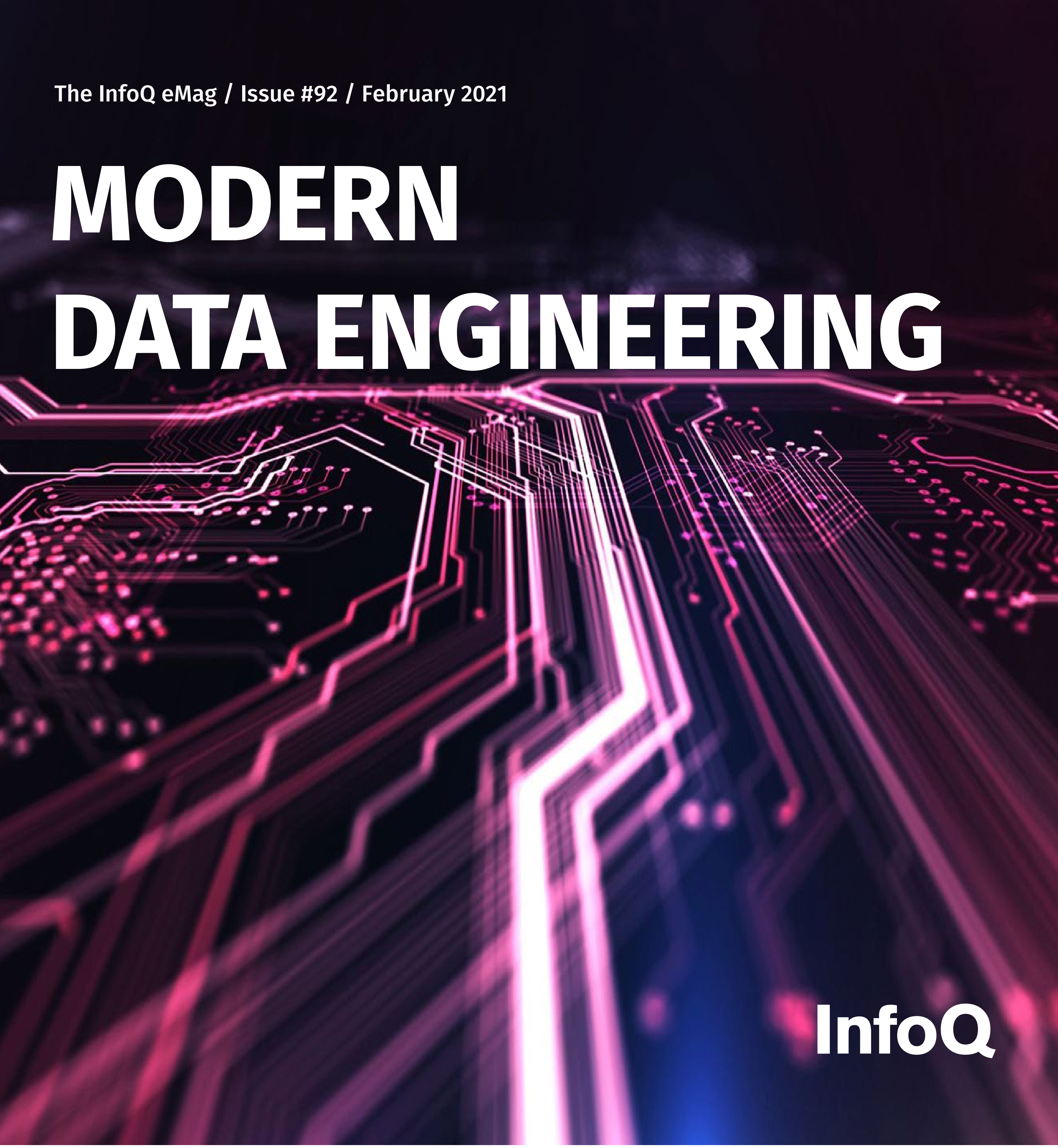
The InfoQ eMag / Issue #92 / February 2021

MODERN DATA ENGINEERING



InfoQ

The Future of Data Engineering

Beyond the Database, and beyond the Stream Processor: What's the Next Step for Data Management?

Combining DataOps and DevOps: Scale at Speed

FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

InfoQ @InfoQ InfoQ InfoQ

Modern Data Engineering

IN THIS ISSUE

The Future of Data Engineering 06

Data Gateways in the Cloud Native Era 32

Beyond the Database, and beyond the Stream Processor: What's the Next Step for Data Management? 24

Combining DataOps and DevOps: Scale at Speed 39

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

CONTRIBUTORS

Chris Riccomini

is a software engineer with more than a decade of experience at Silicon Valley tech companies including PayPal, LinkedIn, and WePay, a JPMorgan Chase company. Riccomini is active in open source. He is co-author of Apache Samza, a stream-processing framework, and is also a member of the Apache project management committee (PMC), Apache Airflow PMC, and Apache Samza PMC. He is a strategic investor and advisor for startups in the data space, where he advises founders and technical leaders on product and engineering strategy.

Ben Stopford

is Lead Technologist, Office of the CTO at Confluent (a company that backs Apache Kafka). He has worked on a wide range of projects from implementing the latest version of Kafka's replication protocol to assessing and shaping Confluent's strategy. He is the author of the book Designing Event-Driven Systems, O'Reilly, 2018.

Bilgin Ibryam

is a product manager at Red Hat, committer and member of Apache Software Foundation. He is an open source evangelist, blogger, occasional speaker, and the author of Kubernetes Patterns and Camel Design Patterns books. In his day-to-day job, Bilgin enjoys mentoring, coding and leading developers to be successful with building open source solutions. His current work focuses on blockchain, distributed systems, microservices, devops, and cloud-native application development.

Sam Bocetta

is a former security analyst, having spent the bulk of his as a network engineer for the Navy. He is now semi-retired, and educates the public about security and privacy technology. Much of Sam's work involved penetration testing ballistic systems. He analyzed our networks looking for entry points, then created security-vulnerability assessments based on my findings.

Thomas Betts

is the Lead Editor for Architecture and Design at InfoQ, and a Sr. Principal Software Engineer at Blackbaud. For over two decades, his focus has always been on providing software solutions that delight his customers. He has worked in a variety of industries, including retail, finance, health care, defense and travel. Thomas lives in Denver with his wife and son, and they love hiking and otherwise exploring beautiful Colorado.

A LETTER FROM THE EDITOR

Data architecture is being disrupted, echoing the evolution of software architecture over the past decade.

The changes coming to data engineering will look and sound familiar to those who have watched monoliths be broken up into microservices: DevOps to DataOps; API Gateway to Data Gateway; Service Mesh to Data Mesh. While this will have benefits in agility and productivity, it will come with a cost of understanding and supporting a next-generation data architecture.

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.

Future of Data Engineering

In his QCon presentation on The Future of Data Engineering, Chris Riccomini focuses on data pipelines and how they mature from batch to real-time processing, which leads to more integration and a need for automation. He advocates for a move from a monolith to micro-warehouses. Because the job of a data engineer is to help an organization move and process data, that means creating self-service tools that enable producers and consumers of data to define the pipelines they need.

Beyond the Database, and beyond the Stream Processor: What's the Next Step for Data Management?

Looking beyond the database and stream processor, Ben Stopford asks, “What's the next step for data management?” The shift from traditional databases to stream processing has fundamentally changed the interaction model, from passive data to active data. Modern applications need to effectively work with both these models, and that means rethinking what a database is, what it means to us, and how we interact with both the data it contains and the event streams that connect it all together.

Data Gateways in the Cloud Native Era

Just as API gateways are necessary to tame microservices, data gateways focus on the data aspect to offer abstractions, security, scaling, federation, and contract-driven development features. In his article, Data Gateways in the Cloud Native Era, Bilgin Ibryam says the freedom for microservices to use the most suitable database leads to a polyglot persistence layer, which necessitates advanced gateway capabilities. There is no one-size-fits-all option for a data gateway, and Ibryam covers several features and various products to evaluate based on your needs.

Combining DataOps and DevOps: Scale at Speed

With all the new tools available and ideas of how to improve data engineering, Sam Bocetta looks at how companies can adapt their business models so that they are better able to process, analyze, and derive value from big data.

In Combining DataOps and DevOps: Scale at Speed, he says development teams need consistent access to high-quality data. Companies need to realize that data is a vital commodity, and embrace a data-centric view of the enterprise, rather than organizing teams around the tools and technologies used to manage data.

When done correctly, DataOps provides a cultural transformation that promotes communication between all data stakeholders.

The Future of Data Engineering

by **Chris Riccomini**, Software Engineer

“The future of data engineering” is a fancy title for presenting stages of data pipeline maturity and building out a sample architecture as I progress, until I land on a modern data architecture and data pipeline. I will also hint at where things are headed for the next couple of years.

It's important to know me and my perspective when I'm predicting the future, so that you can couch mine with your own perspectives and act accordingly.

I work at WePay, which is a payment-processing company. JPMorgan Chase acquired the company a few years ago. I work on data infrastructure and data engineering. Our stack is Airflow, Kafka, and BigQuery, for the most part. Airflow is, of course, a job scheduler that kicks off jobs and does workflow things. BigQuery is a data warehouse hosted by Google Cloud. I make some references to Google Cloud services here, and you can definitely swap them with the corresponding AWS or Azure services.

We, at WePay, use Kafka a lot. I spent about seven years at LinkedIn, the birthplace of Kafka, which is a pub/sub, write-ahead log. Kafka has become the backbone of a log-based architecture. At LinkedIn, I spent a bunch of time doing everything from data science to service infrastructure, and so on. I also wrote Apache Samza, which is a stream-processing system, and helped build out their Hadoop ecosystem. Before that, I spent time as a data scientist at PayPal.

There are many definitions for “data engineering”. I've seen people use it when talking about business analytics and in the context of data science. I'm going to throw down my definition: a data engineer's job is to help an organization move and process data. To move data means streaming pipelines or data pipelines; to process data means data warehouses and stream processing. Usually, we're focused on asynchronous, batch or streaming stuff as opposed to synchronous real-time things.

I want to call out the key word here: "help". Data engineers are not supposed to be moving and processing the data themselves but are supposed to be helping the organization do that.

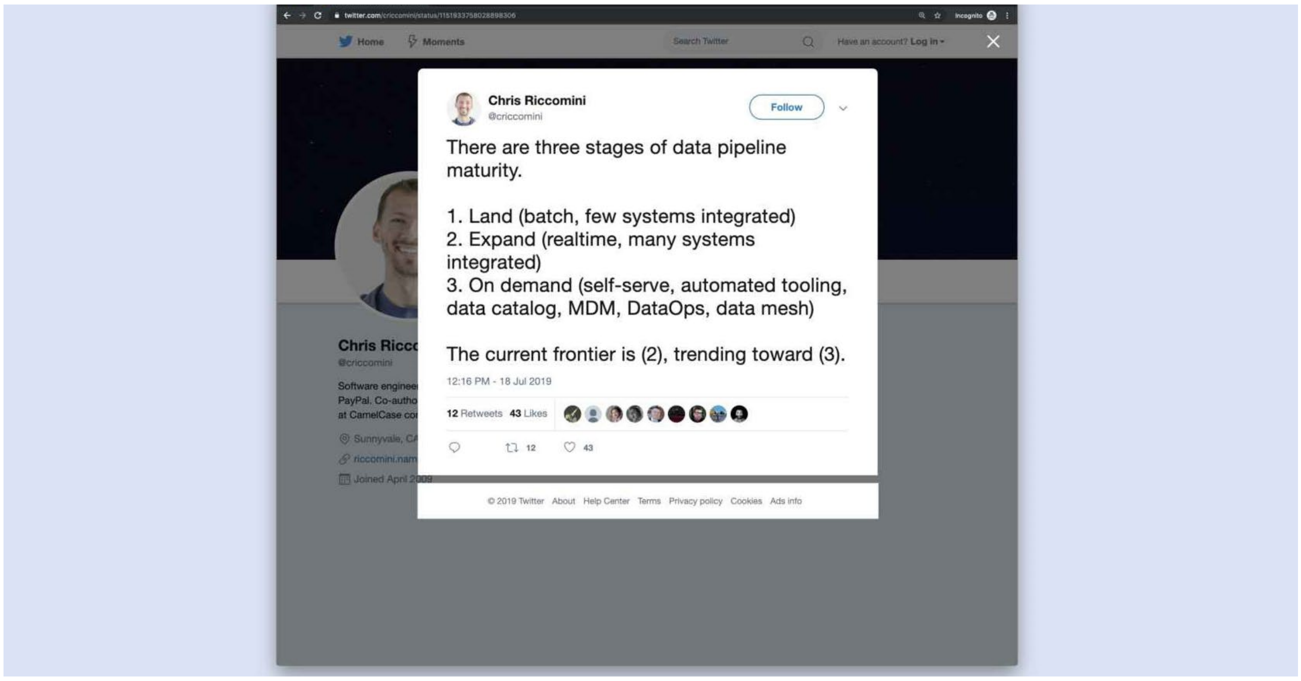
Maxime Beauchemin is a prolific engineer who started out, I think, at Yahoo and passed through Facebook, Airbnb, and Lyft. Over the course of his adventures, he wrote Airflow, which is the job scheduler that we and a bunch of other companies use. He also wrote Superset. In his "The Rise of the Data Engineer" blog post a few years ago, Beauchemin said that "... data engineers build tools, infrastructure, frameworks, and services." This is how we go about helping the organization to move and process the data.

The reason that I put this presentation together was a 2019 blog post from a company called Ada in which they talk about their journey to set up a data warehouse. They had a MongoDB database and were starting to run up against its limits when it came to reporting and some ad hoc query things. Eventually, they landed on Apache Airflow and Redshift, which is AWS's data-warehousing solution.

What struck me about the Ada post was how much it looked like a post that I'd written about three years earlier. When I landed at WePay, they didn't have much of a data warehouse and so we went through almost the exact same exercise that Ada did. We eventually landed on Airflow and BigQuery, which is Google Cloud's version of Redshift. The Ada post and mine are almost identical, from the diagrams to even the structure and sections of the post.

This was something we had done a few years earlier and so I threw down the gauntlet on Twitter and predicted Ada's future. I claimed to know how they would progress as they continued to build out their data warehouse: one step would be to go from batch to a real-time pipeline, and the next step would be to a fully self-serve or automated pipeline.

I'm not trying to pick on Ada. I think it's a perfectly reasonable solution. I just think that there's a natural evolution of a data pipeline and a data warehouse and the modern data ecosystem


Chris Riccomini
@criccomini

There are three stages of data pipeline maturity.

1. Land (batch, few systems integrated)
2. Expand (realtime, many systems integrated)
3. On demand (self-serve, automated tooling, data catalog, MDM, DataOps, data mesh)

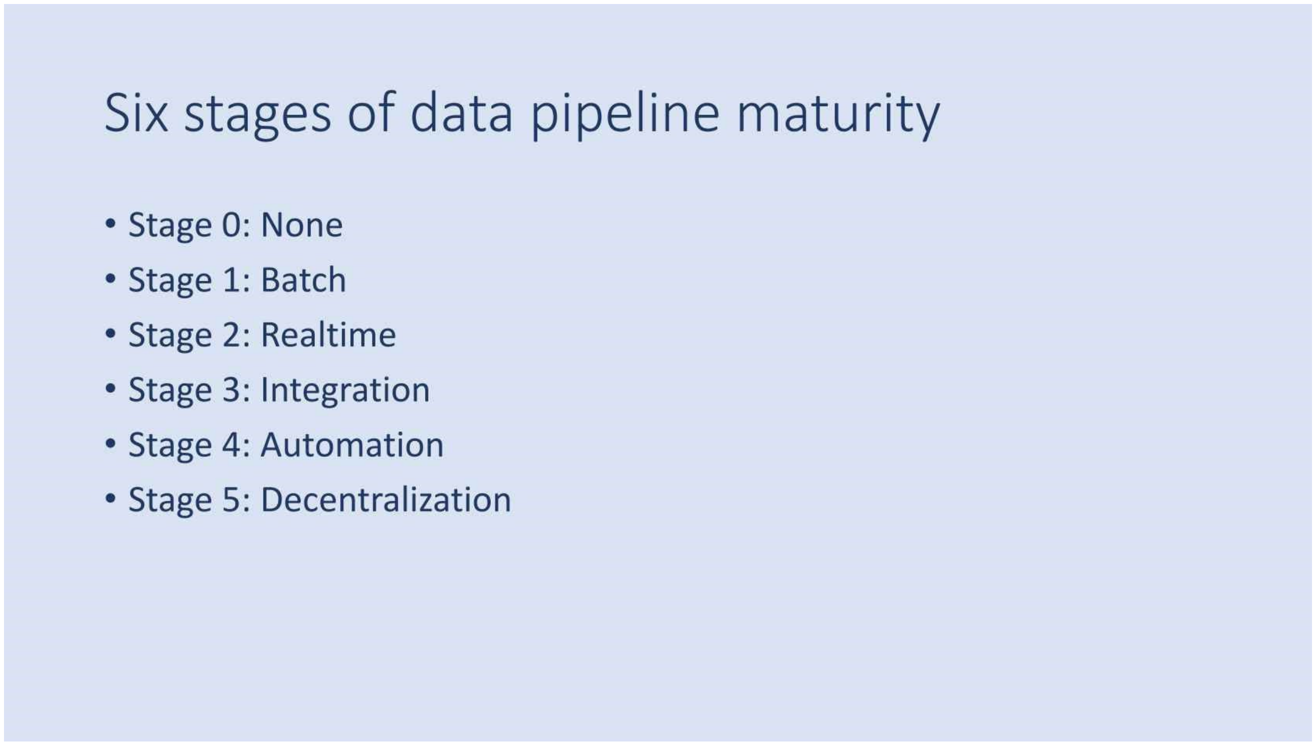
The current frontier is (2), trending toward (3).

12:16 PM - 18 Jul 2019

12 Retweets 43 Likes

Home
Moments
Search Twitter
Have an account? Log in
Follow
© 2019 Twitter About Help Center Terms Privacy policy Cookies Ads info

Figure 1: Getting cute with land/expand/on demand for pipeline evolution



Six stages of data pipeline maturity

- Stage 0: None
- Stage 1: Batch
- Stage 2: Realtime
- Stage 3: Integration
- Stage 4: Automation
- Stage 5: Decentralization

Figure 2: The six stages of data-pipeline maturity

and that's really what I want to cover here.

