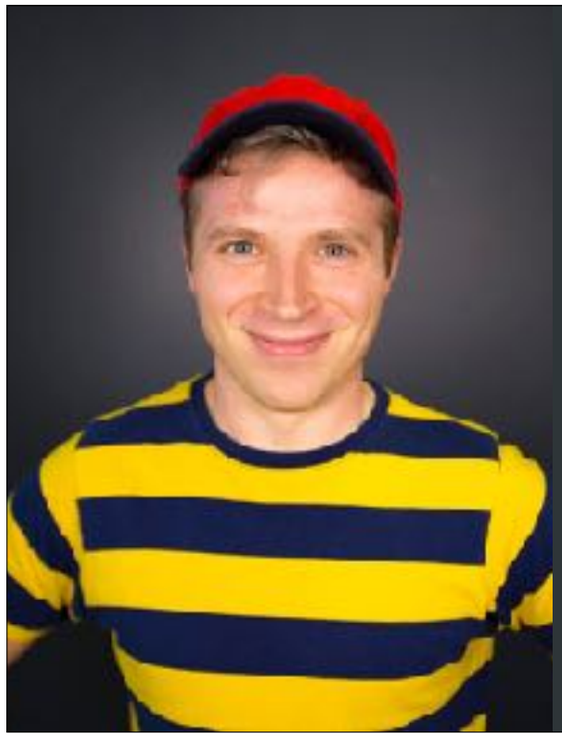
PYCON US 2017

ONE DATA PIPELINE TO RULE THEM ALL

IT'S 11 PM. DO YOU KNOW WHERE YOUR DATA IS?

Raise your hand if you have data.
Keep your hand up if you know all of the datasets you have, how to access them, how they arrive at all the places they live, how to combine them to derive new data or analysis…



@SKIMBREL

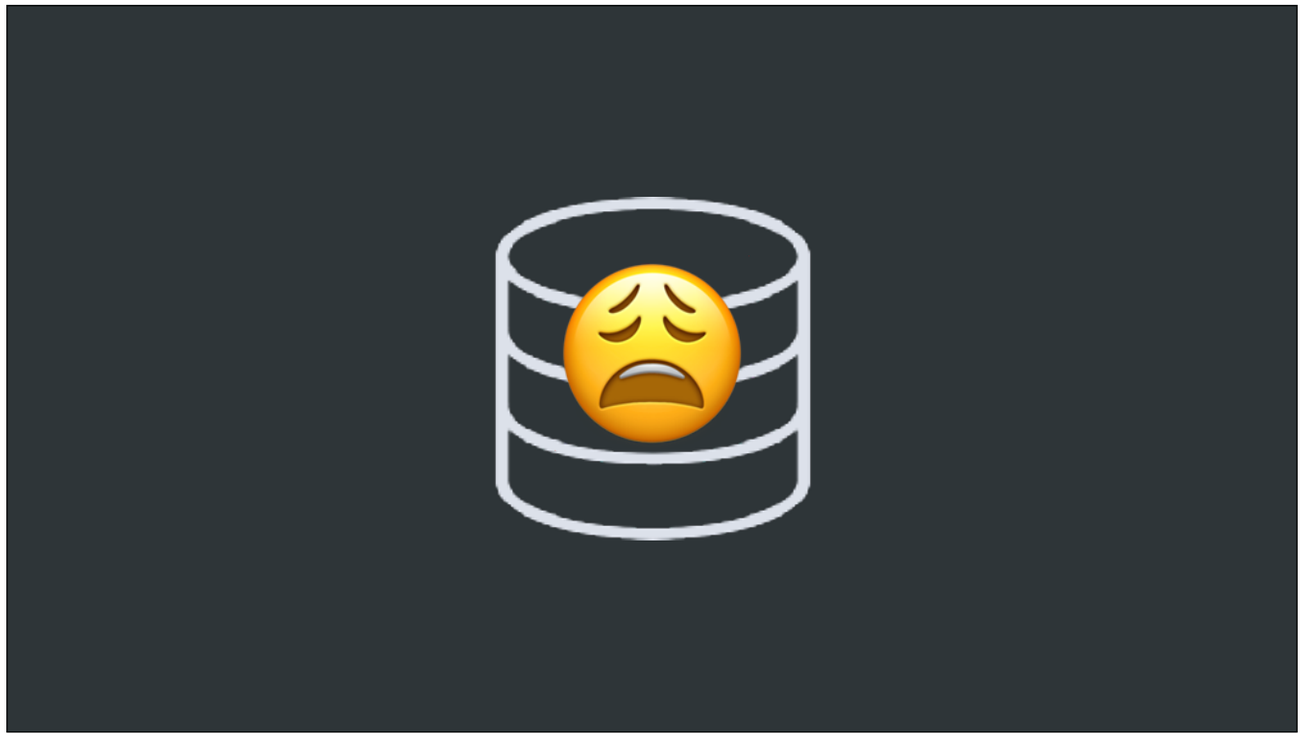
TWILIO // DATA PLATFORM

HEY, I'M SAM

Hey, I'm Sam. I lead the Data Platform team at Twilio, where we build and maintain a unified system for storing, retrieving, and doing computation on all of the data generated by all of our products.

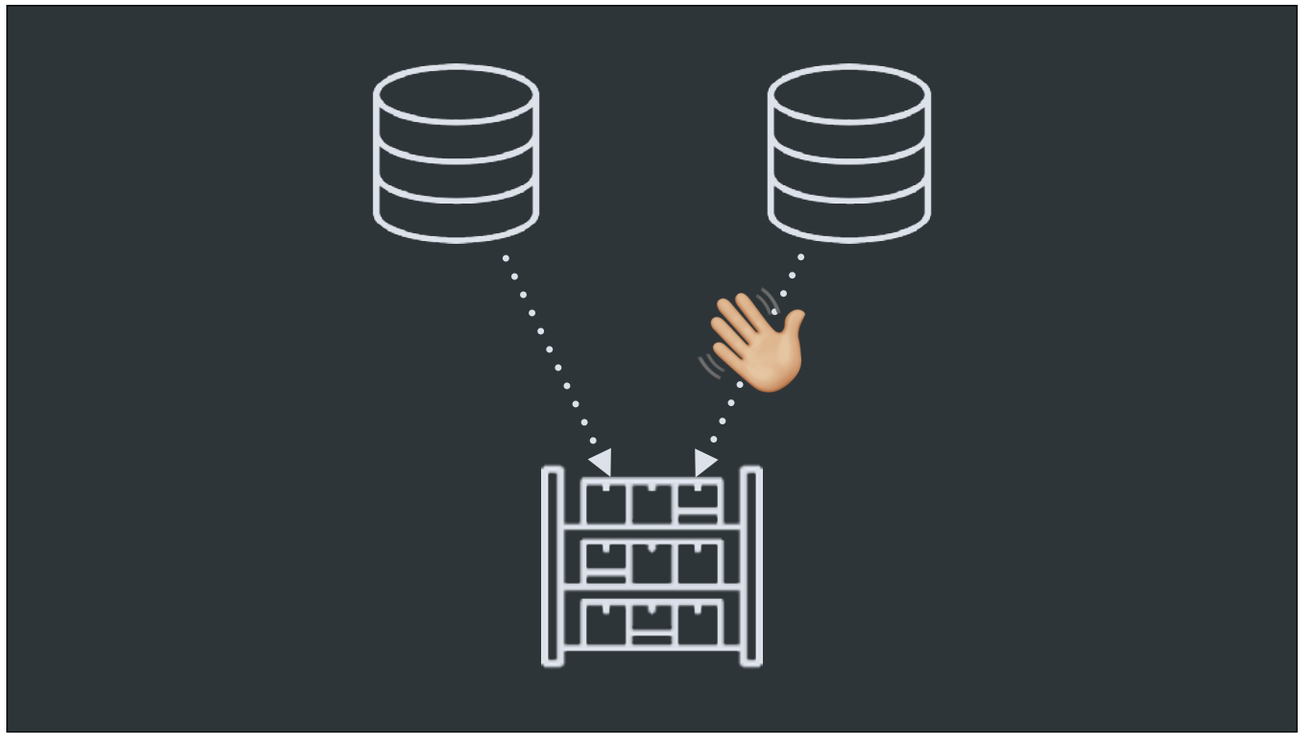
Let’s start with a story.

Here’s Team A’s relational DB. They set it up to store some data generated by their application. It works great.



Until the end of the quarter comes, and Team A's PM wants to run some queries to see how much their product's usage grew. And since the DB is built for single-row transactions, these giant queries are slow. PM: "Can we get the data somewhere that'd make this analysis faster?"

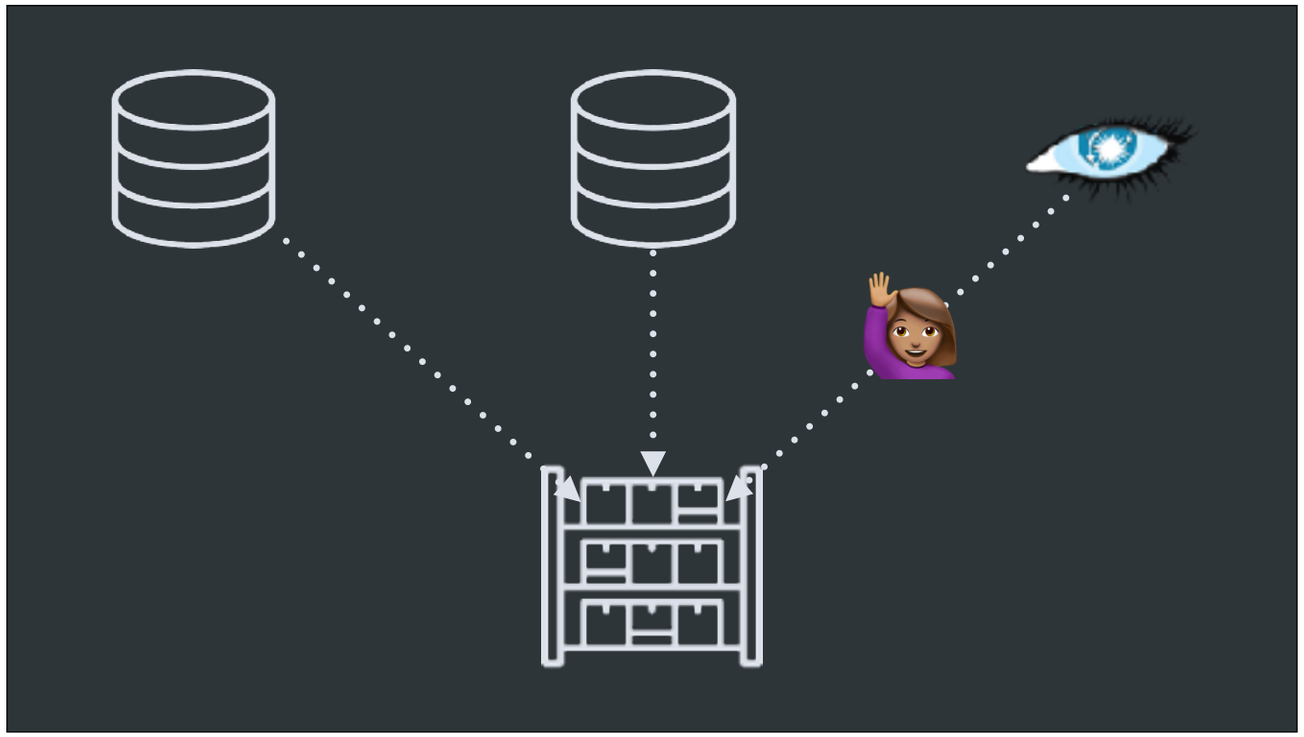
And now we have a data warehouse, copying data out of the app DB. And it works pretty well! Our column store warehouse is super fast at running those giant rollup queries and it's easy enough to copy data in with a cron job and a Python ETL (extract-transform-load) script someone knocked out in an afternoon.



