EPISODE 1136

[INTRODUCTION]

[00:00:00] JM: Robinhood is a platform for buying and selling stocks and cryptocurrencies. Robinhood is complex, fast-moving, and financial. And together, these things require high-quality engineering in distributed systems, observability and data infrastructure. Jaren Glover is an engineer at Robinhood, and he joins the show to talk about the problem space within Robinhood as well as the specific DevOps and software engineering challenges.

We're looking for writers and podcasters for Software Engineering Daily. If you're interested, send an email to erica@softwareengineeringdaily.com.

[INTERVIEW]

[00:00:40] JG: Thank you so much for allowing me to be here. I appreciate that.

[00:00:42] JM: Yeah. Let's start off by talking about the canonical problems of building a trading platform. You work at Robinhood. So tell me, what are the reoccurring problems? What are the canonical problems that you see over and over again in building a trading platform?

[00:00:56] JG: Gotcha. I think just from my perspective I think is to focus more on the infrastructure side. So from my experience, it's just kind of maintaining, scaling, deploying, iterating and kind of maintain a reliability of the distributed systems was a unique challenge, because it was not something that I particularly had the domain expertise in and something we kind of had to learn by doing. And something we want to do to kind of create this magical experience for our application developers, but also our customers.

[00:01:25] JM: Take me through the anatomy of a trade. When a single trade gets executed, what happens?

[00:01:30] JG: So we usually don't talk about the anatomy of a trade, but I can kind of talk at a high-level about some of the things here. But we tend not to talk so specifically about the anatomy of a trade.

[00:01:41] JM: Yeah. Can you take me through in broad brushstrokes some of the difficulties and some of the challenges around executing a trade?

[00:01:49] JG: Yeah. No. I think the unique challenges there are making sure you have a reliable, durable databases, reliable low-latency infrastructure. So some of the examples that we saw that we originally wrote some of the original trading services in Python, and then we rewrote it in Go lang. We did the same thing with our market data services. So those are some of the unique things there. So if I were to recap, that's ensuring low-latency. So sometimes rewriting services for the greater good from Python to Go lang, to those kind of having durable, scalable data stores. And that could be RDS or that could be kind of your distributed system that might be the queuing system of choice, and just ensuring that you have those durability and reliabilities. And ideally, low-latency. And the think that I'm super passionate about is observability. So you kind of have those three things kind of uniquely have the ability to maintain the regulatory requirements, the customer experience and a scalable experience as we onboard more and more users.

[00:02:47] JM: How does regulation affect trading and the programming around trading?

[00:02:51] JG: Yeah. Robinhood is a broker-dealer, and from the high-level, you can kind of imagine we have multiple governing bodies that enforce different regulations. And our job is to kind of implement those work with our general council and work with a regulatory to make sure that now when we implement the constraints and restrictions and enforce the kind of rules and regulations, but also have the metadata in the auto trail to kind of ensure that we're not only seeing what we're doing, but we can prove that we're doing. And that creates unique challenges that are pretty exciting from a developer perspective.

[00:03:22] JM: What do you work on at Robinhood?

[00:03:24] JG: Oh, yeah. I think my claim to fame at Robinhood is I help scale the infrastructure from roughly a 150,000 users to well-over 10 million. And kind of what that translates to, think distributed systems, data stores, logging, alerting, observability. And the two most recent projects that I worked on, spent the ton of time on is taking proof of concept of streaming the platform from a POC to being in my bias opinion is the second most critical system at the company. And then kind of switching langs and pivoting from streaming to redesigning the way that service network discovery happens at Robihood, which was unique and exciting, because I didn't have a ton of network and experience prior. Both of those challenges requires zero to one thinking and thinking from first principles.

[00:04:12] JM: Why is the notion of time so important for a trading company?

[00:04:15] JG: No. That's a really great question. I think it's pretty important from a customer perspective and from a regulatory perspective. Our job is to ensure that we execute and process orders in the order that they gave us, and time is really helpful in allowing us to do that. And using this context, when I'm talking about time, I'm talking about NTP.

[00:04:35] JM: So what is NTP?

[00:04:37] JG: Yeah. NTP is the network time protocol that was developed over the last 20 years. And you can kind of think of this network protocol as a way for clock synchronizations to happen in computer systems to mitigate the effect of like network latency between that. So you kind of imagine at a high-level, you have computers A and computers B and you might have atomic clock. NTP allows you to ensure that computers A, which might be in Seattle; and computers B, which might be in Louisiana are all syncing or churning towards syncing to the atomic clock in Utah.

[00:05:10] JM: And what else have you had to learn about time standards and maintaining equivalent time?

[00:05:14] JG: Yeah. So I think a really great example here is about six months after I joined Robinhood, we got this regulatory notice called clock synchronizations. And as I said earlier, we have these governing bodies that ensure that we enforce this specific regulations on the firms. And this specific request here was to ensure that our trading computers were syncing towards what we call NIST atomic clocks. And the NIST is just I think is National Institute of Standards Timing, which is a non-regulating government body that kind of ensures that those atomic clocks are as they say, great time.

