, API references and a lively forum where you can get advanced tips from Wix Code experts. Check it out for yourself at wicks.com/sed. That's wix.com/sed. You can get 10% off your premium plan while developing a website quickly for the web. To get that 10% off the premium plan and support Software Engineering Daily, go to wix.com/sed and see what you can do with Wix Code today.

[INTERVIEW CONTINUED]

[00:21:44] JM: So let's get to a conversation about AWS Batch, what it does and what the architecture of the project is. Let's just start by defining batch computing as you see it.

[00:21:57] DS: Yeah. So batch computing at its heart is I have some work to do. That work is best done by splitting it up across many, many machines, and the total work is not done until everything has been processed. You have a dataset that you want to process. You could run that dataset on one machine. It will take N-time to process. But if I can break that work up into, say, 10 copies, I can spin it under 10 machines, and in theory get it done in 1-10th of the time. That's the fundamental premise and promise of batch computing. You've taken a job. You divided it up into many smaller chunks and distributed it over many machines helping you get it done faster.

There's a subset of batch called – There's something called high-performance computing, where all the machines are talking to each other and they're running in concert and not independently, which is high-performance computing, and that's a subset of batch computing. That's how I think about batch computing, that at AWS, that's kind of how we've taught about it over the years.

The biggest challenge in batches always been what's the best way to distribute this work across whatever number of machines you have and how sometimes – Because you often have lots of work, how that work gets prioritized on to running on those machines, which is where things like queues come from, so that you can have – Job A has more priority than job B. So job A is always going to get scheduled first and job B will get scheduled if you have the time and space to do it. I think those are the patterns that have emerged in the batch world that we've wanted to support on the beginning.

[00:23:29] JM: Can you give an example of a modern batch computing application that we can perhaps reuse?

[00:23:35] DS: Yeah. I mean, the most simple one is – There's a canonical example. I have a bunch of images. I want to convert them into five different types of thumbnails. Fundamentally, batch jobs. You could do them on a laptop, but if you have a million images, you'll be sitting

there for a very long time, and batch computing just makes it easier for you to distribute that job on to a large number of machines ideally suited for the cloud.

[00:23:57] JM: Perfect. Although, what I like about the example is that it's naively parllelizable. You could schedule each of those jobs on to individual computers and have them finish when they do, and then you've got all your work and it's done. There are other types of workloads that require some serializability between the different pieces of work that you're getting done. Do you have a good example of a serialized –

[00:24:24] DS: So science is a great place where this happens. So I come from a science background. In genomics, you have pipelines where you have s set of data, you process it. The next type of computing you want to do depends on the results of the first set. So you can't actually do it without actually having the first set finished. So you have a dependency graph. You can only do the next step if step one finishes.

Another sort of variant on that is parameter suites, where you have a single executable, a single dataset, but you want to run it across a number of different parameters and you'd apply the same parameter on to the dataset. Those are common enough use cases that those mechanisms actually built into most batch computing systems.

In the case of AWS Batch, you actually have a dependency graph built in. The idea that job B cannot start because it depends on job A, and it needs an exit code from job A, is part of how AWS Batch works. It's not unique to AWS Batch. Most batch systems have that kind of capability.

The second one is a concept called array jobs, where you have an executable. You have a dataset and then you basically have a metrics. You give it a metrics of zero to N variables. You can have I think up to 500,000 with batch. It essentially is one job that spawns 500,000 children jobs. Each is a copy of the same thing just with a different starting point, a different variable that it's optimizing for. Different function it might be trying to optimize for. It's very efficient way of doing very large jobs with a single call. In batch, we call them array jobs. That's, again, not an uncommon feature in most sophisticated batch computing systems. To your point, that's where it gets more interesting.

[00:26:14] JM: Yeah. So we've discussed some different types of batch computing. Explain what AWS Batch is.