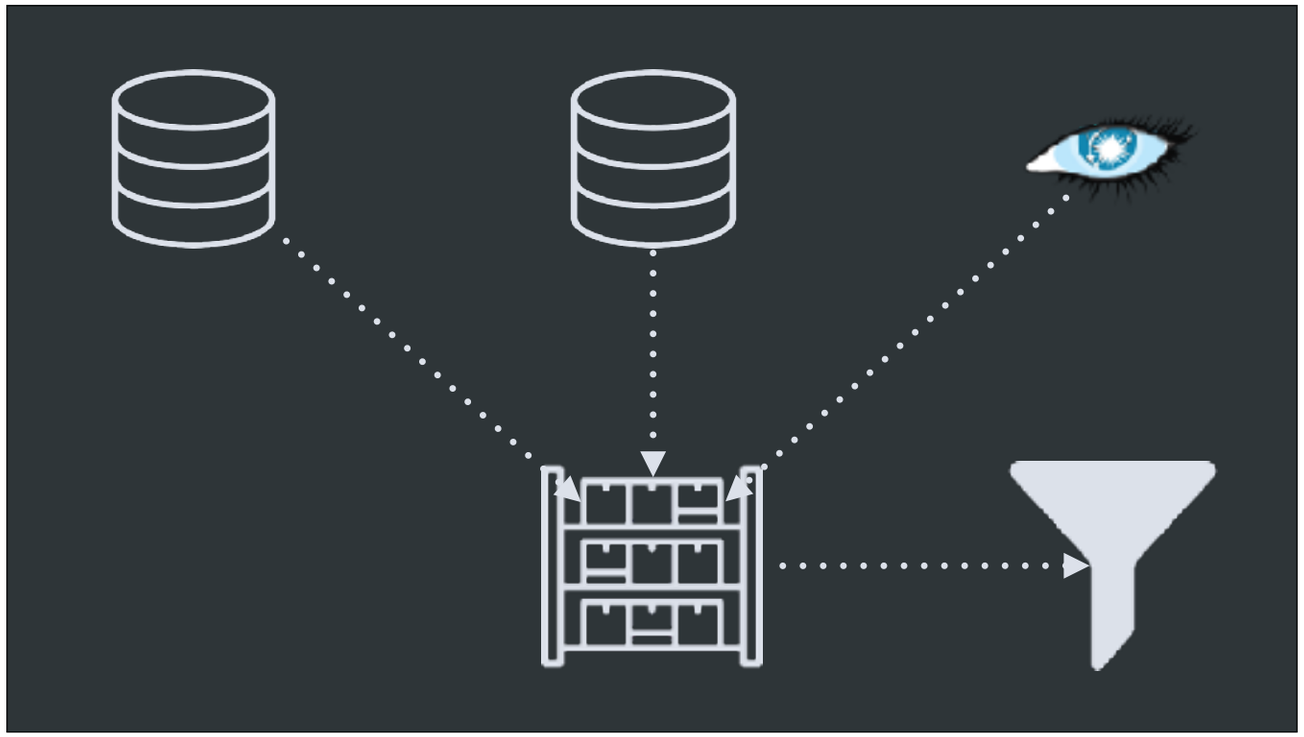
Then Team B comes along: “Hey, this looks great! Can we put some stuff in here?”

So they, or you, the newly-appointed custodian of the Data Warehouse, write another cron job to run a slightly different Python script and copy *their* data in.

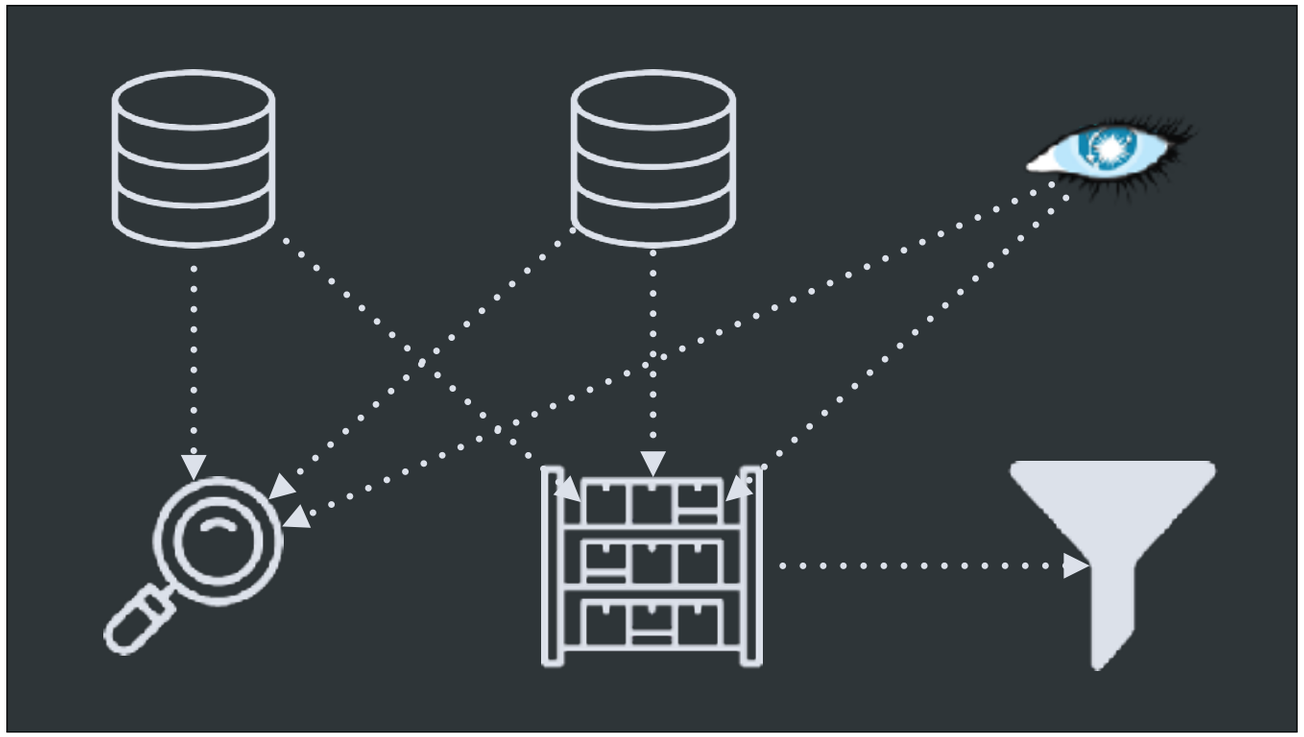
Now Team A decides to change some piece of their schema because they need to support a new feature. But everyone forgets about the data warehouse until a week later when someone on the accounting team (oh hey, when'd you start using this) says "Hey! Why isn't the reporting warehouse updating?". Shit! Someone updates it and life goes on.



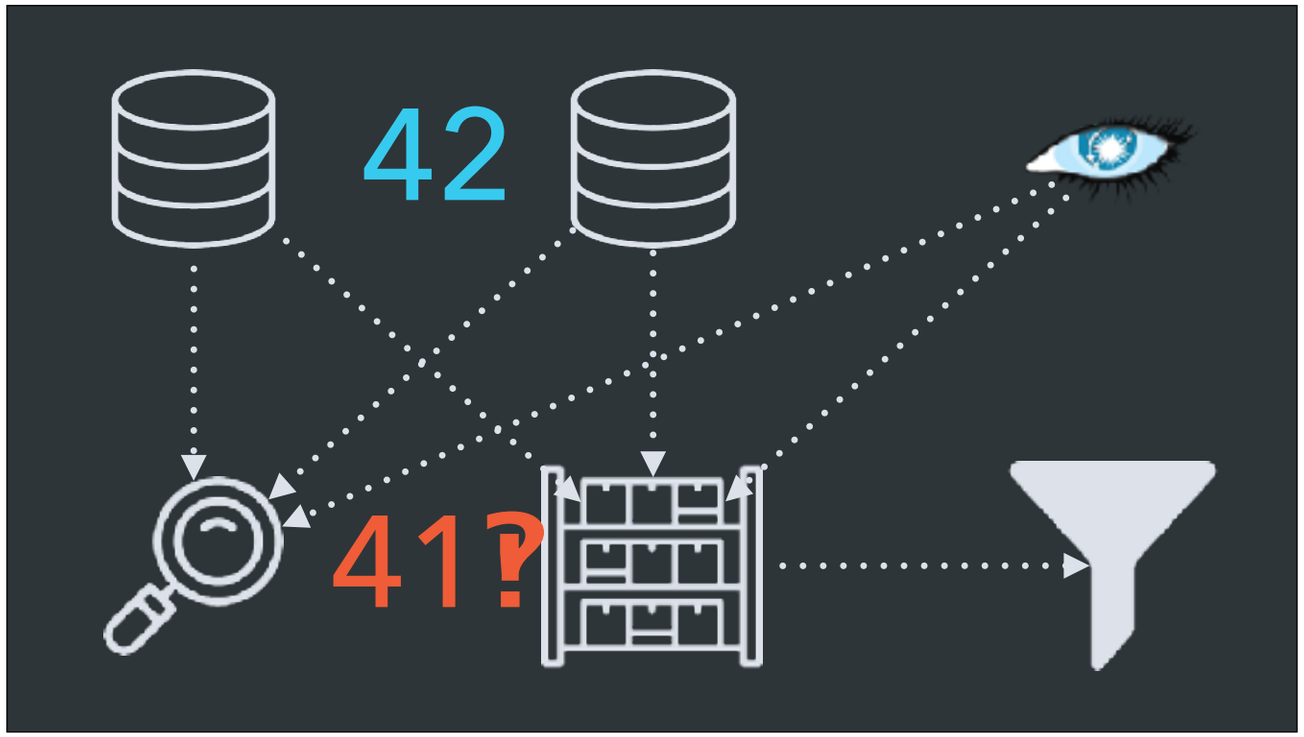
Team C: “Hey, we’ve got this Cassandra cluster we want to dump into the warehouse. Help?”



Team D: “We want to do realtime spam detection but run nightly training jobs for our model over all of our historical data.”



Team E: “Can has fulltext search plz?”
And just like that, our simple ETL flow isn’t so simple anymore. We’ve got a bunch of cron jobs into our warehouse, something rigged up to do realtime stream processing, and we’re running flat out just to keep up with the changes to all of the data.



And then, the accounting team comes back: “Why don’t these two systems agree on how many API calls/pageviews/whatever-it-is-we-sell-anyway we did last month?”

Yikes.

This story is very abstracted but pretty much all of these things have happened to us at Twilio, and I bet they’re familiar to a lot of other companies too.

WHAT HAPPENED?

So, what happened? We had everything working, but it was costing us a lot in developer time and machine time.

When we looked back at our legacy infrastructure, we were able to identify a number of high-level problems with it.

ARCHITECTURAL PROBLEMS

Multiple data sources

Multiple data sinks

N^2 custom connection paths

We had multiple systems we wanted to use as sources of data, and multiple places we wanted to copy it into. There was some code reuse, but each new type of system required major work to add, and even new datasets from systems we knew about had to be hand-configured. We weren't quite at the worst case but the growth trend was clear — with N independent systems to be connected individually, the connections grow as N^2.

ARCHITECTURAL PROBLEMS

Multiple data sources

Multiple data sinks

N^2 custom connection paths

No source of truth for schemas

There was no single source of truth for data schemas, so changing them was risky and required manual work to verify things were working correctly.

ARCHITECTURAL PROBLEMS

Multiple data sources

Multiple data sinks

N^2 custom connection paths

No source of truth for schemas

No correctness guarantees

And finally, we had no way to guarantee all of the data was correct in all of the places it appeared. Network and host failures as well as logic errors could all manifest as data being wrong on one side or the other.

OK. Given those problems, what are our requirements for a replacement architecture? We looked at what industry giants like LinkedIn and Netflix were doing, we looked at our own capabilities, and here's what we came up with to optimize our data architecture for developer productivity, scalability, and correctness.

ONE (AND ONLY ONE) WAY TO PUBLISH DATA

First up, we concluded that there should be one (and only one) way to publish data into the new platform from the applications generating it. Many of our prior headaches were brought on by inconsistencies in how we moved data between systems and having to duplicate effort across them when making changes, so that had to go. We wanted a single system to be the canonical path for records regardless of how many places they had to be written to.

COMMON STORAGE TOOLING

On the flip side, given that we often need to put the same data in multiple storage systems to support all the ways we want to query it, we couldn’t dictate a single method of getting that data into those systems. But we did feel it was important to provide common libraries and tooling to share as much of that logic as possible.

COMMON SCHEMAS

Schemas. I’m going to go into a lot of detail on this later, but for now I’ll just say it’s a lot easier to ensure everything works well together when there’s a single authoritative source of what constitutes a valid record for any given type of data.

VERIFIABLE DELIVERY AND CORRECTNESS

And finally, we wanted a way to make sure that all systems purporting to contain a given set of data had the same data — if something is present and correct in one, it should be the same everywhere else it's expected to appear.

Since the title of this talk includes the words “data pipeline”, you might have guessed that we chose an event pipeline architecture, which we did.

From a thousand meters up, here’s what’s going on.

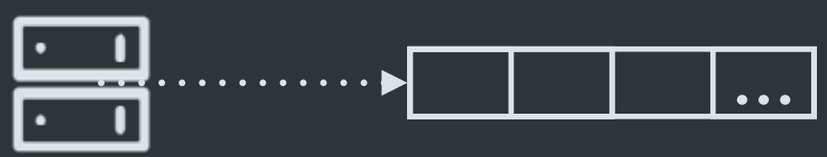
EVENT SOURCING ARCHITECTURE

In an event-sourcing architecture, all data is modeled as an ordered series of events that happened. For log-structured data (web requests, ad clicks, etc.) this is a very natural model; for data where records change over time (configuration data, things that change state), it requires viewing the records as an ordered sequence of changes to the data, from which the original data can be reconstructed.