So for context that I just spoke about earlier, time is super important for two reasons, we want to make sure that we're compliant with our regulatory bodies. But two, we want to ensure that we process the trades and orders in the order that we got that, and NTP is our way that we kind of achieve that. And the NTP is the protocol. And then from a UNIX perspective, NTPD is the software daemon that runs on your UNIX-based systems to kind of ensure and implement that protocol, which is basically implementation, if I remember correctly, or the RFC 5905.

And this seems pretty straightforward, and I think the initial request is something that's pretty easy to do. But I think this actually becomes pretty difficult in our environment, because as you know, Jeff, Robinhood is built inside of AWS, which is a cloud provider and actually keeping time and maintaining time in a dynamic con environment. It's slightly more complicated than physical hardware because of just the dynamic nature of the infrastructure.

So to implement that particular constraints, we have to build up an understanding of time inside the company. Because frankly at the time, no one really had a ton of core expertise around NTP and time how it may or may not exist inside of a cloud provider. Actually, one of things I did was that I reached out to the ntp.org people, kind of blindly emailed them. It was like, "Hey, you have atomic clocks that match the new standards. We needed sync towards those." And they were extremely helpful in answering a lot of nuanced questions that I had around NTP. But more importantly, they were willing to ask those questions, but were very direct and kind of telling us that, "Hey, if you sync all your computers to atomic clocks, we're going to cut you off and ban your IP."

So that was really interesting that I had to kind of first understand what NTP was. High- NTP existed inside of AWS. What is the software to implement it? So going from NTP to NTPD, and then how those two things are implemented inside of AWS. And then I had to kind of work together to ensure that not only that we comply it to regulation, but we also were being good stewards and good partners to ntp.org, which was a pretty exciting opportunity.

And then unique things here is that, also, I had the opportunity to – If you kind of imagine, as a infrastructure engineer, my job is to build and implement these technologies and constraints that meet the regulation. But also I got the opportunity to work with our general counsel, right? To make sure that we were translating and understanding the ask that was being asked at Robinhood and I was kind of delivering on software and infrastructure that kind of ensure that we met that request.

So definitely a couple balls in the air that we had to juggle there, but it was a pretty unique opportunity to take something as important and complicated as time, implement it, to kind of ensure that the systems inside of the trading computers inside of Robinhood kind of maintain that. I’m happy to talk to you in a little bit more detail about some of the unique things that we have to do to kind of achieve that.

[00:08:46] JM: I’d actually like to focus the conversation for a bit on AWS. So I believe Robinhood is the first broker-dealer to be built entirely on AWS. Can you tell me more about the difficulties of that?

[00:08:56] JG: Oh, yeah. So I don't know if that’s factual, but that's actually really interesting. Yeah, I don't know about that, but I know that you can kind of imagine a world that a lot of the things that kind of the one for Robinhood, and the regulations with a lot of that was built in the place where there was a lot more physical world. But some of unique challenges in general is just understanding how do you leverage the scale and platform on AWS in a way that provides the proper leverage for the company? Right?

I think some great examples of that is RDS has been a really great partner for the company. We use the Postgres version of the RDS there. And that's been great and has done a really good job of kind of scaling the company. It’s kind of amazing to see RDS scale as we scale over the last five years. But there's also unique challenges where sometimes there are new products that come off that don’t particularly work or just don’t work for our particular use case. And learning, testing, verifying and kind of making sure that the platform and services that we use does two things, right? Provide the quality experience and reliable experience that we come to know and trust and try to provide to our customers. But also at the same time, making sure that we allow ourselves the flexibility to not always be locked-in to a particular cloud provider.

And that’s why one of the reasons for better or worse, I always say that we manage a lot of our services on EC2 compute. So a lot of our distributed systems are actually managed on EC2.

Compute versus deploying or using the off-the-shelf versions of those that often the cloud providers provide.

[00:10:33] JM: Do you have a set of blessed AWS services? Like the things that you're actually willing to use as supposed to just letting people run wild and choose whatever AWS services they want?

[00:10:42] JG: Yeah, that's a great question. I think the autonomy of decisions to use a PaaS service is kind of independent. Each team kind of makes that decision. And you think of the architecture design that the individual application on it might test. But a lot of the challenges that any time a new decided to use a new technology, we kind of actually sort of heavy test it. Is there proof of concept? PaaS products are amazing when you play inside the guardrails in which they present it. The moment you kind of get outside those guardrails and move from a generic compute to something that might be super domain-specific, those are the ones that you kind of bump against guardrails and now have the kind of quality of service that you want.

From my perspective, when people are pitching new services to use, we tend to not have too strong opinions. The goal is that the individual engineer goes off or the team of engineers goes off in kind of a proof of concept of how does this deploy on the open source version managing on EC2 or inside of container versus the PaaS offering.

