EPISODE 522

[INTRODUCTION]

[0:00:00.3] JM: Streaming architecture defines how large volumes of data make their way through an organization. Data is created at a user's smartphone, or on a sensor inside of a conveyor belt at a factor, or really anywhere. That data is sent to a set of backend services that aggregate the data, organizing it and making it available to business analysts, application developers and machine learning algorithms.

The velocity at which data is created has led to widespread use of the stream abstraction. A stream is a never-ending append-only array of data. To deal with this volume, streams need to be buffered, batched, cached, map reduced, machine learned and munched until they are in a state where they can provide value to the end user.

There are numerous ways that data can travel this path. In today's episode, we discuss the streaming systems, data lakes and data warehouses that can be used to build an architecture that makes use of streaming data.

Ted Dunning is a Chief Application Architect at MapR, and he joins the show to discuss the patterns that engineering teams are using to build modern streaming architectures. Full disclosure, MapR is a sponsor of Software Engineering Daily.

I want to mention our meetups. Meetups for Software Engineering Daily are being planned. Go to softwareengineeringdaily.com/meetup if you want to register for an upcoming meetup. In March, I'll be visiting Datadog in New York and HubSpot in Boston, in April I'll be at TeleSign in LA. We've got some of the talks planned already, but other ones are up in the air. We're not exactly sure what we're going to be doing at these meetups, but the past ones have all been pretty awesome, pretty good energy, great crowd. I think you'll like them. If you can make them, I'd love to see you at the Software Engineering Daily meetups.

We also have summer internship applications to Software Engineering Daily that are being accepted right now. If you're interested in working with us on the Software Engineering Daily open source project fulltime this summer, it's going to be a remote opportunity, send an application to internships@softwareengineeringdaily.com. We'd love to hear from you. We'd love to get your help on the open source project and you'll get paid for it if you're an intern.

Thanks for listening and I hope you like this episode.

[SPONSOR MESSAGE]

[0:02:31.9] JM: Apps today are built on a wide range of back ends, from traditional databases like PostgreSQL to MongoDB and Elasticsearch, to file systems like S3. When it comes to analytics, the diversity and scale of these formats makes delivering data science and BI workloads very challenging. Building data pipelines seems like a never-ending job, as each new analytical tool requires designing from scratch.

There's a new open source project called Dremio that is designed to simplify analytics on all these sources. It's also designed to handle some of the hard work, like scaling performance of analytical jobs. Dremio is the team behind Apache Arrow, a new standard for end-memory columnar data analytics.

Arrow has been adapted across dozens of projects, like Pandas, to improve the improve the performance of analytical workloads on CPUs and GPUs. It's free and open source. It's designed for everyone from your laptop, to clusters of over 1,000 nodes.

Check out Dremio today at dremio.com/sedaily. Dremio solved hard engineering problems to build their platform, and you can hear about how it works under the hood by checking out our interviews with Dremio's CTO Jacques Nadeau, as well as the CEO Tomer Shiran. At dremio.com/sedaily you can find all the necessary resources to get started with Dremio for free.

I'm really excited about Dremio. The shows we did about it were really technical and really interesting. If you like those episodes, or you like Dremio itself, be sure to tweet @dremiohq and let them know you heard about it from Software Engineering Daily.

Thanks again to Dremio and check it out at dremio.com/sedaily to learn more.

[INTERVIEW]

[0:04:32.8] JM: Ted Dunning is a Chief Application Architect at MapR. Ted, welcome to Software Engineering Daily.

[0:04:38.3] TD: Well, howdy?

[0:04:39.6] JM: We've done several shows recently about streaming data. For modern applications, it can make sense to build your logic around a central data stream. Today we're going to talk about that and we're also going to talk about ways to build machine learning systems around that central data stream. Why don't you start up by giving us an overview for a typical streaming architecture and what the different components of that architecture are?

[0:05:04.4] TD: Sure. There's often a reaction to the discussion; people say, "Oh, we had streaming back in the 90s." We definitely had a message passing systems. Then we had service-related architecture and so on, but each one of those became more and more complicated. The passing of messages became expensive, involving database commits, every time you consume or produce a message.

Performance always was an issue. There was also always an issue of schema and things like that and who had access, whether or not you could build a stream, because of the performance issues there was not a lot of ease of use and you didn't wind up with very pervasive streams, also and this is like a 1980s mindset that persist even to the present day. The idea was that we wanted to delete messages as soon as possible. A lot of the technical effort that impacted performance was due to that idea that we would delete things right away.

[0:06:08.9] JM: Because it was expensive to hold on to stuff?

[0:06:10.8] TD: Yeah. Yeah, absolutely. Especially it's expensive to hold onto stuff in the context of a system that is designed around trying to get performant transactions in place. 10,000 or a 100,000 messages per second was really considered high volume and that's really quite low if you look at all the things that are happening in a business.

