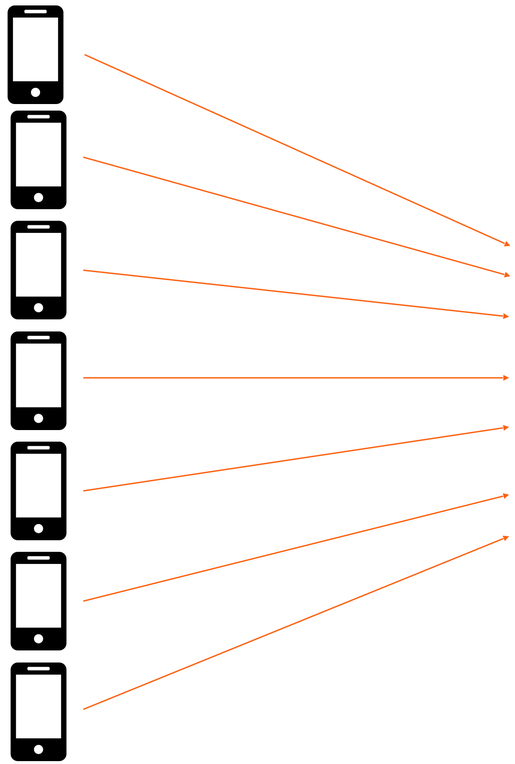
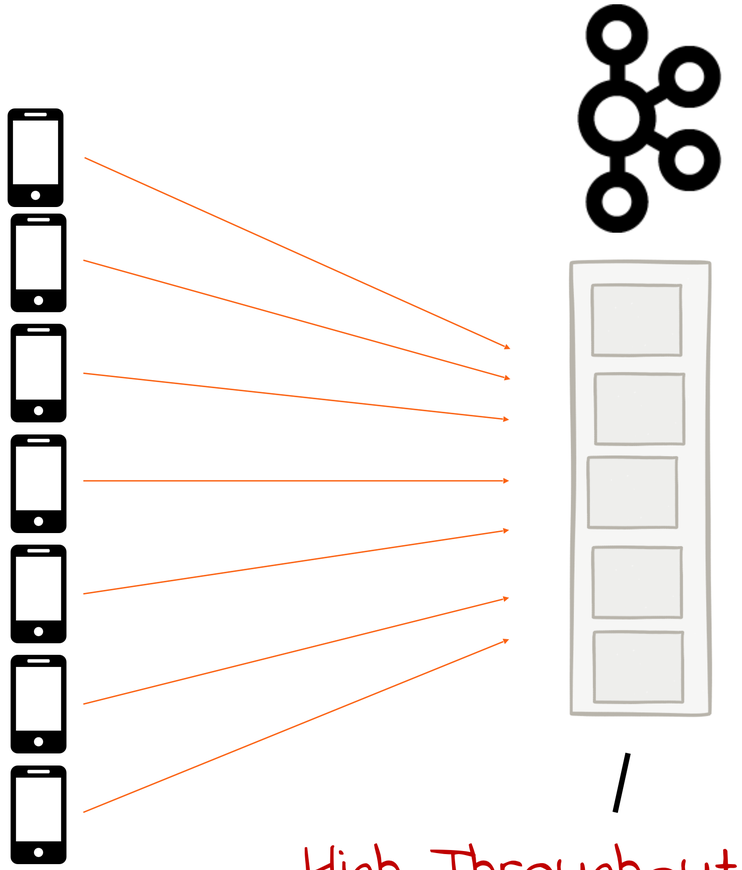

Crossing the Streams:
Rethinking Stream Processing
with
Kafka Streams and KSQL
@gamussa
#NYCkafka
@confluentinc

confluent

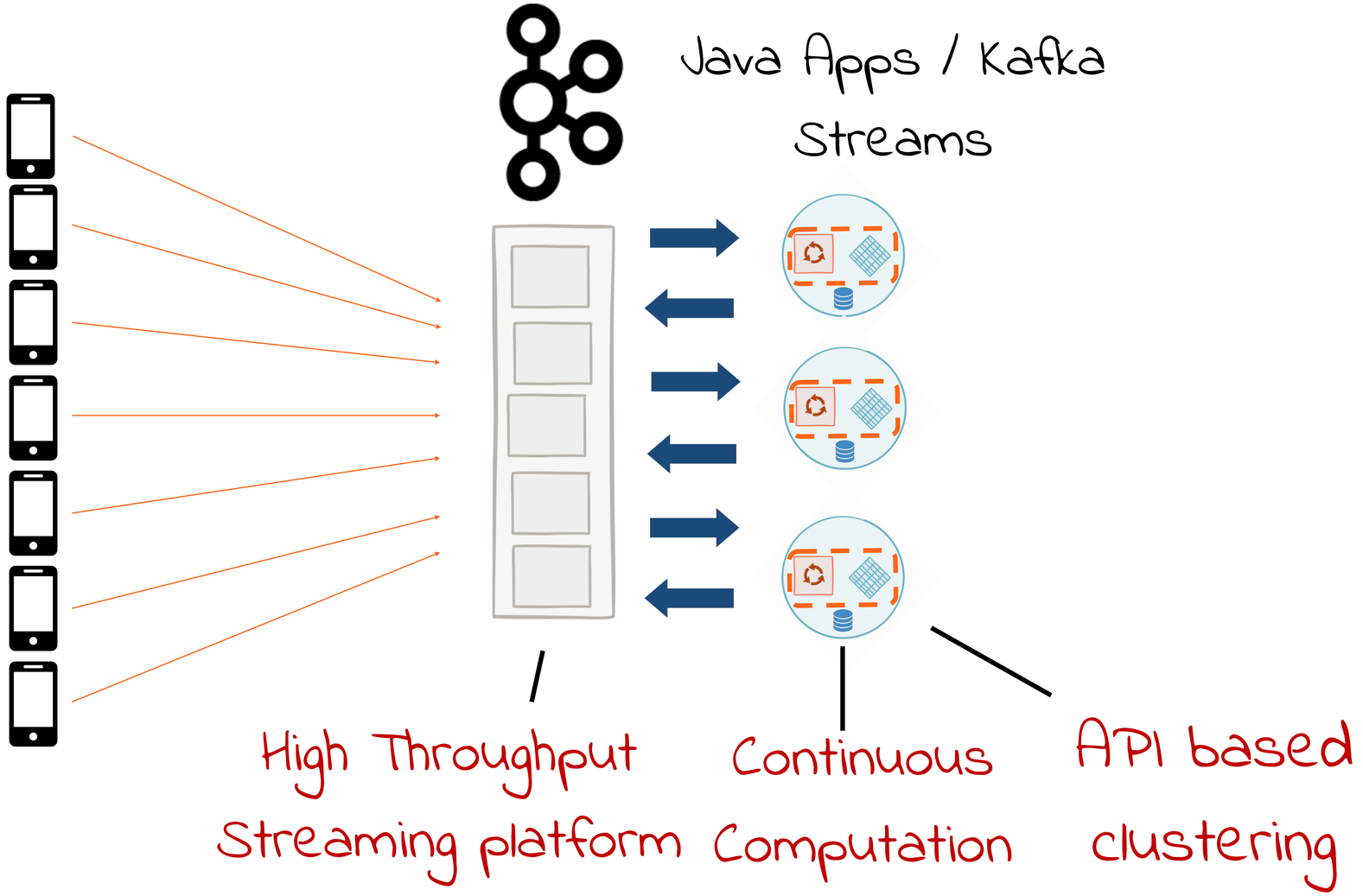
https://twitter.com/gAmUssA/status/1048258981595111424

