EPISODE 1191

[INTRODUCTION]

[00:00:00] JM: Kafka has achieved widespread popularity as a distributed queue and event streaming platform with enterprise adoption and a billion dollar company, Confluent, built around it. But could there be value in building a new platform from scratch. Redpanda is a streaming platform built to be compatible with Kafka and it does not require the JVM nor Zookeeper. Both of which are dependencies that made Kafka harder to work with than perhaps necessary.

Alexander Gallego is a core committer to Redpanda and joins the show to talk about why he started the project and its value proposition.

If you'd like to support Software Daily, go to softwaredaily.com and become a paid subscriber. You can get access to ad-free episodes and it costs 10 a month or a hundred dollars per year.

[INTERVIEW]

[00:00:53] JM: Alexander, welcome to the show.

[00:00:55] AG: Thanks for having me, Jeff.

[00:00:57] JM: You work on Redpanda, which is in some ways a replacement for Kafka. And Kafka is an extremely popular piece of technology. Could you tell me a little bit about your perspective on what the use cases of Kafka are and perhaps what the shortcomings of Kafka are?

[00:01:17] AG: Yeah, I'd be happy to. So Kafka is typically used to do it's – It's so popular that it's almost synonym with real-time streaming. Or at Twitter, it's been used to fit basically your timeline. For ad techs is used to not only to record the ad impressions and compute click-through rates for, let's say, banks and credit card companies. It is used to compute fraud detection in real-time and to merge signals, let's say, from your digital location of the credit card transaction and like the vendor type and the trust of the network, et cetera. So Kafka is really use not – I wouldn't say it's like per industry. It's more per use case. And I would say it's maybe the most popular message bus in the world right now.

And I think what people love about Kafka is that it became sort of this like central plug-and-play system where people can glue together disparate distributed systems and have something useful at the end of the pipeline. One example that I gave earlier, which was with the fraud detection pipeline, is people typically save the data of the credit card transactions into Kafka and then they plug in something like Spark streaming and then they shove that into Elasticsearch. And with very few components, you have actually a relatively sophisticated fraud detection pipeline.

And so Kafka is really sort of this central nervous system for a lot of companies that are doing things in real time. And I could go on and on, but I would say that's sort of the main use of Kafka, is to empower businesses to do real-time, ad sort of like the real-time component to their application whether it's analytics or notification or fraud detection or advertising. It really sort of spans multiple industries and a lot of use cases.

[00:03:26] JM: But what are the shortcomings of Kafka?

[00:03:29] AG: Yeah, great question. What people love about Kafka is the Kafka API, is the fact that you can – Like I mentioned with the use case, you can plug in three or four components relatively easily and sort of leverage millions of lines of code in this massive and vibrant ecosystem, which are called – That's sort of the Kafka API. But through the shortcomings of Kafka are really about management and running Kafka at scale.

So I would say there're probably three big categories where with all of the customers that we interview that they have problems with Kafka. So the main one is really operationalizing Kafka at a scale. So Zookeeper is a really difficult system to manage, and Kafka itself is actually also a difficult system to manage. We can get into that. But that's the first category. And the second category after you pay for the operational complexity of running Kafka either by outsourcing to a vendor or by hiring distributed system experts or with their customers with downtime, the second sort of area that – Sort of the open source system has for improvement is really around performance, which is the byte, the cost per byte –To run a byte through Kafka is relatively high. And so what large companies actually end up doing is that in order to keep up with their load, they often have to over-provision by a significant margin, say, something like 100% percent over-provisioning. So 2X the amount of hardware needed is not uncommon. Then the last one is data safety, which is by default, the settings of the Kafka broker – And so Kafka, really, it's a large ecosystem. So without getting in too much into details, and we can, there are some settings which you might experience data loss. So I would say those are probably the three large categories of kind of improvements and difficulties that people are having.

But by and large, I would say just running Kafka stably at a scale in production, right? Making sure that you're getting sort of the predictable latencies and understanding how to handle failure modes of Zookeeper and Kafka when no brokers crash or when you need to do a rolling upgrade. I would say that's sort of the largest area of improvement for the project.

[00:05:53] JM: So I can agree that those are some shortcomings particularly over-provisioning and complexity of running Kafka. But you do get a battle-tested piece of software for a mission-critical workload. Why would you use anything but Kafka?