[00:26:21] DS: Yeah. So the idea of AWS Batch was it batches two things. The way I like to think about it and where a lot of our customers use it it's an execution system. So let's say I am a bioinformatics workflow engine. I have built the workflow engine to – You start off with some set of genes that you want to do some calculations on. You are using a set of tools that probably most biologists are familiar with. You just want to help them build a workflow system that they understand as not computational experts, that depending on the science that needs to be done, executes the workflow at the end of it to get the results.

On AWS, all the computational part of that is delegated to batch. Batch, from their standpoint, is an execution system that takes an input and takes the requirements on what they want to get done. At the end of it, it spits out some data back to S3. It's the most common pattern. At its heart, it's just an execution system that has all the semantics of a batch system. It has jobs. It has queues and it runs things in parallel.

Whether they're distributed naively across many machines or as a single unit with MPI jobs, or as we call them, multi-node parallel jobs. So that's what it is. It also does two thing specifically. It's a scheduler. So we built our own scheduler that understands things like spot instances. It understands things like auto scaling. It has knowledge of what instances are available to it and it has a rules engine at the back that ties to find the right instance type for the work that you have to do. Sometimes it tries to optimize based on what it knows about the behavior of a particular instance and the quality of – Like the CPU type, the memory bandwidth, things like that. It's also a system that manages the lifecycle of your job. It starts up when you have work, and it spins down when it doesn't have any.

All of that collectively is what we call our batch execution engine, where all the customer needs to do is say, "Here's the work I want to do on this data. Here are the sort of high-level conditions that I want. Please do it for me," and this thing spins up.

[00:28:32] JM: What are those high-level conditions?

[00:28:34] DS: The most commons ones are, “Here’s my executable. Here is roughly the resources that my executable needs. It needs four gigs of memory per core,” as an example, or, “I really need a GPU for this, because it’s [inaudible 00:28:47] code.” We aren’t smart enough to figure that out yet – And priority. This is my higher priority job. So we actually have priority as a specific – It’s a first-class entity. Here’s the fun part that’s unique to AWS, which is, “I’m willing to pay 70% of on-demand,” of the on-demand price for instance. So it puts it into a spot queue. If you have a spot instance available at the right price, it will even pick it even if it’s like the memory to CPU ratio is very different, 8 gigs of RAM per core. But if the price is right, why not? And we’ll spin up an instance. You could build all of these on your own with a bidding engine and trying to figure out how to use spot fleet. The thing that batch does, it takes all of that away from you and just says, “Give me your requirements and I’ll execute them.”

[00:29:34] JM: Now, what makes this an interesting conversation is I’ve had discussions with people who are building schedulers. Usually, financial cost is not one of the axes that they’re exploring on their scheduler system. Certainly, priority is it. Priority is there. Time and space and the different computing primitives that we think about as resources, but the money aspect of it is really interesting, and you have that as an axis, because you have this spot market. I guess, could you just explain that in more detail? Why does financial cost fit into this scheduler model?

[00:30:16] DS: I think financial cost always fits into the scheduler model, except that when you are in a more traditional scheduler sense, it’s built-in to the fact that you have a single shared cluster. The most common problem in any shared cluster, having run on many of them, is do you have the resources you need when you need them?

I’ll give you an example of a customer we talked to. They had this one professor on staff who had the most funding. So when his funding came and the work came, he would take up 90%, sometimes even more, of their shared cluster. Everybody else at the institution had to fight for the remaining 10%. So you have a finite resources. The whole idea of that scheduler and the way traditional schedulers are designed is to take that shared resource and optimize it as best as you can.

Most clusters in shared facilities are run at 100%. What the tradeoff that you make, which is effective to the spot tradeoff is other things are not going to run, because you don't have space, and/or you actually have the right grants and say, "No. No. I'm going to get space on this now and it's important." Somebody is making that decision for you.

SO the calculus in a traditional scheduler is, "I have N-amount of resources. How do I use this N-amount of resources efficiently and effectively across the Y-number of people and the Y-number of jobs that want to run on it?"