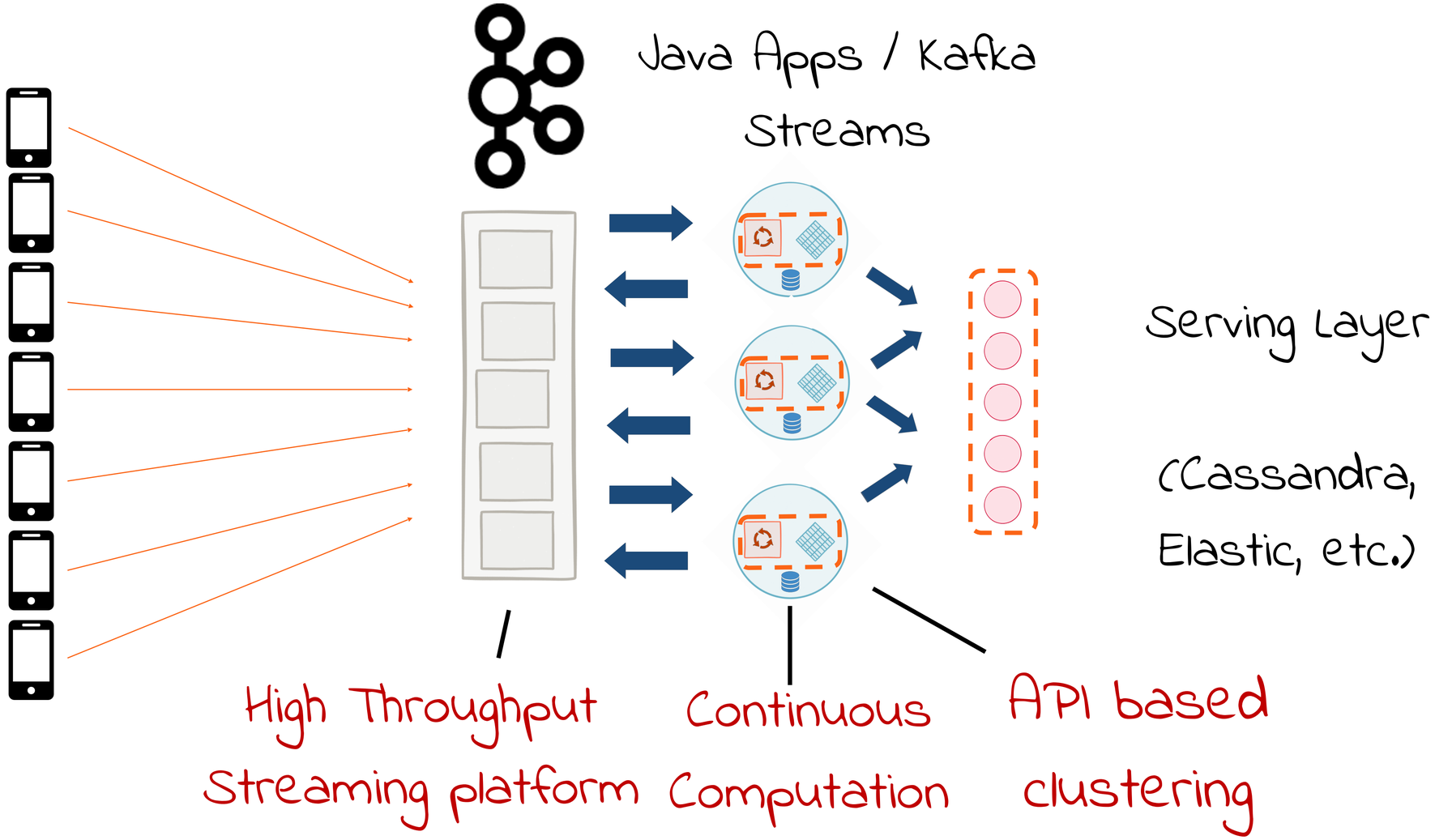
Streaming
is the toolset
for dealing
with events
as they move!





High Throughput
Streaming platform


Java Apps / Kafka Streams
High Throughput Streaming platform
Continuous Computation
API based clustering


Java Apps / Kafka Streams
Serving Layer
(Cassandra, Elastic, etc.)
High Throughput Streaming platform
Continuous Computation
API based clustering

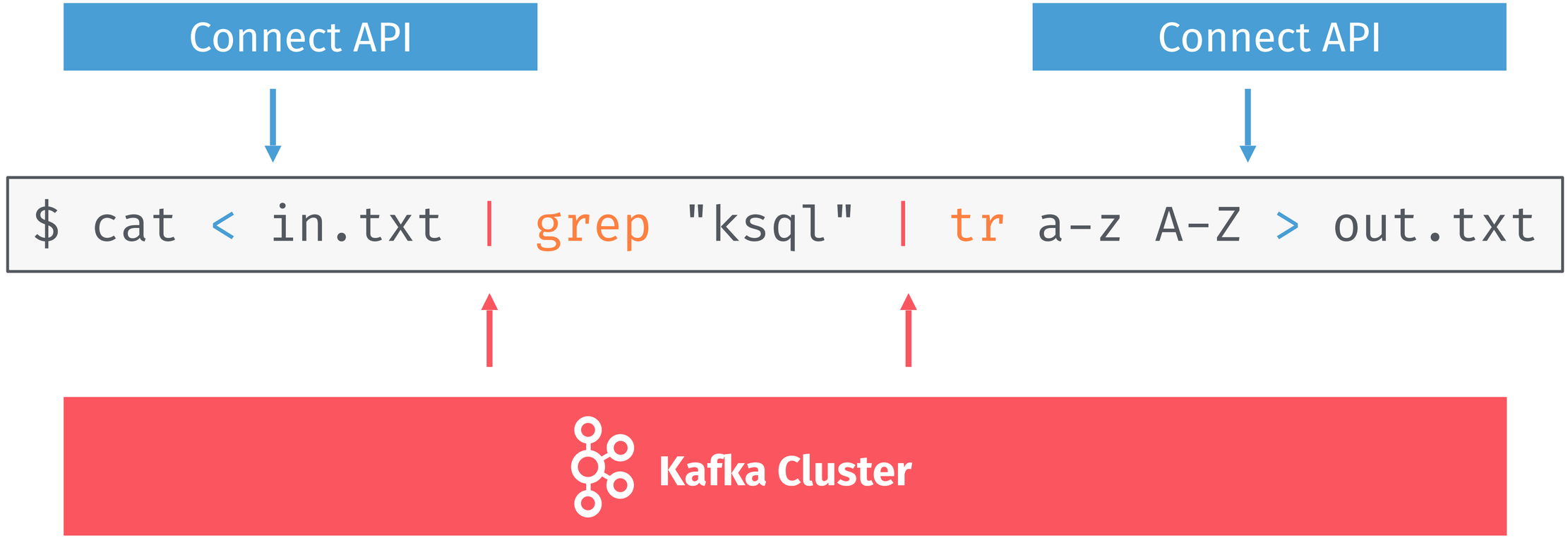
Stream Processing by Analogy

```
$ cat < in.txt | grep "ksql" | tr a-z A-Z > out.txt
```

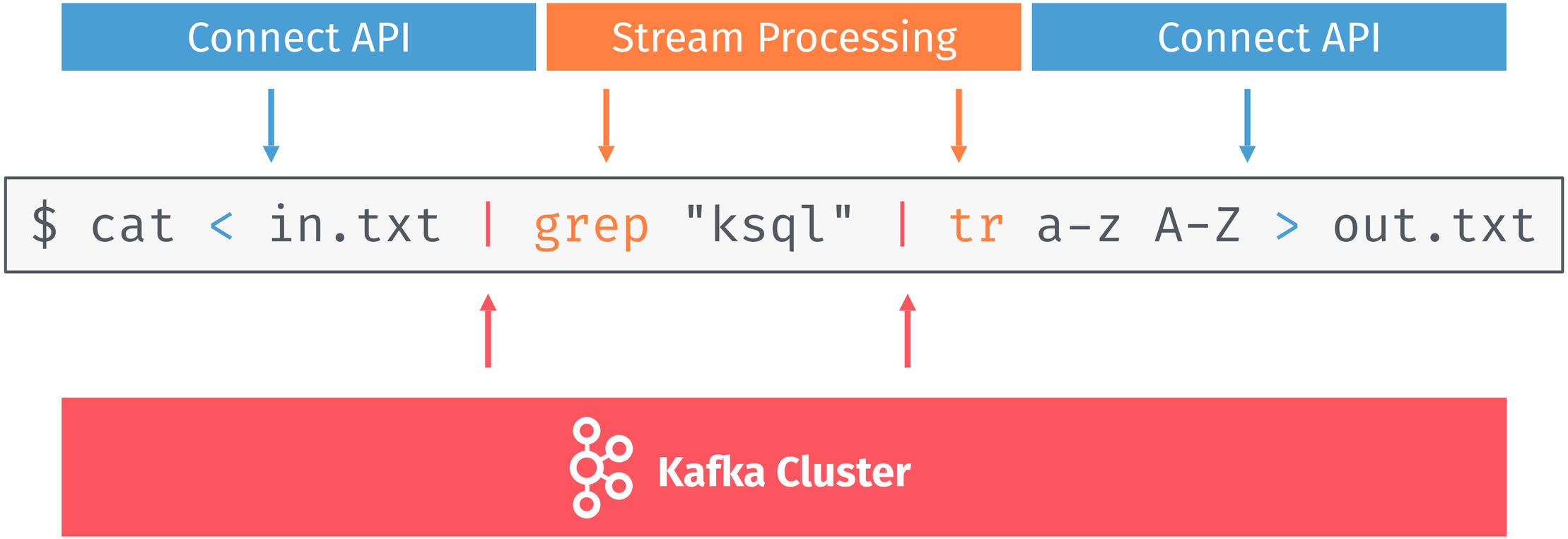
Stream Processing by Analogy

```
$ cat < in.txt | grep "ksql" | tr a-z A-Z > out.txt
```

