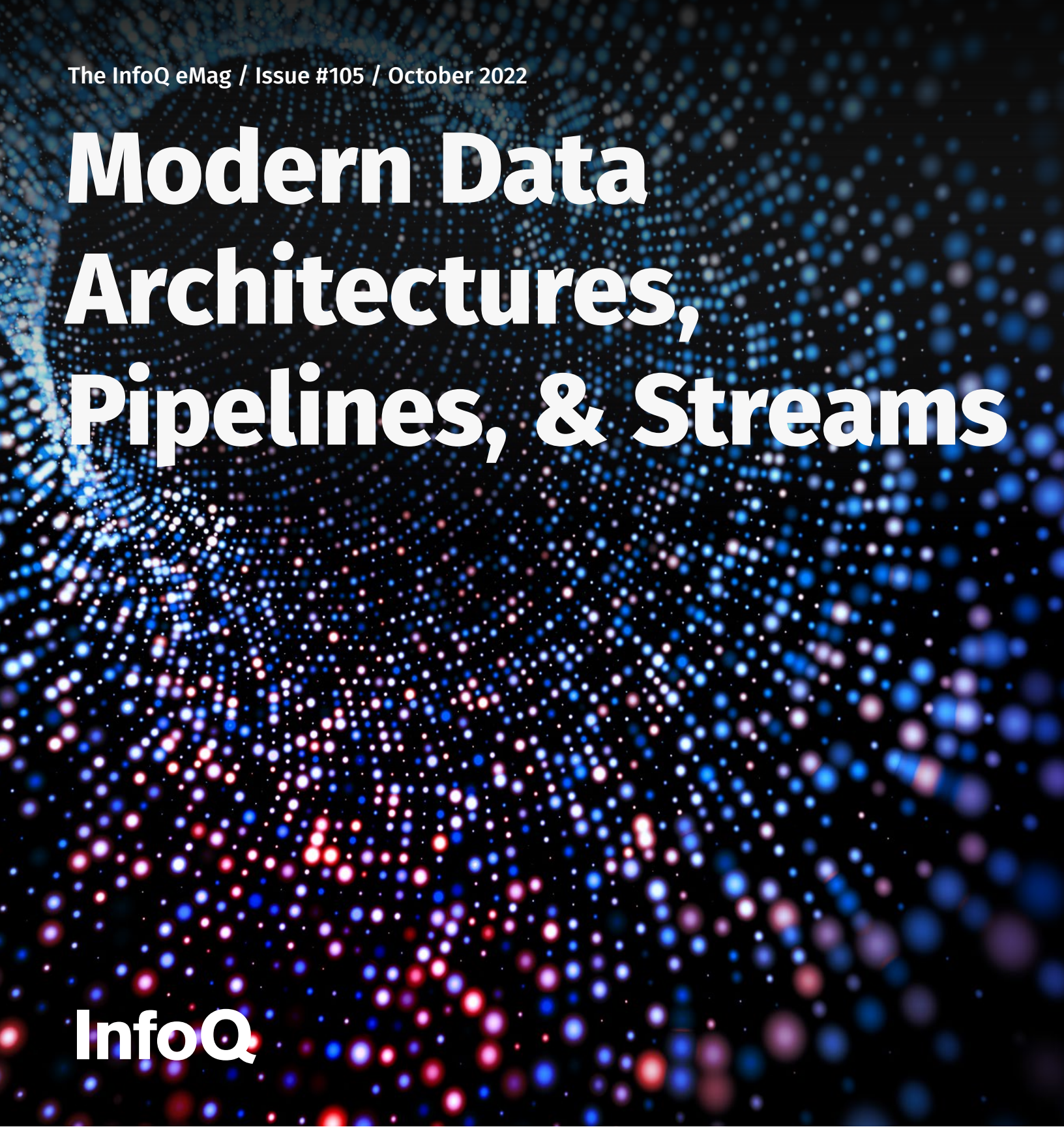
The InfoQ eMag / Issue #105 / October 2022

Modern Data Architectures, Pipelines, & Streams



InfoQ

Building & Operating High-Fidelity Data Streams

Migrating Netflix's Viewing History from Synchronous Request-Response to Async Events

Streaming-First Infrastructure for Real-Time Machine Learning

InfoQ @ InfoQ InfoQ InfoQ

Modern Data Architectures, Pipelines, & Streams

IN THIS ISSUE

PRODUCTION EDITOR Ana Ciobotaru / COPY EDITORS Maureen Spencer DESIGN Dragos Balasoiu / Ana Ciobotaru
GENERAL FEEDBACK feedback@infoq.com / ADVERTISING sales@infoq.com / EDITORIAL editors@infoq.com

CONTRIBUTORS

Sid Anand

currently serves as the Chief Architect and Head of Engineering for Datazoom, where he and his team build autonomous streaming data systems for Datazoom's high-fidelity, low latency streaming analytics needs. Prior Sid served as PayPal's Chief Data Engineer, focusing on ways to realize the value of PayPal's hundreds of petabytes of data. He also held several positions including Agari's Data Architect, a Technical Lead in Search @ LinkedIn, Netflix's Cloud Data Architect, Etsy's VP of Engineering, and several technical roles at eBay. Sid earned his BS and MS degrees in CS from Cornell University.

Sharma Podila

is Software Engineering leader, system builder, collaborator, mentor. He has deep expertise in cloud resource management, distributed systems, data infrastructure,and proven track record of delivering impactful large scale distributed systems of cross functional scope.

Chip Huyen

is a co-founder of Claypot AI, a platform for real-time machine learning. Previously, she was with Snorkel AI and NVIDIA. She teaches CS 329S: Machine Learning Systems Design at Stanford. She's the author of Designing Machine Learning Systems, an Amazon bestseller in AI. She has also written four bestselling Vietnamese books.

Mei Tao

is a product manager at Monte Carlo, a data reliability company. Prior to joining Monte Carlo, Mei worked in product management at NEXT Trucking and Product Strategy at Coinbase. She received her MBA from Harvard Business School and her B.S. in Statistics from the University of California, Berkeley.

Xuanzi Han

is a senior back-end engineer at Monte Carlo, a data reliability company. Previously, she worked as a software engineer at Uber on their Marketplace Intelligence team. She has also held software development roles at Google, Amazon, and eBay. She received her M.S. in Computer Science from the University of Southern California.

Helena Muñoz

is a senior software engineer at Monte Carlo, a data reliability company. Prior to joining Monte Carlo, she served as a senior full-stack engineer at Portchain, a shipping and logistics solutions provider, and a team lead at Infragistics, maker of a leading suite of design technologies. She graduated from the Universidad de la República in Uruguay with a degree in Computer Science.

A LETTER FROM THE EDITOR

Srini Penchikala

currently works as Senior Software Architect in Austin, Texas. He is also the Lead Editor for AI/ML/Data Engineering community at InfoQ. Srini has over 22 years of experience in software architecture, design and development. He is the author of "Big Data Processing with Apache Spark. He is also the co-author of "Spring Roo in Action" book from Manning Publications. Srini has presented at conferences like Big Data Conference, Enterprise Data World, JavaOne, SEI Architecture Technology Conference (SATURN), IT Architect Conference (ITARC), No Fluff Just Stuff, NoSQL Now and Project World Conference. He also published several articles on software architecture, security and risk management, and NoSQL databases on websites like InfoQ, The ServerSide, OReilly Network (ONJava), DevX Java, java.net and JavaWorld.

Data management space has been going through an exponential growth for the last several years in terms of volume, velocity, data structure, diverse data ingestion requirements and a wide variety of data processing & analytics use cases.

Data management comes with a variety of challenges like how it's created (real time, batch), how it's used (transactional, historical, or time-series data), data type (textual, audio, images, video), or the data structure (key-value, columnar, document, or graph).

Data is no longer limited to relational data storage, traditional data warehouses, ETL, or simple one-dimensional database queries and static reports. NoSQL databases and big data technologies have radically transformed the way we generate, ingest, process, and analyze data to perform predictive modeling and generate actionable intelligence and insights.

Recent innovations in AI and ML technologies also resulted in explosion of data we need to manage throughout all phases of data management lifecycle.

Cloud platforms and containerization technologies have significantly influenced how the data is stored, sharded, and replicated in cloud-hosted database clusters.

Data architects, developers and DBA's are realizing it's no longer a luxury to have automated data pipelines as part of their software development process. It has become a must-have, to be able to manage and process data in our applications in a seamless manner.

Data pipelines help with establishing end-to-end data processing solutions with capabilities like performance, scalability, redundancy, and failover. They offer several benefits like flexibility, data consistency, efficiency, and reliability.

A typical data pipeline consists of three stages: 1. data source(s), 2. processing steps, and 3. destination.

Data pipelines bring the same rigor and discipline in the areas of CI/CD and DevOps that the application tier has been benefiting for last 10+ years.

There are now, at the disposal of the data management community, the right tools, frameworks, best practices and to take advantage of data pipeline architectures instead of reinventing the wheel by creating one-off solutions from scratch.

In this eMag on “Modern Data Architectures, Pipelines and Streams”, you’ll find up-to-date case studies and real-world data architectures from technology SME’s and leading data practitioners in the industry.

“Building & Operating High-Fidelity Data Streams” by Sid Anand highlights the importance of reliable and resilient data stream architectures. He talks about how to create high-fidelity loosely-coupled data stream solutions from the ground up with built-in capabilities such as scalability, reliability, and operability using messaging technologies like Apache Kafka.

Sharma Podila’s article on “Microservices to Async Processing Migration at Scale” emphasizes the importance of asynchronous processing and how it can improve the availability of a web service by relieving backpressure using Apache Kafka by implementing a durable queue between service layers.

“Streaming-First Infrastructure for Real-Time Machine Learning” by Chip Huyen nicely captures the benefits of streaming-first infrastructure for real-time ML scenarios like online prediction and continual learning.

And “Building End-to-End Field Level Lineage for Modern Data Systems” authored by Mei Tao, Xuanzi Han and Helena Muñoz describes the data lineage as a critical component of data pipeline root cause and impact analysis workflow, and how automating lineage creation and abstracting metadata to field-level helps with the root cause analysis efforts.

We at InfoQ hope that you find value out of the articles and other resources shared in this eMag and potentially apply the design patterns and techniques discussed, in your own data architecture projects and initiatives.

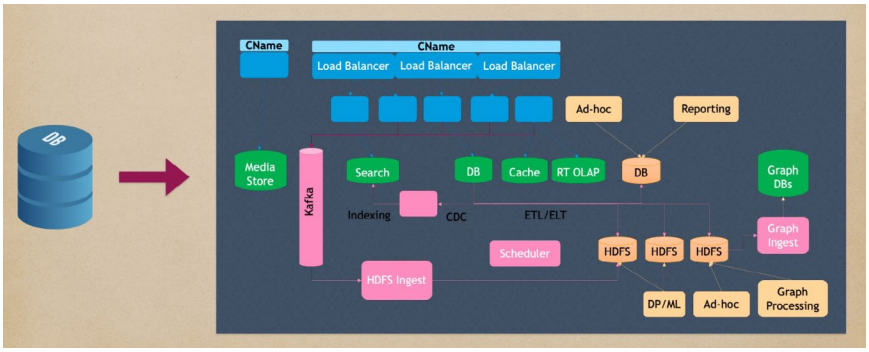
Building & Operating High-Fidelity Data Streams

by **Sid Anand**, Chief Architect and Head of Engineering @Datazoom

At QCon Plus 2021 last November, Sid Anand, Chief Architect at Datazoom and PMC Member at Apache Airflow, presented on building high-fidelity nearline data streams as a service within a lean team. In this talk, Sid provides a master class on building high-fidelity data streams from the ground up.

Introduction

In our world today, machine intelligence and personalization drive engaging experiences online. Whether that's a learn-to-rank system that improves search quality at your favorite search engine, a recommender system that recommends music, or movies, recommender systems that recommend who you should follow, or ranking systems that re-rank a feed on your social platform of choice. Disparate data is constantly being connected to drive predictions that keep us engaged. While it may seem that some magical SQL join powers these connections, the reality is that data growth has made it impractical to store all of this data in a single DB. Ten years ago, we used a single monolithic DB to store data, but today, the picture below is more



representative of the modern data architectures we see.

It is a collection of point solutions tied together by one or more data movement infrastructure services. How do companies manage the complexity below? A key piece to the puzzle is data movement, which usually comes in two forms, either batch processing or stream processing.

What makes streams hard?

There are lots of moving parts in the picture above. It has a very large surface area, meaning there are many places where errors can occur. In streaming architectures, any gaps in non-functional requirements can be very unforgiving.

Data engineers spend much of their time-fighting fires and keeping systems up if they don't build these systems with the "*ilities*" as first-class citizens; by "*ilities*," I mean nonfunctional requirements such as scalability, reliability, observability, operability, and the like. To underscore this point, you will pay a steep operational tax in the form of data pipeline outages and disruptions if your team does not build data systems with "*ilities*" as first-class citizens.

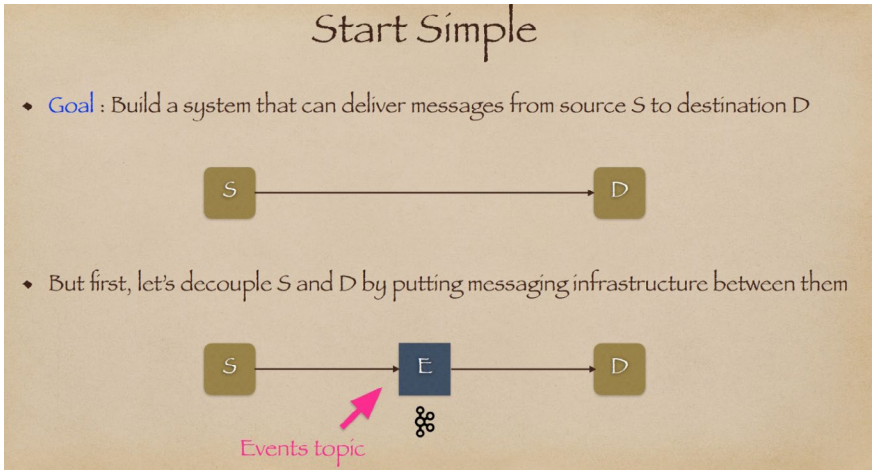
Disruptions and outages typically translate into unhappy customers and burnt-out team members that eventually leave your team. Let's dive in and build a streaming system.

Start Simple

This is an ambitious endeavor. As with any large project or endeavor, I like to start simple. Let's start by defining a goal:

We aim to build a system that delivers messages from source S to destination D

First, let's decouple S and D by putting a message broker between them. This is not a controversial decision; it's conventional and used worldwide. In this case, I've picked Kafka as my technology, but you can use any message brokering system. In this system, I've created a single topic called E, to signify events that will flow through this topic. I've decoupled S and D with this event topic. This means that if D fails, S can continue to publish messages to E. If S fails, D can continue to consume messages from E.



Let's make a few more implementation decisions about this system. Let's run our system on a cloud platform. For many, that just means running it on a publicly available cloud platform like AWS or GCP. It can also mean running it on Kubernetes on-prem, or some mix of the two. Additionally, to start with, let's operate at a low scale. This means that we can run Kafka with a single partition in our topic.

However, since we want things to be reliable, let's run three brokers split across three availability zones, and set the RF or replication factor to 3. Additionally, we will run S and D on single but separate EC2 instances to increase the availability of our system. Let's provide our stream as a service to make things more interesting. This means we can accept inbound messages at an API endpoint hosted at process S. When messages arrive, S will process them and send them to event topic E. Process D will

consume from event topic E and send the message to some third-party endpoint on the internet.

Reliability

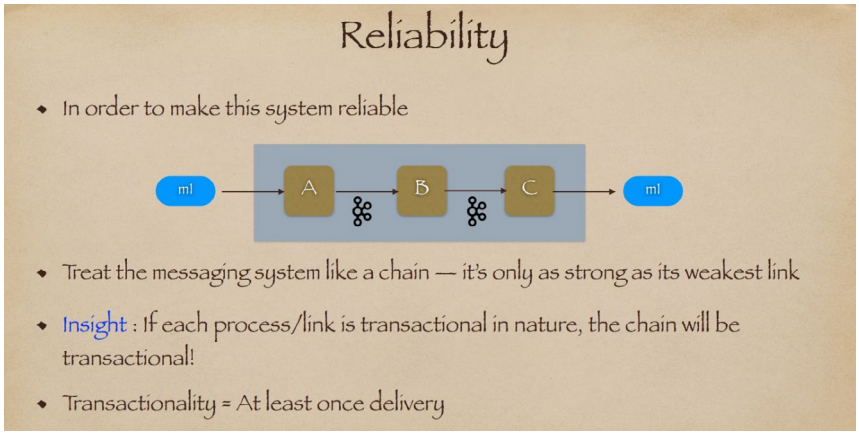
The first question I ask is: "is this system reliable?" Let's revise our goal:

We want to build a system that delivers messages **reliably** from S to D.

To make this goal more concrete; I'd like to add the following requirement:

I want zero message loss

What does this requirement mean for our design? It means that once process S has acknowledged a message to a remote sender, D must deliver that message to a remote receiver. How do we build reliability into our system? Let's first generalize our system. Instead of just S and D, let's say we have three processes, A, B, and C, all of which are connected via Kafka topics. To make this system reliable, let's treat this linear topology as a chain. Like any chain, it's only as strong as its weakest link. If each process or link is transactional in nature, this chain will be transactional. My definition of "*transactionality*" is at least once delivery since this is the most common way that Kafka is used today. How do we make each link transactional? Let's first break this chain into its component processing links.

