EPISODE 910

[INTRODUCTION]

[00:00:00] JM: Ever since Kafka was open sourced from LinkedIn, it has been used to solve a wide variety of problems in distributed systems and data engineering. Kafka is a distributed messaging queue that is used by developers to publish messages and subscribe to topics with a certain message type. Kafka allows information to flow throughout a company such that multiple systems can consume the messages from a single sender.

In previous shows, we've covered the basics of Kafka as well as design patterns within Kafka, and Kafka Streams, and event sourcing with Kafka, and many other subjects relating to the technology. Kafka is broadly useful and new strategies for using Kafka continue to emerge as the open source project develops new functionality and becomes a platform for data applications.

In today's episode, one of my favorite guests, Tim Berglund, returns to Software Engineering Daily for a discussion of how applications are built today using Kafka, including systems that are undergoing a re-factoring, and data engineering applications, and systems with a large number of communicating services.

If you're interested in learning more about how companies are using Kafka, the Kafka Summit in San Francisco is September 30th through October 1st, and companies like LinkedIn and Uber and Netflix will be talking about how they use Kafka. Full disclosure; Confluent, the company where Tim works, is a sponsor of Software Engineering Daily.

[SPONSOR MESSAGE]

[00:01:39] JM: You probably do not enjoy searching for a job. Engineers don’t like sacrificing their time to do phone screens, and we don’t like doing whiteboard problems and working on tedious take home projects. Everyone knows the software hiring process is not perfect. But what’s the alternative? Triplebyte is the alternative.

Triplebyte is a platform for finding a great software job faster. Triplebyte works with 400+ tech companies, including Dropbox, Adobe, Coursera and Cruise Automation. Triplebyte improves the hiring process by saving you time and fast-tracking you to final interviews. At triplebyte.com/sedaily, you can start your process by taking a quiz, and after the quiz you get interviewed by Triplebyte if you pass that quiz. If you pass that interview, you make it straight to multiple onsite interviews. If you take a job, you get an additional $1,000 signing bonus from Triplebyte because you use the link triplebyte.com/sedaily.

That $1,000 is nice, but you might be making much more since those multiple onsite interviews would put you in a great position to potentially get multiple offers, and then you could figure out what your salary actually should be. Triplebyte does not look at candidate's backgrounds, like resumes and where they've worked and where they went to school. Triplebyte only cares about whether someone can code. So I'm a huge fan of that aspect of their model. This means that they work with lots of people from nontraditional and unusual backgrounds.

To get started, just go to triplebyte.com/sedaily and take a quiz to get started. There's very little risk and you might find yourself in a great position getting multiple onsite interviews from just one quiz and a Triplebyte interview. Go to triplebyte.com/sedaily to try it out.

Thank you to Triplebyte.

[INTERVIEW]

[00:03:57] JM: Tim Berglund, welcome back Software Engineering Daily.

[00:03:59] TB: Jeff, it's great to be here. It's been too long, for which I accept full responsibility.

[00:04:03] JM: It has been too long, and today we're going to talk about applications of Apache Kafka. You and I have talked about Kafka in other contexts in the past, and more broadly on this show, we've done plenty of previous episodes about the basics of Kafka. Kafka is a distributed replicated event queue. We'll get into the more advanced topics, but speaking from an industry perspective. Why is a distributed event queue an abstraction that is so widely applicable? Why are people using this thing in so many different ways?

[00:04:39] TB: Yeah, it's how we build databases, right? You have to remember, when you think about a database as a developer, you think about the API, and maybe if you think about a database, of course, that API is SQL and underneath that you've got some sort of tabular data model or whatever data model. And maybe if you think of a database as a DBA, as a person operating it, you think of what's going on in the file system, and maybe indexes, and concerns like that.

But inside, kind of at the Tootsie Roll center of the tootsie pop, inside of every database is an event log, a commit log, right? That's where things start. And this is a little bit – I think sometimes a little bit cutesy of me to give this explanation. But if you could think of a database, it's sort of like there's this log of events and then you build up these materialized views on top of that log, and those are tables, and indexes, and things like that, because reading the log would be a terrible way to live.

So we've virtually always built our databases that way, and now that we're building, more and more of us are in the business of building at least small distributed systems. We're finding that that commit log at the center of things is a useful paradigm and it solves a lot of problems and allows us to make some simplifying assumptions about the system and gives us ways of growing the system that we wouldn't have otherwise.

[00:06:02] JM: In the earliest days, people thought of Kafka as just this thing for publishing and subscribing to message topics. Since then, there's been higher level abstractions. There's been different APIs. Different systems built around it. Describes some of the abstractions that have been built into the Kafka ecosystem.

[00:06:24] TB: Yeah, I love this history. What you see is you see the core thing gets built, and that's this distributed log. That's what Kafka was at first. It was a distributed log. Like you say, at that point, Hadoop was a big deal. I have a big giant HDFS cluster. I have some things that aren't in the HDFS cluster and I need to get them into the HDFS cluster. So Kafka was sort of handmaiden to Hadoop always getting data into it as a big scalable pipe and a part of what we call the big data ecosystem.