I refined this idea with the tweet in Figure 1. The idea was that we initially land with nothing, so we need to set up a data warehouse quickly. Then we expand as we do more integrations, and maybe we go to real-time because we›ve got Kafka in our ecosystem.

Finally, we move to automation for on-demand stuff. That eventually led to my "The Future of Data Engineering" post in which I discussed four future trends.

The first trend is timeliness, going from this batch-based periodic architecture to a more real-time architecture. The second is connectivity; once we go down the timeliness route, we start doing more integration with other systems. The last two tie together: automation and decentralization. On the automation front, I think we need to start thinking about how we operate not just our operations but our data management. And then decentralizing the data warehouse.

I designed a hierarchy of data-pipeline progression. Organizations go through this evolution in sequence.

The reason I created this path is that everyone's future is different because everyone is at a different point in their life cycle.

The future at Ada looks very different than the future at WePay because WePay may be farther along on some dimensions - and then there are companies that are even farther along than WePay.

These stages let you find your current starting point and build your own roadmap from there.

Stage 0: None

You're probably at this stage if you have no data warehouse. You probably have a monolithic architecture.

You're maybe a smaller company and you need a warehouse up and running now. You probably don't have too many data engineers and so you're doing this on the side.

Stage 0 looks like Figure 3, with a lovely monolith and a database. You take a user and you attach it to the database.

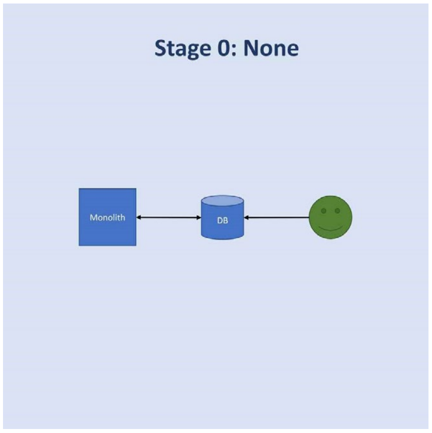


Figure 3: Stage 0 of data-pipeline maturity

This sounds crazy to people that have been in the data-warehouse world for a while but it's a viable solution when you need to get things quickly up and running. The data appears to the user as basically real-time data because you're reading directly from the database. It's easy and cheap.

This is where WePay was when I landed there in 2014. We had a PHP monolith and a monolithic MySQL database. The users we had, though, weren't happy and things were starting to tip over. We had queries timing out. We had users impacting each other - most OLTP systems that you're going to be using do not have strong isolation and multi-tenancy so users can really get in each other's way.

Because we were using MySQL, we were missing some of the fancier analytic SQL stuff that our data-science and business-analytics people wanted and report generation was starting to break. It was a pretty normal story.

Stage 1: Batch

We started down the batch path, and this is where the Ada post and my earlier post come in.

Going into Stage 1, you probably have a monolithic architecture. You might be starting to lean away from that but usually it works best when you have relatively few sources. Data engineering is now probably your part-time job. Queries are timing out because you're exceeding the database capacity, whether in space, memory, or CPU.

The lack of complex analytical SQL functions is becoming an issue for your organization as people need those for customer-facing or internal reports. People are asking for charts, business intelligence, and all that kind of fun stuff.

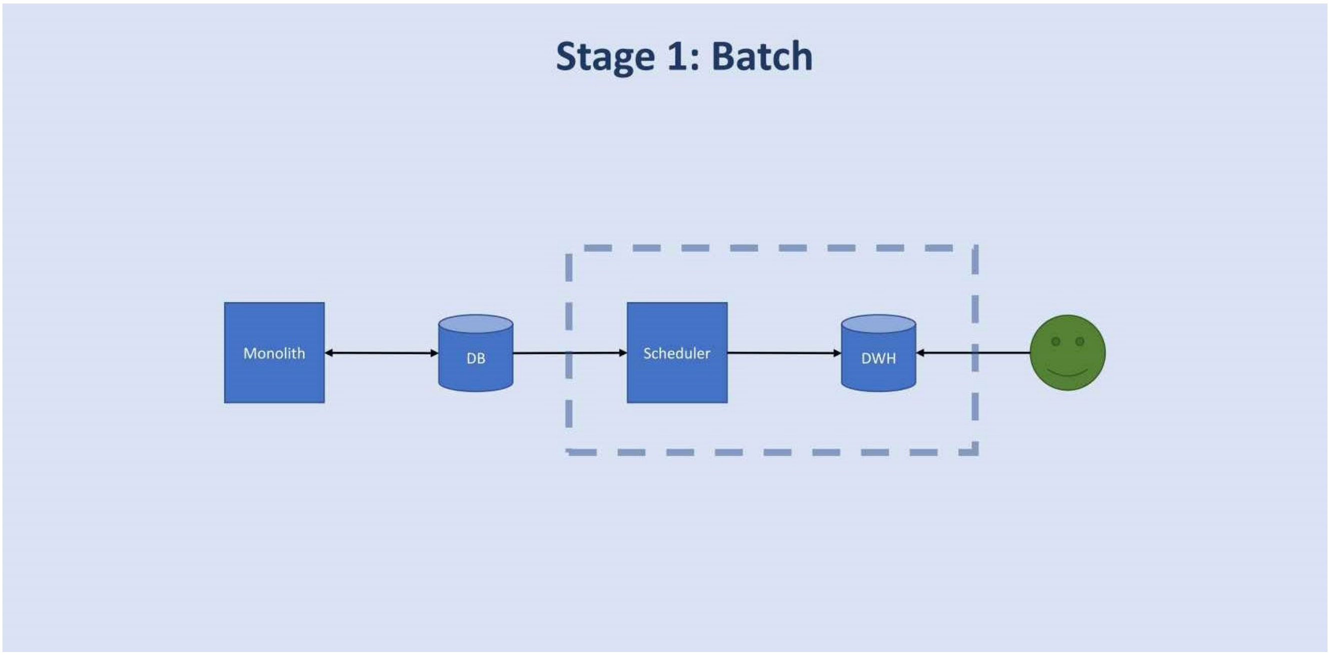


Figure 4: Stage 1 is the classic batch-based approach to data

This is where the classic batch-based approach comes in. Between the database and the user, you stuff a data warehouse that can accomplish a lot more OLAP and fulfill analytic needs. To get data from the database into that data warehouse, you have a scheduler that periodically wakes up to suck in the data.

That's where WePay was at about a year after I joined. This architecture is fantastic in terms of tradeoffs. You can get the pipeline up pretty quickly these days - when I did it in 2016, it took a couple of weeks. Our data latency was about 15 minutes, so we did incremental partition loads, taking little chunks of data and loading them in. We were running a few hundred tables. This is a nice place to start if you're trying to get something up and running but, of course, you outgrow it.

The number of Airflow workflows that we had went from a few hundred to a few thousand. We started running tens or hundreds of thousands of tasks per day, and that became an operational issue because of the probability that some of those are not going to work. We also discovered - and this is not intuitive for people who haven't run complex data pipelines - that the incremental or batch-based approach requires imposing dependencies or requirements on the schemas of the data that you're loading. We had issues with `create_time` and `modify_time` and ORMs doing things in different ways and it got a little complicated for us.

DBAs were impacting our workload; they could do something that hurt the replica that we're reading off of and cause latency issues, which in turn could cause us to miss data. Hard deletes weren't propagating - and this is a big problem if you have people who delete data from your database. Removing a row or a table or whatever can cause problems with batch loads because you just don't know when the data disappears. Also, MySQL replication latency was affecting our data quality and periodic loads would cause occasional MySQL timeouts on our workflow.

Stage 2: Realtime

This is where real-time data processing kicks off. This approaches the cusp of the modern era of real-time data architecture and it deserves a closer look than the first two stages.

You might be ready for Stage 2 if your load times are taking too long. You've got pipelines that are no longer stable, whether because workflows are failing

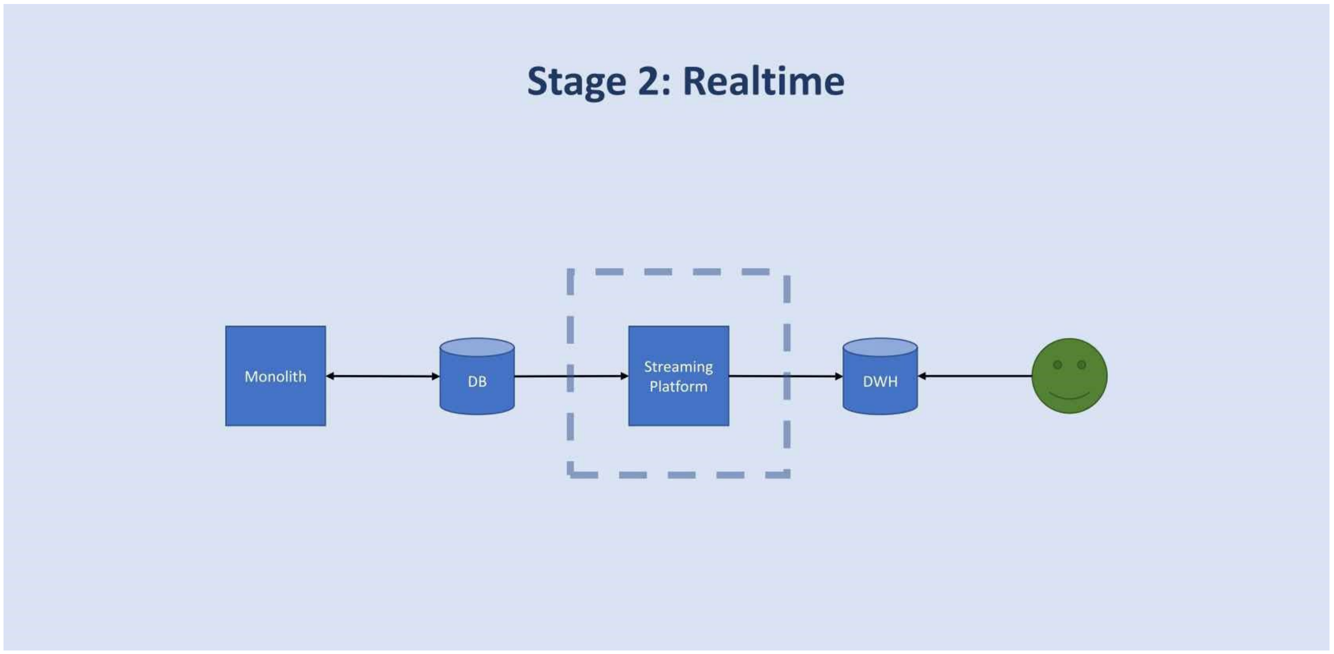


Figure 5: Stage 2 of data-pipeline maturity

or your RDBMS is having trouble serving the data. You've got complicated workflows and data latency is becoming a bigger problem: maybe the 15-minute jobs you started with in 2014 are now taking an hour or a day, and the people using them aren't happy about it. Data engineering is probably your full-time job now.

Your ecosystem might have something like Apache Kafka floating around. Maybe the operations folks have spun it up to do log aggregation and run some operational metrics over it; maybe some web services are communicating via Kafka to do some queuing or asynchronous processing.

From a data-pipeline perspective, this is the time to get rid of that batch processor for ETL purposes and replace it with a streaming platform. That's what WePay did. We changed our ETL pipeline from Airflow to Debezium and a few other systems, so it started to look like Figure 6.

The hatched Airflow box now contains five boxes, and we're talking about many machines so the operational complexity has gone up. In exchange, we get a real-time pipeline.

Kafka is a write-ahead log that we can send messages to (they get appended to the end of the log) and we can have consumers reading from various locations in that log. It's a sequential read and sequential write kind of thing.

We use it with the upstream connectors. Kafka has a component called Kafka Connect. We heavily use Debezium, a change-data-capture (CDC) connector that reads data from MySQL in real time and funnels it in real time into Kafka.

CDC is essentially a way to replicate data from one data source to others. Wikipedia's fancy definition of CDC is "… the identification, capture, and delivery of the changes made to the enterprise data sources." A concrete example is what something like Debezium will do with a MySQL database. When I insert a row, update that row, and later delete that row, the CDC feed will give me three different events: an insert, the update, and the delete. In some cases, it will

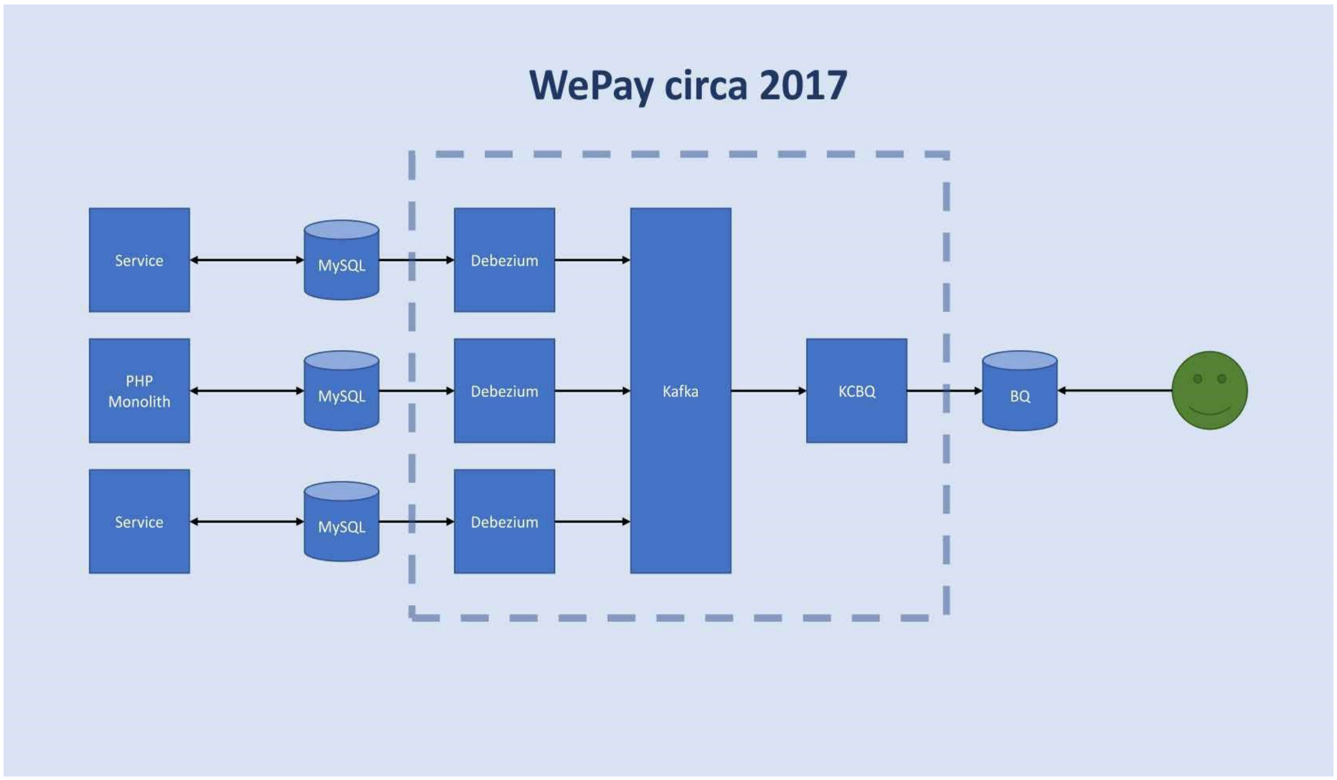
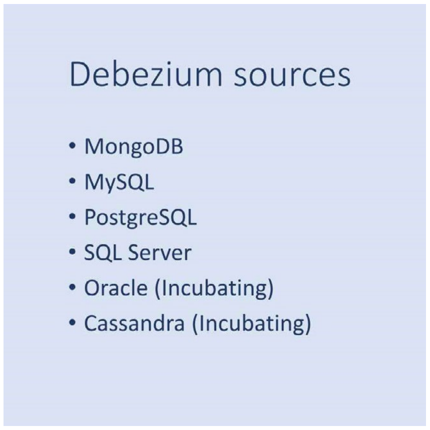


Figure 6: WePay's data architecture in 2017

also provide the before and the after states of that row. As you can imagine, this can be useful if you're building out a data warehouse.

Debezium can use a bunch of sources. We use MySQL, as I mentioned. One of the things in that Ada post that caught my eye was the fact that they were using MongoDB - sure enough, Debezium has a MongoDB connector. We contributed a Cassandra connector to Debezium a couple of months ago. It's incubating and we're still getting up off the ground with it ourselves but that's something that we're going to be using heavily in the near future.


Debezium sources

- MongoDB
- MySQL
- PostgreSQL
- SQL Server
- Oracle (Incubating)
- Cassandra (Incubating)

Figure 7: Debezium sources

Last but not least in our architecture, we have KCBQ, which stands for Kafka Connect BigQuery (I do not name things creatively). This connector takes data from Kafka and loads it into BigQuery. The cool thing about this, though, is that it leverages BigQuery's real-time streaming insert API.

One of the cool things about BigQuery is that you can use its RESTful API to post data into the data warehouse in real time and it's visible almost immediately. That gives us a latency from our production database to our data warehouse of a couple of seconds.

This pattern opens up a lot of use cases. It lets you do real-time metrics and business intelligence off of your data warehouse. It also allows you to debug, which is not immediately obvious - if your engineers need to see the state of their database in production right now, being able to go to the data warehouse to expose that to them so that they can figure out what's going on with their system with essentially a real-time view is pretty handy.

You can also do some fancy monitoring with it. You can impose assertions about what the shape of the data in the database should look like so that you can be satisfied that the data warehouse and the underlying web service itself are healthy.

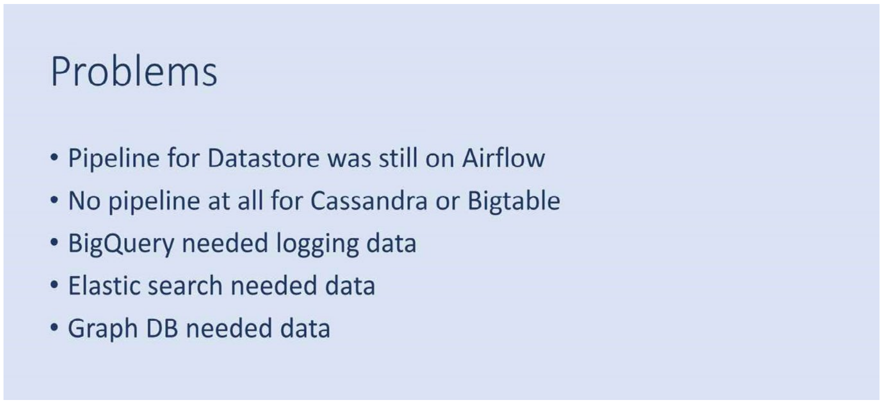
Figure 8 shows some of the inevitable problems we encountered in this migration. Not all of our connectors were on this pipeline, so we found ourselves between the new cool stuff and the older painful stuff.

