EPISODE 1054

[INTRODUCTION]

[00:00:00] JM: Serverless computing is a way of designing applications that do not directly address or deploy application code to servers. Serverless applications are composed of stateless functions as a service and stateful data storage systems such as Redis or DynamoDB. Serverless applications allow for the scale up or scale down of an entire architecture, because each component is naturally scalable, and this pattern can be used to create a wide variety of applications. The functions as a service can handle the compute logic and the data storage systems can handle the storage. But these applications do not give the developer as much flexibility as an ideal serverless system might. The developer still needs to use cloud-specific state management systems.

Vikram Sreekanti is the creator of CloudBurst, a system for stateful functions as a service. The reason that stateful functions as a service don't really exist today is because we don't know a whole lot about how functions as a service are being spun up on cloud providers. We know that we send our code to the cloud and it gets scheduled on to a server and then that code runs at some point on some server, but we don't know a whole lot about the durability of that server and how stable it is, if it's going to fall over, if the code is going to need to be rescheduled? That's why we don't think of these things as stateful or reliable.

CloudBurst is this project from Vikram Sreekanti and it's architected as a set of VMs that can execute functions as a service that are scheduled on to those VMs. Each VM can utilize a local cache as well as an autoscaling key value store called Anna, which is accessible to the CloudBurst runtime components.

Vikram joins the show to talk about serverless computing in general and his efforts to build stateful serverless functionality. His work comes out of the RISELab, which we have done a few shows on before. It's a computer science lab in Berkley.

[SPONSOR MESSAGE]

[00:02:09] JM: When I'm building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust. Whether you are a new company building your first product, like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals.

Go to softwareengineeringdaily.com/g2i to learn more about what G2i has to offer. We've also done several shows with the people who run G2i, Gabe Greenberg, and the rest of his team. These are engineers who know about the React ecosystem, about the mobile ecosystem, about GraphQL, React Native. They know their stuff and they run a great organization.

In my personal experience, G2i has linked me up with experienced engineers that can fit my budget, and the G2i staff are friendly and easy to work with. They know how product development works. They can help you find the perfect engineer for your stack, and you can go to softwareengineeringdaily.com/g2i to learn more about G2i.

Thank you to G2i for being a great supporter of Software Engineering Daily both as listeners and also as people who have contributed code that have helped me out in my projects. So if you want to get some additional help for your engineering projects, go to softwareengineeringdaily.com/g2i.

[INTERVIEW]

[00:03:59] JM: Vikram, welcome to the show.

[00:04:01] VS: Thanks, Jeff. Thanks for having me.

[00:04:02] JM: We've done many episodes about functions as a service. We've talked about the basic engineering concept. Give a brief overview for how functions as a service are used today.

[00:04:14] VS: Yeah. The core idea behind functions as a service is that the application developers shouldn't have to worry about thinking about servers, how they're configured, where they're running and all that kind of stuff. The idea is that the application developer can write some code, whatever their business logic is, upload it to a service like AWS Lambda or Google Cloud Functions and configure some trigger events. Let's say, put it behind an API gateway or maybe configure it to run whenever an object is uploaded to S3, something like that. Then the cloud provider is responsible for taking care of when the function runs. Where it runs? Making sure that security is taken care of and all of that. Then the developer is only built for the amount of time that the code is actually running down the millisecond.

Today, what commercial functions as a service systems are really good at is taking care of things like event processing that doesn't require a lot of state or things that aren't really latency-sensitive. Some of the folks that we've talked to in our research have focused on human in the loop interactions where there is a relatively large latency budget that they can be dealing with so that they can take advantage of the flexibility that the service offers while also providing reasonable guarantees to their end-users.

[00:05:36] JM: What are the shortcomings of functions as a service systems today?

[00:05:41] VS: Yeah. The things that they really haven't focused on thus far are things that have nontrivial amounts of state and things that require relatively tight latency budgets. The reasons for those are obviously only known to the folks who are working at some of these major cloud providers, but I think in some sense, one of the core reasons, because these are nontrivial problem. It's not clear what the right way to think about state in a serverless world is and how while providing strong isolation security and so on, you can provide the latency access to state and make it really easy for programmers to deal with those things.

Even doing really simple things, things that might not even sound stateful, like doing function composition, for example, is pretty prohibitively slow on a lot of fast systems today. In some of our initial benchmarks we found that just running two arithmetic functions like squaring an integer and then incrementing it, it takes as much is 40 to 50 milliseconds per invocation. If you put those in two different functions, you've already paid 100 milliseconds. The sort of rule of

thumb for you interactivity on websites is couple a hundred milliseconds. If doing no computation takes 100 milliseconds, you've used up a lot of your latency budget already.

Those type types of things are already difficult. Then you start thinking about how you could do more complicated things like modifying shared state across multiple parallel requests. The state-of-the-art for serverless infrastructure today is to basically throw a bunch of data in a storage system like Dynamo or S3 and rely on the application developer to resolve inconsistencies. Because as we all know, using these system like Dynamo and S3, you don't get very strong consistency guarantees out of them. So you sort of are in the Wild West. It's up to you as the application developer to make sure that your updates don't accidentally clobber each other that if you're reading separate pieces of data that they were written in some consistent format, whatever that means for your application, and so on. There are a very few guardrails in this space.