The modern view of this is quite different. It's one we really, really, really want to isolate the producer and the consumer in some very fundamental ways. We want to hide the implementation details of the producer and the consumer. Even an implementation like – detail like, has the consumer even been written yet? Also, it is running in batch or continuously?

The idea of this independence and the simplicity of the transport is so that we can build these systems, which consist of independent little entities, things that can be updated independently and which communicate using streaming.

The point of that is that when you interact with the outside world, you often need to give a response and answer a question, render a page, approve a transaction, debit and account, all of those things. Something needs to be done right now and you need to give an answer back. Almost always, there's more work that would be better if you could defer it as well. You do the right now thing and then you say, "Later, I'm going to go through and I'll update the monthly statements," or, "Later, I'm going to go in and make sure that the inventory has been updated."

[0:07:57.8] JM: Lambda architecture.

[0:07:59.0] TD: Yeah. Well, not Lambda. Lambda is all about, "I'm going to fuck it up in real-time and then fix it in batch." This is real-time architecture, not Lambda architecture. Lambda is about an admission with the tools we had at the time, we couldn't build good real-time systems on large data. But frankly, we now can.

The idea that we have to fix it up later in batch really needs to be killed. We can do the right thing. If we're going to build these independent systems, they have to be independent. Otherwise, they're not independent. I mean, otherwise what happens is doing anything to one ripples out and the blast zone of any system changes expands without limit, then you have to get a lot of people in the room, and it's really hard to make any changes or any improvement.

If things are independent, if they can be deployed independently, then we could hide our dirty laundry and we can improve things without other people realizing we're improving, because providing the same service that we always did.

[0:09:10.6] JM: The tool for that independence for that decoupling is almost sure going to be Kafka, or something like Kafka, is that correct?

[0:09:19.0] TD: Yeah, but let's talk about why. We have the classic microservice that people talk about, which is almost always query response, real-time response sort of thing. There's another kind; the key definition of microservice is independently deployable. Independence like that implies that the producer of a message that streaming that message away can fire and forget once it sent the message and then acknowledge that the streaming system has it, it doesn't have to know how many consumers there are, whether the consumers are running right now, or even if they've been written.

That independence implies persistence. Now there's another thing we want and that is we don't want anybody to ever imagine that they need to work around the performance of that streaming system. We want them to say, "Absolutely. I can just send as much as I want. I want to send a million messages a second. It's no big deal."

In addition to persistence, we need performance. The room has to be wide enough that we can't touch the walls and tall enough that we can't touch the ceiling. A third property that we need is pervasiveness. 40 years ago, there were no networks. If you wanted to use a file and you'd have to have this committee meeting and you have to have budget authority, and you'd have to promise that there was somebody to call after hours if things broke and you had to have a contingency plan if your file grew.

Files are really a pain in the butt. They weren't really very pervasive, the idea of long-term storage like that was a difficult concept. Then later when we had networks initially, you had to have your own purpose-build network between a few machines and they were not pervasive.

With the advent of TCP and DNS and with the advent of POSIX and manics for files, both of those have become pervasive. Meaning, nobody even thinks twice about it in getting them, and we need streaming to be just as pervasive. The three properties we need are persistence, performance and pervasiveness. It has to be like they are – nobody imagines that you're going to build a data center without a network. It's just absurd. It needs to be absurd to build it without a super multi-tenant streaming capability as well. It's just as fundamental as files and databases.

[0:11:52.8] JM: Let me ask you a question about how things get on to this stream. Let's say I'm a user and I am changing my profile on a social network. Let's say, while I change the form, maybe I'm updating my job title and I click confirm and it sends a request to the server to do an update operation on my profile database entry. At some point, that is also going to – it's going to write to the database, the profile's database, but it's also going to create some sort of event at some point that will be put onto this Kafka event stream, assuming we're talking about some sort of stream-based architecture, so that other systems could potentially consume that event.

This could be something that – this is the same pattern that we would be doing if we were also doing some large scale logging or analytics, like if you had an IoT device that's sitting out in the wilderness somewhere gathering a bunch of data on soil and you're just sending large volumes of data to some backend service net back and services, throwing it onto this Kafka stream.

Can you give me an overview for how that transaction proceeds? Is the service that is calling original, or is the request calling directly to a service that writes to the Kafka stream, or is there some other sort of service that sits right on top of the Kafka stream and then that performs all the writes, like an event gateway or something like that? I just like to get an overview for the architecture of how you see people typically writing events to this Kafka stream.

[0:13:35.9] TD: Yeah. Let's listen, just so what you said for a moment there. Early on, you said right to the profile database and updates of profile. Now users update their profile every time they do anything, because it's very important in the moment to have some idea of what their intent is and what's happening. The last 20 pages of content that they've seen is important, because that helps you with recommendations.