Datastore is a Google Cloud system that we were using; that was still Airflow-based.

Cassandra didn't have a connector and neither did Bigtable, which is a Google Cloud equivalent of HBase.

We had BigQuery but BigQuery needed more than just our primary OLTP data; it needed logging and metrics. We had Elasticsearch and this fancy graph database (which we're going to be open-sourcing soon) that also needed data.

The ecosystem was looking more


Problems

- Pipeline for Datastore was still on Airflow
- No pipeline at all for Cassandra or Bigtable
- BigQuery needed logging data
- Elastic search needed data
- Graph DB needed data

Figure 8: Problems at WePay in the migration to Stage 2

complicated. We're no longer talking about this little monolithic database but about something like Figure 9, which comes from Confluent and is pretty accurate.

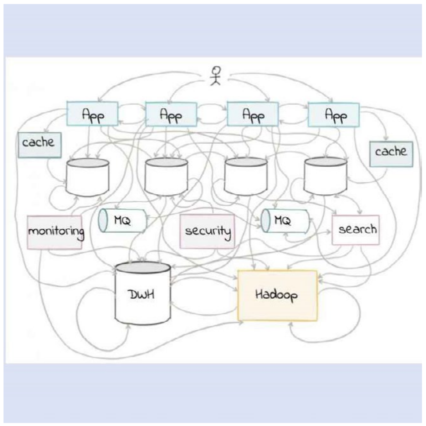


Figure 9: The data ecosystem is no longer a monolith

You have to figure out how to manage some of this operational pain. One of the first things you can do is to start integration so that you have fewer systems to deal with. We used Kafka for that.

Stage 3: Integration

If you think back 20 years to enterprise-service-bus architectures, that's really all data integration is. The only difference is that streaming platforms like Kafka along with the evolution in stream processing have made this viable.

You might be ready for data integration if you've got a lot of microservices. You have a diverse set of databases as Figure 8 depicts. You've got some specialized, derived data systems; I mentioned a graph database but you may have special caches or a real-time OLAP system. You've got a team of data engineers now, people who are responsible for managing this complex workload. Hopefully, you have a happy, mature SRE organization that's more than willing to take on all these connectors for you.

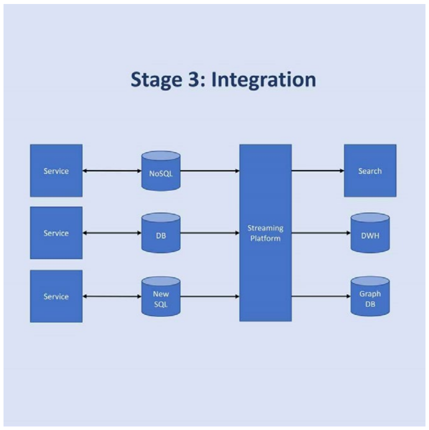


Figure 10: Stage 3 of data-pipeline maturity

Figure 10 shows what data integration looks like. We still have the base data pipeline that we've had so far. We've got a service with a database, we've got our streaming platform, and we've got our data warehouse, but now we also have web services, maybe a NoSQL thing, or a NewSQL thing. We've got a graph database and search system plugged in.

Figure 11 depicts where WePay was at the beginning of 2019. Things were becoming more complicated. Debezium connects not only to MySQL but to Cassandra as well, with the connector that we'd been working on. At the bottom is Kafka Connect Waltz (KCW). Waltz is a ledger that we built in house that›s Kafka-ish in some ways and more like a database in other ways, but it services our ledger use cases and needs.

We are a payment-processing system so we care a lot about data transactionality and multi-region availability and so we use a quorum-based write-ahead log to handle serializable transactions. On the downstream side, we've

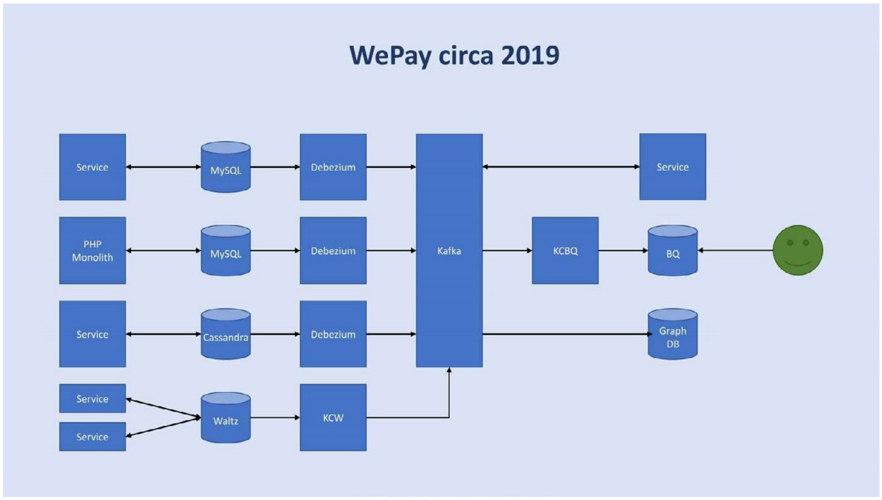


Figure 11: WePay's data ecosystem at the start of 2019

got a bunch of stuff going on.

We were incurring a lot of pain and have many boxes on our diagram. This is getting more and more complicated.

The reason we took on this complexity has to do with Metcalfe's law. I'm going to paraphrase the definition and probably corrupt it: it essentially states that the value of a network increases as you add nodes and connections to it. Metcalfe's law was initially intended to apply to communication devices, like adding more peripherals to an Ethernet network.

So, we're getting to a network effect in our data ecosystem. In a post in early 2019, I thought through the implications of Kafka as an escape hatch. You add more systems to the Kafka bus, all of which are able to load their data in and expose it to other systems and slurp up the data of in Kafka, and you leverage this network effect in your data ecosystem.

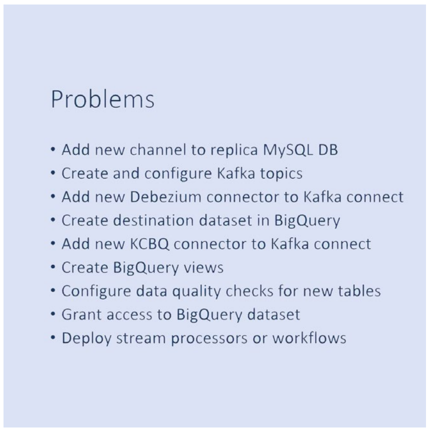
We found this to be a powerful architecture because the data becomes portable. I'm not saying it'll let you avoid vendor lock-in but it will at least ameliorate some of those concerns. Porting data is usually the harder part to deal with when you're moving between systems. The idea is that it becomes theoretically possible, if you're on Splunk for example, to plug in Elasticsearch alongside it to test it out - and the cost to do so is certainly lower.

Data portability also helps with multi-cloud strategy. If you need to run multiple clouds because you need high availability or you want to pick cloud vendors to save money, you can use Kafka and the Kafka bus to move the data around.

Lastly, I think it leads to infrastructure agility. I alluded to this with my Elasticsearch example but if you come across some new hot real-time OLAP system that you want to check out or some new cache that you want to plug in, having your data already in your streaming platform in Kafka means that all you need to do is turn on the new thing and plug in a sink to load the data. It drastically lowers the cost of testing new things and supporting specialized infrastructure.

You can easily plug in things that do one or two things really well, when before you might have had to decide between tradeoffs like supporting a specialized graph database or using an RDBMS which happens to have joins. By reducing the cost of specialization, you can build a more granular infrastructure to handle your queries.

The problems in Stage 3 look a little different. When WePay bought into this integration architecture, we found ourselves still spending a lot of time on fairly manual tasks like those in Figure 12.


Problems

- Add new channel to replica MySQL DB
- Create and configure Kafka topics
- Add new Debezium connector to Kafka connect
- Create destination dataset in BigQuery
- Add new KCBQ connector to Kafka connect
- Create BigQuery views
- Configure data quality checks for new tables
- Grant access to BigQuery dataset
- Deploy stream processors or workflows

Figure 12: WePay's problems in Stage 3

In short, we were spending a lot of time administering the systems around the streaming platform - the connectors, the upstream databases, the downstream data warehouses - and our ticket load looked like Figure 13.

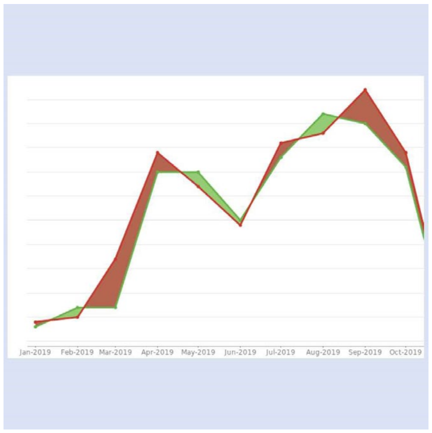


Figure 13: Ticket load at WePay in Stage 3

Fans of JIRA might recognize Figure 13. It is a screenshot of our support load in JIRA in 2019. It starts relatively low then it

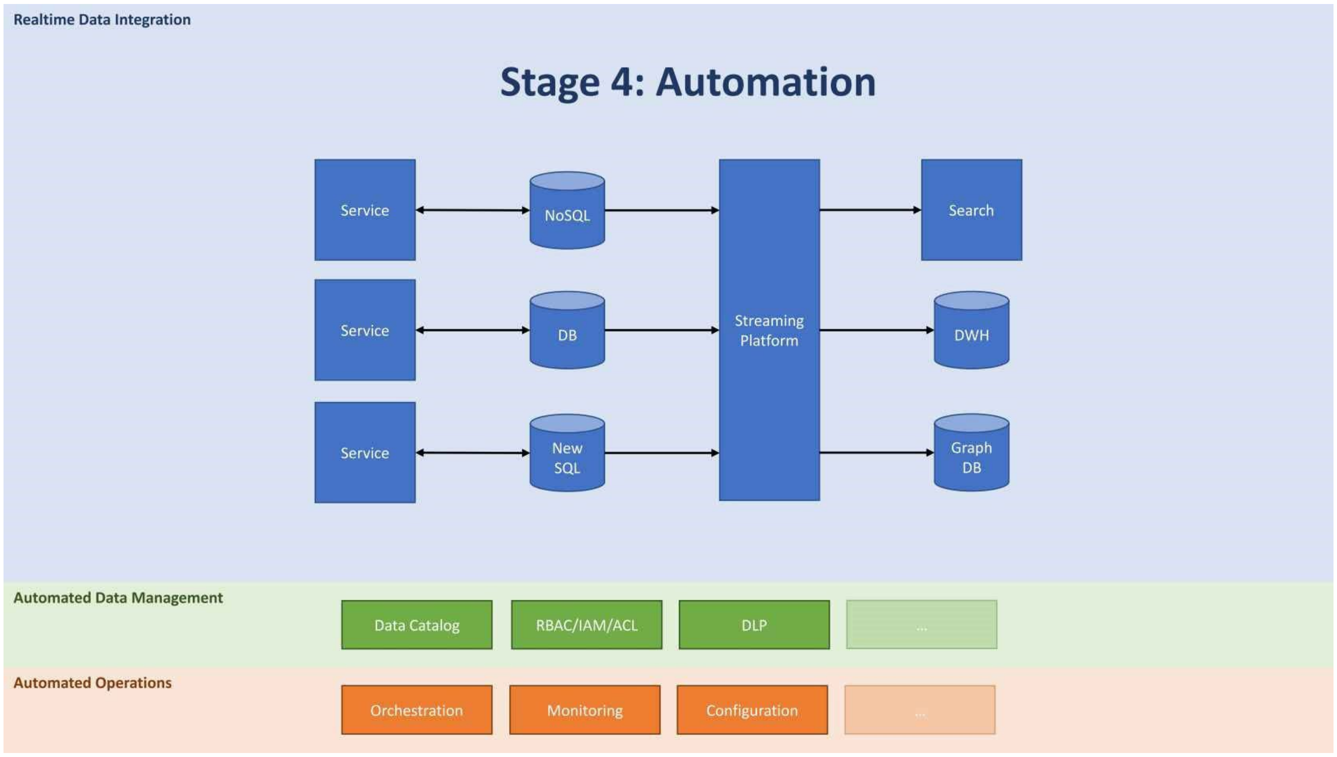


Figure 14: Stage 4 adds two new layers to the data ecosystem

skyrockets and it never fully recovered, although there's a nice trend late in the year that relates to the next step of our evolution.

Stage 4: Automation

We started investing in automation. This is something you've got to do when your system gets this big. I think most people would say we should have been automating all along.

You might be ready for Stage 4 if your SREs can't keep up, you're spending a lot of time on manual toil, and you don't have time for the fun stuff.

Figure 14 shows the two new layers that appear in Stage 4. The first is the automation of operations, and this won't surprise most people. It's the DevOps stuff that has been going on for a long time. The second layer, data-management automation, is not quite as obvious.

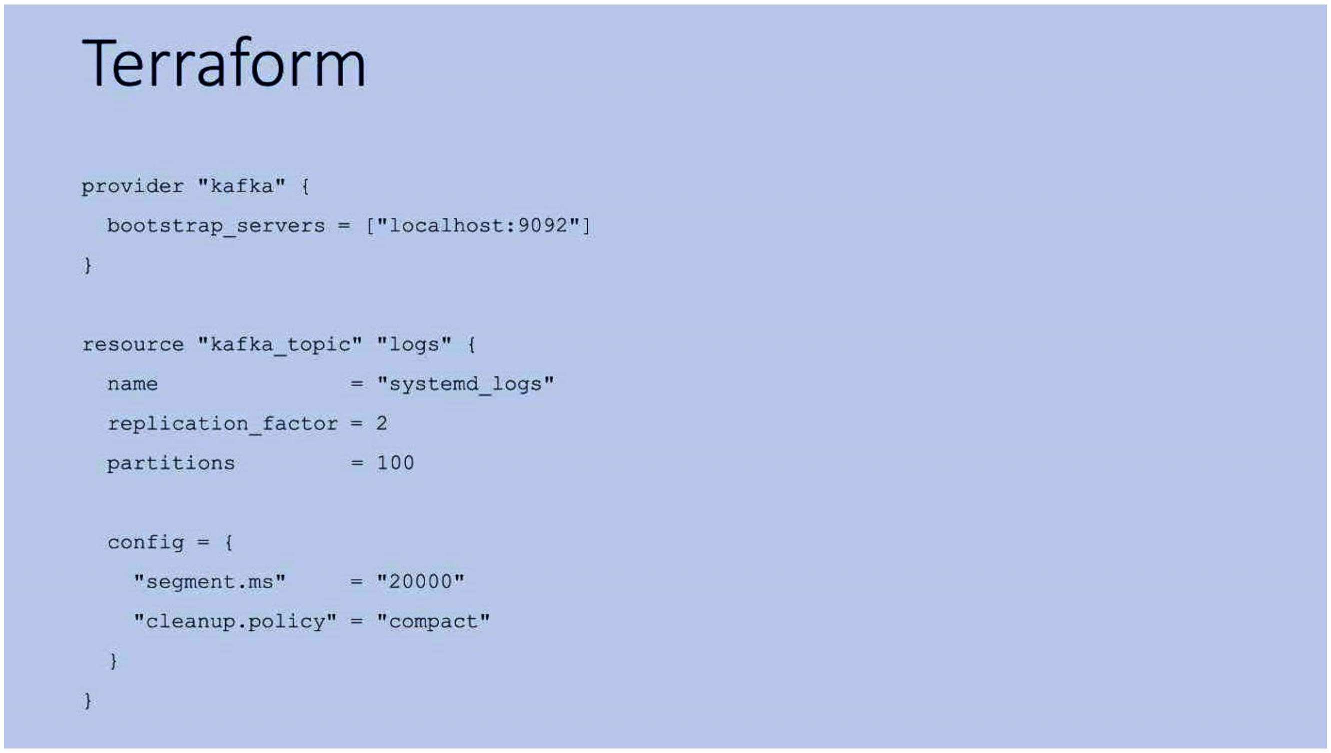
Let's first cover automation for operations. Google's *Site Reliability Engineering* handbook defines toil as manual, repeatable, automatable stuff. It›s usually interrupt-driven: you›re getting Slack messages or tickets or people are showing up at your desk asking you to do things. That is not what you want to be doing.

The Google book says, "If a human operator needs to touch your system during normal operations, you have a bug."

But the "normal operations" of data engineering were what we were spending our time on. Anytime you're managing a pipeline, you're going to be adding new topics, adding new data sets, setting up views, and granting access.

This stuff needs to get automated. Great news! There's a bunch of solutions for this: Terraform, Ansible, and so on.

We at WePay use Terraform and Ansible but you can substitute any similar product.