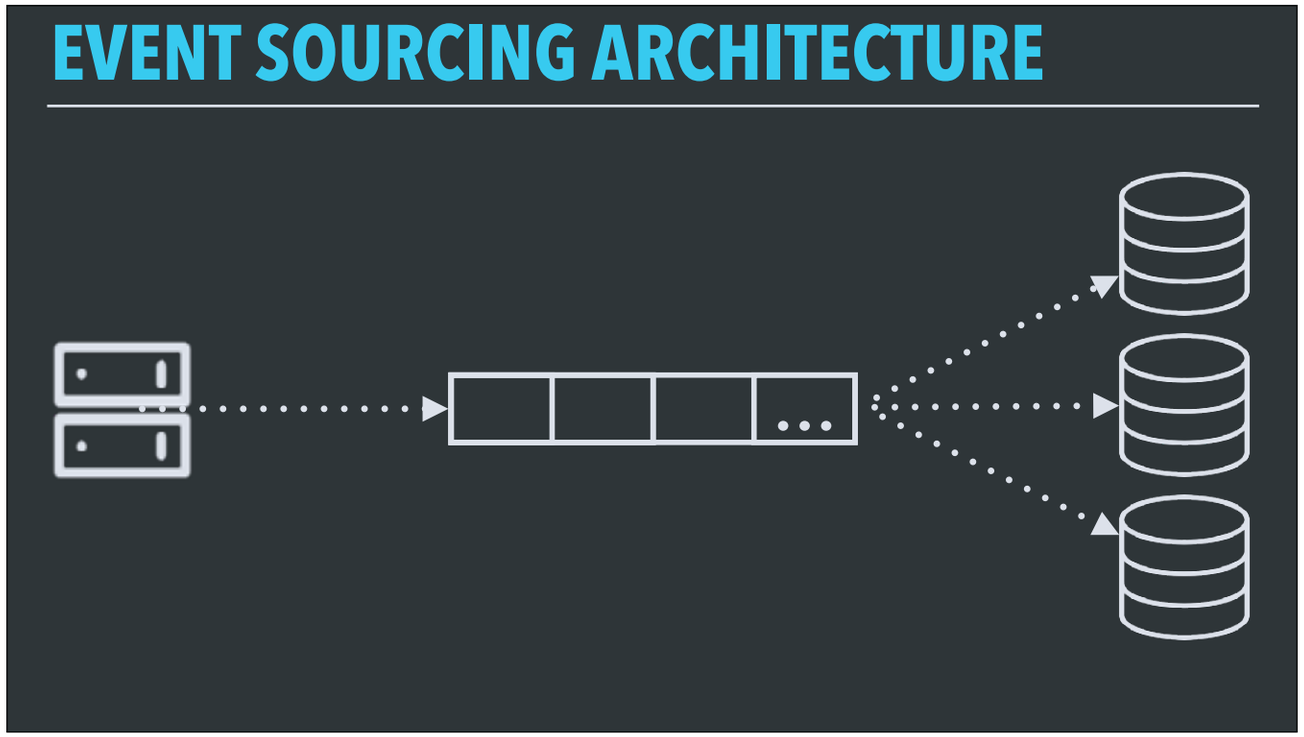
EVENT SOURCING ARCHITECTURE

So, we have some source of events. Let’s say it’s a web server, handling requests and generating a log event every time a user views a page.

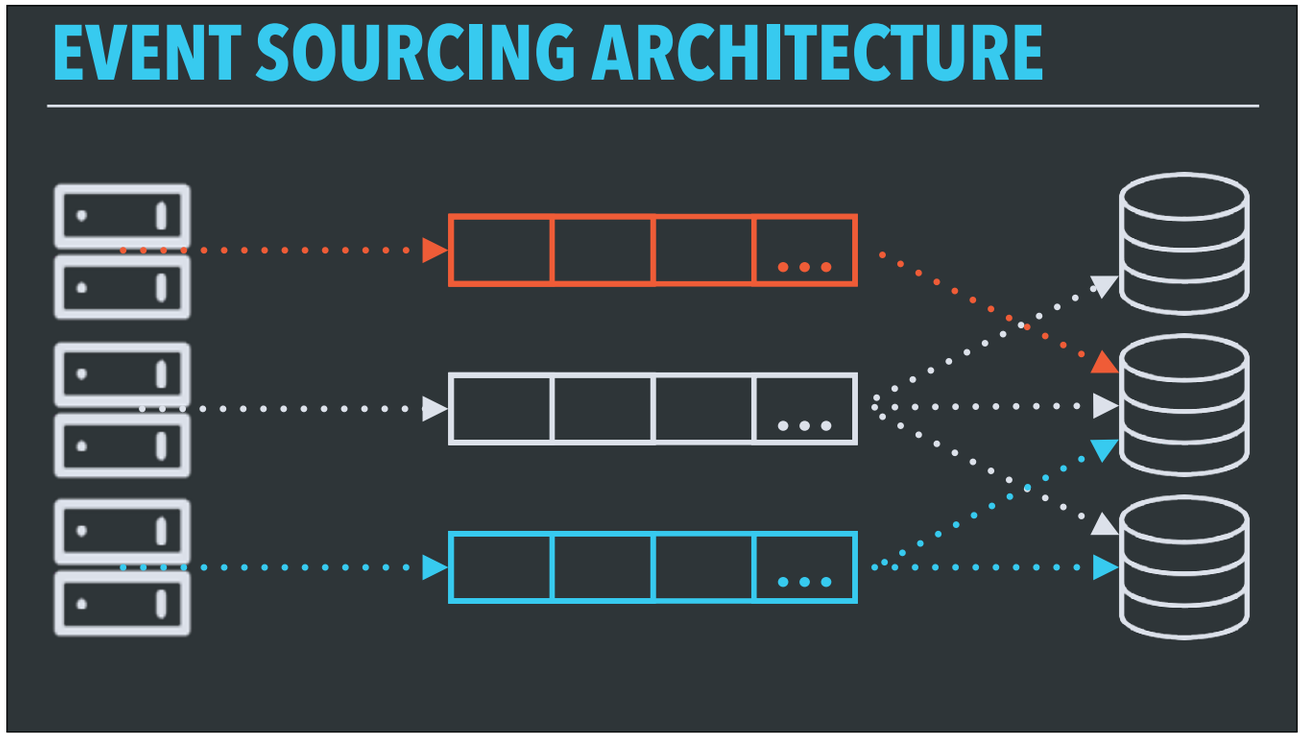
EVENT SOURCING ARCHITECTURE



That server emits a record onto a system that stores all of the events in the order they were received as a first-in, first-out queue.



All systems interested in that type of subscribe to that single stream and consume the events in the same order they were published in.
These don’t have to be the same type of system — one consumer might be a relational database providing near-realtime access to individual rows, while others could be writing a permanent archive of the data to a cold-storage system, doing realtime computation to derive new information, and so on.

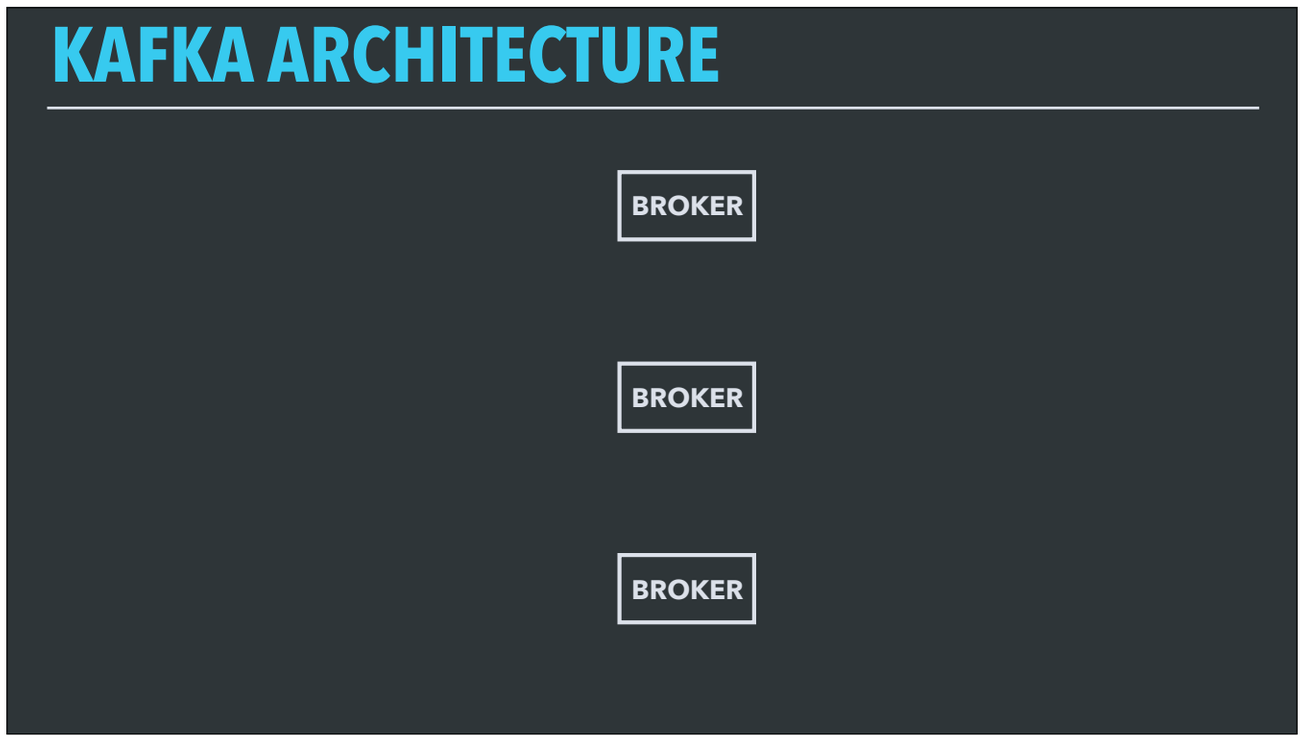


Just like we have multiple systems at the “back” of the queue, we can have multiple distinct systems sending events to their own streams, and set up each consuming system to only read out the event streams they’re interested in.

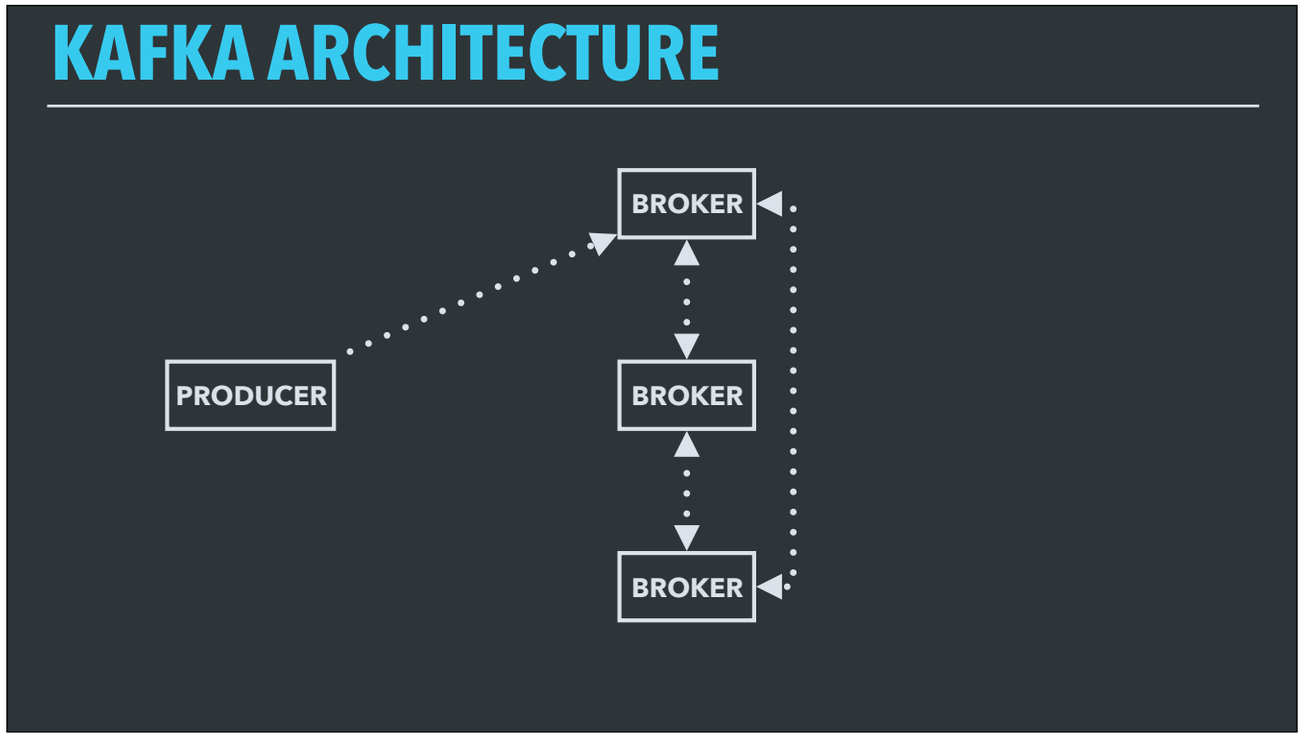
Apache Kafka is the backbone of this architecture. Kafka is a horizontally scalable, fault-tolerant, and very high-throughput streaming message platform.

For those not familiar, Kafka originated at LinkedIn, who currently use it to move over one trillion events per day. It’s become wildly popular since going open-source, and is used by companies ranging from Netflix to Goldman Sachs and, since I’m up here talking about it, Twilio.