In the cloud, that doesn't make sense. What you really care about is, "I have work, and this is how much I'm willing to pay for this work." The cluster is essentially scoped to the job. You're not sharing it with anybody. It is just relevant for the purposes of your work. When the work is done, the cluster goes away. So you're no longer trying to – Although some of our customers still try and share work across a cluster. Sometimes it makes sense. But for the most part, it's job-scoped. You spin it up. You spin it down. You're no longer competing with somebody for resources on the same set of infrastructure. What you're not trying to figure out is, "Okay, how much is it going to cost?" In a traditionally shared cluster facility, that's going to built-in to the fact that you had to buy the machine and put it in and all of that. But somebody is thinking about it. It's usually not the first in running the work.

[00:32:33] JM: One interesting thing about architecting AWS Batch is that you're not just building on kind of normal computer science primitives. You're building on top of other AWS services. So all those different types of runtimes we discussed earlier, EC2, Lambda, Fargate, ECS, these are all options for how you could schedule this computing, this large computing job that might get issued to AWS Batch. How have you explored those different compute options when architecting AWS Batch?

[00:33:14] DS: Yeah. From a customer standpoint today, technically we support only EC2. We express it within containers. To some extent, that's a detailed addition and how to care about. We do hear a lot about Fargate and Lambda as, "Hey, can you take this paradigm that you have for spinning up and shutting down resources and provide this to us with Fargate and Lambda?" It's something we're thinking about quite a lot.

With Fargate and Lambda, it's less of a problem right now because of the way they work. You're not spinning up machines anymore. But the whole idea for queuing system that knows and has farm out everything is something that we hear about a lot. To some extent, that role is played by step functions in those worlds.

Step functions can do a lot of what batch does for EC2. Step functions can also run on top of batch, because it's a big dependency graph in itself. But you can define a graph in step functions. Because lambda functions are so small, the function gets executed the next step if you need to go into the next one, you go on to the next one. If you have to farm out your pipelines – Lambda has the concept of forked events, even forked pipelines, can use a queuing system, like SQS to do that. So you can build those kind of architectures. But customers still have to build them.

So one of the big things that we do get feedback on is, "Can you give us similar capabilities for Lambda and Fargate?" We're trying to figure out exactly how it fits in, because a lot of the feedback and use cases that we historically gone after is people struggling to do that with EC2 or ECS. Fargate and Lambda, I think people still haven't – with exceptions, started running this kind of crazy sophisticated shared cluster scenarios.

[00:34:50] JM: So I guess what I'm trying to ask you is the interface for the developer of interacting with batch is I've got this big batch job. I want AWS Batch to take care of it. I want to think in terms of priority and cost and resources. Under the hood, AWS go do whatever you want. Now, you, Deepak Singh, are working on the architecture of what AWS Batch is actually going to do when the person says, "Okay, go ahead and run this."

Are you saying that all that goes on right now under the hood is gets scheduled on to EC2 instances?

[00:35:25] DS: Yeah. So the end result, it gets scheduled on to an ECS cluster, because we have built a custom ECS scheduler for multiple custom ECS – We actually have multiple schedulers on the batch. It happens to use Lambda and API gateway heavily itself. But from a customer standpoint, the interface they see is a job. They write a job definition, and they put that

job – Basically put that work into a queue, in a kind of multiple queue. That's the only interface that really see. That's the only interface that should care about.

In theory, there's nothing that says at the end of that interface, you shouldn't have an ECS instance. You could have a Lambda function on a Fargate task. We haven't gone there yet. Right now, it's a single container task. We execute it. We send it back. Put the data in S3, or EFS, or wherever you want to put it. But that's the current model. You could extend that model to any compute paradigm. I think the needs and how people think about, because they're coming from traditional batch computing, much more better understood for EC2-based world, which is the way we started.

[00:36:27] JM: Right. Now, when you're thinking about how to execute these batch jobs economically, I mean, running them on ECS makes a lot of sense to me. Would it be even more economical if you could schedule these batch jobs on to Lambda?

[00:36:43] DS: I think the folks at the RISELab in Berkley – yeah. So the stuff they've done with Pywren.

[00:36:49] JM: Yes.