[00:07:46] JM: What do we know about the implementation of AWS Lambda? Because AWS Lambda is the most widely used, the most mature functions as a service platform. Do we know anything about how it's actually implemented under the hood?

[00:08:02] VS: We know a little bit. Recently, Amazon has started to be more and more open about what their core infrastructure is composed of. About a year ago or maybe a little more than that, they open sourced this platform called Firecracker, which is a micro VM hypervisor that basically they've built from the ground-up, they basically wrote a hypervisor in Rust to reduce spin-up times for what's called the cold start problem for FaaS systems.

Cold start problem is basically a new request comes in. There are no resources allocated for that particular piece of code to run. They have to figure out which of those resources be allocated. They have to go set up an environment. They obviously have environments for many different languages and common runtimes. They have to download the code. Set it up in the right VPC, all of that stuff. Traditionally, that cold start time in the initial releases of Lambda could be multiple seconds long.

What Firecracker has done that's really awesome is to bring a lot of that down. Obviously, there still some speed of light constraints here if you're uploading hundred megabytes of code or

something. It's going to take a little bit of time to download. But as far as the core infrastructure goes, they have the spin up times for the individual VMs that user code is running down to about 100 milliseconds believe. We know a little bit about the sort of core infrastructure. But in terms of what the actual sequence of AWS or otherwise components that their requests are going through, it's pretty opaque as far as I'm aware.

[00:09:33] JM: You mentioned this idea of function compositions, and this might be a use case where I would want to have one Lambda function spin up. Call another Lambda function, which calls another lambda function. I mean, in programming, we have nested function calls all the time. I make some high-level method, and that high-level method calls a lower-level method, and that might call two lower-level methods. We want to have this nesting system of function calls. I think more abstractly, we might want to have these complicated long-lived workflows where you have 20 different Lambdas talking to each other and orchestrating some long-lived session.

You could imagine, maybe if I want to build a real-time strategy game or an MMO on top of Lambda, that's pretty hard to do today. I don't really have the durability of my processing system that I think I need for like an MMO. Why is that? What are the issues with having this long-lived compositional compute with Lambda?

[00:10:46] VS: Yeah, it's a really good question. I think a lot of it has to do with the core conception of the system. In some sense, if you're talking about long-lived compute that has probably some state tied up with it. You can imagine that if you're building a game, the environment, and the characters, and the objects, whatever the objects of the game are, probably have some state associated with them that they're responsible for managing. A lot of that starts to sound very little like a function and more like something like an actor.

If you talk about a functions as a service system, it becomes a little difficult to envision at least at first cut how you might manage the statefullness, how you might manage the update to that state? Where that state is actually living? How you deal with fault tolerance? If one of these machines with your function fails, then who's responsible for spinning it back up? Repairing state? Doing whatever is required for that? All of those things are not readily obvious when functions are the core programming paradigm.

One of the things that we've been thinking a little bit about is how you can sort of start with functions as the sort of entry point, because Lambda has become relatively popular in the last few years and people are getting more interested in the paradigm. But how we can also push that programming model forward with introducing state primitives? With introducing maybe even different programming paradigms like actors or like more stateful operators that allow you to think about who's responsible for the state and where that state is living. Who's updating it? And all those kinds of questions that aren't really addressed by the primitives in a traditional functions as a service system.

[00:12:27] JM: If we're talking about adding state management to a Lambda-based system, and we'll get into it to your approach to that stateful approach, that stateful computing, adding state to Lambda. The naïve approach might be I just want to use some in-memory cache. I want to use an object cache like Redis. I could use a database like DynamoDB, and this could manage my entire state. Why isn't that good enough? Why can't I just manage the state of my computer with these object storage systems?

[00:13:08] VS: Yeah. For some applications, it definitely is good enough. I think I mentioned a few minutes ago that some of the production applications that we've seen that are running on Lambda today are focused on really simple state models that are using systems like Dynamo, or S3, or Redis, whatever it may be, to store some intermediary state and to maybe communicate between functions when necessary. But there're some code constraints that you run into pretty quickly for a lot of common application. These are things like data shipping. Lambda doesn't really give you a facility to make sure that if you're accessing certain kinds of data really frequently, you don't have to ship it over the network every time. You can write certain objects or files/temp on Lambda. If you happen to hit the same Lambda executor repeatedly with the request, you can check to see if there's something in /temp. But that's a pretty hacky and a pretty unreliable way of trying to implement caching.

Then on top of that, it's not just data shipping, but there's concerns run consistency. If I'm using DynamoDB and I have multiple requests running in Peril that may be touching the same state for some reason, Dynamo has very loose consistency guarantee. You might think that a system

like Redis, which provides the [inaudible 00:14:26] at least on a single shard would be better, but Redis at least in the cloud native deployment isn't autoscaling, right?