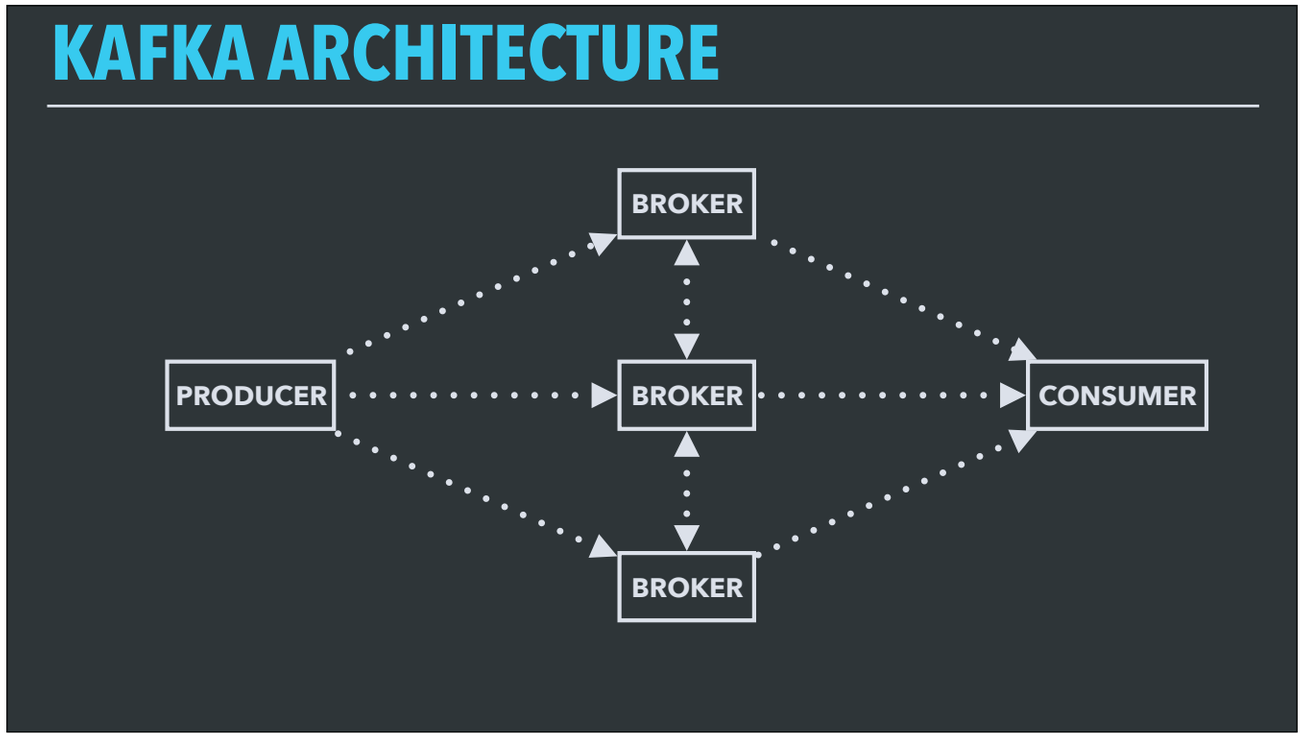
KAFKA ARCHITECTURE



A Kafka deployment consists of a set of *broker* nodes that host *topics*, which are streams of ordered events. Brokers accept writes to topics from *producer* nodes, persisting the produced events to disk and replicating amongst themselves for durability.



A Kafka deployment consists of a set of *broker* nodes that host *topics*, which are streams of ordered events. Brokers accept writes to topics from *producer* nodes, persisting the produced events to disk and replicating amongst themselves for durability.



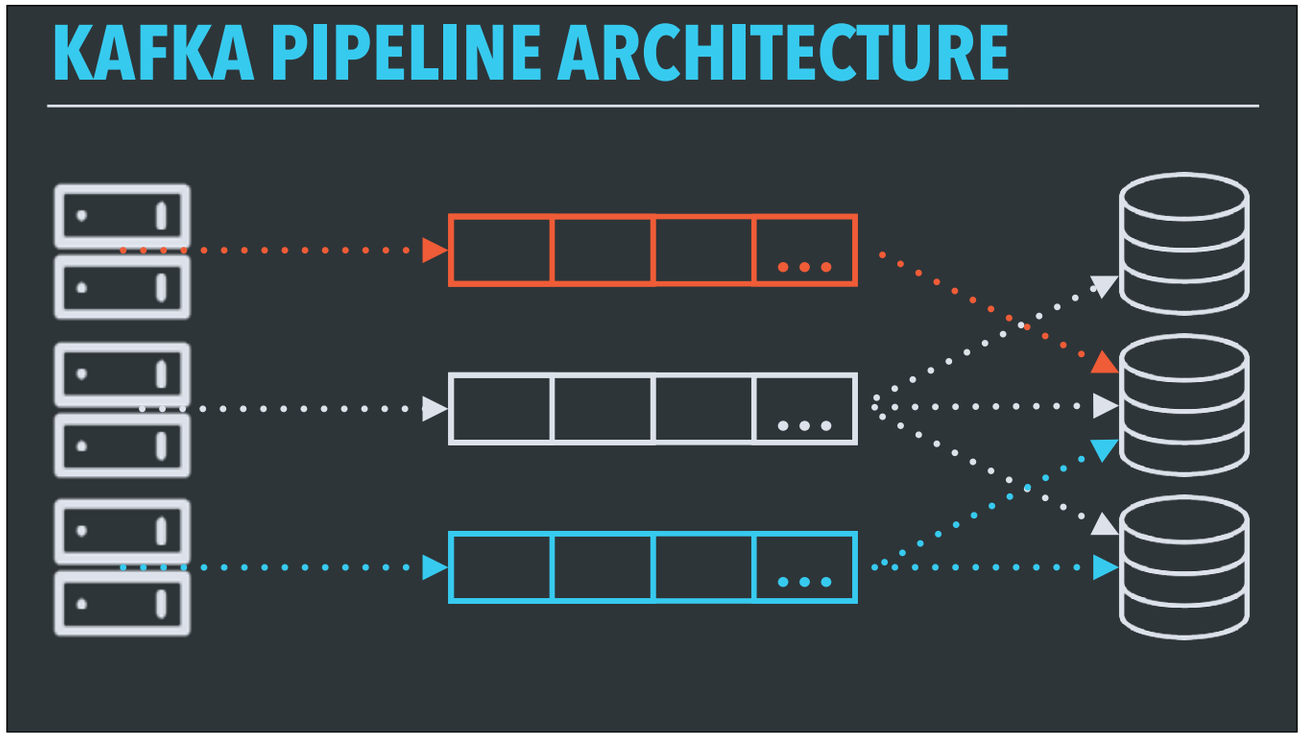
Consumer processes connect to the broker nodes, which deliver events from topics in the same order they were produced. As they process events, consumers send back acknowledgements to brokers of the latest event offset in the topic that they've successfully processed. If a consumer fails and restarts or is replaced, it will resume processing from the latest offset it committed to the brokers. In this way, Kafka guarantees at-least once delivery of each message sent to a topic to all consumers of that topic.

MANAGED KAFKA(-LIKE) SERVICES

Heroku

AWS Kinesis

Kafka is fairly straightforward to operate, but if you’re into managed services, Heroku has a Kafka add-on. Amazon’s Kinesis service also exposes similar APIs and contracts to Kafka.



So to recap: using Kafka as a durable, guaranteed-delivery event bus, we treat every dataset as a separate topic of events produced by the application systems generating them, and connect consumers as needed for each data storage system or stream processing application.

EVENT SOURCING: WHAT THE BLOG POSTS DON'T TELL YOU

So far, this is pretty standard stuff. You can find a dozen blog posts on it from LinkedIn, Netflix, Confluent, and a bunch of other people. I'd like to go into the things we've learned that are less obvious from the blog posts.

SCHEMAS ARE IMPORTANT. LIKE, REALLY REALLY IMPORTANT.

OK. This is probably the strongest opinion in this talk, so here goes. You should absolutely, by all means, definitely use a strongly-typed serialization format with a defined schema and validation library on both sides of your event bus.

There are a lot of systems out there (Hi, MongoDB!) that just let you chuck arbitrary documents at them.

This is tempting. You can get started really quickly with these systems.

But there’s a cost. Your schemas, and enforcement thereof, moves from a single place to many scattered places in your application code. It opens the possibility of sending or storing data that you can’t process later or in other systems.

To give an example: let’s say we have an application writing data somewhere, and a batch job, written in a different language, that picks it up and does something with it — let’s say it generates a report for the sales team.

If someone makes a change and drops a field from the data in the application without telling the batch job’s maintainer, and the batch depends on it, well… boom! Now someone has to go pick up the pieces, fix the code, and rerun the batch. And that’s if you’re lucky. If you’re unlucky, it’s a *silent* boom in the form of incorrect, missing, or corrupt data, and let’s hope that data wasn’t driving your revenue numbers.

SCHEMA LIBRARIES

Avro

Protocol Buffers

Thrift

msgpack

…

Many, many open-source libraries for schema validation and serialization exist.
Please: pick one, and use it everywhere that you transmit data between systems.

AVRO

Multiple language platforms

Dynamic and static bindings

Automatic cross-grading

Compact binary serialization

We ended up choosing Avro, for a number of reasons:
It has official cross-platform support for Java and Python, plus community-supported libraries for just about everything else. The Java library offers both dynamic and static types (Map or codegen). When deserializing an object, the library will automatically convert between compatible versions of the same schema. And finally, it offers a compact binary serialization format that does not include a copy of the schema tagged to every record.

ENFORCE SCHEMAS AT PRODUCE TIME

Last word on record schemas: enforce them when records are produced, without exception. Doing this provides a very powerful guarantee: every record coming down the topic will be valid for consumers to deserialize. It keeps the onus of data validation on producers, where it's easier to test and update since most data types will only be produced by one or a small number of codebases but might be consumed by many more. Within a single topic, only make backwards-compatible changes to a schema — if you need to break compatibility, start a new topic and migrate consumers after they've finished the old one.

With these rules and guarantees in place, the worst that can happen when a change gets missed by a consuming application is a delay in data availability.

TOPIC METADATA

What schema is in this topic?

The record schema alone isn’t the only information that’s useful to know about a topic and the records in it.

Given a Kafka topic name, we need to find out what schema applies to the records in it and what versions of that schema are available.

TOPIC METADATA

How do we resolve duplicates?

Next up — distributed systems are tricky. In the face of network partitions and host failures, we aren't guaranteed that we'll produce or consume a record only once. What's more, even ordering records can be tricky. Kafka only knows about the order in which records arrived at the broker, so if multiple hosts are producing onto the same topic, we'll have to sort the ordering out when we consume the records.

TOPIC METADATA

What fields uniquely ID a record?

What fields and logic let us choose among or merge multiple versions?

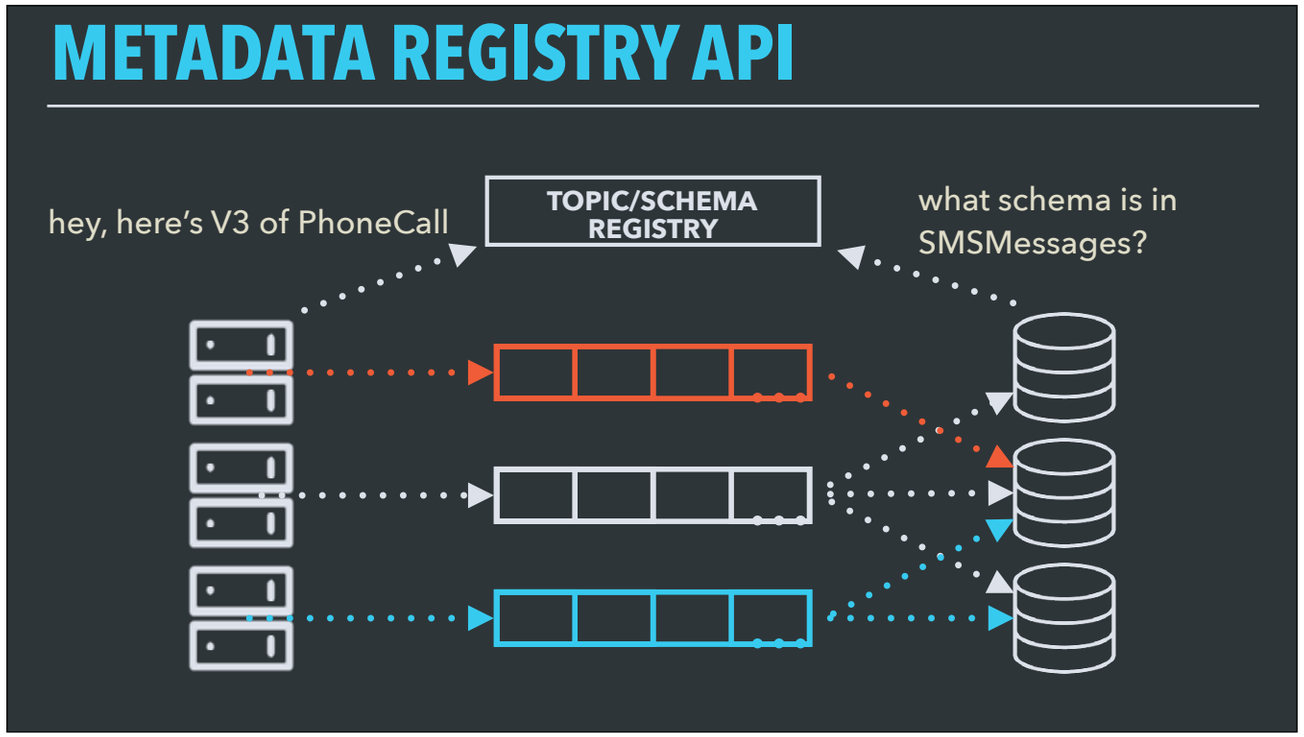
The first thing we need to know in order to do this is what uniquely identifies a record.
After that, we need to know what logic to apply to choose a winner or merge conflicting versions.
One more tip here: wall clocks are not to be trusted across multiple hosts, so don’t just use “latest timestamp wins”. Vector clocks are your friend here, but require some source of serialization and locking for concurrent operations on the same object.