A good example of that, I think that we – Early, Elasticsearch on AWS, we tend to have some issues with that, and we found that kind of our workload and how we were using it and the call straight-outs that we require just didn't make sense for us to use the PaaS. So we end up managing our own Elasticsearch clusters on EC2. But there were kind of use cases were spinning up as simply Elasticsearch cluster that wouldn't scale. That timing, which was many years ago, and this might not be true today, was very well-fitting.

So a good example of that is for a lot of the application logs, we kind of self-host an Elasticsearch cluster on EC2 instances, whereas security logging, we actually spun-up a self-hosted cluster, a PaaS version that really worked for them, that work with them for that particular use case and scale they work before this kind of broader, lot more dynamic use cases didn't particular work. So that's kind of the examples there, that sometimes it's not all or nothing, but sometimes you use it for certain use cases and sometimes you don't. But ultimately, that's kind of a decision that the individual team makes. And then often sometimes if it impacts others teams, you have to kind of get buy-in.

For example, because her self-host Kafka on EC2 instances, we’re probably not going to allow for someone to use the PaaS offering, just because we just built up a ton of domain expertise on the EC2, and we work really hard to provide a reliable quality service that if you decide to kind of go off on those rails, which if you wanted to do, that’s fine, but you might not be able to get the broader organization of support there. But I think we’re open to experimentation. I think there're some teams are excited about trying some new languages out inside the environment. But those are the tradeoffs you make in deploying new infrastructure in your environment. When have the paved role that is kind of smooth and has the quality of service, divisibility, observability and alerting that you’re used to.

[00:13:29] JM: What about tools outside of the AWS ecosystem, like the HashiCorp suite?

[00:13:33] JG: Yeah. We’re are big fans of the HashiCorp team. I think we’ve used VirtualBox or some local development, but not a lot. We use Vault, Packer, Consul, Terraform, right? I think we use everything, but nomad at this point in time. There might be one or two other services that we don't use. But we’re big fans of the quality of open source that the HashiCorp organization has provided for the organization.

[00:13:56] JM: Can you tell me more? You use Terraform a lot, right?

[00:14:00] JG: Yeah. Yeah. Yeah. Terraform was introduced right before I got there in either 2015, I think. But generally, kind of the goal there was, at a high-level, young Robinhood thought that we do want to be beholden into a single cloud provider. And Terraform kind of gave us that opportunity to abstract infrastructure to code. But also, in theory, if we ever want to change cloud providers, have the opportunity to do that. And it was the contemporary technology. And for the most part, it worked extremely well and growing with our organization. I think the only thing that becomes interesting with Terraform is that once you have a ton of infrastructure, your Terraform plans and applies can take a long time. So that's like the unique challenge there.

Also, the UI today is such that it really takes a DevOps system admin or an infrastructure engineer to really understand how to use that. I think that there is a unique opportunity to kind lower or higher the abstractions so that your average engineer who may be only writes Python has the ability to do that to implement the infrastructure requirement in Terraform.

So we're thinking right now how do we kind of increase the abstractions, reduce the time that it takes to use Terraform and kind of reduce the knowledge required to build infrastructure in a safe, reviewable, reliable way.

[00:15:20] JM: Tell me what happens during a deployment. Do you have any interesting processes around CI/CD or anything else around deployment?

[00:15:28] JG: Yeah, we do, but I don't work as much with that, to be honest. But I think at a high-level, the economy that I would say there is that the team is working really hard to kind of migrate from an EC2 world to a world where a large percentage of our compute is running on a self-hosted Kubernetes cluster. So I don't work as closely with those two things, but I'm pretty optimistic. And the benefit – There's no free lunch in infrastructure, but I'm pretty optimistic in the type of compute that makes sense for Kubernetes. And once it's being on boarded, reduce the friction of deploying.

We're a fairly mature organization where you have engineers that are coming in and kind of expecting that type of deployment, blue-greens and container-based deployments and things of that nature. So we're trying to make sure that we build a mature infrastructure with the right level of abstraction. I think one of my peers once said that we want provide a PaaS service, kind of like Heroku, for inside of our organization so that our engineers can have the necessary velocity and get the guarantees that they come used to in the EC2 world.

[00:16:35] JM: What are the programming languages of choice inside Robinhood today?

[00:16:38] JG: Yeah. No. I think the two languages that we've historically spent the ton of time with his Python and Go lang. From my perspective, I think there are some experimentation on the edge with Rust and we're migrating from not only HTTP to GRPC, and also doing a ton of screen computing. But we're historically been known as a Django and Postgres shop. And then we've eventually kind of went into a Go lang and Postgres shop. We're starting to get more exposure in Java and kind of things, to distributed systems. There might be some one-off things where people might do low-tests or things like that in Java. But a lot of my Java exposure is via distributed systems like a Elasticsearch, and ZooKeeper, and Kafka, for example. But most APIs are written in Python and Go.

[00:17:22] JM: So when you say your Java experiences around Elasticsearch, does that mean you're actually going into the open source Elasticsearch code and making alterations to the open source project?