[00:36:49] DS: Yeah. So there's nothing in Pywren that says that we couldn't build a batch system that does what Pywren does. Right now, they're about the only people who use Lambda and that sophisticated way for the kind of sophisticated batch computing that you're doing. There's not that many. I mean, most of the Lambda, batching Lambda work, is very event-process, real-time event processing kind of workload, which function that Lambda can handle pretty easily without having to build job management, etc. But I think that's going to happen, and we do get that. We get a lot for Fargate actually now, now that we've dropped Fargate prices. It's a very common feature request. You should expect batch to start supporting – In a perfect world, we would make the decision on which compute platform it goes to. We haven't gotten there yet.

[00:37:34] JM: Have you talked to the – So the Berkeley folks who are all doing this research in the serverless, Yan Stoica and the rest of the –

[00:37:42] DS: Eric Jonas is the person I've spoken to years ago before he got into Lambda.

[00:37:47] JM: Okay. Did you read the paper that came out kind of recently?

[00:37:50] DS: Yes.

[00:37:51] JM: What do you think of that?

[00:37:52] DS: I mean, the work they've done with Pywren is just fascinating. The fact that they're able to get this performance out of the system that they built. It's a great example of showing that you should not – If you've identified the right kind of workload and you understand what you're doing with it, the scale that you can achieve, the cost targets that it can achieve with Lambda are just fascinating. I think this is the first examples that we have seen of people doing mega-scale computing with just serverless.

It's not naïve computing. It's pretty complex computing. I think much like AMPLab and Gradlab before RISELab defined a lot about how people are going to do big data back in the day. I think you will get some very interesting fallout from the work that they're doing, which in the end make it into more customers will get interested. That will then result in us having to build things to address sort of the broader use case.

[00:38:43] JM: Well, I found so cool about that paper and kind of heretical in some ways was it was like this is a scientific paper that is around like a commercial tool. It was almost like looking at a commercial tool as a scientific project that is yet unexplored.

[00:39:06] DS: I think it's not that. If you go back to 2009, 2010, you'll find a lot of paper from RAD LAB, which was a predecessor of what is not RISELab, which are very similar and they were doing things with EC2, because EC2 was unique at the time. I think it's a question of they are computing – The core purpose of that facility is to look at new way and new patterns of computing. 10 years ago, there was EC2. Now it's serverless. So I think that's just an evolution of where computing is going. Just like their early work on EC2 was an early indicator of where the world was going, I suspect the —

[00:39:41] JM: So in that paper they have this exploration of different storage mediums for serverless functions and they basically conclude, they're like, "We don't have the ideal storage mechanism for doing serverless computing," was what I understood about that. Do you remember that part?

[00:40:02] DS: Not enough to opine on it with any large scale or detail. I do thin with traditional batch computing, I think for a lot of the applications that people run on serverless world, something like Dynamo is just perfect. It fits that model really well. Not necessarily for the kind of computing that the Pywren folks were doing. It's one of the interesting aspects of the computing that happens with batches. The application is very much still expect file systems.

It's one of the reason why adding support for FSx for Lustre was one of the biggest request we had for AWS Batch. Well, support for parallel distributed file system like Lustre from our customers early on, because their applications almost expected it.

I do think now with some of the newer instances, which are very high-bandwidth to S3. I think those are the models, that and DynamoDB, especially with something like DAX in front of it are the models that most customers will be able to use very effectively. There could be new storage types that come up. I haven't looked at it enough to have a meaningful answer to that.

[00:41:05] JM: The things I remember – well, I guess the thing I remember most distinctly about like the stateful serverless workloads were like Lambda function to Lambda function communication or – what are the ways in which storage mediums are insufficient for doing Lambda processing today?

[00:41:26] DS: I have to go back, based on what you said, I'm assuming you take a hop from Lambda to a data store and bring it down again. I think they might have been talking about going directly. I don't know. I'd have to go check.

[00:41:37] JM: Which you can't do today.