You have a Lambda system random that's autoscaling. Load goes up. More executors come up seamlessly. But then with Redis, there has to be someone sitting there adding machines, removing machines. Every time you do that on Redis in cluster mode, there is a pretty high overhead because it reshuffles all of the data. All of these operations become really expensive and cumbersome and they sort of run against the grain of the simplicity and the ease-of-use that Lambda is really great at offering.

You start to wonder why you're using all these complicated techniques that the community have gotten good at in the context of non-serverless or server-full programming. But if the core goal of serverless is to use some of the more easy to use autoscaling, pay-per-use, etc., abstraction, then it starts to become pretty difficult to reconcile that with all these complicated scaling procedures and thinking about the way in which you're supposed to allocate resources to all the auxiliary components in addition to the compute.

[00:15:35] JM: Yeah. It's worth taking a step back here and asking what is the goal? What are we trying to achieve here? I mean, if I want long-lived compute and I want to have scalability, I could just use the traditional AWS stack. I can use EC2 instances. I can use Fargate. I can use autoscaling groups. I can set up a scalable system. Why do I want any of this? What are we actually trying to achieve here? If we're talking about building a stateful functions as a service platform, is there some developer experience ideal that we have in mind that we're trying to get to?

[00:16:22] VS: Yeah. That's a really great question. I think a lot of it goes back to thinking about who the end-users or the developers are. For a lot of folks who've been living in the cloud for the last 10, 15 years, a lot of these stuff like you're saying may sound a little silly. Why are we jumping through all these hoops and trying to reinvent the wheel when we already know how to use Kubernetes and horizontal pod auto scaling and all these tools that are pretty good at what they do. What's the goal here?

I think at least from our perspective, one of the really interesting questions is thinking not about who the developers are maybe today or two years ago, but who they will be in five years. At Berkeley, where I'm a grad student for example, we have recently introduced a whole major around data science. A lot of these folks who are studying data science aren't coming from traditional computer science software engineering backgrounds, but they are folks who are interested in a variety of other fields who are studying data science as more of an applied major to go learn about data analysis techniques and then apply them to whatever they're interested in.

Thinking about those kinds of folks in the next 5, 10, 15 years, there's going to be an increasingly common and increasingly large share of the programmer developer community. It's going to be really difficult for them to have to worry about setting up Kubernetes clusters every time they wanted to do analysis that doesn't fit on a single machine and to worry about how those things scale up and down and to worry about when they should be allocating resources to them or how to configure a pod auto scaler. Are those kinds of things are maybe not in the wheelhouse of people who aren't were traditional software developers.

In general, even for people who are familiar with more traditional server management techniques and autoscaling, I think there's an attractiveness to serverless, which comes from this idea that you only pay for what you use. You don't have to worry about these traditional techniques of provisioning for peak load and worrying about how you're going to reallocate resources during the not peak load times and whether you should be running back jobs on them in the background that can be preempted just in case load spikes, and all these kinds of things that some of the large web companies have gotten really good at doing but maybe aren't what the average developer should be worrying about.

If we can do a really good job with bringing state management to serverless, if we can sort of help developers rethink the way that they are configuring their servers and building their application, fitting all these pieces together, it will actually make everyone more productive. They won't have to worry about mucking around with Kubernetes policies and building their Docker containers and doing all these things that maybe aren't core to the business but are necessary to be running applications today. But the question is, is that necessary in the future? The goal or

the hope of serverless is that we can sortof simplify these attractions and make people more productive by removing some of these obstacles.

[SPONSOR MESSAGE]

[00:19:35] JM: As a company grows, the software infrastructure becomes a large complex distributed system. Without standardized applications or security policies, it can become difficult to oversee all the vulnerabilities that might exist across all of your physical machines, virtual machines, containers and cloud services. ExtraHop is a cloud-native security company that detects threats across your hybrid infrastructure. ExtraHop has vulnerability detection running up and down your networking stock from L2 to L7 and it helps you spot, investigate and respond to anomalous behavior using more than 100 machine learning models.

At extrahop.com/cloud, you can learn about how ExtraHop delivers cloud-native network detection and response. ExtraHop will help you find misconfigurations and blind spots in your infrastructure and stay in compliance. Understand your identity and access management payloads to look for credential harvesting and brute force attacks and automate the security settings of your cloud provider integrations. Visit extrahop.com/cloud to find out how ExtraHop can help you secure your enterprise.

Thank you to ExtraHop for being a sponsor of Software Engineering Daily, if you want to check out ExtraHop and support the show, go to extrahop.com/cloud.

[INTERVIEW CONTINUED]

[00:21:08] JM: What you're saying there's not that if we managed to build stateful functions as a service, this is going to solve the problem of the entry-level data scientist who wants to do processing across 80 terabytes of data without thinking about how to configure a Spark cluster. What we're thinking about is if we can make stateful functions as a service, this is going to be a very interesting infrastructure primitive that will allow platform developers to build the necessary platforms for allowing an entry-level data scientist to do processing across 80 terabytes of data.

