EPISODE 665

[INTRODUCTION]

[0:00:00.3] JM: Robinhood is a platform for buying and selling stocks, cryptocurrencies and other assets. Since its founding in 2013, Robinhood has grown to have more than 3 million user accounts, which is approximately the same as the popular online broker Etrade, but the surge in user growth and transaction volume, the demands on the software infrastructure of Robinhood have increased significantly. When a user buys a stock on Robinhood, that transaction gets written to Kafka and PostgreS. Multiple services get notified of the new entry on the Kafka topic for buying stocks and those services process that new event using Kafka Streams.

Kafka Streams are a way of reading streams of data out of Kafka with exactly-once semantics. Developers at Robinhood use a variety of languages to build services on top of these Kafka Streams, and one of those languages is Python. Some commonly used systems for building stream processing tasks on top of a Kafka topic include Apache Flink and Apache Spark.

Spark and Flink let you work with large datasets while maintaining high-speed and fault tolerance. These tools are written in Java. If you want to write a Python program that interfaces with Apache Spark, you have to pay expensive serialization and deserialization costs as you move that object between Python and Spark.

By the way, I know that Apache Arrow can help with this problem, and we have done a show about that. I'm not sure of the state of Apache Arrow, but it is something that we did not really addressed in this show.

Ask Solem is an engineer with Robinhood and he's the author of Faust, which is a stream processing library that ports the ideas of Kafka Streams to Python. Faust provides stream processing and event processing in a manner that is similar to Kafka Streams, Apache Spark and Apache Flink, although I think the Kafka Streams notion is the most close streaming analogy to Faust.

I also believe there's some similarity between Faust and Kafka as existential authors, but I did not ask Ask about that in today's show. Ask is also the author of the popular Celery, asynchronous task queue. Ask joins the show to provide his perspective on large scale distributed stream processing. He talks about why he created Faust. This is a complicated streaming project and we get in to some complicated distributed system semantics, but I really enjoyed talking to him and I'm hoping to do more shows around this topic of stream processing in the near future.

I also want to quickly mention that we're hiring for Software Engineering Daily and you can find those job applications at softwareengineeringdaily.com/jobs. We've got a variety of job openings including recently a business development position.

[SPONSOR MESSAGE]

[00:03:11] JM: Citus Data can scale your PostgreS database horizontally. For many of you, your PostgreS database is the heart of your application. You chose PostgreS because you trust it. After all, PostgreS is battle tested, trustworthy database software, but are you spending more and more time dealing with scalability issues? Citus distributes your data and your queries across multiple nodes. Are your queries getting slow? Citus can parallelize your SQL queries across multiple nodes dramatically speeding them up and giving you much lower latency.

Are you worried about hitting the limits of single node PostgreS and not being able to grow your app or having to spend your time on database infrastructure instead of creating new features for you application? Available as open source as a database as a service and as enterprise software, Citus makes it simple to shard PostgreS. Go to citusdata.com/sedaily to learn more about how Citus transforms PostgreS into a distributed database. That's citusdata.com/sedaily, citusdata.com/sedaily.

Get back the time that you're spending on database operations. Companies like Algolia, Prosperworks and Cisco are all using Citus so they no longer have to worry about scaling their database. Try it yourself at citusdata.com/sedaily. That's citusdata.com/sedaily. Thank you to Citus Data for being a sponsor of Software Engineering Daily.

[INTERVIEW]

[00:04:55] JM: Ask Solem, you are a principal software engineer at Robinhood. Welcome to Software Engineering Daily.

[00:05:00] AS: Thank you. It's very nice to meet you.

[00:05:02] JM: Yes. I'm looking forward to talking about stream processing, batch processing, data infrastructure and of course a very interesting high throughput application of those things. I'd like to start with a discussion of stream processing and batch processing. When some people hear the term stream, they imagine fast-moving sequences of data, but in some context, a data stream is just a sequence of data points that's sitting somewhere. Maybe it's on disk, maybe it's in memory, maybe it's in a file. Could you help us understand what is a stream and what is a batch?

[00:05:38] AS: So a batch is basically a buffer of data that needs to be processed. A stream can also be a batch, but stream processing is usually used with data in real-time. So you process one event at a time, and as you process it, the result is immediately available.

A stream can be anything. It could be the entire set of data that flows through Facebook, or it could be all account changes on account on Facebook for example.

[00:06:07] JM: Does it matter what kind of storage medium that is in that we're talking about? If something is a database table that's just sitting there, would we consider that a batch or could we consider that also a stream?

[00:06:23] AS: You could consider that stream if you constantly process it. I guess you could process something offline, which means you have it in a table, you want to process it in order. But I don't think that's where stream processing is useful. Stream processing is useful when you have much larger real-time streams that you want to store some state about and make that to make available for further processing.

[00:06:47] JM: Now, does a stream have a set of data points that are constantly getting rotated out? Do they have some time period in which they exist in that stream?

[00:06:58] AS: Well, yeah. That's like the nature of a stream. There is a time element in it. But yeah, I mean, that's one of the properties that we use to process data in it. But that is not necessarily true, like all events in a stream could happen at the same time, I guess. So a stream is just one event happening at the time in sequence. It could be absolutely anything. It could be that the sounds from an audio file. It could be the frames from a video file or a database table, or it's the sensor data for a weather sensor.