[00:41:38] DS: You can use something like Kinesis to stream data from one process to the other. Like I said, I don't know enough. But I do think that how people design their applications is going to evolve. Historically, you always assumed on a shared storage system. Usually, a big shared parallel file system to do it. In the cloud, there are different storage patterns. I'm sure everything will evolve together.

[SPONSOR MESSAGE]

[00:42:09] JM: Today's episode of Software Engineering Daily is sponsored by Datadog, a monitoring platform for cloud scale infrastructure and applications. Datadog provides dashboarding, alerting, application performance monitoring and log management in one tightly integrated platform so that you can get end-to-end visibility quickly, and it integrates seamlessly with AWS so you can start monitoring EC2, RDS, ECS and all your other AWS services in minutes.

Visualize key metrics, set alerts to identify anomalies and collaborate with your team to troubleshoot and fix issues fast. Try it yourself by starting a free 14-day trial today. Listeners of this podcast will also receive a free Datadog t-shirt. Go to softwareengineeringdaily.com/datadog to get that fuzzy, comfortable t-shift. That's softwareengineeringdaily.com/datadog.

[INTERVIEW CONTINUED]

[00:43:17] JM: I want to jump to a totally different topic, and we can come back to batch. But just because of we're at AWS Summit and this was discussed today. But since we've been talking about AWS Batch, which is I think of as very transient workloads. On the other side of the spectrum, you have long-lived workloads, maybe services that we need to run for a longtime. I think the one necessary component for a modern distributed runtime of long-lived services is this idea of a service mesh or a service proxy. Every large company that I've talked to has some kind of service mesh or service proxy, and App Mesh is a tool from AWS that recently went into general availability.

[00:44:08] DS: Yeah, today.

[00:44:09] JM: Today. Yeah. So describe what you see as the functionality of this service mesh category.

[00:44:19] DS: It's kind of funny. When we started – It's one of the lessons we learned from App Mesh. When we started a couple of years ago, we had this notion that people were building certain kinds of applications and containers. When you have lots of services, you basically get this big graph of things talking to each other.

Along the way, something like – You had patterns like SmartStack from Airbnb that have been around for a while. That did something similar to what we call service mesh today that allowed your application to talk to a local proxy that then talk – And the proxies talk to each other. Even in traditional sort of a web computing, you had systems – Like systems we've had at Amazon for a longtime, where you built an application. You talked with some library, which you had to embed the library in the system. But when you deployed it, you certainly got graphs and metrics and alarms and everything that you needed.

As the application then become more and more decomposed into smaller and smaller components and they're polyglot, you're writing them in different kinds of languages, there are different kind of components in the systems. Keeping track of all of these becomes really tricky. That's why I think service mesh has got very popular in the container world, because people are starting to deploy two things.

A lot of these kinds of applications are being written for containers and deployed inside a container orchestration system. The second part was you had an entity, the container orchestrator, that could sort of manage the lifecycle of the control plane of this mesh.

We realized a couple of things. One is, in the end, what is a mesh? It is a network. It is a network that leaves early at 7. If you think about traditional networks, you are thinking about IP addresses. You're thinking about side arranges. When two things communicate with each other, you're thinking in terms of traditional layer 3, layer 4 networking.

Most application developers just want to know, “Can my application FU talk to application BAR and can it find it? Which applications and components in my network is it allowed to talk to?”

For us, App Mesh is not a service mesh. It is an application layer network. It’s a fabric on which applications talk to each other. At GA, these applications are run in containers with a proxy next to it, a sidecar proxy. They are run on EC2, which could also have a sidecar proxy running in it. In this case, the sidecar proxy is Envoy, but we perceive it as an application network and your services get deployed into that network. As supposed to a lot of the other way, I think the other way that a lot of people think about meshes is you have this container orchestrator that is the controller of all things and everything runs inside that. But that gets very limiting, because no one runs inside the – It’s very rare to find your entire – Everything running in the context of one cluster. If you’re running across clusters, now you have to figure out how clusters communicate with each other, etc. All you really want to think about is your application and how do the components of my application communicate.