[00:21:58] VS: Yeah, that's a really good clarification. I think the goal here at least from our perspective is not to solve every problem that you can imagine by building a stateful FaaS system. But the idea is that especially in the last 15 years, we're seen that open-source, whether it's systems like Spark or Pandass more recently have gained a lot of popularity because they're new paradigms or maybe improvements of existing paradigms that allow people to process data to do things that they weren't able to do before really easily and quickly.

The goal here isn't to say that we're going to build system and everyday a scientist is going to use exactly that system. But like you're saying, the infrastructure that we're building we hope will provide a blueprint for enabling people who come along in the future to say, "Hey, I have this really interesting way to think about processing data in a serverless fashion. Can I layer on top of a stateful FaaS system so now my end-users can come along and just write code against this API, and I will take care of distributing it and sharding it and making sure that there are the write guarantees and the write scaling properties and all that kind of things.

A really good example of that is actually another project in the lab that I worked in called Modin, which is an API that sorts of sits in between the Pandas API and distributed processing frameworks. What the folks on the Modin project have done is basically to decouple the Pandas API, which data scientists and data analysts are super fluent with from the underlying infrastructure where it runs. They've really thought hard about how to structure data frames under the hood and what the data layout should be and all that kind of thing that the average data scientist doesn't think about.

Now what they're trying to do is to layer that on top of a bunch of distributed data processing frameworks so that data scientists using Pandas can write code against the Pandas API that they know. Then under the hood, maybe switch a config or something and then have it automatically run in a distributed fashion so they don't have to worry about memory constraints and shuffles and sharding and all that kind of stuff but still get all the nice properties of scalability and distributed compute for free.

[00:24:12] JM: Your goal with CloudBurst – CloudBurst, by the way, I should say is this paper that you published. It's about a stateful serverless platform, which is what we've been talking

about in the abstract. Tell me about why you first started working on the problem of building stateful serverless computing.

[00:24:36] VS: Yeah. My research group sort of has a database background. We all sort of have studied databases and traditional database techniques and we got sort of excited about cloud infrastructure 4 or 5 years ago just trying to think about the level at which you can think about scale and the nice properties that you can rent machines. Not have to worry about resources, all those kinds of things, and do things with relatively high ease-of-use. Serverless sort of just accelerated that excitement when we started thinking about all the different ways in which we can make programming easier like we've been talking about for the last few minutes.

What we realized was when we took a sort of deeper look at serverless from an infrastructure perspective was that there's all these things that it's pretty good at, like event processing for web applications or handling orchestration across a bunch of existing services. That's actually a really common use case. If you look at the AWS Lambda use cases, is that they use Lambda functions to basically trigger events into other services to write things to data warehouses and so on.

What we realized was that if you really wanted to push that forward and to think about how we could restructure the way that developers are going to be writing code building platforms against cloud infrastructures sort of like we've been talking about using data science as an example, serverless didn't really fit the bill. There weren't really abstractions. There weren't really ways to structure the infrastructure to take advantage of all of those nice properties of serverless without having to jump through all these complicated hoops around, like we've already said, consistency and data shipping and so on.

The reason that we started tackling stateful functions as a service was it felt like a concrete way to make cloud programming easier with all the benefits of serverless but while also improving guarantees around consistency and performance and data locality with the sort of nice APIs that already come with serverless.

[00:26:46] JM: Okay. As we talk about the structure that you built with CloudBurst, the architecture, I first want to talk about the design goals. You set out with some design goals for

the problems and the features, the problems you want to alleviate and the features that you wanted out of CloudBurst. One key design goal was logical disaggregation with physical colocation. What does that mean? Logical disaggregation with physical colocation?

[00:27:21] VS: Yeah. Something that we haven't touched on from the perspective of existing functions as a service system, this idea and disaggregation, which I think becoming increasingly popular in a lot of cloud architectures. The idea is that cloud providers can gain a lot of efficiency by logically disaggregating computer and storage, which means that there is a computer service that's running on these machines over here. There is a storage service that's running on those machines over there.

You can imagine a simple example might be S3 and Lambda. When you spin up the Lambda function and you run it, you don't know who else is running Lambda functions on the same physical machine that you're running the Lambda function on. All you know is that you have this core for the next a few hundred milliseconds and you're going to do recomputed, and it's on AWS to make sure that there is no security vulnerabilities or data leakage that happens across that.

Similarly, when you put an object into S3, on the same physical hard drive, there may be 100 other users who have objects living there. The nice thing about that from the cloud provider's perspective is that they can use fewer resources and really aggressively pack users into this multi-tenanted environment. If they could do that, then they can also turn around and provide lower cost to users because they're using fewer resources to service the same number of requests. That's this idea of disaggregation.

We felt that disaggregation was a really nice feature. We don't want to remove disaggregation because for all the reasons that I just talked about. The idea was that having this logical disaggregation, saying that we're going to have some resources where computer runs and resources where storage runs and you don't have to worry about who else is living on those same physical resources. We'll take care of that for you. That shouldn't preclude having physical colocation, which is always been sort of a core tenant in data systems, is that we don't want to be shipping data all over the place over the network especially repeatedly. If we can achieve physical locality for data accesses, that will improve performance, but we also want to make you

that we have disaggregation because it provides these nice properties for both the cloud provider and for the program.