[00:17:32] JG: I think some of the examples with Java exposure, I think the when you're deploying these distributed systems, kind of the scale and the unique workload that we're doing, we're doing a little bit of that. But more of the kind of Java exposure is tuning your JVM or understanding the types of compute that you need to bring to the services that run on that, which is, again, a better expertise. For the longest in my experience, the JVM was a black box. But the more time I spent time understanding with ZooKeeper and Kafka and Elasticsearch, I started to see like, "Oh wow! Not only do we need the observability for the applications and the software that ZooKeeper or Kafka, but you really need to understand what's going on with the JVM, because the JVM would tell you a lot about kind of the health of the service, keeps and things of that nature. And then starting to have opinion on which JVM you are using, and the version, things of that nature. So that's usually what happens.

There's been a few times off top my head that we had to kind of go in and actually alter the codebase. But majority of the time, it's just understanding how to deploy scaling to Java-based distributed systems in a way that provide us the quality of assurance that we like to provide to our customers.

[00:18:48] JM: What does observability mean for Robinhood?

[00:18:50] JG: Yeah. That's great question. That means a lot of visibility with the application is or is not doing. I like to think of observability, meaning log's, alerts, and then instrumentation, which could be tracing, could be counters, could be gauges. But interesting thing about – One of the things as I reflect over a time at Robinhood is that our application engineers became high-users of StatsD. Understanding of latency that is provided by NGINX, and not so much as logging. I think that created this interesting culture while a lot of things were going on with StatsD and latencies and things of that nature. And then, eventually, we kind of up-level our logging along the way. But for better or worse, which I think for the better, the obsessiveness with ensuring that we have visibility to what the application is or is not doing a super important.

And then when you have that, you can train things over time and build alerts that allow you to catch problems before they happen. Be able to kind of catch problems before our customers experience them, right? And I think that's the unique ability of latencies, StatsD. And just the ability to kind of capture not only what your application is doing, but what your host is doing. I think being able to understand those two dimensions is pretty important. And that kind of allows you. It’s just the language that me as an infrastructure engineer, infrastructure owner, thinks a lot about. It’s also something I'm really part of the average application engineer at Robinhood gets excited and comes to expect, right? Where is my latency graph set? So that I can set alerts on that? Or how do I ensure that I have the 200s and how to alert on 500s? Those are kind of a lot of the questions that, with us being a heavy HTTP organizations, some of the questions that we come to kind of expect and know about and understand so that you can kind of know not only your host is doing, but what your applications is doing.

[00:20:41] JM: How do you use Prometheus?

[00:20:43] JG: So yeah, Prometheus is our observability system have today. We went from using open TSB, which is a Java-based open source product, and replacing it with Promethus. And the reason we did that was because, at the time, we started using observability and it became a pretty critical part of the company. As we talked about, engineers were using it a lot, and then inspecting kind of the visibility to their application. And at the time, this is no maybe 2016, 2017, the open TSB was based on HBase, and the data team wasn't using the HBase. So the question was do we want to scaleout open TSB? Or do we want to kind of build something else or buy something else?

So the journey was we kind of made a decision that no one really wanted to scale or tune or learn HBase. I think some of the more senior engineers had saw that their previous lives and didn't really want to reduplicate that. And then to my point earlier, I didn't have a ton of job experience or the thought process of scaling Hbase for metrics didn't really make sense. And at this time, we also want to decouple. At a high-level, we were using open TSD collectors to push model while we push them to Kafka, and then consume them into the data store the time. But the problem with that is that you have this coupling of Kafka. If you were having issues with Kafka, you can have the observability to actually observed your system so that it became an issue that we want to decouple.

And then the question kind of involved, “Okay, do we want to buy a solution? Or do we want to build a solution?” And luckily at the time, we hired someone by the name of Joseph who – I was kind of tech lead, and he was the kind of implementer here, and we were asking the questions, “Do we want to buy something? Or do we want to build something?” And if we want to build something, at the time, all the cool kids were using Prometheus at the time, Grafana and Prometheus. And I really love the simplicity of the implementation, right? It was a Go binary. We could deploy it. It stood by itself.

There was some questions about scaling the backend at the time. But we felt that our time that our systems wasn't complex enough where we should be able to grow with the community and grow with your storage for some time. Or we evaluated some of-the-shelf vendors, and the quotes that we came back were quite expensive. So I don't remember the exact numbers, but almost $1 million for a load that wasn't that large at the time in retrospect.

And so Joseph kind of did the deep dive into Prometheus and came out with the executive decision that, “Hey, not only is this software seem to be pretty robust. There was a big blog post that came out at the time that there were concerns that the pull model couldn't scale. Prometheus team did a great job of making the case for why the pull model would work and would scale. And then the community was so rich. So we were able to make a decision that we want to bail. We did a proof of concept. It worked. StatsD migrated seamlessly over to Prometheus. We have to do some kind of UI things to ensure the transition of some of the other metrics that we had, specifically NGINX, which were pretty important at the time.

