Confluent OEM Program: Grow faster with enterprise-grade data streaming | Learn More →

NOV 5, 2021

READ TIME: 9 MIN

How Do You Change a Never-Ending Query?

Technology > Use Cases

Almog Gavra
Co-Founder

There's a philosophical puzzle of the Ship of Theseus where throughout a long voyage planks in a ship are individually replaced as they begin to rot. At the end, there is not a single original plank left. Is the ship that left the dock at the beginning of the journey the same ship that arrived at the harbor many months later?

In the world of streaming databases and ksqlDB, issuing a persistent query is the beginning of a journey where components become outdated and need to be replaced as the data being processed around them evolves. Continuing the analogy of a ship's voyage, once the query is at sea, it cannot be abandoned and started anew—it must process every event that it receives, without downtime, even as it evolves to accommodate changes in business requirements. Contrast this with the world of traditional databases, where an application powered by such a system would repeatedly issue short-lived queries on a snapshot-in-time of the data with no regard to queries that preceded it.

Drawing a throughline from our previous blog post on streaming SQL, a retailer that processes an incoming stream of book purchases should never be unavailable—it should keep running 24×7, because customers may want to order a book from

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts. Cookie Notice

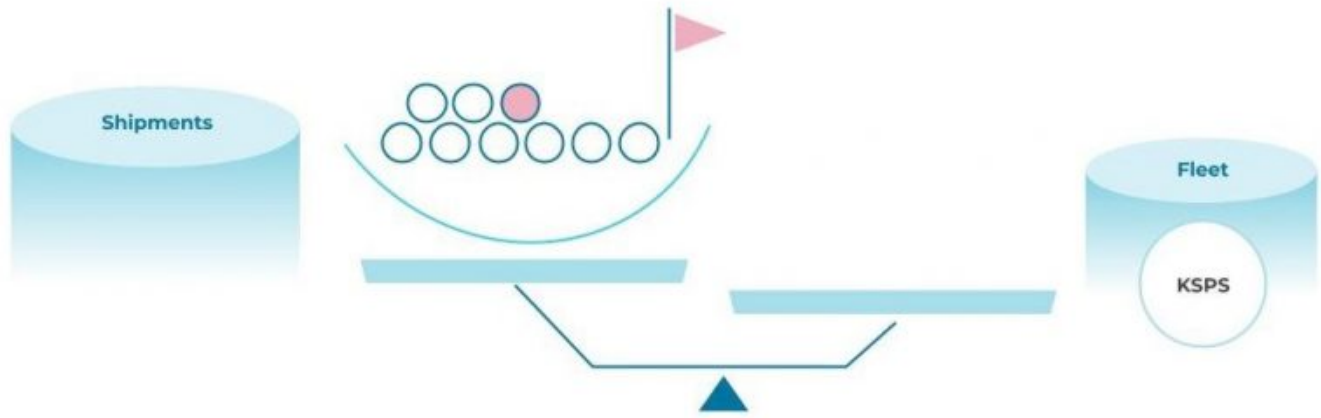
Accept All Cookies

Reject All

Cookies Settings

A cautionary tale

Ksql Parcel Service (KPS), back when it was just a fledgling company, had modeled its data using a stream tracking their shipments and a table describing their fleet. The shipment stream had two events for each shipment: one when the shipment was placed (with positive weight) and one when it was fulfilled (with negative weight).



They introduced the ksqlDB streaming database as a critical component of their architecture to ensure that no ship in the fleet was overloaded on any given day:

```
CREATE TABLE load AS SELECT ship_id, SUM(weight) AS load FROM packages GROUP BY ship_id;
CREATE TABLE alerts AS SELECT * FROM load JOIN fleet ON load.ship_id = fleet.ship_id WHERE l
```

With this in place, KPS felt their business would stay afloat...but here be dragons! As they started delivering an increasing number of mission-critical packages, their engineers calculated that if a storm hit and the vessel held an excess amount of rain water, it would be at risk of exceeding its capacity and having a soiree with Davy Jones' Locker. With this realization, they went back to the drawing board and figured that they should account for a potential 1 kg/L and the knowledge of how much water each ship might hold:

```
SELECT * FROM load JOIN fleet ON load.ship_id = fleet.ship_id WHERE load.weight > (fleet.cap
```

Novice ksqlDB users may fear that KPS is at an impasse: Do they drop their alerts table and recreate it from scratch to account for this change but in doing so, risk firing redundant alerts from historical events?