[00:06:13] AG: Yeah. I think what Kafka basically just doesn't deliver on the promises it's is when people start to have problems with Kafka, right? And Kafka is a battle-tested system, and I think it's fantastic. I think what Kafka really added to the world was really the API. And let me give an analogy I think to help drive the discussion. If you look at, say, SQL as sort of the lingua franca where you can plug SQL into Cassandra or, really, even to Snowflake, right? A totally like different semantics. And then also to MySQL and Postgres. In that way, SQL became sort of the way that you interact with a bunch of databases. But there were improvements on this databases themselves because they didn't deliver on one vertical or another, right? And so sort of to go with this analogy for a second and then I'll come back into

Kafka in just a second, is that the reason NoSQL kind of be became a thing is because they figured out scalability, right? But now you have new SQL databases like how CockroachDB where they promise both the same semantics that you get with Postgres, but now you get geo replication. And similarly then for the streaming, I think that the Kafka API, not the implementation itself right now, but just the actual API and the millions of lines of client code that has been written for the enterprises and open source system, that I think is the think that won in the open source world.

And what kind of my end goal on it when we started this was we can provide a better engine for some guarantees. And so a particularly difficult thing to attain with a JVM system is flat tail latencies, and you need that kind of predictability for particular use cases. So fraud detection is one where we're seeing some pick-up where people expect when you transact a credit card on a web form, you sort of expect to get a text message from Chase or Visa or whoever is your vendor, say like, “Hey this credit card transaction is either above the threshold or is coming from a flaky vendor and so on.”

And so to achieve that with Kafka is relatively expensive for some of our customers. And so they look to us to kind of reduce the hardware footprint, but still deliver on that. And then there's a separate use case in Wall Street, which we're finding traction in, which is the sort of post settlement houses. So you have some hedge funds where they are settling, say, a billion dollars in a day or in a week, however long their trading period is. And what they demand the most is safety.

And so sort of a big improvement that we brought on to the Kafka ecosystem was the ability to run workload safe with zero data laws, with a proven protocol, and we'll talk about in a second. And so for them, they want to leverage the open source ecosystem. They don't want to change their applications, but what they want is sort of a better engine. And so it is true that Kafka is battle-tested and it's actually tested. I would say it’s the most popular streaming system in the world today by a large margin. Almost like 90% of all the companies that we talked to have either exclusively Kafka or Kafka plus something else.

But in a similar way, how people looked for different backend database systems to sort of this SQL layer, we see Redpanda actually enriching the ecosystem for some use cases. There're some use cases where we compete head to head, but there's also a large non-overlapping set of use cases where we provide different guarantees, like stronger guarantees, for example, around data safety, no gaps in your logs and a different consistency on the data replication protocol and lower tail latencies.

So I think when you add richness to Kafka and you sort of decouple the technical implementation of Apache Kafka itself from the API that all of these clients are using, you actually have a much richer picture of what the streaming landscape looks like. And if you even look at Apache Pulsar as the second popular Apache project that is doing real-time streaming, they actually now also added a Kafka API that runs Apache Pulsar in the backend. That we were sort of the first company to realize that there is like two levels to the question of like what is Kafka, maybe more But for simplicity, there's two levels. There's the API, which is that protocol, that kind of lingua franca where there're thousands and thousands of applications and millions of lines of enterprise code already written against it. And then there's the implementation. And so I think that you would look for alternate implementations when the upstream project doesn't deliver either on operational complexity or on performance or on data safety, which are sort of the three tenets that we are focusing on.

[00:11:35] JM: So you've described the Redpanda API is very similar to the Kafka API. Can you talk a little bit more about what the run time of Redpanda is?

[00:11:47] AG: Yeah. It's always a fun question coming from – Because when I started this project, I've been playing around with low latency systems for a really long time. So I really enjoy this question. So the observation that we had in 2018 was, "Okay, we need better guarantees for different systems, for different use cases, et cetera." Like no data loss and super flat tail latencies. So let's build a new storage engine from the ground up. But there's actually a larger overlapping technical observation here, which is Kafka was originally released in 2011 as open source, which means that they were working at LinkedIn on it I think in 2010, probably the whole year. And I'm going to tell you kind of the story of streaming and then Redpanda and how we fit in together in sort of this larger streaming ecosystem.