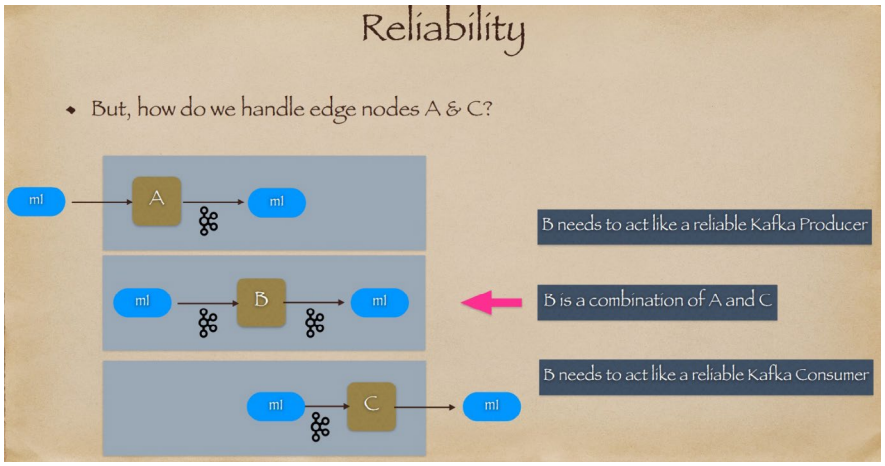


First, we have A. A is an ingest node. Then we have B. B is an internal node. It reads from Kafka and writes to Kafka. Finally, we have C. C is an expel or egest node. It reads from Kafka and sends a message out to the internet.

How do we make A reliable? A will receive a request via its REST endpoint, process the message m1, and reliably send the data to Kafka as a Kafka event. Then, A will send an HTTP response back to its caller. To reliably send data to Kafka, A will need to call kProducer.send with two arguments: a topic and a message. A will then immediately need to call flush, which will flush internal Kafka buffers and force m1 to be sent to all three brokers. Since we have producer config acks=all, A will wait for an acknowledgment of success from all three brokers before it can respond to its caller.

How about C? What does C need to do to be reliable? C needs to read data, typically a batch from Kafka, do some processing

on it, and then reliably send data out. In this case, reliably means that it will need to wait for a 200 OK response code from some external service. Upon receiving that, process C

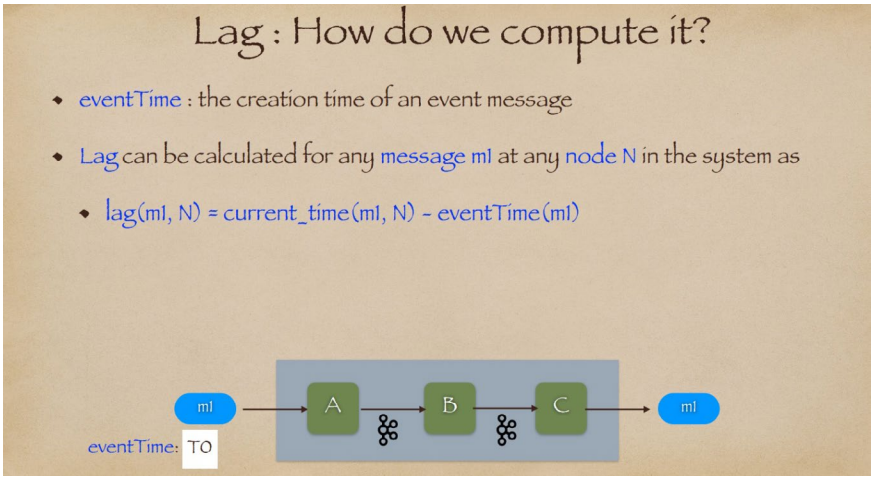


will manually move its Kafka checkpoint forward. If there are any problems, process C will negatively ACK (i.e. NACK) Kafka, forcing a reread of the same data. Finally, what does B need to do? B is a combination of A, and C. B needs to act like a reliable Kafka Producer like A, and it also needs to act like a reliable Kafka consumer like C.

What can we say about the reliability of our system? What happens if a process crashes? If A crashes, we will have a complete outage at ingest. That means our system will not accept any new messages. If instead C crashes, this service will stop delivering messages to external consumers; However, that does mean A will continue to receive messages and save them to Kafka. B will continue to process them but C will not deliver them until process C recovers. The most common solution to this problem is to wrap each service in an autoscaling group of size T. If we do so, then each of the groups can handle T-1 concurrent failures. While the term “autoscaling group” was coined by Amazon, autoscaling groups are available on all cloud platforms and in Kubernetes.

Lag

For now, we appear to have a pretty reliable data stream. How do we measure its reliability? This brings us to observability. In streaming systems, there are two primary quality metrics



that we care about : lag and loss. You might be familiar with these metrics if you've worked in streams before. If you're new to it, I will give you an introduction.

Firstly, let's start with lag.

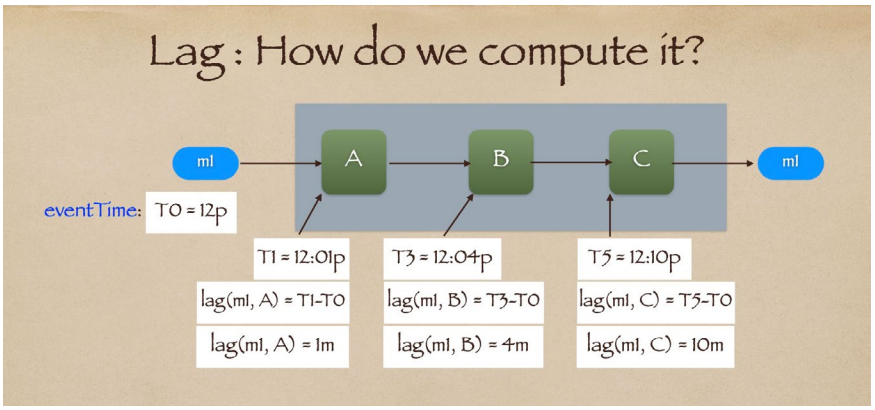
Lag is simply a measure of message delay in a system.

The longer a message takes to transit a system, the greater its lag. The greater the lag, the greater the impact on a business, especially one that depends on low-latency insights. Our goal is to minimize lag in order to deliver insights as quickly as possible.

How do we compute lag? To start with, let's discuss one of the concepts called event time. Event time is the creation time of a message or event. Event time is typically stored within the message body and travels with the message as it transits our system. Lag can be calculated for any message m1 at any node N in the system using the equation shown in the figure above.

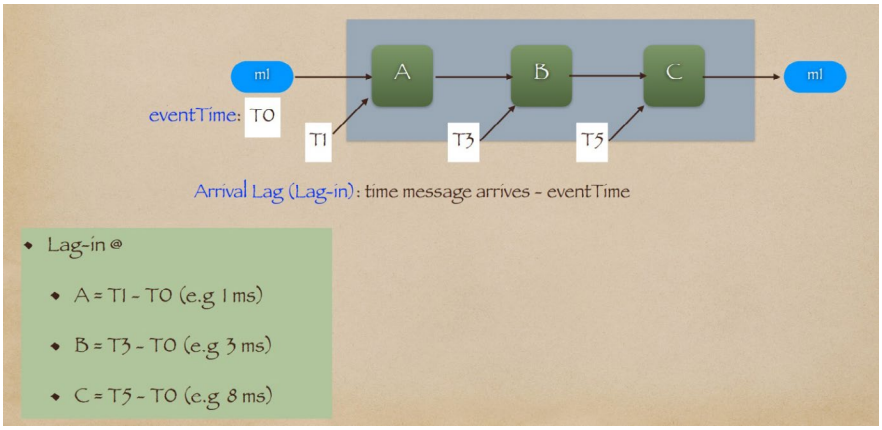
Let's look at a real example shown in the figure below.

Let's say we have a message created at noon (T0). That message arrives at our system at



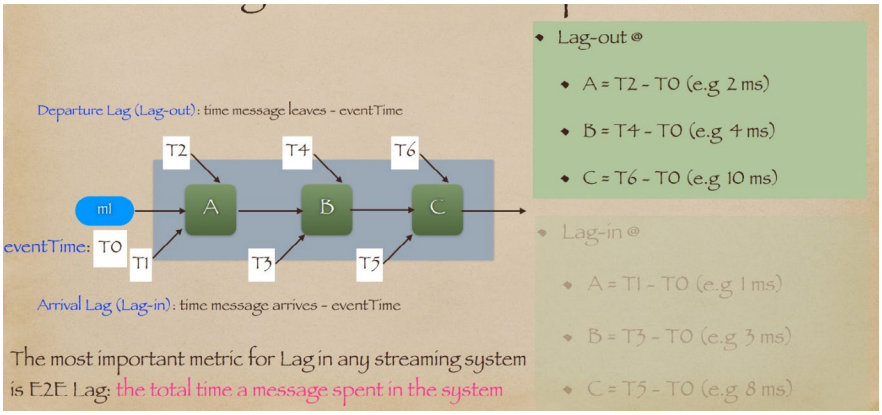
node A at 12:01 p.m (T1). Node A processes the message and sends it to node B. The message arrives at node B at 12:04 p.m (T3). B processes it and sends it to C, receiving it at 12:10 p.m (T5). In turn, C processes the message and sends it on its way. Using the equation from the page before, we see that T1-T0 is one minute, T3-T0 is four minutes, and so on; we can compute the lag of message m1 and node C at 10 minutes.

In practice, lag in these systems are not on the order of minutes but on the order of milliseconds.



One thing I will mention is that we've been talking about message arrival at these nodes; hence, this is called arrival lag or lag-in. Another thing to observe is that lag is cumulative. That means the lag computed at node C accounts for the lag upstream of it in both nodes A and B. Similarly, the lag computed at node B accounts for the lag upstream of it at node A. While we talked about arrival lag, there is another type of lag called departure lag. Departure lag is calculated when messages leave nodes. Similar to how we calculated arrival lag, we can compute departure lag.

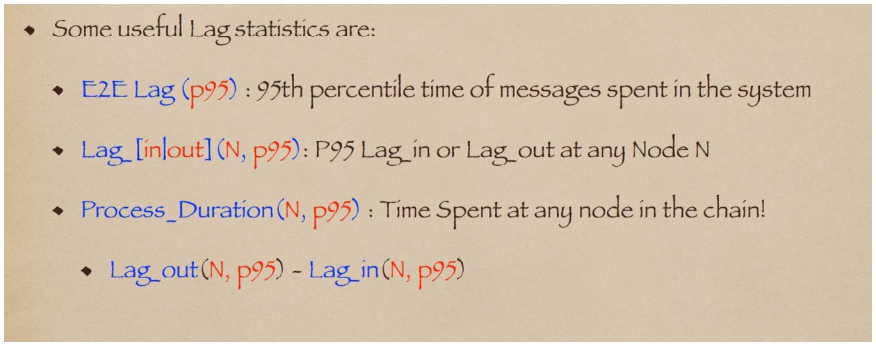
The single most important metric for lag in any streaming system is called end-to-end lag (a.k.a. E2E Lag). E2E lag is the total time a message spends in the system. E2E lag is straightforward to compute as it's simply the departure lag at the last node in the system. Hence, it's the departure lag at node C.

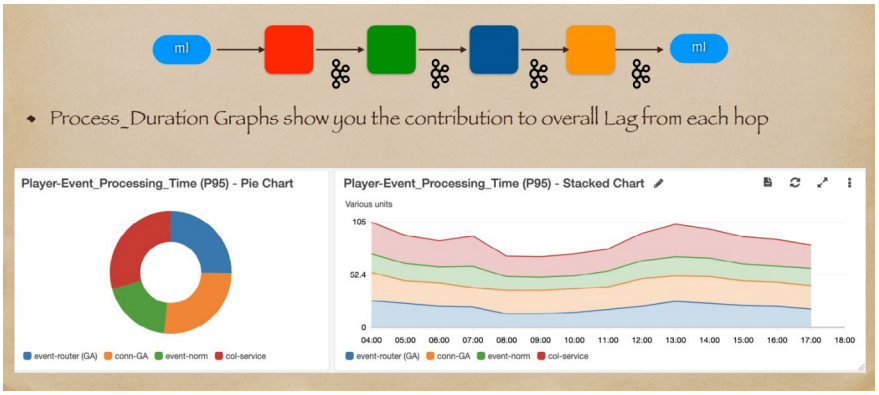


While knowing the lag for a particular message (ml) is interesting, it's of little use when we deal with billions or trillions of messages. Instead, we leverage statistics to capture population behavior. I prefer the use of the 95th (or 99th) percentiles (a.k.a. P95). Many people prefer Max or P99.

Let's look at some of the statistics we can build. We can compute the end-to-end lag over the population at P95. We can also compute the lag-in or lag-out at any node. We can compute what's called a process duration with lag-out and lag-in: this is the time spent at any node in the chain. Why is that useful?

Let's have a look at a real example. Imagine this topology; we have a message that hits a system with four nodes, the red node, green node, blue node, and finally, an orange node. This is actually a system we run in production. The graph below is taken from CloudWatch for our production service. As you can see, we took the process duration for each node and put it in a pie chart. This gives us the lag contribution of each node in the system. The lag contribution is approximately equal for each



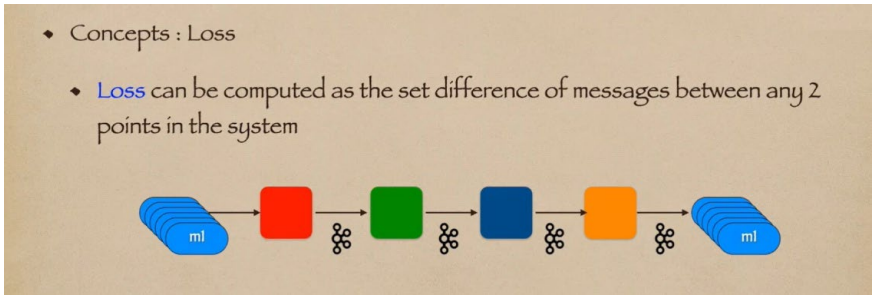


node. No single node stands out. This is a very well-tuned system. If we take this pie chart and spread it out over time, we get the graph on the right, which shows us that the performance is consistent over time. We have a well-tuned system that performs consistently over time.

Loss

Now that we've talked about lag, what about loss? Loss is simply a measure of messages lost while transiting the system. Messages can be lost for various reasons, most of which we can mitigate—the greater the loss, the lower the data quality.

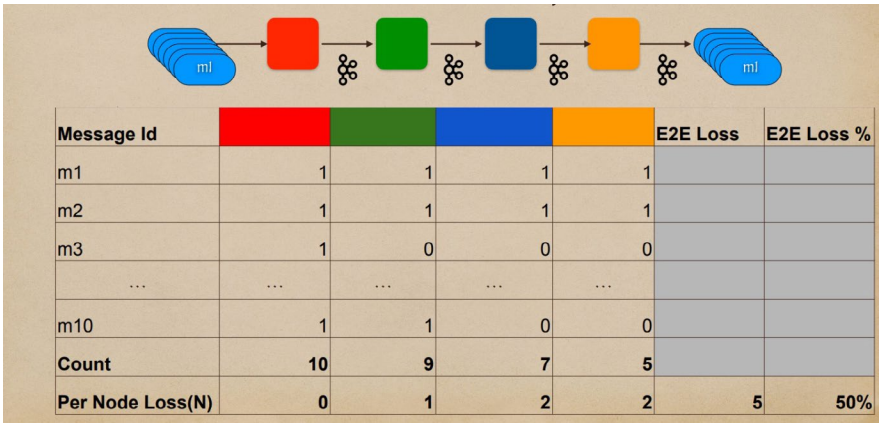
Hence, our goal is to minimize loss to deliver high-quality insights. You may be asking yourself "how do we compute loss in a streaming system?" Loss can be computed as the set difference of messages between any two points in the system. If we look at our topology from before, the one difference you see here is that we have ten messages transiting the system instead of a single message transiting the system.



We can use the following loss table to compute loss. Each row in the loss table is a message, each column is a node in the chain. As a message transits the system, we tally a 1. For example, message one is successfully transited through the entire system, so there's a 1 in each column. Message 2 also transits each node successfully in our system. However, message 3, while successfully processed at the red node, does not make it to the green node. Therefore, it doesn't reach the blue or the yellow node. At the bottom, we can compute the loss per node. Then, on the lower right, as you can see, we can compute the end-to-end loss in our system, which in this case is 50%.