But man! Let me tell you, this was like a beautiful poetic opportunity of picking the right technology with a rich community and just growing with it. For the most part, the storage didn't become a problem until like recently. But that was beautiful to see that. And we learned a lot about who we were, challenge the preconceived notions, update our parts when we got more information, the push versus pull. And then taking the opportunity to remove some technical debt. We want to decouple. We want to ensure that metrics had the highest uptime. Because if a particular third-party infrastructure was down, then it kind of blows a hole in our ability to actually have observability. So this was the opportunity to pick a new technology, but on a new community. Onboard to a language that we were getting a lot more comfortable with, right?

I told you earlier, we were more of a Python shop, and we eventually went to Go lang. And then have the opportunity in general to remove the tech stack and better ourselves and say that we think for the cost tradeoff, we can engineer a solution that really works with us and that kind of allows and provide the quality that we needed. And I thought the implementation that Joseph executed at the time – Gees! Had to be late 2016, beginning 2017. I thought it would give us maybe 18 months, but it actually gave us maybe three years of quality work. And I'm just pretty impressive, our ability to kind of own that. The reality of it, buy versus build, is a kind of a trick-or-treat out that every engineer and organization goes through. And I think, this time, with the benefit of hindsight, I think we did the right thing.

[00:25:29] JM: Can you tell me more about what it takes to monitor such a high-volume of trades and have the right observability information around each of those very sensitive transactions?

[00:25:38] JG: Yeah. One of the things that we did a really great job of is we're inside AWS, and AWS has ELBs, ALBs. One of the things we did first was we ensured that they came with a lot of visibility, and we ensured that we ingested that visibility and also putting alertings at every layer. We use NGINX as a reverse proxy. So ensuring that we not only have visibility ingested into our monitoring system at the highest level, which is an ALB levels. Then at the kind of reverse proxy layers, that kinds of gives load-balancing and then our ability to kind of have dynamic backends. Ensuring that we have the visibility there. And NGINX was really great in allowing us to kind of provide the level of visibility that we need at the API levels.

And then for anything we didn't have, Lua became really interesting. We kind of learned from the tech legends like Dropbox. And we read a ton of the literature and blogs that they put out about expanding in NGINX, abilities using Lua. And then taking the next step of action and say, "How do we can ensure that at an application level, knowing when to properly use gauges and counters and knowing when to introduce logging?"

And then as I think more about it, you have the ability at the ALB level, NGINX level. Then we have exception handling, right? So there're two big projects that do that. I think one is Bugsnag and the other one being Sentry. So we use Sentury self-hosted again, version of Sentury that allow us to kind of capture the exceptions, which actually became something I didn't use a ton with. But our application engineers just love. They just love the ability to kind of have visibility to the exceptions there. And they kind of use that a little bit more than logging, which was really interesting. Honesty, from my perspective, as an infrastructure engineer, I kind of cared more about the logging than the exceptions. But the application engineers were really relentless in integrating and ensuring that they have the projects and the visibility at the exception.

And then eventually we built out a logging pipeline. So you kind of aggregate those three or four components together. And you have like a really good, kind of holistic picture of what over time you kind of understand what is a healthy system or how the API looks like. And then you have the ability to provide the visibility and understand when deploys go bad. And when maybe you need to upsize your compute because you're struggling from success, and it turns out you shipped an API that was high utilization. So you need to not only expand the compute dynamically, but also maybe potentially upsize the instance that we're using there.So that some of the ways that we think about observability.

[00:28:11] JM: Let's just go little bit deeper on this scripting in NGINX, the Lua material. What did you learn from Dropbox and why does it apply to Robin Hood? What kind of scripting are you doing on NGINX?

[00:28:23] JG: Oh gosh! Yeah. There were some modules for tracking networking calls or observability that was provided there. And kind of stepping back, I remember when I came to the Valley in 2015, Dropbox was really well known for kind of the engineering blog posts and their prowess. Frankly, Dropbox use NGINX, and obviously worked at a higher QPS level than we were at the time. So we thought that we can kind of look up there and learn from them and break down the things that really made sense.

So some of the things that kind of allow you to do that – And I don't know this to be a fact, but I think a big part of why Dropbox use a lot of Lua is because they didn't want to use the enterprise version of NGINX. Though I think they potentially did. I think we, our big proponents and investing and also leveraging open source technology. And we wanted to kind of expand in order to get the level visibility that we needed from NGINX. We had to integrate some of the Lua plugins.

And then there's some really interesting routing. I did a bit of research and my colleague ended up productionizing, but Lua can be used to kind of dynamically route traffic based on the HTTP headers. And it's pretty fast. You can kind of build NGINX little a Lua plugin and you don't have to pay the cost of this extra language to be able to kind of ensure that you don't add it too much latency to that request handling.

So two things to repeat. One is there were some visibility to the metrics and observability that we want from NGINX that Lua provides. And then Lua also allows for the ability to dynamically route requests without kind of paying a huge tax when using a relatively same programing language in Lua.

[00:30:02] JM: How do you use Kafka?