Then people notice things, right? People noticed, for example, “Well, there are these kind of standardized legacy interfaces that I need to write code for all the time,” like there is a relational database, and I don’t mean to call it relational databases legacy. But there's legacy system with a relational database. So I need to write some code that does a select on the database and finds new records and produces them into a Kafka topic.

Well, you're only going to write that two or three times before you realize you should extract that into a framework. And then maybe you've done some work in Kafka and you've got some topic that has processed data and you want to put that into files in an S3 bucket or something. So you're only going to write that API code 2 or 3 times before you realize, “There's a framework there.” And this was the first major piece of kind of application framework that Kafka as a platform added, and that was Kafka Connect.

So the first piece of that evolution that you're talking about was Kafka Connect, and that was this pluggable, extendable, standalone integration framework, because the community saw that people were solving that problem over and over again and everybody always ended up with their own buggy partial implementation. So now there's a good one, and there’s like this rich ecosystem of connectors. So that's kind of cool.

After that, okay, I can get data in and out from standard things and everybody doesn't have to write that same S3 code or that same Elasticsearch code that doesn't differentiate you and doesn't add any value. You just do it one time and you get that connector. After you got that, people realized, “Okay. Well, now my consumers,” those are the application programs that are reading from Kafka topics, “they end up doing interesting things,” like they aggregate messages. They group messages. They compute aggregations.

You might enrich one topic with data in another topic, or something like you get a message in a topic and you need to go look a thing up. So this just happens. If sometimes I get the privilege of talking to enterprise architects at a big bank or something like that, it’s always a fun conversation, and you could say, “Hey folks, what do your consumers do?” And those three things that I just mentioned, well, filtering, aggregating and joining, that's like what everybody does. And the community recognized, “Oh, wait.

We're doing this over and over again, and it's really hard, and there's a bunch problems we haven't solved." So Kafka streams emerged, as of I think Kafka 0.10 as the standard framework for doing that kind of stuff in your consuming applications and doing it in a scalable way. I'm pretty sure you've had episodes on that. That's its own topic, but those are examples of Kafka growing kind of platform components and leaving its origins as scalable pipe behind and started to say to the world, "Hey, look. I have an agenda for how you build applications. Let's talk."

[00:09:55] JM: I think one of the reasons why Kafka has become so prominent is that you often have in modern application development a large distributed system and you want to synchronize state of that distributed system across your different clients, across your different services. Can you talk about how Kafka fulfills the purpose of synchronizing state across a large application?

[00:10:26] TB: Yes. Yes. By the way, I think that is an insightful account of what a distributed system is. It is sort of an attempt on the part of many programs running on many unsynchronized processors to negotiate an evolving state and try to come up with an account of it.

So the problem with state is that it changes. And if you have lots of copies of that changing state in different places, negotiating the changes to those copies is very difficult to come to the conclusion that you, yes, as a system, we have a consistent view of what is true right now is hard to do. And even those words, like what is true and right now, now, those are like philosophically robust concepts that I think most of us rely on in our daily lives.

In a distributed system, the idea of right now is a little bit fraud and everything. It gets complicated. But you've got individual applications and we could just make it simple and say that they're microservices and they need to share state. And if they all have a mutable copy of state, that is, each has its own internal key-value store, or each has its own relational database and you're trying to propagate that state between them. You should probably get a different job. And I don't mean leave software engineering. I mean like that system is going to be too hard to work on and you're going to have a more satisfying life if you just stop, because that's hard.

So what Kafka does is it says, “Look, what fundamentally is happening to your system is that events are impinging on it. Things happen in the world. Your system becomes aware of them, and that event is the primitive that you want to think about,” and it gives you a mechanism, that's a topic, for keeping an ordered record of those events.

What we know about events – I was talking about the mutability of state a moment ago and how that's a difficult thing when you've got multiple copies of the state. What we know about events, and we know this real life, is that they are immutable. If you've heard me talk about this before, you’ve probably heard me make the joke. An example of an event in your life that could be tragically immutable is you speak words and maybe you speak words in the heat of passion to someone you love, and they’re words that later on you wish you hadn't said. Well, it's too late. You said them. You can't unsay them. So events just are like that. They happen, and they can't change.

So Kafka sort of puts forward the immutability of events such that when an event is stored in a Kafka topic, you can let it expire after some period of time, but you can't change it. So all of these – Again, I’ll just keep it simple and say microservices that are consuming the same stream of events. Each one of those services might need to realize its own local copy of that state. Just like the database realizes its own local materialized view.

The database creates tables based on what's in the commit log. Each service is going to create some kind of indexable, efficiently readable view of what's in the event log. But the source of truth is what's in the event log. I've got this record of immutable events, and whatever I as a service and I as an application programmer and doing with those events, and let's suppose that’s some interesting thing that may involve interesting kinds of queries and lookups and things like that, fundamentally, I am reading that data, and I'm building that materialized view based on this log of immutable events. And that assumption of the mutability ends up being key and ends up being a source of huge number of simplifying assumptions that make this negotiating of state a lot easier.

[00:14:22] JM: To my mind, there are some use cases where a user might – Or I should an application developer might read data directly from Kafka, and there's other instances where an application might want to build a materialized view from Kafka. They could write that

materialized to you back into Kafka. They could put that materialized view into some kind of database that they're then reading from. And then they're going to build an application on that materialized view.