Stream Processing by Analogy

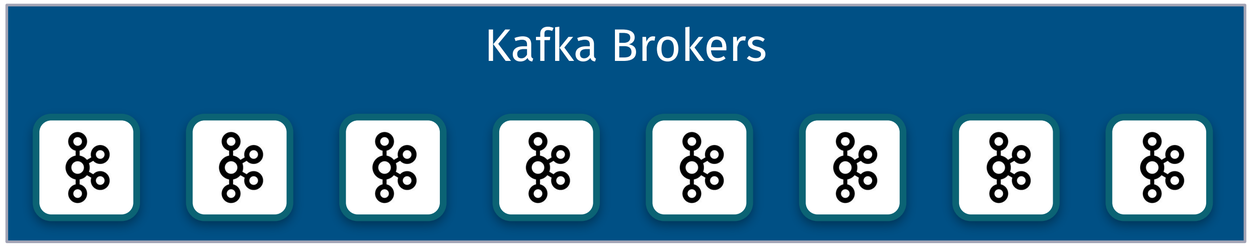


Stream Processing by Analogy

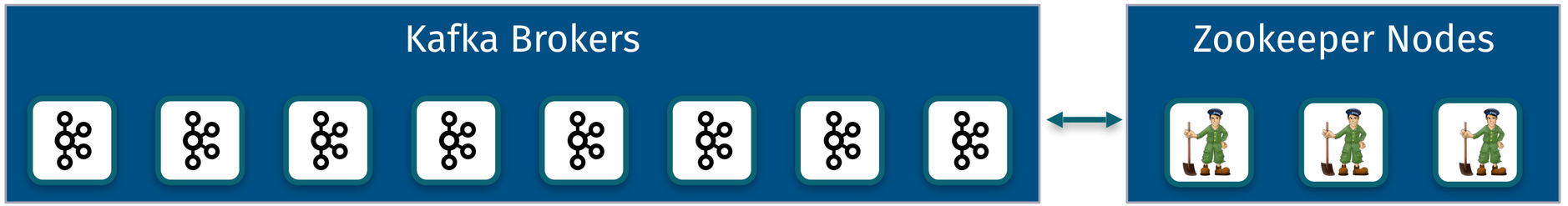


Streaming Platform Architecture

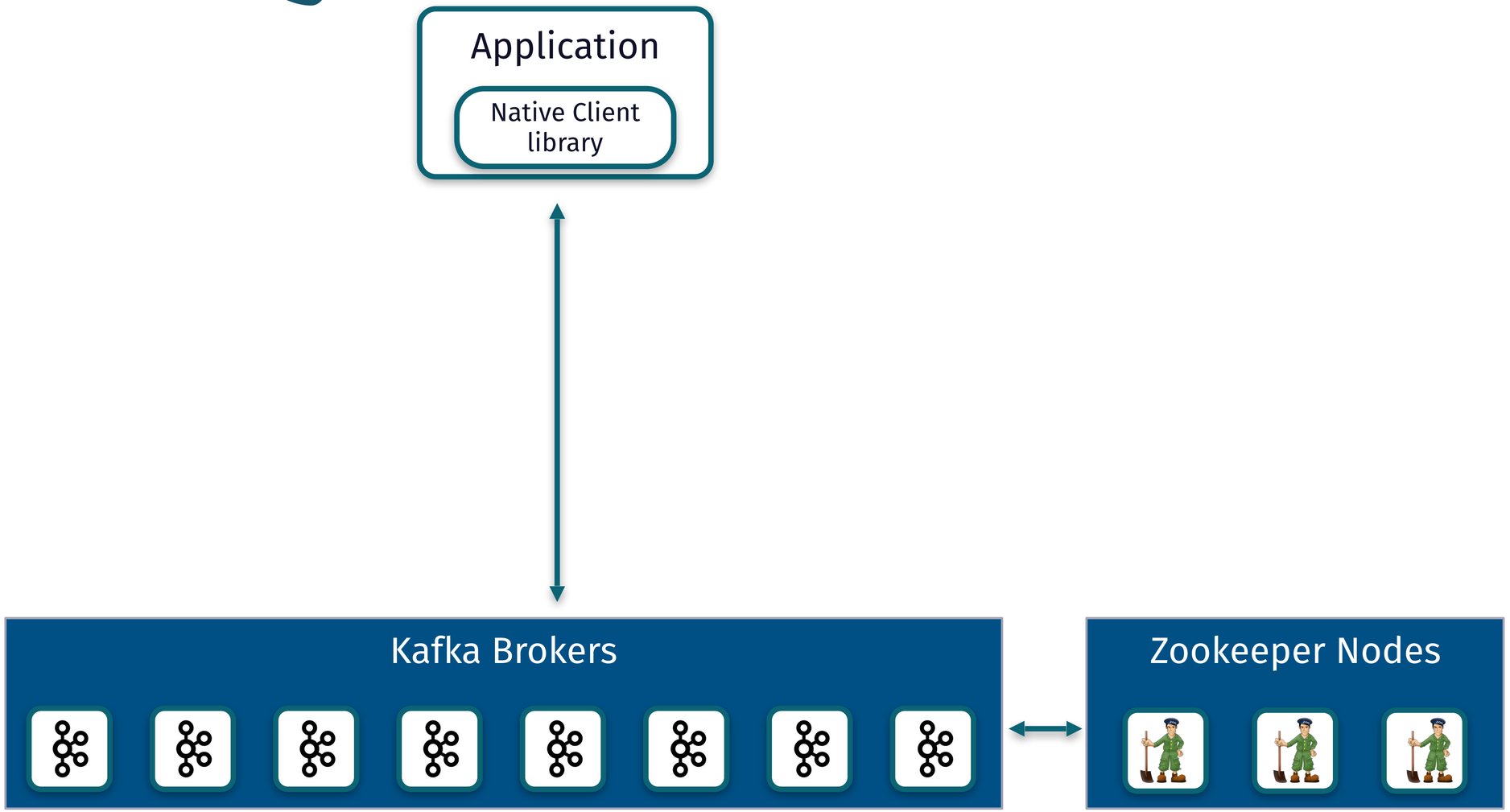
Streaming Platform Architecture



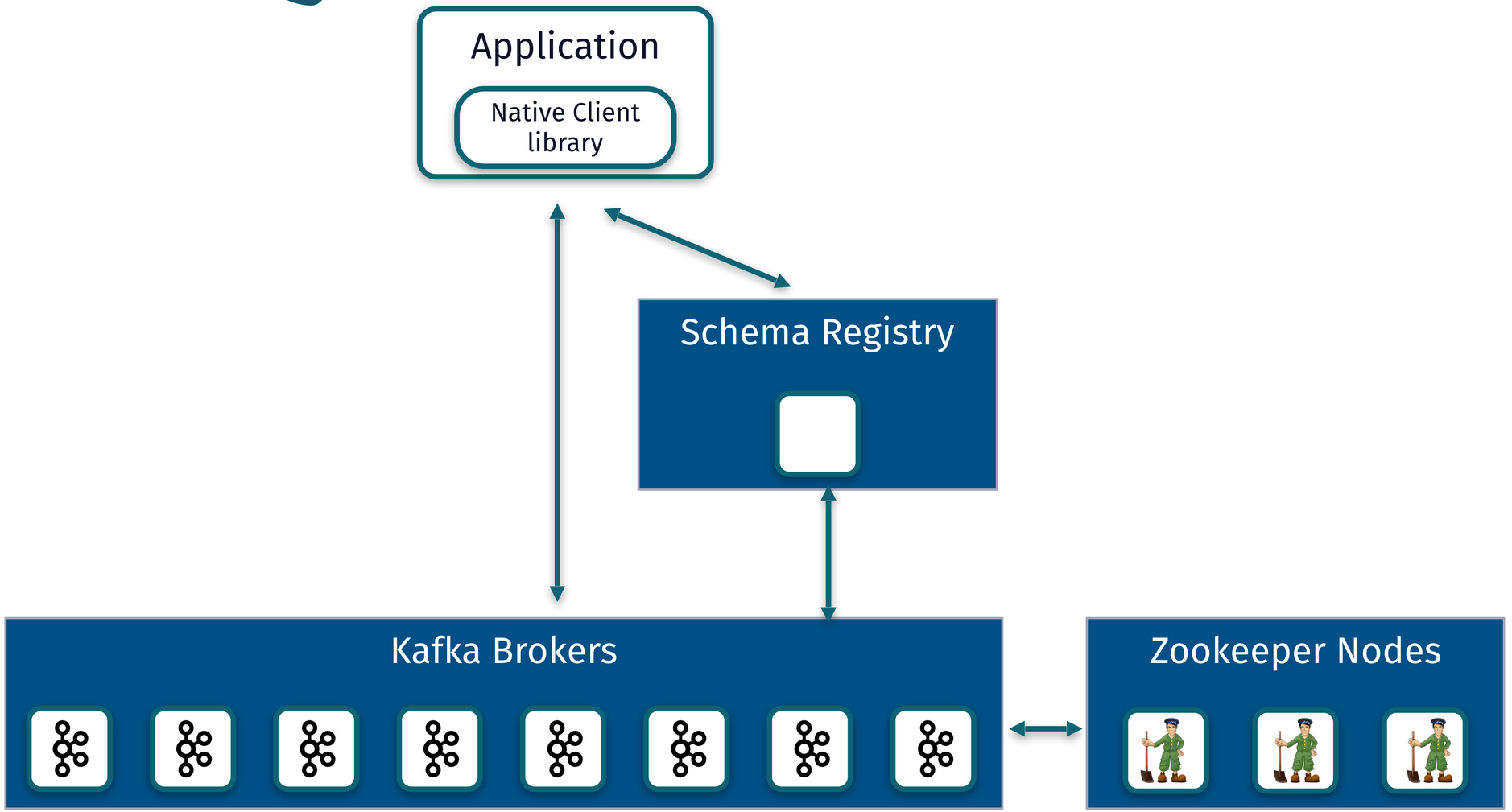
Streaming Platform Architecture



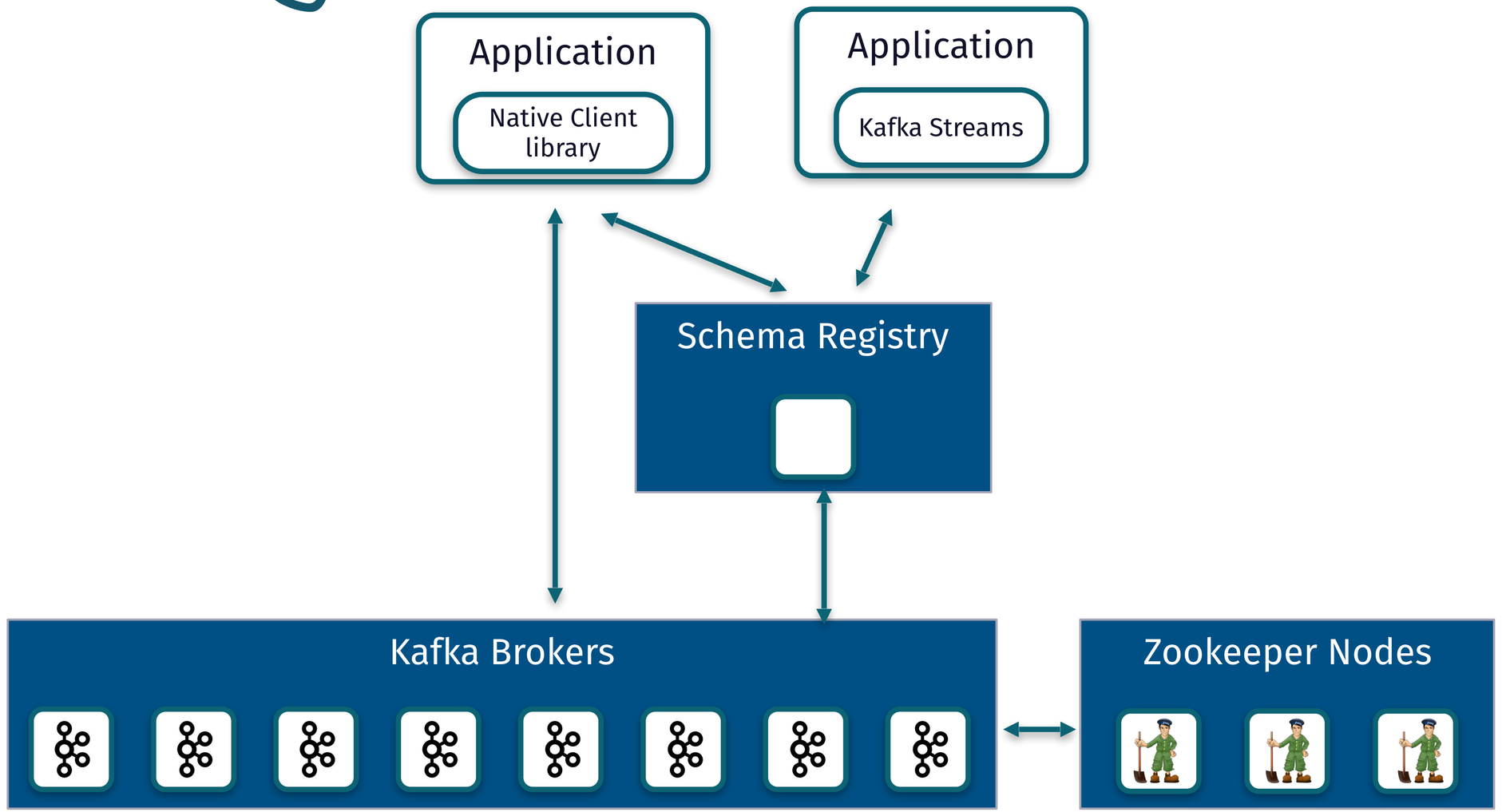
Streaming Platform Architecture



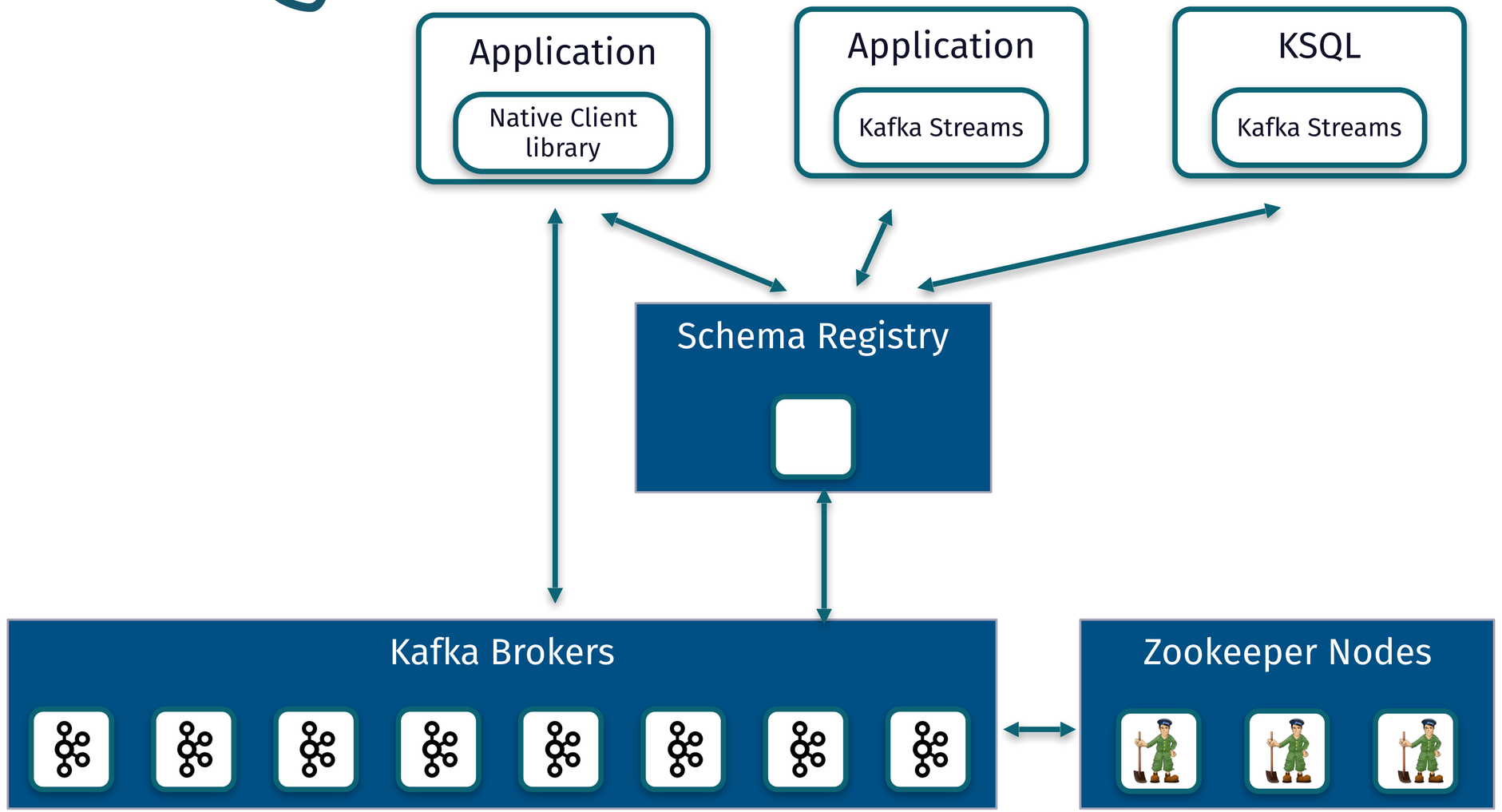
Streaming Platform Architecture



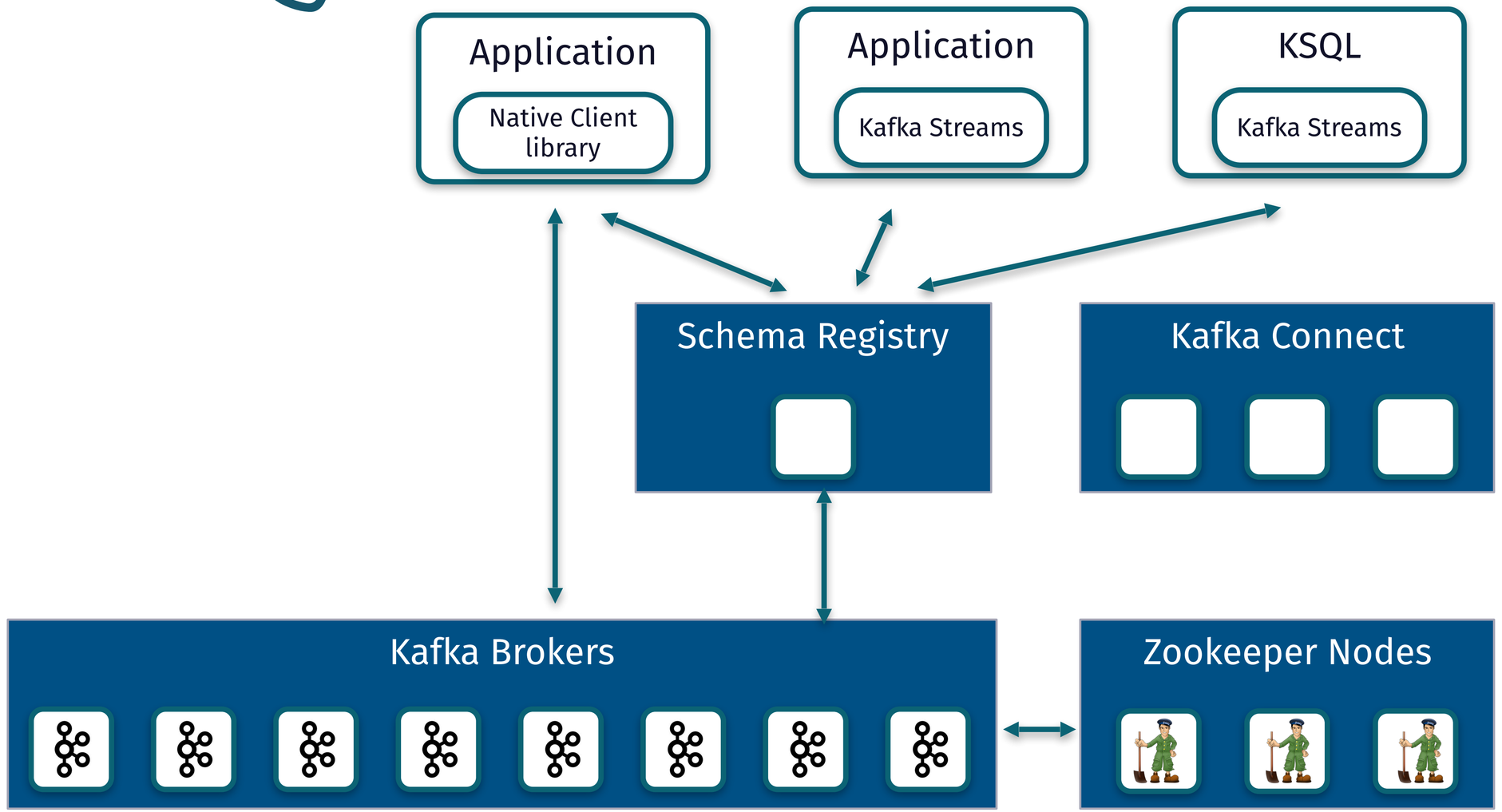
Streaming Platform Architecture



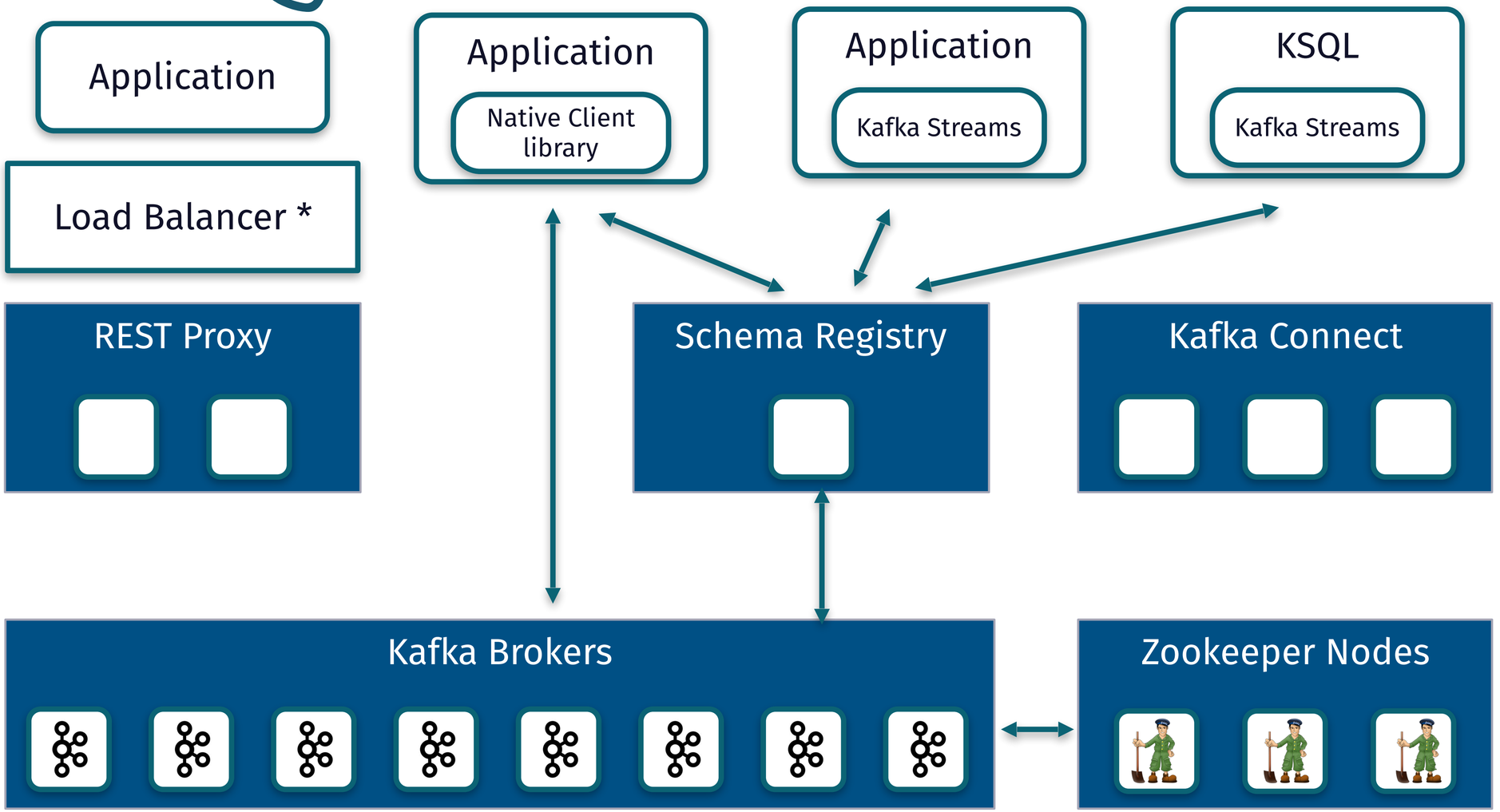
Streaming Platform Architecture



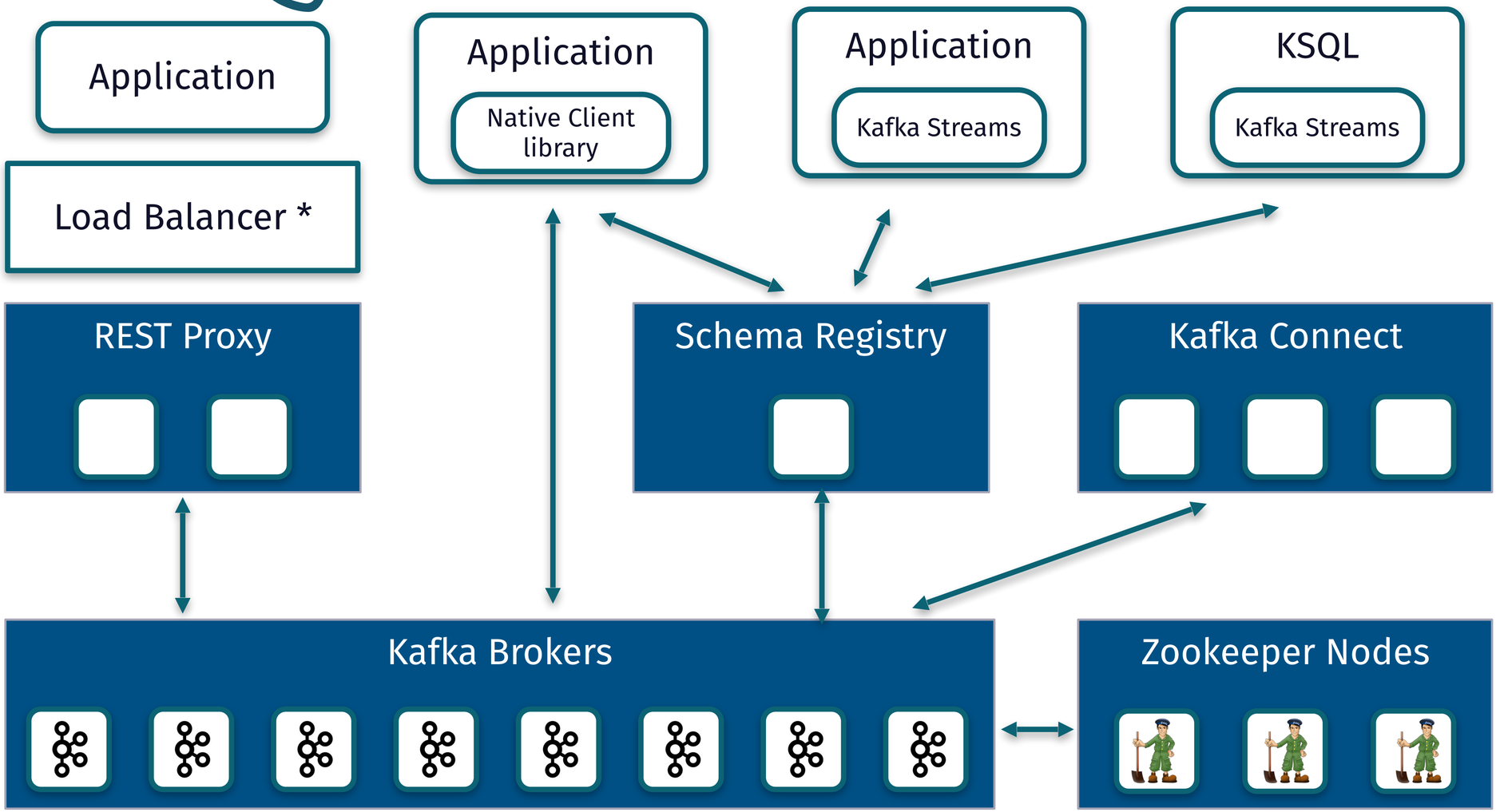
Streaming Platform Architecture



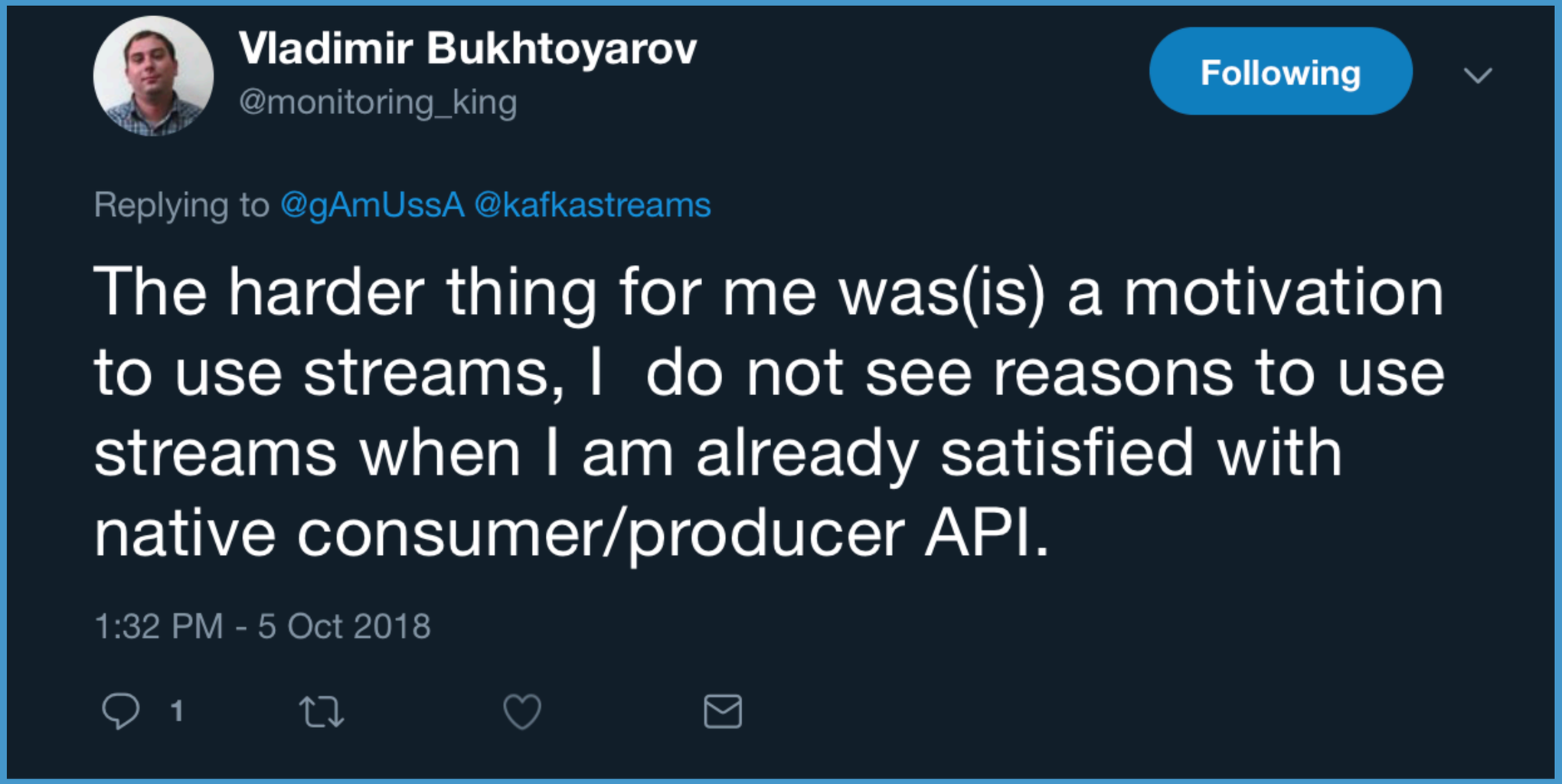
Streaming Platform Architecture



Streaming Platform Architecture



confluent



https://twitter.com/monitoring_king/status/1048264580743479296

```java
// in-memory store, not persistent
Map<String, Integer> groupByCounts = new HashMap<>();

try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProperties());
     KafkaProducer<String, Integer> producer = new KafkaProducer<>(producerProperties())) {

  consumer.subscribe(Arrays.asList("A", "B"));

  while (true) { // consumer poll loop
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(5));
    for (ConsumerRecord<String, String> record : records) {