[00:07:30] JM: Right. So in the batch example, you might say – Let's just make this a little more tangible for people. So Robinhood is an app that allows people to invest without paying large sums for commission. So you can buy and sell stocks or cryptocurrencies, and I can imagine an application of stream processing or batch processing would be to process a set of transactions. So buy order or sell orders. In the stream case, you would want to be processing each of them on the fly as they enter a system. In the batch model, you maybe want to process a historical set of a bunch of transactions overtime. How do the requirements for the processing systems for those two use cases; batch versus streaming? What are the software engineering requirements for each of those use cases?

[00:08:22] AS: Well, at some point batch processing usually doesn't scale. Every night we have hundreds of batch jobs that are performed that do things like risk analysis, or it just verifies the accounting for a ledger or ledgers. Eventually, when you have enough users, it will take so much time to complete that it doesn't complete for the next day starts and job start overlap.

When you use batch processing, it also means that if we have million accounts to process, a user has to wait for his account to be processed before his result is available. So, yeah, the state of the system takes time to update. Whereas in a stream processing real-time system, the things are available in real-time.

[00:09:10] JM: When I think about batch processing today, I think about a MapReduce job, and MapReduce jobs are parallelizable. If you wanted to process larger and larger sets of batch

transactions, why couldn't you just increase the size of your Hadoop cluster and have that scale up magically?

[00:09:29] AS: I guess that's possible, but Hadoop is not really suitable for real-time streaming jobs. They have some extensions for it, but right now we're talking about implementing backend features, not just data science.

[00:09:43] JM: I meant in the sense that you have these nightly batch jobs. I'm just curious where the batch jobs start to fall over.

[00:09:49] AS: Oh, yeah. I mean, we could probably fix them, but what we want is for these processes to be real-time so that we can react quicker to changes in our system.

[00:09:59] JM: Right. Definitely.

[00:10:00] AS: There's just so many of them. So we want to see our Robinhood as a stream of data that we can just connect to. Say I have a service that wants to know when a new account is created. I just consume from the account is created topic, I react accordingly whenever an account is created.

[00:10:17] JM: Okay. Can you describe that in a little more detail? When you're talking about consuming a topic, what piece of software infrastructure consuming that topic off of and what is it being consumed into and then where is the result going?

[00:10:31] AS: Yeah. That's what Faust essentially provides you with, is stream processing. A stream processor reads from Kafka topic. So we publish into the topic, for example, whenever an account is created and whoever is interested in that topic and will consume from that Kafka topic. Some of these topics have elements that expire, but some of them have the history since the beginning of time.

[00:10:56] JM: We've done shows about different applications of Kafka, whether people are writing logging events, or whether data, or any kind of high throughput or low throughput data stream. It's a big distributed pub/sub mechanism where you can write data to a channel and

other people can subscribe to the channel and pull data off of that channel. Then you have stream processing systems that often are the consumers from those Kafka topics, and we've done shows about Spark, and Storm, and then Kafka Stream, which is like a system, a streaming system that's closely coupled with Kafka. What were the shortcomings or the deficiencies in those systems that made you say, "I want to build my own. I want to build Faust, a stream processing system that will allow me to consume information off of Kafka in a way that works for me."

[00:11:52] AS: There were series of events happening that led up to that, but one was the Python 3 asyncio, which is very fast. We use Python extentively at Robinhood. It's very easy to rapidly prototype and actually deploy services in production and our devs really like using it. It's easy to learn. It's easy to use. We wanted something that we can use directly from Python. But moreover, we didn't want to have complicated integration setups like at Spark needs to send something to my Python process. Is it via HTTP? Then we saw Kafka Streams, which basically is a library that you just import into your Java code. Yeah, I think that idea and we think that idea is absolutely phenomenal.

[00:12:41] JM: How does Kafka Streams work? Can you talk more about that?

[00:12:44] AS: So Kafka Streams – Well, actually we had to read docs and read the code for months before we understood all the aspects of it, because it's rather complicated. There's many moving parts when you get into the details of it. Mostly related to partitioning and the tables that they use to store state. But Kafka streams work only by leveraging Kafka. That's the only thing it uses. It can use kafka as an actual database. So, yeah.

[00:13:13] JM: Okay. So with Spark streaming, I think there's some notion of maybe you have a Kafka topic and you might pull it into a Spark RDD so that you have it all in-memory and it's got faster access time and you can perform complex series of operations on those Spark RDDs. You can do recreations and whatnot. Getting it into Spark can allow you to just do iterative data science.

On top of that, how does the experience of pulling off of Kafka into Spark contrast with pulling it off of Kafka topic into a Kafka stream?

[00:13:51] AS: They are possibly similar. I mean, I'm sure Spark has many uses that Faust is not suitable for possibly. But what we really want to do with Faust is write backend services. You wouldn't write a backend service in Spark, and that's kind of what we want to do.

So at Robinhood we have an iOS client and an android client and we have a web app that all use the same backend services. So things like Celery and task queues extensively, and Faust can expand upon that by applying the actual methods of computation to Kafka while also letting us being able to use it as a database.