[00:30:03] JG: I'm glad you asked about that. That's probably the thing that I spent the large percentage of my time here at Robinhood. And I can kind of build out some context here to kind of speak here. So when I joined the company, as I talked about earlier, we were basically an HTTP and batch company, which basically meant we're running a lot of things on cron jobs. We rerunning a lot of things writing Python in these batch efforts nightly jobs. And for the most part, this really work. But thanks to the data team at the time, we sort of asked them the question of, "What would real-time look like? How do we kind of empower our executives and our product managers to be able to make real-time decisions?" When the market opens up, we want to ensure the ability to understand real-time what I customers are seeing.

And a lot of the baking industry and broker-dealers are kind of dumb on this batch mode, and we kind of challenge ourselves, like, "Hey, technology is maturing." Our ability to manage the distributed systems is maturing. I think there's an opportunity to not only take our engineering to the next level, but also provide this great experience, real-time answers. Whether that'd be providing answers to our customers, our third parties, or internal to why executive team answers about how the day is going and processing their data. So as those things kind of evolved and the technology evolved, we decided to embark on this journey of how do we take Robinhood from a batch to a streaming platform. So that was kind of the problem statement in 2016.

One of the things that the time we were thinking a lot about was we just didn't want scalable or distributed cron current jobs. And at the time, if I remember correctly, we either didn't have a lot of spark exposure. Again, that was a Java-based platform. We did have a ton of that, and we didn't think that our workload was going to end up in a state that needed a ton of spark. And we started asking that question. We started doing some interesting things with cron jobs, basically distributed cron jobs with Airflow. But we really want to take it to the next level. And we have some level of confidence that we can manage and scale a fairly complicated distributed system like Kafka, because we had the luxury of learning the ability to kind of manage on the scale of three other distributed systems on the way to that, right?

So the first one we got introduced to was Elasticsearch, which was a V1 version of real-time analytics. Not only was it the data store for our original logging platform, but also some of the queries and things that our PMs want to answer was being driven-out Elasticsearch. Then we decided to kind of deploy ZooKeeper, right? I remember when we started talking about ZooKeeper, the consensus was really robust and definitely can scale to our needs. But tuning and scaling it is always a challenge, and everyone kind of had horror stories of doing that because of ZooKeepers API and the tooling that was provided around that.

And we kind of built our expertise around ZooKeeper. Thanks to our data team who built their internal future flex system that basically use ZooKeeper as a backend. So while the data engineer was responsible for building this robust API that allow future flags, our team was responsible for how do we scale and maintain ZooKeeper and start to learn a lot of things about, "Oh wow! [inaudible 00:33:14] consistent algorithm of Elasticsearch was widely different than Zap or ZooKeeper."

And then we kind of had the luxury of learning our very first Go lang based distributed system, which was our friends, HashiCorp counsel. It allowed us to do DNS-based service-to-service communication. And I really like this, because we could introduce service discovery into our environment without changing a lot of things. And the application engineers were very comfortable using DNS.

Basically, thanks to learning, the tuning, debugging, deploying, managing, I think all of these services in fact were on EC2 compute open source versions. We felt fairly confident that we could kind of migrate to what we thought was the future of streaming and real-time nature with Kafka. And at the time, it actually wasn't clear. But I think, again, we got lucky embedding on the right community that the team would eventually became conformed, really built up a rich ecosystem. And the mothers and fathers of that community really invested in it.

In providing this robust streaming platform that was still pretty operationally-heavy to manage to be quite honest. And happy to kind of spend some time doing that. The unique challenge here also was that when we decided to introduce Kafka into our environment, we also had to double down almost again on ZooKeeper. Because I think now, and definitely in 2016, it was concerning the architecture design meeting of saying like, “Hey, team, you not only want to introduce this thing called Kafka.” But Kafka strongly depends on ZooKeeper. So we have to kind of bet as an organization that Kafka was worth the double downing on ZooKeeper. And we thought as a company, that was the right decision to make. And it was, and we've fully transitioned to a streaming platform company with Robinhood.

Gosh, Jeff, it was different without its kind major challenges. And I think those three major challenges were the client, the infrastructure, and then just general the application owners. And I’m happy to kind of dive in there if you think that would be useful.

[00:35:11] JM: All right, let's go even deeper. Why not?

[00:35:12] JG: So yeah. If I were to break them down, if I’d repeat those again, that’s client, infrastructure and the application owners. So when we talk about clients here, the context that I’m saying is kind of your producer and consumer. So one of things that I take full responsibility of not seeing the forest for the trees was not only should you focus on building reliable, scalable, tunable infrastructure, but the class became very important also. And I think at the time, a lot of the robust Kafka clients were built in Java, and we were using Python and Go lang. So it was a very underdeveloped community at the time. So we had to kind of do a few things. One, we had to build our own wrapper around the clients and consumers. And that UI that was provided for the new developer that were coming on the building producers and consumers was not at the level or abstraction that it really needs to be.