But over 10 years ago, there's really two fundamental hardware changes, and then I'll tell you how streaming sort of overlaps to take advantage of the changes in the real world. Around 2004, the cost per terabyte was somewhere around $2500 per terabyte. And so fast-forward to 2011, a big part of building Kafka was actually to leverage cheap hardware and spining disk. The compute bottleneck in 2010, 2011 was really shoving bytes from the CPU to spinning media, right? And there's like all sorts of scheduling algorithms and like things that people – Very sophisticated things that people wrote about it to leverage spinning disk.

If you fast forward 10 years later, the cost per terabyte went down to about $200 per terabyte, right? So over a 10X cost reduction. But even more fundamentally, there were three things that have been addition to that. One, CPUs actually stopped getting faster. They technically got around 3X faster in the last 10 years. But what happened is that you got in 20 times more cores on a physical computer than you did 10 years ago. So that's a big physical change in the world.

The second one was that the performance, the speed performance to write a single byte to disk went up a thousand times from a spinning disk. So it's a huge, huge performance gap. And then the last one is that networks got much, much better, right? So you could do now 100 gigabits per second as opposed to a gigabit per second then.

And sort of the initial thoughts and sort of the architecture and how the runtime differs is that we looked at the real world and we said like, “Well, what has actually changed in there? Why aren't we getting this performance that software promises?” And we're stick up in performance between what these popular systems give us and what a new system built from the ground up with no code sharing at all, just like Emacs and Gcc. That's how I started this project. How do we leverage modern hardware to do better?

And so the runtime of how we're built is that we're built upon a concept called a thread per core. And so these concepts are largely borrowed from the financial markets, right? These high frequency tradings back in like, say, five or seven years ago, we're using very similar system. Now they use FPGAs and 700 nanoseconds per trade. But the architecture of how we build the system is we pin a thread per core. And this has significant runtime performance improvement. I'll tell you why. I'm going to give you first the categorical differences and then I'll tell you how they interconnect with each other to deliver a system that is 10X faster on the tail latencies.

So the first one is threat per core. So why is this impactful today versus later? The bottleneck in TPU, I mentioned, was CPU, was IO 10 years ago. But actually the modern bottleneck today is the CPU, which means so the cost of sending a method. So let's say now you have a motherboard on your computer and it has two number domains. So let's say two physical CPU sockets. The cost of sending a message between on the same motherboard between one CPU and the other CPU is similar to the cost of making a network a round round-trip with like let's, say, RMDA to a nearby computer especially as core contention sort of increases.

So what this threat record architecture allows us to do is it allowed us to keep to eliminate or alleviate a big part of the pressure of what modern hardware is giving software architects and engineers that people aren't leveraging. And the fundamental thing that we could do now that we couldn't do 10 years ago is that we need to focus on CPU as the primary cost to reduce the next bottleneck. That is currently the bottleneck in storage systems really.

And so that's sort of the first base layer, which means that cross-core communication is explicit through a network of SP executes. Let me pop this back for a second. So that's the base of how we architect the Redpanda. So a thread per core – What that gives us from an application perspective is it gives us cache locality, right? So we start to eliminate some of the compute bottlenecks.

The second thing that we did from an architectural perspective is actually trying to bypass the generic Linux scheduler systems, right? And so maybe the most impactful one that will be most clear to anyone that has done streaming with some of these open source projects is the page cache. So when you write a byte to Linux to file using regular like read and write APIs through the Linux kernel, it actually first goes into a cache and buffer management layer. And then when you call flush on the file, then it actually makes it to disk.

What we did instead is we bypassed that entire and very sophisticated subsystem of the Linux kernel and we talked it straight to the hardware. So we use DMA to bypass the page cache. Why is that sort of a significant architectural departure from Kafka and similar systems, is the first one, it gives us predictability. And this actually surfaces through products in in things like fraud detection. And why does it give us predictability? Well, we understand exactly the throughput and latency that a particular NVME SSD device can give the application. And it allows us to build custom code for this particular use case. So the Linux scheduler, in particular, the page cache algorithm and eviction strategy and so on is a very sophisticated generic scheduler, right? And so when you read a file, it'll try to read ahead a little bit. When you write a file, it'll do it write behind, which means it'll buffer some things in memory for a little bit before it actually writes them to disk and so on. Because we bypass that entire mechanism, we're able to build the custom fit algorithm to talk it straight to the hardware to solve this particular use case of the streaming. And so I think those are the two building blocks of how we differ at runtime from the other systems.