[00:14:33] JM: Fascinating. When you talk about wanting to be able to write backend applications on top of a stream processing system as supposed to doing data science jobs, or just doing – Because I think we might think of processing these streams as they're coming in and maybe we're just writing to a table. How does that contrast with building a backend application? Help me understand that requirement.

[00:14:58] AS: Yeah. So traditionally, backend applications are written as a web server. A web server with our request and respond cycle. Within that request response cycle, it probably communicates with a database to store state. So when a user creates an account or it logs in, we store state about it in a database. But this system can also be processed as a stream. We can use Kafka as the database by partitioning data.

Whenever something related to one account comes in, it always goes to the same machine. That way our database is distributed. If you want to processing something later, like we don't want the user to wait and maybe we need to make sure that it happens. If you have the process in order, we need to make sure that happens. Then we put it into a Kafka topic or a Celery queue, like RabbitMQ or something like that. This system kind of provides us with all of these in a single thing that we can use.

[00:15:56] JM: Okay. Cool. One example might be you want to have a listener for every new account signup that comes in, you gave that as an example. Is there another application example we could think of to help us solidify why the streaming use case for Kafka streams plus Faust, why you build that?

[00:16:17] AS: Yeah. There's also the concept of micro-services. In a company, there's many teams that needs to collaborate. Now we create many small micro-services that communicate. Every micro-service has its own infrastructure requirements. Maybe it's on Redis, maybe it's on database. By using a stream topology instead, it makes it easier to collaborate across teams. That's my guess. I don't know yet if that is the case, but it –

[00:16:47] JM: So you're writing micro-services in Python?

[00:16:51] AS: Yeah, in Python and in Go and in other languages. So the idea is that we put things into Kafka and we don't handle web requests individually. We just put in Kafka and the different file services will read from Kafka and do something with it, and that's how we could find the interfaces.

[00:17:11] JM: So you have – Kafka is almost like your API middleware. Every single API request is making its way through Kafka?

[00:17:20] AS: Well, not yet, but that's the idea. That's also one of the ideas behind Kafka Streams as well. They write about it quite a bit in the documentation, but I guess not so much as to not confuse people, because the idea is rather new.

[SPONSOR MESSAGE]

[00:17:46] JM: Cloud computing can get expensive. If you're spending too much money on your cloud infrastructure, check out DoIt International. DoIt International helps startups optimize the cost of their workloads across Google Cloud and AWS so that the startups can spend more time building their new software and less time reducing their cost.

DoIt international helps clients optimize their costs, and if your cloud bill is over $10,000 per month, you can get a free cost optimization assessment by going to D-O-I-T-I-N-T-L.com/sedaily. That's a D-O-I-T-I-N-T-L.com/sedaily. This assessment will show you how you can save money on your cloud, and DoIt International is offering it to our listeners for free. They normally charge $5,000 for this assessment, but DoIt International is offering it free to listeners of the

show with more than $10,000 in monthly spend. If you don't know whether or not you're spending $10,000, if your company is that big, there's a good chance you're spending $10,000. So maybe go ask somebody else in the finance department.

DoIt International is a company that's made up of experts in cloud engineering and optimization. They can help you run your infrastructure more efficiently by helping you use commitments, spot instances, rightsizing and unique purchasing techniques. This to me sounds extremely domain specific. So it makes sense to me from that perspective to hire a team of people who can help you figure out how to implement these techniques.

DoIt International can help you write more efficient code. They can help you build more efficient infrastructure. They also have their own custom software that they've written, which is a complete cost optimization platform for Google cloud, and that's available at reoptimize.io as a free service if you want check out what DoIT International is capable of building.

DoIt International art experts in cloud cost optimization, and if you're spending more than $10,000, you can get a free assessment by going to D-O-I-T-I-N-T-L.com/sedaily and see how much money you can save on your cloud deployment.

[INTERVIEW CONTINUED]

[00:20:11] JM: When people at writing Robinhood, people are writing micro-services in Go, or Node.js, or Python and they're reading API requests off of Kafka. By the way, if people want to know why you would want to write all your API requests to Kafka, we've done a bunch of shows about –

[00:20:29] AS: No. You don't actually want to write all your API requests to Kafka, but anything that is happening in your system. That's one we'd want to care about. When you create an order or when somebody withdraws money from the bank account or any event that is important in the system, that something needs to happen for.

[00:20:49] JM: The event sourcing, the ability to look back through your history of events and recreate them and replay them, that can't be really useful especially for a financial base system where you want to have an audit trail.

[00:21:03] AS: Yeah, that's even a requirement for our industry.

[00:21:06] JM: Yeah. So when you want allow developers to build micro-services against those Kafka topics, so there's a Kafka Stream that get exposed, and I assume – Why can't they just consume that Kafka stream in their Node.js service or their Go service? What's the role of Faust between the Kafka stream and the micro-service they're writing?

[00:21:29] AS: Oh, no. They can absolutely use Go, or Lang, or Lisp or whatever they please. Faust is just what we use in Python, but it stores data in Kafka using the tables, they would have to have library support in whatever language they use. But if they just want to consume event streams, then they can from any language.

[00:21:50] JM: I see. Okay. So Faust is for consuming Kafka streams to write services that are adapted to Kafka streams. I guess I'm still having a little bit of travel understanding it, because if you could just write a Python micro-service to consume a Kafka stream, what does Faust give you on top of that?