So I see these as two broad use cases, but I'm sure there's more subtlety there. How would you distinguish between the use case of reading directly from Kafka versus building a materialized view on top of it?

[00:15:10] TB: Those are both good things to have in the toolbox, and they're both completely valid tools to pull out of the toolbox. So I think I often kind of assume that, "Oh no! Whatever is in that topic must be super primitive than we all know all of our services are going to have to do a bunch of things to it." That really isn't true. You may well write a service that uses the Kafka consumer API to read a message and then do some unit of work on that message and cause side effects in the world and write a result to another topic. That's totally legit. I want to come back to that in the minute.

So let's just say maybe there's a topic that has what you need and all you need to do is read it and do the work and produce a result to another topic and do a thing. We'll come back to that. But let's say maybe you are – What are you doing? You're shipping orders. Okay? Let's say you're a service that ships orders. Well, what do I need to ship orders? I need to know about the orders.

So there's probably a topic that some other service is producing order events into. But I also need to know where to ship them. I'm going to need to get an address. And let's assume that those order events have a foreign key that goes to some user ID. Now I need to do a lookup on that user ID table.

Well, I could have a topic that's a change log on that table, right? Anytime a user record changes, I produce a new copy of that user record to that topic. Well, very simplest kind of materialized view. I could create of a topic is really just taking that change log and materializing it as a table that supports efficient key lookups. This is a thing that Kafka streams API does. This would be a great use case for Kafka Streams. Kafka streams is that Java API that emerged to do what people have been doing in consumers.

So I've got this change log topic and it's like a one-liner in Kafka Streams to say, “Make that into a thing called a K-table.” And then that K-table object has an API on it that lets me look things up by key. So I now am consuming that shipment object or that order message, and I can do a lookup on my user table, this materialized view of my user table, to have everything in one place to be able to create the order.

Of course, for that matter, I could also join that stream to that table. That's another Kafka Streams operation that would seem fairly natural there. But to be able to do that, I have to have created this materialized view.

If I could – I want to make this answer too long, but just to riff on that a little bit longer. Yeah, you might have like some kind of rating app where people are rating. I have this dream startup that's a real-time movie rating app. The killer is that it's considered rude to use your phone in a movie theater. So I don't think I can ever launch this. But maybe like for Netflix users. You're watching a movie and you can rate the movie in real time as you're watching it. This would be hugely valuable data to studios if they could know what people felt 23 minutes in. Did they think it was a nine? Did they think it was a two?

But if you can imagine, all of these real time movie ratings coming in. Another kind of materialized view you might want to create on that topic that – That topic is just user ID movie rating all over and over and over again. Well, you might want to group by movie and then compute an average of the ratings for the last 10 minutes or something over some window.

That's another kind of materialized view that would then be state inside that service, right? That's literally the results of computations that exist in-memory that are a materialized view of these immutable events in the log. So I can rebuild that at any time, because the immutable event log still exists. I don't need to worry, “Oh no! Is my version of that data in sync?” Well, because I'm consuming from this record of this log of immutable events. As long as I'm not behind in my consumption, I know I'm up-to-date. I don't need to worry about some other source of data having been replicated to me. Those are all concerns that as an application developer I just put aside, because I consume this topic.”

[SPONSOR MESSAGE]

[00:19:39] JM: Today's episode is sponsored by Datadog, a cloud scale monitoring service that provides comprehensive visibility into cloud, hybrid and multi-cloud environments with over 250 integrations. Datadog unifies your metrics, your logs and your distributed request traces in one platform so that you can investigate and troubleshoot issues across every layer of your stack. Use Dataodog's rich customizable dashboards and algorithmic alerts to ensure redundancy across multi-cloud deployments and monitor cloud migrations in real time.

Start a free trial today and data dog will send you a t-shirt. You can visit softwareengineeringdaily.com/datadog for more details. That's softwareengineeringdaily.com/datadog, and you will get a free t-shirt for trying out Datadog.

Thanks to Datadog for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:20:40] JM: Kafka can be useful for a company that's moving towards a microservices architecture, and I'd like to discuss that with you. But before we talk about that, actually, I want to explore a quick side note. Because you go to a lot of conferences, we make a point of communicating with a lot of people throughout the ecosystem. I've had conversations with a few engineers I respect a lot recently about the "microservices industrial complex", which is kind of this idea that – Not to like paint conspiracy theories, but there is an incentive of vendors to encourage people to buy things that cause them to spend more money.

And perhaps microservices could be categorized sometimes in some cases as one of those things. Maybe there could be an alternative path that we as an industry maybe should be going down. Like how could we architect our monoliths better? Is there a possibility that our industry has moved too far in the direction of microservices?

[00:21:50] TB: Yeah. No. There always is for a hyped thing like this, and I feel like a little part of me dies every time I say the word, just because it's such a hyped thing and I know where we

are in that cycle right now and I feel like I'm a part of it and I could stop. You're absolutely right. And if you take it a step further –

[00:22:11] JM: Hey, man. I'm the one that airs the ads for these products.