Charting the sea of upgrades

The situation described above is a foundational example of a query evolution. There are many ways for a query to evolve, but the taxonomy can be described by combining three characteristics: the *source query*, *upgrade type*, and (optionally) the *environment*.

Drawing from the table below, KPS's desired modification is a *stateful data selection* upgrade under a *live* environment. Another particularly interesting topology upgrade for a fully managed cloud service like Confluent Cloud are *transparent topology* upgrades (defined below)—versions of ksqlDB are upgraded automatically behind the scenes, and often the new code contains powerful processing upgrades. It is desirable to migrate old topologies to this improved runtime, but there may be challenges under certain query/upgrade characteristics, and it is critical that the application doesn't experience any significant disruption to their real-time processing.

Category	Characteristic	Description
Query	**Stateful**	Stateful queries maintain local storage
	Windowed	Windowed queries maintain a limited amount of state specified by a window in time
	Joined	Joined queries read from multiple sources
	Multistage	Multi-stage queries contain intermediate, non-user visible topics in Apache Kafka®
	Nondeterministic	Non-deterministic queries may produce different results when executing identical input
	Simple	Queries with none of the above characteristics
Upgrade	**Transparent**	Transparent upgrades change the way something is computed (e.g., improving a UDF performance)
	Data Selection	Data selecting query upgrades change which/how many events are emitted
	Schema Evolution	Schema evolving query upgrades change the output type of the data
	Source Modifying	These upgrades change the source data, whether by means of modifying a JOIN or swapping out a source
	Topology	These upgrades are invisible to the user, but change the topology, such as the number of sub-topologies or the ordering of operations (e.g., filter push down)
	Scaling	Scaling upgrades change the physical properties of the query in order to enable better performance characteristics
	Unsupported	Unsupported upgrades are ones that will semantically change the query in an unsupported way; there are no plans to implement these migrations
Environment	**Backfill**	Backfill requires the output data to be accurate not just from a point in time but from the earliest point of retained history
	Cascading	Cascading environments contain queries that are not terminal but rather feed into downstream stream processing tasks
	Exactly Once	Exactly-once processing environments do not allow for data duplication or missed events
	Ordered	Ordered environments require that a single offset delineates pre- and

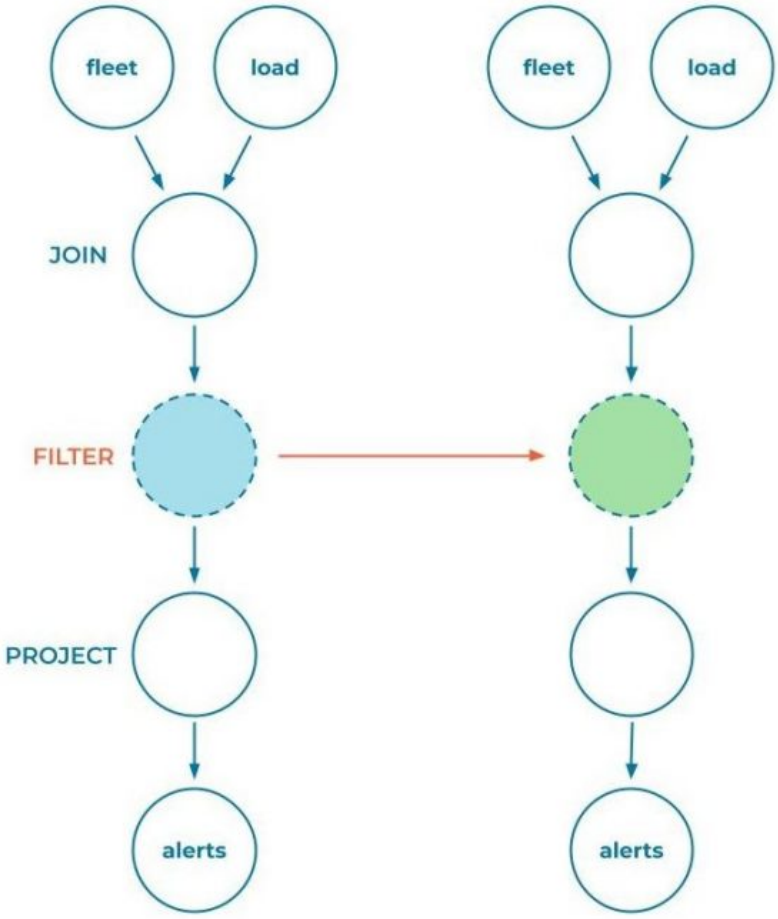
		post-migration (no events are interleaved)
	Live	Live environments describe queries that cannot afford downtime, either by means of acting as live storage (e.g., responding to pull queries) or feeding into high-availability systems (powering important functionality)