The fact that they're coming from this particular IP address and it's a known one and so on is part of their profile. Now of course, the publicly visible aspects are also part of their profile. Those are subject to explicit change, but there's a lot of implicit aspects of the profile. Now you also said the profile database and imagine if you will, you have a master profile database. I say, "Wait, we need to change that over. We have to change the underlying technology. We have to switch it over to an end-memory database, or we had to put a caching layer on it, or we have to switch to using document database, or relational, or whatever we want."

There's a lot of different choices and a lot of reasons for making those choices, but making a wholesale change to a global database resource like that is liable to get you assassinated, because you're going to break something. The problem is the databases for different applications have different optimizations. The optimization could be the actual technology, could be the data model, could be whether or not indexes are created or not.

One person's optimization is another person's pessimization. This is inherent in the problem of databases, which are very flexible. Therefore, many of the operations are very expensive, and trying to make one operation cheap puts a tax on other users.

The assumption that there is a singular database there is a fundamental error in some sense and that it leads to coagulation of the entire development organization, because it puts dependencies between everybody, and so somebody will –

[0:16:02.7] JM: Sure. Well, even at a minimum, I mean the early conversations I had about Kafka when I did my first couple shows on Kafka, I talked to people from LinkedIn, which is where Kafka originated from, and the whole problem they were trying to solve was for example, a user is updating their profile and the user profile database needs to be updated. Also, you need to update the search index for example. You have multiple databases that you want to be updated in a way where they are consistent in a reasonable amount of time, a lot amount of time.

[0:16:34.6] TD: Well no. Actually, I think you want them consistent relative to a point in time, which may not be the present. Where that leads I think is that streams which have very flexible schema and I'm not going to say Kafka, I'm going to say Kafkaesque, because frankly we provide a more advance streaming platform than Kafka, but we think that Kafkaesque protocols and style of persistence, performance and pervasiveness are critical. We just think it's also really important that they natively span the glove that they have file names, that you should have a stream, it should be in a directory like a file list.

It's just another form of persistence. It has a different byte life cycle. Aside from that nomenclature and think of, do we assume it's Kafka, or do we admit that it should be just a general instance of these 3P types of persistence? I think that the persistence that we share is much more appropriately a stream than it is a database.

Databases are all about the definition of now and changes that happen in the now. Streams are all about being able to say, "Before this and after this." Before and after or much safer things to express in a global setting. Global means bigger than 30 centimeters in diameter.

[0:17:59.5] JM: Okay. I totally agree with –

[0:18:02.5] TD: That transaction should go to the stream first. The person making the change push a business event, not a database level effect, but a business level event into the stream.

[0:18:12.5] JM: Yeah. This is something that I see as crucially different between the way that people start writing their applications, like a typical CRUD application and how things work in an ideal streaming system, because if you're just building a CRUD application like you're building the first version of Airbnb and people are just posting their site list, their home listings, people are reserving homes and it's just a very transactional system at low volume, it's totally fine to just have a request response, you know you're hitting the ruby on rails thing that's backed by Mongo. You don't need this streaming abstraction.

Once you start to get to this large volume and you have multiple data stores, multiple consumers, then you should change things to where the requester is just creating events that get thrown onto this stream and you have subscribers that pull off of that streams, so you have this intermediate broker of information. Is that accurate?

[0:19:12.4] TD: Yeah, it is. Again, there's a huge assumption that that simple CRUD is the way that business is done. In fact, the real-world never worked that way and businesses actually never work that way. We have the example of Alice and Bob, Alice giving Bob a check for $5, and Alice's account being debited and Bob's account being credited. Isn't that wonderful if we had transactions that would be beautiful, but that doesn't happen that way in the real world.

In a real world, a check or an ACH debit goes to a clearinghouse, which is essentially a stream and Alice loses the money as soon as the transaction goes to the stream, they create clearinghouse in the sky and Bob gets the money when the transaction hits his account from that stream. That is exactly streaming.

[0:20:06.4] JM: You aren't suggesting that if Airbnb is just struggling to get their business off the ground, they should hook up this elaborate streaming system before they even get traction with their business, are you?

[0:20:17.3] TD: No. I don't think they should hook up at an elaborate streaming system. I think streaming should be pervasive. It should be as easy as creating a file and it should be distributed, which is part of pervasive. It should be the easy way to do this. Then if the web frontend wants to go all Mongo mode, cool. They should read the – they should be writing those updates to the stream and they should read them for their own database into the stream.

Now somebody just a little bit later says, "Damn, that was naïve guys. That is really a bad way to do this." They can say, "Oh, I'm going to uses MapR-DB, or I'm going to use Aerospike, or I'm going to use something else," and they could start reading that stream as well and they have their database.

Now the two databases I contend are exactly consistent, not consistent soonly. They're consistent, meaning that if they have read up to the same point in the stream, they will have exactly the same contents. That's the true meaning of consistency, because there is no now. You can view it as asymptotic once they pass the same point, then all the transaction will have been seen.

We cannot know for instance what is the temperature in Berlin right now. Right now, meaning 1 millisecond, because the speed of light is too slow, and we cannot know the state of the other side of the data center even.

