EPISODE 567

[INTRODUCTION]

[0:00:00.3] JM: Stripe processes payments for thousands of businesses. A single payment could involve 10 different networked services. If a payment fails, engineers need to be able to diagnose what happened. The root cause could lie in any of those services. Distributed tracing is used to find the causes of failures and latency within network services.

In a distributed trace, each period of time associated with a request is recorded as a span. The spans can be connected together because they share a trace ID. The spans of a distributed trace are one element of observability. Other elements of observability include metrics and logs. Each of these components makes their way into services like Lightstep and Datadog. The path traveled by different elements of observability is called the observability pipeline. There can be lots of data engineering and service engineering associated with building an observability pipeline.

In an episode last year, Cory Watson explained how observability works at Stripe. In today's episode, Cory describes how observability is created and aggregated. It's a useful discussion for anyone working at a company that is figuring out how to instrument their systems for better monitoring. I think today's episode will be a little bit more technical than the last one I had with Cory Watson.

If you are looking for a higher level cultural discussion of observability, you should definitely check out the previous episode I did with Cory, because he talked about building a culture of observability. You can find that episode on software daily.com, or you can find it in our apps, our iOS and Android apps, which are in the iOS and Android App Store's. We've got tons of episodes on blockchains and business and distributed systems and tons of other topics if you take a look at those apps. All of those episodes are indexed, they're easily searchable.

If you want to become a paid subscriber to Software Engineering Daily, you can hear all of those episodes without ads. You can subscribe at softwaredaily.com. All of the code for those apps is open-source. If you're looking for an open-source community to be a part of, come check us out

at github.com/softwareengineeringdaily. We'd love to have you as part of our community, so you can come to softwaredaily.com or github.com/softwareengineeringdaily.
Now let's get on with this episode with Cory Watson.

[SPONSOR MESSAGE]

[0:02:45.4] JM: You're a successful developer and you couldn't have gotten to where you are without help in your education and career. Maybe you're thinking about ways to give back in the community where you live. The TEALS Program is looking for engineers from across the country to volunteer to teach computer science in high schools. Work with a computer science teacher in the classroom to bring development concepts to life through teamwork and determination.

Pay your success forward by volunteering with high school students in your area by encouraging them on the computer science path. You can make a difference. If you'd like to learn more about the Microsoft TEALS Program, or submit your volunteer application, go to tealsk12.org/sedaily. That's T-E-A-L-S-K-1-2.org/sedaily.

Thank you to the TEALS Program for being a sponsor of Software Engineering Daily.

[INTERVIEW]

[0:03:50.2] JM: Cory Watson, welcome back to Software Engineering Daily.

[0:03:52.5] CW: Thanks for having me.

[0:03:53.7] JM: You work on observability at Stripe. The last time we spoke was about a year ago. We did a high-level exploration of observability at the company. How was your work changed since that conversation that we had?

[0:04:08.0] CW: That's a good question. I thought you might ask that. I think I'd have maybe a more prepared answer, but I don't. I was going to wing that answer, because I thought that day of it'd be a good thing to think about. There are more people on the team now, which is

interesting. Our scope has increased. I think when we spoke last, I re-listened just yesterday to try to make sure I didn't duplicate any information.

One of the things I noticed that we mentioned was that we weren't doing distributed tracing in earnest. We had poked at it, but we haven't really dove in with both feet. Now we're doing that too, so we cover tracing, we cover logging, we cover all forms of metrics, all the point metric stuff and then we we're a lot of vendor management, like we use a lot of outside vendors for backing a lot of this stuff.

I guess the scope's increased, the number of people has increased and then there's a lot of – Stripe continues to grow, so we're continuing to support lots of folks. I guess it's pretty much all just bigger and better and faster, right?

[0:04:58.7] JM: What are some of those off-the-shelf vendor tools that you use at Stripe?

[0:05:03.4] CW: I'm trying to think if there's any outside of just service provider. We use Splunk, for logging we use Datadog and signal effects right now. We use Lightstep for tracing. We use a bunch of open source stuff, or we have over time, we have some of our own internal open source projects that we use and then also external open source projects. Mostly we use vendors on the back end of this stuff. I like for them to get woken up when the time series database breaks.

[0:05:30.9] JM: One thing I find interesting about the monitoring and logging space is that a lot of these vendors have products in all of these different spaces. They'll have a logging tool, they'll have a distributor tracing tool, they'll have a graphical interface that's very flexible, but there's – it's not a winner-take-all market. In fact, it seems to be an ever-expanding cardinality of different observability service providers. It seems like you don't go with one single tool. You buy multiple tools. Why is that? Why are there so many different tools for this and why don't you just centralize on one of them, one of the vendors?

[0:06:11.7] CW: Sure. Well, I want to call back to you using the word cardinality there, because that's a very observability problem word, so I'm glad you used it there, so kudos for that. I think the reason that there are so many is I think it's twofold; one, observability is broad and hard,

right? Monitoring and visualizing and understanding all the different tools that we've got running in all these places takes – I mean, different organizations and different technologies all require you to approach the problem slightly differently.
You may, if you're a organization that's got a lot of Windows boxes, then maybe you use some tool that's for that ecosystem, or if it's Linux then you're over here, or if you like high-frequency trading space, there's maybe some stuff that's normal for you. I think, aside from industry, the other problem is that all of this information collection is very fractured, because we've spent most of the time in this industry, like this is still very nascent. We forget sometimes how young computer engineering is, but the nature is that some people started with logs and we called that observability.

Then some people – well, maybe we didn't back then, but we do now. Then some people have things like StatsD that push metrics and some people built out – now we've got distributed tracing and I'm sure in a few years we'll have other things. Wven within these spaces you see things like networking agents that collect things raw at the at the protocol level and then you've got service meshes that get involved in the conversation, and then you've got just logs that you spit out of services.

I think each of those starting points make it easier for organisations to use the tools, right? If you and I at our new startup that makes coffee hammocks, or whatever dumb thing we would think to make, then we're going to probably have some way we started. We're going to start like, “Ah, let's just omit some logs. This is pretty small.” That shapes I think a lot of the choices that we make.

For us, you mentioned that we use a lot of different tools. I think that's a little bit weird. I think that I personally and then also my team are interested in trying to build from the best-of-breed in each of these areas. We haven't picked one tool to do everything. Also Stripe being a very security-conscious company, we also have a tendency to do some of the stuff on prim and run it in-house. We mix the two of those. That's why I think we've made a lot of different choices there.