Source: Understanding ksqlDB query upgrades

Topology cartography

Some classes of upgrades, such as *simple data selection*, are easily supported in stream processing systems as the underlying topology may not even require a change. A more challenging problem, however, is to map out whether or not a desired upgrade falls into the class of supported upgrades.

ksqlDB and Kafka Streams are uniquely positioned to answer that question. Whenever a SQL query is processed by the ksqlDB engine, it transforms it into a JSON representation of the physical execution plan, which in turn compiles down deterministically to a Kafka Streams application. Given two such execution plans, we have two detailed blueprints that contain all the information that we need to classify whether or not they are compatible:



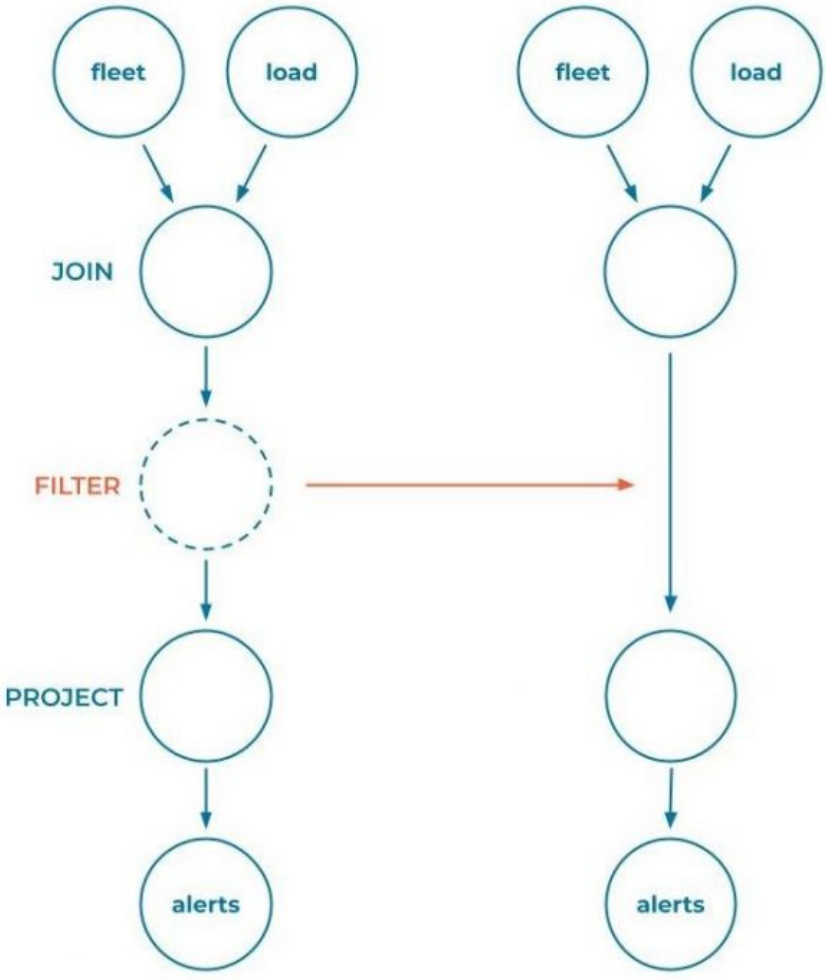
In this example, ksqlDB traverses both topology trees in lockstep, comparing the contents of each node to ensure that they are compatible. For a filter step (which is the

portion of the topology that changed in the KPS example), any modification is permitted so the entire upgrade is considered compatible.