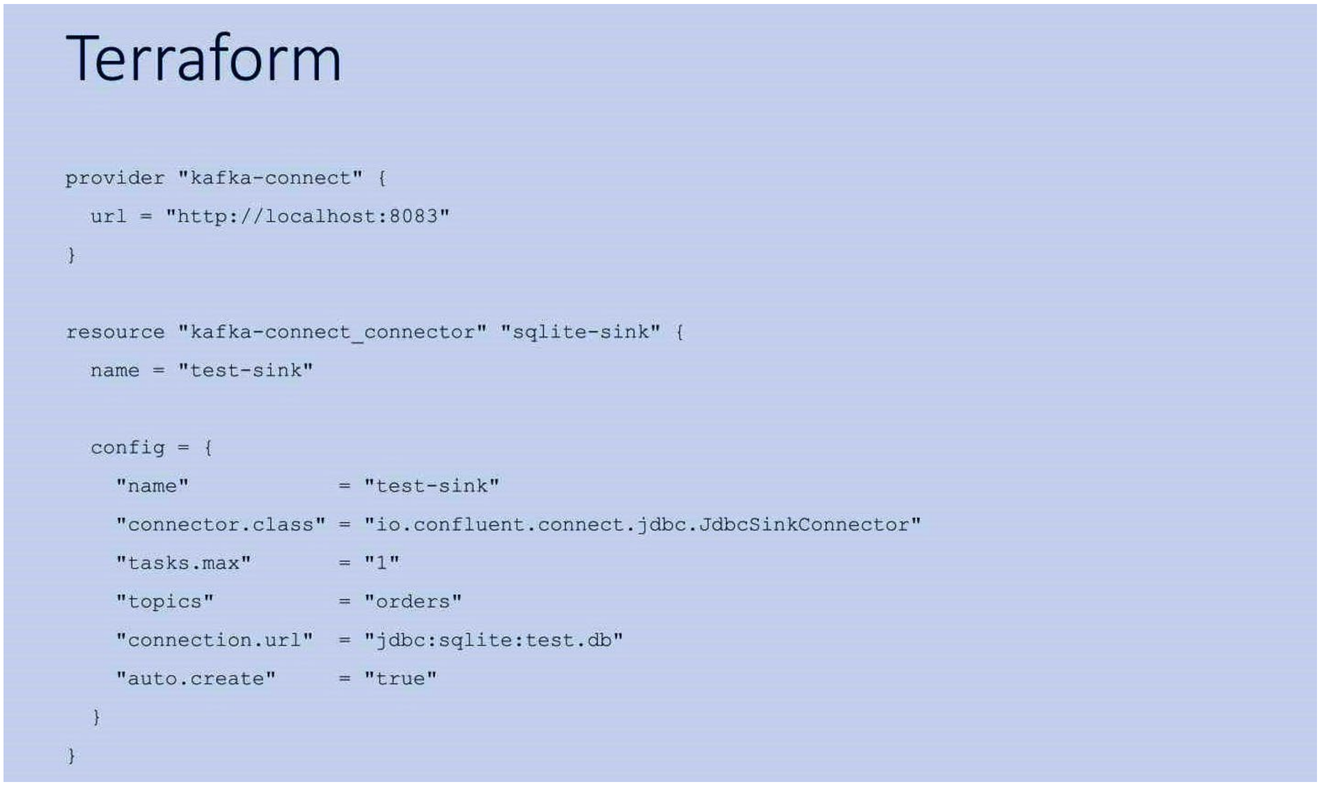
Terraform

```
provider "kafka" {
  bootstrap_servers = ["localhost:9092"]
}

resource "kafka_topic" "logs" {
  name               = "systemd_logs"
  replication_factor = 2
  partitions         = 100

  config = {
    "segment.ms"     = "20000"
    "cleanup.policy" = "compact"
  }
}
```

Figure 15: Some `systemd_log` thing in Terraform that logs some stuff when you're using compaction (which is an exciting policy to use with your `systemd_log`s)



Terraform

```
provider "kafka-connect" {
  url = "http://localhost:8083"
}

resource "kafka-connect_connector" "sqlite-sink" {
  name = "test-sink"

  config = {
    "name"            = "test-sink"
    "connector.class" = "io.confluent.connect.jdbc.JdbcSinkConnector"
    "tasks.max"       = "1"
    "topics"          = "orders"
    "connection.url"  = "jdbc:sqlite:test.db"
    "auto.create"     = "true"
  }
}
```

Figure 16: Managing your Kafka Connect connectors in Terraform

You can use it to manage your topics. Figures 15 and 16 show some Terraform automations. Not terribly surprising.

Yes, we should have been doing this, but we kind of were doing this already. We had Terraform, we had Ansible for a long time - we had a bunch of operational tooling. We were fancy and on the cloud.

We had a bunch of scripts to manage BigQuery and automate a lot of our toil like creating views in BigQuery, creating data sets, and so on. So why did we have such a high ticket load?

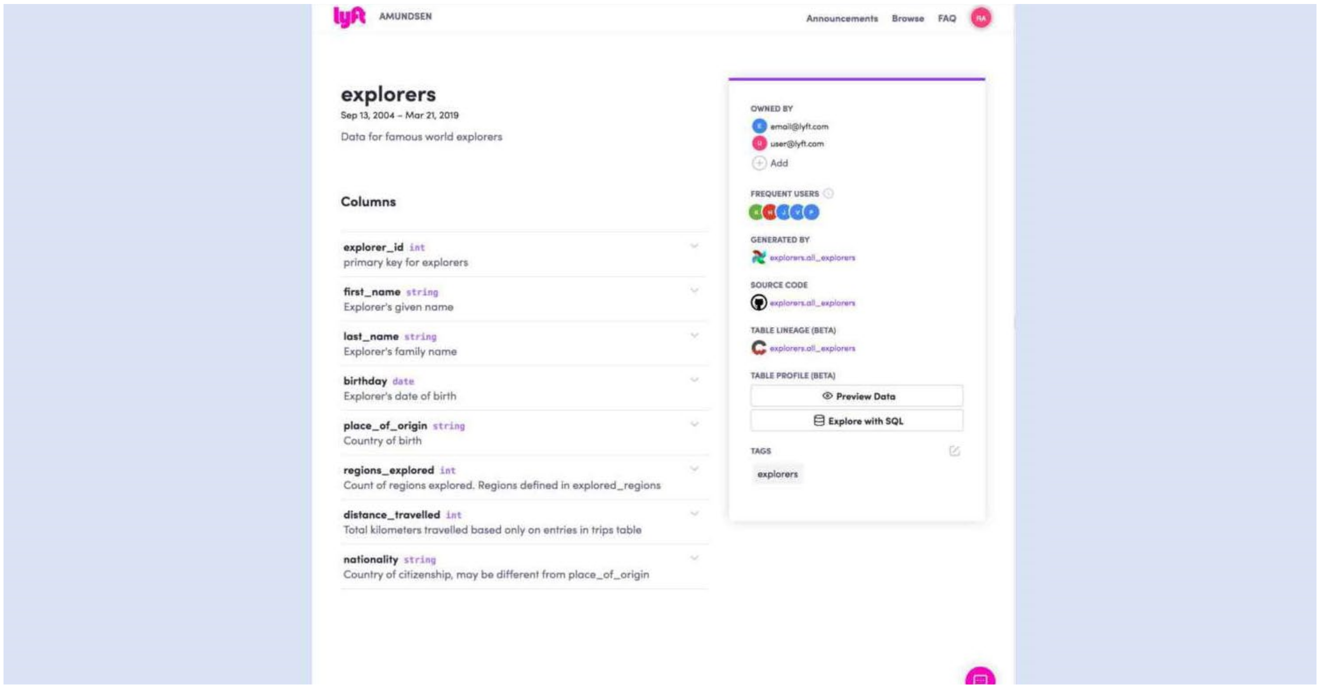
The answer is that we were spending a lot of time on data management. We were answering questions like "Who's going to get access to this data once I load it?", "Security, is it okay to persist this data indefinitely or do we need to have a three-year truncation policy?", and "Is this data even allowed in the system?" As a payment processor, WePay deals with sensitive information and our people need to follow geography and security policies and other stuff like that.

We have a fairly robust compliance arm that's part of JPMorgan Chase. Because we deal with credit cards, we have PCI audits and we deal with credit-card data. Regulation is here and we really need to think about this. Europe has GDPR. California has CCPA. PCI applies to credit-card data. HIPAA for health. SOX applies if you're a public company. New York has SHIELD. This is going to become more and more of a theme, so get used to it. We have to get better at automating this stuff or else our lives as data engineers are going to be spent chasing people to make sure this stuff is compliant.

I want to discuss what that might look like. As I get into the futuristic stuff, I get more vague or hand-wavy, but I'm trying to keep it as concrete as I can.

First thing you want to do for automated data management is probably to set up a data catalog. You probably want it centralized, i.e., you want to have one with all the metadata.

The data catalog will have the locations of your data, what schemas that data has, who owns the data, and lineage, which is essentially the source and path of the data.



explorers
Sep 13, 2004 – Mar 21, 2019
Data for famous world explorers

Columns

explorer_id int
primary key for explorers

first_name string
Explorer's given name

last_name string
Explorer's family name

birthday date
Explorer's date of birth

place_of_origin string
Country of birth

regions_explored int
Count of regions explored. Regions defined in explored_regions

distance_travelled int
Total kilometers travelled based only on entries in trips table

nationality string
Country of citizenship, may be different from place_of_origin

OWNED BY
email@lyft.com
user@lyft.com
Add

FREQUENT USERS

GENERATED BY
explorers.all_explorers

SOURCE CODE
explorers.all_explorers

TABLE LINEAGE (BETA)
explorers.all_explorers

TABLE PROFILE (BETA)
Preview Data
Explore with SQL

TAGS
explorers

Figure 17: An example of Amundsen

The lineage for my initial example is that it came from MySQL, it went to Kafka, and then it got loaded into BigQuery - that whole pipeline. Lineage can even track encryption or versioning, so you know what things are encrypted and what things are versioned as the schemas evolved.

There's a bunch of activity in this lineage area. Amundsen is a data catalog from Lyft. You have Apache Atlas. LinkedIn open-sourced DataHub as a patch in 2020. WeWork has a system called Marquez. Google has a product called Data Catalog. I know I'm missing more.

These things generally do a lot, more than one thing, but I want to show a concrete example. I yanked Figure 17 from the Amundsen blog.

It has fake data, the schema, the field types, the data types, everything. At the right, it has who owns the data - and notice that Add button there.

It tells us what source code generated the data — in this case, it's Airflow, as indicated by that little pinwheel — and some lineage. It even has a little preview. It's a pretty nice UI.

Underneath it, of course, is a repository that actually houses all this information. That's really useful because you need to get all your systems to be talking to this data catalog.

That Add button in the Owned By section is important. You don't as a data engineer want to be entering that data yourself. You do not want to return to the land of manual data stewards and data management.

Instead, you want to be hooking up all these systems to your data catalog so that they're automatically reporting stuff about the schema, about the evolution of the schema, about the ownership when the data is loaded from one to the next.

First off, you need your systems like Airflow and BigQuery, your data warehouses and stuff, to talk to the data catalog. I think there's quite a bit of movement there.

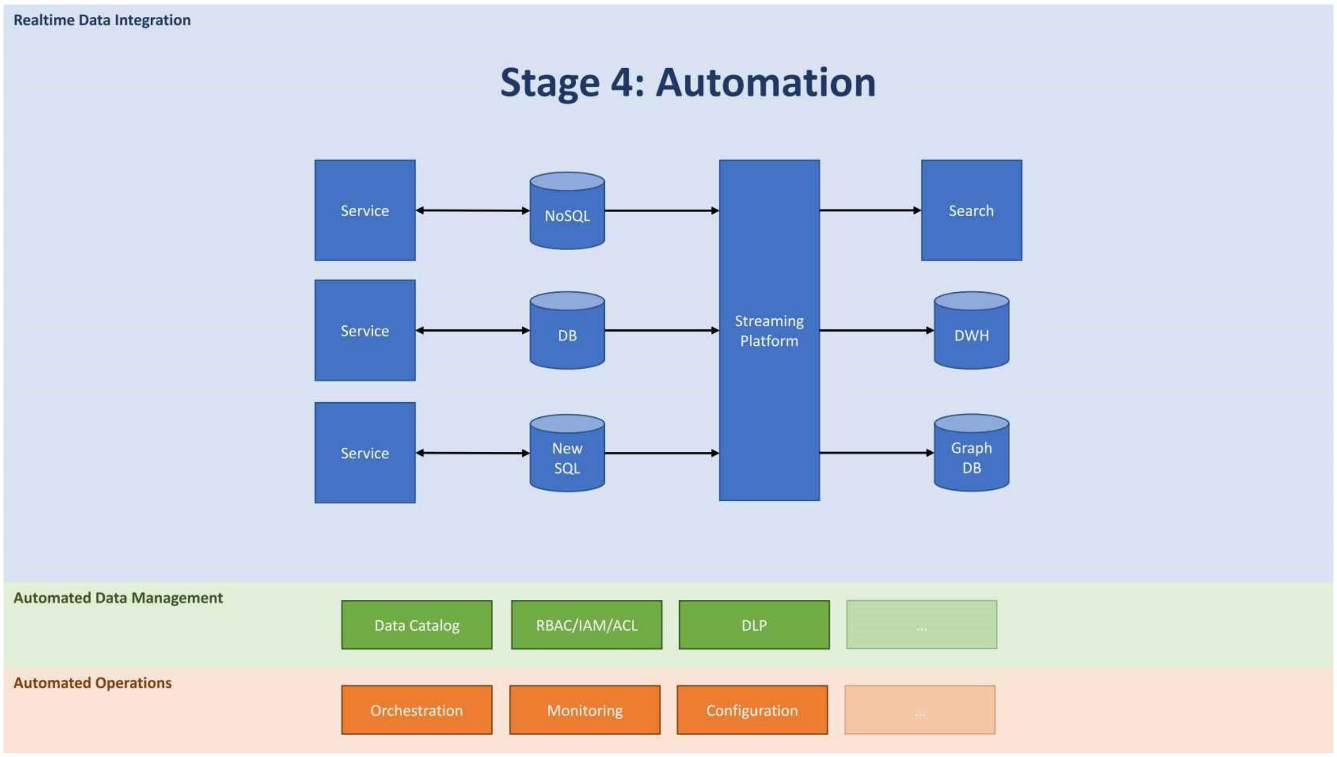


Figure 18: Your data ecosystem needs to talk to your data catalog

You then need your data-pipeline streaming platforms to talk to the data catalog. I haven't seen as much yet for that. There may be stuff coming out that will integrate better, but right now I think that's something you've got to do on your own.

I don't think we've done a really good job of bridging the gap on the service side. You want your service stuff in the data catalog as well: things like gRPC protobufs, JSON schemas, and even the DBs of those databases.

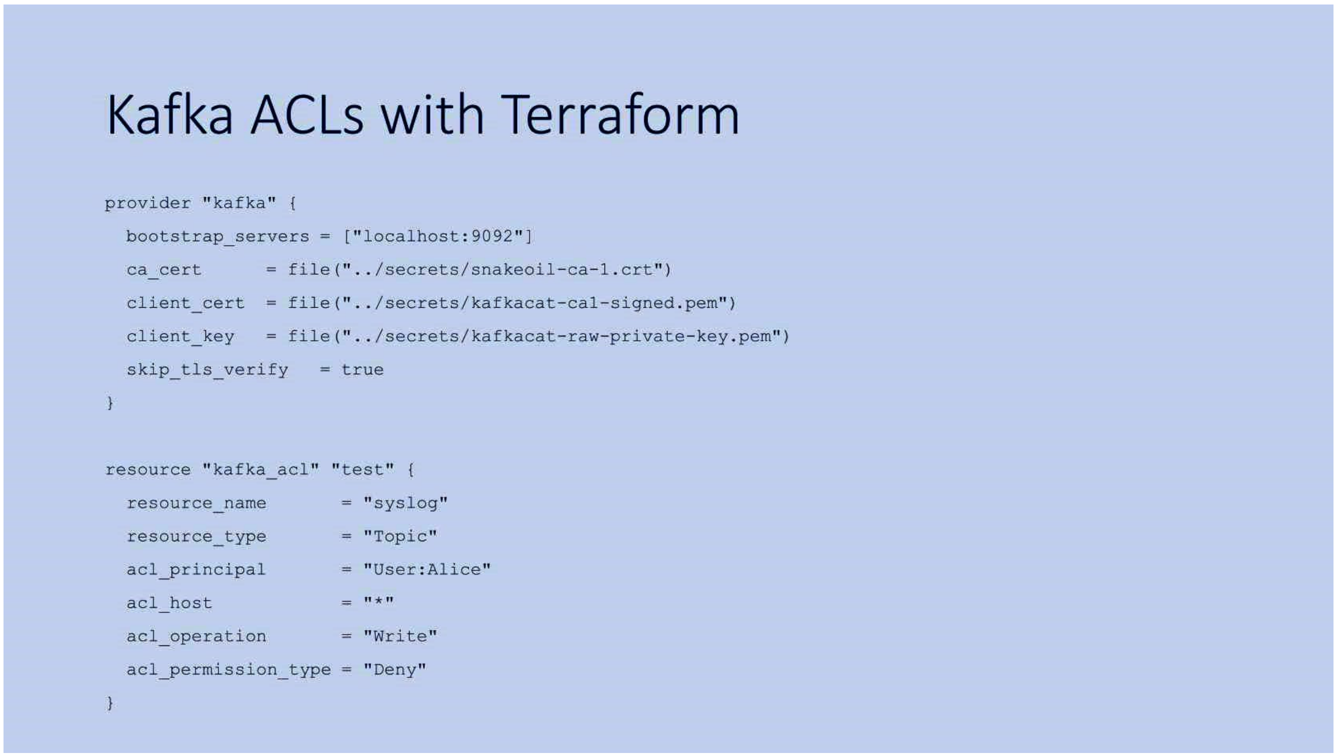
Once you know where all your data is, the next step is to configure access to it. If you haven't automated this, you're probably going to Security, Compliance, or whoever the policymaker is and asking if this individual can see this data whenever they make access requests - and that's not where you want to be. You want to be able to automate the access-request management so that you can be as hands off with it as possible.

This is kind of an alphabet soup with role-based access control (RBAC), identity access management (IAM), and an access-control list (ACL). Access control is just a bunch of fancy words for a bunch of different features for managing groups, user access, and so on.

You need three things to do this: you need your systems to support it, you need to provide tooling to policymakers so they can configure the policies appropriately, and you need to automate the management of the policies once the policymakers have defined them.

There has been a fair amount of work done to support this aspect. Airflow has RBAC, which was a patch WePay submitted. Airflow has taken this seriously and has added a lot more, like DAG-level access control. Kafka has had ACLs for quite a while.