For us, App Mesh is this application layer fabric, application layer network, where today you can deploy it in ECS, you can deploy it in EKS, Fargate, run your own arbitrary homegrown container scheduler that you build however you want to. You could build something just on EC2 and have no containers at all. They’re just services.

The heart of this is something called – There’re two hearts. One is Envoy, which is the sidecar proxy that we support. That allows us to get a very rich set of partners. We announced I think 13 – Very large number of partners today in the monitoring and security space. The second one –

[00:47:58] JM: Because they all have Envoy plugins.

[00:47:59] DS: They all have Envoy plugins, or in some cases, like X-Ray. We upstream the plugin.

[00:48:03] JM: Oh, cool.

[00:48:04] DS: Yeah. The second part is Cloud Map, which is our service discovery service from AWS. Essentially think of it as you can register any arbitrary resource into it. App Mesh

uses Cloud Map to say, “Hey, this is service foo. It wants to talk to service bar.” Foo and bar are registered into Cloud Map. But bar could also be a DynamoDB table in theory. Suddenly your mesh knows that this application depends on this DynamoDB table over here. In the future, we don’t do it yet, you could have ALBs and API gateways as part of this mesh as well. Your mesh is just all these things that communicate with each other. The goal of App Mesh is to make sure you have this consistent dynamic configuration. App Mesh also has safe deployment practices built into it. Has deployment APIs. It can do weighted traffic, rollback. All of that’s built into the mesh, because we assume people are going to use it that way.

[00:49:02] JM: Are there any interesting places of overlap between the increase in event-driven architecture and how you have built App Mesh?

[00:49:13] DS: I don’t think so. I think App Mesh assumes that – What App Mesh assumes is two things want to talk to each other. They need permissions to talk to each other. They need policies that define is they’re allowed to talk to each other, and they need visibility into that communication. App Mesh focuses on providing the configuration. Are things allowed to talk? How are they allowed to talk and what kind of visibility can we give you? In many ways, for me, something like X-Ray is a feature of the mesh. It’s the map. It gives you all the communication. I think that’s how we think about it.

So if you have events coming in, they’re still part of this network that you are using and you manage and control, communicate over. So I do think event-driven patterns are super interesting, but I don’t think that’s almost orthogonal to at least – The goal of App Mesh it’s your fabric that you’re deploying your applications into.

[00:50:04] JM: Yeah. Okay. That’s what I thought. So not to make you repeat yourself, but can you go a little bit deeper into the distinction between the service mesh idea and the App Mesh idea?

[00:50:17] DS: I think in some ways, they’re similar. I think App Mesh is a little more broad, where most service meshes, at least the way people talk about them, are very constrained, and I have a container scheduler. I have services running inside that. I’m installing a proxy next to it.

We think of the network proxy, Envoy, as just one component of the data plane that App Mesh will manage.

[00:50:40] JM: Okay. I see. Because you have these abstractions like S3, or ELBs, and these things are not like addressable containers, like most of the things that you want instrument with your service mesh.

[00:50:53] DS: Exactly. So you want to be able to instrument ELB if you're getting ingress traffic and authenticate over there. The part that we don't have in App Mesh but it's in roadmap for App Mesh is public server. You can go look at it as having end-to-end encryption and mutual TLS and very, very fine grained policy management. So as a security operator, it becomes a very powerful tool in your portfolio. So I think that's where we get super excited.

[00:51:22] JM: I've had some conversations around that topic, where like one advantage is if you have the sidecar that's doing policy management, the policy lookups for your application are much cheaper, because otherwise you would have to go out to some central policy manager. Is that main advantage?

[00:51:41] DS: That is one of the advantages. In client side, at its simplest, your sidecar is a client side load balancer and it is a local agent that has cache of what policies you allowed, that we allow to talk to. The goal of the control plane for App Mesh is just to keep reconfiguring it and making sure it has the right policy attached to it. Then your app was only talking to this thing right next to it. It is one of the big advantage. I don't think it's the only one.

[00:52:09] JM: What are some other security benefits?