[0:08:31.0] JM: The question of how you get all of this data from different instances to the right place where it can be observed, where it can be processed for observation, this is – well you refer to it as the observability pipeline; the matter of communicating the data to the right places

and processing it. We actually did a show about this with somebody from Wavefront. It was a pretty interesting show, because there's obviously a ton of infrastructure that goes into shuttling all of this logging and metrics and monitoring data.
It's not directly business data, so it's unclear whether you need to treat it with the same level of dedication and sensitivity and garbage collection that you would, for example customer transaction data. Maybe you could tell me, how do you define that term observability pipeline?

[0:09:30.9] CW: Yeah, you've made a great point by talking about how this is maybe the part that's the least glorious of what we're doing here shuttling all this data around, and it's not small. We process billions of data points a day. I think that the roots of this, like when you think about a push-based system like StatsD, a lot of those metrics are lossy and they're not meant, as you said, to represent every individual customer transaction.

You'll often see some other system that's got different right capabilities that's capable of doing durable synchronous writes, like a write ahead log or something, so that you can say I guarantee that we will record this transaction, because you don't want to use that data for billing data if you can't guarantee that it happened.

I think that as the capabilities of our tools expand and improve, part of what I think you're going to see is this data will I think trend toward being more thorough and being more reliable. I would to see us get to a world where observability data is the same data that the business runs off of. I don't think we're quite there yet, because the tools are not are not necessarily built that way. They're often considered to be fine to lose a little bit of data, or they're not maybe synchronous with the work that's being done, or they're not atomic with the work that's being done.

I think that in this case, we think of the pipeline as the aspiration that we should be able to send all of this data that is as rich as it can possibly be, so that we can use it to make decisions, because I think traditionally you said this isn't business data, but I would like it to be more so, because a lot of times the business data is what governs the success or failure of the transaction. It's one thing to say that a transaction happened, it's another thing to say that it happened correctly. I think that distinction is important and I hope that the pipelines get better over time.

[0:11:16.0] JM: You'd like to be able to look at a high level transaction. For example, a user makes a payment for something and you would to be able to dive into whatever level of granularity for that transaction that somebody would like. If they would want to debug it, they should be able to look at all the logs associated with. They should be able to look at the metrics that for any piece of infrastructure that might have been going wrong around the time of that transaction, they should be able to look at all the different spans that compose the distributed trace associated with the services that aggregated to that transaction. That would be your dream of observability.

[0:11:59.7] CW: That would so much be the dream. You stole what I was going to say, because I was going to say, “That's the dream, man. That's what we want.” That's what the engineer wants. At the end of the day, to me the reason that I do this work is because I've worked on a lot of different systems, e-commerce and social things, places like Twitter and here at Stripe.

At the end of the day, I'm a visual learner. Also, what we do is incredibly scary. Could you imagine? I mean, I'm sure you can, like you do this stuff too. At the end of the day you're like, “Man, there are so many people depending on the work that I do. How do we sleep other than hubris?” I think that for me, observability is about creating confidence. I think that one of the ways that you can ensure confidence with yourself or with your engineers is by making sure that as much of this data is available as it can possibly be.

We're probably going to talk about some of the stuff that we do specifically, but at the end of the day it's about having as rich a data set as possible for an engineer to be able to go and find it, because I think I said this in the last time you and I spoke that it's not possible to enumerate all of the ways that a system will fail. That's just you're going to run out of time. You're going to have to take so much energy and time to think of every way something will fail, that the best you can do is just make it so that you're logging as much data, capturing as much data as possible so that when something does fail you've got it all there and you feel confident that you'll be able to get to the root of the problem.

[0:13:20.6] JM: I'd love to get into a deeper discussion of observability and elements of it and eventually talk about Veneur, which is this project that you're working on at Stripe, that you’ve built at Stripe. Talking from just a level of these primitives of monitoring; so logs, metrics and

spans, these are different elements of monitoring and you may think that logs are the core of your observability, but spans are the core. Logs I think of as these raw long-form error messages and a developer has created a logging message that is emitted every time something happens in their logs, or maybe it's written into the library that they're using that just gets automatically logged and the log is the is the history, the long-form history of what has gone on in your application.

A span is a notion that I have mostly heard in the distributed tracing world, where a span defines the length of time that a distributed trace spends within a given service. A span for a given payment transaction – well a given payment transaction might have a tree of spans, where you have all of these different services that play a role within a distributed trace. It's not just a single line of different segments of time that aggregate to the entire distributed trace.

It's actually a tree, because you have oftentimes parallel asynchronous processes that are going down different trees of execution and then they're leading to a result that they're working on in parallel. You can imagine both spans and logs are important. I think that spans are oftentimes derived from logs, or you can correct me if I'm wrong about that. Actually that's probably not right. I think that spans are probably generated by agents on those services.

Anyway, this discussion about logs versus spans and which are important to observability and how they play into observability and how you came to the conclusion that spans are perhaps more fundamental to the aspect of observability. Why do you believe that? Why did you come to the conclusion that spans are the core of this observability process?

[0:16:00.4] CW: It's interesting, because I think when I – at a previous job when I worked on an observability team, we didn't have log indexing. We didn't have a centralized place to put logs. I sort of, I don't know, came of age in that environment and got used to the idea that metrics could probably represent most of what you needed. If you needed to go and debug an individual place where a box was broken, then you could always look at that box as individual logs. You could dial into the specific problem by looking at high level metrics.

When I joined Stripe, they were fairly heavily dependent on logs and they didn't have a lot of other aggregation that they could use. We had to reconcile how to marry all this data together,

right? You couldn't just say you can't use logs. We've got to use all these stuff together. The reason that I arrived at this is what you just described about spans is all very true. They're complex relationships between time and causality and provenance.

If you sort of, I don't know, look at them in a mirror, or if you look at them through, I don't know, a glass of water so that their shape changes, at the end of the day if we think about spans just in terms of the data that they contain, and we don't think about how information is propagated through the system and we don't think about them as causal and we don't think about them in terms of distributed systems, a trace is pretty easy to think of as just a log line, because a trace and a log both have the same major components.