      String key = record.key();
      Integer count = groupByCounts.get(key);

      if (count == null) {
        count = 0;
      }
      count += 1; // actually doing something useful

      groupByCounts.put(key, count);
    }

    if (counter++ % sendInterval == 0) {
      for (Map.Entry<String, Integer> groupedEntry : groupByCounts.entrySet()) {

        ProducerRecord<String, Integer> producerRecord =
            new ProducerRecord<>("group-by-counts", groupedEntry.getKey(), groupedEntry.getValue());
        producer.send(producerRecord);
      }
      consumer.commitSync();
    }
  }
}
```

```
// in-memory store, not persistent
Map<String, Integer> groupByCounts = new HashMap<>();

try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProperties());
     KafkaProducer<String, Integer> producer = new KafkaProducer<>(producerProperties())) {

  consumer.subscribe(Arrays.asList("A", "B"));

  while (true) { // consumer poll loop
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(5));
    for (ConsumerRecord<String, String> record : records) {

      String key = record.key();
      Integer count = groupByCounts.get(key);

      if (count == null) {
        count = 0;
      }
      count += 1;

      groupByCounts.put(key, count);
    }
  }
```

```
// in-memory store, not persistent
Map<String, Integer> groupByCounts = new HashMap<>();

try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProperties());
     KafkaProducer<String, Integer> producer = new KafkaProducer<>(producerProperties())) {

  consumer.subscribe(Arrays.asList("A", "B"));

  while (true) { // consumer poll loop
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(5));
    for (ConsumerRecord<String, String> record : records) {

      String key = record.key();
      Integer count = groupByCounts.get(key);

      if (count == null) {
        count = 0;
      }
      count += 1;

      groupByCounts.put(key, count);
    }
  }
```

```
// in-memory store, not persistent
Map<String, Integer> groupByCounts = new HashMap<>();

try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProperties());
     KafkaProducer<String, Integer> producer = new KafkaProducer<>(producerProperties())) {

  consumer.subscribe(Arrays.asList("A", "B"));

  while (true) { // consumer poll loop
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(5));
    for (ConsumerRecord<String, String> record : records) {

      String key = record.key();
      Integer count = groupByCounts.get(key);

      if (count == null) {
        count = 0;
      }
      count += 1;

      groupByCounts.put(key, count);
    }
  }
```

```
// in-memory store, not persistent
Map<String, Integer> groupByCounts = new HashMap<>();