[SPONSOR MESSAGE]

[0:21:46.2] 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]

[0:23:21.0] JM: I think what we can agree on is that the value of the stream abstraction, and this is something that was really confusing to me for a pretty long time is that you can almost view the stream, or you can view the stream as this append-only array like sequence, where any type of consumer that wants to read from that array of events can start at any place in that array, as long as that array has been persisted, or it's sitting in in-memory, or any other kind of available storage system.

You can just read from it and you can start at any place in time, you can read as few or as many events as you want and you're reading a stream, you're reading an array. That is in contrast to the way that I saw things early on when I heard the word streaming I was like, “Oh, okay. It’s like you’ve just got this stream that’s passing you by and you just stick your neck in and you just catch whatever comes by that appeals to you.”

A stream is not this transient thing. As you’ve said multiple times, you like to view streams as files where you can go to the beginning of a file, you can go to the end of a file, you can process it however you want.

[0:24:37.8] TD: I don’t view them as files. Files have one byte life cycle. Databases have another; they’re record-oriented, they’re mutable very nicely in the middle. Files are mutable, but they’re only by byte offsets. To delete a file, you can delete the middle of it by writing nulls, but you may or may not delete things. You can basically just delete the entire file.

Databases like you do all kinds of wonderful record-level mutability, delete records when it’s time and then delete the database at the end of the life cycle. Streams are record-oriented as well, but they only allow you to append and then they magically evaporate from the front. They just have a different life cycle, but there are three forms of these things. They are different. Database is not a file, it could be implemented on top of a file, but it isn’t that the database mentality and life cycle of a byte is different than the life cycle of a byte and a file.

A stream is the third form, fundamental form of that. They should be co-equal within a file system. You should be able to make a directory with a config file and a stream in it, or whatever you like, or a database within a directory. You and I should be able to run the same application on this **[0:25:52.0]** data fabric. You have your directory, I have my directory, we’re on the same application and we against our private database say and read from the same stream.

I think you’re exactly right in that streams are not a moving river, which when past us we can never relive. It’s not that poetical –

[0:26:15.4] JM: That’s a common confusion, right?

[0:26:17.2] TD: It really is. Partly, it’s common because it has never before been plausible to make the stream be the system of record. It was never plausible because storing them for a long time was never practical. Even with Kafka, it has [inaudible 0:26:33.8] because of technical reasons. It really should be the moving hand having writes on, but leaves the ink on the paper. It's a brilliant thing.

You could be able to make snapshots, have a snapshot service so somebody doesn't have to read all from the beginning of time. They can start with a snapshot and read forward. There's a lot of things you may want to do there, but it makes a better system of record than a database.

We were talking before we started about machine learning and I said, "One of the key problems of machine learning is what did you know and when did you know it?" To build it on the systems that's going to make some decision, you have to know what did you know just before the decision point. The decision points are different for every different account or in the date that you're deciding about, and so you need to be able to know exactly what did you know at all these different moments in time and to be able to build snapshot of your knowledge then. A stream makes that not dead simple, but tractable. A database, because of mutability and files because of mutability make that intractable. You have to approximate and guess a lot.

[0:27:50.8] JM: If there's one thing that I am hoping people take away from the series of shows about streaming that we've been doing, it's I think that the stream or the queue, or the pub sub system, whatever you want to call this component of your architecture is really important, and it turns things on a new perspective than if you're just thinking about having your – like the batch world, where you just got all your files in HDFS and you've got to do this really heavy protracted operation to pull all your HDFS files into memory and perform large operations on them and then you have a result. Then if you want to do another result – if you want to do another computation there, it's going to be another big protracted operation.

The fact that we have this stream, this big data stream and it's not cost-prohibitive really changes the kind of processing you have access to because of that data availability and we can certainly get into the processing. We've done some shows about Flink and Spark recently and these are the processing systems that you would do on top of this long data stream. Really the emphasis that I want us to have here is that you have access to this data stream now and it changes how you're going to build systems around that stream.

[0:29:13.2] JM: Absolutely. Absolutely. Frankly, files naturally lead to patch. That way lies madness and Lambda. Life doesn't happen in batches as a friend of mine says, and it is not a good model of the real world. Streams are the way the world happens. That's a much more natural way to do that.

That doesn't mean that files are not useful. It doesn't mean that batch is not useful. If I have some training set of data and I want to build a model, I want to use that training set and I want to freeze it so that I can do reference builds and so on, I want that to be batch. Files are totally natural and appropriate there.
If I want to model the real-world and things happening in the real world and be able to process them in a natural way, then streams are the natural construct. I don't think that one has ultimate primacy over the other, they are both valid and very, very useful. You should have a directory with both things in it. It should be natural and it should be distributed, so that I don't care what machine I'm running on.

