EPISODE 04

[INTRODUCTION]

[00:00:00] JM: Apache Kafka was created at LinkedIn. Kafka was open sourced in 2011 when the company was 8 years old. By that time, LinkedIn had developed a social network with millions of users. LinkedIn's engineering team was building a range of externally-facing products and internal tools, and many of these tools required a high-throughput system for publishing data and subscribing to topics.

Kafka was born out of this need. Overtime, Kafka's importance within LinkedIn has only grown. Kafka plays a central role for services, log management, data engineering and compliance. LinkedIn might be the biggest user of Kafka in the entire industry. Kafka has many use cases and it's likely that they are almost all on display within LinkedIn.

Nacho Solis is a senior software engineering manager at LinkedIn where he helps teams build infrastructure for Kafka, as well as Kafka itself. Nacho joins the show to discuss the history of Kafka at LinkedIn and the challenges of managing such a large deployment of Kafka. We also talk about streaming, data infrastructure and more general problems in the world of engineering management.

Full disclosure; LinkedIn is a sponsor of Software Engineering Daily.

[INTERVIEW]

[00:01:28] JM: Nacho Solis, welcome to Software Engineering Daily.

[00:01:32] NS: Hey, thanks for having me.

[00:01:33] JM: So you work on Kafka infrastructure at LinkedIn. Famously, LinkedIn started the Kafka project. Describe how Kafka fits into LinkedIn's software architecture today.

[00:01:45] NS: Yeah. So at LinkedIn we use Kafka for everything. It's the right tool, came out at the right time. It is simple to use and people just picked it up. Everything from click tracking, to logging, to metrics, which you would expect, as well as database replication and everything in between, stream processing obviously, we use Kafka. As a matter of fact, I think that on our IDEs, so we use here IntelliJ for stuff. IntelliJ talks to Kafka to get events when our builds succeed or fail or things happen. So, yeah, everything goes through Kafka.

[00:02:18] JM: Can you use that as an example? How does IntelliJ integrate with Kafka?

[00:02:23] NS: Yeah, it's a very good question. The way we build stuff when you commit code, you submit some PR equivalent. It gets approved, and then builds and goes through tests. Eventually when everything passes, it gets some feedback saying, "Yeah, your things passed." We have extensions to IntelliJ to let us know what happened. So in this case, the things were countered, not work out. It's a minor use case compared to like everything we do. I'm just kind of showing the breadth.

[00:02:46] JM: I'd like to understand more about how platform engineering fits into LinkedIn. So that example of IntelliJ integrating with Kafka in order to have information about up-to-date builds being pulled into your IDE, that's really useful. But that's kind of a common good. So it's the kind of thing that would make sense for basically anybody in the organization to have. But it begs the question who at LinkedIn is actually responsible for building that?

[00:03:22] NS: Yeah. So we have the benefit and the luck that we're kind of a good size. LinkedIn is a good size. We don't have a lot of duplicate efforts. Obviously, big companies have duplicate efforts. But in our case, Kafka is a one single bastion that everybody knows.

So whenever you're building a system, whenever you're building some platform or some tooling, you're thinking of creating a new part of our infrastructure. If you need to do anything that is not necessarily real-time, but it can be asynchronous, it can be near real-time, like, "Oh! How am I going to do my inter-process communication, or like my communication between my microservices and Kafka? It's just the thing that is going to come up."

So that spans, like I said, all the way from like inside. You have some large database you're building. Internally to your large distributed system, you need to communicate and you might as well use Kafka inside your system, or you're writing some tool and you need to produce something that may be ETLd or maybe consumed by another tool, and you put it into Kafka. So everybody knows. So it's very common for this to be a thing that everybody – It's your hammer, right? You can use it to hit anything.

[00:04:27] JM: You're on the Kafka team at LinkedIn. So is that the team that built the IntelliJ plugin for example?

[00:04:35] NS: No. So my team is in charge of building, running, maintaining, supporting Kafka. So my team runs the clusters. My team runs the base clients, the slightly higher abstracted clients, REST Proxy, that sort of stuff. So other teams just product and consume from Kafka.

[00:04:52] JM: How many people are on the team that's entirely dedicated to Kafka at LinkedIn?

[00:04:56] NS: So we're about 25 devs and about 10 SREs.

[00:04:59] JM: Why does it make sense to have 35 engineers working exclusively on Kafka infrastructure?

[00:05:05] NS: It is a large system. Like I said, we depend on it. So in this case, for example, when Kafka fails, depending on what part of Kafka fails, LinkedIn as a whole can fail. [inaudible 00:05:13] can fail. For example, we use Kafka to move data from our source of truth database in one data center to another source of truth database in a data center. This is asynchronous and eventual, but it's actually not that long. It's like seconds.

So if we fail, if that part of Kafka fails, we won't be able to get up that information. Hence, we can't serve stuff to our customers. To keep this up, we need to make sure that Kafka is up and solid. So we maintain Kafka. We maintain the Kafka ecosystem, and that is a number of products. Not just the broker itself. I mentioned that we handle the clients, and all that adds up.

Then obviously we need an SRE team here working 24/7. We have part of our team in Bangalore. Yeah, that's up.

[00:05:55] JM: What I find so interesting about this architecture, I mean, I know that there's a lot of companies out there that have some kind of pub/sub system that is a really important backbone of how their infrastructure operates. But it seems like at LinkedIn, it's like that idea taken to the logical extreme. I can't think of another company that I've talked to that has this intensive level of pub/sub infrastructure internally. Have you talked to any other – I think Spotify actually. I talked to Spotify, and they have a pub/sub system that's this crucial. But are there any other case study companies that have a pub/sub system at this scale that you've talked to?

[00:06:38] NS: I don't think so. So we are, as far as I know, the biggest Kafka users and consumers. We're normally the ones that notices the problems at scale. We don't sell Kafka as a product. So obviously our friends at Confluent can probably tell you about all the little different places where Kafka is essential. But the people that we know and that we talk to, including Microsoft by the way, which is also a big Kafka user. They don't have it at the scale.

So for us it was ingrained in the culture, in the engineering culture. People are aware of it. Like I said, it's there and it's just easy to use. People use it. It's reliable and people trust it, and hence you just build it into your system.