try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProperties());
     KafkaProducer<String, Integer> producer = new KafkaProducer<>(producerProperties())) {

  consumer.subscribe(Arrays.asList("A", "B"));

  while (true) { // consumer poll loop
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(5));
    for (ConsumerRecord<String, String> record : records) {

      String key = record.key();
      Integer count = groupByCounts.get(key);

      if (count == null) {
        count = 0;
      }
      count += 1;

      groupByCounts.put(key, count);
    }
  }
```

```
while (true) { // consumer poll loop
  ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(5));
  for (ConsumerRecord<String, String> record : records) {

    String key = record.key();
    Integer count = groupByCounts.get(key);

    if (count == null) {
      count = 0;
    }
    count += 1;

    groupByCounts.put(key, count);
  }
}
```

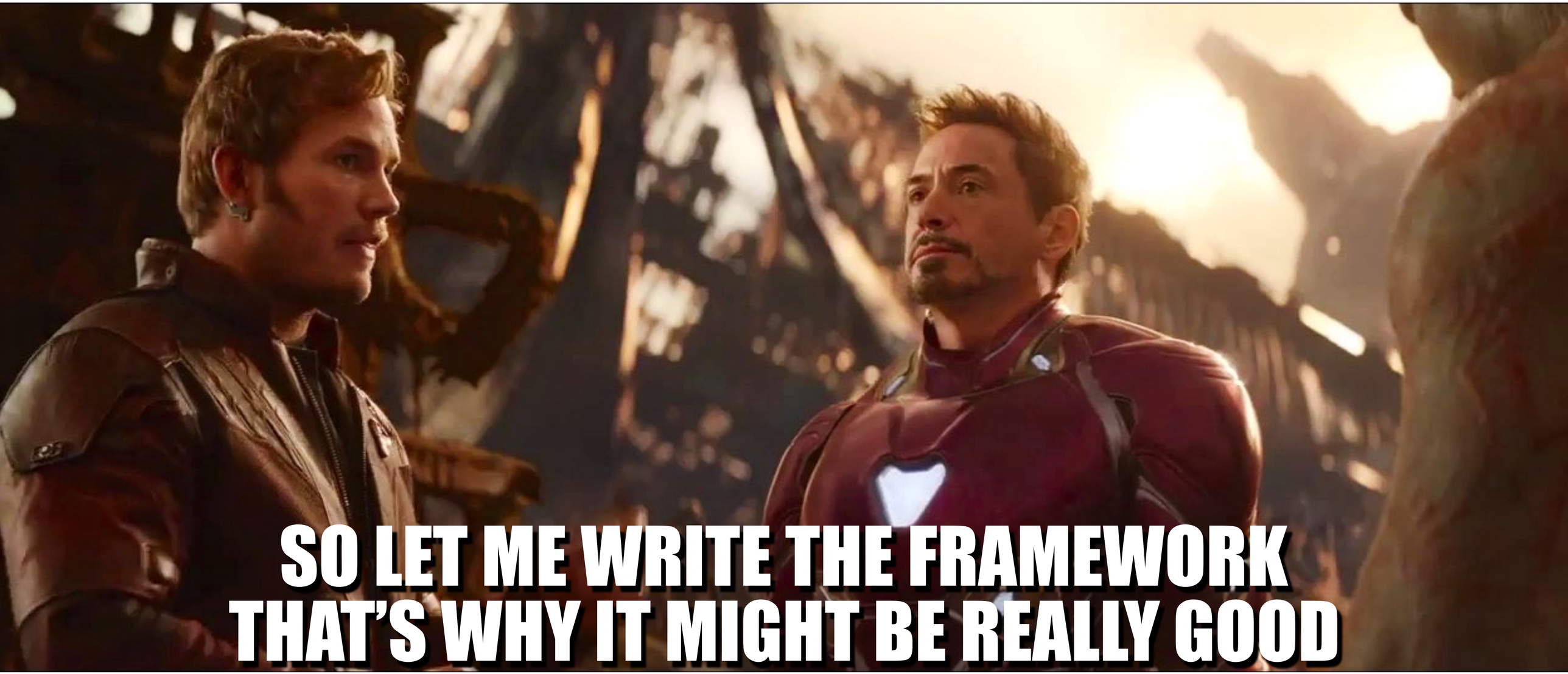
```
  while (true) { // consumer poll loop
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(5));
    for (ConsumerRecord<String, String> record : records) {

      String key = record.key();
      Integer count = groupByCounts.get(key);

      if (count == null) {
        count = 0;
      }
      count += 1; // actually doing something useful

      groupByCounts.put(key, count);
    }
}
```

```