Those perform the function of transport of data from place to place, from program to program and so on and then things like Flink and Spark and so on and confusing Kafka streams, provide the processing aspect. These are the dual nature of computing is process and transport; munching the data and moving it to the munchers and from the munchers.

[0:30:52.9] JM: At MapR, you have looked at Kafka and said that there were some tweaks that you wanted to make to this Kafka-like system and you guys ended up developing something called MapR streams. What were the designed decisions that led to the creation of that system?

[0:31:11.5] TD: Well, the process was quite simple. It was immediately apparent. The Kafka's pretty radical designed decisions were incredibly right for the modern world. Eras in time in engineering design are often described by ratios. With disks for instance, the ratio size to speed has radically changed over time and that has really radically changed how we think of computing.

1980 something, the Fujitsu Eagle is 300 megabytes, and you could read about 2 megabytes per second off of it. That means the size of the disk is perhaps best expressed as a 150 seconds, because that's how long it took to spill the disk or fill the disk. A modern disk drive could be 8 terabytes and a 100 megabytes per second. It's gone up massively in size and pretty substantially in speed, but it is now 80,000 seconds in size.

Before it was quite plausible that is rewrite the entire disk is now inconceivable. You cannot have a reasonable program rewrite an entire disk if you imagine it's going to be responsive at all, because it will take a day. It will take a day at best. If you put any randomness into the writing, it will take months, not a single day.

The design ratio there, the size of the disk expressed in seconds has fundamentally changed what we can do. Kafkaesque ideas of a very persistent stream that is very stupid happily and triumphantly stupid and that it does not have execution list, it does not have triggers, it does not have all of these stuff build into it. All it does is reliably let you send a message and then consume it.

Basically, the producer cares only about producing and waiting for an acknowledgment and the consumer keeps track of where it is. That degradation of the mission fit perfectly with this change in disk size from 150 seconds to 80 kiloseconds. That's a big, big fundamental difference. We started with that. That was brilliant. That's good. It really needs more than that, because people need simplicity in their lives where there are things that don't concern them very much. Security.

You've got permissions on files. You got permissions on directories. In the MapR system, you got volumes that an administrator can put bounds on the security. Why do streams not have exactly the same permissions and user IDs and so on as files? They are just a byte life cycle that's slightly different. They are just a performer persistence, which is very useful. They should be co-equal. We should factor a way to question the permissions and authentication from the idea of the life cycle. Those are orthogonal components.

Another one is where is it? It should not reside on any single machine. It should magically be distributed to where it needs to be. There should be no user actions if some machine fails. The system should deal with that, like any good data platform would. In fact, that should be handled for files and tables and streams as a single fundamental capability.

You have these cross-cutting concerns that really are important to have. That's where we saw – we saw immediately that Kafkaesque, it was designed lacked all of these. It had the insight streaming in a modern sense is far better than streaming in an older sense and is a much better way to build things, especially large systems. It's a great way to communicate between small systems in order to build a large system, but it fundamentally lacked those other capabilities. Instead of trying to come up with a minimum viable product, we said, "Where should it be?" We looked ahead and build a system that is where it needs to be for the future.

[0:35:35.8] JM: Now that we've discussed in detail the Kafkaesque system that you're going to be writing all of your events to, we can talk a little bit about streaming and some of the opportunities for – I'm sorry, stream processing. Some of the opportunities for stream processing and how that eventually fits into machine learning. I talked to your colleague, Tug and much of what we discussed was the idea of you've got the stream and your writing events to it, and oftentimes these events are fairly raw, like we talked about the example of you're just writing all – we give the example of Fitbit, where you're just writing a bunch of data points that are GPS coordinates, timestamps and user ID, so that you can get the velocity and acceleration of a user through time, because you can get their different coordinates at different times.

You might be writing those raw events to the stream and then you might have some stream processing system, like Apache Flink or Apache Apex, or Spark and you might be using that stream processing system to process the high volumes of events and do some simple enriching. For example, you could take the GPS coordinates and aggregate them into velocity and acceleration. You could write the velocity and acceleration back onto the Kafka stream. That was the basic example of using the stream processing systems to create and then to write enriched data back onto this giant stream that you have.

What are some other patterns around using stream processing together with that giant stream that you can have, that you've seen be particularly useful?

[0:37:22.7] TD: Yeah. I think that if you listened with your head sideways what you said, you can actually say that you describe business as enriched data. I think that's a very viable way to look at it. The enrichment doesn't have to be simplistic or minor. It could be massive. It could be the thing that adds value to everything.

[0:37:46.5] JM: Right. Just to interrupt and just to take your point a little bit further, you could say if you're running an e-commerce website, you could consider enrichment the gathering of all of the transactions that took place across your e-commerce site and finding the most popular items that were purchased for example, which is totally going to drive how you change your business.

[0:38:07.3] TD: You could view enrichment as the causing of an item to be shipped to my house.

[0:38:12.2] JM: Right.