[00:07:12] JM: Is there something about the nature of the LinkedIn product? The fact that this is a at-scale social network that makes it sensible for Kafka to be the center piece, or do you think this is just more of almost like a coincidence, where LinkedIn happened to be the one that developed Kafka and it coincidentally became this center piece of the company.

[00:07:40] NS: I think it was coincidental with a lot of work. So it's not just by magic. Obviously, Jay and June, they have created an amazing product, again, at the right time when there was this need to have faster processing to unify stuff, and it just kind of blossomed. Now in the case of LinkedIn, it unifies some of the philosophy and culture. I know other companies do similar things.

But there was this discussion in the past, or at least this way to frame the problems. What is your unifying waste of an engineering large ecosystem? You can imagine having APIs that evolve. But you can also imagine having data that evolves and you can like handle data one system or another.

So Kafka allowed the creation of this federated infrastructure where you just put the data in. You're a producer. You may not know your consumers. Consumers may come up later, right? So mixing Kafka with that philosophy just kind of like made it work out, and that allowed us to continue working on Kafka. Making it stronger and making it better. Eventually, new use cases come up. Those use cases forced us to extend Kafka to then do that next use case of having lower latency or having now, for example, thinking of Kafka inside of another system. It's not exposed to anybody else. It's just inside a system.

Now, you can think of it as a very strong IPC between your two components. So in one of our systems, for example, Venice, which is a derived data store. It replaced our old system, Voldemort. You push data into Kafka. You consume data into Kafka. It's kind of similar [inaudible 00:09:05] architect stuff. Then you create and delete topics on-the-fly. It's kind of just an API, right? For them it's an internal part of the component.

So luck, perseverance, good work, being able to see what we have to see. I mean, now that we're perfect. Obviously, as a leading kind of Kafka team here, I know most of the parts where I wish it was different and I wish we could improve, and we're working towards that right now.

[00:09:26] JM: Like what? What's an example?

[00:09:28] NS: Oh! Scalability for us. So I don't think every other company hits this problem, but in our case, we hit scale issues. This is going to sound a little bit bad. But let's say we have a cluster as N-number of nodes, and the current Kafka design, which it's potential evolving, but [inaudible 00:09:44] has one controller. The controller kind of decides what happens to the cluster. Whenever something happens, it tells the other nodes, "Hey, this is what changed. You need to update your situation." This is done via communicating some state to the other nodes.

This state can be potentially large. If the number of nodes are large and the state is large, there's a lot of information that has to happen. That may, in some circumstances, be printed out in a logline. It just so happens we had a situation where loglines for us were 100 megs each, and we had many nodes and a lot of state. That caused us to like run out of memory for printing a logline.

Now, I don't think a lot of people hit this problem, right? It is just the nature of this particular large cluster that had a lot of things going on.

[00:10:29] JM: What did you do there?

[00:10:30] NS: Oh, we didn't print the logline.

[00:10:32] JM: Okay.

[00:10:34] NS: So in this case, obviously, everything [inaudible 00:10:35] is a tradeoff. So we could have chosen way. We could have chosen another way, and we obviously have multiple avenues that we're trying to approach. We could have reduced the logline. We could have made it binary. We could have spread out the printing of the logline. We could have spread out the communication. We could have reduced it. But at the end of the day, in this particular case, we have that same data that that logline provides in the other nodes. So it wasn't required. So we had patched it. Kind of not print the logline.

[00:10:58] JM: So Kafka was built in this golden age of new Hadoop infrastructure, and around that time, ZooKeeper was widely used for doing the distributed systems management. I know there's been some effort recently about factoring out ZooKeeper.

[00:11:21] NS: That's KIP-500. Yeah.

[00:11:22] JM: What is it called?

[00:11:23] NS: Whenever you want to change Kafka and add something new to Kafka, you create what is called a KIP, a Kafka improvement proposal. Hopefully I got the acronym right.

Those get numbered and get discussed by the company. Proposals are made. Eventually they're approved. They're kind of design docs. They're community design docs. Once they're approved, somebody starts working on them and you continue designing. In this case, that one in particular, if I'm not mistaken, is 500. KIP-500 is getting rid of ZooKeeper.

[00:11:48] JM: What's the motivation for getting rid of ZooKeeper? ZooKeeper, by the way, I believe it's still Paxos-based distributed consensus implementation.

[00:11:59] NS: That's right. Yeah. Paxos or Raft. It doesn't matter. But it's basically a server. Kind of looks like a key value store, and it is consistent, and hopefully fast and is useful.

[00:12:07] JM: It can be described as a lock server, I think.

[00:12:10] NS: You can use it for log server. You can use them for a number of things. Also, a simple problem that once you hit it at scale, becomes potentially an issue. So in the case of Kafka, ZooKeeper access like the memory of the brain. So the brain being, let's say, the controller. But everything it knows is in ZooKeeper. So whenever something changes, everything has to go in and out of ZooKeeper.

In the past, when Kafka started, ZooKeeper was used for everything. So all clients would talk to ZooKeeper, and the controller would talk to ZooKeeper. The nodes talk to ZooKeeper. So at some point, there is interdependency, and you can have like failures in one that cause failures in the other. Eventually you have to protect one from the other. At some point, in the Kafka 2.0 I believe, we upstream deprecated the consumers that will talk to ZooKeeper. But it's kind of in the picture.

When you reach a certain part of scale, then talking to ZooKeeper becomes an issue. There're a lot of updates that may happen. You have to communicate. You have to elect. It's just an extra requirement that is not needed. Given that Kafka already solves all these problems, it already has to deal with distributed system issues. Then would it be possible for just Kafka to handle this part and not rely on external system. At least for us, Kafka is at the bottom layer of all of LinkedIn infrastructure.

When we build a new data center, when we build a new thing, Kafka is the first thing that we try to build. Sometimes there's like just kind of funny loopy dependency to even the deployment tools, which normally talk to Kafka. But ZooKeeper is our previous dependency. So ZooKeeper itself is required. So if you remove one dependency, your system becomes hopefully easier to manage. I see a lot of hand waving, because obviously that means that you now have logic inside Kafka that would have to –

[00:13:48] JM: I don't understand how you would factor this thing out though. Don't you need a log server?

[00:13:52] NS: You do.