[00:22:14] TB: Right, that's true. That's true. You're much more a part of the industrial complex than I am. But if you –

[00:22:21] JM: Manufacture consent.

[00:22:24] TB: It's totally cool. When I talk about microservices and Kafka and this kind of broader – The architectural properties you can get out of a system by making Kafka your system of record. I try to start by talking about what a terrible idea microservices actually are. And I want people to realize that.

This is awful and there has to be some really good reason that you would do this. Because there are – For one part of your monolith to call another part of your monolith, there's literally hardware in Silicon to make that operation perform well and take like in a small number of clock cycles. And nothing could be more optimized than a method call. Yet, that's not good enough for us. We're going to take what used to be a method call and turn it into this lumbering traversal through a network stack and a network interface and [inaudible 00:23:18] and all that stuff.

[00:23:18] JM: And a VM, and a container.

[00:23:19] TB: And a VM! Like what is the matter with us?

[00:23:22] JM: Or a Lambda function.

[00:23:23] TB: So I'm really trying to dramatize – Right. Yeah! I always really try to dramatize how horrible that is. And I also try to tell people, "If there's any way you can possibly live your life as a software engineer, pursue that calling and not build distributed systems, please take that option. You're going to have more time for hobbies and you're going to go home earlier and probably sleep better and maybe live a few years longer. It's better." And nobody ever listens.

You make a good point, and I think there's got to be some truth that it's – As architectural trends and language and framework selections become status indicators, then there will be maybe lower integrity software architects and developers who make decisions to adopt to them, because it makes them look or feel good. And that's going to be the case here. Like there's going to be some adaption that's a thing, right?

Obviously there's a but coming, and that but is I don't think microservices are an irrational bubble. I do think despite the gross stupidity of taking a method call and turning it into call across a network and to making everything a distributed system. All that's bad and has real costs. The benefits actually exist for a certain class of system, and that class of system is not that large.

In the days when we said big data, I think it turned out that most of the systems we were talking about were really useful not all that much of the time, right? All the web scale examples we always gave, and I've been a part of this, are actual web scale companies, like Netflix and Amazon, and there aren't too dang many of those.

There's this whole rest of the world of people working for medium-size companies writing enterprise applications. What about us? I think that us, we actually do get utility out of microservices as an architecture. I'd love to dig into that more, but I think it is actually worth it. But we need to get there through the gate of remember how bad this is going to make your life. And if it's not radically improving some other part or your life, don't go there.

[00:25:37] JM: So let's assume I do want to break up my monolith, and I think we could go a lot deeper on the things you just said. I think that'll be a very interesting conversation. Probably not totally well-suited to the focus of this conversation, because there's a lot more I want to explore with you. So let's assume we do want to break up the monolith. We've got some –

[00:26:00] TB: We're doing it.

[00:26:00] JM: What's that? We're doing it. Maybe we are that architect. Maybe we're giving a prescription for malicious software architects to just spin their wheels for several years. But let's

assume we want to break up the monolith. Why is Kafka useful for breaking up monolithic software systems?

[00:26:20] TB: Got it. Okay. Well, the first microservices deployments communicated through a shared database, and the reason they did that is because, “Well, we had one lying around and we already knew how to use it and everything was there.” And it seems so obvious, right? It seems obvious that I need to pick up new orders and I need to query the users table and find – There is a service that maintains updates to my users table and the shipping service is just going to go query that sucker and get what it needs and it's real fast and it's great.

And what happened with those was that they became extremely inconvenient monoliths in the sense that the shared schema of the database tended to lock the versioning of the services. If you change one thing that involves the schema change, now you have to change every other service that relies on that schema. And all those had to be released at the same time. Maybe in a particular order in really bad cases, and it was all of the bad things about microservices and none of the good ones. I was not able to independently evolve and version and think about a single service. I still had to think about the whole thing.

By the way, I think that's fundamentally the reason why monoliths fail as monoliths, is that they’re just too hard to think about. So that one thing that I was trying to get, I couldn't get. We didn't know any better. We tried that. We quickly discovered as a community. Terrible idea. And next step was and really is having services that don't share a database. They have maybe local databases, but they talk through some kind of RPC mechanism, GRPC, or REST, or something. And that was better. It made it easier to independently version services, and didn't have this locking together through a shared schema.

But it created greater problems of service discovery for which any number of tools have emerged to help alleviate, and failure cascades became a problem. If you're synchronously calling one service and it’s synchronously calling another and another then another, then a failure six services down the line is going to cascade up. There are ways to mitigate that. But these are just some of the balls you had to juggle. Service discovery, failure cascades were harder.

Now that restful integration of services was still super easy, because calling a service feels like calling a function, calling a method. It's just over the network instead of through a call instruction on a processor. So part of I think why that paradigm is feeling some strain is because it's easy and all the APIs we used to do that RPC integration make it look like a method call. Yet, it is entirely unlike a method call in terms of its operational characteristics. It can fail in all kinds of ways. Takes, literally, orders of magnitude longer. So getting the developer in the mood of the method call while it not really being a method call I think is turning out to be a bad thing. So that gets us to why Kafka and microservices matter.