But what if some nodes in the topology are removed or added? Imagine KPS engineers felt bold and decided to remove the capacity check altogether, allowing the ships on their fleet to accept an infinite load until they one day met their oceanic demise:

```
SELECT * FROM load JOIN fleet ON load.ship_id = fleet.ship_id;
```

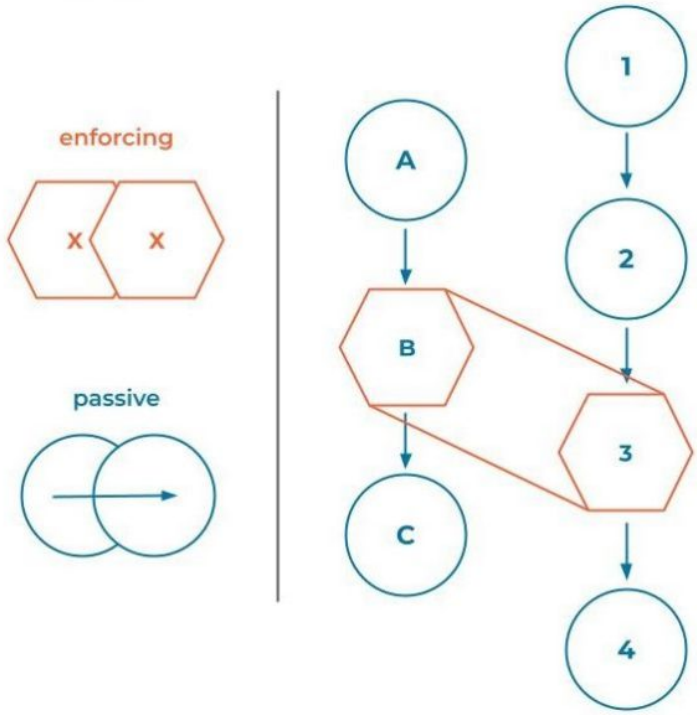
In this case, the topology evolution would look a little different:



It is easy to imagine that increasingly complex topologies would become difficult to compare and determine compatibility. So how does ksqlDB compare complex topologies? The first step is to categorize each execution step as either passive or enforcing.

- A *passive* step can be added/removed to topologies without causing any issues to upgrades. They delegate their upgrade validation to the next node in the topology. A filter is one such passive step.
- An *enforcing* step is a "stopping" point in the validation tree. When comparing two

trees, an enforcing step will force traversal of the other tree until it finds an enforcing step (skipping over passive steps). It then compares itself to the match in the other topology tree to ensure that they are compatible. Any aggregation, join, or otherwise stateful operation is an enforcing step.

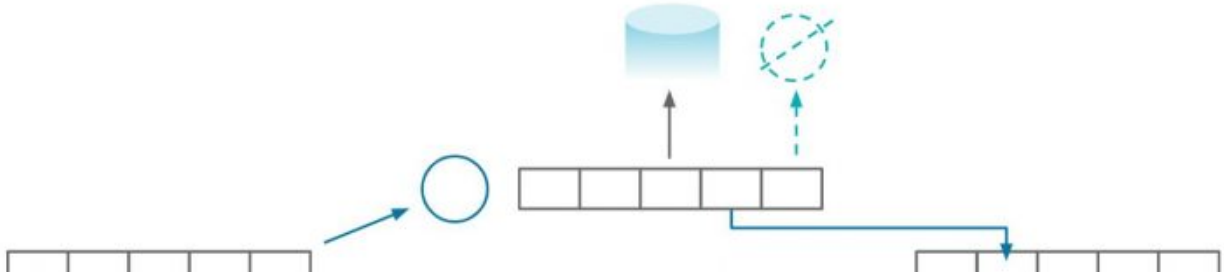


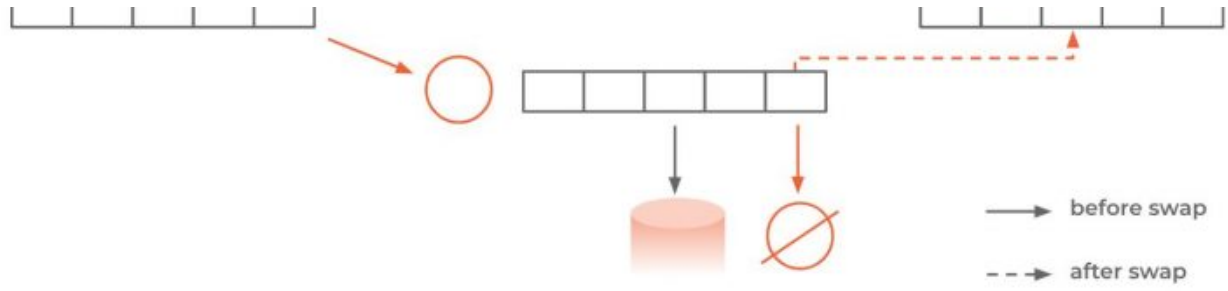
There's one little detail in the algorithm that observant readers might notice would cause an issue—if the topology tree contains multiple joins, how do we know where to start? ksqlDB solves this issue elegantly by leveraging knowledge that topologies only have a single output: It starts at the final node of the topology and traverses up the tree in the inverse direction of the data flow.