TOPIC METADATA

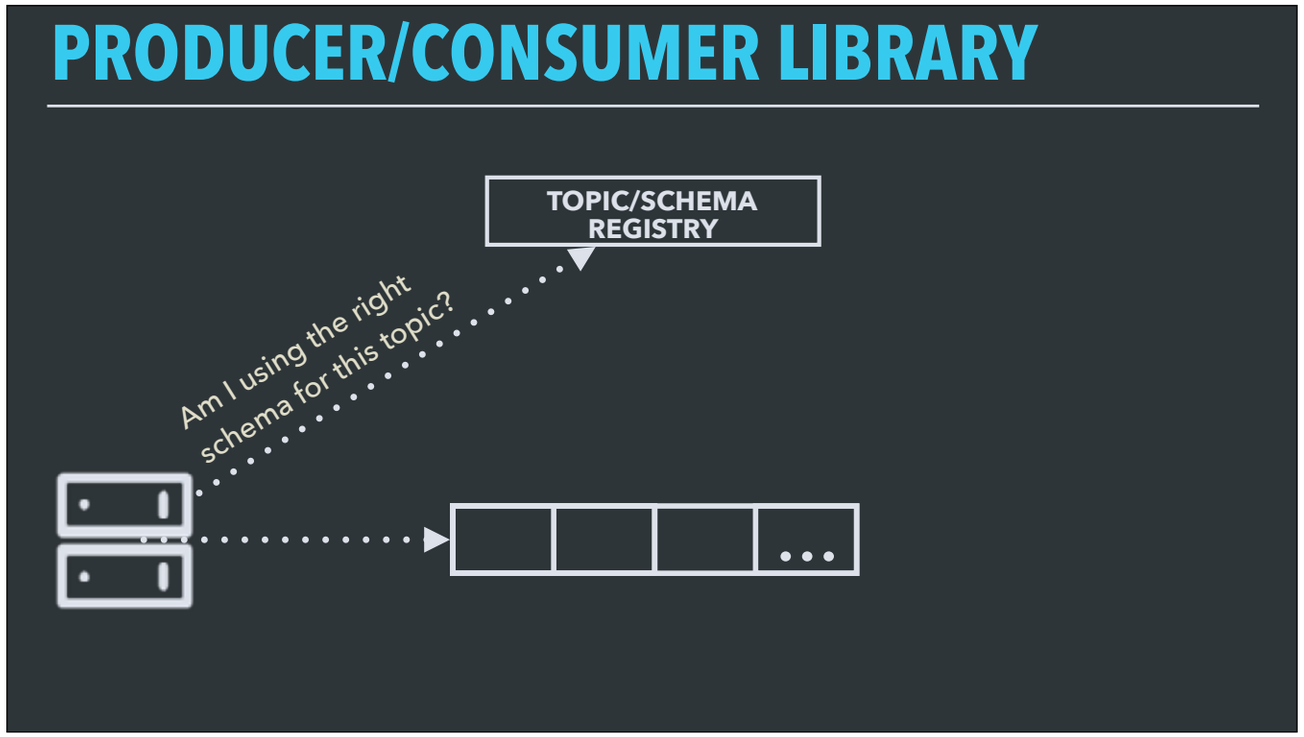
What tells us that the data is correct?

And finally, it’s useful to know how to verify that a data set is complete and correct. We can reuse the unique key we specified before for a first pass at reconciliation between different consumer systems — COUNT DISTINCT is always a good first step. If there are additional checks we can perform, we’ll want to know the aggregate operations and which fields to run them over.

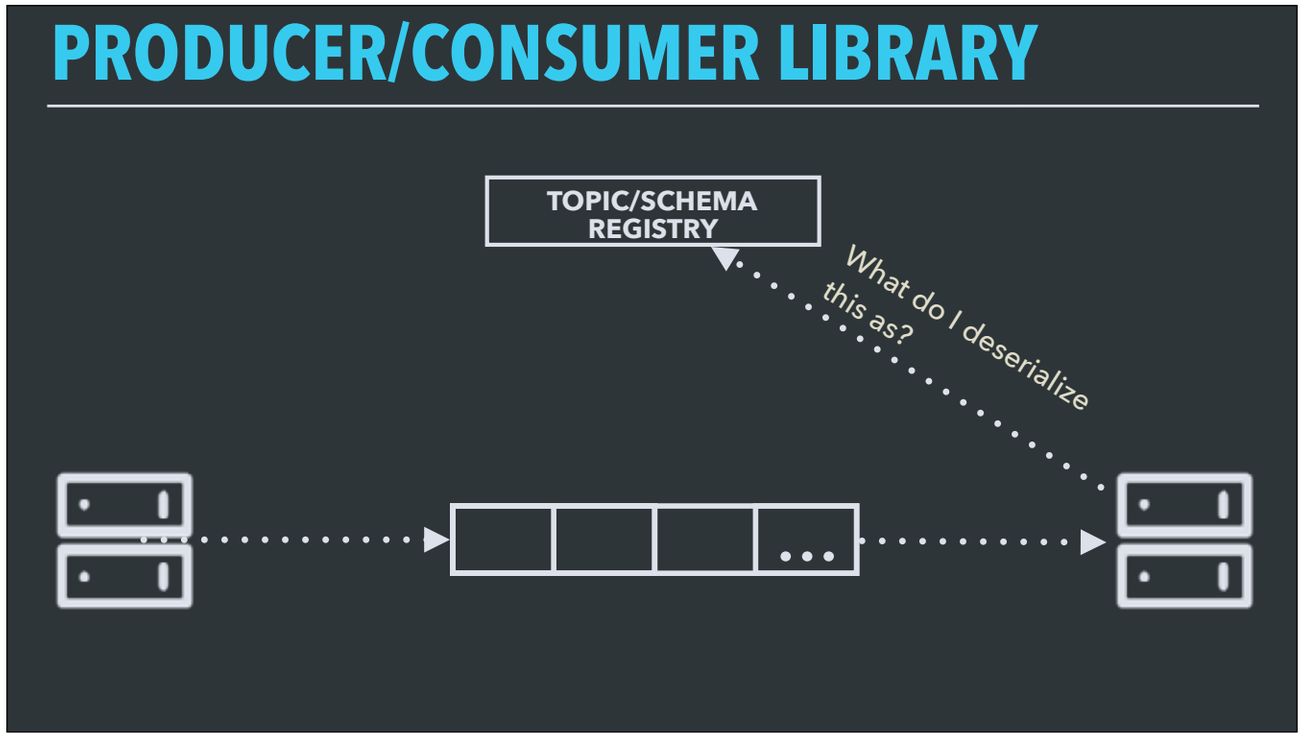
So I’ve covered what we started off with, what we disliked about our original systems, what we wanted out of our new architecture, and the core abstractions involved in a Kafka-based event pipeline. Next, I’d like to tell you about the libraries and systems we built to augment our Kafka cluster and connect it to our data sources and sinks.



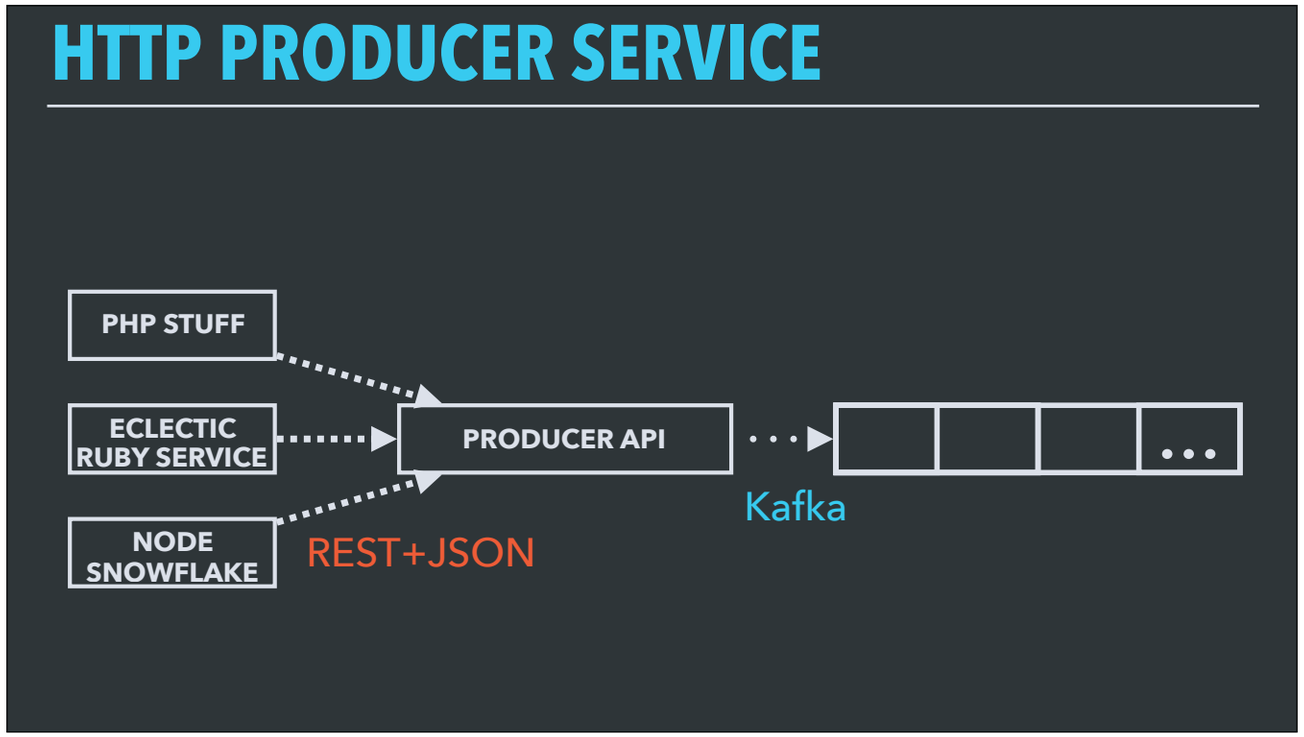
Storing the record schemas and all of the additional metadata about uniqueness, ordering, and correctness in a registry service gives us programmatic access to that data from any system producing or consuming records.
This system doesn't handle records itself — it just stores schemas for records and the extra metadata for topics, validates that new versions of a schema are compatible with previous ones, and serves this metadata back out to any system that needs it.



Kafka comes with producer and consumer implementations for many programming languages, but they, like Kafka itself, work at the level of byte streams. Kafka brokers don’t view records in a topic as anything other than binary blobs, so we built a wrapper around the interface to handle serialization and validation.
Our producer library accepts Avro record objects, validates that they conform to an accepted schema for the requested topic, and serializes to Avro’s binary encoding (with an extra byte on the front to signal the exact version of the schema).



The consumer interface does the opposite — looks up the topic in the registry and retrieves the schema for the version specified on the serialized record it just got, and deserializes it back to a native object for the application code.



Twilio has a polyglot development environment: we run a lot of Python, a lot of Java and Scala, some PHP, and there's even a little Ruby, Node, and Go scattered around. Our data pipeline libraries support Python and Java, but porting the logic to four more platforms would've been a lot of effort for diminishing returns. We also already have very well-defined and supported standards for internal RESTful HTTP APIs with JSON serialization. So we designed a simple service to produce to Kafka topics in response to HTTP requests, and anyone needing to write to an event topic from these systems can just chuck records at that API as JSON.

CONSUMER SYSTEMS

Archival

Warehousing and structured analytics

Batch processing

Ad-hoc analysis

Stream processing

Online query systems

That about covers it for producing records. What do we do with them now?

This breaks down into roughly six types of system. First up, we can archive a dataset to inexpensive and scalable cold storage. We use this primary archive data to power three more use cases: we can load it into data warehouses for doing analysis on structured subsets of the data; we can run batch processing jobs over arbitrarily large amounts of data with on-demand computational capacity; and we can do ad-hoc analysis of archived data without needing to load it into a warehouse. Next up is stream processing: since Kafka is high-throughput and low-latency, we can connect stream applications directly as Kafka consumers to process records close to realtime. And finally, we can connect transaction-processing databases like MySQL or ElasticSearch to provide access to small amounts of data with very low end-to-end latency from the time data is produced to when it is available to be retrieved.

CONSUMER SYSTEMS

Archival

Warehousing and structured analytics

Batch processing

Ad-hoc analysis

Stream processing

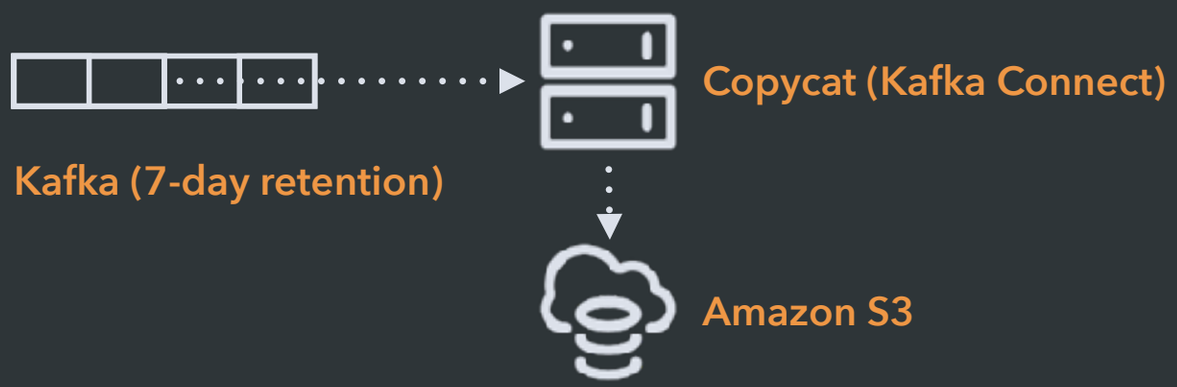
Online query systems

Let’s go through these in more detail, starting with archival.