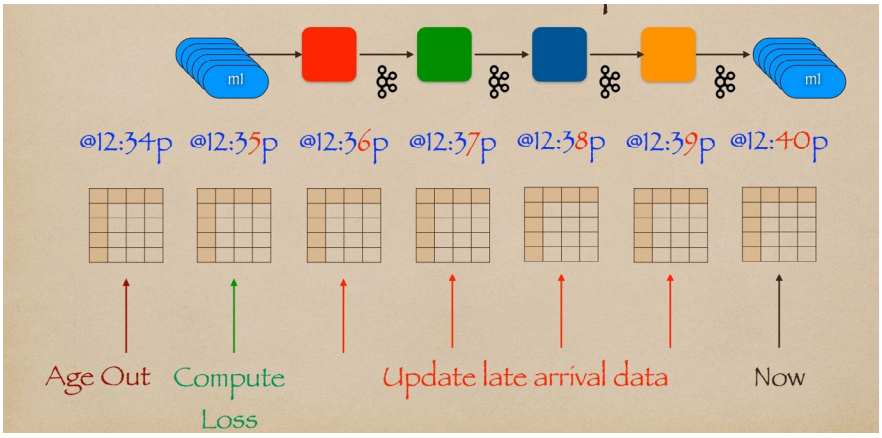
In a streaming data system, messages never stop flowing. How do we know when to count? The key is to allocate messages to 1-minute wide time buckets using message event time. For example, at the 12:34 minute of a day, we can compute a loss table, which includes all the messages whose event times fall in the 12:34 time.

Similarly, we can do this at other times in the day. Let's imagine that right now; the time is 12:40 p.m. As we know, in these systems, messages may arrive late. We can see four of the tables are still getting updates to their tallies. However, we may notice that at 12:35 p.m. table is no longer changing; therefore, all



Message Id					E2E Loss	E2E Loss %
m1	1	1	1	1		
m2	1	1	1	1		
m3	1	0	0	0		
...	...	...	...	...		
m10	1	1	0	0		
Count	**10**	**9**	**7**	**5**		
Per Node Loss(N)	**0**	**1**	**2**	**2**	**5**	**50%**

messages that will arrive have arrived. At this point, we can compute loss. Any table before this time, we can age out and drop. This allows us to scale the system and trim tables we no longer need for computation.



To summarize, we wait a few minutes for messages to transit and then compute loss. Then we raise the alarm if loss occurs over a configured threshold, for example, 1%. Now we have a way to measure the reliability and latency of our system.

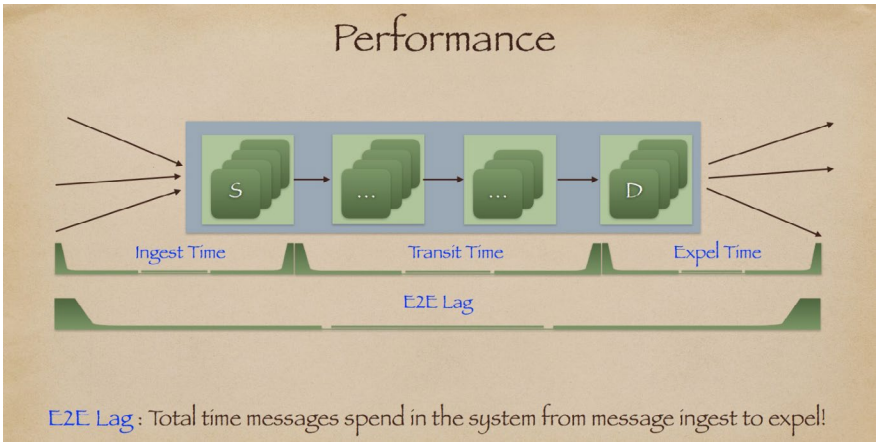
Performance

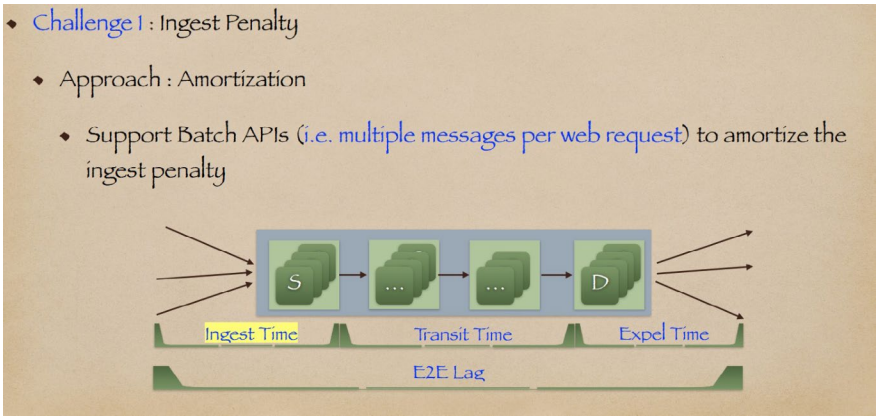
Have we tuned our system for performance yet? Let's revise our goal. We want to build a system that can deliver messages reliably from S to D with low latency. To understand streaming system performance, we need to understand the components of end-to-end lag. The first component I'd like to mention is called the ingest time. The ingest time is the time from the last byte in the request to the first byte out of the response. This time includes any overhead we incur in reliably sending messages to Kafka. At the end of our pipeline, we have something called the expel time or the egest time. This is the time to process and egest a message at D. Any time between these two is called transit time. All three times together are part of the end-to-end lag.

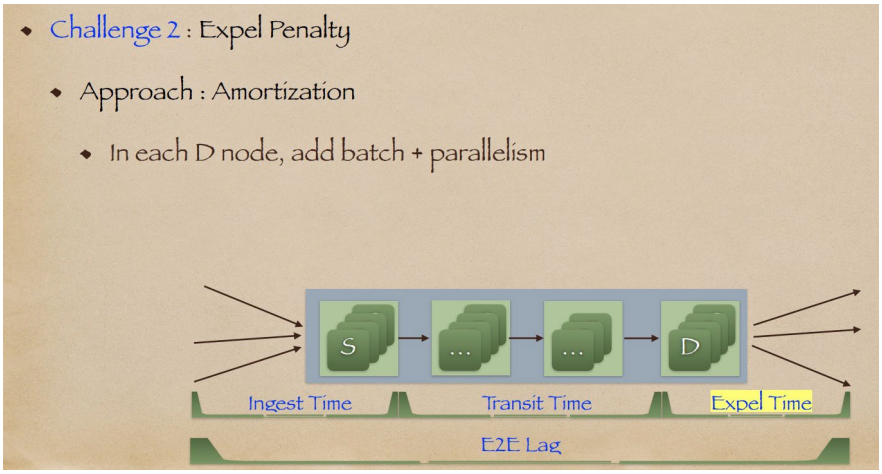
Performance Penalties

When discussing performance, we must acknowledge that there are performance penalties. In other words, we need to trade off latency for reliability. Let's look at some of these penalties. The first penalty is the ingest penalty. In the name of reliability, S needs to call **kProducer.flush** on every inbound API request. S also needs to wait for three acks from Kafka before sending its API response. Our approach here is to amortize. That means that we will support batch APIs; therefore, we get multiple messages per web request. We can then amortize the cost over multiple messages, thereby limiting this penalty.

Similarly, we have something called the expel penalty. There's an observation we need to consider. Kafka is very fast. It is many orders of magnitude faster than typical HTTP round trip times. In fact, the majority of the expel time is the HTTP round trip time. Again, we will use an amortization approach. In each D node, we will add batch and parallelism. We will read a batch from Kafka; then, we will re-batch them into smaller batches and use parallel threads to send these out to their destinations. This way, we can maximize throughput and minimize the expel cost or penalty per message.





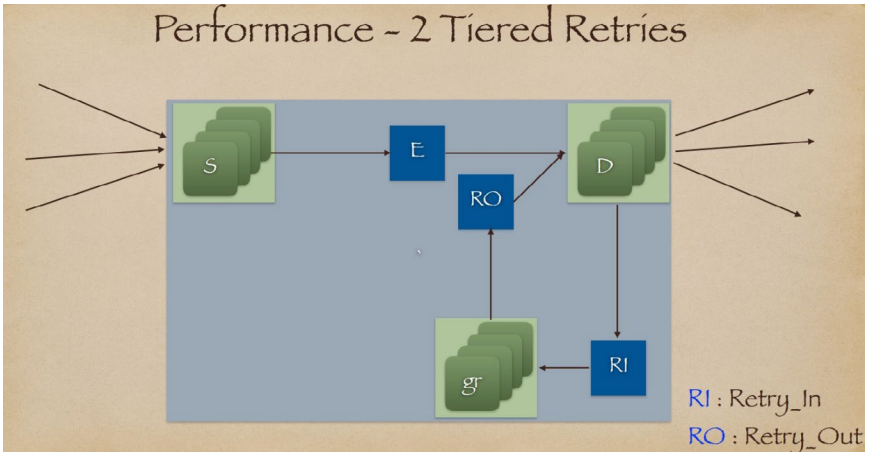


Last but not least, we have something called a retry penalty. In order to run a zero-loss pipeline, we need to retry messages at D that will succeed given enough attempts. We have no control over the remote endpoints. We have no control over those endpoints. They could be transient failures; there might be throttling. There might be other things going on that we have no control over. We have to determine whether we can succeed through retries. We call these types of failures recoverable failures. However, there are also some types of cases or errors that are not recoverable. For example, if we receive a 4xx HTTP response code, except for the throttle 429s, if we receive these 4xx types, we should not retry because they will not succeed even with retries. To summarize, to handle the retry penalty, we have to pay some latency penalty on retry. We need to be smart about what we retry, which we've already talked about. We're not going to retry any non-recoverable failures. We also have to be smart about how we retry.

Performance – Tiered Retries

One idea that I use is called tiered retries. We have two tiers with tiered retries: a local retry tier and a global retry tier. In the local tier, we try to send a message a configurable number of times at D. If we exhaust local retries, D transfers the message to a global retrier. The global retrier then retries the message over a longer span of time. The architecture looks something like this.

At D, we will try multiple times to retry a message. If we exhaust those local retries, we send the message to a topic called a retrying topic. A fleet of global retries will wait a configurable amount of time before they read from this topic and then immediately send the message to the retry out topic. D will



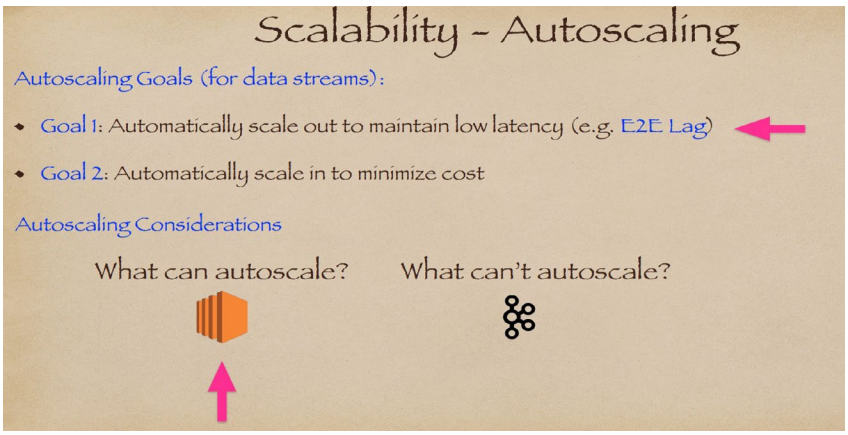
re-consume the message and try again. The beauty of this approach is that in practice, we typically see much less than 1% global retries, typically much less than 0.1%. Therefore, even though these take longer, they don't affect our P95 end-to-end lags.

Scalability

At this point, we have a system that works well at a low scale. How does this system scale with increasing traffic? First, let's dispel a myth. There is no such thing as a system that can handle infinite scale. Many believe that you can achieve this type of goal by moving to Amazon or some other hosted platform.

The reality is that each account has some limits on it, so your traffic will be capped. In essence, each system is traffic-rated, no matter where it runs. The traffic rating comes from the running load test. We only achieve higher scale by iteratively running load tests and removing bottlenecks.

When autoscaling, especially for data streams, we usually have two goals. Goal one is automatically scaling out to maintain low latency, for example, to minimize end-to-end lag. Goal two is to scale and to reduce cost. For now, I'm going to focus on goal one. When autoscaling, there are a few things to consider; firstly, what can we auto-scale? At least



for the last ten years or so, we have been able to auto-scale, at least in Amazon, compute. All of our compute nodes are auto-scalable. What about Kafka? Kafka currently does not support autoscaling, but it is something they're working on.

The most crucial part of autoscaling is picking the right metric to trigger autoscaling actions. To do that, we have to select a metric that preserves low latency, that goes up as traffic increases and goes down as the microservice scales out. In my experience, the average CPU is the best measure.

There are a few things to be wary of. If you have locks, code synchronization, or an IO waits in your code, there will be an issue. You will never be able to saturate the CPU on your box. As traffic increases, your CPU will reach a plateau. When that happens, autoscaler will stop, and your latency will increase. If you see this in your system, the simple way around it is to just lower your threshold below the CPU plateau. That will get you around this problem.

Conclusion

At this point, we have a system with non-functional requirements that we desire. While we've covered many key elements, we've left out many more: isolation, aautoscaling, multi-level autoscaling with containers, and the cache architecture.

How Disney+ Hotstar Modernized its Data Architecture for Scale

by **Cynthia Dunlop,** Senior Director of Content Strategy at ScyllaDB

Disney+ Hotstar, India's most popular streaming service, accounts for 40% of the global Disney+ subscriber base.

Disney+ Hotstar offers over 100,000 hours of content on demand, as well as livestreams of the world's mostwatched sporting events.

The "Continue Watching" feature is critical to the ondemand streaming experience for the 300 million-plus monthly active users. That's what lets you pause a video on one device and instantly pick up where you left off on any device, anywhere in the world. It's also what entices you to binge-watch your favorite series: complete one pisode of a show and the next one just starts playing automatically.

However, it's not easy to make things so simple. In fact, the underlying data infrastructure powering this feature had grown overly complicated. It was originally built on a combination of Redis and Elasticsearch, connected to an event processor for Apache Kafka streaming data. Having multiple data stores meant maintaining multiple data models, making each change a huge burden. Moreover, data doubling every six months required constantly increasing the cluster size, resulting in yet more admin and soaring costs.

Previous architecture: Here's how the "Continue Watching" functionality was originally architected.

First, the user's client would send a "watch video" event to Kafka. From Kafka, the event would be processed and saved to both Redis and Elasticsearch. If a user opened the home page, the backend was called, and data was retrieved from Redis and Elasticsearch. Their Redis cluster held 500 GB of data, and the Elasticsearch cluster held 20 terabytes.

Their key-value data ranged from 5 to 10 kilobytes per event. Once the data was saved, an API server read from the two databases and sent values back to the client whenever the user next logged in or resumed watching.

Redis provided acceptable latencies, but the increase in data size meant that they needed to horizontally scale their cluster. This increased their cost every three to four months. Elasticsearch latencies were on the higher end of 200 milliseconds. Moreover, the average cost of Elasticsearch was quite high considering the returns. They often experienced issues with node maintenance and manual effort was required to resolve the issues.

Modernized architecture: First, the team adopted a new data model that could suit both use cases. Then, they set out to adopt a new database. Apache Cassandra, Apache HBase, Amazon DynamoDB, and ScyllaDB were considered. The team selected ScyllaDB for two key reasons. 1) Consistently low latencies for both reads and writes, which would ensure a snappy user experience for today's demanding customers.

Try ScyllaDB Cloud with your projects - 30 days free

Migrating Netflix's Viewing History from Synchronous Request-Response to Async Events

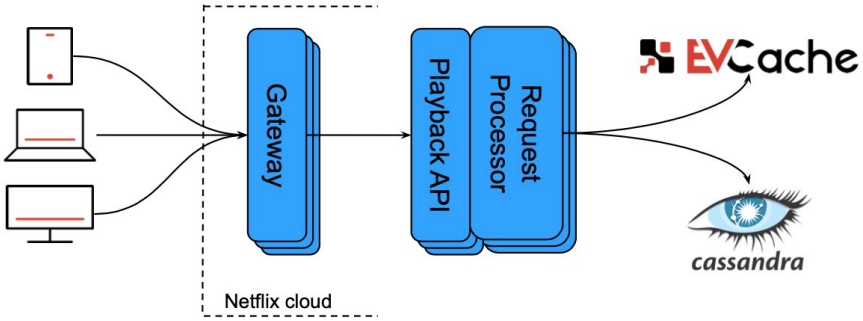
with **Sharma Podila**, Software Engineering Leader, System Builder, Mentor

Suppose you are running a web-based service. A slowdown in request processing can eventually make your service unavailable. Chances are, not all requests need to be processed right away. Some of them just need an acknowledgement of receipt. Have you ever asked yourself: "Would I benefit from asynchronous processing of requests? If so, how would I make such a change in a live, large-scale mission critical system?"

I'm going to talk about how we migrated a user-facing system from a synchronous request-response based system to an asynchronous one. I'm going to talk about what motivated us to embark on this journey, what system design changes we made, what were the challenges in this process, and what design choices and tradeoffs we made. Finally, I'm going to touch upon the validation process we used as we rolled out the new system.

Original Architecture

Netflix is a streaming video service available to over 200 million members worldwide. Members watch TV shows, documentaries, and movies on many different supported devices. When they come to



Netflix, they are given a variety of choices through our personalized recommendations. Users press play, sit back, and enjoy watching the movie.