For readers who want to dig into what is considered passive vs. enforcing in ksqlDB today, as well as what is considered an incompatible change among enforcing nodes, the documentation and the code are both rather detailed in this regard.

Stateful waters

There will be a time when evolving a topology by manipulating passive steps falls short. ksqlDB does not yet implement a solution to this, but we have some ideas looking forward to what we can do. One such promising approach is to manage a "swap in place" deployment for such upgrades:





At a high level, the approach is to run a parallel processor in the background (pictured in orange) and have it process all the historical data that is available from the same input topic (pictured in dark blue). During this time, the migration will require extra physical resources to power this target topology. At some point, the stream processor will automatically, and perhaps more importantly atomically, stop the old topology from producing any data and cut the new processor over to produce to the existing output topic. This complexity comes in at this atomic cutover point.

Leveraging a lesser-known Apache Kafka internal component can help address this challenge: Control events are events that are not returned via the poll() API but instead incite some response from the consumer that is reading it. With slight modifications, we could use these events to indicate that a Kafka Streams application should start or stop producing to certain topics and facilitate the topology cutover:

The diagram above contains two topology diagrams: The blue topology is the “existing” topology while the orange one represents the target, upgraded topology. To make things more interesting, we’ve also modified the output and internal topics to have fewer partitions than the input topic. Control events are pictured as solid circles and other shapes represent normal events.

If we think of the control events as a vector clock, then the goal of the topology migration is to ensure that the output topic contains only events causally emitted from the blue topology before the control event vector and only events causally emitted by the orange topology after the control event vector. The presence of repartition topics makes this tricky, because it is possible that events are reordered across partitions (e.g., notice that the events represented by diamonds in the diagram above are not in the same order in the blue/orange deployments—this can happen as each input partition is handled asynchronously from the others but may output events to the same partition).

To set up a migration, the topologies are (re)deployed with additional steps:

1. The original (blue) topology contains a “stop gate” step as the first processing node in the graph. This node will stop reading any further events from a partition as soon as it encounters a control event in that respective partition.
2. The target (orange) topology has two additional steps, both a “stop gate” as the first processing node of the graph and a “start gate” at the tail end of the topology. The stop gate is similar to that which was added in the original topology—it will

temporarily stop emitting any non-control events from a source partition once it encounters a control event from that partition but continues to forward control events down the topology. Meanwhile, the start gate discards any non-control event that it receives and collects control events in a local buffer. Once it collects control events from each input partition (three in the diagram above) it (1) signals to the stop gate at the start of the topology to continue processing events and (2) stops discarding events and begins outputting to the original output topic.

To initiate the migration, an out-of-band producer produces one control event into each of the source topic partitions along with metadata such as how many control events it produced. From that point on, the stop/start gates function as logical barriers ensuring that we have an "atomic" cutover point, demarcated by the presence of the control event vector clock in the source topic.

There are some aspects of this design that remain to be flushed out, such as the out-of-band producer that produces the control vector, communication between the stop and start gate, and failure handling during migration. Nevertheless, the approach has shown promise in our early exploration.

The North Star

Query evolution in ksqlDB is an evolving (pun intended) area of research and development. Most of the discussion in this blog post only scratches the tip of the iceberg of what a fully mature streaming database would be capable of—we still need to consider the effect of upgrades on other parts of the system, such as ongoing pull and push queries.

If every node in a query topology is updated one by one, but the query never stops processing, is it still the same query? Philosophically, we'll never know, but we have the technology to do it. Try it out yourself by getting started with ksqlDB.