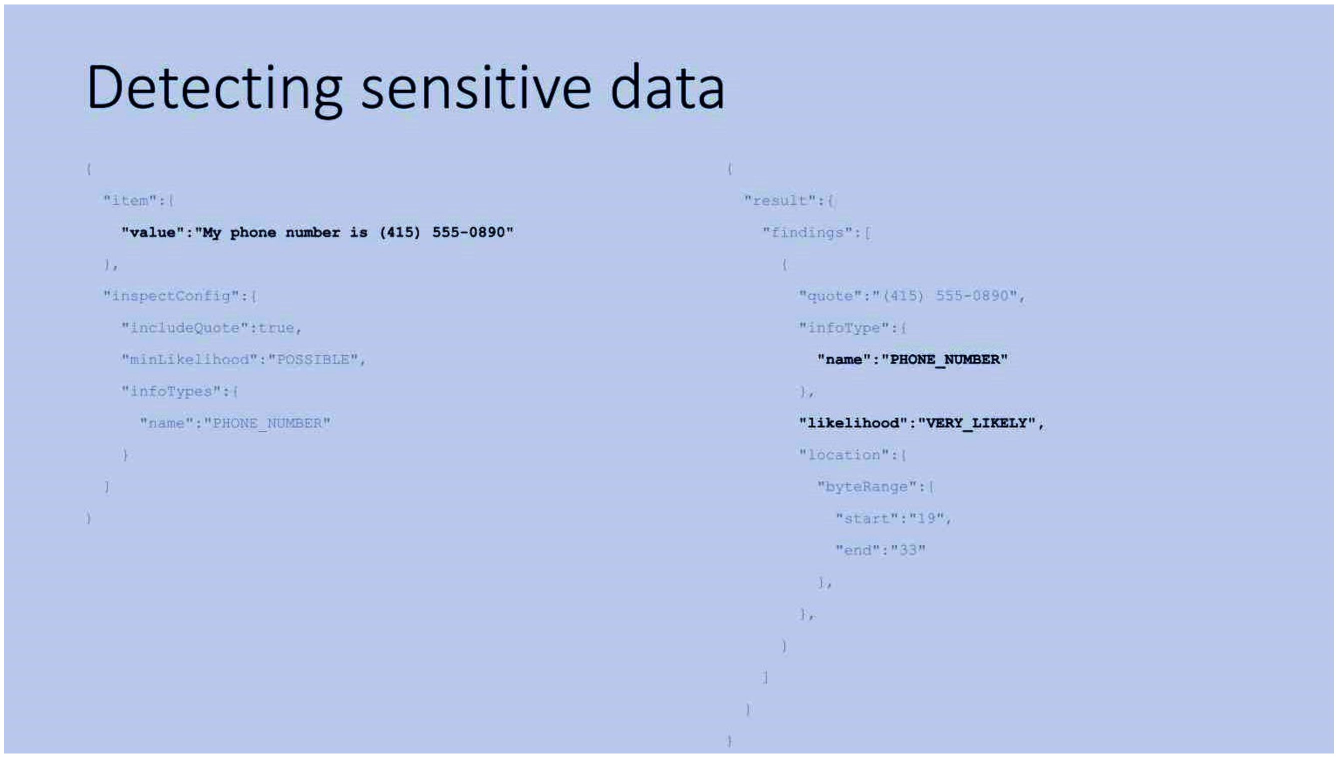
You can use tools to automate this stuff. We want to automate adding a new user to the system and configuring their access. We want to automate the configuration of access controls when a new piece of data is added to the system. We want to automate service-


Kafka ACLs with Terraform

```
provider "kafka" {
  bootstrap_servers = ["localhost:9092"]
  ca_cert      = file("../secrets/snakeoil-ca-1.crt")
  client_cert  = file("../secrets/kafkacat-ca1-signed.pem")
  client_key   = file("../secrets/kafkacat-raw-private-key.pem")
  skip_tls_verify   = true
}

resource "kafka_acl" "test" {
  resource_name       = "syslog"
  resource_type       = "Topic"
  acl_principal       = "User:Alice"
  acl_host            = "*"
  acl_operation       = "Write"
  acl_permission_type = "Deny"
}
```

Figure 19: Managing Kafka ACLs with Terraform



Detecting sensitive data

```
{
  "item":{
    "value":"My phone number is (415) 555-0890"
  },
  "inspectConfig":{
    "includeQuote":true,
    "minLikelihood":"POSSIBLE",
    "infoTypes":{
      "name":"PHONE_NUMBER"
    }
  }
}
```

```
{
  "result":{
    "findings":[
      {
        "quote":"(415) 555-0890",
        "infoType":{
          "name":"PHONE_NUMBER"
        },
        "likelihood":"VERY_LIKELY",
        "location":{
          "byteRange":{
            "start":"19",
            "end":"33"
          },
        },
      }
    ]
  }
}
```

Figure 20: Detecting sensitive data

account access as new web services come online.

There's occasionally a need to grant someone temporary access to something. You don't want to have to set a calendar reminder to revoke the access for this user in three weeks. You want that to be automated. The same goes for unused access. You want to know when users aren't using all the permissions that they're granted so that you can strip those unused permissions to limit the vulnerability of the space.

Now that your data catalog tells you where all the data is and you have policies set up, you need to detect violations. I mostly want to discuss data loss prevention (DLP) but there's also auditing, which is keeping track of logs and making sure that the activities and systems are conforming to the required policies.

I'm going to talk about Google Cloud Platform because I use it and I have some experience with its data-loss solution. There's a corresponding AWS product called Macie. There's also an open-source project called Apache Ranger, with a bit of an enforcement and monitoring mechanism built into it; that's more focused on the Hadoop ecosystem. What all these things have in common is that you can use them to detect the presence of sensitive data where it shouldn't be.

Figure 20 is an example. A piece of submitted text contains a phone number, and the system sends a result that says it is "very likely" that it has detected an infoType of phone number. You can use this stuff to monitor your policies. For example, you can run DLP checks on a data set that is supposed to be clean - i.e., not have any sensitive information in it - and if a check finds anything like a phone number, Social Security number, credit card, or other sensitive information, it can immediately alert you that there's a violation in place.

There's a little bit of progress here. Users can use the data catalog and find the data that they need, we have some automation in place, and maybe we're using Terraform to manage ACLs for Kafka or to manage RBAC in Airflow.

But there's still a problem and that is that data engineering is probably still responsible for managing that configuration and those deployments. The reason for that is mostly the interface. We're still getting pull requests, Terraform, DSL, YAML, JSON, Kubernetes ... it's nitty-gritty.

It might be a tall order to ask security teams to make changes to that. Asking your compliance wing to make changes is an even taller order. Going beyond your compliance people is basically impossible.

Stage 5: Decentralization

You're probably ready to decentralize your data pipeline and your data warehouses if you have a fully automated real-time data pipeline but people are still coming to ask you to load data.

If you have an automated data pipeline and data warehouse, I don't think you need a single team to manage all this stuff. I think the place where this will first happen, and we're already seeing this in some ways, is in a decentralization of the data warehouse. I think we're moving towards a world where people are going to be empowered to spin up multiple data warehouses and administer and manage their own.

I frame this line of thought based on our migration from monolith to microservices over the past decade or two. Part of the motivation for that was to break up large, complex things, to increase agility, to increase efficiency, and to let people move at their own pace. A lot of those characteristics sound like your data warehouse: it's monolithic, it's not that agile, you have to ask your data engineering team to do things, and maybe you're not able to do things at your own pace. I think we're going to want to do the same thing - go from a monolith to microwarehouses - and we're going to want a more decentralized approach.

I'm not alone in this thought. Zhamak Dehghani wrote a great blog post that is such a great description of what I'm thinking. She discusses the shift from this monolithic view to a more fragmented or decentralized view. She even discusses policy

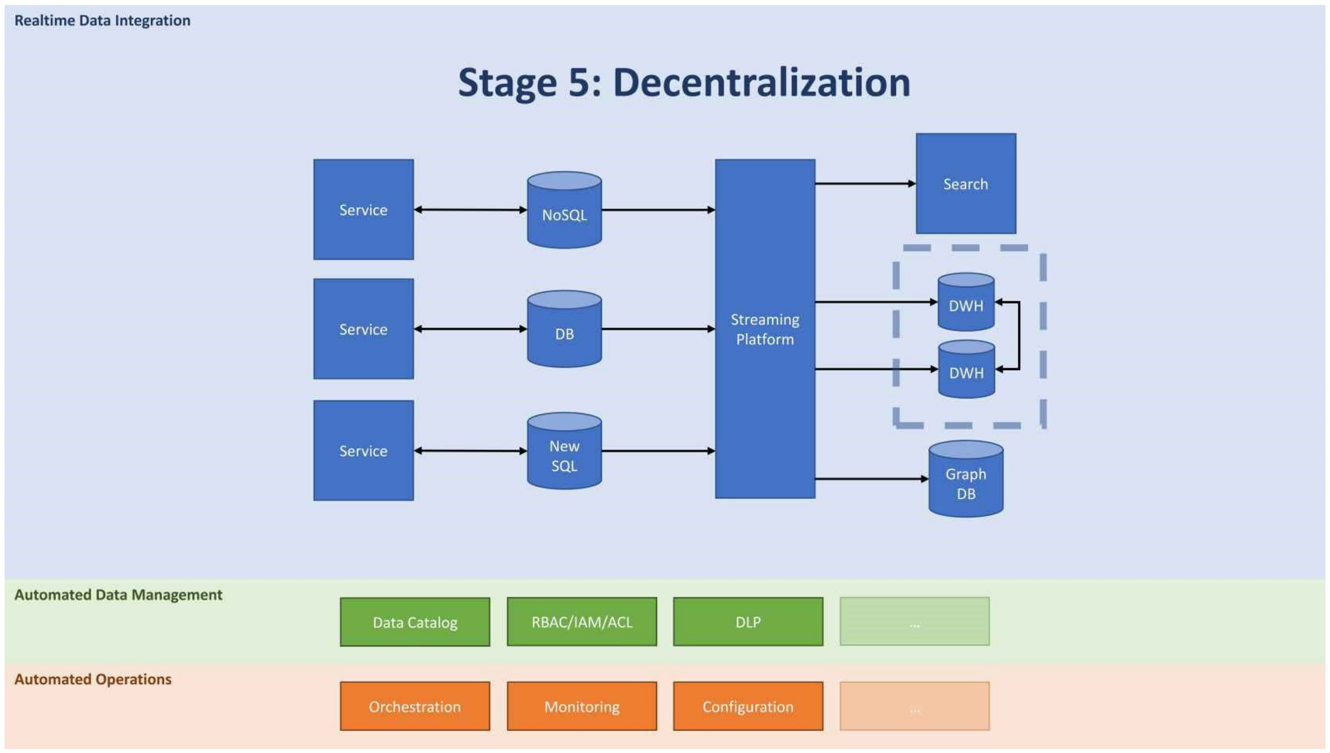


Figure 21: Decentralization is Stage 5 of the data ecosystem

automation and a lot of the same stuff that I›m thinking about.

I think this shift towards decentralization will take place in two phases. Say you have a set of raw tools - Git, YAML, JSON, etc. - and a beaten-down engineering team that is getting requests left and right and running scripts all the time. To escape that, the first step is simply to expose that raw set of tools to your other engineers.

They're comfortable with this stuff, they know Git, they know pull requests, they know YAML and JSON and all that. You can at least start to expose the automated tooling and pipelines to those teams so that they can begin to manage their own data warehouses.

An example of this would be a team that does a lot of reporting. They need a data warehouse that they can manage so you might just give them keys to the castle, and they can go about it. Maybe there's a business-analytics team that's attached to your sales organization and they need a data warehouse. They can manage their own as well.

This is not the end goal; the end goal is full decentralization. But for that we need much more development of the tooling that we're providing, beyond just Git, YAML, and the RTFM attitude that we sometimes throw around.

We need polished UIs, something that you can give not only to an engineer who's been writing code for 10 years but to almost anyone in your organization.

If we can get to that point, I think we will be able to create a fully decentralized warehouse and data pipeline where Security and Compliance can manage access controls while data engineers manage the tooling and infrastructure.

This is what Maxime Beauchemin meant by "... data engineers build tools, infrastructure, frameworks, and services." Everyone else can manage their own data pipelines and their own data warehouses and data engineers can help them do that. There's that key word "help" that I drew attention to at the beginning.

Figure 21 is my view of a modern decentralized data architecture. We have real-time data integration, a streaming platform, automated data management, automated operations, decentralized data warehouses and pipelines, and happy engineers, SREs, and users.

TL;DR

- The job of data engineers is to help an organization move and process data but not to do that themselves.
- Data architecture follows a predictable evolutionary path from monolith to automated, decentralized, self-serve data microwarehouses.
- Integration with connectors is a key step in this evolution but increases workload and requires automation to correct for that.
- Increasing regulatory scrutiny and privacy requirements will drive the need for automated data management.
- A lack of polished, non-technical tooling is currently preventing data ecosystems from achieving full decentralization.

SPONSORED ARTICLE

5 Elements for a DevOps-Friendly Analytics Solution

When analytics providers say their platform and tools will work with your tech stack, what do they really mean? Here are five important factors to consider when evaluating solutions for data visualization, dashboards and reporting:

1. Security

If the analytics solution is DevOps-friendly, it will continue handling security with the same methods your DevOps team is currently using. You won't have to recreate or replicate BI security information in two different places. The rising trend of DevSecOps teams is an indicator of how essential security is to DevOps teams everywhere. Consider the following:

- **Authorization controls.** Whatever DevOps is currently using—credentials, enterprise services account, or something else—the analytics platform should adapt to it.
- **Standard-based SSO.** It's important for user experience that embedded analytics applications play within a single sign-on (SSO) environment—and important for DevOps that they do it based on standards. Any custom elements divert DevOps resources to figuring out SSO peculiarities and decrease the odds of being able to quickly debug any problems.
- **Defense against known vulnerabilities.** Look for an embedded analytics vendor who is already protecting against common security vulnerabilities that any web application will face.
- **Incident visibility.** If there is a security incident, does the platform provide detailed event logging? DevOps should be able to quickly find the Who, What, and When of everything going on at the time.

2. Data

A DevOps-friendly analytics solution will not force you to use a proprietary data store, replicate data, or change your current data schemas. You should be able to store your data in place via relational databases, web services, or your own proprietary solution. Applications incorporating analytics should be able to use data in your existing systems and deliver strong performance across disparate sources.

3. Environment

A DevOps-friendly analytics solution will work in any environment. That means applications will work on clouds, in containers, and across mixed and changing deployment environments. These may include Windows or Linux; on-premises, cloud, and hybrid architectures; and mobile devices and browsers. It also means applications can be deployed in a container, or parts of them dispersed across multiple containers.

4. Architecture

If the analytics solution is DevOps-friendly, you should be able to deploy it into your current architecture using standard methods (with minimal architecture-specific steps). For instance, it may be as simple as installing an ASP.NET application on an IIS server or a Java application on an Apache Tomcat server.

5. Release Cycles

A DevOps-friendly embedded analytics solution won't drag release cycles. If you're moving toward continuous integration and continuous delivery (CI/CD), you want to control exactly what gets deployed and when through your build pipelines. Does the embedded analytics platform allow you to deploy smaller size incremental update packages when broader changes aren't needed?

Sustainable Innovation and Differentiation

Your analytics solution will affect how frequently you release standout software, how competitive you are, and how sustainable your advantage is over time. In summary, look for one that:

- **Utilizes your technology—** including security frameworks and tech stack architecture—as it is
- **Leverages your existing processes** so you can build and release application updates faster
- **Offers flexible scaling** so it grows with your business

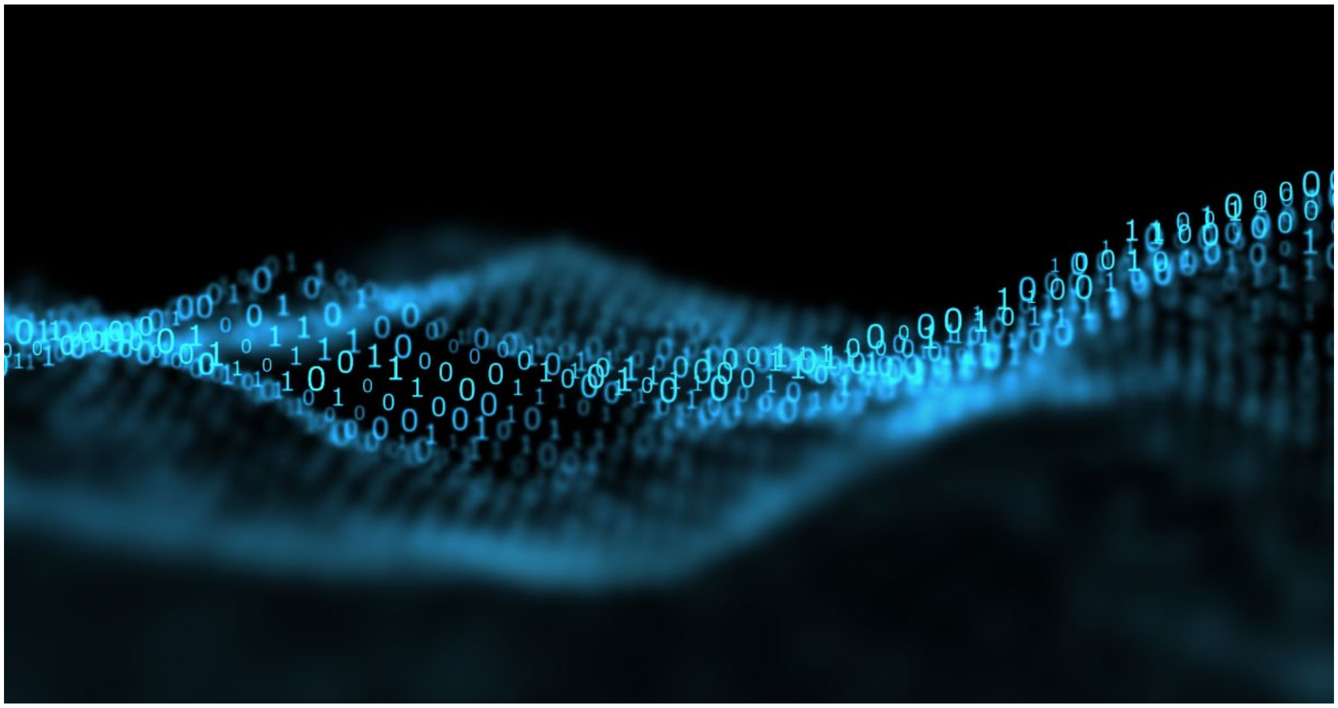
Logi Analytics: The Embedded Analytics Experts

Logi Analytics empowers the world's software teams with the most intuitive, developer-grade embedded analytics solutions and a team of dedicated people, invested in your success.

Logi leverages your existing tech stack, so you can quickly build, manage and deploy your application. And because Logi supports unlimited customization and white-labeling, you have total control to make the application uniquely your own.

Over 2,200 application teams have trusted Logi to help power their businesses with sophisticated analytics capabilities.

Because it shouldn't take an entire team to develop in-app:
Dashboards, Data Viz & Reporting
Logi's low-cost, low-code development tools can save up to 50% in up-front investment for dashboards, data visualization and reporting.
Logi ANALYTICS



Beyond the Database, and beyond the Stream Processor: What's the Next Step for Data Management?

by **Ben Stopford**, Lead Technologist, Office of the CTO at Confluent

At QCon London in March 2020, I gave a talk on why both stream processors and databases remain necessary from a technical standpoint and explored industry trends that make consolidation likely in the future. These trends map onto common approaches from active databases like MongoDB to streaming solutions like Flink, Kafka Streams, or ksqlDB.