[00:52:11] DS: I think the main one is you can apply a policy at a very, very fine grained level. Today, your roles are applied at an infrastructure level, not at the application level as much. Now you're applying very find-grained policy at an application level, which means that at the infrastructure level you don't need to necessarily go in and apply very detailed policies, because you're managing your traffic – It's much easier to reason about if you're doing it at the application layer. I think those are some of the advantages.

It's not just about performance and speed. It's also about how you reason about it and simplicity, especially in a world where you're doing devops. I like to joke how many developers understand [inaudible 00:52:49] and all the details of network infrastructure. Not that many. A few, not that many.

I think on the other hand, they all understand application behavior. I think that's one of the biggest advantages. But you also have somebody who cares about compliance, etc., being able to define those policies. So I think it's a great way to build robust, scalable systems that are very developer friendly, but also are resilient and allow the compliance people to be effective as well.

[00:53:18] JM: So we're wrapping up. I talked to you for a long time, but because we're at the AWS Summit in Sta. Clara, can give you me your brief state of AWS address? What's new today? Where is the organization going? What are you excited about? What kinds of new customer trends are you seeing?

[00:53:41] DS: Lots to be excited about. I'll pick a few macro things. I think we are definitely seeing – We got modern applications where even people are building applications today that tend to be either targeted at something like Fargate or Lambda, container in general, Fargate specifically and Lambda. So most serverless models with your state living in a managed system like – Especially now that we have so many databases that are purpose built for a particular use case.

These decomposed, decoupled architectures, they used to be the internet savvy sort of hip companies doing it. Now, enterprises are doing it. Along with that is coming this razor-focused and success desire to be very good at CICD and automated deployments. I will almost say that is the first start of it before you do anything with microservices or fancy, you want to do CICD, and we see our customers really embracing that.

So automated deployment, so the CICD world using serverless for building applications, especially as microservices. Huge! We see that across the board regardless of what type of customer you are. The other part is – This is not a surprise, is the growth of machine learning as a competency. The tools have become so much more powerful, so much more accessible. You don't have to have a PHD in computer science from a fancy university to be a machine learning

scientist. There are tools available, frameworks available that will get you going really quickly. Then you have things like purpose built services, like Transcribe, and Comprehend that solve so many problems for you.

So then you combine those three that you've seen, at least from where I'm sitting, the things I look at. Those are the trends that are just fascinating to see evolve. Machine learning is especially interesting, because it sort of marries things. It marries batch computing and super-computing with big data in very interesting ways.

So those are trends that and I think I see I'm super excited about, and I think the last one, and I'll just call it out, is our customers are getting – Customers used to build with the lift and shift and then build new applications. I think they started fundamentally rethink how they build applications, because the tools are available to them. They're much more willing and want to run these optimized systems and they're happy to let AWS make decisions on their behalf, where five years ago, they may not have been. But the tools have reached a point where they're much more comfortable doing it, and that allows us to build more interesting tools and services for them.

[00:56:22] JM: Deepak Singh, until next time.

[00:56:24] DS: Thank you very much.

[END OF INTERVIEW]

[00:56:29] JM: GoCD is a continuous delivery tool created by ThoughtWorks. It's open source and free to use, and GoCD has all the features you need for continuous delivery. Model your deployment pipelines without installing any plug-ins. Use the value stream map to visualize your end-to-end workflow, and if you use Kubernetes, GoCD is a natural fit to add continuous delivery to your project.

With GoCD running on Kubernetes, you define your build workflow and let GoCD provision and scale your infrastructure on-the-fly. GoCD agents use Kubernetes to scale as needed. Check out gocd.org/sedaily and learn about how you can get started. GoCD was built with the

learnings of the ThoughtWorks engineering team who have talked about building the product in previous episodes of Software Engineering Daily, and it's great to see the continued progress on GoCD with the new Kubernetes integrations. You can check it out for yourself at gocd.org/sedaily.

Thank you so much to ThoughtWorks for being a longtime sponsor of Software Engineering Daily. We are proud to have ThoughtWorks and GoCD as sponsors of the show.

[END]