[00:29:36] JM: Right. It almost sounds like – I mean, when Lambda first came out, the way that I thought about it was, “Okay, these are just functions that people are using as glue code. They're getting scheduled on to this questionable server infrastructure. The server might fail at any time. Maybe these are old servers. Maybe this is just spare capacity on a dusty old Amazon server, this dedicated EC2 instances. It's about to fall over, and you don't get really strong guarantees about whether your Lambda function is going to finish. When it's going to spin up? It’s this flaky function.”

As Amazon, you think of Amazon as just scheduling these functions on to random servers throughout their infrastructure. Of course, we don't know. But here you're saying if we can have these dedicated blocks of infrastructure that we’re scheduling functions on to, at least the stateful kinds of functions where we’re going to want to scale it up to many, many instances of functions and perhaps large amounts of data, we actually want larger dedicated quantities of infrastructure for these serverless function. It almost sounds like a move towards an entirely different conception of how the infrastructure should be managing the serverless functions.

[00:31:06] VS: Yeah, that's a really good point. I think we’ve already seen more and more of a move to this with hosted infrastructure services, things that are sometimes referred to as backend as a service system. If you look at AWS, a good example might be a system like Athena, where you can run arbitrary SQL queries over structured or semi-structured data that’s stored in S3.

What’s that effectively done is it's built a disaggregated system. S3 is running on some machines over here. Athena is running over there, and your API is just a SQL query. It is serverless in the sense that you upload your SQL query and you basically pay for, I believe, the number of gigabytes of data that's processed and maybe how long the query takes something like that. You don’t have to worry about how the data is moved. Maybe Amazon under the hood is being really smart and is pushing some of those compute cycles down on to the same machines that are running yesterday. Maybe they're not. We don't know. But at the end of the day, we don't have to care about it.

The nice thing about this disaggregated architecture is that you pay for the data that's stored in S3, and that's one service that you worry about. Then if you want to run SQL queries over that, you pay for the data that's processed in Athena, and that's a separate service that you worry about. You don't have to think about I'm going to write some data to S3 and then I'm going to spin up an EC2 instance over here and I don't have to worry about the get requests from S3 and how much they cost and how long that's going to up my EC2 bill. All that stuff is sort of abstracted away from you because we can disaggregate the services that are doing storage and compute and allow you to manage each one of those separately.

[00:32:51] JM: To run through some of the other concepts, the design goals of CloudBurst before we get into the architecture. You wanted distributed session consistency. You wanted users to be able to compose functions together. You wanted the ability for functions to communicate with each other. In order to get these features, you needed some state management and you needed storage semantics out of your functions as a service runtime. Tell me about the requirements for state management, the requirements for storage and what you did to implement that.

[00:33:35] VS: Yeah. I think you mentioned distributed session consistency, and I think it's a really important idea in the broader picture here ,because if you start just doing simple function compositions and think about how you might do it in Lambda, let's say you're just running two functions each of which is going to read some data from a storage system. Let's say you're using DynamoDB, because that seems to be a common sort of component in architectures that use Lambda. Let's say that your first function might read some data from Dynamo, do some processing then trigger the second function. The second function happens to read the same piece of data.

With DynamoDB's default consistency guarantee, you are not guaranteed that the same logical request which happened have two functions are going to read the same piece of data. It might read X twice and read version 1 the first time and version 2 the second time, and that makes it really difficult for programmers to reason about what's happening in the world. Did the data change or did function F happened to read the stale version? How do you even know what the versions are? Dynamo doesn't really provide an API for you to understand what's newer and

what's older. It's on the application to maybe write that into the payload, for example. All of these things become really thorny questions really quickly and you're sort of forced into hacks like having a versioning schema on top of Dynamo or maybe explicitly shipping data that you require consistency for between functions. Whatever it is that you need to do as the application developer to make sure that there's sort of a sane view of the world from the perspective of your code.

Having distributed session consistency, you write as a sort of storage-esque primitive, but it's really important for the sort of sanity of the programmer to be able to say, "I have a function over here. It's going to execute." Then as a part of the same request, another function is going to execute, and I want to make sure that there's some consistency guarantee across the two functions that execute. From a database perspective, that guarantee that I just mentioned is often called repeatable read. I read a key once. I read it later on in the same request. I should see the same version of it.

In terms of what the requirements are from a storage system, you can imagine schemes that would build maybe some version of these guarantees across multiple storage engines. We actually built on top of an existing key value store that we had developed as a part of our research called Anna, which is a really low-latency autoscaling key value store that avoids all coordination techniques.

The way that it does this is by using these data structures called lattices or CRDTs that provides semantics for automatically merging conflicting updates without any user intervention. A simple example would be a set. if I have two replicas of a set, they can accept updates in Peril, and then asynchronously they can tell each other what values are in the set. Because of the semantics of a set, once an object is put into the set, we can just say that it should be inserted into the other replica of the set as well and we can asynchronously resolve and combine those conflicting updates.