Get Started

Other posts in this series

- Overview: Readings in Streaming Database Systems
- Part 1: The Future of SQL: Databases Meet Stream Processing
- Part 2: 4 Key Design Principles and Guarantees of Streaming Databases

---

Almog Gavra is a Co-Founder at Responsive helping build ksqlDB. His introduction to stream processing was at LinkedIn, where he worked on various parts of the search infrastructure including the real-time index updates.

Get started with Confluent, for free

Get started >

Watch demo: Kafka streaming in 10 minutes

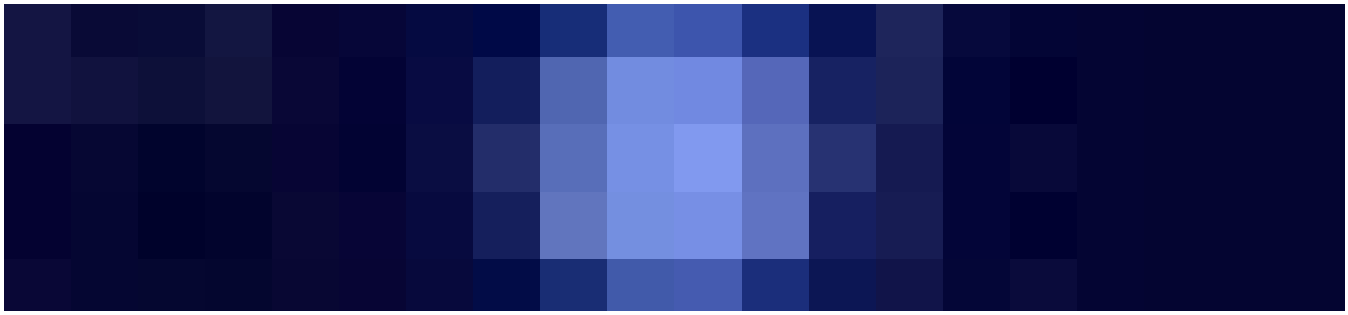
Watch now >

Did you like this blog post? Share it now

Technology < Use Cases

Subscribe to the Confluent blog

Subscribe

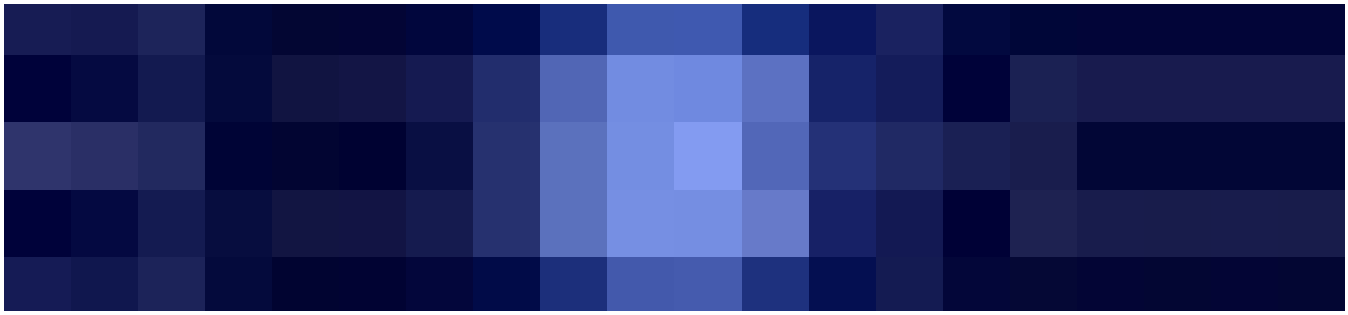


Introducing Versioned State Store in Kafka Streams

AUG 24, 2023

Versioned key-value state stores, introduced to Kafka Streams in 3.5, enhance stateful processing capabilities by allowing users to store multiple record versions per key, rather than only the single latest version per key as is the case for existing key-value...

VICTORIA XIA



Delivery Guarantees and the Ethics of Teleportation

APR 25, 2023

This blog post discusses the two generals problems, how it impacts message delivery guarantees, and how those guarantees would affect a futuristic technology such as teleportation.

WADE WALDRON

Terms & Conditions | Privacy Policy | Do Not Sell My Information | Modern Slavery Policy | Cookie Settings

Copyright © Confluent, Inc. 2014-2024. Apache®, Apache Kafka®, Kafka®, Apache Flink®, Flink®, and associated open source project names are trademarks of the Apache Software Foundation