[00:22:08] AS: On top of that, it also gives you the tables. So a table in Faust is something you define. It ends up being just a Python dictionary that you can use. So it's a key value store. So you could create a table of accounts. They key is the ID to the account object. This is an actual object. So you connect the store objects in there. It stored in-memory or backed by RocksDB locally. But we have failover whenever a key in that dictionary changes, we send that change as a message to a Kafka change log topic. So if the node dies, we can reread the change log from Kafka to rebuild the state at it was before. We have standby nodes. So some nodes will always consume the change log from other nodes so we can quickly recover in case one of the nodes crash. It's basically a distributed database as well with windowing.

[00:23:03] JM: It's a distributed database with windowing that allows you to do operations with Python. I see. It allows you to get the data off of the Kafka stream into Python so that you can

perform data science type operations in Python and you don't have to do some complex serialization, deserialization into Java and stuff like that, or Pi Spark would have to hit Java in order for you to work in Python.

[00:23:32] AS: Yeah, exactly. Yeah.

[00:23:34] JM: So I've heard of Apache Arrow, which allows some data interchange between Spark and Python could you have used Apache Arrow?

[00:23:41] AS: I have never used it, but there's many tools. I'm sure there's many tools like Apache Spark. But I still think that this gives something unique to this space.

[00:23:52] JM: Yeah. Well, I'm curious about the design choices. So RocksDB for example and the failover cases, tell me about what RocksDB does and why that was your choice of database?

[00:24:06] AS: RocksDB is just a local cache of the data for any particular node. So every node has an embedded RocksDB database. But this is configurable, so you can also store the tables in-memory because they're backed by the change log topics. So the source of truth is in Kafka. But if you restart th service and it's in-memory, it's gone, and then you have to read everything from Kafka again, which will be very slow if there's millions of records or something like that. RocksDB just acts like a fast cache, a fast local cache. RocksDB is an embedded database, so it doesn't have a server.

[00:24:39] JM: Let's say I have a service that wants to constantly watch for the topic of new transactions that are coming in and you want to have a running tally of the transactions over a given time window, like that's what your service in Python is going to do. It's going to watch a Kafka topic for all new buy orders that come in and it's going to have a running tally over them over, let's say, a five hour window.

[00:25:10] AS: Yeah. That is not over five – Windowing doesn't mean that you just have the latest window, right? If your window size is five minutes, it means you can go back a week ago

as well to see what the counts was a week ago. Depending on the retention settings of that table.

[00:25:26] JM: That would be the retention settings of a table in Faust or in Kafka?

[00:25:31] AS: In Kafka. Well, you configure it in Faust, but eventually it's the retention settings of the change log topic in Kafka.

[00:25:39] JM: Okay. So in your Kafka cluster, do you keep everything there? Can I go back in history to the beginning of time and find all the buy orders?

[00:25:48] AS: Yes. I mean, not the beginning of time, but I guess – I don't actually know right now if it keep – I don't think the idea is that we can rewind from the beginning of time. I don't think we want to depend on that. We back it up and it's not unbounded. We don't have unbounded storage.

[00:26:07] JM: Right. So just going back to Kafka for a second, because I want to focus a little bit more on Kafka to get the full picture here. What kind of retention policy do you have for your data and what kinds of applications do you expect to be built off of that data?

[00:26:21] AS: That's a bit of broad question. I mean, we want to start absolutely everything in Kafka, I think. Anything that is of interest, we will store in Kafka.

[00:26:31] JM: Okay. Do you have a general advice for retention policy? The account creation example, how long would you want to keep all that data in Kafka?

[00:26:41] AS: Oh! So one thing is the account creation, which is a real-time stream of account changes or something, I guess. The other one if you store that as a table in Faust, you replace a database. You no longer have a PostgreS, instead you store the account information in Faust. Then the tables change log topic is backed by a topic which deletes old keys. So whenever you set a new key, it can purge he old value from it. It has a name. I don't remember the name. But, yeah. Only the latest value for a key is stored.

[00:27:14] JM: So you mentioned PostgreS there. What PostgreS database are you talking about? Is that the accounts database?

[00:27:19] AS: Yeah. Right now we store our accounts in PostgreS.

[00:27:22] JM: Okay. So you're saying you could use Faust as a database instead of PostgreS?

[00:27:28] AS: Exactly. Yes. As a distributed database. That is a distributed database that is available.

[00:27:34] JM: Right. But would it be less durable?

[00:27:38] AS: Sorry. Less?

[00:27:39] JM: Or less – Let's see, what's the term I'm looking for?

[00:27:42] AS: Durable.

[00:27:43] JM: yeah. I was going to say Durable. What would be the pros and cons if you replaced your PostgreS? So just to be clear, you're saying that you could replace the PostgreS database that you're using as a source of truth for your accounts with basically a Faust table, distributed table.

[00:27:59] AS: Yeah. The main disadvantage would be that centralization is very easy. In this case, we have to partition the data, so it's sharded by account ID or something. So every change to an account always goes to the same node. So we can't easily join information together just like the limitation of a distributed database is like – That's also difficult in PostgreS if you want to have a cluster. But we can still store the individual accounts. So in some cases, it's less flexible, but it also scales and it's easy to deploy and maintain, I hope. Well, not yet.