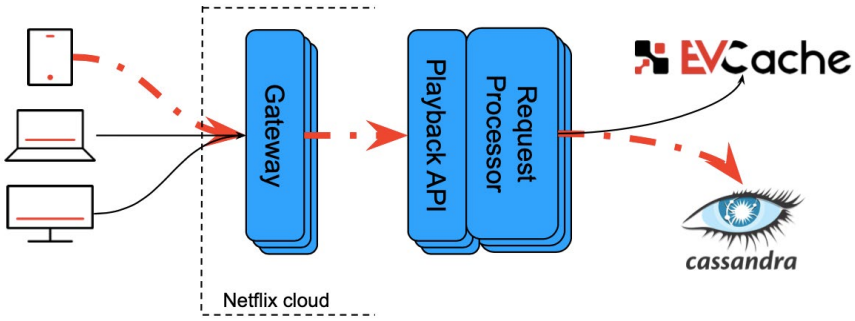
While the movie plays, during the playback we collect a lot of data, for both operational and analytical use cases. Some of the data drives our product features like continue watching, which lets our members stop a movie in the middle and come back to it later to continue watching from that point on any device. The data also feeds personalization and recommendations engines, and the core business analytics.

I'm going to talk about our experience migrating one of the product features, viewing history, which lets members see their past viewing activity and optionally hide it. Let's look at our existing system before the migration. At a high level, we have the Netflix client on devices such as mobile phones, computers, laptops, and TVs, that is sending data during playback into the Netflix cloud.

First the data reaches the Gateway Service. From there it goes to the Playback API, which manages the lifecycle of the playback sessions. In addition, it sends the playback data into the Request Processor layer. Within the Request Processor, among other things, it is storing both short term and long term viewing data into persistence, which for us is Apache Cassandra, and also into a caching layer, EVCache, which lets us do really quick lookups.

Backpressure

Most of the time, this system worked absolutely fine. Once in a rare while, it was possible that an individual request being processed would be delayed because of a network blip, or maybe one of the Cassandra nodes slowed down for a brief time.



Backpressure

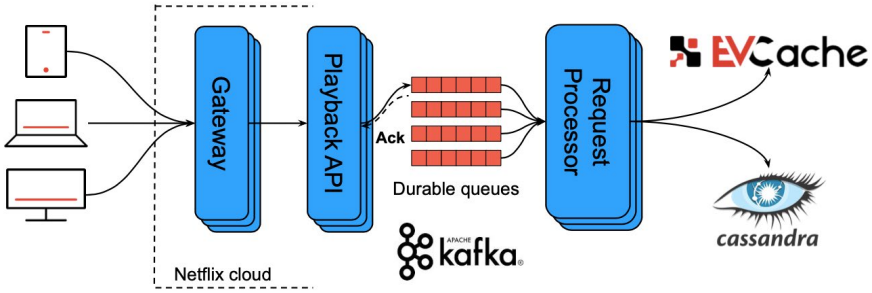
When that happened, since this is synchronous processing, the request processing thread in the Request Processor layer had to wait. Then this in turn slowed down the upstream Playback API service, which in turn slowed down the Gateway Service itself.

Beyond a few retry strategies within the cloud, the slowdown can hit the Netflix client that's running on the user's device. Sometimes this is referred to as backpressure. Backpressure can manifest itself as unavailability in your system and can build up a queue of items that the client may have to retry. Some of this data is very critical to what we do and we want to avoid any data loss; for example, if the clients were to fill their local queues, which are bounded.

Our solution to this problem was to introduce asynchronous processing into the system. Between the Playback API service and the Request Processor, we introduced a durable queue. Now when the request comes in, it's put into the durable queue, and immediately acknowledged. There is no need

to wait for that request to be processed.

It turns out, Apache Kafka fits this use case pretty well. Kafka presents a log abstraction to which the producers like Playback API can append to, and then multiple consumers can then read from the Kafka logs at their own pace using offsets.



This sounds simple. But if we simply introduce Apache Kafka in between two of our processing layers, can we call it done? Not quite. Netflix operates at a scale of approximately 1 million events per second. At that scale, we encountered several challenges in asynchronous processing: data loss, processing latencies, out of order and duplicate records, and intermittent processing failures. There are also design decisions around Kafka consumer platform choice as well as cross-region aspects.

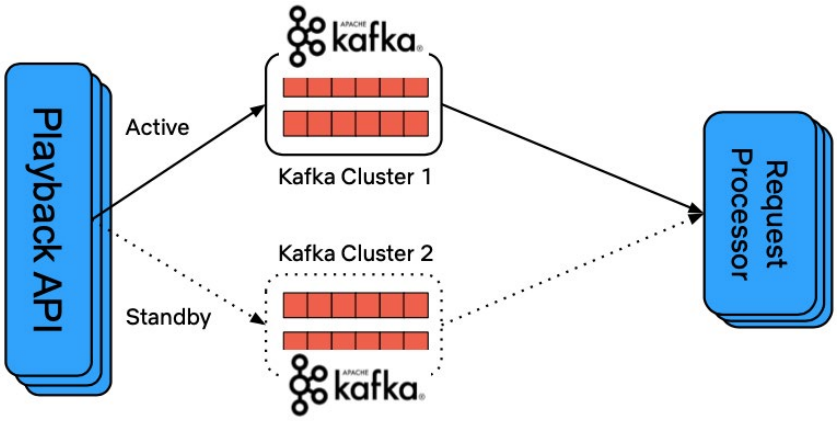
Challenge: Data Loss

There are two potential sources of data loss. First: if the Kafka cluster itself were to be unavailable, of course, you might lose data. One simple way to address that would be to add an additional standby cluster. If the primary cluster were to be unavailable due to unforeseen reasons, then the publisher---in this case, Playback API---could then publish into this standby cluster. The consumer request processor can connect to both Kafka clusters and therefore not miss any data.

Obviously, the tradeoff here is additional cost. For a certain kind of data, this makes sense. Does all data require this? Fortunately not. We have two categories of data for playback. Critical data gets this treatment, which justifies the additional cost of a standby cluster. The other less critical data gets a normal single Kafka cluster. Since Kafka itself employs multiple strategies to improve availability, this is good enough.

Another source of data loss is at publish time. Kafka has multiple partitions to increase scalability. Each partition is served by an ensemble of servers called brokers. One of them is elected as the leader. When you are publishing into a partition, you send the data to the leader broker. You can then decide to wait for only the leader to acknowledge that the item has actually been persisted into durable storage, or you can also wait for the follower brokers to acknowledge that they have written into persistence as well. If you're dealing with critical data, it will make sense to wait for acknowledgement for all brokers of the partition. At a large scale, this has implications beyond just the cost of waiting for multiple writes.

What would happen if you were to lose the leader broker? This happened to us just a couple of months after we deployed our new architecture. If a broker becomes unavailable and you



are waiting for acknowledgement from it, obviously your processing is going to slow down. That slowdown causes backpressure and unavailability, which we're trying to avoid.

If we are only waiting to get acknowledgement from just the leader broker, there's an interesting failure mode. What if you were to then lose the leader broker after a successful publish? Leader election will come up with a different leader. However, if the item that was acknowledged by the original leader was not completely replicated into the other brokers, then doing such an election of the leader could make you lose data, which is what we're trying to avoid. This is called an unclean broker leader election.

How did we handle the situation? Again, there's a tradeoff here to make. We have a producer library that is a wrapper on top of the Kafka producer client. There are two optimizations that are relevant here. First, because we use non-keyed partitions, the library is able to choose the partition it writes to. If one partition is unavailable because the leader broker is unavailable, our library can write to a different partition.

Also, if the partition is on an under-replicated set of brokers---that is, the leader broker has more items than the follower leaders, and the replication has not caught up completely---then our library picks a different partition that is more well replicated.

With these strategies, we eventually ended up choosing to write in asynchronous mode, where the publisher writes it into an in-memory queue and asynchronously publishes into Kafka. This helps scale performance, but we were interested in making sure we have an upper bound on the worst-case data loss we would incur if multiple errors are all happening at the same time. We were happy with the upper bound we were able to configure based on the in-memory queue size, and the strategy of avoiding under-replicated partitions.

We monitor this data durability, and we consistently get four to five nines from it, which is acceptable for us. If your application must not lose any items of data, then you may want to pick acknowledgment from all brokers before you call that item processed.

Challenge: Processing Latency and Autoscaling

One unavoidable side effect of introducing Kafka into our system is that there is now additional latency in processing a request; this includes the time required for the Playback API to publish the data to Kafka and for the Request Processor to consume it.

There is also the amount of time that the data waits in the Kafka queue. This is referred to as the lag, and it is a function of the number of consumer worker nodes and of the traffic volume. For a given number of nodes, as traffic volume increases, so will lag.

If you have a good idea of the peak traffic you're going to get, then you can figure out the number of consumer processing nodes you need in your system to achieve an acceptable lag. You could then simply provision your system to manage the peak traffic volume, or "set it and forget it."

For us, the traffic changes across the day and across the day of the week as well. We see a 5x change from peak to trough of our traffic. Because of such a big volume change, we wanted to be more efficient with our resources, and we chose to autoscale. Specifically, we add or remove a certain number of consumer processing nodes based on the traffic.

Whenever you change the number of consumers of a Kafka topic, all of that topic's partitions are rebalanced across the new set of consumers. The tradeoff here is resource efficiency versus paying the price of a rebalance. Rebalance can affect you in different ways.

If your processing is stateful, then you would have to do something complex. For example, your consumers may have to pause processing, then take any in-memory state, and checkpoint that along with the Kafka offset up to which they have processed. After the partitions are rebalanced, the consumers reload the checkpointed data, and then start processing from the checkpointed offset.

If your processing is a little simpler, or if you are storing state in an external fashion, then it is possible for you to let the rebalance happen and just continue normally.

What's going to happen here is that since you may have had items that were in process when the rebalance starts, and have not been acknowledged into Kafka, those items would show up on another processing node, because that node now gets that partition after the rebalance. In the worst case, you are going to reprocess some number of items. If your processing is idempotent or if you have other ways of dealing with duplicates, then this is not a problem.

The next question is: when and by how much to auto scale? One might think lag is a good metric to trigger auto scaling. The problem is that you cannot easily scale down by this metric. When the lag is zero, how do we know if we should scale down by one processing node, 10, or 50? Removing too many nodes at once might result in "flapping": removing and re-adding nodes over and over in a short time span.

In practice, many developers use a proxy metric; for example, CPU utilization. For us, records-per-second (RPS) turns out to be a good trigger for autoscaling. When our system is in a steady state, we measure the RPS to establish a baseline. Then we can add or remove nodes as our throughput changes relative to that baseline.

We also have different patterns for scaling up vs. scaling down. We want to avoid rebalances during scale-up, because we are already seeing a lot of incoming data, and rebalances will temporarily slow down consumers, so we want to quickly scale up. Scaling down can be done more gradually, since the current throughput is higher than it needs to be and we can accept the slowdown from a rebalance.

Challenge: Out of Order and Duplicate Records

Out of order and duplicate records are going to happen in a distributed system. How you address this problem will depend on the specifics of your application. In our case, we apply sessionization, which is collecting events for a video playback session that has specific start and stop events. Therefore, we collect all events for a session within those boundaries.

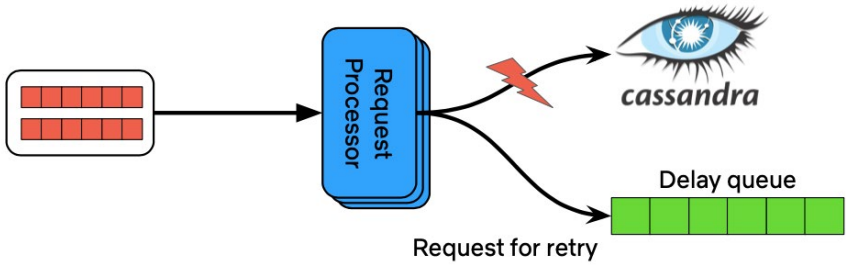
Given multiple events of a session, and based on certain attributes within the data, we can order them and also deduplicate them. For example, each event could have a monotonically increasing ID or a timestamp from the client. For writes, we are able to deduplicate them by using the timestamp when the event reaches our server.

Challenge: Intermittent Processing Failures

On the consumer side, we still have to deal with intermittent failures of processing. Typically, we wouldn't want to hold up processing the entire contents of the queue because of one failed item---sometimes referred to as head-of-line blocking. Instead, we put the failed item aside, process the rest of the queue, and then come back to it later.

A characteristic we would like for such a system is that there should be a finite time elapsing before we try it again; there is no need to try it immediately. We can use a separate queue, a delay queue, for these failed items.

There's a variety of ways you can implement this. Maybe you could write it into another Kafka topic, and then build another processor that builds in a delay.



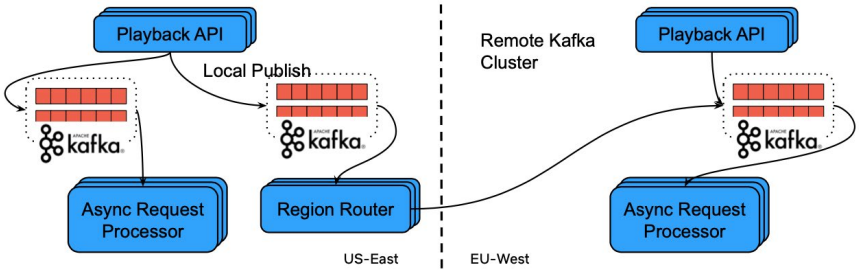
For us, it's very easy to achieve this using Amazon Simple Queue Service (SQS), since we already operate on Amazon Elastic Compute Cloud (EC2). We submit failed items to an SQS queue, and then the queue has a feature to optionally specify an interval before the item is available to consume.

Consumer Platform

There are multiple platforms we could use for consuming and processing items from Kafka. At Netflix, we use three different ones. Apache Flink is a popular stream processing system.\ Mantis is a stream processing system that Netflix open-sourced a few years ago. Finally, Kafka has an embeddable consumer client, which allows us to write microservices that just process the items directly. We started out with the question of: which platform is the best one to use? Eventually we realized that's not the right question; instead, the question is: which processing platforms benefit which use cases? Each of these three have pros and cons, and we use all three for different scenarios.

If you're doing complex stream processing, Mantis and Apache Flink are a very good fit. Apache Flink also has a built-in support for stateful stream processing where each node can store its local state, for example, for sessionization.

Microservices are very compelling, at least for us, because Netflix engineers have excellent support for the microservices ecosystem, all the way from generating or starting with a clean code base, to CI/CD pipelines and monitoring.



Cross-Region Aspects

Cross-region aspects are important because Netflix operates in multiple regions. Since we are running a large distributed system, it is possible that a region may become unavailable once in a while. We routinely practice for this, and multiple times a year, we take a region down just to make sure that we exercise our muscle of cross-region traffic forwarding.

At first glance, it might make sense that an item that was meant to be for another region could be just remotely published into a Kafka topic, using a tunnel across the regions. That normally would work except when you do encounter a real outage of that region, that remote publish is not going to work.

A simple but subtle change we made is that we always want to publish locally. We publish to another Kafka topic and asynchronously have a region router send it to the other side. This way, all events of a single playback session can be processed together.

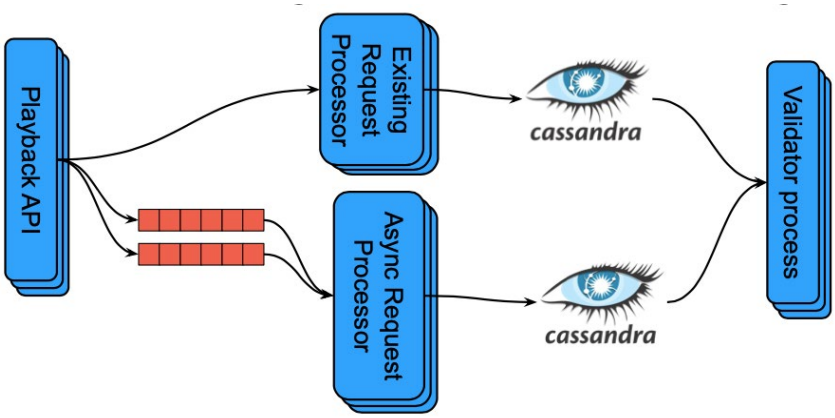
Testing, Validation, and Rollout

Now that we have challenges figured out and trade offs made, how did we test and roll it out? Shadow testing is one technique. Chances are, you may already be using a similar strategy in your environment. For us, this consisted of having Playback API dual-write to the existing

synchronous system and to Apache Kafka, from which the asynchronous request processor was consuming. Then we had a validator process that would validate that the in-flight request processing is identical.

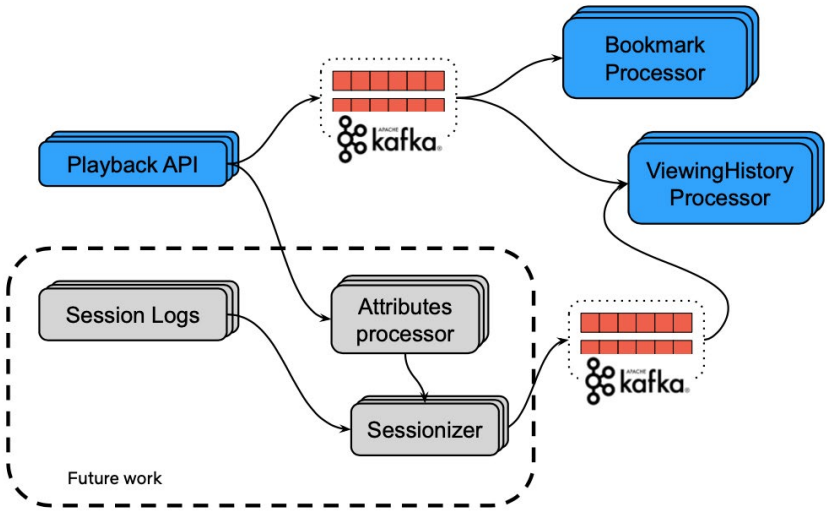
hen it gives you the additional benefit of confidence before rolling it out.