[0:38:13.0] TD: That is an enrichment of data; the data being what is the location of the thing. It is now my house, and that's cool because I wanted that thing. All of business ultimately now becomes a data operation and that's an exciting step. Many of these data operations, many of these business actions involve decisions. At the rate that an amalgamated business runs, you know there is benefit to scale here, you cannot feasibly have humans make those decisions often just due to speed.

Also often, it's really valuable to have vaguely human level of correct decision-making and complexity of decision-making, but a human would just cost too much even if you were to exploit them mercilessly, they would just cost too much. We need to upscale the human into strategic sorts of decisions and tactical decisions that need to be made in the microsecond have to be made by machine.

That leads us immediately to machine learning. As I like to say, machine learning is just a different way of programming. It's a way of programming by example from data, but it's just a way of programming. Now unfortunately, that way of programming is not susceptible to a lot of the software engineering that we've invented over the many years.

The idea that you know what you intend to do, that you have a specification and so on that you can somehow validate that you have done what you intended to do is much harder when you don't know what you're going to do. If you don't know what fraud looks like, how can you write a specification for how to find it? How can you write a unit test for how to find it? Well you can't.

You have to reduce yourself to some statistical notion of it works better on average than before. Instead of unit testing, you have to run multiple of these systems at the same time. Machine learning is exciting and great and amazing what you can do some of these things, but it also is limiting and that we have to start thinking about how do we run multiple versions of the same program at the same time in order to compare them in real-time.

[0:40:43.8] JM: You're saying, because you have to compare different versions of a model to one another in order to figure out which direction to go in?

[0:40:50.3] TD: Yeah. Also because we have a bit of a divergence here. The people who build these things who like to be styled as data scientist, because scientist is such a cool thing, you imagine you get purple robes or a white lab coat when you become a data scientist. By nature, because of the even now complexity of doing machine learning, you have to focus on that pretty hard.

Your concerns become very focused around how to build that machine learning system around accuracy. Necessarily when you focus like that, concerns of reliability and security and so on wind up over there out of your frame of view. Now that isn't acceptable for the focus of the entire team to only be on accuracy and not on reliability and acceptable speed and so on.

That needs to be a different part of the team, and we've got now a new kind of team. It's no longer just dev ops. It becomes data ops. Dev is of course in there, but it becomes part of the data mission. We need separates in concerns; the data person needs to think data and accuracy, but the ops person needs to think reliability. Did it made SLAs? Can I guarantee it made SLAs?

[SPONSOR MESSAGE]

[0:42:19.6] JM: If you enjoy Software Engineering Daily, consider becoming a paid subscriber. Subscribers get access to premium episodes, as well as ad free content. The premium episodes will be released roughly once a month and they'll be similar to our recent episode, "The Gravity of Kubernetes." If you like that format, subscribe to hear more in the future.

We've also taken all 650 plus of our episodes. We've copied them and removed the ads, so paid subscribers will not hear advertisements if they listen to the podcast using the Software Engineering Daily app.

To support the show through your subscription, go to softwaredaily.com and click subscribe. Softwaredaily.com is our new platform that the community has been building in the open. We'd love for you to check it out, even if you're not subscribing. If you are subscribing, you can also listen to premium content and the ad free episodes on the website if you're a paid subscriber. You can use softwaredaily.com for that.

Whether you pay to subscribe or not, just by listening to the show you are supporting us. You can also tweet about us, or write about us on Facebook or something. That's also additional ways to support us. Again, the minimal amount of supporting us by just listening is just fine with us. Thank you.

[INTERVIEW CONTINUED]

[0:43:54.3] JM: I'm with you that there are changes we need to make to team structures in this environment. I would really like –

[0:44:02.7] TD: We got machine structures as well. The programming structure means to reflect that separation of concerns. We can't have everybody worry about everything all the time.

[0:44:14.5] JM: Okay. Let's take an example, like the whole Fitbit thing. Let's say we want to build a model for where people are walking in a given day, with the same GPS coordinate stuff that we've been talking about, the simple straightforward enrichment where the basics of the enrichment process where we're just calculating velocity and acceleration from the GPS coordinates and timestamps of different users, that's pretty straightforward. If we were trying to build – if we were trying –

[0:44:45.2] TD: It already is not. It already is not, because that data is noisy. If you try to present that data back to the user they would go, "This is shit. This is horrible." Already you need just to –

[0:44:56.2] JM: Okay. Let me simplify it even further. Let me simplify it even further. Let's say we periodically try to calculate the acceleration and the velocity of a user and then we send that to the user, and we ask the user, "Is this your velocity and acceleration? Is this a reasonable velocity and acceleration?" Then you can click yes or no. That gives a very simple way that we are building a "machine learning model," and we have a feedback system where the users can rate how good we're doing with the acceleration and velocity calculations, and then we can update our models, we can AB test them respectively. Help me understand how we are going to implement that system, how we're going to deploy it.