There's a bunch of data structures in the research literature that have taken advantage of this property that sometimes summarize as like associative, commutative and item-potent. As long as your conflict resolution satisfies those three properties, we can asynchronously resolve conflicting updates without any user intervention.

The reason that we decided to use Anna as the sort of underlying storage substrate to help provide these semantics is because it sort of fits with the standard mold of some of these autoscaling cloud systems like Dynamo and S3 and that it doesn't have strong consistency. It doesn't use sort of really restrictive coordination techniques, but it provides reader semantics that helped us build some of these distributed session guarantees on top of it without requiring us to build a versioning scheme on top of a system like Dynamo or really think about where data lives and how we should be managing conflicting updates and all those kinds of things.

Using Anna as sort of the storage substrate was a really nice way to sort of delegate some of the conflict resolution and consistency mechanisms that it already took care of and then we were able to layer protocols on top of it and the CloudBurst layer to provide this notion of distributed session consistency.

[SPONSOR MESSAGE]

[00:38:14] JM: Over the last few months, I've started hearing about Retool. Every business needs internal tools, but if we're being honest, I don't know of many engineers who really enjoy building internal tools. It can be hard to get engineering resources to build back-office applications and it's definitely hard to get engineers excited about maintaining those back-office applications. Companies like a Doordash, and Brex, and Amazon use Retool to build custom internal tools faster.

The idea is that internal tools mostly look the same. They're made out of tables, and dropdowns, and buttons, and text inputs. Retool gives you a drag-and-drop interface so engineers can build these internal UIs in hours, not days, and they can spend more time building features that customers will see. Retool connects to any database and API. For example, if you are pulling data from Postgres, you just write a SQL query. You drag a table on to the canvas.

If you want to try out Retool, you can go to retool.com/sedaily. That's R-E-T-O-O-L.com/sedaily, and you can even host Retool on-premise if you want to keep it ultra-secure. I've heard a lot of good things about Retool from engineers who I respect. So check it out at retool.com/sedaily.

[INTERVIEW CONTINUED]

[00:39:51] JM: I want to revisit Anna, this autoscaling key value store a little bit later. But let's talk about the architecture for CloudBurst at this point. A CloudBurst cluster has multiple VM's and functions get scheduled on to those VM's. Each of those VM's can run some number of functions and these functions can use local cache and there's also this autoscaling key value store that can be used by the entire CloudBurst runtime.

First of all, the VM's. The CloudBurst cluster has a bunch of VM's on it and functions that get issued to the entire CloudBurst cluster get scheduled on to those VM's. What happens when a function gets scheduled on to a VM? What does the VM do?

[00:40:53] VS: Yeah. What the VM basically does is that it reads in the code from the storage engine. Ore initial implementation of the wall system is in Python. So it reads in a pickled Python function. It de-serializes. It checks to see if any of the arguments to that function are of a special reference type. If any of the arguments are references, it basically treats those as KVS keys. Whatever keys are stored in those references, it automatically sucks the amount of storage. If they're stored locally in the cache, then we just read things from the cache and don't have to go over the network.

Then we'll execute the function, passing in those arguments and the references are automatically resolved. Then once the code finishes executing, it either will trigger a downstream function if there is a downstream function or it'll write the result to storage or optionally it can also just send the user synchronously a result back to the requesting client.

[00:41:53] JM: Each of these VM's has a set of executors. Can you explain what an executor is and what role an executor plays in the function scheduling?

[00:42:06] VS: Yes. An executer is basically a single thread that does the process that I just walked through. Each thread has a separate ID. The scheduler will basically look at all of the threads in the system. See how much work it's assigned to each of them recently. It uses some heuristics based on data locality, based on load management, those kinds of things. It picks an executor thread to assign the function request to. Then once the executor receives that request,

it does the fetches and the execution and then returning to the user. That whole process is done by the executor thread.

[00:42:43] JM: Each of these VM's also has a cache. What kinds of data is being stored on the local cache of the VM that – Again, these VM's are accepting the functions that are getting scheduled on to them and then they're executing those functions. What does the local cache in the VM do?

[00:43:07] VS: The cache is responsible for intermediating between all of the requests between the functions themselves and the storage engine. The functions, as a part of their code, can do arbitrary writes and reads from the storage engine. Whenever a read or write is issued, the cache sort of sits in between and says, "If I have this data stored locally, then I'm just going to return it automatically from the cache. If not, the I'm going to suck it in from the storage engine. I'm going to cache it locally and then return it to the client."

As we start doing more and more request, especially if data access is skewed in some way or there is some hot set of keys, it's likely that that data is going to be stored in cache. Whenever a function requests that data, we're just going to read it from cache and avoid that expensive network round trip that we were talking about early on in the conversation.

[00:43:58] JM: Okay. That's a clever design. Basically, the idea is you've got these different functions, and the functions might need to do some data sharing. They might need to reference variables that have been created or operated on by another function. If you need to read one of those variables and you don't have access to it in your VM's local cache, you can reach out to this autoscaling key value store, Anna, that is accessible from all of the different VM's that functions are getting scheduled on to. That's kind of like the disk of the overall CloudBurst computer that you've got there. Do I understand it correctly?