We rolled this out, splitting traffic by userId; that is, all traffic for a given userId was consistently



The next step was to also make sure that the stored artifacts were identical. For this we created a shadow Cassandra cluster. Here, you're trading off costs for higher confidence. If you have an environment where it is relatively easy to get additional resources for a short period of time, which is certainly possible in a cloud environment like ours, t

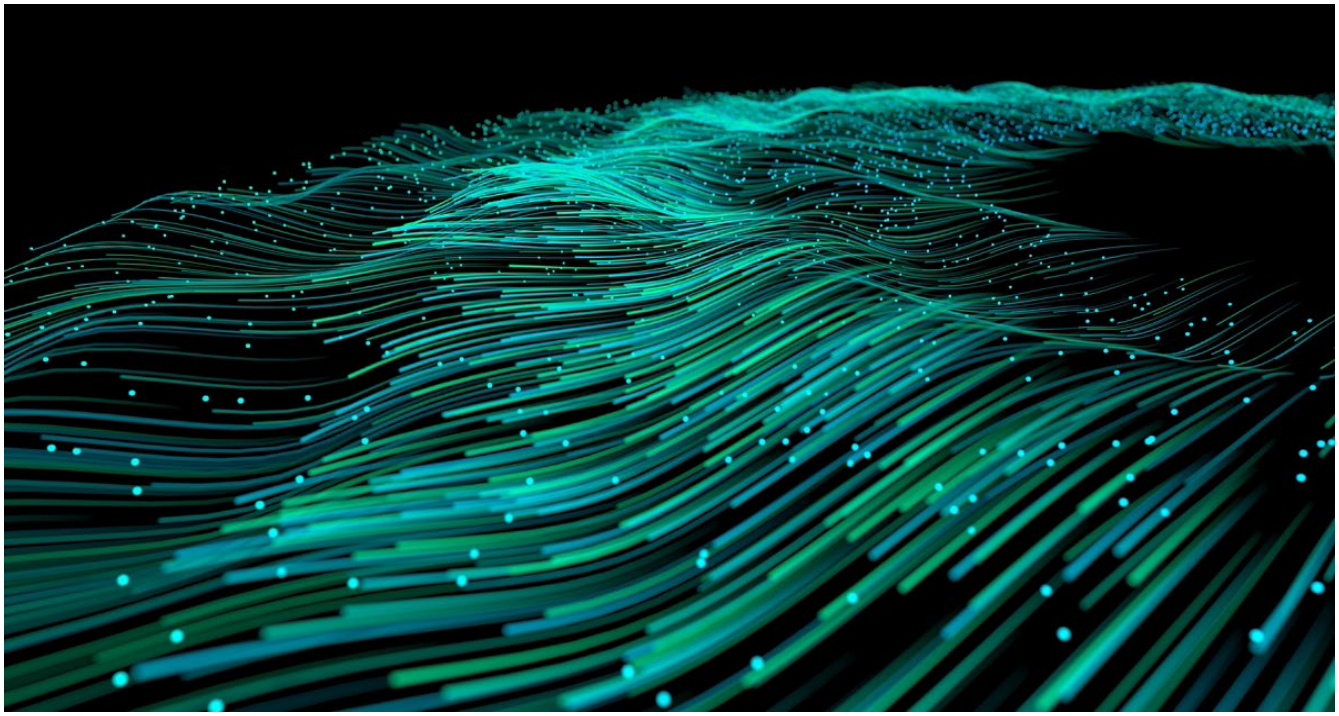
written into either the new system or the old. We started with 1% of users' data written to the new system, then incrementally increased the percentage all the way to 100% of users. That gave us really smooth migration without impact to upstream or downstream systems.



The second figure below shows where we are today and where we're going next. The items in blue, the Playback API, the Viewing History Processor, and the Bookmark Processor along with Kafka, are out in production now. We have the rest of the system that deals with additional data. There's an Attributes processor and Session Logs service, which will be interesting because the size of the data is very large: much larger than what you would normally write into Kafka. We will have some new challenges to solve there.

Conclusion

We've seen how asynchronous processing improved the availability and data quality for us, and how we reasoned about the design choices and what tradeoffs made sense for our environment. After implementation, shadow testing and incremental rollout gave us a confident and smooth deployment. With this information, think about how you could apply these lessons to your environment, and what other tradeoffs you may make for a similar journey.



Streaming-First Infrastructure for Real-Time Machine Learning

by **Chip Huyen**, CEO @Claypot AI & Teaching ML Sys @Stanford

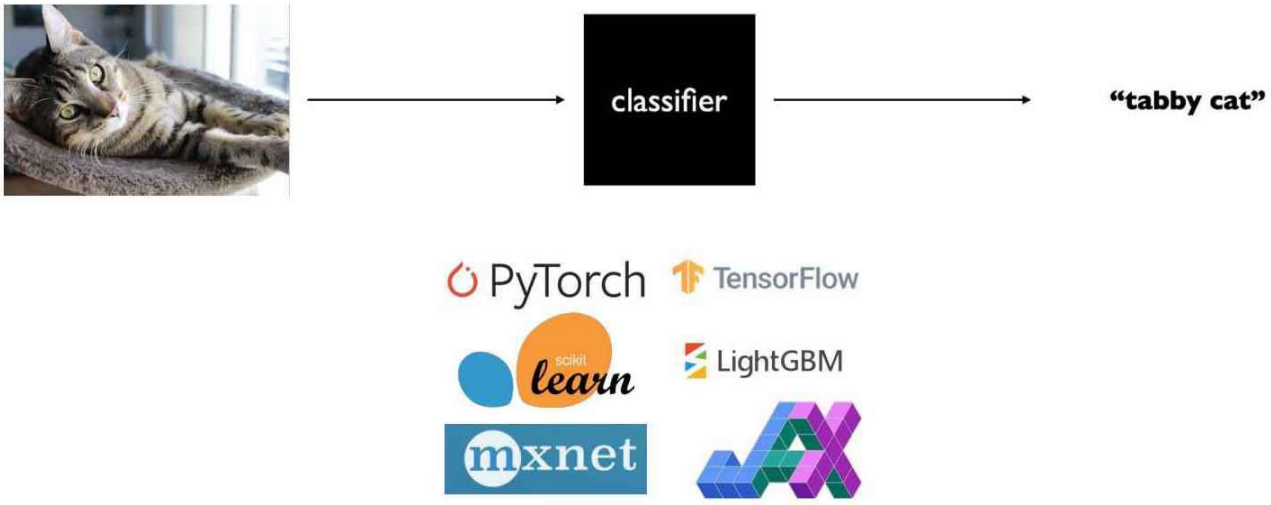
Many companies have begun using machine learning (ML) models to improve their customer experience. In this article, I will talk about the benefits of streaming-first infrastructure for real-time ML. There are two scenarios of real-time ML that I want to cover. The first is online prediction, where a model can receive a request and make predictions as soon as the request arrives. The other is continual learning. Continual learning is when machine learning models are capable of continually adapting to change in data distributions in production.

Online Prediction

Online prediction is pretty straightforward to deploy. If you have developed a model, the easiest way to deploy is to containerize it, then upload it to a platform like AWS or GCP to create an online prediction web service endpoint. If you send data to that endpoint, you can get predictions for this data.

The problem with online predictions is latency. Research shows that no matter how good your model predictions are, if it takes even a few milliseconds too long to return results, users will leave your site or click on something else. A common trend in ML is toward bigger models. These give better accuracy, but it generally also means that inference takes longer, and users don't want to wait.

How do you make online prediction work? You actually need two components. The first is a model that is capable of returning fast inference. One solution is to use model compression techniques such as quantization and distillation. You could also use more powerful



hardware, which allows models to do computation faster.

However, the solution I recommend is to use a real-time pipeline. What does that mean? A pipeline that can process data, input the data into the model, and generate predictions and return predictions in real time to users.

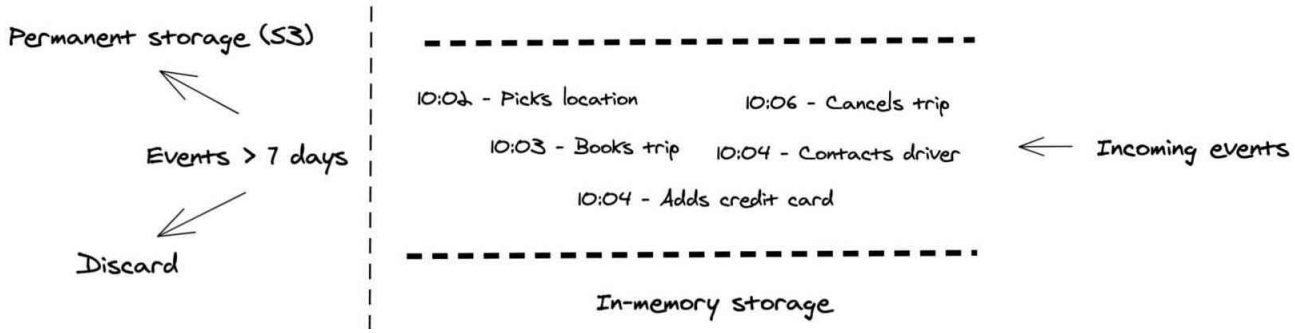
To illustrate a real-time pipeline, imagine you're building a fraud detection model for a ride-sharing service like Uber or Lyft. To detect whether a transaction is fraudulent, you want information about that transaction specifically as well as the user's other recent transactions. You also need to know about the specific credit card's recent transactions, because when a credit card is stolen, the thief wants to make the most out of that credit card by using it for multiple transactions at the same time. You also want to look into recent in-app fraud, because there might be a trend, and maybe this specific transaction is related to those other fraudulent transactions.

A lot of this is recent information, and the question is: how do you quickly assess these recent features? You don't want to move the data in and out of your permanent storage because it might take too long and users are impatient.

Real-Time Transport and Stream Processing

The solution is to use in-memory storage. When you have incoming events - a user books a trip, picks a location, cancels trip, contacts the driver - then you put all the events into in-memory storage, and then you keep them there for as long as those events are useful for real-time purposes. At some point, say after a few days, you can either discard those events or move them to permanent storage, such as AWS S3.

The in-memory storage is generally what is called real-



Static data	Streaming data
CSV, PARQUET, etc.	Kafka, Kinesis, Pulsar, etc.
Static features: • age, gender, job, city, income • when account was created	Dynamic features • locations in the last 10 minutes • recent activities
Bounded: know when a job finishes	Unbounded: never finish
Can be processed in batch • e.g. MapReduce, Spark	Processed as events arrive • e.g. Apache Flink, Samza, Spark Streaming

time transport, based on a system such as Kafka, Kinesis, or Pulsar. Because these platforms are event-based, this kind of processing is called *event-driven processing*.

Now, I want to differentiate between static data and streaming data. Static data is a fixed dataset, which contains features that don't change, or else change very slowly: things like a user's age or when an account was created. Also, static data is bounded: you know exactly how many data samples there are.

Streaming data, on the other hand, is continually being generated: it is unbounded. Streaming data includes information that is very recent, about features that can change very quickly. For example: a user's location in the last 10 minutes, or the web pages a user has visited in the last few minutes.

Static data is often stored in a file format like comma-separated values (CSV) or Parquet and processed using a batch-processing system such as Hadoop. Because static data is bounded, when each data sample has been processed, you know the job is complete. By contrast, streaming data is usually accessed through a real-time transport platform like Kafka or Kinesis and handled using a stream-processing tool such as Flink or Samza. Because the data is unbounded, processing is never complete!

One Model, Two Pipelines

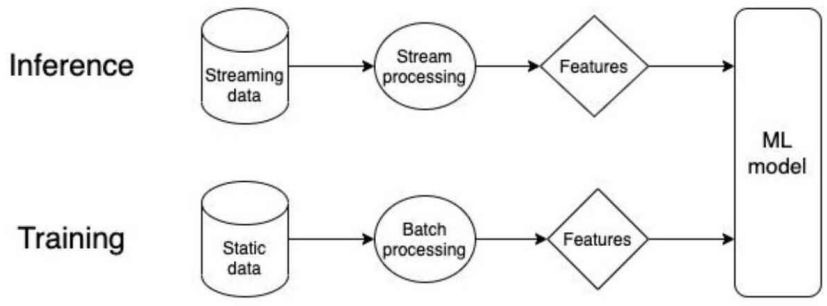
The problem with separating data into batch processing and stream processing, is that now you have two different pipelines for one ML model. First, training a model uses static data and batch processing to generate features.

During inference, however, you do online predictions with streaming data and stream processing to extract features.

This mismatch is a very common source for errors in production when a change in one pipeline isn't replicated in the pipeline. I personally have encountered that a few times. In one case, I had models that performed really well during development. When I deployed the models to production, however, the performance was poor.

To trouble-shoot this, I took a sample of data and ran it through the prediction function in the training pipeline, and then the prediction function in the inference pipeline. The two pipelines produced different results, and I eventually realized there was a mismatch between them.

The solution is to unify the batch and stream processing by using a streaming-first infrastructure. The key insight is that batch processing is a special case of streaming processing, because



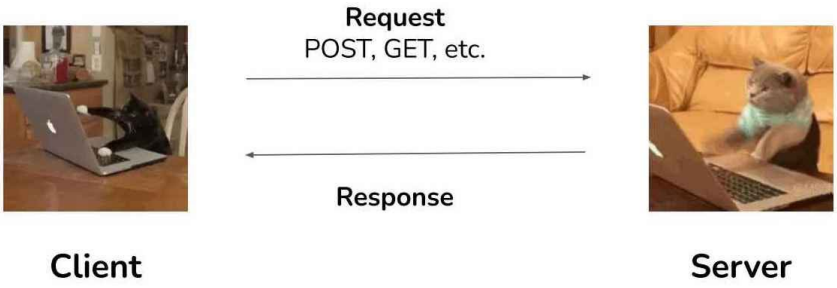
a bounded dataset is actually a special case of the unbounded data from streaming: if a system can deal with an unbounded data stream, it can work with a bounded dataset.

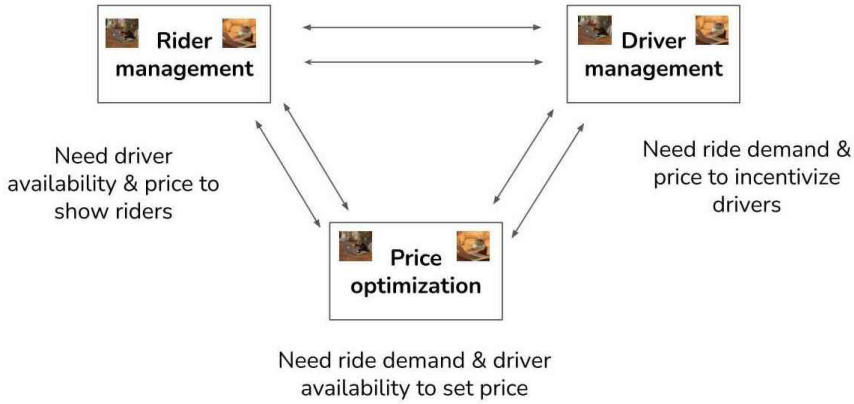
On the other hand, if a system is designed to process a bounded dataset, it's very hard to make it work with an unbounded data stream.

Request-Driven to Event-Driven Architecture

In the domain of microservices, a concept related to event-driven processing is event-driven architecture, as opposed to request-driven architecture. In the last decade, the rise of microservices is very tightly coupled with the rise of the REST API. A REST API is request driven, meaning that there is an interaction of a client and server. The client sends a request, such as a POST or GET request to the server, which returns a response. This is a synchronous operation, and the server has to be listening for the requests continuously. If the server is down, the client will keep resending new requests until it gets a response, or until it times out.

One problem that can arise when you have a lot of microservices is inter-service communications, because different services will have to send requests to each other and get information from each other. In the figure below, there are three microservices,





and there are many arrows showing the flow of information back and forth. If there are hundreds or thousands of microservices, it can be extremely complex and slow.

Another problem is how to map data transformation through the entire system. I have already mentioned how difficult it can be to understand machine models in production. To add to this complexity, you often don't have the full view of the data flow through the system, so it can be very hard for monitoring and observability.

Instead of having request-driven communications, an alternative is an event-driven architecture. Instead of services communicating directly with each other, there is a central event stream. Whenever a service wants to publish something, it pushes that information onto the stream. Other services listen to the stream, and if an event is relevant to them, then they can take it and they can produce some result, which may also be published to the stream.

It's possible for all services to publish to the same stream, and all services can also subscribe to the stream to get the information they need. The stream can be segregated into different topics, so that it's easier to find the information relevant to a service.

There are several advantages to this event-driven architecture.

First, it reduces the need for inter-service communications. Another is that because all the data transformation is now in the stream, you can just query the stream and understand how a piece of data is transformed by different services through the entire system. It's really a nice property for monitoring.