[00:13:53] JM: Consensus protocol.

[00:13:54] NS: You do.

[00:13:54] JM: To build a distributed system like Kafka?

[00:13:56] NS: You need some part of the system to handle that part, to handle the – Make sure everything is consistent and know who's storing so that when you have a failure, you can have still your data. I'm not familiar with the KIP-500 proposal exactly. So I couldn't tell you what the exact design is, but I can assure you that it is possible.

[00:14:12] JM: Do you have any perspective on if Kubernetes or etcd perhaps has made this stuff easier to build?

[00:14:20] NS: So I myself don't know the details of ZooKeeper even as much as I use it. I mean, I know enough. I know slightly less of what etcd, but they're equivalent. So you could swap one for the other. Is the primary goal to solve this kind of use case of Kafka? Partially. In this case, Kafka is just trying to make sure that it can act independently. In many cases, like I said, ours being a good example, Kafka is used to monitor our systems. That includes monitoring our ZooKeeper installation, right? Which kind of creates this like chicken and egg problem. Not everybody might have the same situation, but for us it would be useful if Kafka didn't rely on external systems.

Having said that, I'm still highly in favor of having modules and compartmentalization and being able to like separate the issues. KIP-500, we'll see how it develops and what it provides. If it helps us with scalability and issues, that would be great.

[00:15:06] JM: Okay. That sounds like an entire show in and of itself. Not to go even further on this subject of why LinkedIn became a company that was such a heavy user of Kafka. So just a very unique software architecture. I wonder if it also has anything to do with the fact that LinkedIn got started at this very distinct time I guess between – I think when was LinkedIn started? Like '05, or '07? Anyway, maybe '04, but somewhere around this time where it ended up needing to build large scale systems, but it was before public cloud was a big deal.

So do you think LinkedIn's unique architecture has anything to do with the fact that it was kind of a pre-cloud company and it's one of these few at-scale companies of the web 2.0 set or web 2.5, whatever you want to say, that was started pre-cloud? In this strange window where it built scaled infrastructure that was not cloud?

[00:16:05] NS: It is very possible. So I wasn't here to know the exact details on when these choices were made. But LinkedIn had a slow start at the very beginning, right? Eventually it took off.

[00:16:15] JM: Not a hockey stick growth company. Interesting.

[00:16:17] NS: No, it was not exactly – It's slightly different. So from that perspective, things were built, things existed. How we end up deciding not to do the early, obviously, early cloud stuff. I didn't participate in the discussions. I believe you're talking to Kapil in another podcast. He may be able to give you more insights into how that happened.

Yeah, oddly enough, as people probably know, we're moving to Azure at some point in time. So we have to consider now a lot of these issues. We are very proud of our data centers and we're very proud of the stack that we have. So it's not like were throwing anything away. They serve quite a bit of purpose. I mentioned before, the example of we have the largest Kafka installation. We rely on it, even larger than what Microsoft has on the Kafka side and stuff. So for us, at that point in time, how systems got built – Yeah, I don't know all the details of –

[00:17:03] JM: All right. Let's talk about usage of Kafka. So let's say I'm an applications developer within LinkedIn. Maybe I'm working on the profile team, or the search team, or the ads team. Something like that. Probably I'm interacting with Kafka in a number of ways. There're probably a number of patterns I'm employing. The first one I'd like to explore is this idea of using Kafka as the medium of communications for talking to another service. So I think a lot of people who are listening and they're building systems at whatever company they're working at, their notion of one service calling another is I make a direct HTTP call to this other service.

But in certain models where you're using a system like Kafka, you can essentially issue – You can have different communication patterns where basically Kafka is middleware between every communication between different services. Could you help describe in a little more detail how that pattern works.

[00:18:12] NS: Right. For us, when you're dealing with – Let's say what we would call online, which is your request. User request comes in. It's on servers. You get your profile back. That particular request doesn't go through Kafka. So that all goes through, like you said, HTTP. In our case, we use restly. So restly is our framework to do like RPCs and it's what we used for like the real-time, we need fast response and things.

All of that produces Kafka traffic in one way or another sometimes hidden from the actual product or application, like we move, like I mentioned, click data, or whatever it is we end up doing. Eventually a lot of that is ETLd out. It can be processed in Hadoop. That can eventually be served. That again moves through Kafka. Kafka actually access the middle system between online and offline, and all that will go through Kafka one way or another. From the perspective, you could be in the profile team and doing something and you will not directly use Kafka. You would use some system that will do Kafka for you one way or another.

Even if you're doing things, I guess an example would be you're looking at people you may know. So people you may know is data we compute. So we can suggest, "Hey, you may know this person." That will go through Kafka in many directions. But at the end of the day, you're calling our database that serves you that data after it was computed offline. You will not call Kafka directly.

[00:19:29] JM: So let's say a user loads a page that includes the people you may know, module, does that mean that when the user requests the people you may know, there's some service that's going to get the data from the database and serve that to the user, but asynchronously, there's maybe an event that's written to Kafka that says –

[00:19:51] NS: Many events, yeah.

[00:19:52] JM: Many events. Okay.

[00:19:54] NS: That database was populated by Kafka.

[00:19:56] JM: Okay. Great. Got it. So is Kafka – Is it like an asynchronous notification system that allows anybody to create and read events that describe the traffic that's happening across LinkedIn's infrastructure?

[00:20:15] NS: You could think about it that way. For us it's a little bit more structured. So in our case, all the click data, all the page views, all that generates data that goes through Kafka and eventually it's ETLd for analysis, right? Anybody can subscribe, assuming you have permissions and we have ACoLs and all that kind of stuff. But assuming you have permissions, you can subscribe to that data and then just start processing that data for whatever reason. There are many reasons why you may use this data.

Things that happen that are transactional. Like you go and you edit your profile. That communication that edits your profile goes to our database, our source of truth. Internally that ends up replicating via Kafka, but let's say that's hidden from you. There's a database that source it. From that database, we put obviously change events, and then that goes into Kafka. Then now you can take actions on those events, like something changed, like you changed your title from X to Y. We build many systems, for example, that will then take actions on these and then augment your profile in one way or another.