[00:44:43] VS: That's exactly right.

[00:44:46] JM: Now, does this lead to any cache coherence issues? The fact that you have these different functions as a service. They've got local caches that they're accessing and then

you've got this autoscaling key value store that they might need to reach out to. Can that lead to cash coherence issues?

[00:45:06] VS: Yes. In general, we have not thought about managing cache consistency as a problem that should require strong coordination just because it sort of introduces all these performance barriers that are really difficult to reason about. What we've basically done to take advantage of the coordination freeness of Anna and to leverage that to keep the caches roughly as up-to-date as we can expect the storage engine to be.

What we do is, under the hood, Anna sort of has a distributed multicast mechanism that allows us to keep all of the replicas of each key in sync with each other using those sort of associative, commutative and item-potent merge functions that I talked about earlier. The way that works is that if there is three replicas of a key in Anna, all of them can update themselves in parallel without any coordination, and then periodically, they'll all send each other any updates that they received for each key in the storage engine and use that merge function to make sure that they all do deterministically and up at the same value.

What we did was that the caching layer, we actually used the same CRDT or lattice data structures that Anna uses. The cache basically is treated as an extra KVS replica. Whatever updates are received at the KVS layer, the caches can actually subscribe to certain keys in the KVS. When the KVS performs its sort of distributed multicast protocol, it looks to see if any caches are subscribed to the keys that it's currently multicasting and it basically just sends an extra message in addition to the storage replicas to the cache to say, "Hey, here's a new version of the key that you might want to know about." If the cache still has that data cache, then it'll merge it in using that same merge logic. If it doesn't, then it will just drop the message and will treat it as just an empty message.

[00:47:07] JM: The implementation of CloudBurst, we're talking abstractly about VM's and caches on those VM's. You had to actually implement this. Did you implement it on a cloud provider?

[00:47:27] VS: Yes. Everything runs on AWS EC2. The underlying infrastructure uses Kubernetes and kops to interface with EC2, and we basically have our own Docker containers

the run. We have our own sort of autoscaling infrastructure that looks at load metrics and machine failures and all those kinds of things and uses sort of kops as an API to scale the whole service up and down in response to load changes and sort of standard heuristics or unload management.

[00:48:03] JM: What were the implementation difficulties in actually building this out?

[00:48:09] VS: I think the core sort of implementation challenges that we ran into was sort of layering the infrastructure in a way that allowed us to be as a sort of seamless as possible akin to the way that existing functions as a service systems are while also sort of dealing with the constraints of existing serverfull cloud infrastructure.

Sort of making sure that every time that we want to autoscale, for example, we maybe don't want to wait the three minutes it might take for an EC2 instance to come up and then the extra two minutes that Kubernetes takes to download all of the infrastructure containers that it runs and then download our containers and all those kinds of things. Finding sort of the right ways to manage the infrastructure. Make sure that we're sufficiently nimble but aren't wildly over allocating resources. We are working on a research budget after all. So we can't just leave tons of machines lying idle all the time if we're not doing anything with them. Trying to find a balance around sort of managing the severfull infrastructure and how we can translate that into serverless infrastructure. Keeping in mind of course that we're not running sort of a large-scale multi-tenanted production system that tons of people are using every day but sort of are doing this in their research context. Trying to navigate sort of all those kinds of things has been a really interesting challenge.

Another one that sort of comes to mind is dealing with moving data across the network across multiple languages. Anna is implemented in C++. The caching layer is as well just because we wanted to take advantage of the same data structures, but the programming interfaces in Python. Anyone who sort of worked on distributed Python will know pretty intimately that paying for serialization and deserialization in Python is really expensive and really, really slow. Trying to think about how we can mitigate some of those overheads. Maybe add caching in addition to in sort of on a per VM basis. Also on a per executor basis so that we can sort of de-serialize versions. Reducing overhead of de-serializing functions on every request. All those kinds of

things were things that we sort of had to work through at a fine-grained and sort of with a fine-tooth comb to make sure that we were reducing the overheads as much as possible, because it started to add up pretty fast when we didn't consider these things.

[00:50:37] JM: The prospect of productionizing this, it seems realistic. You could imagine offering what you have built with CloudBurst as a function as a service company. You could imagine, if it works as you intended, selling compute time on a CloudBurst cluster. Is that a realistic possibility?

[00:51:04] VS: Yeah. I mean, we're still trying to finish grad school. So we haven't really thought super-hard about what a commercial offering of this would look like, but we have been starting to play around with a bunch of applications that run on top of it to get a feel for what the sort of sharp edges are here and where the applications that run on top can really derive benefit from some of the abstractions that we've been providing.