A log line pretty much always starts with a timestamp. You say this happened now and it typically is happens after pattern, where if a log line is after the previous log line, then temporarily it happened after. It usually has some sigil in it, to say like this is a thing I'm doing right now. You may prefix it with database query, or service name, or something to that effect.

When you were talking about log lines at the beginning, I started to think to myself the way I like to describe log lines as developers talking to themselves, because they're like, “I'm just going to say that I did this right now.” If you've got a fancy ORM, maybe it logs query durations and stuff like that.

The way that I think about them is that the data is the same between them. When you think about a logline, it's got a timestamp, it's got an operation name. You can get its duration by subtracting its time, or subtracting the timestamp before from its timestamp. The elapsed time is the amount of time between the log lines and then then you've just got a problem of well how do we link them together so that we can see the causality to say that this call beget that call.

Pretty much everybody gets to the point where they're like, “You know what? We need to generate a unique ID at the beginning of a request for maybe support purposes,” so if there's an error you can say to the customer like, “Here is transaction ID ABC123,” and it bombed out if you want to reference that for us in a support ticket.

If you pass that unique ID throughout all your different systems in a header and then you start to log it, now you've basically created a tracing system, and that's the evolution that you're going to work your way through, because as your logging becomes more sophisticated, you will inevitably trend toward tracing. I think that predominantly due to convenience, we have created logging libraries that are minimal. They just say like, “Here's the time and here's some words.” Structured logging and all these other things, at their core they're all really just events that have occurred. We call them by different names, logs, or metrics, or spans predominantly because our technology is what we're imposing on our engineer. If the tools that we use at our company, like at our coffee hammock factory, or whatever I said earlier, turns out to be based on logging technology, then that's the interface we provide our engineers.

Then one day says, “You know what? These logs, they're expensive to store. Let's do pre-aggregated metrics.” Then we give them some different library. I don't know. A colleague of mine, Aditya Mukerjee, he pointed out one day that we're basically imposing a view on to our engineers. We're making them look at these logs, spans, metrics, because that's the way our back ends work.

As programmers, we a long time ago learned that you shouldn't impose the view on people, like we should be view agnostic. To me, they're all just the same thing. It's just an event or a span. I to call it a span because the span implies that you need all this data. You need start time, end time, operation, trace ID and what my parent was if that exists, so I call it a span.

[SPONSOR MESSAGE]

[0:20:35.4] JM: Users have come to expect real-time. They crave alerts that their payment is received. They crave little cars zooming around on the map. They crave locking their doors at home when they’re not at home. There is no need to reinvent the wheel when it comes to making your app real-time.

PubNub makes it simple, enabling you to build immersive and interactive experiences on the web, on mobile phones, embedded into hardware and any other device connected to the internet. With powerful APIs and a robust global infrastructure, you can stream geo-location data. You can send chat messages, you can turn on sprinklers, or you can rock your baby’s crib

when they start crying. PubNub literally powers IoT cribs. 70 SDKs for web, mobile, IoT and more means that you can start streaming data in real-time without a ton of compatibility headaches. No need to build your own SDKs from scratch.

Lastly, PubNub includes a ton of other real-time features beyond real-time messaging, like presence for online or offline detection and access manager to thwart trolls and hackers. Go to pubnub.com/sedaily to get started. They offer a generous Sandbox to you that's free forever until your app takes off, that is. Pubnub.com/sedaily, that's P-U-B-N-U-B.com/sedaily.

Thank you PubNub for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[0:22:18.7] JM: Do you use a – the high-level transaction ID for if a user – I mean, Stripe's core product is payments and every payment is a piece of Stripe's revenue, so that's the core of the business. I can imagine that it would make sense to have transaction IDs that are associated with any payment and then to just reuse that transaction ID as a distributed trace ID, or perhaps have just one mapping from a transaction ID to the trace ID and then you have that trace ID propagate to all of the data that's associated with that transaction, so debugging a transaction becomes a lot easier. Is that what you do?

[0:23:05.0] CW: We do, I think for business and customer reasons, they need something that's maybe – if I'm doing this just for myself, I'm going to generate some gigantic GUID, global unique identifier or something that's got a lot of letters and numbers and weird things. Whereas, if I want to share that with a customer, I'm probably going to use something a little more, I don't know, easy to think through or to copy and paste.

We, for the purposes of presentation, we don't have a – there's a one-to-one. I can probably say that if you generate a charge at Stripe, I can tell you that we have a unique trace ID that goes with that, but we don't expose it to the customer that way, because there's a difference in what they need from that output. Yeah, in theory we basically do that.

[0:23:47.3] JM: Okay, so just to give people a little bit more perspective and picture in their head for what this data looks like, that you're capturing at Stripe that you need to be able to assess in the event of an outage, or when you're just trying to capture all of the data that's associated with each transaction so that you can look into it and improve the infrastructure, improve the latency associated with requests. Maybe you could walk us through to some degree what happens on an infrastructure level, what your trying to capture, why that's hard to capture, and just the ways that your infrastructure is responding to a request for a simple payment, and then we'll get into some of the infrastructure solutions that you've built around capturing that data and assessing it more effectively.

[0:24:40.1] CW: Sure. Basically, you need to generate a unique ID at the earliest possible part of your infrastructure. The first time that you touch bytes that the user has sent you, you optimally want to have an identifier. In some cases, you may even want to generate it on the client side. If they're using a client library, then maybe you want to generate that ID. How we do that differs greatly depending on the type of transaction, but basically as soon as you touch Stripe's infrastructure, we've generated a unique ID.

Then where this gets – most of this is not Stripe-specific, like the details that we'll probably talk about soon or, but generally you want to get that identifier made and then you need to find a way to propagate it throughout the rest of your infrastructure. If you've got an HTTP-based system, it's pretty easy to do with headers, that you can just say like every time I make a call somewhere else, I want to pass that ID along.

Now as you pointed out, there's difficulty, like the other end of this transaction as you propagate it deeper and deeper into your infrastructure, everything that touches it needs to be aware of how to pull that ID out of the request and then propagate it through its own internal runtime. Let's say that I connect to Stripe and then Stripe goes to a database to find some information, maybe to authenticate, to determine whether or not you should be allowed to make this API call, then now I've got to pass that ID down through all this infrastructure.