I work at Confluent, the company founded by the creators of Apache Kafka. These days, I work in the Office of the CTO. One of the things we did last year was to look closely at the differences between stream processors and databases. This led to a new product called ksqlDB. With the rise in popularity of event-streaming systems and their obvious relationship to databases, it's useful to compare how the different models handle data that's moving versus data that is stationary. Maybe more importantly, there is clear consolidation happening between these fields. Databases are becoming increasingly active, emitting events as data is written, and stream processors are increasingly passive, providing historical queries over datasets they've accumulated.

You might think this kind of consolidation at a technical level is an intellectual curiosity, but if you step back a little it really points to a more fundamental shift in the way that we build software. Marc Andreessen, now a venture capitalist in Silicon Valley, has an excellent way of putting this: "software is eating the world". Investing in software companies makes sense simply because both individuals and companies consume more software over time. We buy more applications for our phones, we buy more internet services, companies buy more business software, etc. This software makes our world more efficient. That's the trend.

But this idea that we as an industry merely consume more software is a shallow way to think about it. A more profound perspective is that our businesses effectively become more automated. It's less about buying more software and more about using software to automate our business processes, so the whole thing works on autopilot — a company is thus "becoming software", in a sense. Think Netflix: their business started with sending DVDs via postal mail to customers, who returned the physical media via post. By 2020, Netflix has become a pure software platform that allows customers to watch any movie immediately with the click of a button. Let's look at another example at what it means for businesses to "become software".

Think about something like a loan-processing application for a mortgage someone might get for their home. This is a business process that hasn't changed for a hundred years. There's a credit officer, there's a risk officer, and there's a loan officer. Each has a particular role in this process, and companies write or purchase software that makes the process more efficient. The purchased software helps the credit officer do her job better, or helps the risk officer do his job better. That's the weak form of this analogy: we buy more software to help these people do a better job.

These days, modern digital companies don't build software to help people do a better job: they build software that take humans completely out of the critical path. Continuing the example, today, we can get a loan application approved in only a few seconds (a traditional mortgage will take maybe a couple of weeks, because it follows that older manual process). This uses software, but it's using software in a different way. Many different pieces of software are chained together into a single, fully automated process. Software talks to other software, and the resulting loan is approved in a few seconds.

So, software makes our businesses more efficient, but think about what this means for the architecture of those systems. In the old world, we might build an application that helps a credit officer do her job better, say, using a three-tier architecture. The credit officer talks to a user interface, then there's some kind of back-end server application running behind that, and a database, and that all helps the person do risk analysis, or whatever it might be, more efficiently.

As companies become more automated, and their business processes become more automated, we end up with many applications talking to one another: software talking to software. This is a humongous shift in system design as it's no longer about helping human users, it's about doing the work in a fully automated fashion.

From Monoliths to Event-Driven Microservices: the evolution of software architecture

We see the same thing in the evolution of software architecture: monolith to distributed monolith to microservices, see Figure 1. below. There is something special about the event-driven microservices though. Event-driven microservices don't talk directly to the user interface. They automate business processes rather than responding to users clicking buttons and expecting things to happen. So in these architectures there is a user-centric side and a software-centric side. As architectures evolve from left to right, in Figure 1, the "user" of one piece of software is more likely to be another piece of software, rather than being a human, so software evolution also correlates with Marc Andreessen's observation.

Modern architectures and the consequences of traditional databases

We've all used databases. We write a query, and send it to a server somewhere with lots of data on it. The database answers our questions. There's no way we'd ever be able to answer data-

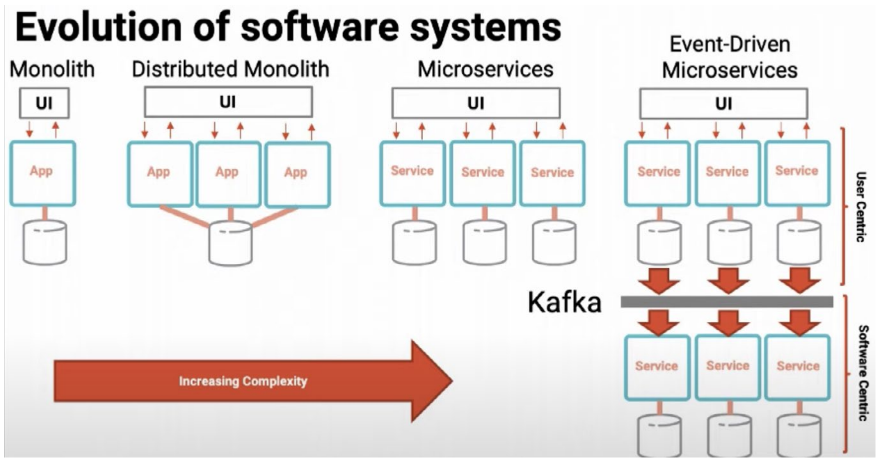


Figure 1: The evolution of software systems.

centric questions like these by ourselves. There's simply too much data for our brains to parse. But with a database, we simply send a question and we can get an answer. It's wonderful. It's powerful.

The breadth of database systems available today is staggering. Something like Cassandra lets us store a huge amount of data for the amount of memory the database is allocated; Elasticsearch is different, providing a rich, interactive query model; Neo4j lets us query the relationship between entities, not just the entities themselves; things like Oracle or PostgreSQL are workhorse databases that can morph to different types of use case.

Each of these platforms has slightly different capabilities that make it more appropriate to a certain use case but at a high level, they're all similar. In all cases, we ask a question and wait for an answer.

This hints at an important assumption all databases make: data is passive. It sits there in the database, waiting for us to do something. This makes a lot of sense: the database, as a piece of software, is a tool designed to help us humans — whether it's you or me, a credit officer, or whoever — interact with data.

But if there's no user interface waiting, if there's no one clicking buttons and expecting things to happen, does it have to be synchronous? In a world where software is increasingly talking to other software, the answer is: probably not.

One alternative is to use event streams. Stream processors allow us to manipulate event streams similar to the way that databases manipulate data that is held in files.

That's to say: stream processors are built for active data, data that is in motion, and they're built for asynchronicity. But anyone who has used a stream processor probably recognizes that it doesn't feel much like a traditional database.

In a traditional database interaction, the query is active and the data is passive. Clicking a button and running the query is what makes things happen. The data passively waits for us to run that query. If we're off making a cup of tea, the database doesn't do anything. We have to initiate that action ourselves.

In a stream processor, it's the other way around. The query is passive. It sits there running, just waiting for events to arrive. The trigger isn't someone clicking a button and running the query, it's

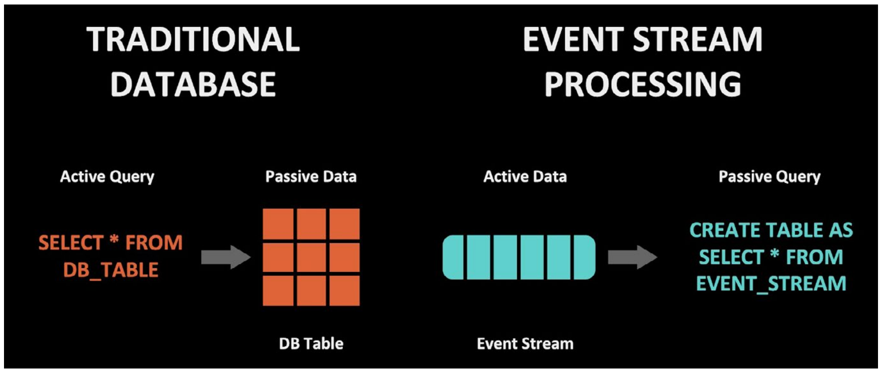


Figure 2: The database versus stream processing.

the data itself — an event emitted by some other system or whatever it might be. The interaction model is fundamentally different. So what if we combined them together?

Event Streams are the key to solving the data issues of modern distributed architectures

With that in mind, I want to dig a little deeper into some of the fundamental data structures used by stream processors, and compare how those relate to databases. Probably the most fundamental relationship here is the one between events, streams and tables.

Events are the building block and, conceptually, they are a simple idea: a simple recording of something that happened in the world at a particular point in time. So, an event could be somebody placing an order, paying for a pair of trousers, or moving a rook in a chess game. It could be the position of your phone or another kind of continuous event stream.

Individual events form into streams, and the picture changes further. An event stream represents the variance in some variable: the position of your phone as you drive, the progress of your order, or the moves in a game of chess.

All, exact recordings of what changed in the world. By comparison, a database table represents a snapshot of the current state at a single point in time. You have no idea what happened previously!

In fact, a stream closely relates to the idea of event sourcing, this programming paradigm that stores data in a database as events with the objective of retaining all this information.

If we want to derive our current state, i.e. a table, we can do this by replaying the event streams.

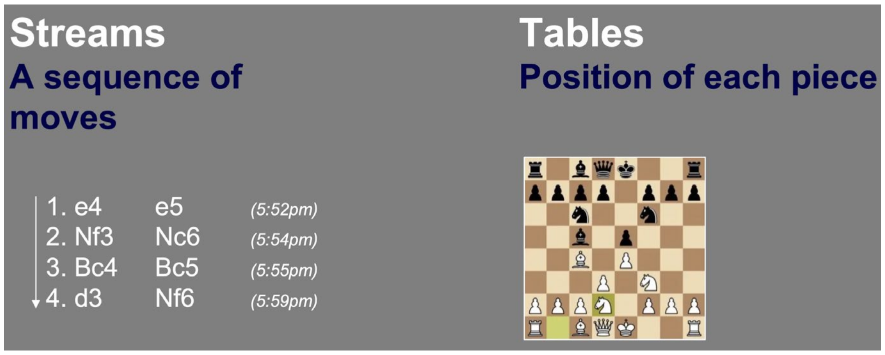


Figure 3: We can think of chess as a sequence of events or as records of positions

Chess is a good analogy for this. Think of a database table as describing the position of each piece. The position of each of those pieces tells me the current state of a game. I can store that somewhere and reload it if I want to. But that's not the only option, representing a chess game as events gives quite a different result.

We store the sequence of events from the well-known opening position all chess games start from. The current position of the board at any point in the game can then be derived by applying all subsequent moves to the opening position.

Note that the event-based approach contains more information about what actually happened. Not only do we know the positions at one point in the game, we also know how the game unfolded. We can, for example, determine whether we arrived at a position on the board through a brilliant move of one player versus a terrible blunder of the opponent.

Now we can think of an event stream as a special type of table. It's a particular type of table that doesn't exist in most databases. It's immutable. We can't change it. And it's append-only — we can only insert new records to it. By contrast, traditional database tables let us update and delete as well as insert. They're mutable.

If I write a value to a stream, it is automatically going to live forever. I can't go back and change some arbitrary event, so it's more like an accounting ledger with doubleentry bookkeeping. Everything that ever happened is recorded in the ledger. But when we think about data in motion there is another good reason for taking this approach.

Events are typically published to other parts of the system or the company, so it might already have been consumed by somebody else by the time you want to change it.

So unlike in a database where you can just update the row in question when data is moving you actually need to create a compensating event that can propagate that change to all listeners.

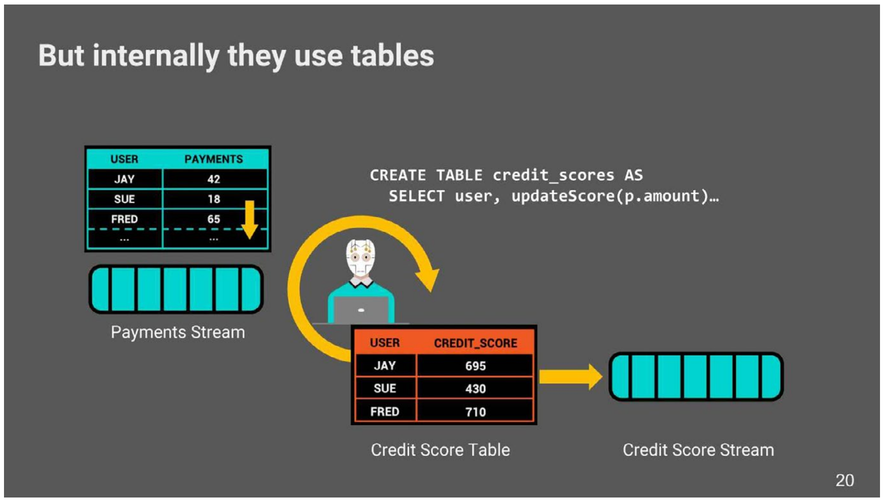
All in all, this makes events a far better way to represent in a distributed architecture.

Streams and Tables: two sides of the same coin?

So events provide a different type of data model — but counterintuitively, internally, a stream processor actually uses tables, much like a database.

These internal tables are analogous to the temporary tables traditional databases use to hold intermediate results, but unlike temporary tables in a database, they are not discarded when the query is done.

This is because streaming queries run indefinitely, there is no reason to discard them.



But internally they use tables

USER	PAYMENTS
JAY	42
SUE	18
FRED	65
...	...

```
CREATE TABLE credit_scores AS
  SELECT user, updateScore(p.amount)…
```

USER	CREDIT_SCORE
JAY	695
SUE	430
FRED	710

Figure 4: Stream processors use tables internally to hold intermediary results, but data in and out is all represented by streams

Figure 4 depicts how a stream processor might conduct credit scoring. There's a payment stream for accounts on the left. A credit-scoring function computes each user's credit score and keeps the result in a table. This table, which is internal to the stream processor, is much smaller than the input stream.

The stream processor continues to listen to the payments coming in, updating the credit scores in the internal table as it does so, and outputting the results via another event stream.

So, the thing to note is, when we use the stream processor, we create tables, we just don't really show them to anybody. It's purely an implementation detail. All the interaction is via the event streams.

This leads to what is known as the stream/table duality, where the stream represents history, every single event, or state change, that has happened to the system. This can be collapsed into a table via some function, be it a simple "group by key" or a complex machine learning routine.

That table can be turned back into an event stream by "listening" to it. Note that, most of the time, the output stream won't be the same as the input as the processing is usually lossy, but if we keep the original input stream in Kafka we can always rewind and regenerate.

So, we have this kind of duality: streams can go to tables, tables can go back to streams. In technologies like ksqlDB these two ideas are blended together: you can do streaming transformations from stream to stream, you can also create tables, or 'materialized views' as they are sometimes referred to and query those like a regular database. Some database technologies like MongoDB and RethinkDB are blending these concepts together too, but they approach the problem from the opposite direction.

How Stream Processors and Databases differ

A stream processor performs some operations just like a database does. For example, a stream-table join (see Figure 5) is algorithmically very similar to an equi-join in a database. As events arrive the corresponding value is looked up in the table via its primary key.

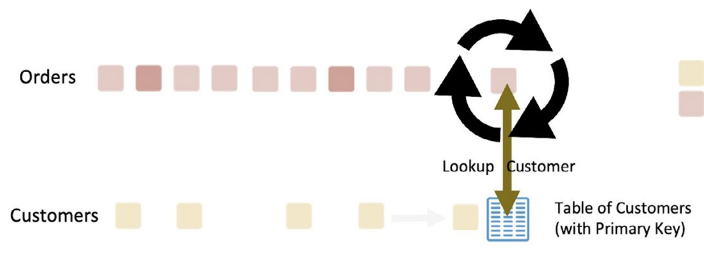


Figure 5: Joining a stream with a table.

However, joining two streams together using a stream-stream join is very different because we need to join events as they arrive, as depicted in Figure 6 below. As the events move into the stream processor, they get buffered inside an index and the stream processor looks for the corresponding event in the other stream. It also uses the event timestamp on the event to determine the order in which to process them so it doesn't matter which event came first. This is quite different from a traditional database.

Stream processors have other features which databases don't have. They can correlate events in time, typically using the concept of windows.

For example, we can use a time window to restrict which messages contribute to a join operation. Say we want to aggregate a stream of temperature measurements using windows that each span five minutes of time to compute a five-minute temperature average. It's very hard to do that inside a traditional database in a realtime way. A stream processor can also do more advanced correlations. For example, a session window is a difficult thing to implement in a database.

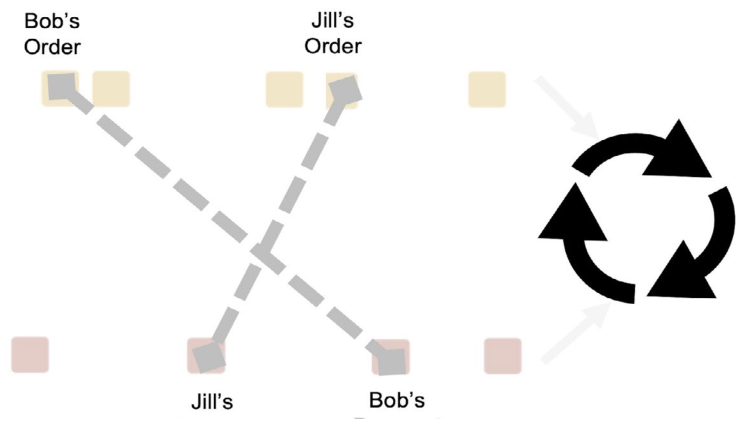


Figure 6: Joining two streams

A session has no defined length; it lasts for some period of time and dynamically ends after a period of inactivity. Bob is looking for trousers on our website, maybe he buys some, and then goes away — that's a session. A session window allows us to detect and isolate that amorphous period.

Another unique property of stream processors is their ability to handle late and out-of-order data. For example, they retain old windows which can be updated retrospectively if late or out of order data arrives. Traditional databases can be programmed to do similar types of queries, but not in a way that yields either accurate or performant realtime results.

The benefits of a hybrid, stream-oriented database

So by now, it should be clear that we have two very different types of query. In stream processing, we have the notion of a push query: queries that push data out of the system.

Traditional databases have this notion of a pull-query: ask a question and get an answer returned back. What's interesting to me is the hybrid-world that sits between them, combining both approaches. We can send a select statement and get a response. At the same time, we can also listen to changes on that table as they happen. We can have both interaction models.