A good example is actually the way we do centralization of titles, right? So the example I'd like to use is you call yourself Java guru, and that's what you put, because we allow you free form entry and you say, "Oh, I'm a Java guru." At the end when we're suggesting jobs to you, we're probably not going to suggest meditation. We're probably not going to suggest coffee. We will suggest software engineering jobs, and that is because some systems saw that you changed your title, decided that process this and eventually say, "You know what? That actually means software engineer with Java expertise." Then we augment your thing to be able to do that. That is then because an event was generated out of the data they could change in our database.

[00:21:42] JM: There's a "pattern" that is sometimes mentioned in this kind of discussion called event sourcing. I don't know if you think this is a useful term, but maybe could you give a definition of event sourcing and describe whether it applies to this situation.

[00:21:57] NS: Yeah. So I have to admit, I am not clear on what event sourcing encompasses. Whether it is –

[00:22:01] JM: Okay. That makes two of us. Yes. Wonderful.

[00:22:03] NS: Whether it is the fact of having multiple systems use the same event to deal different things or whether it is a series of event that you can use to reconstruct something from the start and you have all the events so you can now, whatever, backtrack or look at intermediate states.

[00:22:17] JM: Replay.

[00:22:18] NS: I am not certain which of those – The term means, but we do both of those.

[00:22:23] JM: Okay.

[00:22:24] NS: Our different systems will deal with this in one way or another, right? Obviously, we have systems that can replay stuff and like construct state and go to intermediate states if necessary. We have things where some change in the system is replicated to multiple systems downstream to take different actions. So both of those happen.

[00:22:39] JM: Okay. So let's zoom in on just the subject of the word event. So event is a data type that describes a change that has occurred in the system, and if you took all the events that happened across LinkedIn, I think to some degree it would describe the updates to the databases that are happening across LinkedIn's infrastructure.

All these events are getting stored in Kafka. To what extent do you need to keep those events? How long do you need to keep them for? Because if this event data is describing everything that's going on across all of LinkedIn, do you want to save all of those events? Do you need to keep them in-memory? Do you want to flash them to disk at some interval? Could you just discuss the topic of event data?

[00:23:25] NS: Yeah, I don't think your podcast is long enough to have this discussion. But let's talk about it for a little bit.

[00:23:30] JM: Sure. Yeah.

[00:23:30] NS: If it was my choice, I would keep all events forever of all type and have the latest snapshots so I could do any computation at any point in time. That is not always possible. In the case of Kafka, Kafka has retention and is durable. So we store to disk. So Kafka will store to disk, the events that you send to Kafka. After sometime it will flash them out and delete.

By default, currently we have a four-day retention on like new topics that created with no specific settings. We have multiple pipelines, but let's use that as a general term. It could be less. It could be more. We can configure it. At the end of the day though, there are multiple reasons why you don't want to keep everything into Kafka. There might be changes that happen. It costs you money to keep all that data, which is not considered social truth, right? It's only kind of temporary buffer. But, apparently, for those that haven't worked in this area, it also has legal ramifications, right? If you're a system that stores data, you have to comply with a bunch of regulations. You try not to have all your systems have all these extra efforts that you have to go through.

A good example would be things related to, “You need to retain data and you have a maximum X-number of days.” In Kafka, we’re going to delete it in four days. So we don’t have to worry about if there’s a legal maximum how long you retain that as long as it’s longer than four days.

If we did keep it longer than four days, then we need extra systems to be able to track and that we comply with regulations. By the way, we do. There are other situations where Kafka may keep something slightly longer. Then we have an extra system, goes and tracks and makes sure that it gets deleted if required. Then under whatever circumstances, we do the right thing. So, yeah, storing infinitely is not trivial not only from the engineering side, but also from the policy side.

[00:25:06] JM: You’ve worked on Kafka for three years?

[00:25:10] NS: Right.

[00:25:10] JM: What’s something new that you learned about Kafka this year?

[00:25:14] NS: So the first thing is to say that I basically knew not a lot of thing about Kafka or this data ecosystem before I joined LinkedIn. It is amazing. In general, just the fact that Kafka itself is very simple. You give it a message, it gives you a message. How hard could that be, right? Why do you need a team so big if they just put one messaging at one message? But, yeah, it is surprising how much you need to do to get things to work correctly.

Things I learned from Kafka, there are all kind of issues and bugs that we end up solving. This like print logline of 100 megs is one of those thing where I learned about Kafka. But normally we end up trying to create new stuff. So at this point, there’s not much about Kafka that surprise us. We’re actually trying to set the best way to predict the future is to invent it and in this case in the Kafka broker side, we try to move forward and are driving stuff that we need.

Surprising things, things that work, things that don't work, but nothing in specific. I mean, I can talk about bugs and things that we deal with on a day-to-day. For example, we decided to do something incorrectly where there was a variable that got exposed in new Kafka clients that changed the time out and we didn't expose it and it changed the behavior from the previous clients and we have to change it back and created some issues, or the fact that under some conditions, partition moves take longer than it should. Bunch of stuff that I'm happy to talk about, but I know the detail would be – If you want to, I can go into detail. But I don't know the best for all your listeners.

[00:26:34] JM: Let's talk about one specific engineering feature of Kafka, and that is exactly-once processing. Can you explain what exactly-once processing means in the context of Kafka?

[00:26:48] NS: Yeah. So first is let me start by saying that we don't use exactly-once processing for Kafka. Even though we participated in the initial discussions when Confluent, and a.k.a the community, was working on this. We had numerous video calls and we go back and forth and we reviewed and talked about these things. We ourselves don't use it. There's a number of reasons why. But let's talk about exactly-once kind of processing. Kind of implies that there are some data you send to Kafka. It will get processed once, and that's it. One and only one. Not zero, not two, not three. So you can do things like transactions, right?

Obviously, you can do things that happen only once. You can do maybe financial processing. Things that we care about getting the exact number, etc., etc. So that's such a powerful feature. But it comes at some cost from the system. In the case of LinkedIn, the parts where we have simple systems, they don't require exactly-once processing. So it comes kind of a tradeoff of performance to some degree.

But in the cases that we do require this sort of these, our pipelines are more complicated than a single one transaction. So having this multi-stage transaction support gets very complicated, because until you have the guarantees to your input and output systems, you will not be able to guarantee the exactly-once.