If instead every microservice is, to use another buzzword, a reactive microservice or we could avoid that and simply say event-driven, then the system at some point receives an input, and it doesn't matter what that interface is. Some service receives an input. Maybe a web form is submitted or whatever. It does some computation and produces a message. Now, whoever is interested in that message, like my shipment service a moment ago. The first service maybe as an order validation service and the e-commerce frontend throws some blob of JSON at an endpoint and that service and it validates it and then produces a message to a Kafka topic. Well, the shipment service now can pick that up and do its thing that we talked about with the user data and it's able to share data that way and it it's able to do its work simply when that event arrives in the topic.

Now, that's clear enough. But the amazing thing that happens here is those orders are in the topic now, and in our little minimum viable product e-commerce system here, we get it, there's orders, there's users. You got to ship things. You might need to make a payment service at some point. It seems like a good idea for MVP. What other concerns might arise that have to do with validated orders? Just stop and think about that for a second. Well, the answer is literally anything, and you don't even need to be the person who knows that.

I could have some mobile app that is location aware, and if you just bought something that has some product complement that’s sold by partner retail business that's near where you are, I'll send you a notification. That actually sounds a little creepy. I just made that up. But that's another fairly complex service that you might develop that is going to process this topic of validated orders. Maybe fraud detection is another one. Maybe I want to look at the history of

orders and see if something looks suspicious. Has nothing to do with payment. Has nothing to do a shipment.

Off the top of our heads here, we're just kind of making up some of the reasons to consume from that topic of validated orders. As the system evolves, as the team grows, as the company grows, scale this in any way you want. Here's that topic with those orders in it. They can last in that topic for as long as you want them to. You can make that retention infinite if you want. And any service anybody wants to write that can do something with that, well, they can write the service subject to authorization and any regulatory concerns or anything like that. The data is there.

Now new services can kind of grow up in that. It's like I always compare it to soil. Messages in that topic are like fertile soil and software can grow around that in a very decentralized, potentially uncoordinated way. All of these services can bloom and do valuable things on the basis of that immutable commit log.

[00:32:43] JM: So if I hear you correctly, and putting it in another way, you can tell me if I'm wrong. Instead of the point-to-point communication that you will typically have, the way that we think about microservices, the reason that the migration from a monolith to a microservices or service-oriented architecture with the onboarding path of Kafka. The reason that's useful is it allows you to effectively do multicasting so that you can multicast your messages to both your old architecture and your new architecture at the same time allowing for a smoother migration process from that old architecture to the new architecture.

[00:33:35] TB: Yeah. Yeah. I wasn't even thinking of it in terms of migration from one architecture to another, but that's also true. Once the data is in a topic and there is some legacy set of services or potentially even you've re-factored the monolith to read data out of the topic. Once the system of record is that topic, the migration gets a lot easier.

And the point I really want to emphasize is not as much migration from the monolith to the services as the future evolution of services. Because if you don't get that, if your microservices architecture does not give you a very credible story about how services themselves can be versioned and evolved independently of other services and new services can be deployed

independently of other services. If you don't get that out of your architecture, there's a chance that you have paid all the costs of microservices and you're not getting that primary benefit. You're not really realizing evolveability in your new architecture.

[00:34:40] JM: Let's talk about a separate topic, which is data engineering, and I think this gets to the point of the flexibility of Kafka and how widespread its adoption has been for so many different kinds of application. So I think of a microservices concept or this multicasting concept as business application. The things that you're using for your transactions in your day-to-day workflows.

Data engineering is often thought of as more of an offline process. It's maybe an eventually consistent process. It's the machine learning models that are getting written on a nightly basis or on an hourly basis. But it's interesting that Kafka can suit both of these applications. So Kafka can fit into a data engineering workflow in a variety of ways. Can you describe some of the common patterns of a data engineer working with Kafka?

[00:35:44] TB: , and this is really the other big use case. I think there are two – In the contemporary landscape, there are two reasons to adopt Kafka. Microservices are one. And what we call streaming ETL or streaming analytics is the other. And that really I think is synonymous with data engineering. The way I hear data engineering being used more and more is really just what we used to call ETL developer. Just with a slightly different toolset.

So the basic problem is things are happening and you want to understand something about those things that are happening. That used to be this nightly batch process. Things happened. They were reflected in a relational database whose schema was built and whose operational characteristics were optimized for fast reads and writes. And then at night, you pull that out and you mushed it around a little bit and you put into this different schema on different database servers whose operational characteristics were optimized for scans, analytic queries. That was the old days of ETL.

But it was still solving the same problem, right? Stuff happened. It happened yesterday, but I want to understand today something about what happened yesterday, which is legit. I mean, compared to the way things used to be, 24-hour latency is miraculous by some accounts.

The way this works in Kafka though is those things happen and with very low latencies. So this is probably sub-second latency. They show up in a Kafka topic. Now, let's just pause to reflect for a minute on what are those things that are happening and where are they being stored. If we use the microservices architecture I was just talking about, which is I think still a little bit space-age where we are right now at the time of this recording. But if you're doing that, well, good for you. They're already in the topics. So everything is easy. The rest of the analytic story we're about to describe is easy.