[00:19:22] JM: Now, the easy question for me to ask is all of that sounds like interesting improvements on Kafka for some use cases, but unfortunately it's really hard to get people to use new technology when the old technology works pretty well. Why would you be able to get anybody to use Redpanda?

[00:19:44] AG: Yeah. So there're basically two camps. Let's talk about I think in each segment so we can understand why people would use Redpanda and why they do use Redpanda today. So on one end, we managed to put – By the way all of this complexity and the complexity of Zookeeper and Kafka into a single binary. So we don't need two systems. Like part of the operational complexity of Kafka is that it’s actually the interaction between three or four components, but there is the people, the users that are writing to Kafka. The interaction between Kafka and Zookeeper, and the users that are consuming from Kafka.

And so what we managed to do is of course we can't eliminate the consumers and producers. So let's leave them out for this discussion. But the architectural complexities of running two systems, we managed to combine it into one system. And so for some people, the burden and operational overhead of running two systems that are both difficult to actually run at a scale is good enough to transition. And so I would say those are the classical enterprise use cases. And so that's one group of users that we handle.

Some are for new use cases that we enable. In particular, because we're single binary and we're super mechanically friendly. Like we can use very little memory, we give predictability. We're seeing a lot of use cases that Kafka doesn't handle well. And frankly a lot of the JVM systems don't handle well, which is actually product embedding into IoT and edge devices. So I mentioned that Kafka as the lingua franca for streaming API, right? So if we decouple again the implementation from the API, people love using the API. And the reason they love it is because you have sophisticated client APIs and you have like this entire richness of ecosystems that you can just download from open source and plug and play and you can build a really good and sophisticated product with a bunch of open source components. So people love that. But the runtime overhead of embedding Zookeeper and Kafka through JVM systems to different configurations and security settings and all of that on edge computing is cost prohibitive from a computational perspective.

And so that's actually an entire new set of use cases that we're seeing in the wild, is security appliances embedding us into their applications for doing like for recording the logs of like intrusion detection or intrusion prevention. I would still saying this is on the classical side of Kafka thing. I mean, also, maybe a think that gets most people excited is what are the other things that we do beyond the Kafka API? So up until this point we've only covered the base. When you start a company, you kind of have to start with an idea and a product. So that's what we started. But as we've talked to probably a hundred enterprise or more right now, we've discovered two new use cases. One was the embedding. And then the second one was inline WASM.

And so when you ask who runs Kafka, it’s mostly enterprise users, like big enterprise users that either have the money to pay for a large distributed system, expert team to run and maintain Kafka, or people that offset to a vendor, right? But generally they have kind of a ton of money, and so the classic of big enterprises. Those are the majority of users that are on Kafka.

What we've discovered is because Redpanda is so easy to use, again, it's a single binary, the JavaScript and Python community are really welcoming us with open arms, because up until this point if you ask maybe less sophisticated users from a technical perspective or people that are actually more interested in solving business problems then running infrastructure, they're like, “Oh, this is great. It's as easy to run as nginx.” And no one really complains from running nginx.

So I think there's actually an entire market and category of programmers that have largely frankly been ignored by stream processing technologies. And I tell you this, as someone that built themselves the computational framework to Akamai in 2016 where we were also focused on the large data, sort of the large enterprises. And so what we stumble upon accidentally is that by being so resource friendly on a computing level, we actually opened up to the JavaScript and Python community in particular. It's easy. t's it's not JVM. This community really doesn't want to become the distributed system experts. They want a system that just [inaudible 00:24:33] and it gets out of the way and it's one binary and you can just do app get installed. So that's a large community for us.

There are new use cases like embedding and delivering on flat pay latencies for fraud detection. And then there's the classical Kafka systems that we also compete. And so I think there's really a lot of motivations for people to kind of change over. And that, we haven't even covered sort of the things that people do once you get started with the Kafka API system.