Because of this requirement, for us, we end up building this sort of behavior at the immediately higher layer, right? So I like I mentioned, we have databases that replicated via Kafka. They care about not sending stuff or processing stuff or replicating stuff twice. So those systems are built with this in mind. From our cases, they are stream processing frameworks. We also build with this in mind normally because there are multiple stages. You never know who's going to consume and produce. At the end, we guarantee the exactly-once will give you – Will not be the exact thing that you really wanted to achieve.

So for simple systems, such as single hop and are trying to do just the single process nuance, they're very useful. So in this perspective, it's a very useful thing for Kafka streams, right? So it can do these sorts of things. But for our pipelines that are a little bit more complicated, we can't really take advantage of that easily. Not to mention that we're more like a tanker, right. So even if we could in some minor circumstances, turning us to kind of use a new feature is not as easy.

[00:28:53] JM: Let's move the conversation towards streaming frameworks for a while. What's the purpose of a streaming framework?

[00:29:01] NS: So, streaming frameworks normally refer to stream processing frameworks, right? So it's a way in which out of a stream ware data, you can process, augment, filter data. The frameworks allow you infrastructure way that provides you tools. So this is easier. So you don't have to build everything from scratch. In our case at LinkedIn, we do a lot of stream processing. It is one of our main ways to deal with our data. We're very familiar with stream processing. We use Samza, and it's a common available tool just like Kafka is.

[00:29:29] JM: How does a stream processing framework compare – I can think of two comparisons. One would be Hadoop MapReduce. So Hadoop MapReduce can process large volumes of data in a singular batch fashion. Another comparison might be I could write a simple Python script that literally pulls data from some source and performs an operation on each data point from that source. Could you contrast streaming frameworks with those two examples?

[00:30:02] NS: Yeah. So we have that discussion regularly both in our offline team and our near line team, so into how things contrast and map. At the end of the day, there's a notion to this, this big gap and disagreement. But the truth is we do things very, very similar. Trying to find the places where things differ is in itself an interesting thing. People normally would say that for streaming frameworks, you have a process that continuously run and processes infinite amount of data with no stop. The best side that you have on your finite data, it is done, right? But both of those can be obviously used in similar ways. You can just run your batch job numerous times, or you can stop doing the stream processing.

In our case, in the case of the MapReduce, there are some computations that are more efficient. If you use MapReduce and if you know the whole dataset, you can immediately optimize. You know how much you're going to need. You're doing a matrix multiplier or some other thing.

[00:30:52] JM: Like if you did Hadoop MapReduce.

[00:30:54] NS: Yeah, in a MapReduce situation. So knowing all these in advance allows you to make some optimizations. On the flipside, on the stream processing, other advantages, let's say latency. You can compute intermediate results and use those. But they do have the requirement that whatever computation you're doing has to be possible to be done incrementally. If it's not possible to have an incremental computation, then doing the stream processing situation will be hard.

In the case of a Python script just running and computing, it is kind of like a stream processing thing, but without a framework helping you. In this case, for example, for Samza, Samza will help you do store intermediate results? If your process dies, what happened? Did you lose half of your competition or can you restore and continue? So stream processing frameworks will help you do all these.

They'll provide you with a high-level abstraction, and you can write less number of lines of code, debug easier, because a framework will do all the hard lifting for you. All you need to know is like the higher level thing. The example being, if you provide an API, let's say, in SQL, which is kind of the hot thing right now, you just write some SQL. Then the system underneath will figure out how to do everything else for you.

[00:31:55] JM: Do those stream processing systems, they also parallelize the data so that data would be processed faster?

[00:32:00] NS: They do. So the different systems will have different requirements. Depending on how full feature your system is. By the way, all modern systems kind of like – They don't copy each other, but they all kind of went in the same direction, right? They're all trying to make things easier. In the case of Samza, but it will be similar for Flink or [inaudible 00:32:15].

The first thing that your system might end up doing is splitting the data in the right way so that your next computation happens correctly. The example being, if you're trying to compute something where you want to know something on users grouped by country. The first thing that might happen if your data is not grouped by country, you'll split it up into by country. So in our case, we would call this a repartitioning job. Streams come in, organized in whatever way. The first thing that happens is they get reorganized so that streams come out per country. Then you can perform with locality, with obviously "caching". The competition on that like per country thing that you want to do.

It may be that one country has more than others, and that one has to be done in two locations and then join. I mean, there's plenty of things that a system can do. Having a framework will abstract this out from you and it will figure it out.

[00:33:03] JM: My understanding is that Kafka is often used as a platform for storing data that stream processing systems are going to read from and write to. So you might have a situation where you've got all these raw events that are being created within Kafka you could have stream processing systems that are reading from Kafka, sucking in those events, processing those events, perhaps refining those events and then writing them back to Kafka in a cleaned up fashion or in a materialized view fashion. Could you describe the interaction between Kafka and stream processing systems at LinkedIn?

[00:33:47] NS: Yeah. So what you described is actually correct. Our stream processing framework like I said, Samza, uses Kafka extensively but are not Kafka specific. It can read off of Kinesis, or obviously Event Hub, but it can also read off of HDFS. The source of the data itself may not specifically matter. We can actually read off of like our database is Oracle or Espresso, but that we do because we first we send it to Kafka and then we read it off. Then we use Kafka as an intermediary way to store any temporary results or between stages of the computation.

We also use it to store, let's say, checkpoints. So we can like revert back or we can kind of restore if one our set of nodes failed. So Kafka is kind of integral of doing this like stream processing back and forth, back and forth, back and forth. Depending on the job that you're writing and the framework that you're using, you try to optimize and not to have to redo work or not to have to go back and forth to Kafka a lot. But it is one thing that we do to be able to continuously compute and then move forward.

Like I said, Samza would just assume that we use here, Apache Samza to be exact, relies on Kafka for a lot of the things on the internal path, whether we're doing, like I mentioned, SQL, or some of the native Java APIs. It goes back and forth to Kafka.

[00:34:57] JM: Do you have a good maybe example or use case of Kafka plus Samza in action that comes to mind?

[00:35:04] NS: Right. So everything that I mentioned before kind of works. I mentioned, let's say this thing that you're trying to get a notion of, like the top view page at LinkedIn, the top view page maybe per region. First, we will figure out per regions. But even just the top view page by itself is already an interesting thing.