Now each time that the infrastructure does something notable, like make a query to the database, or check a given flag, or a feature flag, or something like that, in each of these cases

we have to emit I just did this and this is how long it took. Since we also have that unique identifier with us, then we can emit that as well.

If we go back to the log analogy that we were using earlier, then we can just prefix every log line with here is the unique identifier that this transaction was a part of. Now too descend deeper each of these operations wants to generate its own ID that is unique, and then it passes basically its trace ID and its ID downstream, so that the next bit of the transaction can basically continue this forever, until you hit the very bottom of the infrastructure.

At every point, all these disparate systems need to understand a protocol for how to get this information out and then how to propagate it downstream. This can get a little complicated, because sometimes you're interacting with systems that you actually don't have a lot of control over. If I use a database like Cassandra, or MySQL, or something, those systems don't necessarily know how to do this. People downstream of us are not going to accept a patch that's do tracing Stripe's way, because that wouldn't work for anybody, right? By choosing off-the-shelf infrastructure, we've made that impossible.

Service meshes help with that a lot, because they often get involved in the transaction ahead of time and there are also efforts like open tracing, which is a way to standardize around this without necessarily tying yourself to a back-end. Long story short, they're basically these systems all across Stripe that are collecting this information, they need to be able to emit that information somewhere and then that information needs to be collected up again and stored somewhere, so that you, when you're trying to debug why our coffee hammock trend – I'm eventually going to forget what I said at the beginning and not remember that we're talking about coffee hammocks.

When we sell one of those coffee hammocks, we're going to need to debug why that thing blew up and you need to find a central place to get that, because I don't think you want to have to SSH to 20 different machines and guess as to what happens. Optimally we can bring all this together. I think that's all the challenges, unless I'm left – I'm sure I'm leaving something out, because it takes a lot of work to get all this stuff stitched together.

That's also I think why there's a lot of opportunity in the market for this and why there are so many vendors, because integration is hard, instrumentation is hard. I think it's the hardest part is just figuring out how to instrument and get all this information together in a way that works for all the different things that you do with your company.

[0:28:40.5] JM: That's what baffles me. If it's so hard, why would you want four of these different providers integrated with your system?

[0:28:50.5] CW: I think most people probably pick one. We choose to do it this way for a couple of reasons. I think one, it's all about giving the best experience that we can. We can, for example like I can use graphite to store data about my time series, but I can't store tracing data in graphite. It doesn't work that way. I could store that information in Jaeger, which is Uber's distributed tracing system. It doesn't store time series data. The nature of a lot of the off-the-shelf components is I have to do this, vendors are starting to consolidate them, because the exact reason that you specified.

If I've got a really great tracing product, it's inevitable that if someone else builds a reasonable tracing product and also offer something that I don't have, it's going to be a little more attractive for that organization to maybe want to make that switch. Back to your earlier question about seeing the breadth that a lot of different companies are offering, I think that's why.

[0:29:46.1] JM: What you touched on with your breakdown of how Stripe would want to log and maintain access to all of the data associated with a given user facing transaction so that you could have all this data for further analysis or debugging, this is essentially a problem of aggregating data from all of these different services that play a role in servicing a user request.

I think this is how you came upon the project of Veneur, which is a global aggregator for push-based metrics, so that term global aggregator I think of all of these different services that are associated with transactions basically aggregating their metrics into a single place, so that global aggregator. What's an example of a metric that you're trying to aggregate with this system of Veneur?

[0:30:47.5] CW: Got it. I think the most common one that speaks to everyone is I'm running an API as a service or whatever, some something that handles transactions. It is going to be extremely common for me to want to capture the duration of that API call, because everybody wants them to be as fast as possible. If I need to capture the overall total global view of a given API call, if I start – if you and I at coffeehammocks.whatever, .com, we're going to do this by the way you and I.

[0:31:19.6] JM: .biz.

[0:31:20.3] CW: .biz, even better. Yeah, maybe we can get our own TLD.coffeehammock or something, I don't know. I'm still trying to think of what that maybe is. I think it's a place where you drink coffee in a hammock, maybe it's a pop-up shop, I don't know. We're going to track these transactions and that's straightforward to do when you and I are small and the coffee hammock market is small. There's just one box maybe that we use to do all this work.

Over time, we're hopefully going to be wildly successful as we become titans of industry in our coffee hammock line. We're going to have maybe tens of thousands of hosts aspirationally. It suddenly gets a lot harder, like how do you aggregate the view from ten thousand different machines or something like that?

There are a lot of ways to do it. You could just write all this stuff out in your log files, or with your tracing data, or whatever and put them in a place and then collect them all and then look at them. But read time aggregation can also be slow, or expensive, because you're maybe storing gigabytes of data to be able to answer a straightforward question.

If you can ahead of time say to yourself, “I know I want to measure this and I want to collect it all in one place,” we can pre-aggregate that. What Veneur provides to us is in the case of something that we know ahead of time that we want to know, we can pre-aggregate that and then merge all of the different machines together and create one final view that allows us to say, emit a percentile, to say that the P50, the P75, the P90, the P999, these are the different values for each of those, so that we can monitor those individually. We do that for millions of metrics across the entire organization how.

[0:32:55.9] JM: How did you get started working on Veneur? What was the motivation? What was the set of problems that you realized were not solved by the providers that you were using at Stripe?

[0:33:10.0] CW: I think it's polite of you to assume that there was that a plan and that it wasn't just hubris. I have a pattern that when I look at a problem like that, like you described a vendor or an open source solution. Pretty much all – computer people, I guess all people really, but we're all a whiny bunch. If you follow a lot of people on Twitter, people are always just complaining about stuff.

One of the ways that I think is more constructive than just complaining about it, stopping and going why is this so hard? Given a shot at implementing it myself, 99 times out of a 100, the thing that I build becomes an abandoned Github repo, where I tried to write something and realize that it was actually super hard and I didn't know how to fix it.

With Veneur we thought, “Okay, we don't have in our current configuration, we didn't have a way to do global percentiles. We had to do host local percentiles.” For those following along at home that maybe aren't sure what this problem is, like if I want to measure the time that it takes to execute an API call, I want to be able to see that across all my infrastructure. I want to see all 150 million of my machines, or all five of them, whatever it is, I want to know holistically what that duration looks like.

I don't want to take five different percentiles from five different machines an average of them, because that's not really mathematically what I'm after here. It means something, but it doesn't mean what I want. What I really want to know is how many of my transactions take more than a second, or something like that.