If they're not already in topics, if everything is conventional relational database, maybe you've got a monolith, maybe you've got SAP doing things. It doesn't matter. It's a database. There are various ways of capturing the changes in that database and producing those changes as events messages in a Kafka topic. And the keyword there if you want to look into more that, if you're a listener and you want to look into more of that, we call that change data capture, or CDC. It's not to be confused with the US government CDC. Different CDC. This is change data capture, which is the way you get stuff from a relational database into Kafka.

So you get that business activity into a Kafka topic, and that is in a sense the input side of your data engineering. And I think I was just literally waving my hands. It's a podcast. So you can't see me do that. Certainly, my verbal explanation of CDC was super hand-wavy.

Now, if you're a data engineer, you don't wave your hands about that, and there's a meaningful process of tool selection and operational considerations and all these stuff. It is some work and some specialization to do a good job getting that data into Kafka. But the toolsets are pretty stable. There's a fairly simple competitive environment. There's like an open source option and a few vendors and things that do that. So you get the data in.

And once you do, well, you don't have to wait until tomorrow, because there isn't. It's still just a commit log. The whole problem with ETL in the past, I think there were two problems. One is that the kinds of queries that you want to run to do analysis are usually scans, right? You want to start at some point in a table and read all the records until some other point.

And the kinds of queries you want to do when you're doing operational things and when you're recording business transactions and dealing with customers, those are usually insert a record, look up a record. And you can't do those on the same server, because caching relies on there being a power law of what records are interesting. And analytic queries don't. You couldn't ever do those in the same computers. You had to have this process, if for no other reason, just to get you another server that whose caches you wouldn't be wrecking.

In the case of Kafka, once the data is in a topic, it's fine. You can read that. We still have – Caching still is potentially an interesting question. But there isn't any other way to format that data. And if you've got services that are doing analysis – We talked about a topic with e-commerce orders in it. Well, you might want to keep a dashboard of that and roll that up by region and by product and product category and all these kinds of things.

Well, your services writing new orders to that topic, your order services. And whatever services are doing analysis on that, well, if everything is built correctly, they're keeping up-to-date. And every time there's a new message, they read that cached message out and include it in their aggregation and publish that aggregation to the dashboard in whatever way they do that. So streaming ETL, the idea is not to take stuff and put it somewhere and then scan through it later to do some big expensive query. But as events happen, do the analysis and constantly create up-to-date roll ups or aggregations of the events as the events happen.

[SPONSOR MESSAGE]z

[00:41:17] JM: The O'Reilly Software Architecture Conference is coming to Berlin November 4th through 7th, 2019. I've been going to O'Reilly Software Conferences for the last four years, ever since I started Software Engineering Daily. O'Reilly Conferences are always a great way to learn about new technologies. You get to network with people. You get to eat some food, and there's no better type of conference than a software architecture conference, because it's a great high-level explorational topic. And on November 4th through 7th, the O'Reilly Software Architecture Conference is coming to the City of Berlin, and you can get a 20% discount on your ticket by going to oreillysacon.com/sedaily and entering discount code SE20.

O'Reilly conferences are a great place to learn about microservices, domain-driven design, software frameworks and management. There’re a lot of great networking opportunities to get better at your current job or to find a new job altogether. I've met great people at every O'Reilly conference that I’ve gone to, because people who love software go to O'Reilly conferences.

You can go to oreillysacon.com/sedaily to find out more about the Software Architecture Conference. These conferences are very educational and your company will probably pay for it, because you're going to learn about how to improve the architecture of your company. But if they don't want to pay, then you can pay and you can get 20% off by going to oreillysacon.com/sedaily and use promo code SE20. That link is also in the show notes for this episode.

Thank you to O'Reilly for supporting us since the beginning of Software Engineering Daily with passage tier conferences. Thanks for producing so much great material about software engineering. I particularly enjoyed the episode that I did with Tim O'Reilly a year or two ago, and in that episode I really got a better understanding of how Tim O'Reilly built his conference business. So you can always check that episode out.

Thanks again to O'Reilly.

[INTERVIEW CONTINUED]

[00:43:37] JM: There’s an abstraction called a data lake, and most people who are still listening at this point probably have familiarity with a data lake. It's a thing that we’re dumping a ton of data into. Kafka can look like a data lake in some ways, because if you're dumping all of these – The change data capture, all of the changes to your infrastructure through Kafka, and maybe it's getting flushed out and there’s some kind of garbage collection policy to it. But in any case, there’s a lot of data there. How does the usage of Kafka compared to a data lake?

[00:44:17] TB: That is – I don't think I've reflected on that question before. That’s great question. It's not a lake. It's a stream, Jeff. So it’s entirely different. I know, but that’s cheesy. But let's explore that analogy just a minute. The idea with the data lake is that you do put things in it and you'll opportunistically go back and kind of fish through the lake later on. You’ll write Spark

jobs, let's say, now is probably the thing you're going to do to do those batch analyses of that data.

I'll be frank, I don't think Kafka makes that a bad idea. I think having something like a data lake for offline experimental data science development is probably recommended in systems of even medium scale. What's different about Kafka is I think ultimately operational expectations that when you do a thing in a data lake, latency is – You're not too worried. Everybody wants everything to be fast. But if a Spark job takes a few minutes to run a large set of data, that feels pretty good. A latency of a few minutes through a Kafka cluster for a streaming ETL pipeline would be pathological in the extreme. A latency of five seconds is something you want to stop and look at figure out what's broken.