One example is an application that I already mentioned, which is the distributed Pandas infrastructure that we've been working with with folks in our research lab to implement on top of CloudBurst. We've also been thinking about how to make data science easier in general ,but building a serverless backend for Jupyter Notebooks to basically enable folks who are writing code against Jupyter to not worry about where the state is living and maybe even scale up their workload seamlessly without having to worry about where Jupyter hub is running and what the resource constraints on all of that infrastructure is, but to sort of get a more serverless feel and say, "Hey, for this particular cell, I want to scale it up really fast," and then the rest of the time I just want to use one thread to run my code or something like that.

Sort of as like a compromised answer to question, we've been thinking about some of the more concrete applications that we can layer on top of this and derive really interesting benefits from the model that we implemented with CloudBurst, but we haven't really thought yet about what some of the more commercial applications of it might be.

[00:52:33] JM: Have you talked to anybody at AWS about CloudBurst?

[00:52:38] VS: Yeah. The nice thing about our research lab is that we're actually sponsored by all of the major cloud providers. We get to chat with them couple times a year and get their feedback on what we've been working on. I think they are definitely interested in the ideas. They see these ideas as I think interesting future directions for some of their services. But at the end of the day, they are running a multimillion dollar production service. So they can't go running around implementing sort of newfangled research ideas. I think a lot of the work from my understanding that's gone into some of these systems in recent years has been really focused on hardening the system. Making sure that it provides reasonable guarantees, for example, cold start latencies like I mentioned earlier and to really sort of tighten the nuts and bolts around the core offering. But I'm sure they're thinking about what the future directions of these services are and have definitely shown interest in sort of learning more about what we've built and how we think the ideas might be more applicable in their setting.

[00:53:49] JM: You're part of the RISELab, and I've done a few shows with people from the RISELab. Why are the RISELab people so interested in serverless? I'd love to know just what your experience like has been? What your experience has been like at the RISELab?

[00:54:13] VS: Yeah. To maybe answer this question backwards, the RISELab is super unique from my perspective, because it has a really strong research background obviously, but also has a ton of contact and interaction with industry that allows a lot of the researchers in the lab to choose problems that I think are really applied and they're really focused on moving the needle for problems that folks in the industry are running into.

I think that sort of perspective, that close interaction that we get just by virtue of being in Berkeley and being really close to the Silicon Valley in San Francisco and having those interactions on a somewhat regular basis sort of tempers and motivates a lot of the research that we do.

I think one of the reasons that folks have gotten really interested in serverless, not to be super repetitive here, is really starting to think about how we can bring some of the simplicity of serverless to a broader variety of applications because we are starting to see more and more people who don't maybe have traditional computer science backgrounds, writing code and

wanting to deploy code, but not knowing what the right abstractions and the right infrastructure to layer on top of is.

Serverless provides a really neat way to sort of simplify the lives of those people. That's why I think a lot of the interactions, conversations that we've had have moved us in the direction of serverless and have pushed us to think about simpler abstractions for cloud programming that don't require the complexity of servers and Kubernetes and all these kinds of things that people are dealing with today.

[00:55:57] JM: What would be your research focus? What would you be working on if you are not working on serverless right now? What kinds of problems are you thinking about?

[00:56:09] VS: Yeah, that's really interesting question. I think I got interested in cloud infrastructure before I got interested in serverless. I think if I wasn't working on serverless, I would still sort of be in the general space of thinking about neat ways to take advantage of the scale and the sort of resource availability of the cloud was really possible before the advent of systems like EC2 an S3 that operated this massive scale that people were really dealing with before.

I think finding ways to take advantage of that and bring developer guarantees, whether it’s around consistency or fault tolerance. Making those things more seamless, which I know is starting to sound a lot like serverless, but I would say that the reason that we came to serverless was because it felt like a vehicle for making those things easier rather than we looked at serverless and then decided that we should make those things easier, if that makes sense.

[00:57:05] JM: It does. Vikram, thank you for coming on the show. It's been a real pleasure talking to you.

[00:57:10] VS: Yeah! Thanks, Jeff, for having me. This has been awesome.

[END OF INTERVIEW]

[00:57:21] JM: Apache Cassandra is an open source distributed database that was first created to meet the scalability and availability needs of Facebook, Amazon and Google. In previous episodes of Software Engineering Daily we have covered Cassandra's architecture and its benefits, and we're happy to have DataStax, the largest contributed to the Cassandra project since day one as a sponsor of Software Engineering Daily.

DataStax provides DataStax Enterprise, a powerful distribution of Cassandra created by the team that has contributed the most to Cassandra. DataStax Enterprise enables teams to develop faster, scale further, achieve operational simplicity, ensure enterprise security and run mixed workloads that work with the latest graph, search and analytics technology all running across hybrid and multi-cloud infrastructure.

More than 400 companies including Cisco, Capital One, and eBay run DataStax to modernize their database infrastructure, improve scalability and security, and deliver on projects such as customer analytics, IoT and e-commerce. To learn more about Apache Cassandra and DataStax Enterprise, go to datastax.com/sedaily. That's DataStax with an X, D-A-T-A-S-T-A-X, @datastax.com/sedaily.

Thank you to DataStax for being a sponsor of Software Engineering Daily. It's a great honor to have DataStax as a sponsor, and you can go to datastax.com/sedaily to learn more.

[END]