ARCHIVAL: TWILIOFS DATA LAKE

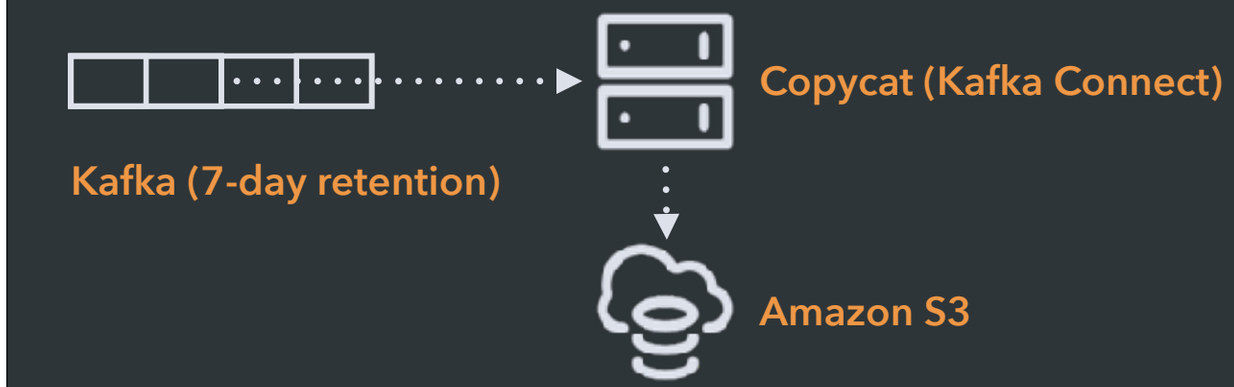
Our primary data archive is something we call TwilioFS. It’s essentially a data lake, which is a term some data architects made up to describe what happens when you dump everything you collect into one place so you can extract chunks of it for further processing or analysis. So that’s what we do.

ARCHIVAL: TWILIOFS DATA LAKE



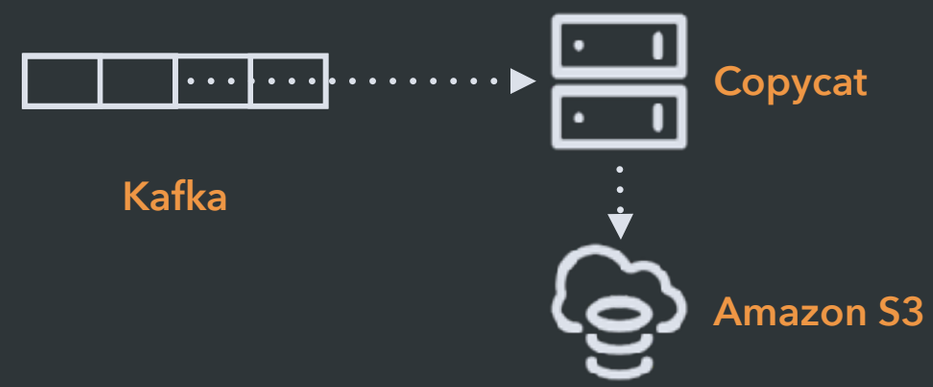
TwilioFS works like this: incoming Kafka records are consumed by a system we call Copycat. Kafka brokers don’t have infinite storage space and therefore only retain a fixed window of time for each topic before data expires, so Copycat has one job: to copy, byte for byte, entire Kafka topics into Amazon S3.

ARCHIVAL: TWILIOFS DATA LAKE

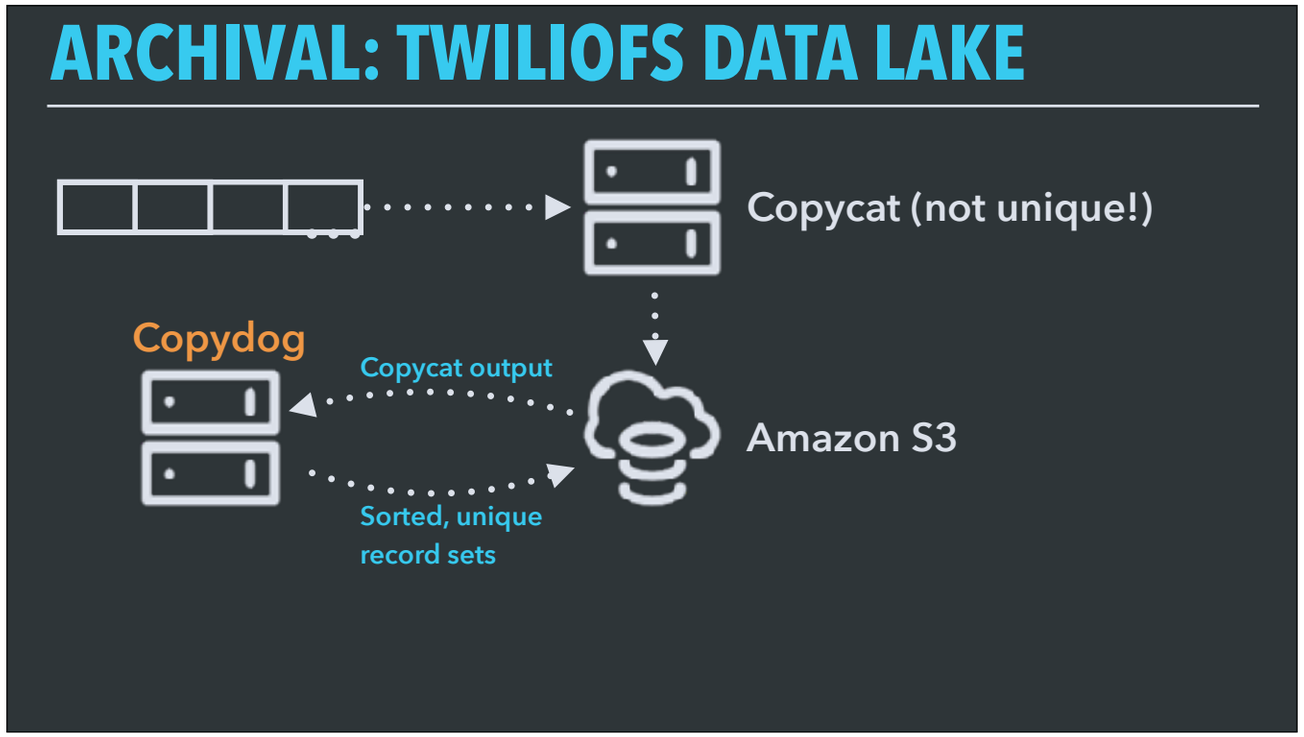


Copycat is a Kafka Connect application — Connect is a framework distributed with Kafka designed for exactly this operation of consuming data from Kafka and writing it somewhere else, or reading data from another source system and producing it onto Kafka. We use it for pretty much every sink system we write Kafka records to. Copycat's output is organized by the topic and partition, then the date and hour, and then the highest offset in the Kafka batch flushed into that file.

ARCHIVAL: TWILIOFS DATA LAKE

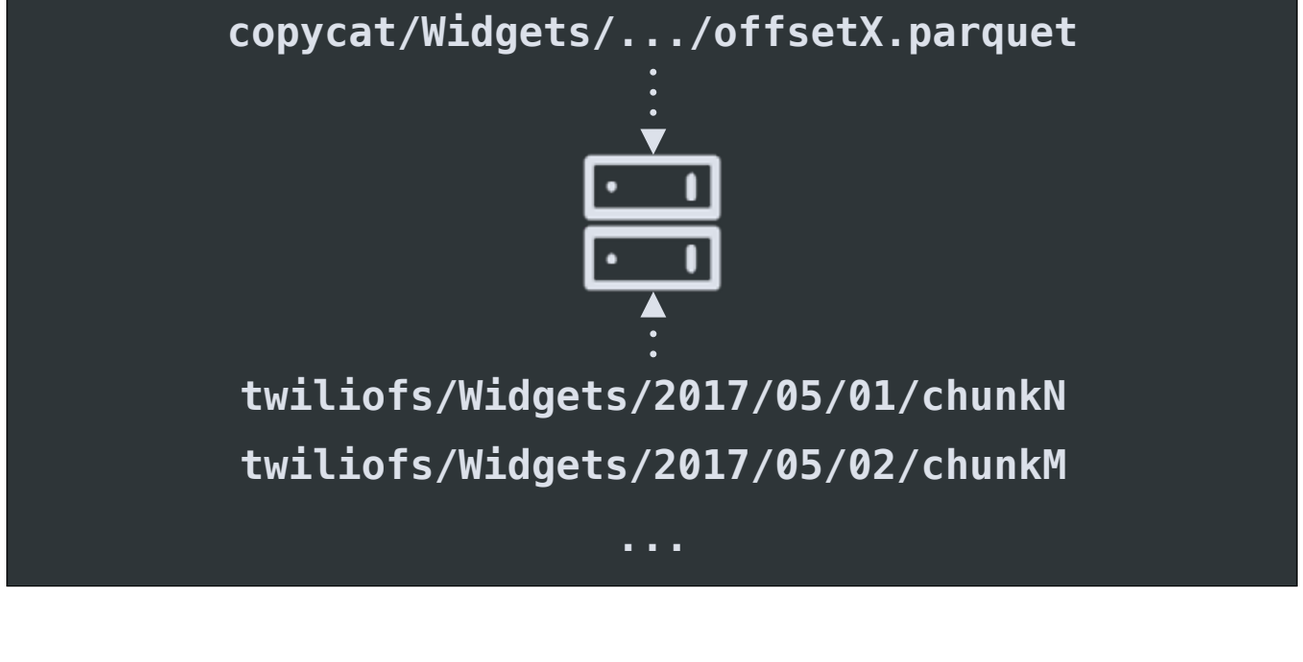


Before we've even done anything else with this data, Copycat has already given us a really neat capability: since the output is a perfect copy of the original Kafka topic, we can replay any subset of it back through Kafka and it's like we've gone back in time. This is really useful if someone finds a logic bug and needs to reprocess old data once they've fixed it.

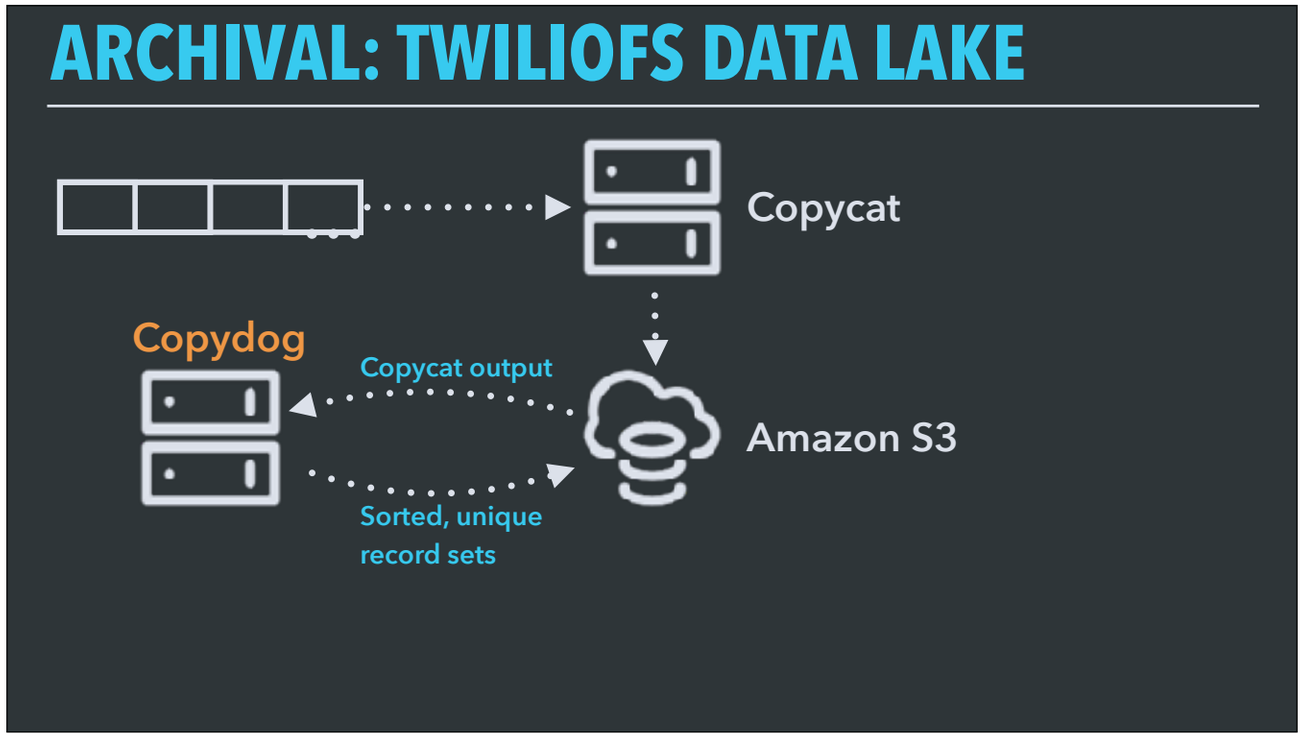


But Copycat is just the first step. Because the output from Copycat isn’t guaranteed to be unique or in order (since we have producers running in parallel, and since, well, distributed systems), we run a system called Copydog to clean it up.
Copydog is a Spark job and it organizes the raw topic data that Copycat sends to S3. This is the first place our topic metadata about de-duplication and ordering comes into play: as configuration for Copydog. Copydog uses its knowledge of how to sort and merge records to emit a verified, sorted, and de-duplicated archive of each topic’s data, also to S3.