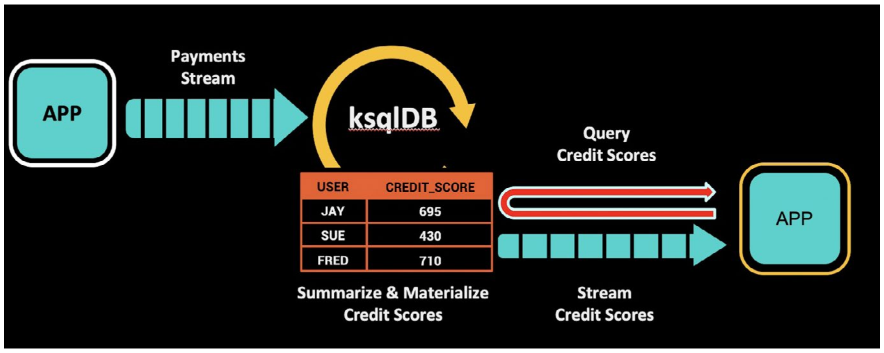


Figure 7: A materialized view, defined by a stream processor, which the second application interacts with either via a query or via a stream.

So, I believe we are in a period of convergence, where we will end up with a unified model that straddles streams and tables, handles both the asynchronous and synchronous and provides users with an interaction model that is both active and passive. There is, in fact, a unified model for describing all of this, which we can create using something familiar like SQL, albeit with some extensions — a single model that rules them all.

A query can run from the start of time to now, from now until the end of time, or from the start of time to the end of time. "Earliest to now" is just what a regular database does.

The query executes over all rows in the database and terminates when it has covered them all. "Now to forever" is what stream processors do today.

The query starts on the next event it receives and continues running forever, recomputing itself as new data arrives and creating new output records. "Earliest to forever" is a less-well-explored combination but is a very useful one. Say we're building a dashboard application.

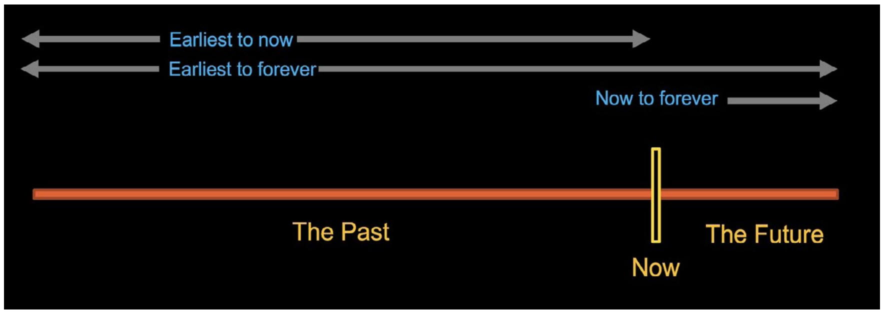
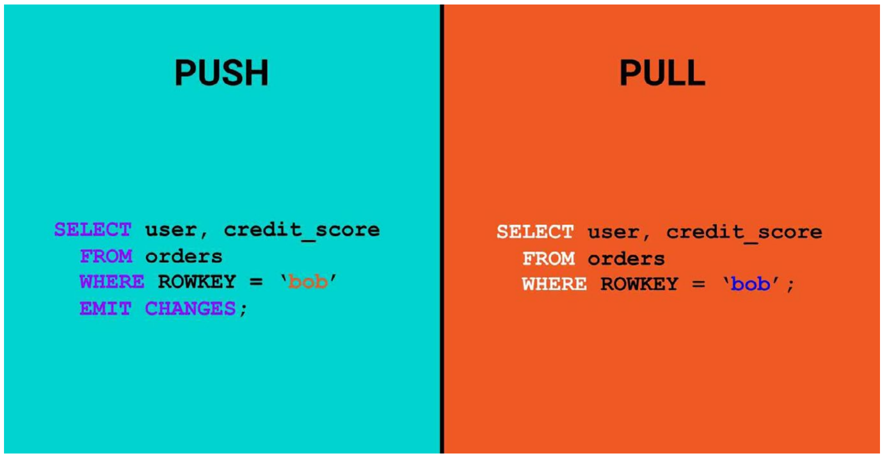


Figure 8: The unified interaction model.

With "earliest to forever", we can run a query that loads the right data into the dashboard, and then continues to keep it up to date. There is one last subtlety: are historical queries event-based (all moves in the chess game) or snapshots (the positions of the pieces now)? This is the other option a unified model must include.

Put this all together and we get a universal model that combines stream processing and databases as well as a middle ground between them: queries that we want to be kept up to date. There is work in progress to add such streaming SQL extensions to the official ANSI SQL standard, but we can try it out today in technologies like ksqlDB. For example, here is the SQL for push ("now to forever") and pull ("earliest to now") queries.

While stream processors are becoming more database-like, the reverse is also true. Databases are becoming more like stream processors. We see this in active databases like MongoDB, Couchbase, and RethinkDB. They don't have the same primitives for processing event streams, handling asynchronicity, or handling time and temporal



```
SELECT user, credit_score
  FROM orders
  WHERE ROWKEY = 'bob'
  EMIT CHANGES;
```

```
SELECT user, credit_score
  FROM orders
  WHERE ROWKEY = 'bob';
```

Figure 9: Push and pull queries in the unified interaction model.

correlations, but they do let us create streams from tables and compared to tools like ksqlDB they are better at performing pull queries, as they're approaching the problem from that direction.

So, whether you come at this from the stream processing side or the database side there is a clear drive towards a centreground. I think we're going to see a lot more of it.

The database: is it time for a rethink?

If you think back to where we started: Andreasen's observation of a world being eaten by software, this suggested a world where software talks to software. Where user interfaces, the software that helps you and me, is a smaller part of the whole package.

We see this today in all manner of businesses across ride sharing, finance, automotive — it's coming up everywhere.

This means two things for data. Firstly, we need data tooling that can handle both the asynchronous and the synchronous. Secondly, we also need different interaction models.

Models which push results to us and chain stages of autonomous data processing together.

For the database this means the ability to query passive data sets and get answers to questions users have. But it also means active interactions that push data to different subscribing services.

On one side of this evolution are active databases: MongoDB, Couchbase, RethinkDB, etc. On the other are stream processors: ksqlDB, Flink, Hazelcast Jet.

Whichever path prevails, one thing is certain: we need to rethink what a database is, what it means to us, and how we interact with both the data it contains and the event streams that connect modern businesses together.

TL;DR

- As companies become more automated, and their business processes become more automated, we end up with many applications talking to one another. This is a humongous shift in system design as it's about doing the work in a fully automated fashion by machines.
- In traditional databases, data is passive and queries are active: the data passively waits for something to run a query. In a stream processor, data is active and the query is passive: the trigger it's the data itself. The interaction model is fundamentally different.
- In modern applications, we need the ability to query passive data sets and get answers for users actions but we also need active interaction through data that is pushed as an event stream to different subscribing services.
- We need to rethink what a database is, what it means to us, and how we interact with both the data it contains and the event streams that connect it all together

Data Gateways in the Cloud Native Era

by **Bilgin Ibryam**, Product Manager at Red Hat

These days, there is a lot of excitement around 12-factor apps, microservices, and service mesh, but not so much around cloud-native data. The number of conference talks, blog posts, best practices, and purpose-built tools around cloud-native data access is relatively low. One of the main reasons for this is because most data access technologies are architectured and created in a stack that favors static environments rather than the dynamic nature of cloud environments and Kubernetes.

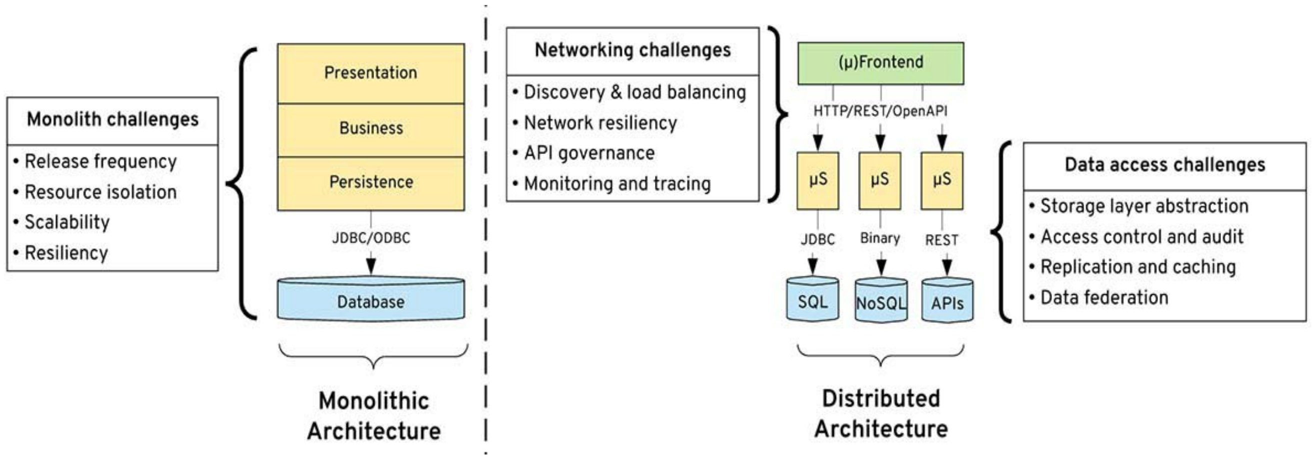
In this article, we will explore the different categories of data gateways, from more monolithic to ones designed for the cloud and Kubernetes. We will see what are the technical challenges introduced by the Microservices architecture and how data gateways can complement API gateways to address these challenges in the Kubernetes era.

Application architecture evolutions

Let's start with what has been changing in the way we manage code and the data in the past decade or so. I still remember the time when I started my IT career by creating frontends with Servlets, JSP, and JSFs. In the backend, EJBs, SOAP, server-side session management, was the state of art technologies and techniques. But things changed rather quickly with the introduction of REST and popularization of Javascript.

REST helped us decouple frontends from backends through a uniform interface and resource-oriented requests. It popularized stateless services and enabled response caching, by moving all client session state to clients, and so forth. This new architecture was the answer to the huge scalability demands of modern businesses.

A similar change happened with the backend services through the Microservices movement. Decoupling from the frontend was not enough, and the monolithic backend had to be



Application architecture evolution brings new challenges

decoupled into bounded context enabling independent fast-paced releases. These are examples of how architectures, tools, and techniques evolved pressured by the business needs for fast software delivery of planet-scale applications.

That takes us to the data layer. One of the existential motivations for microservices is having independent data sources per service. If you have microservices touching the same data, that sooner or later introduces coupling and limits independent scalability or releasing. It is not only an independent database but also a heterogeneous one, so every microservice is free to use the database type that fits its needs.

While decoupling frontend from backend and splitting monoliths into microservices gave the desired flexibility, it created challenges not-present before. Service discovery and load balancing, network-level resilience, and observability turned into major areas of technology innovation addressed in the years that followed.

Similarly, creating a database per microservice, having the freedom and technology choice of different datastores is a challenge. That shows itself more and more recently with the explosion of data and the demand for accessing data not only by the services but other real-time reporting and AI/ ML needs.

The rise of API gateways

With the increasing adoption of Microservices, it became apparent that operating such an architecture is hard. While having every microservice independent sounds great, it requires tools and practices that we didn't need and didn't have before.

This gave rise to more advanced release strategies such as blue/ green deployments, canary releases, dark launches. Then that gave rise to fault injection and automatic recovery testing.

And finally, that gave rise to advanced network telemetry and tracing. All of these created a whole new layer that sits between the frontend and the backend. This layer is occupied primarily with API management gateways, service discovery, and service mesh technologies, but also with tracing components, application load balancers, and all kinds of traffic management and monitoring proxies. This even includes projects such as Knative with activation and scaling-to-zero features driven by the networking activity.

With time, it became apparent that creating microservices at a fast pace, operating microservices at scale requires tooling we didn't need before. Something that was fully handled by a single load balancer had to be replaced with a new advanced management layer. A new technology layer, a new set of practices and techniques, and a new group of users responsible were born.

The case for data gateways

Microservices influence the data layer in two dimensions. First, it demands an independent database per microservice. From a practical implementation point of view, this can be from an independent database instance to independent schemas and logical groupings of tables. The main rule here is, only one microservice owns and touches a dataset. And all data is accessed through the APIs or Events of the owning microservice.

The second way a microservices architecture influenced the data layer is through datastore proliferation. Similarly, enabling microservices to be written in different languages, this architecture allows the freedom for every microservices-based system to have a polyglot persistence layer. With this freedom, one microservice can use a relational database, another one can use a document database, and the third microservice one uses an in-memory key-value store.

While microservices allow you all that freedom, again it comes at a cost. It turns out operating a large number of datastore comes at a cost that existing tooling and practices were not prepared for. In the modern digital world, storing data in a reliable form is not enough.

Data is useful when it turns into insights and for that, it has to be accessible in a controlled form by many. AI/ML experts, data scientists, business analysts, all want to dig into the data, but the application-focused microservices and their data access patterns are not designed for these data-hungry demands.

This is where data gateways can help you. A data gateway is like an API gateway, but it understands and acts on the physical data layer rather than the networking layer.

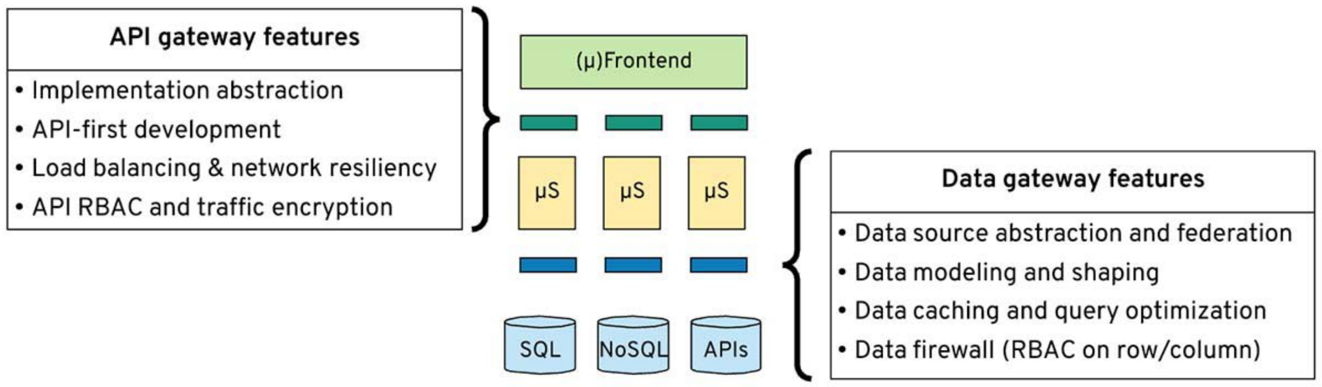
Here are a few areas where data gateways differ from API gateways.

Abstraction

An API gateway can hide implementation endpoints and help upgrade and rollback services without affecting service consumers. Similarly, a data gateway can help abstract a physical data source, its specifics, and help alter, migrate, decommission, without affecting data consumers.

Security

An API manager secures resource endpoints based on HTTP methods. A service mesh secures based on network connections. But none of them can understand and secure the data and its shape that is passing through them. A data gateway, on the other hand, understands the different data sources and the data model and acts on them. It can apply RBAC per data row and column, filter, obfuscate, and sanitize the individual data elements whenever necessary. This is a


API gateway features
• Implementation abstraction
• API-first development
• Load balancing & network resiliency
• API RBAC and traffic encryption
(μ)Frontend
μS
μS
μS
SQL
NoSQL
APIs
Data gateway features
• Data source abstraction and federation
• Data modeling and shaping
• Data caching and query optimization
• Data firewall (RBAC on row/column)

API and Data gateways offering similar capabilities at different layers

more fine-grained security model than networking or API level security of API gateways.

Scaling

API gateways can do service discovery, load-balancing, and assist the scaling of services through an orchestrator such as Kubernetes. But they cannot scale data. Data can scale only through replication and caching. Some data stores can do replication in cloud-native environments but not all. Purpose-built tools, such as Debezium, can perform change data capture from the transaction logs of data stores and enable data replication for scaling and other use cases.

A data gateway, on the other hand, can speed-up access to all kinds of data sources by caching data and providing materialized views. It can understand the queries, optimize them based on the capabilities of the data source, and produce the most performant execution plan. The combination of materialized views and the stream nature of change data capture would be the ultimate data scaling technique, but there are no known cloud-native implementations of this yet.

Federation

In API management, response composition is a common technique for aggregating data from multiple different systems. In the data space, the same technique is referred to as heterogeneous data federation. Heterogeneity is the degree of differentiation in various data sources such as network protocols, query languages, query capabilities, data models, error handling, transaction semantics, etc. A data gateway can accommodate all of these differences as a seamless, transparent data-federation layer.

Schema-first

API gateways allow contract-first service and client development with specifications such as OpenAPI. Data gateways allow schema-first data consumption based on the SQL standard. A SQL schema for data modeling is the OpenAPI equivalent of APIs.

Many shades of data gateways

In this article, I use the terms API and data gateways loosely to refer to a set of capabilities. There are many types of API gateways such as API managers, load balancers, service mesh, service registry, etc. It is similar to data gateways, where they range from huge monolithic data virtualization platforms that want to do everything, to data federation libraries, from purpose-built cloud services to end-user query tools.

Let's explore the different types of data gateways and see which fit the definition of "a cloud-native data gateway." When I say a cloud-native data gateway, I mean a containerized first-class Kubernetes citizen. I mean a gateway that is open source, using open standards; a component that can be deployed on hybrid/multi-cloud infrastructures, work with different data sources, data formats, and applicable for many use cases.

Classic data virtualization platforms

In the very first category of data gateways, are the traditional data virtualization platforms such as Denodo and TIBCO/Composite. While these are the most feature-laden data platforms, they tend to do too much and want to be everything from API management, to metadata management, data cataloging, environment management, deployment, configuration management, and whatnot. From an architectural point of view, they are very much like the old ESBs, but for the data layer. You may manage to put them into a container, but it is hard to put them into the cloud-native citizen category.

Databases with data federation capabilities

Another emerging trend is the fact that databases, in addition to storing data, are also starting to act as data federation gateways and allowing access to external data. For example, PostgreSQL implements the ANSI SQL / MED specification for a standardized way of handling access to remote objects from SQL databases.

That means remote data stores, such as SQL, NoSQL, File, LDAP, Web, Big Data, can all be accessed as if they were tables in the same

PostgreSQL database. SQL/MED stands for Management of External Data, and it is also implemented by MariaDB CONNECT engine, DB2, Teiid project discussed below, and a few others.