From Monitoring to Continual Learning

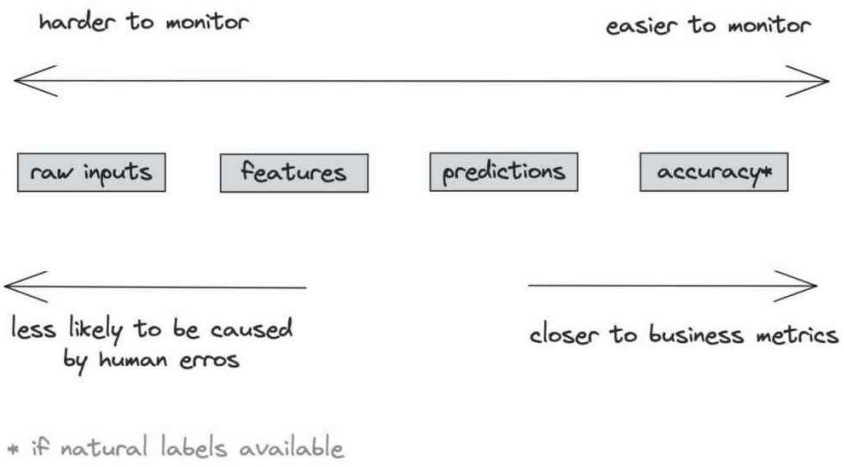
It's no secret that model performance degrades in production. There are many different reasons, but one key reason is data distribution shifts. Things change in the real world. The changes can be sudden - due to a pandemic, perhaps - or they can be cyclical. For example, ride sharing demand is probably different on the weekend compared to workdays. The change can also be gradual; for example, the way people talk slowly changes over time.

Monitoring helps you detect changing data distributions, but it's a very shallow solution, because you detect the changes…and then what? What you really want is continual learning. You want to continually adapt models to changing data distributions.

When people hear continual learning, they think about the case where you have to update the models with every incoming sample. This has several drawbacks. For one thing, models could suffer from catastrophic forgetting. Another is that it can get unnecessarily expensive. A lot of hardware backends today are built to process a lot of data at the same time, so using that to process one sample at a time would be very wasteful. Instead, a better strategy is to update models with micro-batches of 500 or 1000 samples.

Iteration Cycle

You've made an update to the model, but you shouldn't deploy the update until you have evaluated that update. In fact, with continual learning, you actually don't update the production model. Instead, you create a replica of that model, and then update that replica, which now becomes a candidate model. You only want to deploy that candidate model production after it has been evaluated.



First, you use a static data test set to do *offline evaluation*, to ensure that the model isn't doing something completely unexpected; think of this as a "smoke test." You also need to do *online evaluation*, because the whole point of continual learning is to adapt a model to change in distributions, so it doesn't make sense to test this on a stationary test set. The only way to be sure that the model is going to work is to do online evaluations.

There are a lot of ways for you to do it safely: through A/B testing, canary analysis, and multi-armed bandits. I'm especially excited about those, because they allow you to test multiple models: you treat each model as an arm of the bandit.

For continual learning, the iteration cycles can be done in order of minutes. For example, Weibo has an iteration cycle of around 10 minutes. You can see similar examples with Alibaba, TikTok, and Shein. This speed is remarkable, given the results of a recent study by Algorithmia which found that 64% of companies have cycles of a month or longer.

Continual Learning: Use Cases

There are a lot of great use cases for continual learning. First, it allows a model to adapt to rare events very quickly. As an example, Black Friday happens only once a year in the U.S., so there's no way you can have enough historical information to accurately predict how a user is going to behave on Black Friday. For the best performance you would continually train the model during Black Friday. In China, Singles' Day is a shopping holiday similar to Black Friday in the U.S., and it is one of the use cases where Alibaba is using continual learning.

Continual learning also helps you overcome the continuous cold start problem. This is when you have new users, or users get a new device, so you don't have enough historic confirmations to make predictions for them. If you can update your model during sessions, then you can actually overcome the continuous cold start problem. Within a session, you can learn what users want, even though you don't have historical data, and you can make relevant predictions. As an example, TikTok is very successful because they are able to use continual learning to adapt to users' preference within a session.

Continual learning is especially good for tasks with natural labels, for example on recommendation systems. If you show users recommendations, and they click on it, then it was a good prediction. If after a certain period of time there are no clicks, then it's a bad prediction. It is one short feedback loop, on order of minutes. This is applicable to much online content like short videos, Reddit posts, or Tweets.

However, not all recommendation systems have short feedback loops. For example, a marketplace like Stitchfix could recommend items that users might want, but would have to wait for the items to be shipped and for users to try them on, with a total cycle time of weeks before finding out if the prediction was good.

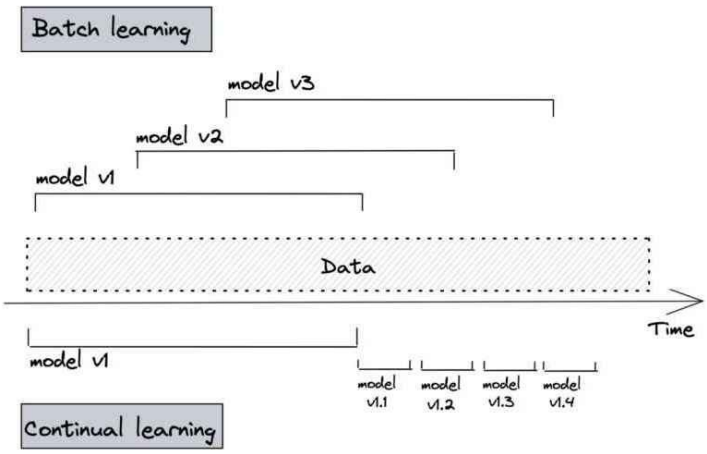
Is Continual Learning Right for You?

Continual learning sounds great, but is it right for you? First, you have to quantify the value of data freshness. People say that fresh data is better, but how much better? One thing you can do is you can try to measure how much model performance changes, if you switch from retraining monthly to weekly, to daily, or even to hourly.

For example, back in 2014, Facebook did a study that found if they went from training weekly to daily, they could increase their click-through rate by 1%, which was significant enough for them to change the pipeline to daily.

You also want to understand the value of model iteration and data iteration. Model iteration is when you make significant changes to a model architecture, and data iteration is training the same model on newer data. In theory, you can do both. In practice, the more you do of one, the fewer resources you have to spend on the other. I've seen a lot of companies that found out that data iteration actually gave them much higher return than model iteration.

You should also quantify the value of fast iterations. If you can run experiments very quickly and get feedback from the experiment quickly, then how many more experiments can you run? The more experiments you can run,



Going from monthly training to daily training gives **45x cost savings** and **+20% metrics increase**

the more likely you are to find models that work better for you and give you better return.

One problem that a lot of people are worried about is the cloud cost. Model training costs money, so you might think that the more often you train the model, the more expensive it's going to be. That's actually not always the case for continual learning.

In batch learning, when it takes longer to retrain the model, since you have to retrain the model from scratch. In continual learning, however, you just train the model more frequently, so you don't have to retrain the model from scratch; you essentially just continue training the model on fresh data. It actually requires less data and fewer compute resources.

There was a really great study from Grubhub. When they switched from monthly training to daily training, they saw a 45x savings on training compute cost. At the same time, they achieved a more than 20% increase in their evaluation metrics.

Barriers to Streaming-first Infrastructure

Streaming-first infrastructure sounds great: you can use it for online prediction and for continual learning. So why doesn't everyone use it? One reason is that many companies don't see the benefits of streaming; perhaps because their systems are not at a scale where inter-service communication has become a problem.

Another barrier is that companies simply have never tried it before. Because they've never tried that before, they don't see the benefits. It's a chicken and egg problem, because to see the benefits of streaming-first, you need to implement it.

Also, there's a high initial investment in infrastructure and many companies worry that they need employees with specialized knowledge. This may have been true in the past, but the good news is that there are so many tools being built that make it extremely easy to switch to streaming-first infrastructure.

Bet on the Future

I think it's important to make a bet in the future. Many companies are now moving to streaming-first because their metric increases have plateaued, and they know that for big metric wins they will need to try out new technology. By leveraging streaming-first infrastructure, companies can build a platform to make it easier to do real-time machine learning.

Building End-to-End Field Level Lineage for Modern Data Systems

by **Mei Tao** Product Manager, **Xuanzi Han** Senior Architect, **Helena Muñoz** Senior Software Engineer

"*What happened to this dashboard?*"

"*Where did our column go?*"

"*Why is my data ... wrong?*"

If you've been on the receiving end of these urgent messages, late night phone calls, or frantic emails from business stakeholders, you're not alone. Data engineers are no strangers to the schema changes, null values, and distribution errors that plague even the healthiest data systems. In fact, as data pipelines become increasingly complex and teams adopt more distributed architectures, such data quality issues only multiply. While data testing is an important first step when it comes to preventing these issues, data lineage has emerged as a critical component of the data pipeline root cause and impact analysis workflow. Akin to how Site Reliability Engineers or DevOps practitioners might leverage commands like git blame to understand where software broke in the context of larger systems, data lineage gives data engineering and analytics teams visibility into the health of their data at each stage in its lifecycle, from ingestion in the warehouse or lake to eventual analysis in the business intelligence layer. As part of a larger approach to data observability, lineage is critical when it comes to understanding the ins and outs of broken data.

Data lineage refers to a map of the dataset's journey throughout its lifecycle, from ingestion in the warehouse or lake to visualization in the analytics layer. In short, data lineage is a record of how data got from point A to point B. In the context of data pipelines, lineage traces the relationships across and between upstream source systems (i.e., data warehouses and data lakes) and downstream dependencies (e.g., analytics reports and dashboards), providing a holistic view of data as it evolves. Lineage also highlights the effect of system changes on associated

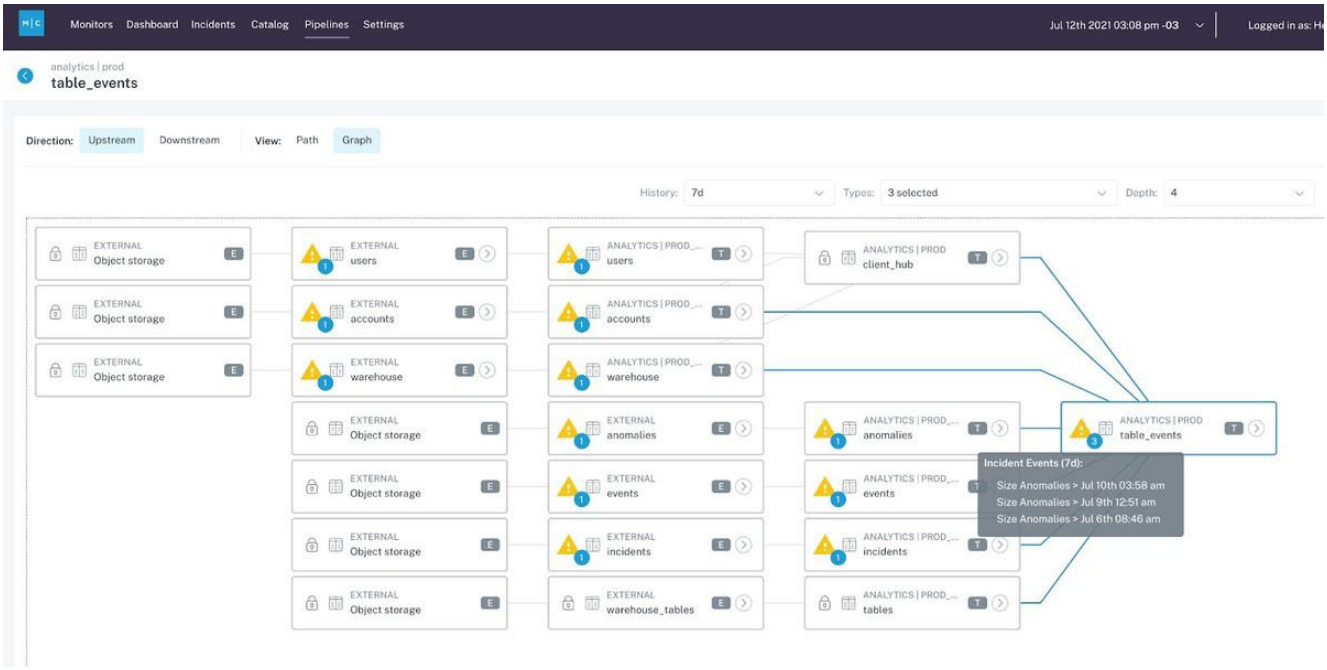


Figure 1. TABLE-LEVEL LINEAGE. Table-level lineage with upstream and downstream connections between objects in the data warehouse and tables.

assets, down to individual columns.

Due to the complexity of even the most basic SQL queries, however, building data lineage from scratch is no easy feat. Historically, lineage is parsed manually, requiring an almost encyclopedic knowledge of a company's data environment and how each component interacts with each other.

Adding further complexity to the equation, keeping manual lineage up to date becomes more challenging as companies ingest more data, layer on new solutions, and make data analysis more accessible to additional users through codeless analytics tools and other reporting software.

In this article, we walk through our journey to building field-level lineage for our customers, focusing on our backend architecture, key use cases, and best practices for data engineering teams planning to build field-level lineage for their own data systems.

High-level business requirements

Our company, Monte Carlo, is the creator of a popular data observability platform that notifies data engineers when pipelines break. We work with companies like Fox, Intuit, and Vimeo to alert them to, root cause and fix data quality issues before they become a serious problem for the business, from broken dashboards to inaccurate ML models. Such data quality issues can have big ramifications, including lost revenue, poor decision making, model drift, and wasted time.

For the past several years, data teams across industries have relied on table-level lineage to understand the root cause analysis (RCA) and impact analysis of upstream data issues on downstream tables and reports. While useful at the macro level, table-level lineage doesn't provide teams with the granularity they need to understand exactly why or how their data pipelines break.

As with building any new product functionality, the first step was to understand user requirements, and from there, scope out what could actually be delivered in a reasonable timeframe. After speaking with our customers, we determined that our solution required a few key features:

- **Fast time to value**: Our customers wanted to quickly understand the impact of code, operational, and data changes on downstream fields and reports. We needed to abstract the relationships between data objects down to the field-level; the table level was too broad for quick remediation.
- **Secure architecture**: Our customers did not want lineage to access user data or PII directly. We required an approach that would access metadata, logs, and queries, but keep the data itself in the customer's environment.
- **Automation**: Existing field-level lineage products often take a manual approach, which puts more responsibility in the lap of the customer; we realized that there was a quicker way forward, with automation, that would also update data assets based on changes in the data lifecycle.
- **Integrations with popular data tools**. We needed a knowledge graph that could automatically generate nodes across an entire data pipeline, from ingestion in the data warehouse or lake down to the business intelligence or analytics layer. Many of our customers required integration with data warehouses and lakes (Snowflake, Redshift, Databricks, and Apache Spark), transformation tools (dbt, Apache Airflow, and Prefect), and business intelligence dashboards (Looker, Tableau, and Mode), which would require us to generate every possible join and connection between every table in their data system.
- **Extraction of column-level information**. Our existing table-level lineage was mainly derived from parsing query logs, which couldn't extract parsed column information - the metadata necessary to help users understand anomalies and other issues in their data. For field-level lineage, we'd need to find a more efficient way to parse queries at scale.

Based on this basic field-level lineage, we could also aggregate the metadata in further steps to serve different use cases, such as operational analytics. For example, we could pre-calculate for a given table and each of its fields how many downstream tables are using the field. This would be particularly useful when it came to identifying the impact of data quality issues on down-stream reports and dashboards in the business intelligence layer.

After all, who doesn't want a painless root cause analysis?

Use cases

At its most basic, our field-level lineage can be used to massively reduce TTD and TTR of data quality issues, with the goal of bringing down the total amount of time it takes to customers to root cause their data pipelines. In an analytics capacity, data lineage can be used for a variety of applications, including:

- **Review suspicious numbers in a revenue report**. One of our customers, a 400-person FinTech company, generates monthly revenue forecasts using data collected and stored in Snowflake and visualized by Looker. They can use field-level lineage to trace which table in their warehouse has the source field for the "suspicious numbers in this report, and through this process, realize that the culprit for the data issue was a dbt model that failed to run.
- **Reduce data debt**. Many of our customers leverage our data observability solution to deprecate columns in frequently used data sets to ensure that outdated objects aren't being used to generate reports. Field-level lineage makes it easy for them to identify if a given column is linked to downstream reports.
- **Managing personal identifiable information (PII)**. Several of our customers deal with sensitive data, and need to know which

columns with PII are linked to destination tables in downstream dashboards. By being able to quickly connect the dots between columns with PII and user-facing dashboards, customers can remove the information or take precautions to deprecate or hide the dashboard from relevant parties. These use cases just scratch the surface of how our customers leverage field-level lineage.

By integrating it with their existing root cause analysis workflows, getting to the bottom of these questions can save time and resources for analysts and engineers across their companies.

Solution architecture

When it came to actually building field-level data lineage for our platform, the first thing we needed to do was architect a way to understand which columns belonged to which source tables. This was a particularly challenging task given that most data transformations leverage more than one data source. Further complicating matters, we needed to recursively resolve the original sources and columns in the event that some of the source tables were aliases of existing subqueries derived from other subqueries.

The sheer number of possible SQL clause combinations made it extremely difficult to cover each and every possible use case. In fact, our original prototype only covered about 80% of possible combinations. So, to cover every possible clause and clause combination across all of our data warehouse, data lake, extract, transform, load (ETL), and business intelligence integrations (dozens of tools!), we chose to individually test each clause with one another and ensure that our solution still worked as intended before moving on to the next use case.