So I feel like I'm missing something subtler. I'm just going to go with that for right now, that they're similar and that they both have water in them, but their water flows in one of them. And if you're trying to build a real time ETL pipeline, you will not do that with a data lake. You definitely need a streaming technology for that.

[00:46:02] JM: Well, let's talk more about the streaming technology. So stream processing is obviously a tool that is widely used with Kafka. There are Kafka streams. Then there are other streaming frameworks like Flink. Do you have suggested patterns for which streaming tools to use in which contexts?

[00:46:22] TB: Yes. So I spend my time as an advocate for things in the broader Kafka ecosystem. So unsurprisingly I normally come out with a fairly Kafka friendly set of recommendations. Worth talking about Flink though, because it's a super powerful framework. A lot of smart people behind it. A lot of interesting adaption.

I think we are seeing a pattern emerging where Flink gets adopted where there is a big stream of data to process. Like it's the feeling of, "Oh wow! I have this giant stream. I have to do things with it," and almost as if there's one big stream and I need to manage this one big stream. And that's true, because, operationally, Flink is its own cluster. In Kafka, you have a Kafka cluster. The brokers doing storage and pub/sub. And in Flink, that Kafka cluster is going to feed the Flink cluster on which the distributed stream processing computation takes place, which means

my stream processing programs are kind of the special-purpose things that I deployed to the cluster using that cluster's opinion of how applications are built and deployed. There's like one way to do it. You run that application.

And super simplifying, the account of Flink. I know that you've had great discussions out on the show and there are a lot of interesting things to say about it. But because of its deployment decisions, it seems to be gravitating towards, "Hey, I have this giant stream. Let me process it."

Kafka Streams takes a very different approach, and that it's a Java API. Literally, a dependency that you put in your build file and it's an API you code against. So you have some application. One of these microservices that's doing whatever it does. Like maybe it's a spring boot app with a web frontend or some RESTful thing that it's exposing to the world, whatever. It's a program. It does things. Plus also it has to do some processing of some topics to get some key value stores arranged inside the service to be available to that spring boot app to do things.

Well, Kafka Streams gives you essentially a conceptual parity with Flink. We're all doing the same things. We're joining and we're filtering and we're aggregating and we have lower-level APIs that we can dig down and do more detailed things. So you're doing all those stream processing things, but you're doing them in the context of your application.

There is a scale story for that, by the way, for how you horizontally scale that application, since you're adding all the stream processing application. Your server is getting hotter. So you're going to need to scale it, and that is simple to do to scale that application horizontally with Kafka streams. But that's the idea, right? So when that stream processing is married to some other application functionality, like I'm a microservice and I want to process streams. It feels incredibly difficult now to go do that with Flink and very natural to do that with Kafka Streams. So there's more that story. But just as a very first blush way of differentiating them. That's how I'd do it.

[00:49:27] JM: So this is to say that – Oh, go ahead.

[00:49:29] TB: I want get KSQL in there too, because it's kind of an interesting twist on this. All right. So this gets back early on, you asked, "Is it okay for a service just to consume from a topic? Do I have to do Kafka Streams and stream processing?" And I said, "Yes, and we'll will

talk about it more later." It's totally fine. So if you're a service and there is a topic that has what you need in it for you to do your work, you just consume that thing and go on your merry way. And you start to look a lot like serverless function at that point. There's this event. You get it. You statelessly do your work and go on your way. And that's a beautiful way to live.

It's unlikely that that topic is just there for you lying around. When you're writing new service X and that microservice is sort of starting to feel more like a serverless function, a lambda, or something like, because it's a really simple stateless microservice. How do you get that topic ready with that stuff in it?
Well I could go write another Kafka Streams application and deploy it as another service that does the joining and grouping and whatever computation I need do on the various streams to get the work ready for that other thing. Or I could use KSQL.

KSQL does more than this by the way. I'm giving you one use case. But it's a SQL-like language to do the same kind of stream processing that Kafka Streams does. So you can write a line or a few lines of SQL to kind of prepare that work for that service to digest. And it may be clear also that in the streaming ETL pipeline case that we are talking about a minute ago, KSQL tends to pop up as the way people prefer to do the analysis of things. Once you've got the data in a topic, you can just kind of KSQl it into the form that you want and then dump it out into whatever system you are using to visualize.

[00:51:20] JM: So I think one way of interpreting what you've just said is that if you're looking at different streaming frameworks, one tradeoff you're going to be faced with is how close is this streaming framework going to be sitting to your data. And from what I heard, this Kafka Streams library is going to be sitting on the same application that is running Kafka or the same server that's running Kafka. The same abstraction that's running Kafka. So the data is right there as supposed to Flink.

[00:51:55] TB: Not on the broker.

[00:51:56] JM: It's not on the broker. Okay.

[00:51:58] TB: Not on the broker. Yeah, it's in the consumer basically. So it's still your application, which is running outside the consumer.

[00:52:04] JM: Okay. So why is it that that's able to get you to lower a latency than something like Flink?