And so one of the immediate question that happens is like once you get data into Kafka, well, how do you get it out of Kafka or like what are the useful things to do with Kafka? And I think we have a particularly neat solution where we solve about 60% of streaming use cases, but we're not sort of trying to solve 100% of the use cases with this particular feature, which is inline Lambda transformations. So what are they?

For users that have a relatively small footprint, let's say three, five, seven nodes and so on, which is like I think almost the majority of the market. When you kind of want to do stream processing on top, what happens is you have to send up a separate system, like in a Spark streaming cluster or a Kafka streams cluster or Blink cluster and so on. It's a separate hardware, separate configuration profile, separate file domain to read data from Kafka and, for example, save it to Elasticsearch.

And so what inline Lambda transformations allows folks to do is that this storage engine, you can upload a little JavaScript function uh or anything that compiles to WebAssembly through the storage engine and will run this code as data as coming in. And so for use cases that really require flexibility, this is sort of a really easy solution to eliminate an entire other system of complexity. And so what are the use cases? The majority of users doing streaming, what they do is you have a JSON object and you kind of plug two or three fields. Let's say GDPR compliance is really useful as an example. You get a JSON object with a bunch of private information and you want to remove the social security number, maybe obfuscate the address a little bit or something like that. And then you want to save it back on to another stream for your machine learning people to leverage this information.

And so currently as it stands today, folks have to send up a separate cluster. Consume the data. Plug a couple of fields and then save it back into the queue. And so by having inline stream transformations, people can just upload a simple JavaScript function and then it guarantees that computationally add the storage engine. Two examples that are useful here to drive the point, we talked to a food delivery company, they're massive, and what they are prototyping our WASM engine for is actually to guarantee GDPR compliance.

And then there's another health provider company, and what they do is they actually want to remove all of the private and identifiable information and just simply open up all of the streams from their patients, because for them, what is useful is understanding the symptoms, prognosis and – Sorry. Diagnosis and prognosis of particular treatment in large. And so they just want to do – They want to train their machine learning models, but by us, Redpanda, allowing them to send inline stream transformations that do very simple things. Again, only like 60% of the use cases. It sort of gives this health provider company streaming compliance.

So we can give their chief security officer computational guarantees that say any data item that goes through this data stream will get processed through this lambda function. And so you don't have to worry about GDPR compliance, because you've sort of solved it structurally with inline Lambda transformations. And as far as we know, we're the only engine that accepts this WebAssembly, which allows a much richer ecosystem. Really any programming language now compiles WebAssembly. And so I would say I think those are kind of where we start to depart away from the classical Kafka installations. Does that help?

[00:29:06] JM: That was a great architectural breakdown. One thing I do want to point out is I think that hasn't Kafka removed the Zookeeper dependency?

[00:29:17] AG: Yeah, great question. So they are working on a KIP called KIP-500. Super famous. So they haven't yet. I think they plan to in 2021. But let's look at sort of the differences between Kafka and Redpanda from a protocol perspective. So this is going to get super nerdy right now as if it wasn't already.

So KIP-500, what it says is they're going to rewrite the controller, which is a role within the Kafka ecosystem that handles things like topic creation or topic deletion or increasing the cluster size and so on. The replication mechanism of the controller is going to be Raft. So what that doesn't actually solve is the number of fault domains. For small clusters, they will – So what does that mean? It means that actually instead of having a single binary, they still have two separate binaries, which means you have two fault domains. So the number of fault domains is the thing. They have improved sort of the reliability of Kafka by eliminating Zookeeper, because then this controller becomes the source of truth rather than the source of truth through the proxy of Zookeeper. And there's a great talk by Colin McCabe, who's I think wrote KIP-500 on Kubecon San Francisco where he talked about really the low level detail and motivations of this and the particular fault domains that even to this day, by the way, sometimes the only way to get out of a Kafka pickle in terms of a configuration or a particular failure semantic is to go into Zookeeper and actually remove the Kafka controller node, right?

So there are very complicated distributions failures that that will do. What that will not do is it will not solve the two fault domains problems. Like you will have two configurations even if there is the same configuration, the processes are still two, right? And so when you're running in a large cluster, you'll see that the suggestion is to run a separate controller quorum. Very similar to how you would run Zookeeper and then then regular Kafka.