[00:28:36] JM: What are the scalability advantages? Why does it scale more easily than PostgreS?

[00:28:41] AS: PostgreS is centralized. You can shard it, but once you start sharding your database, you can't join anymore. So I would say PostgreS is probably absolutely the best choice for anyone storing data right now. But the idea is that we can store state in Kafka itself and possibly replace PostgreS, and we do in some case, like the Robinhood feed, the real-time chat that we have added is using Faust as the backend. The same service that reads the stream also serves the web sockets and we store the state in tables, like banned users store it in tables. That means that if you want to look up, "Is this user banned?" We know it because it's in-memory.

[00:29:23] JM: Oh, okay. So you do it really quickly I guess? It will be faster than PostgreS in that example.

[00:29:29] AS: Yeah, definitely faster in PostgreS.

[00:29:30] JM: Oh, because it's in-memory. Of course.

[00:29:31] AS: It's in-memory. Yeah.

[00:29:33] JM: Wow! Yeah.

[00:29:33] AS: But the data locality is also an issue. Some things becomes very difficult, but that's related to the partitioning, right? If you want to suddenly get information about another user, then you would need to go to that box to look it up. So some things are very easy. Other things are more difficult. But that's the same with any distributed system. Like with any distributed database, there are tradeoffs.

[00:29:56] JM: Let's contrast this with Redis, if that's an appropriate comparison. So Redis is, as I understand, an in-memory object storage system. People often use it for fast access to objects that they want, and then they will maybe have some retention policy there or keep everything for a certain application in-memory with Redis. How does Redis contrast with what you're building with Faust?

[00:30:20] AS: Well, Redis, it's not really in-memory. I mean, it's a server that you call. So it's in another machine, right? So it's not exactly the same. But we also use Redis from it in Faust. For some things, we actually prefer using Redis for – Like, I don't think this system is ready to store absolutely all data as of yet. Exactly what we can store and what we can use it for is unclear at this point.

[00:30:43] JM: So you're cutting down on network hops in the comparison with Redis?

[00:30:47] AS: Yeah, but also no serialization.

[00:30:50] JM: What are Redis objects? What's the serialization format for this?

[00:30:54] AS: I think they only have strings and counters. If you want to store something more complex, then you would have to serialize it with JSON or some other format. But in this case, you're actually communicating with the objects themselves.

[00:31:08] JM: Communicating with the objects themselves. So RocksDB lets you just keep these things in –

[00:31:13] AS: If you use RocksDB, then you would have to actually de-serialize on access. But the thing is that you can store part of it in-memory and just keep a reference to RocksDB.

[00:31:24] JM: Oh! What would be an application to that? When would I want to have a part of my object in-memory?

[00:31:30] AS: Well, I guess if you really wanted to optimize for node de-serialization.

[00:31:35] JM: So you talk about database centralization. What do you mean by database centralization? How do you define that term?

[00:31:41] AS: I mean a centralized database is usually just one node, right? So PostgreS and MySQL is usually just one single node, a central point of failure. You can add sharding to that, but that is a very tricky thing. Like you could shard a table by account ID. So some accounts are

on one server, another on another server. But then joins get very tricky. You can't join data together.

[00:32:09] JM: In contrast, how does Faust avoid the centralization problem?

[00:32:13] AS: Well, it uses sharding. It uses partitioning, which is the same that you would in PostgreS, but in a slightly different way. You don't need a client and expressing the code for it is very easy. We've wrote some Faust services that we had that were like hundreds of lines of code and it ended up being just like 10 lines of Faust.

[00:32:33] JM: Is it masterless, like for the queries? Do you not have to have some central orchestrator?

[00:32:37] AS: No, you don't need any central orchestrator. But you can't easily create complex queries in the same way as you can with a SQL database, because the state is stored over many machines, right? So if you're a new company starting, like storing all your data, this is probably less convenient. I probably would suggest to use PostgreS for that, because even though it's centralized, it works. Well, you want to store some of these data there, because we have many users and we want to scale and the database is a bottleneck and a single point of failure.

[00:33:14] JM: The database is a bottleneck in terms of latency.

[00:33:18] AS: In terms of latency and we can't grow it forever. It's going to run out of space at some point, and then you just have to shard it.

[00:33:26] JM: What about Cassandra? Would Cassandra be an alternative here?

[00:33:30] AS: Yeah. I mean, you can use Cassandra. Then the nature of how it's used to similar too in Faust, like sharding and portioning. In that case, it's eventually consistent.

[00:33:42] JM: But it's not in Python.

[00:33:43] AS: It's not in Python. In this case, it's just like you have everything in one library, right? So you can express things more easily. I think it's more about like being able to express things more easily and faster, and that's what excites me more about it, I think.

[00:33:57] JM: That's because it's all on the same node. You could just write Python code and you don't have to have any unanticipated latencies from calling some opaque function that's going to do some serialization or deserialization. Okay. Interesting.

I'd love to get a broader perspective for data engineering at Robinhood and where else, because I know Robinhood is surging in usage and I'd love to know where else the scalability bottlenecks have occurred and what kinds of problems you encountered. We did do a show recently about Uber's data infrastructure where they talked about their ETL process and their OLTP databases versus their OLAP store. Can you give me a brief tour of the data infrastructure and maybe the ETL process at Robinhood?