The data that comes into Kafka, because there's a lot, of all webpages are being viewed may not be initially distributed by page that you're viewing. We're just going to say article. Let's say you're looking at the published articles on LinkedIn. It will be per like geographic region or per user or per – We're actually distributed, right? So we spread it out.

But if you want to compute something per page, then you have to perform this computation of like dividing and calling all the pages, computing how many times this page has been see in the last hour or the last 10 minutes or whatever window you're trying to achieve. Then aggregating that data and figuring out what is the top one that gets used. There are multiple ways to [inaudible 00:35:56]. So there are multiple ways you can imagine this computational [inaudible 00:35:59] of getting the data that you want. The framework will calculate it for you. Then we output that out normally into another Kafka topic. Then that can be used to serve like content that we end up sending.

Also, an example of how some of these things are done relates to security. Without giving any specific algorithms, we look at attack vectors, right? So if we somebody sending too many messages, that might be somebody trying to spam. We'll detect that with a Samza job. That's looking at all the things that are being sent, computing averages. See if there're outliers. Things that we need to look into and then potentially issuing output that the separate system will then either take a look at it closer or block. Let's say we might want to block a user or something along those lines.

The same thing is true for everything. So at LinkedIn we use stream processing quite a bit. It is a third of our Kafka traffic I believe. We have 3,000 jobs to do stream processing. So, yeah, it's something very, very common. That's actually one of my headache, because big clusters are maintained to do stream processing.

[00:36:55] JM: There are a lot of different streaming frameworks. My understanding is that LinkedIn has mostly focused on Samza, because for similar reasons that LinkedIn has focused on Kafka. Samza was I think built within LinkedIn around this time when there was kind of a golden age of Hadoop infrastructure. People were just building a lot of new things. There was a lot of movement going around. But does LinkedIn also use other streaming frameworks, like Kafka streams, or Apache Spark, or Apache Flink? Do you have an understanding of how these different streaming systems contrast with each other?

[00:37:33] NS: Yes, we don't use other streaming frameworks. Although in the case of things like Spark, it doesn't have to necessarily – So we do use other frameworks to do data processing. But in the case of streaming, we use Samza exclusively.

For some jobs that do "stream processing", they actually might not rely on Samza. They might do stuff by themselves for some very, very specific particular reason. But everything that we do is normally Samza in the stream processing arena.

How they contrast to each other? I think a while ago, when things started, the contrast was big, right? Some would focus on like stateful processing. Others would focus on like latency. Others would focus on API. But as time goes by and all the communities kind of like cross-pollenate, then we always end up kind of in a similar situation.

An example, and I think you did have in your previous podcast, you've talked about Beam in a number of places. But Beam coming out of Google, which is basically this like layer on top of some frameworks both online and offline is something that we thought was very useful. Now, for example, Samza is one of the supported runners for Beam.

So it is part of the community. Internally, we can use Beam as well. So now some feature that's offered by Beam then is just available on Samza as well. That's true for Flink, and that's true for Spark, and that's true for many other things. Yeah, that convergence is quite bit.

At this point, for anybody choosing a new system in the stream processing arena – Originally, when iPhone and Android came out, the best answer were like use whatever your friends are using, because you'll know who to ask questions if you need support. In this case, if you're familiar with one, you're probably pretty close to what you need. You can use that one. But, yeah, they're all kind of like self-equivalent.

For us, we love Samza, so that's why I can continue talking about Samza or why we think it's the good option. How we're innovating and pushing stuff forward. How we focus on stateful processing. I don't know if [inaudible 00:39:22] for Samza is public, right? But obviously we're trying to make things better. We rely on it quite a bit. For us, it's a really actual mean and it's a production system that is used like to serve a bunch of the – Like I just mentioned, like threat detection. For us, that's very important. I can assure that we maintain it and it runs. If anything is not working, we'll fix it.

We also want to make sure that like whenever a node fails, we restore as fast as we can. So if you imagine a simple system where a node goes down, if that restore of that job that was happening there takes minutes. That might be too long. We might want to switch in seconds or less. We'll continue working on these features. So if anybody wants to join and use Samza, feel free to come to our meet ups or our discussion or forums.

But having said that, if you like Flink, Flink has the advantage that you have companies behind it that support it. If you want to use Beam, you have like the backing of Google to go on like

Google stuff. So it depends on your other factors. But in terms of like functionality per se, we're happy with Samza and I think they're relatively equivalent.

[00:40:21] JM: Some questions about management. So you manage a 35-person Kafka team. Is that right?

[00:40:29] NS: Kind of. The SREs don't directly report to me. So 25. Let's say 25.

[00:40:32] JM: Okay, 25. What's the hardest part of managing a 25-person Kafka team?

[00:40:37] NS: I don't know. That's different from managing 25-person anything else and engineering. There are some young people that need to be mentored and coached. There's some like more senior experienced people that want to drive and set the agenda, and there's kind of the balance that we have with the goals of the company and what needs to be done and what we need to achieve.

So far, at LinkedIn that has been relatively easy in terms of like mapping the goals with what we're doing internally, and our management change has been very, very supporting. I always say, "You know what? We need to work on this, or build on that, or do that kind of stuff." So it hasn't been that big of a problem. We have lot of personalities. We have very smart people. You have to make sure that everybody gets along. I think they are regular management jobs.

In terms of technology, trying to get to a place where we can plan for the future, it's just a matter of like bouncing your resources, right? I kind of joke in my job at LinkedIn, is to make sure Kafka runs today, and then to make sure Kafka runs tomorrow. It's like how much do you balance on fixing issues today versus planning for the next feature to scale? How am I going to let Kafka scale another X-number of times in the next number of years?

By the way, Kafka, even though it's a matured system, at LinkedIn grows continuously. It sometimes grows faster than we immediately expect it. Not to mention that we have tolerate peaks and valleys and all that kind of stuff. So it is an interesting job.

[00:41:57] JM: Are you still writing code?

[00:41:59] NS: I don't write code directly. I don't write Java code directly. I sometimes script reports, because I need to get some data. I'm always teaching some of my younger engineers to kind of like do the shell more, even though a lot of people just prefer like UIs, which is kind of funny. I review code, and I review a lot of the designs. So you're asking, because you know my background is on the IC part. So I normally spend my time on design review more than code review, or for that matter, writing code.