[0:45:37.7] TD: Well, as I see it, there's a couple things that are going to happen. We're going to have to run multiple versions of the model. Fine. We also are going to have to record the data that the models saw at a particular moment that they were asked to make a decision in order to be able to improve those models and build more versions of the models.

We're going to need a recorder. I call it a decoy. It looks like a model. It gets inputs just like a model, but it doesn't actually quack like a duck. It just records the data. We're going to need canaries, which are long-term stable things that we understand, so that we can compare new challengers to that venerable canary.

We heard and want all of the models to get exactly the same input at pretty much exactly the same time. Well, streaming is natural for that because the stream can have many readers. It could have a decoy, canary and five versions of the model all reading the same variables as input. That would be excellent, because then the models that we're comparing get exactly the same input and the decoy that's recording things records exactly what we knew at the moment.

How do we know it's what they got? Because that's what all the models got, because they all read from the same stream. Then we need something to decide which answer to return, and I call that a rendezvous server. Rendezvous meaning that the incoming request is read by the rendezvous server, and then results that start popping out of the different models into the stream are rendezvous'd with that original request, and some decisioning process; reliability, desirability, accuracy tradeoff is made.

We say for instance, "Here is the model that we think is the most accurate," but we aren't totally sure that it's quick or reliable. We'll wait 30 milliseconds for it to answer. At 30 milliseconds, we're going to go and say either the preferred model or the one we had last week are what we'll accept up to 40 milliseconds. We have a 50-millisecond deadline looming and so now we will take a very, very simple but reliable model as one of the acceptable answers.

Whatever we get first now, we're going to take. At 45 milliseconds, we panic and we just say, "If we don't have an answer yet, we're just going to give default no, or out of range or something, some default answer, so that we can guarantee that we meet our SLA with our best estimate. The rendezvous server separates the concern of reliability from accuracy and allows models to be deployed with I wouldn't say little regard, but far less regard, far less due diligence that necessary.

[0:48:32.4] JM: How are you tracking – in that model, in that architecture where you have a set of different – if I understood correctly, you've got a set of different models that are trained on the same set of data, or they can be trained on slightly different sets of data, but you've got a set of models that could conceivably respond to a user request and you might query one of them and if it doesn't respond in the right amount of time, maybe you query a different one of them and –

[0:49:01.0] TD: No, no. I would query all of them simultaneously.

[0:49:03.6] JM: Query all of them. Okay.

[0:49:05.4] TD: Then I would look for my preferred one to answer. If it hasn't answered in time, I will accept anybody else's answer.

[0:49:14.6] JM: Right. Okay. Then are you doing some kind of – some ID, like are you keeping an ID of each model so you know that – so you can keep track of which models served which request so that you can essentially have a testing system, because you want to be able to use –

[0:49:31.1] TD: Absolutely. All of them should be writing their results to the same stream so I can easily compare their results on exactly the same queries. There should also be an ID on every request, which is basically a return address, which is a way of knowing where to send the result back to.

[0:49:52.5] JM: Do you see this type of rendezvous model deployed in any of the clients that you've worked with or people you've talked to?

[0:50:00.6] TD: I don't see the whole thing, but I do see pieces of it. I do see people deploying multiple models and using a 100% speculative execution. Meaning, they evaluate all the models. I do see wide acceptance of the thesis that you have to have multiple models and you have to have multiple model development frameworks.

It is no longer acceptable to say, "We use SAS." That's the end of the story. If you're going to make lots of different kinds of models, different kinds of systems excel at different scales and technologies. H2O is really, really good for some models, XG boost is awesome at tree models at a very large size, is really good for simple models of small scale, or not so simple models, but it doesn't build on really large scale very well. Tensorflow of course is wonderful, Café is wonder, MXnet is wonderful. They each have different tradeoffs.

That's good. You should be able to deploy any kind of the model to production very easily. A little bit of containerization and streaming later, you're good to go that way. I see ubiquitous multi-framework development. I see lots of people deploying fewer frameworks than they develop in. I see everybody saying, "You have to have multiple versions of the model." This full-on rendezvous architecture where they just bite the bullet and go for it, I haven't seen yet.

[0:51:36.3] JM: Let's take something like Tensorflow. If you have Tensorflow and you want to train a Tensorflow model based off of the stream, this append-only stream of data that you have for let's say again the GPS example. If you wanted to train a Tensorflow model based off that and then you wanted to have that Tensorflow model be periodically updated, what does that look like? Does Tensorflow itself read off of the stream, or do you need some stream processing system that gathers batches of the data and loads it into Tensorflow? How does that work?

[0:52:15.1] TD: Well, you need some scaffolding in both sides, both the training and the deployment side. On the training side, on the Fitbit idea, we've got measurements that were marked as a user, "Well, that was just crazy. I wasn't in Oakland. I was in San Francisco." There were just some error there. You got good and bad measurements in history in that stream. You need to collate them and say, "For this measurement and this context, that was good or bad and we're going to build a model that decides whether they're good or bad."