And so I think it's important to understand how systems evolve. I'm sure if they had a clean slate, which I had when I first fired up Emacs in n 2019, I get to learn from a lot of the mistakes that people have made and a lot of the lessons learned and things that they wish to be different because we didn't have production systems to support in 2019. It was very easy for us to start with a very radical and different architecture. Kafka, as an open source project, has a much more difficult problem that we'll ever have, which is they have millions and millions of lines of code that are dependent on the implicit behavior and interactions of the systems.

So it's actually not as trivial as saying, “Oh, they're removing Zookeeper.” The actual number of fault domains, which is they think that matters the most from a distributed system perspective, has not changed. And then the last one that I'll add is that now, with Kafka, you have two different replication protocols. You have the in-sync replica set protocol, which is called ISR in short, and then you have Raft.

When we wrote Redpanda, what we focused on was not to invent our own Facebook protocol like ISR for Kafka. Instead what we said was, “Hey, let's use a protocol that has a mathematical proof Raft, and instead let's focus our engineering effort in making Raft superfast for both controller and data replication. And so what this gave us as system designers is actually a much smaller system in terms of code footprint and it's simpler architecture, because we only have one replication protocol, and it's a strongly consistent replication protocol.

And so I will actually invite users to check out the code and these claims. We're going to probably open source next Tuesday. We were going to open source this Tuesday, but then elections happened. And so we just kind of like didn't – There's like real important things in the world. So we're like, “Okay, we'll wait a week,” and hopefully people will be in a good mental state next Tuesday. So these claims are relatively easy to find out for people. But I think that is sort of a misconception with KIP-500. People think KIP-500 is going to solve all of their production systems and fault domains and it will solve some problems, for sure, but it's not going to boil the ocean. There's like a very long transition period between Zookeeper and the new controller with Raft application. And then there's also the fact that they didn't remove default domains. They just replaced the controller with the Raft quorum system.

[00:34:11] JM: How big is the Redpanda ecosystem?

[00:34:16] AG: It depends on how you look at it. So we're still a really young company. And what we did, we have – This company is extremely technical. I am the CEO and I wrote the storage engine with kernel bypass. Every single person in the company including the sales person have built actually code and systems. And so part of the thing that we wanted to release, we only made it publicly available about six weeks ago or so, was the first release, we wanted to ensure that we had no data loss. That was the thing that I focused on the most. We were like the architectural is going to allow us to get performance improvement super easily just because we wrote it and we sort of eliminated categorical problems with our write record architecture.

And so it's relatively small today because we've only been out for like six weeks. But the fundamental thing that we wanted to give users, it sucks when you have to run a system someone tells you and then you lose data. That's really like – There's an expectation of quality in new infrastructure projects that you sort of just expect them to be rock solid. It's amazing how open source has changed in the last 10 years really. So the first release we ran actually an internal extended version of Jepsen, which is an empirical test suite to ensure that you in fact can do the things that your product claims to do with regards to consistency and no data loss.

And so we brought in Dennis who has worked on a lot of extensions on Jepsen. And the reason for us extending this was that Jepsen performance, when you evaluate a Jepsen [inaudible 00:36:07] is actually N-factorial. So computationally speaking, it takes a very long time to evaluate relatively short histories. We're talking about you run the system for two minutes or something like that and then you tear it down and then you run a history evaluation on the logs. And what Dennis has done is he like added additional constraints to Jepsen to allow to reduce the computational complexity from N-factorial to linear and in terms of evaluating the history of this. This was important, because it gives us confidence when we go into enterprise conversations, I can go to the CTO and look at him straight in the eye and say like, “We do not lose data. Here's the proof.”

Not only do we base ourselves on a protocol that has a mathematical proof around zero data loss, which is Raft, but the empirical implementation and testing shows that we don't lose data. And so for that, we actually waited – Man! 20 months or 18 months or something like that to just release the first version. But if you look at as a player in sort of the larger streaming ecosystem, then we plug into every single streaming platform that exists because of the Kafka API compatibility. And again, the analogy here was that if we were a database, we would speak SQL. But because we're a streaming engine, we speak the Kafka API. Because we speak the Kafka API, we can get to leverage the entire ecosystem.