[00:42:25] JM: Was that a hard transition to make for you? Because I know there are some managers who have trouble giving up the keyboard.

[00:42:31] NS: It is hard. So I've never been – I've been a competent coder, but out of my peers, I've never been the best coder. So giving up the code is not the harder part for me. I've always wanted to achieve more than what you can do by yourself. So that always means working in a team. Here we have a great team that delivers. But they're very dependable, and we have a great charter. People know us. It is relatively good.

Harder part is figuring out how to map the mundane stuff, right? There was a discussion when you think about systems, most of the people out there are not writing new code or new systems. They're making sure that the current code works well, right? That is something that we don't recognize enough. I think it's something that both managers and engineers need to think about. Not only when you're doing and working on code, but even when you're writing new code. Highly likely, you're not writing code for yourself. You're writing code for somebody else to maintain, for somebody else to look and debug. So kind of helping all the young engineers do that is kind of one of the important things to do.

[00:43:28] JM: This is kind of why I actually left the software industry, is because I really didn't want to maintain legacy code. I just didn't like it. It wasn't fun to me. To me, it was not – I was just told this is what software engineering is. I was like, "Maybe this is your definition of Software Engineering Daily. I don't want to maintain your legacy code."

I've had some really interesting conversations with Facebook managers, actually, because I was interviewing a Facebook – Actually the guy that manages the React team, kind of the open source React Team. His team was very, very happy. I was just trying to understand who does the work that's not fun? There's always work on an engineering team that's not fun. To me, it does seem like a huge measure of a manager's skill to be able to allocate that work in a conscientious fashion and know that if you're allocating legacy work that's not fun to somebody, you are telling that engineer, "Look. I understand this is not the most fun thing in the world, and I want you to know that I recognize the fact that you are working through this not very fun process."

[00:44:43] NS: Absolutely. So having that conversation probably helps a lot just as a start. Then achieving that balance is not easy. I try. I don't think I'm like amazing at it, but I think at least we still feel like we're all in a team and we're all in it together. Having said that, what is boring for some person may not be boring for another.

[00:44:59] JM: That's for sure.

[00:45:00] NS: Right? As an example, even on not just code maintenance, let's say documentation. Some people actually like to document code.

[00:45:06] JM: Some people love it.

[00:45:07] NS: Some people like to do good, like technical writing. I really appreciate those people. I appreciate when somebody on the team will come and deliver a good document thing. Sometimes they love to do it, right? Of course you can also go and document and realize, "Hey, this code could be improved." Then you end up wanting to improve the code before the document.

In the case of maintaining code, I don't think it's that – Maybe I'm a bad example. I don't think it's that bad for multiple reasons. In the case of Kafka, first of all, all our problem seem interesting. When something fails, first we have to like do the detective work. It's like film noir and like, "Oh, the node failed. Who killed it?" We have to go and figure out why it died. Was it poison? Did the other node kill it because it sounded too many – I don't know. Like just doing that investigation.

[00:45:51] JM: What's the log message?

[00:45:53] NS: It was the log message. I mean, we joke, but there's a sense of achievement when you understand what the problem was, right? So getting to that point I think is very rewarding.

Second, the maintenance part of like – There's a book. How do I solve it? I find it exciting. Again, I don't know that I'm a representative here, but it is basically trying to solve a problem given some constraints. The constraints are you can't change the whole system. You're limited by this, right? It's kind of like you're in a escape room and you have some limited options here.

What can you do in a reasonable amount of time that is going to deliver, solve the problem? Maybe it's multi-stage. Maybe you need to find like the immediate solution today to like mitigate and make sure that side is up. Then you'll find the long-term solution that will do the right thing. Sometimes is there a long-term solution in like two quarters in the log time solution? You know what? We rewrite this. We are never going to do this ever again. ZooKeeper being a good example, right?

So at the time it was the right solution, and now we're trying to move away from it. Do we consider moving out ZooKeeper for Kafka maintenance? I don't know. Some people might consider maintenance. Other people might say, "Man! That is really exciting. Some of it will be maintenance, like how do we like plum some code here, some there? There'll be some part that is exciting, like, "How do we do Raft, or Paxos, or what else do we use to replace how ZooKeeper does coordination? Do we use a library [inaudible 00:47:11] existing open source library we can just load, or are we going to write it from scratch?"

Do you really want to tackle like the mathematical proof of correctness in the code that you write, or are you willing to delegate and just use like some library that somebody else did? In which case, then you have to debug the library. So that's a plus or minus. I find it all interesting. Still, there's still stuff that you have to deal that are less mundane and it's just like maintenance.

Yeah, for that part, sometimes you just have to do it and we'll distribute the work. We'll to have fun. We'll celebrate at the end of the day. We'll celebrate and recognize the achievement when we got everybody to like deprecate code that nobody wanted to use and nobody wanted to maintain, and you force some team whose job was already working to change something even though they didn't want to.

If you feel you're on the same team, I think you celebrate together. If you don't feel you're in the same team, you all suffer together. At least if you feel in the same team and you're suffering, it's still good. It's when you don't feel you're at the same team and you're competing against your peers where you come in to this like everything is horrible and like it's not worth it to do that.

[00:48:12] JM: When you make an improvement to, let's say, a Java client, a Kafka Java client. Assuming that is work that your team does. Let's say you just improve some failover case in a Kafka Java client. Do you go through all the codebase in LinkedIn and update those clients or do you send an email to every team that is consuming that client library and say, "Hey, we've got a new update."

[00:48:43] NS: Were you prompted for this question?

[00:48:45] JM: No. Are you doing this right now?

[00:48:46] NS: Yeah. No. First of all, at LinkedIn, we have a process called horizontal initiatives, which is a commitment company-wide that some percentage of your time may be required to be spent on these type of upgrade tasks. So, for example, let's say that we have a library or we're deprecating, whatever, Python 2. Then anybody that has Python to cod will get notified, "Hey, you know what? This quarter or these two quarters, you have to drop it. Move over to Python or whatever it is that we're deprecating."