DATA DEDUPLICATION



Copydog actually has two inputs: the latest input file from Copycat since its last run, and the existing set of data in the clean output set. It first takes the set of new records and sorts it using the sort key. Then it identifies the set of date prefixes for which it found new records and reloads all of the *existing* records for those dates, finds any duplicate records using the configured unique key, and merges them appropriately. Finally, it writes out all of the changed date prefixes to S3 again. Since S3 doesn’t have an atomic move operation we actually write versioned filenames and maintain a metadata file that points to the current version — this keeps other tasks seeing a consistent view of the data.



Copydog's output is our true data archive. Every record produced to a topic is in this S3 bucket, exactly once, in the latest state we've seen for it, and sorted according to the specified ordering for the dataset.

The pivotal element of both these systems is that they are 100% configuration-driven: the logic is reusable across data types and the metadata API provides the configuration needed to specify input/output directories, schemas, and sorting/merge strategies. We can provision new Copycat tasks with an API call, and Copydog jobs just require pushing new configuration into our task scheduler.

CONSUMER SYSTEMS

Archival

Warehousing and structured analytics

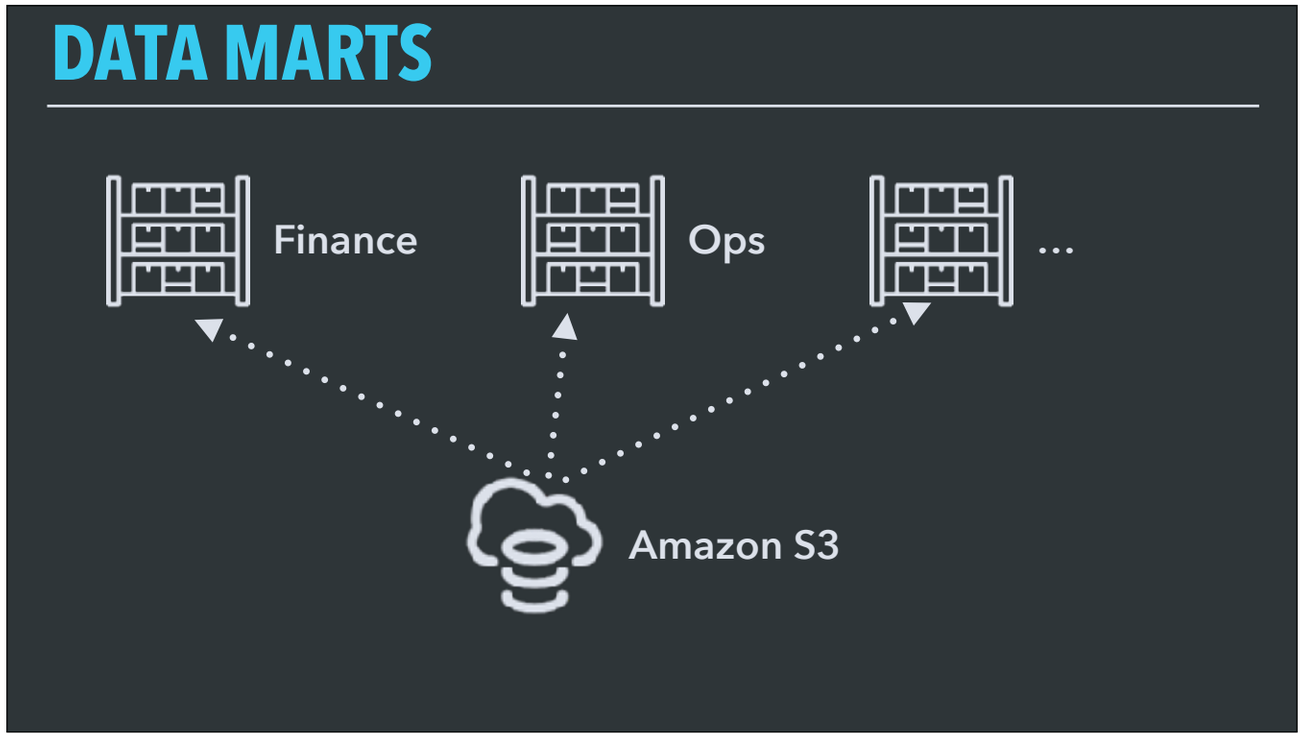
Batch processing

Ad-hoc analysis

Stream processing

Online query systems

This archive of data is now immediately usable for three things. First up, we can load it into data warehouses for interactive querying.



We use Amazon Redshift as our warehouse engine. Redshift is Amazon's managed columnar database. It speaks Postgres-compatible SQL so our business analysts love it and it's compatible with any visual BI dashboard system that speaks SQL, and it's optimized for bulk loading of data from S3. We've built custom tooling around the Redshift management APIs to let us easily provision multiple Redshift clusters and generate schedules based on our metadata to load data from S3 into one or more warehouses. Each cluster has a different set of data based on who is using it and what their needs are — this lets us keep them small and performing optimally for each use case (and keeps sensitive financial data away from people who don't need to see it).

CONSUMER SYSTEMS

Archival

Warehousing and structured analytics

Batch processing

Ad-hoc analysis

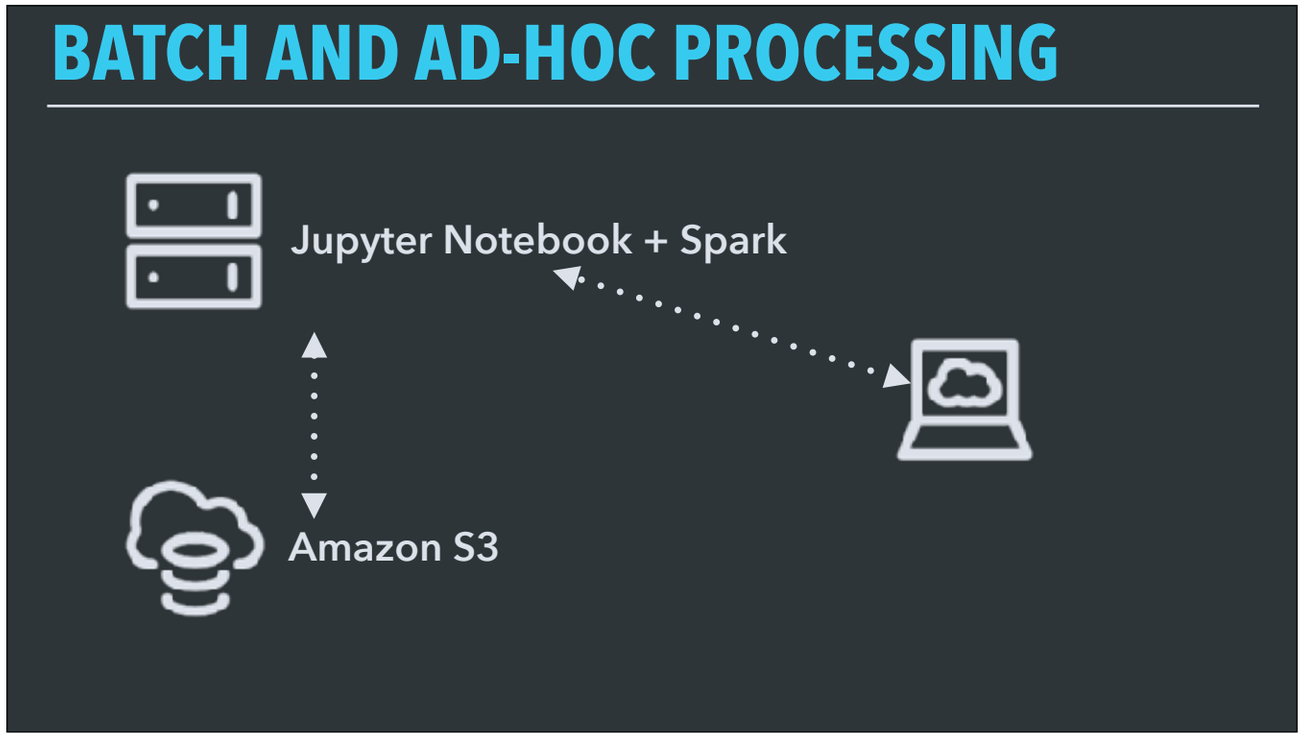
Stream processing

Online query systems

We can also use our primary Copydog archives to drive batch processing for derived data and for ad-hoc analysis.

BATCH AND AD-HOC PROCESSING

This is Spark again. Spark has a well-supported S3 data driver, and we've extended it to support the organizational and schema metadata available to us. Engineering teams that need to process large amounts of data write dedicated Spark jobs and use the data platform infrastructure to schedule them.



We also use Spark to do ad-hoc processing. Jupyter Notebook is an open-source project for using and sharing interactive data notebooks. This lets engineers and PMs query large amounts of data from S3 without having to load it into a Redshift warehouse first — this tends to come up during outage response and when devs are prototyping new scheduled batch jobs.

CONSUMER SYSTEMS

Archival

Warehousing and structured analytics

Batch processing

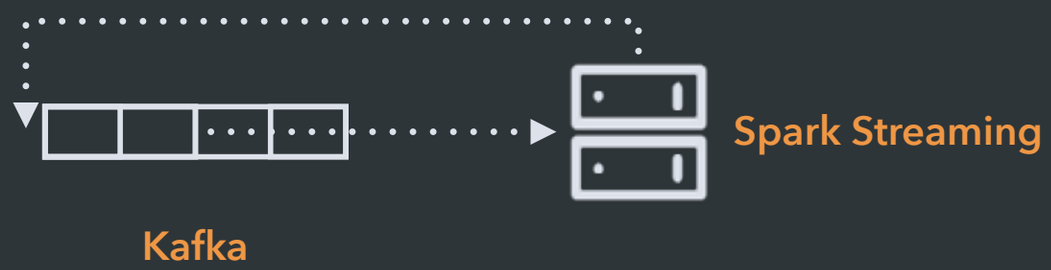
Ad-hoc analysis

Stream processing

Online query systems

I’m sounding like a broken record right now, but we also use Spark for stream processing.

STREAM PROCESSING



We can connect Spark tasks in streaming mode directly to a Kafka topic, process data in real time, and emit results back into a new Kafka topic for storage elsewhere.

CONSUMER SYSTEMS

Archival

Warehousing and structured analytics

Batch processing

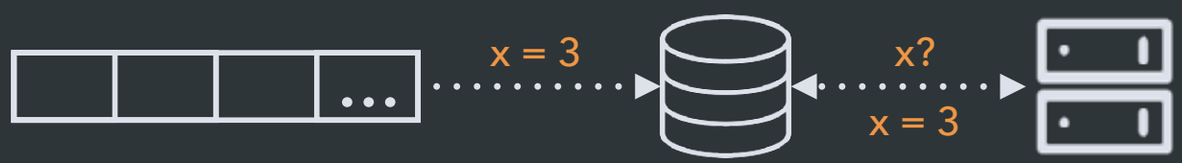
Ad-hoc analysis

Stream processing

Online query systems

And finally, we can consume directly from Kafka topics and write the data to an online data storage system.

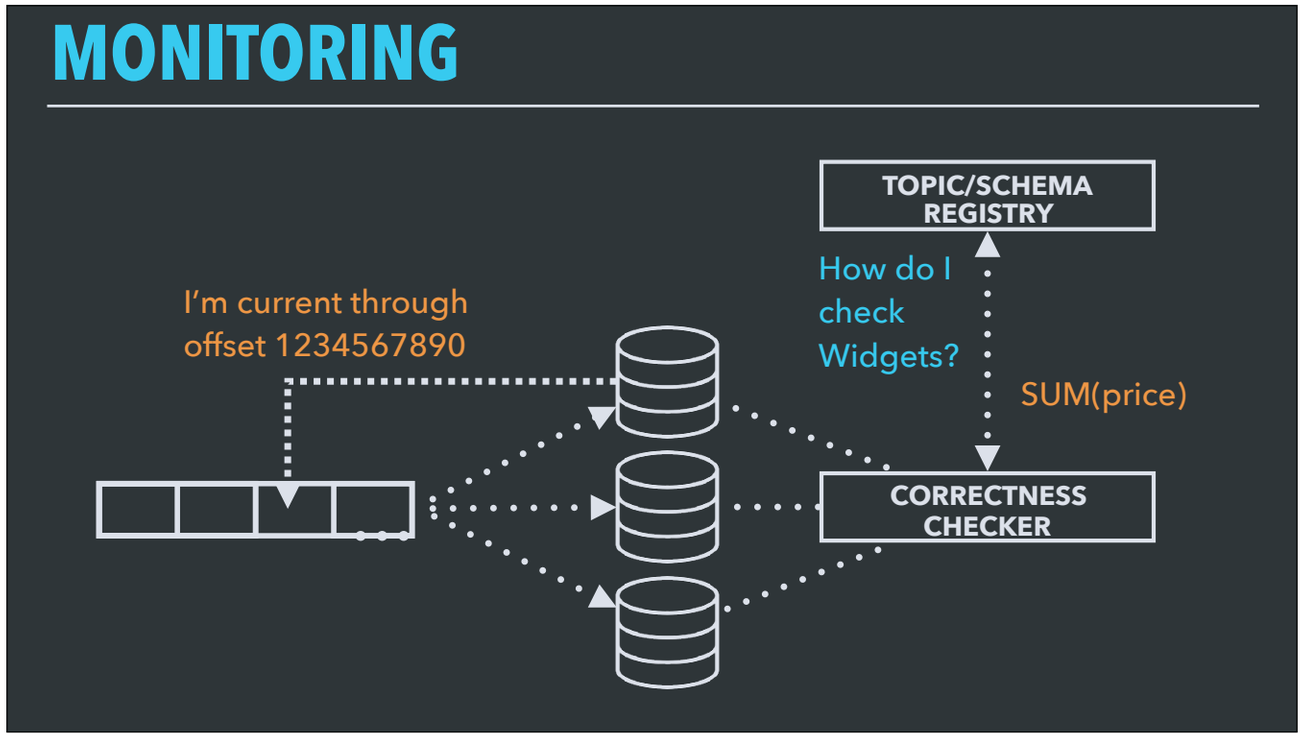
ONLINE STORAGE AND QUERY



And this is about as simple as it sounds. We run a Kafka Connect service that copies data from a Kafka topic into a database (MySQL, Cassandra, ElasticSearch, whatever) as it comes in, and then we can query it from other systems to satisfy user requests.