[00:34:50] AS: Yeah. I mean, our systems are – Most of our system are written in Python and some are written in Go. Our data infrastructure is all over the place. We have many different kinds of services. Yeah, I think just like at Uber, they have different teams and different micro-services. Yeah, there's a lot of them.

We use Spark for some things. We use other things from machine learning. Yeah, we have a lot of systems in production. Are there any databases that you've standardized on? For example, for transactional processing? Are there regular jobs that pull those transactions into analytic processing databases?

[00:35:30] JM: Yeah. For example, for orders and stuff like that, we always write it to PostgreS first, because it's atomic, and then we know that we have it. Then we usually send it to some queue for further processing. Yeah, and that's where the stream processing step come in, I guess. But for database, we always use PostgreS and we use Redis for some things. We use Memchached. This is quite a fairly normal stack I would say. The data science part, I'm less familiar with actually. It was a more of a backend engineering.

[SPONSOR MESSAGE]

[00:36:09] JM: Accenture is hiring software engineers and architects skilled in modern cloud native tech. If you're looking for a job, check out open opportunities at accenture.com/cloud-native-careers. That's accenture.com/cloud-native-careers.

Working with over 90% of the Fortune 100 companies, Accenture is creating innovative, cutting-edge applications for the cloud, and they are the number one integrator for Amazon Web Services, Microsoft Azure, Google Cloud Platform and more. Accenture innovators come from diverse backgrounds and cultures and they work together to solve client's most challenging problems.

Accenture is committed to developing talent, empowering you with leading edge technology and providing exceptional support to shape your future and work with a global collective that's shaping the future of technology.

Accenture's technology academy, established with MIT, is just one example of how they will equip you with the latest tech skills. That's why they've been recognized on Fortune 100's best companies to work for list for 10 consecutive years.

Grow your career while continuously learning, creating and applying new cloud solutions now. Apply for a job at Accenture today by going to accenture.com/cloud-native-careers. That's accenture.com/cloud-native-careers.

[INTERVIEW CONTINUED]

[00:37:47] JM: I agree. It sounds like it's typical kind of pieces of infrastructure for data engineering. There are some people who are listening who are less familiar with even the simpler data engineering process. Could you walk me through, if you know them, the steps that a buy order, for example, might go through? Does it hit Kafka before PostgreS? Give me a more color on that.

[00:38:09] AS: A buy order comes through – So you have our mobile app, right? Like the iOS app. Maybe you see the stock, you want to buy it. So you hit the buy button. That will call our

backend guys, which is a web server. It goes through nginx, I think. Then the backend written in Python will immediately store it in the PostgreS database so that we don't lose it. Then it's sent to our order execution service written in Go.

From thereon, many different things happen. It's sent to Kafka and it's sent to the execution menus and stuff like that. Yeah, there are many steps. But the core step is the backend API just writes it to PostgreS.

[00:38:53] JM: Why do you write to PostgreS before Kafka?

[00:38:56] AS: We could write it to Kafka, but we didn't actually use Kafka at the time this was written. We could definitely write it to Kafka. If you wrote it to Kafka, we would pretty sure that we will process it, right? But at the time, Kafka was not used at all. So that part happens later in the process. I think it should happen first.

[00:39:15] JM: Why do you think that?

[00:39:16] AS: Yeah. Just because like it serves the same purpose of having a record of it.

[00:39:21] JM: Right. Yeah. I mean, I figure if you want Kafka to be your audit trail, then Kafka should be the first thing to hit.

[00:39:28] AS: Yeah, right. If it's audit trail, then we need to know that it can persist in our data in a durable way.

[00:39:35] JM: Yeah, definitely. Did you evaluate Kinesis or Google Pub/Sub before going with Kafka?

[00:39:43] AS: I looked into it. We looked into it. What we had was Kafka Streams. We had the Kafka stream source code. We were seeing how it worked. The easiest way to translate that was just by using Kafka. But we want to add support for Kinesis and maybe Redis streams down the line.

[00:40:01] JM: Before you even onboarded with Kafka, because you said you had PostgreS for a while before Kafka. So I'm just wondering when you were evaluating Kafka, were you looking at the other solutions?

[00:40:08] AS: Yeah, we were definitely looking at other solutions. But we wanted something that was more of a library. We already used Spark. We already used Kafka and all of these things.

[00:40:17] JM: Right. So that Kafka ecosystem and the plugins and everything. That was really important to you.

[00:40:23] AS: No. I don't think that was important, but we didn't want to use another AWS service, I guess.

[00:40:30] JM: Tell me about how you got the idea for Faust, because the architecture, I'm still wrapping my head around it, but how did you get the idea for it?

[00:40:39] AS: We have all these batch processing jobs here, right? People are writing more and more batch jobs. One of the engineers here [inaudible 00:40:47] came and asked me, like, "What can we do to fix this? Because it doesn't scale."

Then I told him that maybe you should do like real-time stream processing or something instead. He had read about Kafka streams. So we started looking together and we proposed that maybe we should write a client in Python. But it turned out that was impossible. Just like Kafka itself, they put all of the implementation burden on the client itself. But still we wanted to use it, and it was a very exciting technology, I think, Kafka Streams. So it excited me.