    if (counter++ % sendInterval == 0) {
      for (Map.Entry<String, Integer> groupedEntry : groupByCounts.entrySet()) {

        ProducerRecord<String, Integer> producerRecord =
            new ProducerRecord<>("group-by-counts", groupedEntry.getKey(), groupedEntry.getValue());
        producer.send(producerRecord);
      }
      consumer.commitSync();
    }
  }
}
```

```
    if (counter++ % sendInterval == 0) {
      for (Map.Entry<String, Integer> groupedEntry : groupByCounts.entrySet()) {

        ProducerRecord<String, Integer> producerRecord =
            new ProducerRecord<>("group-by-counts", groupedEntry.getKey(), groupedEntry.getValue());
        producer.send(producerRecord);
      }
      consumer.commitSync();
    }
  }
}
```

```
    if (counter++ % sendInterval == 0) {
      for (Map.Entry<String, Integer> groupedEntry : groupByCounts.entrySet()) {

        ProducerRecord<String, Integer> producerRecord =
            new ProducerRecord<>("group-by-counts", groupedEntry.getKey(), groupedEntry.getValue());
        producer.send(producerRecord);
      }
      consumer.commitSync();
    }
  }
}
```


LET'S TALK ABOUT THIS FRAMEWORK
OF YOURS.
I THINK ITS GOOD, EXCEPT IT SUCKS


SO LET ME WRITE THE FRAMEWORK
THAT'S WHY IT MIGHT BE REALLY GOOD

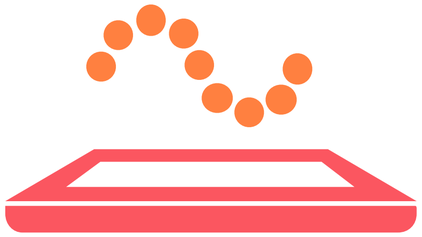
Every framework wants to be when it grows up