Data model

At a foundational level, the structure of our lineage incorporates three elements:

- The destination table, stored in the downstream report
- The destination fields, stored in the destination table
- The source tables, stored in the data warehouse

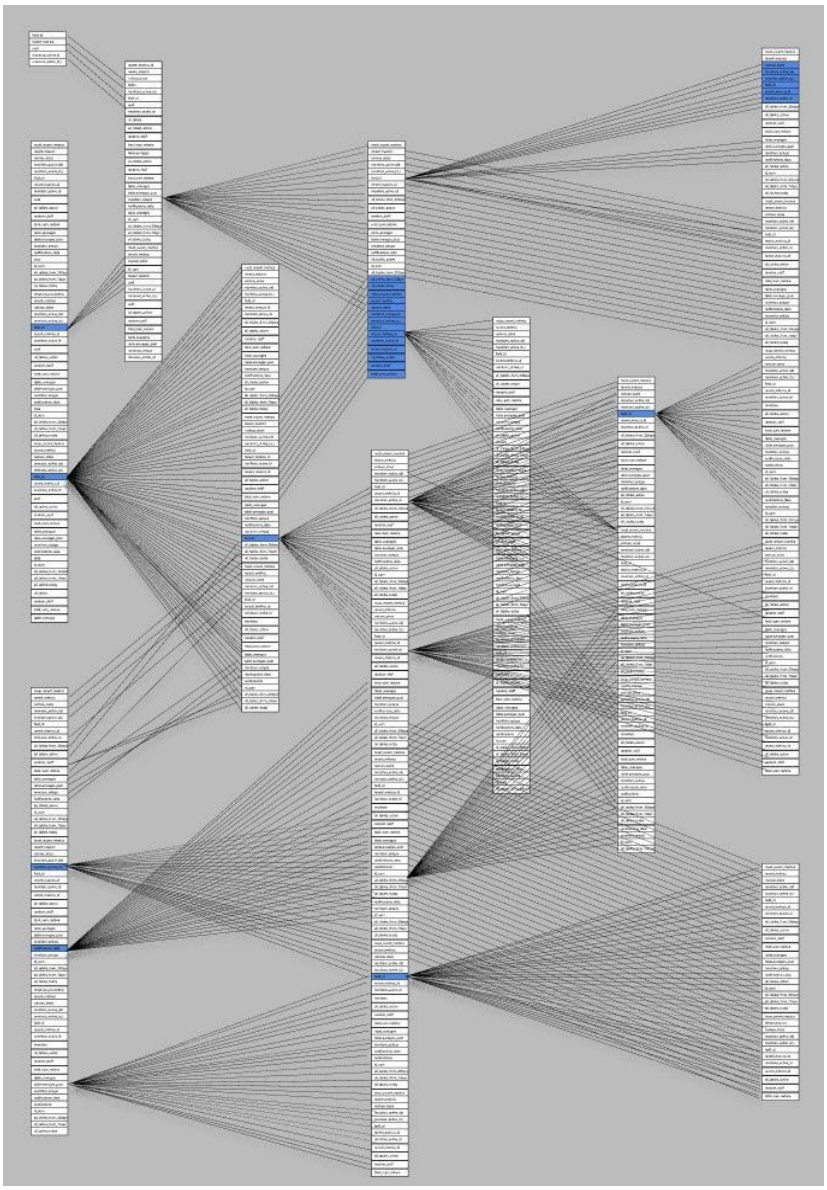


Figure 2. LINEAGE GRAPH. Field-level lineage with hundreds of connections between objects in upstream and downstream tables.

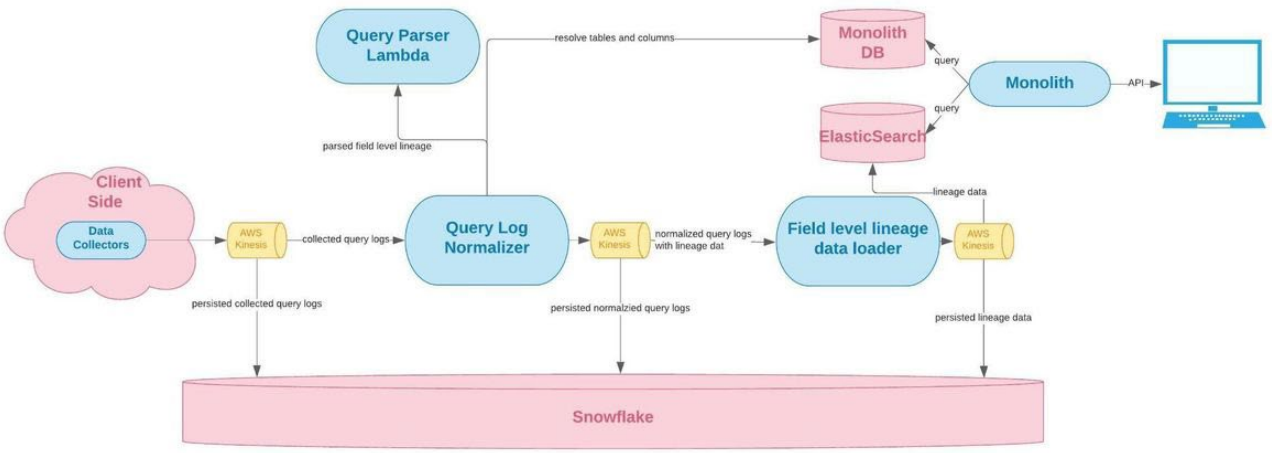


Figure 3. FIELD-LINEAGE ARCHITECTURE. The back-end architecture of our field-level lineage solution, built on top of Snowflake and Elasticsearch.

As previously mentioned, there are infinite relationships between destination and source objects, which required us to leverage a data model that was flexible enough to capture multiple queries at once.

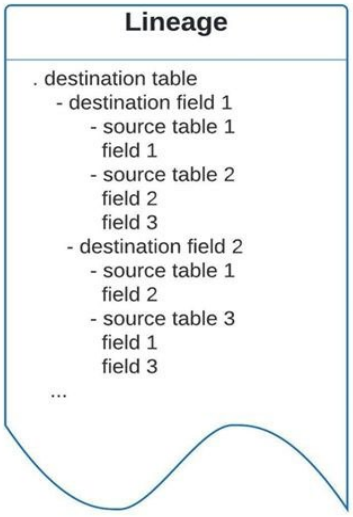


Figure 4. FIELD-LEVEL LINEAGE STRUCTURE. The structure of our field-level lineage includes several downstream destination fields per upstream table.

We decided to use a logical data model, a table_mcon ID, and hashed field-level lineage objects together as the ID for the document. For the same destination table, there could be several different queries to update it. Using the destination table mcon and hashed field-level lineage object, we would be able to capture all the different lineage combinations for a given destination table.

On Figure 5 we share one of our proposed index schemas.

In our lineage model, we have one destination table. For each of the fields in the destination table, there is a list of source tables and source columns that define the field, referred to as selected fields.

Our model also contains another list of source tables and columns containing the non-selected fields. (Figure 6)

In our case, our model incorporates one denormalized data structure which contains edges between fields in a destination table to their source fields in some source tables. (Figure 7)

Above, we offer a real example of how field-level lineage can 'simplify' a complex query. The WITH clause contains nine temporary tables, with some of the tables using other temporary tables defined before them. Additionally, in the main query, there could be joins between real tables, temporary tables declared in the WITH clause, and subqueries.

In each query or subquery's SELECT clause, there are fields that apply additional functions, expressions, and subqueries. In even more complex examples, lineage can reflect queries that have many

```
{
    "edge_id": "37d65dc5c943cab124398b2c43f0d8f2c0ff5e76a2ba3052
    "account_id": "ee7c21ae-9af9-4ce0-ac51-fa953065d6f7",
    "version": "normalized_v0.25",
    "job_ts": "2021-08-06 18:51:02.439000",
    "expire_at": "2021-08-13 18:51:02.439000",
    "destination_table_mcon":  "", // destination table mcon
    "source_table_mcons": [
      "", // mcon 1
      "", // mcon 2
    ], // adding the destination table mcon, and source table mc
    "sources": [
      {
        "table_mcon": "", // mcon of the table
        "field_name": ""
      },
      ...
    ],
    "destination_field": "new field name",
    "created_time": "2021-08-06 06:29:44.341000",
    "last_update_time": "2021-08-06 18:51:02.439000",
    "last_update_user_id": null,
    "parsed_query": ""
}
```

Figure 5. FIELD-LEVEL LINEAGE QUERY. Lineage query between a destination (analytics report) and one or more source tables in the warehouse.

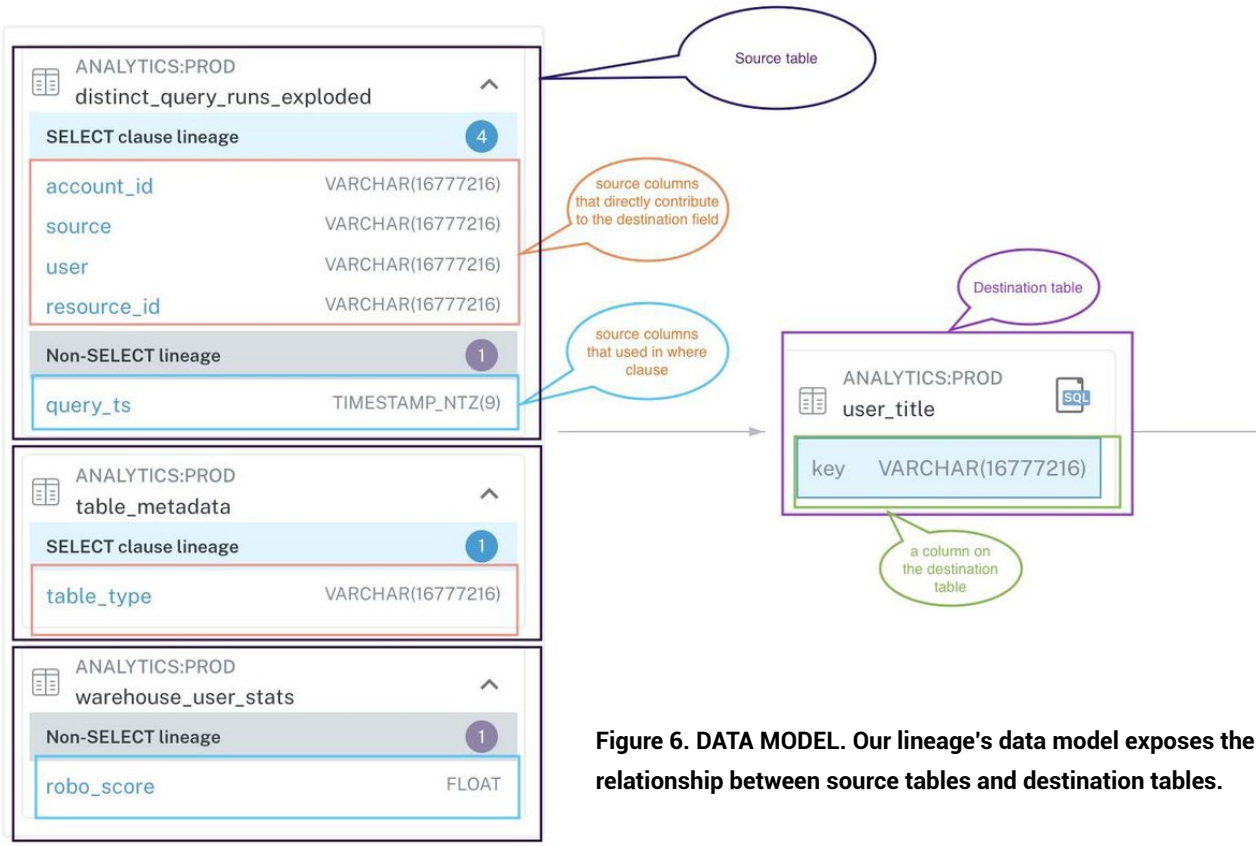


Figure 6. DATA MODEL. Our lineage's data model exposes the relationship between source tables and destination tables.

```sql
CREATE OR REPLACE TABLE decom.usage_timelines.pdt_usage_activities AS (
WITH usage_stuck_to_be_processed AS (
    SELECT s.usage_id,
        s.created_date
    FROM `decom.processed.subscriptions` s
    JOIN `decom.processed.usages` u
      ON s.usage_id = u.id
    WHERE (s.state = 'to_be_processed' AND  u.activated_at IS NOT NULL)
),
usage_subscription_state_updated as (
    SELECT *,
        rank() OVER (PARTITION BY usage_id ORDER BY created_at desc AS sub_update_no_desc
    FROM `decom.usage_timelines.usage_subscription_states` al_s
),
usages_batch_removeled as (
   select distinct u.id as usage_id
   from `decom.processed.usages` u
   join `decom.processed.subscriptions` s on u.id = s.usage_id
   left join usage_subscription_state_updated ussu on ussu.usage_id = u.id and ussu.sub_update_no_desc = 1
   where s.state = 'in_question' and ussu.to_value = 'active'
   ),
usage_subscription_state_change_actions AS (
    SELECT ussu.usage_id AS usage_id,
        CASE
          WHEN (
               ussu.from_value = 'to_be_processed' AND
               ussu.to_value = 'active' AND
               sub_update_no = 1
          ) THEN 'activate subscription'
          WHEN (
               ussu.from_value = 'to_be_processed' AND
               ussu.to_value IN ('in_question', 'disabled') AND
               sub_update_no = 1
          ) THEN 'remove from to_be_processed'
          WHEN (
               ussu.to_value IN ('in_question', 'disabled')
          ) THEN 'remove'
          WHEN (
               ussu.to_value = 'active' AND (
                   ussu.from_value IN ('in_question', 'disabled') OR
                   (from_value = 'to_be_processed' AND sub_update_no >= 3))
          ) THEN 're-activate'
          WHEN (
               ussu.to_value = 'to_be_processed'
          ) THEN 'other - to to_be_processed'
          WHEN (
               ussu.to_value = 'active'
          ) THEN 'other - to active'
          ELSE 'other change'
        END  AS action,
        ussu.created_at AS action_at
    FROM decom.usage_timelines.usage_subscription_states ussu
),
lead_subscription_orders AS (
  SELECT usage_id AS usage_id,
      CASE
        WHEN order_type = 'lead'
          THEN CAST('lead order'   AS string)
        WHEN order_type = 'regular'
          THEN CAST('regular order' AS string)
      END AS action,
      MIN(order_placed_at) AS action_at
  FROM `decom.cart.usages_orders_process`
  WHERE usage_legit_order_no = 1 AND order_placed_at IS NOT NULL
  GROUP BY
      1, 2
),
lead_subscription_order_send_dates AS
(
  SELECT usage_id                                    AS usage_id,
      CASE
        WHEN order_type = 'lead'
          THEN CAST('send lead order'   AS string)
        WHEN order_type = 'regular'
          THEN CAST('send regular order' AS string)
      END                                            AS action,
      MIN(timestamp_add(send_date, interval 10 hour)) AS action_at
  FROM `decom.cart.usages_orders_process`
  WHERE usage_legit_order_no = 1 AND send_date IS NOT NULL
  GROUP BY
      1, 2
),
submit_email AS
(
  SELECT id                                          AS usage_id,
      CAST('submit email' AS string)                 AS action,
      created_at                                     AS action_at
  FROM `decom.processed.usages`
  WHERE created_at IS NOT NULL
),
activations AS
(
  SELECT id                                          AS usage_id,
      CAST('activate' AS string)                     AS action,
      activated_at                                   AS action_at
  FROM `decom.processed.usages`
  WHERE activated_at IS NOT NULL
),
unioned AS
(
  SELECT * FROM activations
    UNION ALL
  SELECT * FROM usage_subscription_state_change_actions
    UNION ALL
  SELECT * FROM lead_subscription_orders
    UNION ALL
  SELECT * FROM lead_subscription_order_send_dates
    UNION ALL
  SELECT * FROM submit_email
)
SELECT
    u.*,
    COALESCE(usa.action_general_order, -1) AS action_general_order,
    usa.subscription_phase_change_to       AS subscription_phase_change_to
FROM unioned u
INNER JOIN (
    select usage_id,
        max(created_time) as max_created_date
    from usage_stuck_to_be_processed
    group by usage_id
) as usage_created_date
    on usage_created_date.usage_id = u.usage_id
LEFT JOIN tfddatawarehouse.cart.usage_subscription_actions usa
    ON u.action = usa.action
WHERE
    NOT (u.usage_id IN ((SELECT usage_id FROM usages_stuck_to_be_processed)) )
    and NOT (u.usage_id IN ((SELECT usage_id FROM usages_batch_removeled)) )
);
```


Figure 7. JSON QUERY. A JSON query that simplifies a complex WHERE query to identify the root cause of a data quality issue, down to the field level.

nested layers of subqueries, and even more complex expressions.

The red rectangle shows the selected fields in the lineage that derived from this query. The selected fields are the fields that define the result table.

The fields in the purple rectangle are extracted as non-selected fields. Non-selected fields have an impact on the rows to be fetched from source tables, but they don't contribute to the field values in the result table, which offers a more intuitive UI and quick root cause analysis process because unaffected lineage is obscured.

Database and deployment details

Compared to building table-level lineage, field-level lineage is far more challenging. In order to efficiently query the field level lineage data, we needed to store the field level lineage data into Elasticsearch or another graph database. Since we already leveraged Elasticsearch, we decided it would be a good fit for our use case.