We need to go through history and find a cortal snapshot, a snapshot of different moments in time, different inputs and different decision outcomes subsequent to that, so that we have training data. The training data probably be in a file. If the training data is really large, the ability to snapshot the files and so on so that you can manage versions. If it's small, you might put it into Git or something like that, but that's going to be file like and you'll do Tensorflow training just like you ever have done Tensorflow training.

Then for deployment, Tensorflow natively doesn't read streams. You'll need a little bit of scaffolding around it and the simplest would be like a Python script that says, "Give me a record from the stream, parse out the different features and call a Python method which evaluates the Tensorflow model," and then gets the result back and writes that one record out to its output.

You could batch them up and if it's a GPU sort of thing, or if it's something like an ad targeting system, or you might have thousands of queries, model evaluations for a single placement, then yeah, batching them might improve the performance, because the performance of any kind of these models eventually comes down to something very much like matrix qualification and matrix times vector is not nearly as efficient as matrix times matrix, purely due to memory caching effects.

You might want to batch them, but I would never do that as a first round. I'll build a simple thing first and then I would deploy the fancy thing as a challenger, and I would mark how often did it meet our SLAs, how often did it meet our stability requirements, how many times did the safety net catch it.

[0:54:45.8] JM: Okay. I know your time is short. I want to close off by zooming out and talking about this at the business level. We've done a lot of shows about enterprise rebirth. Basically, you've got companies like insurance companies and oil companies and consumer package good companies where they are realizing that they are software companies that produce consumer package goods, or they're software companies that produce insurance. They are doubling down on the technological processes that they have in place. They're updating their architecture. They're bringing in consultants and trying to figure out how to become a technology company.

A lot of the shows we've done around this have focused on the microservices or dev ops rebirth thing. Equally important is probably the data streaming, the data lake, the building up the core competency to be able to do machine learning type of stuff. All of that is pretty important. My question to you is if for one of these companies, like an understaffed in terms of technology resources, insurance company for example, where do they start when they have an architectural system and they're just looking to start to gain a toe hold where they can maybe look at in to two years or three years really having this streaming type of architecture to take advantage of?

[0:56:15.1] TD: The key need is to take a look at the business and find places where there are bottlenecks. But not just bottlenecks that slow down process, but bottlenecks that slow down process and impact value, and bottlenecks that are susceptible to better information handling. I wouldn't call them software companies. I would call them information companies. In fact, that blew my mind eight years ago I was sitting down with the number three guy at a really big bank, consumer credit card company largely and he described their company not as a financial institution, but as a data company. The flat out statement there. I was thinking, "Well, yeah." It's just stunning to hear an executive saying that.

We need to look at the entire activity of the business to more or less granularity and we have to view it as an information flow. Then we have to see where does that information flow work and where does it not work? Where are the problems with it and where are the problems that would be susceptible to these new technologies making it better?

We have a huge medical insurance company. One of their problems was that it took a human a lot of skill and a lot of judgment to look at provider claims and to figure out which ones are within process and more importantly which ones are actually just out and fraud. There's not a lot of that among medical providers, among real medical providers, but there are a number of bad actors who are happy to pretend to be that. They don't have to be very many of those be having a significant financial impact.

It was hanging up the entire business to try to deal with the bad actions of a tiny minority. By automating the targeting of that human attention, most of the business was released from that critical step and that's a huge thing. It's hundreds of millions of dollars, that's multiple thousand percent ROI in the first year for what started as a very basic information flow, improvements and very simple machine learning models intended to just make the flow work better.

The core thing here is to look at what makes the business really go and look at where the business really drives value. Then as every business opportunity has ever been, look for where you have leverage that can make a difference and value.

[0:59:03.9] JM: Okay. Well, Ted Dunning thank you for coming on Software Engineering Daily. It's been great talking to you.

[0:59:08.1] TD: It's been great talking to you too.

[END OF INTERVIEW]

[0:59:13.3] JM: If you are building a product for software engineers, or you are hiring software engineers, Software Engineering Daily is accepting sponsorships for 2018. Send me an e-mail jeff@softwareengineeringdaily.com if you're interested.

With 23,000 people listening Monday through Friday and the content being fairly selective for a technical listener, Software Engineering Daily is a great way to reach top engineers. I know that the listeners of Software Engineering Daily are great engineers, because I talk to them all the time. I hear from CTOs, CEOs, Directors of engineering who listen to the show regularly. I also hear about many newer, hungry software engineers who are looking to level up quickly and prove themselves.

To find out more about sponsoring the show, you can send me an e-mail or tell your marketing director to send me an e-mail jeff@softwareengineeringdaily.com. If you're a listener to the show, thank you so much for supporting it through your audienceship. That is quite enough, but if you're interested in taking your support of the show to the next level, then look at sponsoring the show through your company.

Send me an e-mail at jeff@softwareengineeringdaily.com. Thank you.

[END]