Starting in SQL Server 2019, you can now query external data sources without moving or copying the data. The PolyBase engine of SQL Server instance to process Transact-SQL queries to access external data in SQL Server, Oracle, Teradata, and MongoDB.

GraphQL data bridges

Compared to the traditional data virtualization, this is a new category of data gateways focused around the fast web-based data access. The common thing around Hasura, Prisma, SpaceUpTech, is that they focus on GraphQL data access by offering a lightweight abstraction on top of a few data sources. This is a fast-growing category specialized for enabling rapid web-based development of data-driven applications rather than BI/AI/ML use cases.

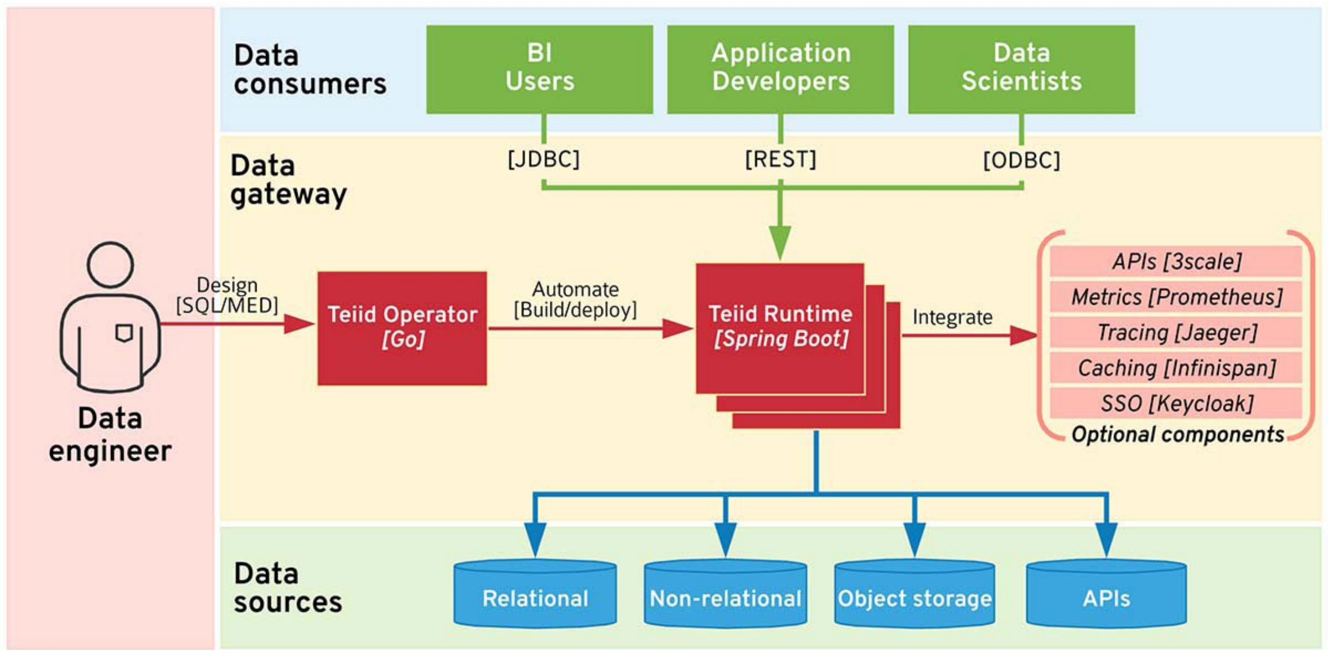
Open-source data gateways

Apache Drill is a schema-free SQL query engine for NoSQL databases and file systems. It offers JDBC and ODBC access to business users, analysts, and data scientists on top of data sources that don't support such APIs.

Again, having uniform SQL based access to disparate data sources is the driver. While Drill is highly scalable, it relies on Hadoop or Apache Zookeeper's kind of infrastructure which shows its age.

Teiid is a project sponsored by Red Hat and I'm most familiar with it. It is a mature data federation engine purposefully re-written for the Kubernetes ecosystem. It uses the SQL/MED specification for defining the virtual data models and relies on the Kubernetes Operator model for the building, deployment, and management of its runtime on Openshift.

Once deployed, the runtime can scale as any other stateless cloud-native workload on Kubernetes and integrate with other cloud-native projects. For example, it can use Keycloak for single sign-on and data roles, Infinispan for distributed caching needs, export metrics and register with Prometheus for monitoring, Jaeger for tracing, and even with 3scale for API management. But ultimately, Teiid runs as a single Spring Boot application acting as a data proxy and integrating with other best-of-breed services on Openshift rather than trying to reinvent everything from scratch.



Architectural overview of Teiid data gateway

On the client-side, Teiid offers standard SQL over JDBC/ODBC and Odata APIs. Business users, analysts, and data scientists can use standard BI/analytics tools such as Tableau, MicroStrategy, Spotfire, etc. to interact with Teiid. Developers can leverage the REST API or JDBC for custom built microservices and serverless workloads. In either case, for data consumers, Teiid appears as a standard PostgreSQL database accessed over its JDBC or ODBC protocols but offering additional abstractions and decoupling from the physical data sources.

PrestoDB is another popular open-source project started by Facebook. It is a distributed SQL query engine targeting big data use cases through its coordinator-worker architecture.

The Coordinator is responsible for parsing statements, planning queries, managing workers, fetching results from the workers, and returning the final results to the client. The worker is responsible for executing tasks and processing data. Recently the PrestoDB community split and created a fork called PrestoSQL that is now part of The Linux Foundation.

While forking is a common and natural path for many open-source projects, unfortunately, in this case, the similarity in the names and all of the other community-facing artifacts generates some confusion. Regardless of this, both distributions of Presto are among the most popular open-source projects in this space.

Cloud-hosted data gateways services

With a move to the cloud infrastructure, the need for data gateways doesn't go away but increases instead. Here are a few cloud-based data gateway services:

AWS Athena is ANSI SQL based interactive query service for analyzing data tightly integrated with Amazon S3. It is based on PrestoDB and supports additional data sources and federation capabilities too. Another similar service by Amazon is AWS Redshift Spectrum. It is focused around the same functionality, i.e. querying S3 objects using SQL. The main difference is that Redshift Spectrum requires a Redshift cluster, whereas Athena is a serverless offering that doesn't require any servers. Big Query is a similar service but from Google.

These tools require minimal to no setup, they can access on-premise or cloud-hosted data and process huge datasets. But they couple you with a single cloud provider as they cannot be deployed on multiple clouds or on-premise. They are ideal for interactive querying rather than acting as hybrid data frontend for other services and tools to use.

Secure tunneling data-proxies

With cloud-hosted data gateways comes the need for accessing on-premise data. Data has gravity and also might be affected by regulatory requirements preventing it from moving to the cloud. It may also be a conscious decision to keep the most valuable asset (your data) from cloud-coupling. All of these cases require cloud access to on-premise data. And cloud providers make it easy to reach your data. Azure's On-premises Data Gateway is such a proxy allowing access to on-premise data stores from Azure Service Bus.

In the opposite scenario, accessing cloud-hosted data stores from on-premise clients can be challenging too. Google's Cloud SQL Proxy provides secure access to Cloud SQL instances without having to whitelist IP addresses or configure SSL.

Red Hat-sponsored open-source project Skupper takes the more generic approach to address these challenges. Skupper solves Kubernetes multi-cluster communication challenges through a layer 7 virtual network that offers advanced routing and secure connectivity capabilities. Rather than embedding Skupper into the business service runtime, it runs as a standalone instance per Kubernetes namespace and acts as a shared sidecar capable of secure tunneling for data access or other general service-to-service communication. It is a generic secure-connectivity proxy applicable for many use cases in the hybrid cloud world.

Connection pools for serverless workloads

Serverless takes software decomposition a step further from microservices. Rather than services splitting by bounded context, serverless is based on the function model where every operation is short-lived and performs a single operation.

These granular software constructs are extremely scalable and flexible but come at a cost that previously wasn't present. It turns out rapid scaling of functions is a challenge for connection-oriented data sources such as relational databases and message brokers. As a result cloud providers offer transparent data proxies as a service to manage connection pools effectively. Amazon RDS Proxy is such a service that sits between your application and your relational database to efficiently manage connections to the database and improve scalability.

Conclusion

Modern cloud native architectures combined with the microservices principles enable the creation of highly scalable and independent applications.

The large choice of data storage engines, cloud-hosted services, protocols, and data formats, gives the ultimate flexibility for delivering software at a fast pace. But all of that comes at a cost that becomes increasingly visible with the need for uniform real-time data access from emerging user groups with different needs. Keeping microservices data only for the microservice itself creates challenges that have no good technological and architectural answers yet.

Data gateways, combined with cloud-native technologies offer features similar to API gateways but for the data layer that can help address these new challenges. The data gateways vary in specialization, but they tend to consolidate on providing uniform SQL-based access, enhanced security with data roles, caching, and abstraction over physical data stores.

Data has gravity, requires granular access control, is hard to scale, and difficult to move on/off/between cloud-native infrastructures. Having a data gateway component as part of the cloud-native tooling arsenal, which is hybrid and works on multiple cloud providers, supports different use cases is becoming a necessity.

TL;DR

- Application architectures have evolved to split the frontend from the backend, and further split the backend into independent microservices.
- Modern distributed application architectures created the need for API Gateways and helped popularize API Management and Service Mesh technologies.
- Microservices give the freedom for using the most suitable database type depending on the needs of the service. Such a polyglot persistence layer raises the need for API Gateway-like capabilities but for the data layer.
- Data Gateways act like API Gateways but focusing on the data aspect. A Data Gateway offers abstractions, security, scaling, federation, and contract-driven development features.
- There are many types of Data Gateways, from the traditional data virtualization technologies, to light GraphQL translators, cloud-hosted services, connection pools, and fully open source alternatives.

Combining DataOps and DevOps: Scale at Speed

by **Sam Bocetta**, Security Analyst, semi-retired, educates the public about security and privacy technology

Over the past decade, hundreds of organizations have made the shift to adopt the cloud as a way to obtain access to its automated, scalable, and on-demand infrastructure. The shift has changed software development requirement timeframes from weeks to mere minutes.

Around the same time, the cloud's scalability has also encouraged organizations to look at new development models. DevOps and the cloud have, together, broken down the walls between people and technology.

DevOps and continuous delivery processes have become widespread in most of our industries, enabling enterprises to radically increase the integrity, constancy, and output of new software.

Organizations are rushing to advance to the latest and best technological advancements. New strategies are being implemented through data-driven decision making and the infrastructure needed to integrate new breakthroughs - from artificial intelligence (AI) to machine learning and automation - is easily accessible.

But even in a world where software has become lightweight, scalable, and automated, there's one thing that prevents organizations from truly shining - and that is how readily their development teams can actually access their data.

In order to move quickly, development teams need consistent access to high-quality data.

If it takes days to refresh the data in a test environment, teams are caught in a difficult position: move slightly slower, or make concessions on quality at the detriment of your customers, subscribers, or users.

DataOps and DevOps - A Better Understanding of How It Works

DataOps is really an extension of DevOps standards and processes into the data analytics world. The DevOps philosophy underscores consistent and flawless collaboration between developers, quality assurance teams and IT Ops administrators. DataOps does the same for administrators and engineers who store, analyze, archive and deliver data.

To put it another way, DataOps is about streamlining the processes involved in processing, analyzing and deriving value from big data. This aims to break down the silos in the data storage and analytics fields which have historically isolated different teams from each other. Improved communication and cooperation between different teams will lead to faster outcomes and better time-to-value. DataOps is a way to automate the data processing and storage workflows in the same way that DevOps does when creating applications.

DevOps

DevOps combines IT / Ops and developers to work closely together in order to deliver software of a higher quality.

DevOps works in a simulated environment, and due to the radical advances of cloud-based developments, we can witness how organizations are now moving DevOps to their cloud environments. With additional continuous integration and automation of testing and delivery, DevOps breaks complicated tasks into much simpler ones.

DataOps

Adopting DevOps will require multiple alterations to your infrastructure. To make the most of DevOps, you'll want to move to a microservice-based workflow that benefits from containers and other progressive technologies - hence the massive rise in Software-as-a-Service (SaaS) offerings specifically due to the prior rise of DataOps. SaaS appeals to a massive entrepreneurial demographic, since almost anyone with knowledge or skills can help build a SaaS company.

DataOps also includes administrators and engineers to make use of next-generation data technology to develop their data storage and analytics infrastructure. They need solutions for data processing that are scalable and readily available - think cluster-based, robust storage.

The DataOps architecture also needs to be able to handle a number of workloads in order to achieve the same versatility as the DevOps implementation pipeline. Creating a data management tool set of diverse solutions, from log aggregators such as Splunk and Sumo Logic and Big Data Analytics applications such as Hadoop and Spark, is crucial to achieving this agility.

Embracing the Changes

We need to step away from organizing our teams and technologies around the tools we use to manage data, such as application creation, information management, identity access management and analytics and data science. Instead, we need to realize that data is a vital commodity, and to put together all those that use or handle data to take a data-centric view of the enterprise.

When building applications or data-rich systems, development teams learn to look past the data delivery mechanics and instead concentrate on the policies and limitations that control data in their organization, they can align their infrastructure more closely to enable data flow across their organization to those who need it.

To make the shift, DataOps needs teams to recognize the challenges of today's technology environment and to think creatively about specific approaches to data challenges in their organization. For example, you might have information about individual users and their functions, data attributes and what needs to be protected for individual audiences, as well as knowledge of the assets needed to deliver the data where it is required.

Getting teams together that have different ideas helps the company to evolve faster. Instead of waiting minutes, hours or even weeks for data, environments need to be created in minutes and at the pace required to allow the rapid creation and delivery of applications and solutions. At the same time, companies do not have to choose between access and security;

they can function assured that their data is adequately protected for all environments and users without lengthy manual checks and authorisations.

When done correctly, DataOps provides a cultural transformation that promotes communication between all data stakeholders. Data management will now be the collective responsibility of personnel, database managers, and development developers, as well as security and compliance officers. And although Chief Data Officers track data governance and efficiency, they seldom take any interest in non-production needs.

Innovation fails when no-one takes charge of cross-functional data management. Nevertheless, companies can ensure that confidential data is secure through powerful collaborative data systems and that the right data is made accessible to the right people, whenever and wherever they need it. Right through from the engineers who supply the data to the data scientists who analyze it, to the developers who check it.

The next ten years is set to reshape the face of computing as IoT devices, machine learning, augmented or virtual reality, voice computing, and more become common across all industries. With this change, more data, more privacy and security concerns and much more regulation will come into play.

This will put extraordinary pressure on organizations, and whoever comes up with solutions first will reap the benefits. With DataOps, IT can overcome the expense, sophistication and risk that comes with the management of data to become a business enabler while users can get the data they need to unlock their innovative capacity. If DevOps and cloud had been the key enablers of today's digital economy, DataOps is set to be the generator of our future data economy.

Uber and Netflix Show Us the Way Forward

While the way in which DataOps is implemented will be different in every organization, it can be instructive to look at the way in which the concept has been applied in real-world contexts. Two of the most pioneering companies in this regard have been Uber and Netflix, both of whom have been very open about the way in which they use DataOps within their businesses models.

Uber, for instance, uses a machine learning model (ML) known as Michelangelo to process the huge amounts of the data that the firm collects, and to share this across the organization. Michelangelo helps manage DataOps in a way similar to DevOps by encouraging iterative development of models and democratizing the access to data and tools across teams.

This system also makes use of a number of bespoke tools - one called Horovod, which coordinates parallel data processing across hundreds of GPUs, and Manifold, a visualization tool that is used to assess the efficacy of ML models.

Netflix is also a company that processes huge amounts of data every day, and one in which these data need to be accessible to thousands of individual clients. The core of the Netflix user experience is their recommendation engine, which currently runs in Spark. However, the company is continually testing new models in order to improve data availability and the accuracy of the recommendations that their ML algorithms provide.

Unlike many companies, however, Netflix runs these tests offline, and only deploy new models in consumer-facing systems once they have been proven to be effective. In other words, they are conscious to ensure the balance between stability and flexibility that characterises effective DataOps approaches.

Why Dataops and Devops Are a Match Made in Heaven

Today's millennial consumers are more aware of their brand engagement and not only want great products but also want great customized experiences when using these products. Many forward-thinking companies are therefore in the midst of the transition from a product economy to a service economy.

For example:

- Android and iPhone integrate customer support in their product bundle
- When buying a new vehicle, BMW includes daily car maintenance in the buying price
- Our smartphone technology now includes food delivery, maps, GPS and even online banking as a service with their product delivery

This shift from product to service as a priority is also reflected in the delivery of software, enabling companies to provide innovation, speed, reliability, frequency and operation on the customer's behalf.

With cloud automation, companies are now able to shift their focus and assimilate user experience seamlessly from machine-based functions to IaaS (infrastructure-as-a-service), PaaS (platform-as-a-service), and SaaS. DevOps helps this by removing the discrepancy between development and support.

We Can Shift from Stability to Agility

With an increase in production speed, the industry has been challenged to adjust their go-to-market strategy but mostly to shift their focus from stability and efficiency to innovation and flexibility. Faster technology innovations result in shorter production stages, creative designs and higher delivery rates.

The emergence of social media marketing and future technologies are shifting control away from production and keeping customers or users at the core. Branding and marketing mechanisms now react to consumer preferences rather than unlocking it. From SMEs to start-ups, companies need to encourage and support creative responsiveness and focus on waste reduction.

It's time for IT organizations to enable software as a service with the aid of DevOps methodologies and Cloud automation. DevOps combined with cloud helps to assess the quality of the customer's experience. This cross-department and cross-functional cooperation strengthens an organization's operations and helps them achieve the advantage in their market.

One thing digital transformation has taught us is that software and hardware have to work in unison. Each corporation must adapt to the combination of digital applications with material systems or components.

While DevOps offers advancements in software development and ongoing efficiency to its users, the cloud offers simplicity in the use of, and quality in the product by optimizing operational performance. As a result, DevOps in conjunction with Cloud fulfills user expectations with the help of sophisticated execution.

TL;DR

- DataOps is all about streamlining the processes that are involved in processing, analyzing and deriving value from big data.
- Development teams need to learn how to look past the data delivery mechanics and instead concentrate on the policies and limitations that control data in their organization.
- Uber and Netflix have both been very open about the way in which they use DataOps within their businesses models.
- Many forward-thinking companies have found themselves in the midst of the transition from a product-based economy to a service-based economy.