[00:41:21] JM: What's the difference between your ability to access data in Kafka versus the ability to access data in a Kafka stream?

[00:41:30] AS: It's the same. Kafka stream is just stored in Kafka. So it's a Kafka topic, right? A stream is a Kafka topic. A table is Kafka topic.

[00:41:38] JM: Okay. So is the storage medium any different? Is a Kafka guaranteed to all be in-memory and maybe a normal Kafka topic is not necessarily guaranteed to be in-memory?

[00:41:47] AS: No. It's the same. A stream is just a Kafka topic. So the Kafka stream's library is just a library written on to of Kafka. But what they have added since is transactions. I don't know if you heard about exactly-once processing. I do quotations marks here.

[00:42:05] JM: I've heard that was really hard to build.

[00:42:07] AS: Yeah. But it frankly means that when you write - when you have tables and windowing, you can ensure that you don't have like double counting because the most one's processing. So you can do deduplication in the table.

[00:42:20] JM: So that's one of the core values of using the Kafka stream's API – It's more like a way of reading data off of Kafka and getting guarantees that you can process this data exactly-once. Is that right?

[00:42:34] AS: Yeah. I mean, it's a way to read data and store state about that data while you read it in a way that means you can in a highly available way and a durable way. So you can use it as – Kafka Streams actually calls it inside-out database.

[00:42:48] JM: What does that mean?

[00:42:49] AS: I guess it means that the database is in-memory basically.

[00:42:52] JM: I know you're not a Kafka developer, but you've clearly studied this stuff intently. Why was exactly-once processing so hard to build? Why is that a hard thing to do?

[00:43:03] AS: It's impossible in general. It's absolutely impossible. It's physically impossible to have exactly-once processing. What they actually have is deduplication. So they have implemented deduplication, but they call it exactly-once processing. I'm not sure it's right to call it. But it essentially means that you have exactly-once processing in relation to tables.

[00:43:23] JM: Do you know why that's physically – Or can you explain why it's physically impossible? I'm sorry. I just don't understand.

[00:43:29] AS: I've read about it. Okay. So exactly-once processing. So you never want to process an event twice, right? It's a distributed system. So was there something about the general or something? Okay, yeah.

[00:43:49] JM: Something about the Byzantine Generals.

[00:43:51] AS: Yeah, something like that. But like say you are in war and you have to tell your general when to attack. You write him a letter and he's going to write a letter back when he's received it so that you know that he's aware of your plan. But what if you never receive the reply? Does the general know your plan or does he not? You can't know that. You need receipt to get it back. But if you don't get a receipt, you don't know if they received it or not, and I think that's the problem that makes it impossible.

[00:44:25] JM: Why does a deduplication operation heal that problem a little bit?

[00:44:31] AS: It's because you can send the ID back in reply, or when you send it – Usually, the idea here is that you don't want something to happen twice, right? So if you add an ID to the message, the receiver can check the ID to see if it's already seen it before. These are kind of two different things. It doesn't work so well with generals' analogy.

[00:44:53] JM: Okay. So Kafka Streams gives you exactly-once guarantees.

[00:44:58] AS: Quotation marks, semantics. Exactly-once semantics.

[00:45:00] JM: In quotations. Yeah, semantically. Okay. So you've got exactly-once data. Thanks to the Kafka stream. You want to pull that into a Python service and be able to perform Python operations over that data in pure Python without serialization/de-serialization issues, which would happen if you were using Pi Spark, or Spark in general. That's the motivation for building services around Kafka Streams with some unique piece of technology.

RocksDB – I'm sorry. Can you revisit why RocksDB is important to the architecture? I guess maybe you could give an example of a service that is taking advantage of Faust that needs to replicate data or have a distributed model –

[00:45:50] AS: Yeah. So for example, the feed that I mentioned, which is our real-time chat. The blurb is feed is Robinhood's chat platform where investors can discuss cryptocurrencies and share instant status updates when they buy or sell coins in the Robinhood. Basically, they discuss things and they can just chat, right? Just chat.

Sometimes users are banned because they're not behaving. The way we do that is that we store in Faust table. When a user is banned, we store it in the table. If we didn't do that, whenever we replay messages, we would have to hit Redis server or something for every message and say is this user banned? Is this user banned? Is this user banned? So it gives us immediate access to it. The table is Python dictionary. Whenever a user is banned, we send a message to a Kafka change log topic.

So if that node dies, we can recover our state by reading the change log topic, right? But that change log topic in this case is probably not that long since it's just banned users and they're pretty nice. But if it was millions of records, then it could take hours to recover. So the RocksDB is a cache that basically gives us instant recover when you restart, when you have to deploy.

[00:47:04] JM: Okay. So contrasting that again with PostgreS, if you used PostgreS, what would follow again in PostgreS or what would be the problem? I guess it's just the centralization.

[00:47:14] AS: Well, if PostgreS is not highly available. So if PostgreS is down, our whole service is down. But in this case, we can survive. It's fully available. It's highly available. The system is highly available.

[00:47:24] JM: Right. Right. Right. Okay. Right. A Redis server might be the obvious other option, because Redis would be more available.

[00:47:33] AS: Yeah, or less. I don't know. Possibly.