And I think there was an opportunity to keep it simple there, and we didn't actually deliver on that end up creating this wild goose chase of, “Is it ZooKeeper? Is it Kafka? Or is it the clients?” I think if we had an opportunity to kind of double down and investing those client, we would have had the opportunity of leveling the abstraction. I give all credit to what the data team did at the time. So, at the time, actually the Kafka Streams was only implemented in Java. And our data team was like, “Well, we need Kafka Streams for a lot of the fraud and data processing and risk checks that we did.” So instead of migrating over to a Java ecosystem, we have the luxury that we have the author of Celery, which we’re using RabbitMQ, which was kind of our first streaming platform was RabbitMQ and Celery. And we had the co-creator or the creator of Celery who kind of took on the challenge of saying, “Hey, I think we can build our own version of Kafka Streams and Python called Faust.” And she developed that, and we eventually open sourced it.

Jay Kreps tweeted about it. So it was pretty rich to kind of not only migrate to a streaming platform like Kafka, but also take the attributes of the community and leverage our engineering expertise, which at the time we had – Thanks to the journey Celery, that we built our own Kafka streams implementation in Python. And this is open source today. So for your listeners, if you go to Robinhood GitHub, you see the Faust application there.

And then secondly, we dive into the infrastructure. Gees! Jeff, let me be honest with you. Taking Kafka from 0 to 1 and not having tooling for managing stateful services. And this was a lot of the feedback I gave to the Confluent team at the time, was that managing these systems at Scale became very difficult. One of the reasons of managing stateful services – And as you know, in AWS, servers die every day, Jeff. So we have stateful services and you have – The way that the replication worked in Kafka. So maybe I’d take some time for Kafka and Elasticsearch.

Elasticsearch is really easy because I could just spin up new servers. I could take an auto-scaling group from 10 servers to 25 servers and the data would just replicate automatically and migrate over. It would set the script, thanks to [inaudible 00:38:15] algorithm, will self-discover. Then other Datadogs, and then the data would just kind of replicated outwards.

But with Kafka, because of the unique structure that it had in place there, that when you added new brokers to your cluster, they didn't automatically get the data in the data didn’t migrate over. So we had to build tooling that would allow us, one, to be able to zero touch operations when a broker died and the auto-scaling group kicked in a new broker. We need to kind of ensure that the unique broker ID was generated so that that data could be replicated. And it would have an operation’s overhead.

Because what was happening, we were getting these emails from AWS saying that they were retiring the underlying hardware for a particular Kafka broker. And that would mean I would be spending next few hours manually kind of moving the data around the cluster using some tools in Python script and batch scripts that we developed to makes this easy.

So what we have to do instead of generating runbooks, which I tend to focus on building tooling and mental models for like solving problems versus runbooks. But built out some tooling that allows the dynamically generate these consistent broker IDs so that when broker 5 died because AWS decided to terminate the host, when the ASG kicked in, it would use this tooling that was this Python script as kind of backed by Consul to kind of provide to the distributor a locking mechanism. Would allow us to deregister the broker ID out of Consul. So when the new note came into auto-scaling group, it would pick that broker ID 5, and when it registers itself inside the Kafka cluster with broker 5, it got all the data. And this was a zero-touch model. So that made it super easy to be able to have this new broker ID.

The second big thing we did is – This was full circle. We talked a little bit about Terraform earlier. We really liked the concept of owning your infrastructure via GitHub and doing pull requests and code reviews. But the problem was, with Kafka, you kind of have the high-level. You have this concept of brokers. And each broker cluster has topics. And each of those topics are broken into partitions that kind of allow for the parallelizations of producing and consuming.

The problem we were having is we had built up a robust resilient infrastructure, but the rate at which we are onboarding new engineers into the Kafka clusters, they all want their own Kafka topic and they maybe wanted the topic to be in different partitions, different consistencies, right? Order keys and things of that nature to ensure like consistency throughout the data. And the toll there was very high.

So what one of my colleagues, Harry, did was he effectively built a Terraform for Kafka topics called Metamorphosis. And I'm trying to get the team to open source it right now. That was pretty great, because it allowed for this application. So moving the creation and managing of Kafka topics from command line tools and scripts to be in this codebase that was code review that had great examples for creating. And we can kind of offload that to allow for self-service. So self-service topic creation and expansion was pretty impactful.

And at the time, that was a bad tradeoff we were making, because it was one command. It didn't take a ton of time. But the team couldn't see the forest for the trees. That a lot of that was adding up over time. Thankfully, Harry saw that. Built out this tool using some Go and using that robust API that Kafka had and built effectively Terraform for Kafka topics.

And then lastly, the big thing here was we had a ton of visibility and knowing when consumers were lagging. But we just didn't have the information to be able to answer to the application engineer while their consumers were lagging. Historically, we didn't have a ton of problems with producing. But what was happening is the application owners were building these new distributed applications, and the consumers were lagging, and they didn't have a great way to answer that.

So that was particularly challenging here. And that kind of actually parlays in the last bucket that we've talked about, which is the application owners. By this time that we kind of evolved in this journey of moving from batch to being a streaming platform, Jeff, we went from maybe 50 engineers into 150 engineers. And each new engineer that would join would have a variety of experience, a wide range of experience with building distributed applications and dealing with stream processing.