I thought, “Well, I'll just try to write something that's like StatsD, but that process is a billion data packets a second or something like that.” It turns out it's actually hard, but I had a glimmer of something interesting and I brought it in to the office and shared it with the rest of the team and other people thought it was interesting too. We started by just looking at this as how do we do time series pre-aggregation, how do we work on that? How do we build it in such a way that it's scalable and we can use it across the infrastructure without creating a single point of failure, or

that one box that you have to scale up really big? In the end, we've built a lot more on top of that, but that was really the beginning.

[0:35:14.0] JM: You built a system that is push-based. Veneur is a global aggregator for push-based metrics. This discussion of push-based versus pull-based metrics is, I don't want to call it a debate. It's is more like a set of trade-offs, where from what I understand and you have some blog posts that at least one of which enumerates some of the trade-offs between these is with pull-based, so pull-based being the system that is aggregating the metrics is pulling them from the host machines.

With pull-based, you've got the host machines aggregating information that are – that is leading to a metric, or the host machine is even calculating the metric and waiting for the centralized pull- based system to pull those into the centralized system. With the push-based system, whenever a metric is ready for consumption by that centralized system, it's pushed out of the host machine. In both of these examples, you have information that is coming off of a host, like a micro service that's processing some segment of a transaction, or doing some minor fraud detection, or something like that; converting currency from one thing to another.

These different micro services are going to be generating metrics all the time and the question is are you pulling the metrics off of those hosts, or is the host pushing those metrics? They're simply trade-offs. From what I understood, the tradeoff really comes down to do you want to put the onus on the host, the micro service instance that is potentially aggregating all these information and going to have to flush it eventually, hopefully is waiting to be flushed by the pull-based system, or is just able to push whenever it has data to spare to the centralized system? Is that the core debate between the pull-based and the push-based sides of this conversation?

[0:37:21.9] CW: There's a metaphor for this that I like, which is around cash registers and brick-and-mortar stores. When the transaction occurs, the cash register could push that information back into a database in the back of the store. In that moment, we have a currently 100% up-to-date awareness of all the transaction balances. At the same time, what if that system in the back is unavailable due to maintenance, or something to that effect? We put ourselves in a pickle now, because we have to shut down everything.

We've all dealt with some customer service scenario, where somebody's like, “Sorry, the computers are down,” and they're unable to work, because the system operates in a mode where it has to speak downstream. Let’s not get into the eccentricities of credit card transactions, because clearly you have to do those online. Anyway, the other side is that if the cash register store is a local total, then the machine in the back can periodically ask the cash registers, “Hey, give me what you've got currently. Show me your transaction ledger.”

That tradeoff that you described, putting the onus on one system versus the other, there's a lot of good things about each one. Push-based systems tend to be very timely like, “Hey, I've got a new one, I got a new one, I got a new one.” You know it up to whatever resolution you operate at up to the millisecond.

If on the other side, you've got a pull-based system, a pull-based system includes a really interesting feature that when it goes to the transaction source, like to the cash register or to the host in question and says give me your information, it knows in that moment is the remote system up or down? Is it active? Is it alive? If you and I are sitting in the back, like Scrooge McDucking the piles of money we've made, we might not know if the cash registers are down or not based on a push system, because like, well maybe there's just nobody in-line right now.

If we're using a pull-based system, we'll be able to pull it and will go, “Hey, I'm here. Just haven't done any transactions lately.” That's one example of the trade-offs that you have is that centralization of effort, versus timeliness of data. There's also a lot of scaling issues. I'm biased a little bit towards pull systems, because when I was at Twitter we had a pull system and we had a lot of problems scaling it, but there are also modern systems like Prometheus that use pull and they do so very well. Again, I think you very succinctly pointed out that it's just a trade-off.

[0:39:35.8] JM: The trade-off that you can sacrifice sometimes with those pull-based systems, I think this is the motivation for Veneur. The trade-off is that the pull-based systems when you leave it to the hosts to aggregate some of these metrics, then you're going to lose some of the granularity of those metrics, because something like – assuming a host is going to generate an average of a number of latencies.

Let's say you have a hundred transactions that occur and on a host, and that host is just averaging the latencies across those hundred transactions, and then occasionally the centralized system is pulling that data and it's saying, “Oh, okay. We're just measuring the average latency over the last hundred transactions and you're losing granularity there, because you may have – that average maybe 50% at one extreme and 50% at the other extreme and could signify some dramatic problem in your system if you had the outliers. Is that the thrust of building a push-based system?

[0:40:43.8] CW: Yeah, so a push based system gets the benefit of as I said timeliness. It can react and we can then go on and say, “Hey, let's build a really great back-end store for putting in time series data. It's full fidelity, it doesn't pre-aggregate, it doesn't sample, it's stores every individual transaction in the raw form.

With the system like that, you would be able to do things like say, if you do – it's common for example to take a counter like the number of transactions that have occurred and average it out over a 1-second, 10-second, 30-second period. As you point, out that leaves you without the fidelity to say, Well, are we really busy on the front end of that 30 seconds, or really busy on the tail end of that 30 seconds? Because our systems tend to have that weirdness to them that you say like, “Oh, well if we distributed this better it would be more timely.”

In our system, so what Veneur allows us to do is we run a sidecar basically on every machine that receives those pushed-based metrics. In traditional push-based systems, the collector runs off the machine. It runs on some other system, sometimes a centralized system, sometimes a service tier, but we don't go over the network for any of this. We push locally.

This is also a boon for systems that don't have threads, who aren't able to in a background process manage the data structures and all the stuff that you have to do to be able to effectively count an average and all these other things, because when we keeping as you said, all that data could take up a lot of memory. It could maybe be inefficient, so our systems we prefer to do the sidecar as a child process that we use to basically do all the aggregation in a different place, but then I think what's innovative about what we do or what's novel at least is we run Veneur again a little further downstream and we take all of the metrics that we've done locally and then we push them down to the next instance of Veneur.

We don't send it the raw data points, but we also don't send it just the average of say 5 seconds, or 10 seconds. At least for the purposes of percentiles, we actually send it a histogram, a sketched histogram. We say like, here were the components of the histogram, merge this with other histograms representing the time elapsed. Then from that, we can generate one final percentile in a way that doesn't limit us to one machine, but I guess it sits in the middle of the trade-offs, because it doesn't have the same trade-offs that either of the other systems has. It represents a golden middle ground for the two.