We chose to store our denormalized data structure in Elasticsearch, which enables easy and flexible querying. While our Elasticsearch instance stores metadata about our customer's data, the data reflected in these lineage graphs is stored in the customer environment.

Our approach to deploying the field-level lineage incorporates two key parts: (1) a control plane managed by the Monte Carlo team, and (2) a data plane in the customer's environment.

The control plane hosts the majority of the lineage's business logic and handles insensitive metadata. It communicates with the data plane and delegates sensitive operations (such as processing, storing or deleting data) to it. The control plane also provides web and API interfaces, and monitors the health of the data plane. The control plane runs entirely in the Monte Carlo environment and typically follows a multi-tenant architecture, though some vendors offer a single-tenant control plane (often for a price premium) that runs in a customer-dedicated, completely isolated, VPC.

The data plane typically processes and stores all of the customer's sensitive data. It must be able to receive instructions from the control plane, and pass back metadata about its operations and health. The interface between the control and data plane is implemented by a thin agent that runs in the customer's environment.

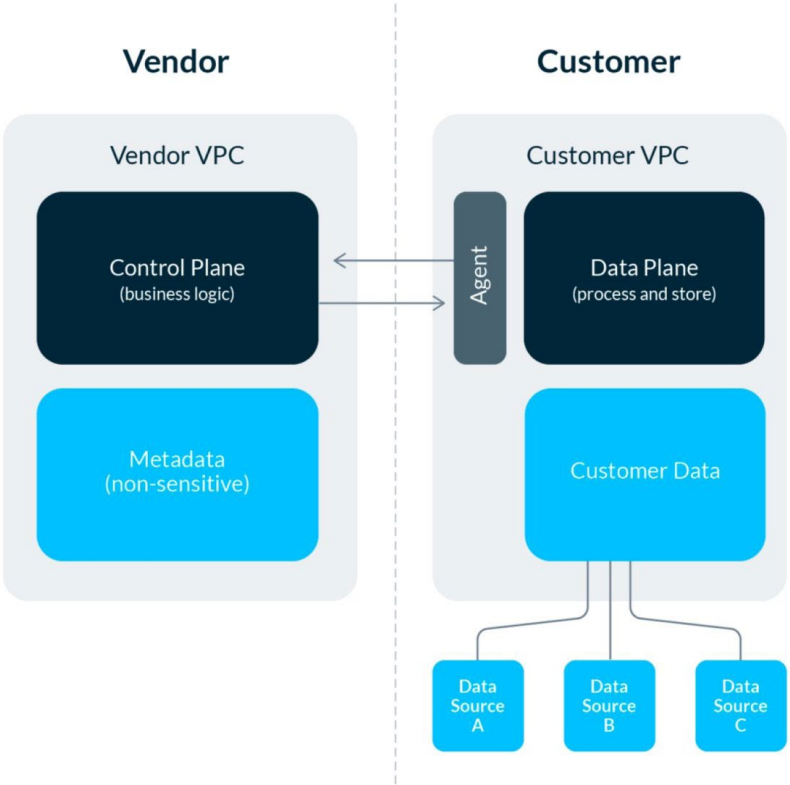


Figure 8. HYBRID DEPLOYMENT MODEL. A hybrid deployment model that allows our field-level lineage to store non-sensitive metadata and keep customer data in their environment.

At its essence, separating the customer's data from the managed software gives our customers greater agility while data compliance and security remains in the hands of the customer.

Other technologies used in our solution

To build the rest of our architecture, we leveraged multiple technologies, including:

- **AWS & Snowflake**. We built our field-level lineage functionality on top of AWS (one of our platform's public cloud hosts) and a Snowflake data warehouse. We chose Snowflake as the processing power behind our lineage due to its flexibility and compatibility with our customers' data architectures.
- **Queryparser and ANTLR**. As mentioned earlier, we leveraged a queryparser with a Java-based lambda function to parse through the query logs and generate lineage data. We support basic Redshift, BigQuery, AWS Athena, Snowflake, Hive, and Presto queries by defining the grammars in ANTLR.
- **AWS Kinesis**. Whenever we collect queries, we process them in real-time. To do this, we use Kinesis to stream the data to different components of our normalizer, so we can pass and resolve the table in real-time. We then store the resulting lineage data in Elasticsearch. We chose Elasticsearch over Postgres, Amazon DynamoDB, and other solutions due to its ability to scale and support a broader variety of queries.
- **Homegrown data collector.** We generated our automatic lineage based on queries executed at our clients' data warehouses and lakes. We utilized a data collector to extract all the query histories from the client's data center, then process them once the data enters our system.

By leveraging a mix of proprietary and open source technologies, we built a production version of field-level lineage that met the needs for detail and abstraction necessary to make root cause analysis more intuitive and seamless for our customers.

We could have taken several different paths to build our backend, and there were no obvious answers - at least at the outset. For our broader data observability architecture, we already used ANother Tool for Language Recognition (ANTLR) as a queryparser generator to read, process, execute, and translate structured text or binary files. After consulting with our broader engineering team, we decided that we could utilize ANTLR for field-level lineage as well.

By playing around with our table-level queryparser to extract the columns that were defining ANTLR grammars, we were able to access the field-level relationships for more basic SELECT clause use cases. From there, we were confident that we could build a fully functional backend.

However, we quickly ran into parsing performance issues when working with complex queries. In some cases, the queries were too long to be easily parsed: some used WITH clauses that defined some subqueries, and those subqueries were referenced in the main queries themselves. For example, if a column doesn't have quotes around it, it's parsed as a column, and if it has quotes, it's parsed as a string. To fix this, we modified the grammar of our query log parser to better support Presto, Redshift, Snowflake, and BigQuery, each with its own parsing nuances and complexities.

This SQL query complexity manifested itself in another design challenge: architecting our user interface. To build useful lineage, we needed to ensure that our solution provided enough rich context and metadata about the represented tables and fields without burdening the user with superfluous information.

Between the backend and frontend complexities, we'd need

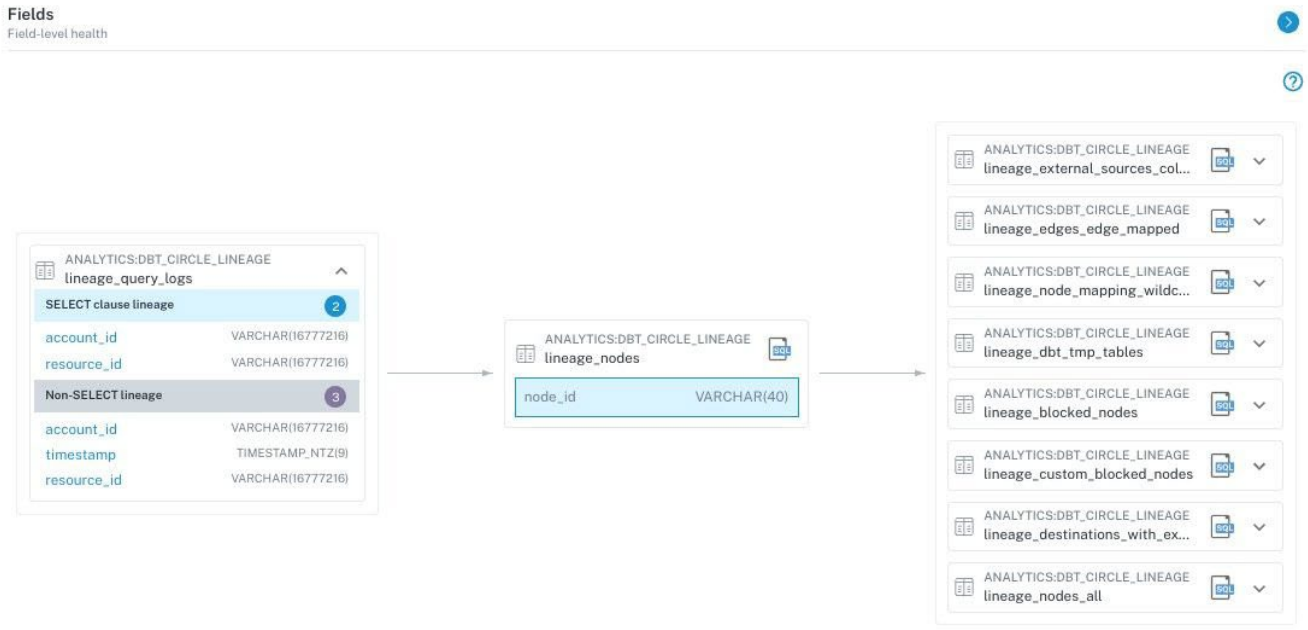


Figure 9. FIELD-LEVEL LINEAGE. Field-level lineage UI featuring the source table and the destination reports.

to abstract away this spider web of relationships and possible interactions in order to deliver on our vision of offering a truly powerful product experience for our customers. We'd need to architect a Magnolia tree with only the most relevant blossoms, leaves, and roots showing.

User interface

When it came to building the frontend interface, we needed to decide which technologies to use and determine the most useful and intuitive way to display field-level lineage to our customers. To strike the right balance between showing too much and too little information (the goldilocks zone for data lineage, as we liked to call it), we again turned to our customers for early feedback.

We also wanted to augment lineage as opposed to just automating it. In other words, we needed to determine whether we could automatically highlight what connections are likely to be relevant to their given use case or issue. After all, the most effective iteration of our product wouldn't just surface information - it would surface the right information at the right time.

To answer these questions, we released beta versions of our lineage product with the most upstream layer of information and the most downstream layer of information for several customers, asking them whether or not the feature fell within their goldilocks zone. Our larger data observability product serves data engineers and data analysts across cloud and hybrid environments, helping them manage analytical data across hundreds to thousands of sources.

Consistently, customers considered two layers (the upstream data source and the downstream data field) to be the ideal number for root cause analysis. If users wanted a full view of all the field-level relationships in a specific pipeline, they would simply need to click on the fields, then click through associated connections to those fields to traverse the field-level relationships layer by layer.

This exercise taught us something interesting: our customers care most about either A) the most downstream layer (business intelligent objects in tools like Looker or Tableau), which are business intelligence reports or B) the most upstream

layer (the source table or field stored in the warehouse or lake, which are frequently the root cause of the issue). The layers in the middle were deemed less important than middle layers (i.e., data transformations) when it came to conducting early root cause analysis.

The most downstream layer, BI reports and dashboards, are the end products that data consumers use for their day-to-day work. This is important because customers want to know the direct impact on the consumer facing data products, so they know how to communicate with end users, i.e. “Hey finance analyst, don't use this number, it's outdated”.

The most relevant upstream layer, the source table in the warehouse or lake, is leveraged when customers trace the lineage layer by layer and find that one upstream layer with the table/field has a data quality issue. Once they find it, then they can fix the issue in that table / field and solve the problem.

To write the field-level lineage UI, we created a reusable component using React TypeScript to make sure we could easily port the field lineage UI to other parts of the product or even to other MC products in the future. To display both the upstream and downstream lists, we used React Virtuoso, a React framework that makes it easy to visualize large data sets.

Since fields could potentially have tens of upstream or downstream tables, which in turn could have hundreds or thousands of fields, rendering all of those components without affecting the performance of the app was crucial. Virtuoso, which supports lists with items of different sizes, gave us the flexibility to only render the relevant components for a given lineage graph.

How our field-level lineage works

Currently, our product displays two types of field relationships:

- **SELECT clause lineage**: field relationships defined by the SQL clause SELECT; these are field-to-field relationships where a change in an upstream field directly changes the downstream field.
- **Non-SELECT lineage**: field relationships defined by all other SQL clauses, e.g., WHERE; these are field-to-

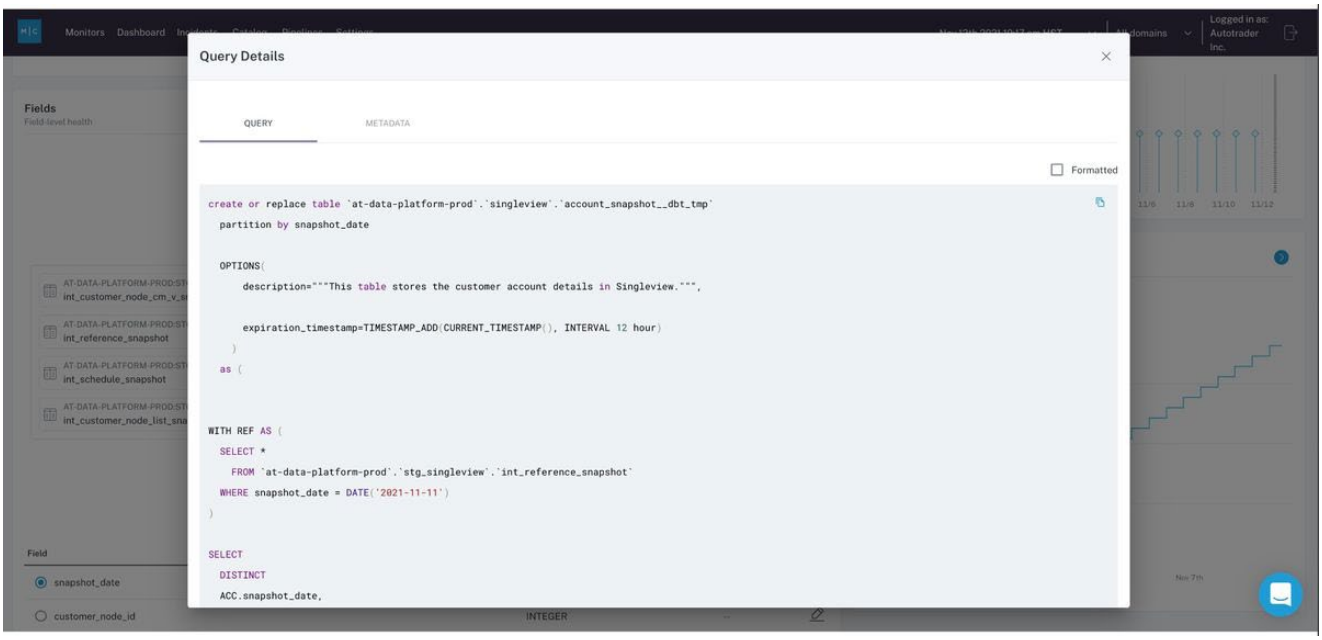


Figure 10. LINEAGE QUERY DETAILS. A lineage query that helps users understand table-to-field relationships.

table relationships, where the downstream fields are often shaped by a filtering or ordering logic defined by upstream fields.

A chosen field's upstream non-SELECT lineage fields display as the filtering/ordering fields that result in the chosen field. Its downstream non-SELECT lineage fields are the resulting fields from the filtering/ordering logic defined by the chosen field.

Lessons learned

Like any agile development process, our experience building field-level lineage was an exercise in prioritization, listening, and quick iteration. Aside from some of the technical knowledge we gained about how to abstract away query log complexity and design ANTLR frameworks, the vast majority of our learnings can be applied to the development of nearly any data software product:

- **Listen to your teammates** and take everyone's advice into consideration. When we began to work with the query parser, we underestimated how challenging it would be to parse queries. One of our teammates had previous experience working with the parser, warned us of its quirks, and suggested we might build a different one to complete the task. If we had listened to our teammate early on, we would have saved a good chunk of time.
- **Invest in prototyping**: Our customers are our north star. Whenever we create new product features, we take care to consider their opinions and preferences. Doing so effectively requires sharing a product prototype. To speed up the feedback cycle and make these interactions more useful for both parties, we shipped an early prototype to some of our most enthusiastic champions within weeks of development. While this first iteration was not perfect, it allowed us to demonstrate our commitment to meeting customer demands and gave us some early guidance as we further refined the product.
- **Ship and iterate**: This is a common practice in the software engineering world and something we take very seriously. Time is of the essence, and when we prioritize one project we have to ensure we are optimizing the time of everyone who is involved in that project. When we began working on this feature, we didn't have time to make our product "perfect" before showing our prototype to customers - and moving forward without perfection allowed us to expedite development. Our repeatable process included building out the functionality of the feature, showing it off to our customers, asking for feedback, then making improvements and or changes where needed.

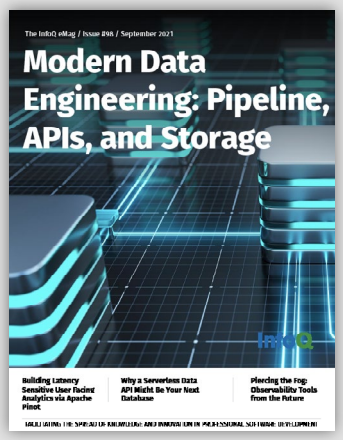
We predict that more and more data teams will start adopting automatic lineage and other knowledge graphs to identify the source of data quality issues as the market matures and data engineering teams look to increase efficiencies and reduce downtime.

Until then, here's wishing you as few "what happened to my data?!" fire drills as possible.

Read recent issues

The InfoQ Trends Report 2021

In this eMag, you will be introduced to the paths to production and how several global companies supercharge developers and keep their competitiveness by balancing speed and safety.



Modern Data Engineering: Pipeline, APIs, and Storage

In this edition of the Modern Data Engineering eMag, we'll explore the ways in which data engineering has changed in the last few years. Data engineering has now become key to the success of products and companies. And new requirements breed new solutions.

Modern Data Engineering 2021

Data engineers and software architects will benefit from the guidance of the experts in this eMag as they discuss various aspects of breaking down traditional silos that defined where data lived, how data systems were built and managed, and how data flows in and out of the system.