[00:52:10] TB: I don't know. I don't want to say on the air that it does, because I actually don't. I could not produce for you benchmarks that validate that that's true. So I don't know that it's lower latency. I do know that it's lower deployment complexity. Yeah, because in Flink, like you're writing your "Flink program" and deploying it to the cluster. In a Kafka Streams, you're writing a microservice and you're using the stream processing API, and you deploy your microservice however the heck you want to. Streams doesn't have any opinions about that at all. It kind of stays out of your way. Whereas with Flink, it's this special thing that goes to the other cluster that you also have to maintain.

[00:52:53] JM: Got it. To take a step back, there are organizations to this point that have pretty mature Kafka installations. Could you tell me about what are the struggles that late stage Kafka users have? Is there anything in terms of cost management or infrastructure sprawl? What kinds of issues do large Kafka users have?

[00:53:22] TB: Schema is always a thing, right? Because I'm talking about this wonderful world in which you just produce messages into topics and thousands services sprout out of them, and each one is a unique and beautiful flower, which is all true. But like, "Hey, what's in that topic?" We kind of need to know that. So schema management is a thing that becomes more important at scale. And because at scale, there's a topic, and I don't know the folks who work on that topic. I don't know what's in it. I can't go ask them easily. The wiki pages out of date, blah-blah-blah.

If I want this evolutionary architecture to happen at scale, which is even better than at a smaller scale, I need to have a reliable way of knowing what's in that topic. And Confluent has a component called the Confluent Schema Registry, and its available as part of Confluent Cloud. It's free. You can download it and use it. It's not a part of Apache Kafka. I mentioned Confluent, because it's our thing and it's not a part of the Apache Project. But it helps with all those things.

So schema is a big deal, and I always see big organizations struggle with standardizing on APIs and they all write a wrapper around things, which tells me that there's probably a good reason for that. I have not managed an engineering team at the scale of a thousand engineers in a multibillion-dollar business. So I don't know what those things are. I can say I never like it, right? I never like seeing company X develop the company X wrapper around the core Kafka APIs. But they'll do it.

So kind of getting that right and building standards, building compliance into those things. All the stuff that you just want to make sure everybody, even if they're not a Kafka expert, does the right thing and everybody does that by building a wrapper. And it seems that those wrappers are usually a big investment. Like I said, I personally have a negative reaction to them, but it's hard for me to make the case that they're e malinvestment, because of a lot of smart people do them and they do have a lot of hard problems to solve at scale.

[00:55:42] JM: To wind down the conversation, you work at Confluent. Confluent is a company that builds application systems and do consulting style work around Kafka. And Confluent has grown tremendously. The company is doing really, really well. Tell me about your work at Confluent and what it's been like over the past 2-1/2 years I think it's been. What's been your experience at the rapidly growing Kafka company?

[00:56:14] TB: Yeah. So I always wish I could say that I came to Confluent because I'm really good at being an industry analyst and picking horses and I said, "Hey, this Kafka thing is going to be good," and it just wasn't that at all. It took me about six months after getting here to realize, "Oh, wait. Hey! No, seriously. This is great."

[00:56:33] JM: Yeah. Right.

[00:56:34] TB: This is a great idea.

[00:56:35] JM: Right. it's a great idea.

[00:56:36] TB: Yeah, and I had to come to that from the inside, which, hey, it always exposes me to the risk of just having drunk the Kool-Aid. But, for me, has been the experience of seeing the developer community broadly embrace this technology and figure out the right ways to do it and getting to see customers. It's always good to see people with money on the table or on the line building things and making engineering investments. What problems do they have? What problems are they solving?

But the commitment, I do feel like this is sort of a generational commitment to this kind of technology. Like this is a pivot point. And that's exactly what the Kool-Aid drinker would say, Jeff. But just watching people adapt and think about and seeing what's going on through this kind of Kafka-centric lens does seem like it's a pretty big deal.

Also, kind of from a theoretical standpoint, I've had the opportunity to think about those questions, and none of that is like original scholarship on my part. I'm being influenced by guys like Ben Stopford, Neil Avery, and Martin Kleppman and these folks, but those arguments are awfully good and it seems like this is providing the right solutions to the kinds of problems we've made for ourselves in the last five or 10 years. So I'm buying it, and it's fun to see.

[00:58:07] JM: All right. Well, Tim, thank you for coming back on Software Engineering Daily. It's been a pleasure, once again.

[00:58:12] TB: Always delighted. Thanks, Jeff.

[END OF INTERVIEW]

[00:58:22] JM: Software Engineering Daily reaches 30,000 engineers every week day, and 250,000 engineers every month. If you'd like to sponsor Software Engineering Daily, send us an email, sponsor@softwareengineeringdaily.com. Reaching developers and technical audiences is not easy, and we've spent the last four years developing a trusted relationship with our audience. We don't accept every advertiser, because we work closely with our advertisers and we make sure that the product is something that can be useful to our listeners. Developers are always looking to save time and money, and developers are happy to purchase products that fulfill this goal.

You can send us an email at sponsor@softwareengineering.com even if you're just curious about sponsorships. You can feel free to send us an email with a variety of sponsorship packages and options.

Thanks for listening.

[END]