In the case of Kafka, we have a number of abstraction layers. Some people use Kafka directly and use the Kafka clients.

Some people use our simplified kind of Kafka client, and we get to modify that code. That code, the client code is something that is available for anybody to pick up. What's end up happening is what if I need everybody to pick up that new version because it has something that let's say makes our system more efficient, or in this case, migrates over.

The reason I mentioned you were prompted is because one of the big things that happened with Kafka 2.0 is the community decided to deprecate what we call the Scala clients or old consumer. These are clients that relied on some Scala code and they have some specific behavior. They don't abstract a lot of the internals to Kafka to the user code and it was the right thing to do for the community. So upstream deprecated these. We could not, or we could not in time to deprecate that.

So now we're going to this thing where we want to deprecate that code and it will take us a long time. I'm happy to go into the details, but involves a lot of like help on our side from a system perspective to the people that use this old code to move to the new code that we found available for a while, but it's non-trivial. So were going to spend a lot of time helping them, like live.

Because the problem is that, again, it's simple. You publish them. You subscribe. But when you have a complicated system where now there's some service that is running 100 instances. As modern practice go, now you have things like you want to do a canary and you want to test your new code. Does it work the same as your old code? Does it perform better, bad? So that means that we have to allow people to roll forward, roll back, roll forward again, kind of half way roll back and eventually deploy the whole thing. Then commit and like switch things over.

It is very involved, and it will involve a lot of people on my team doing potentially mundane work, kind of helping the rest of the company. We do it happily because we want everybody to migrate, and at the end of the day, our lives will be easier. But it is quite a bit of work.

So to answer your question in a simplified way, we don't normally modify people's code, but we support. We change our code and then we support the migration of this new thing that happens. Sometimes we do the work for them, but not always. As you can expect, we try to automate as much as possible.

[00:51:34] JM: Well, you're not alone. This is a pretty prevalent issue in companies of LinkedIn's size, and this is one way in which – I talked to Facebook, and Google, and Stripe, and actually a bit about they have this monolithic git repository model, or not – Well, they're not all Git, Perforce whatever. That allows platform engineering teams to easily go in and update libraries.

If the platform engineering team makes an update to a client library, they can literally just go through the codebase and update everything. I'm sure it's not as easy as I just described, but it's certainly alleviate some of the problems that you just mentioned. Now, it takes a ton of work to get a monolithic thing going. But that's a conversation for a different day. Any comments on that or –

[00:51:54] NS: Yeah. I was going to say that sometimes it's not just to quit itself, right? So if you imagine, then first I have to, let's say, prepare the field to migrate protocol versions. But that having this intermediate step of having half the people want to protocol, half the other is an expensive situation. You have to have a more concerted effort when everybody participates at the same time and there's some process to coordinate and make a switch. Yeah, just the code itself might not be enough.

In our case, I could look for all the code. I can do a code search and see everybody that calls, like Kafka Produce. It's not as simple as just changing. Also it's not simple, because sometimes the Kafka Produce, because we use Kafka for everything at LinkedIn, that includes like metrics. That includes like logging. That includes any monitoring. Then it is used everywhere. When we make a change, a lot of dependencies might change too. So it's not just changing our system. It may be changing all of their other dependencies as well.

So it gets kind of complicated for the Kafka case, which are at the way bottom of the stack. Again, we're going to do it. We will live through it. We will get a badge or something.

[00:53:21] JM: Okay, last question. I already asked you about management, but there are a lot of people who listen to this that are quite curious about management. What's something that you've learned about management in your time at LinkedIn?

[00:53:33] NS: So are you asking a LinkedIn specific question or a management question?

[00:53:36] JM: No, management question. Abstract management advice.

[00:53:39] NS: Right. So abstract management advice, and I think it's not just management, is know how to resolve conflicts. In this case, you have your team and maybe inside your team you could act as a dictator and then rule. I wouldn't recommend that. But when you're dealing with people in other teams and this is the agreement, know how to resolve the conflict and work together. Try to set the stage. Then if you don't resolve the conflict and things just do not work out, don't wait forever.

So at LinkedIn we followed this policy of clean resolution, which is do as much as you can. If you can't go forward, state your facts. Try to get somebody else resolve it up the chain. But in general, as a general scale, just conflict resolution is going to be the best thing that you can do. Having a good way to communicate is the best skill.

I personally try to coach people on this technique. They may or may not have a name. I think it's come out in numerous places, which is try to find the place where you agree, the first place where you agree. If you think of the decision making process as a cree of decisions, like in this case, let's say my case. Do we both agree we want the best for LinkedIn? Yes. All right. Do we agree that we want the best for whatever? Then you continue down. Ultimately at the end of the day there's like, "Do you agree that the system has to serve all our customers?" Yeah, we both agree. "Do you agree that we can split it off geographically in snow?" All right. So you found this split point. You found the part where you don't agree, and now you can discuss that.

Because it may happen when the discussion started, you are disagreeing about like the deployment code and you're like, "Oh, no. We should do this. Oh, no. We should do that." It's because you had the wrong assumptions. You were not agreeing on some previous point up the chain. So find the first place you disagree with. I would suggest going in this fictional scenario from the part where you agree to the part where you disagree. Because you probably agree about almost everything up until like one key point.

So if you start from the disagreement at the bottom you're going to be like, "Oh, I like whatever, garbage collection." They're like, "Oh, no. I like whatever. Doing native C in like kernel mode or whatever." Going from the bottom up is going to be more complicated. Anyway, so my suggestion is find a place that you agree and then find the first place you disagree. Let that be the place where you start discussing what you need to decide.

[00:55:43] JM: Okay. Nacho Solis, thank you for coming on the show. It's been really fun talking to you.

[00:55:45] NS: Right. Awesome. Thanks for having us.

[END OF INTERVIEW]

[00:55:57] JM: LinkedIn is a software company with the goal of creating economic opportunity for every member of the global workforce. LinkedIn is hiring data scientists, software engineers, researchers and many more roles for its engineering team. To find out more about the problems that LinkedIn is solving and the teams that are solving these problems, check out engineering.linkedin.com, where you can read about culture, open source projects and hard problems that LinkedIn has worked through and blogged about.

Thank you to LinkedIn for sponsoring this show and for creating software that I use every day.

[END]