So we needed to create a way to make onboarding into what it takes to build a robust distributed application to kind of raise the table stake of information and build out the tooling. Because, actually, building a distributor application, you have to kind of account for a lot of different things, right? You need to be able to build resilient applications, producers an consumers, because there's a lot of challenges that come with that, right? What happens if the cluster isn't available? What happens if you have networking problems, right? What happens when service churned? How do you kind of ensure the consistency of the data?

I like to say Kafka does a great job of providing delivery guarantees, but they're not for free. You have to kind of know what knobs to turn, and that's the double edge of Kafka as a platform. There're a lot of knobs to turn, and they can be can kind of not intuitive. And one of the big things we have to do was not only teach people build the right abstractions, but also kind of onboard people the understanding how to build distributed spirit applications. But also kind of let them know that you can just point and produce and consume and get all the guarantees all the [inaudible 00:43:40]. There are some kind of mechanisms and knobs you need to turn to be able to do that.

So I think generally doing a better job of increasing the abstraction and the understanding of the platform needed to be used, and also giving less knobs to turn was a particular challenge that we learned. Actually, we're thinking a lot about how we might use Kafka proxies to kind of build the right abstractions out so that these teams have the ability to not think about some of those nuanced details or distributed processing application.

The beauty thing here is that going from a company that was heavily leveraged on HTTP and batch and then going to streaming, it kind of built the foundation first to do some pretty magical things. Two of them being the Robinhood clearing, which we did a blog post on, which was a pretty big feat, honestly, from a technology perspective. A lot of broken dealers take years and years to kind of develop that. And we built that out. And half the time, the last team that did it, which I think was Vanguard that did it. We didn't have the time. But that has kind of able to be able to be delivered because of the expertise that we built up on stream processing.

And then very similar, same story can be said. I think there is a similar engineering blog post related to the data lake. Thanks to the kind of the expertise that we had there. It really allowed for the data team to build a robust data lake that allowed them to kind of execute and answer a lot of questions about the data. I like to people we're a data-driven, product-driven engineering organization. So the ability – I remember in my previous job with GE. So before I was at

Robinhood, I was at GE. There were dreams of talking about building a data lake, and it never happened. And to come to Robinhood, build up the expertise and streaming and to have the data team kind of take those Lego blocks and build out a data lake is pretty magical and exciting. So that was a mouthful, Jeff. Hopefully that was at the right level and detail there. And I'm happy to double click anywhere there.

[00:45:34] JM: I'm quite sure the listeners are going to enjoy all of that. And I think we should do a follow-up show in the future, because there are so many things that I wanted to ask about that we didn't get to. I'm just going to close off with a simple question, which is what has been the hardest thing to build?

[00:45:49] JG: Oh wow! So I think from two verticals, I came to Robinhood and got the unique opportunity to solve two big problems. That was take the company from batch to streaming, and to redesign, the service-to-service communication layer. Honestly, Jeff, I didn't have a ton of expansion distributed systems. I remember, the very first time I got exposes to distributed systems was etcd at CoreOS, and this had to be 2014 at Strange Loop in St. Louis. The team was there and I got exposed to that, and I was like, "Wow!" I just kind of knew that this would be the future naïvely and somewhat optimistically. And reading about or distributed systems and the education you would always see on Twitter and read these blogs of like distributed systems fell in these almost magical ways that you never thought of. And yeah, they do. The more and more you use it, these kind of third cases. And I have developed this mental model where always asking how can these systems fell and how do we build resilient applications to ideally be resilient to them and recover without operational overhead. But ideally do that, but in the minimal, be able to alert when those edge cases occurs so that we can alert and kind of get people involved and get the right visibility.

So the hardest thing was just really going from 0 to 1 and understanding distributed systems and what it took to kind of manage them now not only in an academic sense, but actually in a production sense that would allow us to actually deliver this product to our customers. And then secondly, effectively doing the exact same thing. I was always kind of sheepish. I had one of the most senior engineers on the team, Arvin, was kind of expert in networking. And I always sheepishly not really understanding how DNS works or HTTP worded. And going from kind of knowing a million miles, 100,000 feet high level understanding of how the network happens. To being like, “Oh, no. I need to understand all the nuances of how DNS work.” And then understanding how DNS works inside of Robinhood. And then be able to design a system that would allow for the company to scale all while kind of dealing with imposter syndrome of not knowing either these two technologies.

And I think the hardest thing for me was scaling as an engineer. Going from 0 to 1 and then quickly taking these things I read and concepts I've learned and the productionalize them in in the way that really fit our Robinhood infrastructure in a way that could provide the delightful experience for our internal application owners, and then ultimately our customers. So those two things with a decent amount of imposter syndrome made that all challenges. But that’s kind of the beauty of scaling a high-growth startup and being in a kind of an environment that was extremely fertile in the sense that people would invest in me and allow me to grow with the company. I had the opportunity to grow with the company. And that took me across the engineering chasm from good to great engineer, and super excited about the experiences I had growing and delivering the value to the customer.

[00:48:50] JM: Well, that's a great place to close off, Jaren. I hope we can do round two sometime in the future.

[END]