And so, transitively, our ecosystem is huge. It's like if you've written a Kafka application, whether it's in Go using the Sarama or the Conflict client or in C++ using librdkafka or in Java using the upstream drivers, or in Python using the Python clients. It doesn't matter. Because we speak the exact same language ,then we get to leverage this huge and evolving ecosystem. There was no way that we would be able to rewrite client APIs and client protocols and test them and a bunch of streaming things. So instead the only thing that we thought was manageable was let's speak this lingua franca through the Kafka API. And therefore we are compatible with the entire Kafka ecosystem a 100%.

[00:38:31] JM: Could you go a little bit deeper on one of the case studies that you've discussed?

[00:38:35] AG: Yeah. So happy to. Maybe we have two extremely interesting use cases. The first one I would say is the intrusion detection system. So we are working with a really large security company, and what they do is that they monitor the network traffic. And so once they monitor, the way they charge their customers is after they detect anomalies and so on, they write them to a durable storage. And they wrote their own a storage subsystem and so on.

And so when we came in, we actually replaced the storage subsystem for them and we just shipped the binary along as part of their installation with their application. And so what's really interesting there is that now we're running at the true edge. Like, literally, things that are just almost the first point of contact with the world. A separate use case was this web lab, this medical center that we're talking to, is they wanted to do DNA sequencing and a bunch of like web lab analysis. So web lab is you basically take your spit and other bodily fluids and they run some systems and the data of the systems goes into Kafka right now.

And so because we can go at the speed of hardware, they are thinking about using us to do like live web lab analysis where before you leave the hospital they'll tell you, it's like, “Oh, can you please go in again and spit again on this too so that we can rerun the analysis because we found a problem or whatever.” And the hope there is that that will basically detect more cancer patients is sort of the gist of that.

And then the last one from a technical perspective is we talked to a database company and they have this massive, massive data pipes. We're talking about like four gigabytes per second and replicated. So let's say like the tallest AWS instances, like 100 gigabits on the I3 metal instances, right? So huge, huge loads and huge traffic.

And the way they thought about the Kafka API in particular with Redpanda with tiered hierarchical storage, is that Kafka gives user total log addressability. What does that mean? Every time you save a record to Kafka, you get a uniquely identifiable ID. That's called your offset. And that is guaranteed never to have the same ID forever, because you just add one to the record batch. And so every time you save an item, then you get one. The next one will be two, three, four and so on.

But from a database perspective, if you treat this write ahead log as actually a storage engine where you can save, let's say, a megabyte of data for every record batch, then all of the sudden you have this mechanism to fetch and retrieve hierarchical data of a write ahead log that could span petabytes of data. And so how does this work on a technical level? What happens is people put Kafka on physical computers, because that's how software works. These physical computers have some disk limitations, let's say, 10 terabytes of disk. Or, let's say, for simplicity of discussion, a terabyte of disk. If you're pushing multi gigabyte per second workloads, this terabyte of disk is going to fail in a few hours, right? And so what happens is that with Kafka there is this assumption that people will consume and de-queue and basically remove the back pressure from the writer by consuming from the log and moving that data out onto a separate system, let’s say Elasticsearch or something like that.

So what we've done, and I think Kafka Upstream is also working on something similar, not quite the same, but similar. When instead of deleting data from local disks so that you can keep accepting writes, we can push them to a data lake. Let's say most commonly Amazon S3. And so what that does for users is that it effectively unifies historical and real-time access with the same API. Our addition on top of what some of the classical tiered storage systems have done, which is they just take all data, they put it to S3 and then you never look at it again unless there is a disaster. Is that we actually understand the structure because we publish alongside the data. We publish index and information about the data that we publish to S3. And so it allows us to dynamically fetch for users historical data.

And so what this gives us is that we can decouple the clusters that are doing most of the writes from clusters that are doing analytical read-only workloads. And so it's not like really kind of maybe the most famous example of someone that decoupled compute from a store. And really the architecture, if you read the Snowflake paper, it says that the cold data lives on S3 and then there's some like frontend compute similar to Presto.