[SPONSOR MESSAGE]

[0:43:20.8] JM: Azure Container Service simplifies the deployment, management and operations of Kubernetes. Eliminate the complicated planning and deployment of fully orchestrated containerized applications with Kubernetes.

You can quickly provision clusters to be up and running in no time, while simplifying your monitoring and cluster management through auto upgrades and a built-in operations console. Avoid being locked-in to any one vendor or resource. You can continue to work with the tools that you already know, so just helm and move applications to any Kubernetes deployment.

Integrate with your choice of container registry, including Azure container registry. Also, quickly and efficiently scale to maximize your resource utilization without having to take your applications offline. Isolate your application from infrastructure failures and transparently scale the underlying infrastructure to meet growing demands, all while increasing the security, reliability and availability of critical business workloads with Azure.

To learn more about Azure Container Service and other Azure services, as well as receive a free e-book by Brendan Burns, go to aka.ms/sedaily. Brendan Burns is the creator of Kubernetes and his e-book is about some of the distributed systems design lessons that he has learned building Kubernetes.

That e-book is available at aka.ms/sedaily.

[INTERVIEW CONTINUED]

[0:44:57.5] JM: Shed a little bit more light on the deployment model here, and maybe you could run through an example of once you have Veneur deployed, how these metrics are making their way off of hosts into the centralized metric aggregation system.

[0:45:16.3] CW: There's a blog post on my website, which maybe you'll be in the notes or something that you can point to, it's got some charts that illustrate some of this. The basic gist is that we run three different instances of Veneur. Let's say that we've got a server, or a few hundred servers, because our coffee hammock website has been so amazing and so many people want to be able to lounge in their backyard between two trees drinking coffee, that they – we had to run a thousand different servers.

On each of these servers, we run Veneur, which generally speaking is depending on what you're sending to it runs in a fairly small footprint of memory. We can send all of our transaction metric data into Veneur. Now at some point in the not-too-distant minutes, we're probably going to talk about how Veneur does more than just metrics; it can actually do tracing and a lot of other stuff.

We take that data, we push it into the local Veneur, where it usually hangs out for about 10 seconds. It's all configurable, but we hold it for about 10 seconds, because there's a trade-off between sending every piece of data and sending just the 10-second aggregates. We're able to save a lot of bandwidth and throughput and money by only sending metrics every 10 minutes.

Then we send it through the next layer, which is called the Veneur proxy. What the Veneur proxy does is it takes the payload of metrics, which is just a big pile, I think of it as a giant JSON document full of metrics. Each of those metrics is then chopped up and put into a bucket using a consistent hash to say this particular combination of metrics and tags, it goes to this global instance of Veneur.

This other metric and tag goes to this other global instance of Veneur. Then it takes those buckets and then it passes them down to the global Veneur instances who then aggregate them up and then ship them out after 10 seconds.

From this, we're able to create a situation where the failure of any Veneur proxy, they're completely stateless, means that we don't lose any metrics. If a global instance of Veneur fails, then because we service discover and monitor the hosts that Veneur proxy sends to, we only lose the metrics from that one 10-second window of aggregation. Another global instance can replace it immediately after, because we use a consistent hash ring to choose where we send it to.

Basically, we've got this pipeline of data going out that is consistently delivered, such that it's very easy for us to scale. We just add more hosts. The only thing that can make it messy is if there is one combination of tags and metrics that's very, very large, but because we're pre-aggregating it has to be – Usually the onus is on the local server for emitting too much information and they pay the price for that with usually CPU time or something like that. Generally speaking, it's very stable and it's worked really well for us and gives us a global view into everything without any single point of failure.

[0:48:04.6] JM: To drive this point home further, can you explain what problem Veneur solved in in the context of how did the system work before and after Veneur?

[0:48:19.8] CW: Yeah, so before at least for pre-aggregated metrics, there was an instance of something like StatsD that sat at the center, and it received packets from every single machine at Stripe and it was attempting to aggregate them. You can only send so many packets to a single machine before things start to break. There's also a law of diminishing returns for how much engineering effort you should spend on trying to do that, right? Because at some point it's going to be like, “Gosh, why don’t we just run two machines?”

The other problem that you have in the other direction is there are other vendors who implement something where they just run a StatsD instance on the localhost and then they ship it to their time series database directly, so it's a per host metric. That's fine, but then you've got to do a read time aggregate. It's very common. Let's go back to the very beginning of our chat when you mentioned cardinality. One of the biggest problems with all of this is that if I want to look at the sum of all transactions done in my store that's an O of N problem, right?

The more transactions you've done, the more time it's going to take to collect them all together, add them all up. It's even harder if you want to do something more complex than adding them up, like finding the percentiles or something like that so that you can – it's a common pattern to not look at the average, which can be misleading and instead look at the P50s, the P79, or the P70, the 75, 90, 99, all that other stuff, because those are more representative of how many people are having this experience. Because if the P99 is a good number, that means that 99% of all transactions you've practiced, or that you've run through the system are all being done in a good time.

Both of these systems had problems for us. Seeing the global percentile, or the global counters was difficult because we were having trouble scaling it. Then at the low-end, we weren't willing to look at this as you can only look at so many lines on a chart. If you look at five lines on a chart, it looks good. If you look at 500 lines on a chart, it turns into just a big brown smear, like when you try to eat nine different kinds of Skittle, you just end up with that weird brownish color.

At some point, the visual fidelity goes away and you don't want to look at that anymore. At a scale that many of these companies now operate, we don't want you to think about an individual host, like hosts are cheap and easy to replace. We replace them with containers, or pods, or whatever the kids are calling them these days. Our goal was to try to find a way to marry both worlds. We wanted the performance and we wanted our push-based metrics to be –

Oh, I'm sorry. There's actually another really good point, which is you can scale push-based metrics in something like StatsD by teaching your client library to distribute rights across multiple systems.One of the design things that we were really keen to do was avoid building smart clients. We didn't want our clients to have a bunch of routing and a bunch of consistent hashing built into them, because we can't get everyone at Stripe to patch their client libraries every afternoon. That's just not feasible. We thought it was better to control this at a service tier, which is how we operate Veneur.