VERIFICATION AND CORRECTNESS

OK. We've gotten data into our pipeline, we've consumed it for a variety of storage and computational needs… how do we know we got all the right data to all the places it needed to be?



First off, Kafka's consumer offset tracking assures that we only acknowledge a record as delivered once we've safely stored or processed it. These offsets are available as Kafka streams themselves, meaning we can track and monitor our consumers' progress over time. If a particular system gets stuck or slows down, we can trigger an alert to make sure it's repaired.

For the correctness of the data itself, we use our topic metadata to do some lightweight smoke testing. Knowing how to identify unique records means that we can ask data stores to count up the unique records, and we can add additional checks (e.g., how much did we make in Widget sales yesterday, or even just a full-row checksum) to ensure the different storage systems agree with each other.

RECAP

That’s about all I’ve got, so let’s put it all together and review.

RECAP

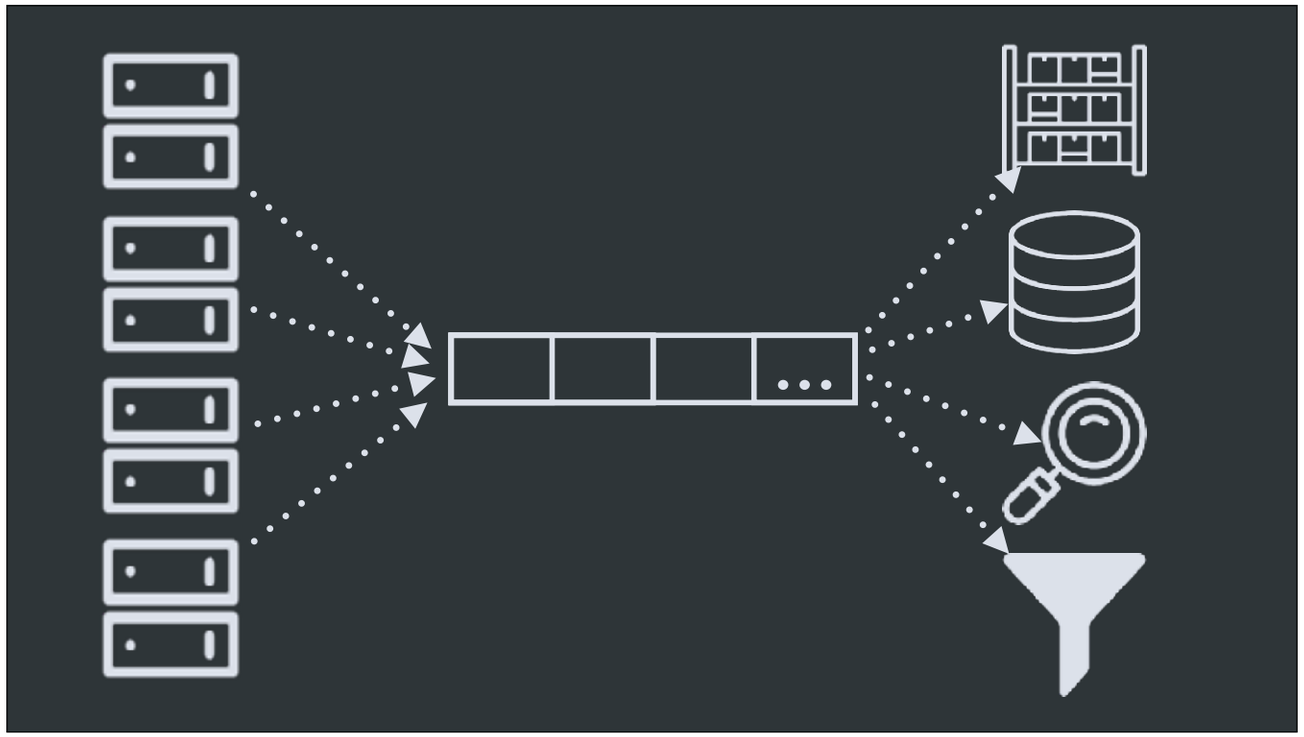
Event-oriented data pipeline

Common producer and consumer libraries

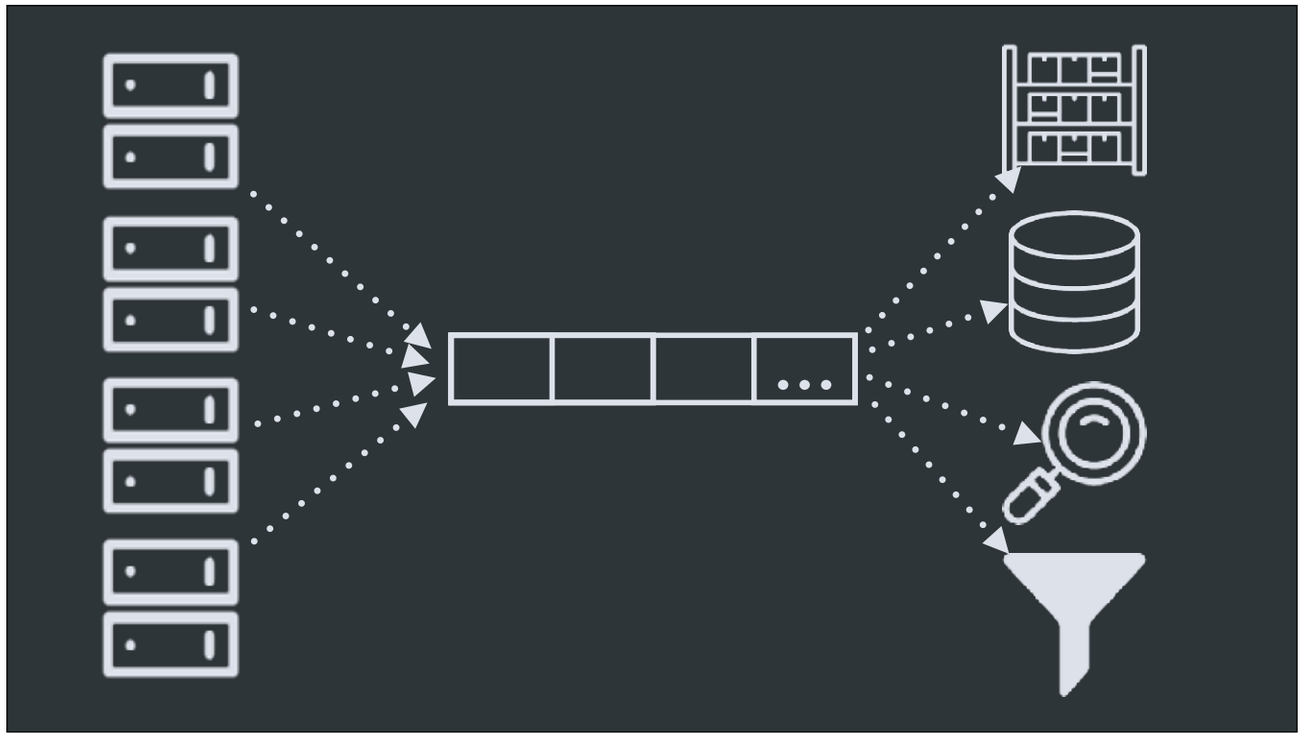
Strong schema validation and planned migrations

Verifiable delivery and correctness

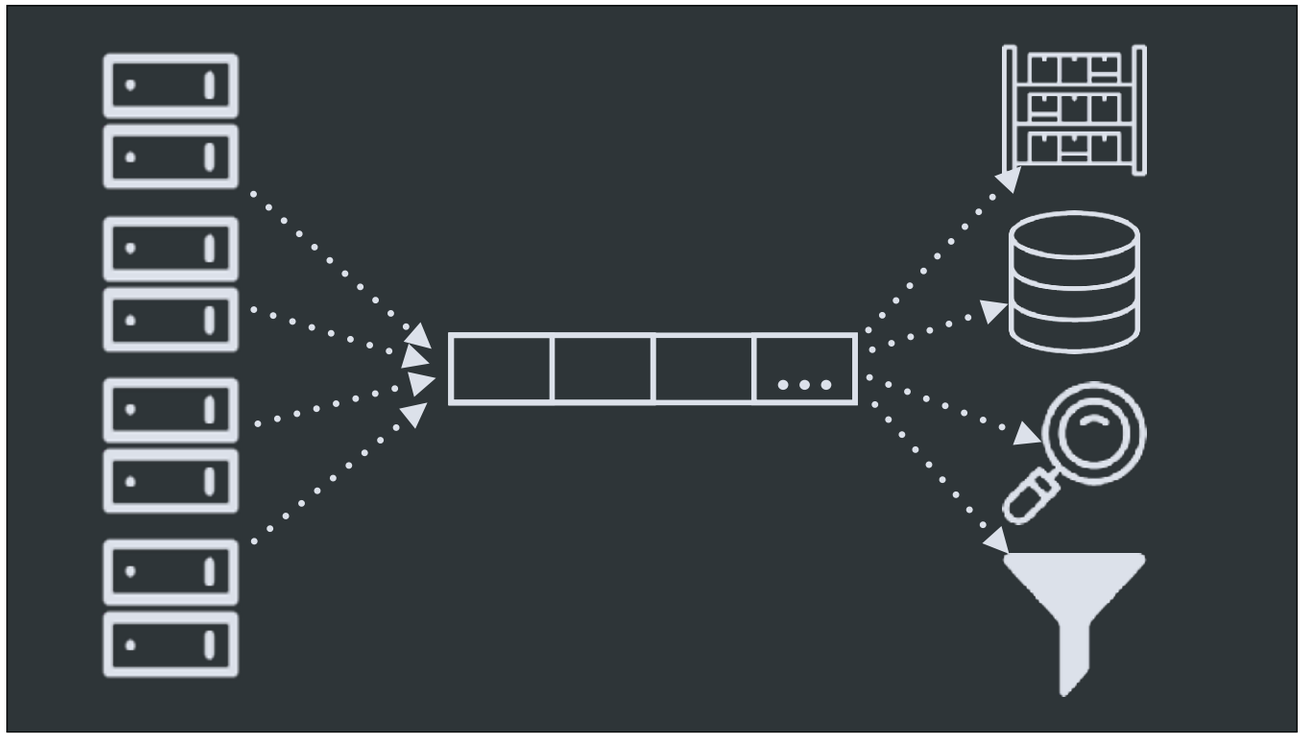
We've seen how an event-oriented data pipeline architecture, built around Kafka as a reliable and high-performance data bus, gives us a much stronger foundation for reasoning about and planning data flows between systems producing data, systems processing and consuming data, and systems storing that data.
We've talked about why strongly-typed data with a single source of truth for schemas is important and why your data processing systems will benefit.
And we've briefly touched on how the properties of these systems both inherently guarantee data is delivered everywhere it's needed, and how to build lightweight monitoring systems to verify that that is indeed the case.



So let’s revisit our motivating story from the beginning of this talk.
Instead of a growing web of connections made directly between data sources and sinks, we have a single path: data flows as streams of events through our Kafka bus, and any system that needs a particular type of data can connect and listen for it.



When Team A comes along and changes their schema, the systems connected to their data either seamlessly convert data to the version they expect, or gracefully process to the end of the topic with the old version and can be updated to pick up the new data. The end of the year rolls around and Team B's PM just requests a temporary Redshift cluster with the historical data loaded in so they can get numbers for their slide deck.



Team C tries out a shiny new data store, finds out it totally suits their needs, and just drops it in as a new flavor of Kafka consumer. Team D’s spam-detection code trains itself nightly as a Spark batch, and the same code loads up the new model and runs it on incoming messages in streaming mode. Team E hooks up ElasticSearch and their new full-text search feature works flawlessly. And so on.

@SKIMBREL // SAM@TWILIO.COM

THANKS!

And that’s all I’ve got. Thanks for listening and I hope the ideas I’ve shared today help you with your own data scaling challenges.

ATTRIBUTIONS

http://www.flaticon.com/authors/madebyoliver

http://www.flaticon.com/authors/freepik

http://www.flaticon.com/authors/vectors-market

Heat pipes: Bill Ebbesen on Wikimedia Commons (https://commons.wikimedia.org/wiki/File:Heatpipe_tunnel_copenhagen_2009.jpg)