And so we kind of took that idea and it's like, “Oh, what if we can just give people the ability to spin up separate clusters on Amazon but then they can use the exact same API?” So the application code that the machine learning people are doing for doing real-time streaming doesn't have to change even if they have to re-consume petabytes of data on a separate cluster. And so anyway, so just to wrap up the argument, was that this database company is really using us for this tiered storage and this ability to read cold index data in a way that unifies their API. So they treat us literally as an infinite write ahead log with geo distribution.

[00:44:31] JM: Tell me a little bit more about what you see as the near near-term future for Redpanda.

[00:44:38] AG: So our biggest challenges and the things that we wanted to work on is I wanted to release this as open source. And I was here, because I talked to a cloud vendor during the early stages of the company and they told me in person, which I still find a bit tricky for the record, if we open source this permissively licensed, they would just simply take it because they want a better implementation. And so it took a really long time to understand how do we become a company that allows people to run our infrastructure software but we can still survive as an infrastructure company, right? Like it's sort of this balance between community building and making money.

And so after being an expert in this, and I'll note this on the blog post that we're releasing soon. There's really no company that I know that is going to release a very finished product kind of like what we're doing that is compatible with this rich ecosystem that is also going to be open source. So we're going to choose a BSL license, which is similar to what CockroachDB does. And what that says is really three key things and I think three challenges for Redpanda. The first one, we're the only company that can run Redpanda as a service. If you run it as a service, you can either run upstream Kafka or you can pay us and we can work with you. So that's really the first one. So that sort of eliminates the hyper cloud problem. But it gives users that aren't trying to compete with us on a hosted service, like they can just run. They can run the base.

And so with these two primitives, then we can open source the code. By the way, even the enterprise features, we're going to make in source available. We're not trying to hide what we do. I think we are internally already a really transparent company and we want to bring that to the community as well. And so the next challenge for us is really trying to nurture a community that can coexist within some of the larger streaming community, in particular with projects that are already using the Kafka API, right? And so it's trying to find business, which we think is going to be the JavaScript and python developers and people that are really looking for just super simple systems to start up, but can allow them to scale to petabyte scale.

And so I think that's going to be the focus of the company and the product where we get to coexist in a larger ecosystem. Of course we will compete with some users. There's like no doubt about that. But I also think that streaming is so young and so [inaudible 00:47:15] that there's going to be sort of space for multiple projects. And I actually expect a lot more systems to come out with Kafka API compatibility support that may offer a totally different thing. This is just normal in software evolution. If you think about what happened to the data lake API, right? There's a project called MinIO, which does S3 API. There's a Python library like inside your application if you need to migrate something. You can just stand up a Python, a little Python server that is fixed to S3 API and so on. Once an API becomes popular, I think people will start looking for different guarantees about the particular engines. This happened to SQL and all of these rise of from SQL to NoSQL to NewSQL and so on. And so I think the next steps for us as a community is to invite users uh to participate and sort of create with and they could they get to use the product for free, the majority I think. But it still leaves room for us as a company to survive.

And I think that's going to be challenging in part because we are creating new territories for a large legion of programmers that aren't doing streaming because the existing systems are too difficult to operate. And so I think we'll need to understand that it's going to look different. So I'm not sure if the final API for this new community is actually going to be the Kafka API or not. And so I think it's going to be an interesting challenge. It might be the WASM engine where we create a library of components, of open source components, where people can just connect and they can get GDPR compliance with a single click or they connect to Elasticsearch with a single click or they do all of this with just a single command. It might be that that's how the community develops.

So I think for a Redpanda, the project is going to be about how do we integrate into the larger streaming community, which is just growing like so fast as the sort of consumers demand real-time components from products. The fact is this, infrastructure is – Engineers are a utility function to society, which means the infrastructure that we built is only required because users, like you and I, vote with their dollars to demand real-time things. We demand that from our phones we can turn on the oven 30 minutes before we get out of bed if you want to make bread in the morning, right? That kind of real-time interactivity is pressure that we put on companies. Transitively, that pressure gets put onto systems like Redpanda. So we'll see how this community evolves. I think it's too early and I'm just excited.

[00:50:07] JM: Great. Well, thanks for coming the show. It's been great talking to you about Redpanda.

[00:50:13] AG: Thanks, Jeff. Talk to you soon.

[END]