[0:51:22.6] JM: How does this overlap with other changes in infrastructure that are going on – I've talked to different people building service meshes, where you have this additional layer, this is typically associated with Kubernetes, where people create a sidecar that is a service camera, or what it's called.

Anyway, service mesh is basically this agent that runs on all of your different services and it does handy things like circuit breaking and measurements and – it's like all of the things that you want associated with your service, including certain aspects of monitoring. Are there any overlaps between Veneur and the service mesh idea, or any other insights that you've taken from the whatever infrastructure refactoring is going on across Stripe right now?

[0:52:24.0] CW: I don't think that there's anything that's directly, like Veneur somehow. I don't think there's a lot of relationship between them, except to go back to something that I mentioned earlier. Service meshes are, there's many advantages to a service mesh. I've talked about both smart clients, which you now get to avoid, because service meshes deal with you say to the service mesh, I want to talk to an instance of MySQL right now. It figures that out for you.

You don't have to write all that into your application. You can just leverage the service mesh to do all of the things that a good client library should have done for you. This turns out to actually be a boon for us, because of instrumentation. The difficult part to me of any observability endeavor is getting all the disparate systems that you operate to emit similar information to the same place. It would be very easy if we had all unified around one format for everything, but we inevitably don't because we're always improving and growing and we're still in the early stages of doing this for as a profession.

In many of these cases, the service meshes are very beneficial, because they not only provide a common source of instrumentation. In fact, you could if you wanted to not emit any metrics out of your clients and instead just let the service mesh as it services the mesh and that's not really what I was going to say, but I could not say that. As its handling those requests, you want it to then say, “Okay, well cool. Since I'm the linchpin that holds all your stuff together, I'll just emit metrics from here.”

That's really helpful, because now you suddenly get information you didn't have before. Let's say that you've got an application that makes calls to some of the services that we've written, so we've now scaled our coffee hammock business so big that we need micro services. We maybe were really good about instrumenting our micro services, but maybe we never invested in instrumenting MySQL.

If you've got a service mesh, you now can intercept the calls to MySQL and be able to say, “I just made a call to MySQL. It came from here and it took this long. That's a big boon for us, because we now get even more information. If your systems already happen to be well instrumented, then it's still valuable information because the amount of time that it takes my machine to talk to the MySQL database, it's not instantaneous, because it inevitably takes some amount of time for light, or photons, or electrons to travel through those wires and for the protocols to be negotiated and for the routers to put things in places.

Having that extra little bit of visibility at the service mesh gives you the ability to detect things like network latency. Because now I know when my client made the call and I know how much time it took for it to traverse the network. Then so I get all that information. If anything, it makes our information more rich and more valuable, because we're now able to dive in at an even deeper level and ask our engineers to do less instrumentation on their side. They don't have to go in and be really fine-grained. They can start with a basic level of observability without having to invest in instrumenting their clients, which is huge.

[0:55:24.5] JM: One more question on Veneur. Just to wrap things up on that note, once you have this data that Veneur is aggregating, just to drive home the point of what we're actually doing with that data, can you explain what is the data that is being aggregated one more time and how it will be used for example in incident response, or debugging, or improving infrastructure. Explain it a little bit more detail what is this information that we're taking off of the hosts that we need to be thinking so much about push versus pull-based monitoring, or the garbage collection policy for this data, or how we associate this data with a given high-level transaction? Once again, what is this data used for?

[0:56:08.3] CW: Every piece of information that we can possibly collect about this transaction, so the amount of time spent where it came from, what calls it made to other services, we're collecting all of this and putting it somewhere. The thing that Veneur provides to us is the ability to take all that data and treat it like one sort of – back to the beginning of what you mentioned, this being a pipeline, all the data just gets put into the pipe.

Then what Veneur does with it, so we talked about aggregation. We talked about being able to take all of the different transactional information and generate one pre-aggregated value or what have you out of it. Extension of that is we can take that information now and then put it into different backing stores. We can put some of it into Kafka, for example and have that data be processed in real-time for circuit breakers to make sure that say the load test that we did this afternoon can be stopped if it has an adverse effect on latency, or success rates.

We can take that information about say the tracing, because we also collect tracing information through Veneur and we can look at that information and say what services were involved in this particular transaction and use that information to generate a cost analysis of how much it actually costs to service that API call for our coffee hammocks. Because we may be not charging people for it and realize that that API is actually really expensive, because it touches a bunch of different EC2 instances that cost a lot of money.

Or we may just be doing the boring thing, which is just making dashboards that show us that there have been a hundred transactions per second sustained for this period of time, or that suddenly the number of errors have spiked up. The reason that the pipeline concept is valuable to us is each of these things is coming from one place. We emit all this information from Veneur in the form of spans and metrics and also sometimes logs, and then we funnel that out to all these different systems, so that we can build observability products out of them that can then be consumed by other people.
We also generate very fine-grained dashboards for specific merchants who have specific SLAs around specific things, right? That's another use case. If your account manager, or your support people need to take a look and say why is this huge customer for coffee hammocks, like why are they having this trouble? Let's dial in and look at them specifically. We can service all of these requests from one place and Veneur provides us with an abstraction layer to make sure that we can write each one of them out and we don't have to think about it.

It also selfishly gives us leverage as a company, because we're not tied to a specific thing, like if you and I went to super awesome time series database and said we're going to use your product, we're stuck with it, right? Or if we're using a specific vendor for our tracing; we have a lot of leverage now, because we have the ability to take that pipeline and just tee it and push it

into another system and evaluate that system, and if it turns out to be better, we can make the switch. When it comes at contract renewal time, that leverage is really helpful, right?

[0:59:06.3] JM: That's pretty cool. It's in that sense a middleware where all of the data for your observability is going to sit at some point or another and you can just basically have a switch statement for whatever the piece of data is.

[0:59:22.3] CW: Yeah, it's observability middleware. I've not thought of it that way before, but it's basically exactly that. Optimally, we want you to emit just the highest, most fine-grained information, we want you to emit that span with all those tags and all that information that's so valuable to you, and then we'll distill out of that the other primitives. We'll condense it down and just make a point metric out of it, because all that span really said was we just processed a transaction.