[00:47:35] JM: Right.

[00:47:36] AS: I mean, Redis would have to – You need the Redis cluster, I guess, and then you write at least three nodes or something like that.

[00:47:45] JM: Yeah. In this case – So you need a – Do Faust services, are they clustered by default?

[00:47:50] AS: Yeah. Yeah, you can start as many workers as you want.

[00:47:54] JM: Okay, interesting. I think I get it. So maybe just to wrap us up, we're nearing the time. This has been really interesting. Can you just tie a bow on this? Give a summarization for what is unique about Faust and why you built Faust?

[00:48:11] AS: Oh, wow! I think of coming – Well, the reason we built Faust is to make easy to express complex distributed systems, I guess, and to make them easier to deploy. Yeah, that's the idea behind it.

[00:48:27] JM: Yeah.

[00:48:27] AS: But we don't know how to explain everything of it yet in a easy to –

[00:48:32] JM: That's fine.

[00:48:34] AS: The Kafka Streams documentation has quite a lot of information as well.

[00:48:39] JM: So do you think of this as leveraging Kafka? The development of Kafka Streams, that was a key inspiration point for Faust.

[00:48:49] AS: Oh, yes. So we took Kafka streams and we made them into actors basically. So there's no DSL. Instead it's actors with a mail bot. Do you know what actors are?

[00:49:00] JM: Yeah.

[00:49:01] AS: Yeah. So we took Kafka Streams and added actors to it.

[00:49:04] JM: How would you explain actors to somebody who's listening who doesn't know what those are?

[00:49:08] AS: Actors is a method of concurrency, where each actor communicate by message passing, where actors communicate by message passing. It's a method to keep concurrencies sane and safe. But it also works for distributed systems.

[00:49:24] JM: So you said your idea was to use Kafka Streams to build an actor model.

[00:49:29] AS: Basically, yeah. But it's not really an actor model, because in the actor model you communicate with specific processes. That's why we call them agents, because an agent runs on many machines. But it's inspired from the actor model.

[00:49:45] JM: Fascinating. Okay. I feel like I keep talking to you about distributed systems for a long time. What are the future plans for Faust? What do you see as the important implementation steps you need to make?

[00:49:55] AS: I think the exactly-once processing is one, and there's also a lot of just testing it, battle testing it. We have integration test to ensure that things are consistent. We have like a live integration stressor suite as constantly sending data into the tables to check that the values are correct, the counts are correct, just like defining what it is, I guess and what it will be used for.

[00:50:20] JM: Yeah, and that banned users use case seems like a really good testing environment, because it sounds like Faust isn't exactly on the critical path.

[00:50:30] AS: No. Faust is – But Faust is serving all of that, right? It's a Python process. It uses Faust. Alongside that, there's web sockets that would be pushed through. We read from Kafka. We push the web sockets. It's using Faust all over the place.

[00:50:45] JM: But if you had some critical failure with Faust, you would be able to rebuild from that Kafka topic pretty quickly. Whereas as you said, if you were to use – Using Faust for like the ordering service and there was some problem with the Faust software, that would not be a good situation.

[00:51:00] AS: Yeah. No, we're not going to do that just yet.

[00:51:03] JM: Just yet.

[00:51:04] AS: Yeah. Not just yet.

[00:51:05] JM: All right. Well, when you decide to do that, let me know. I'd love to do another show about that, standing up that service.

[00:51:12] AS: That sounds cool.

[00:51:13] JM: Okay. Well, Ask, is there anything else you want to add about Faust or data engineering in general?

[00:51:19] AS: I want to give a shout out [inaudible 00:51:20], my co-worker that I created Faust with. That's all. He had a birthday recently and is sick today.

[00:51:27] JM: Well, cool. I'll add an additional shout out to Robinhood as a personal user of it.

[00:51:32] AS: Oh, cool!

[00:51:33] JM: If anybody else from the Robinhood team is listening, you want to do another show about something data engineering or otherwise, I'm very curious about how this platform is built. So I would love to know more.

Thank you, Ask. It's been really great talking to you.

[00:51:47] AS: You too.

[END OF INTERVIEW]

[00:51:50] JM: We are all looking for a dream job, and thanks to the internet it's gotten easier to get match up with an ideal job. Vettery is an online hiring marketplace that connects highly qualified job seekers with inspiring companies. Once you have been vetted and accepted to Vettery, companies reach out directly to you because they know you are a high quality candidate. The Vettery matching algorithm shows off your profile to hiring managers looking for someone with your skills, your experience and your preferences, and because you've been vetted and you're a highly qualified candidate, you should be able to find something that suits your preferences.

To check out Vettery and apply, go to vettery.com/sedaily for more information. Vettery is completely free for jobseekers. There're 4,000 growing companies, from startups to large corporations, that have partnered with Vettery and will have a direct connection to access your profile. There are full-time jobs, contract roles, remote job listings with a variety of technical roles in all industries and you can sign up on vettery.com/sedaily and get a $500 bonus if you accept a job through Vettery.

Get started on your new career path today. Get connected to a network of 4,000 companies and get vetted and accepted to Vettery by going to vettery.com/sedaily.

Thank you to Vettery for being a new sponsor of Software Engineering Daily.

[END]