We'll drop all the data we don't need and we'll just send a point metric down to graphite, or to Datadog, or to signal effects or whatever. Then we'll also take the spans and push those off into Jaeger, or Lightstep or something like that, or we may take the full fidelity information and take that and package it up and send it off to something like honeycomb or to Splunk and then be able to dig into it at a very fine-grained level in the future. We do all of that in one place and then it's used for everything you just described.

We'll use it for real-time alerting, we use it for non-real-time analysis, we use it for cost analysis, we use for circuit braking. Basically all these different places we use it, and if we find a better thing tomorrow that serves those needs, we'll just write a new back-end for – new sync for Veneur is what we call it and then we'll just shovel that data off there and then we'll have new capability and the folks at Stripe will suddenly be more confident and more powerful. That's what it's about at the end of the day.

[1:00:38.9] JM: Okay, great. I think we've covered the pipeline in a lot of detail and we've probably given people a good perspective for how this Veneur idea is useful and whether it could be useful to people. I think the idea is if not the open-source project itself will be useful. Let's conclude by zooming out a little bit, because you're working on a lot of different things at Stripe within the observability domain.

I'd like to conclude with just a two-pronged question, which is in your time and Stripe what have you – what kinds of practical monitoring advice have you accumulated in your brain and how has that led to how you advise different teams within Stripe to design their systems for observability?

[1:01:26.7] CW: It's a really good question, because I think that very often they don't think about designing something to be observed. This leads to what the observability word even means. It comes from control theory and it's about can you determine the internal state of a system by looking at its external outputs?

That cuts really to the quick of what people need to be doing, which is many years back, like I'm not saying that I ever did this, but other people probably did. You wrote programs and you didn't write unit tests. You just threw some code together, you ran it and you said, “Well, that made coffee hammocks work,” and then you just shipped it and you were done.

Then at some point we all learned that, you know what, it's actually useful in the long-term to write tests that verify that these systems are doing what we expect. I don't know about you, but when – well, I said I didn't do this, but I guess I did. When you design for testing, it changes your code, because suddenly there are abstractions that become a little more powerful, or places where an argument needs to be that it previously wasn't, or maybe you end up writing a more modular piece of code so that the underpinnings of it can be tested.

A good example of this is if you write it in Go, there's an underlying part of the HTTP library called a transport. The transport can be overridden when you create an HTTP client, which is very convenient in testing, because now you can hook in a harness that says when I do an HTTP call, I want to intercept it, and I want to do something with it.

If you want to have an observable system, you need to build it in. You need to be thinking about it from inception. So often, people come to me with a service and they're like, “I don't know what it's doing. I don't know how to observe this. Help me.” I'm like, “Well, what does your system do and are you measuring that?” In many cases they're not. They're trying to figure it out by looking at some of the lights and stuff on the outside.

If you want to know if your car is busted, then you can only get so much information from a tachometer and a speedometer, right? That's what those little lights are for and even that's not very helpful. It mostly just says take me in to get serviced, but then the tech will immediately connect in the little plug to the ODB port on your car and they'll pull all the statistical and diagnostic information out. Your programs need to work this way. You need to think of it as an API for visibility into your system. How can you reach in and get that information that governs the beating heart of what your service does?

Then after that's over, you should be able to – after you've designed some of that, you should be able to reduce what you're monitoring to a relatively small set of key performance indicators, or KPIs. The Google SRE book I think calls these the golden signals. There are a couple of different things, like you can look up the read method, or the use method. I'd try to remember exactly the names of everyone who came up with them, but I forget. I think it was Tom Wilkie did the read method and Brendan Greg is the use method.

The read method is requests, errors and durations. How many requests are you servicing? How many of them are errors? How long have they taken? That's all you really need to know, those three things, because you've probably got some internal SLA around each one of them. Those should be the top three charts on your dashboard. They should be the only things that you really page on, because if your system under the hood maybe has – maybe it's Go, or Java and has garbage collection, it doesn't make any sense to alert on garbage collection, because I don't know about you, but I don't want to get woken up at 3:00 in the morning because something is garbage collecting. I only want to get woken up if people aren't able to buy their coffee hammocks. I don't know how you feel about that, but that's the only reason I want to be woken up.

If that happens, then I can use the dashboard to figure out why, like the next highest level things on my dashboard should be common situations that occur, like maybe a high amount of garbage collection, or maybe we've run out of coffee, or there's no more yarn to weave the hammock. These things all need to be secondary. Too often, people start to enumerate everything that's ever broken, and they put that at the top of their dashboard.

It ends up being a archeological dig to say what broke most recently, versus what broke a long time ago. We said before, you can't enumerate all the methods of failure. If you base your system around, this you're less likely to be woken up in the middle of the night. If you are, you're at least being given something actionable, because I'm not getting enough transactions, or they're taking too long are all things that you need to know how to deal with.

That's the most advice I think we give is you need to think about this from inception, you need to be designing it in, you need to have these high-level KPIs, these golden signals need to be at the top of what you're monitoring and then you should be familiar with them, right? They're not something you should not know. You should know how all those systems work.

I mean, I think that's the same advice I'd give anyone. It's not specific to Stripe. I think the best advice wouldn't be, right? Wherever possible, you should be doing all of those things. I think you'll have a really good experience if that's the way you build stuff out.

[1:06:19.8] JM: All right, Cory Watson thanks for coming on Software Engineering Daily.

[1:06:22.3] CW: No problem, man. Thanks for having me again.

[END OF INTERVIEW]

[1:06:28.4] JM: GoCD is a continuous delivery tool created by ThoughtWorks. It’s open source and free to use and GoCD has all the features you need for continuous delivery. Model your deployment pipelines without installing any plugins. Use the value stream map to visualize your end-to-end workflow. If you use Kubernetes, GoCD is a natural fit to add continuous delivery to your project.

With GoCD running on Kubernetes, you define your build workflow and let GoCD provision and scale your infrastructure on the fly. GoCD agents use Kubernetes to scale as needed. Check out gocd.org/sedaily and learn about how you can get started. GoCD was built with the learnings of the ThoughtWorks engineering team, who have talked about building the product in previous episodes of Software Engineering Daily, and it’s great to see the continued progress on GoCD with the new Kubernetes integrations.

You can check it out for yourself at gocd.org/sedaily. Thank you so much to ThoughtWorks for being a long-time sponsor of Software Engineering Daily. We're proud to have ThoughtWorks and GoCD as sponsors of the show.

[END]