Every framework wants to be when it grows up

Scalable

Every framework wants to be when it grows up

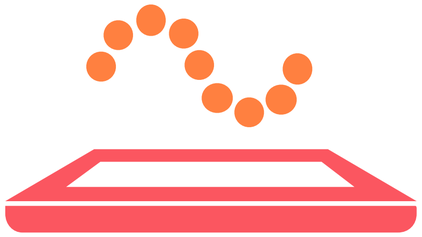
Scalable



Elastic

Every framework wants to be when it grows up

Scalable

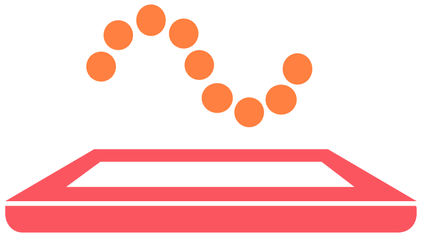


Elastic

Fault-tolerant

Every framework wants to be when it grows up

Scalable



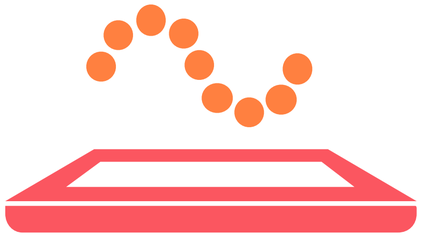
Elastic

Fault-tolerant

Stateful

Every framework wants to be when it grows up

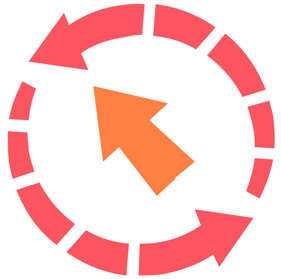
Scalable



Elastic

Fault-tolerant

Stateful



Distributed

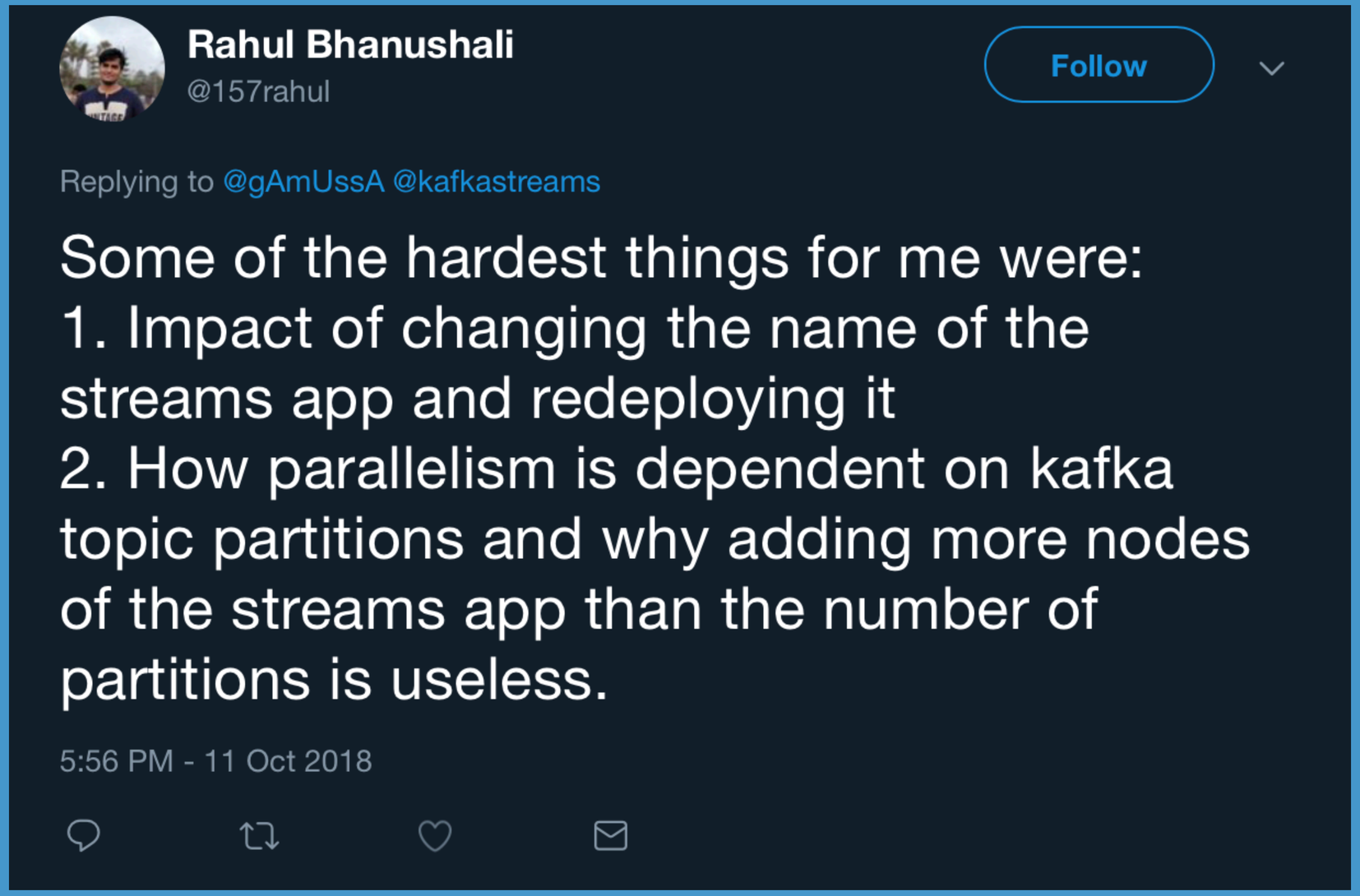
```
final StreamsBuilder streamsBuilder = new StreamsBuilder();
final KStream<String, Long> stream = streamsBuilder.stream(Arrays.asList("A", "B"));


stream.groupByKey()
    .count()
    .toStream()
    .to("group-by-counts",
        Produced.with(Serdes.String(), Serdes.Long()));

final Topology topology = streamsBuilder.build();
final KafkaStreams kafkaStreams = new KafkaStreams(topology, streamsProperties());
kafkaStreams.start();
```

```
final StreamsBuilder streamsBuilder = new StreamsBuilder();
final KStream<String, Long> stream = streamsBuilder.stream(Arrays.asList("A", "B"));

// actual work
stream.groupByKey()
    .count()
    .toStream()
    .to("group-by-counts",
        Produced.with(Serdes.String(), Serdes.Long()));

final Topology topology = streamsBuilder.build();
final KafkaStreams kafkaStreams = new KafkaStreams(topology, streamsProperties());
kafkaStreams.start();
```

```
final StreamsBuilder streamsBuilder = new StreamsBuilder();
final KStream<String, Long> stream = streamsBuilder.stream(Arrays.asList("A", "B"));

// actual work
stream.groupByKey()
    .count()
    .toStream()
    .to("group-by-counts",
        Produced.with(Serdes.String(), Serdes.Long()));

final Topology topology = streamsBuilder.build();
final KafkaStreams kafkaStreams = new KafkaStreams(topology, streamsProperties());
kafkaStreams.start();
```

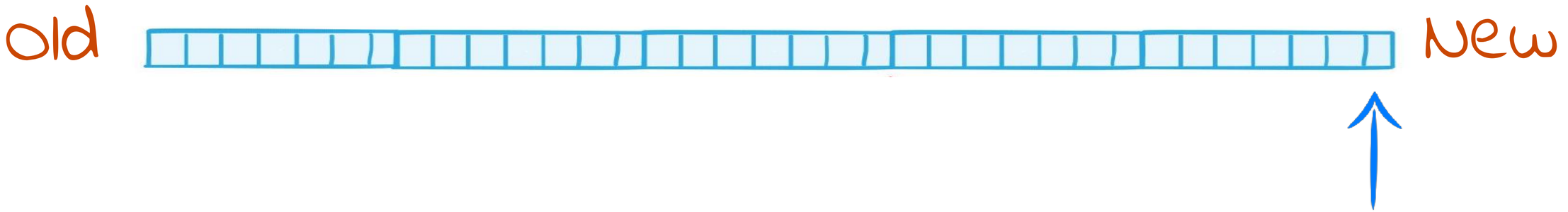
```
final StreamsBuilder streamsBuilder = new StreamsBuilder();
final KStream<String, Long> stream = streamsBuilder.stream(Arrays.asList("A", "B"));

// actual work
stream.groupByKey()
    .count()
    .toStream()
    .to("group-by-counts",
        Produced.with(Serdes.String(), Serdes.Long()));

final Topology topology = streamsBuilder.build();
final KafkaStreams kafkaStreams = new KafkaStreams(topology, streamsProperties());
kafkaStreams.start();
```

confluent



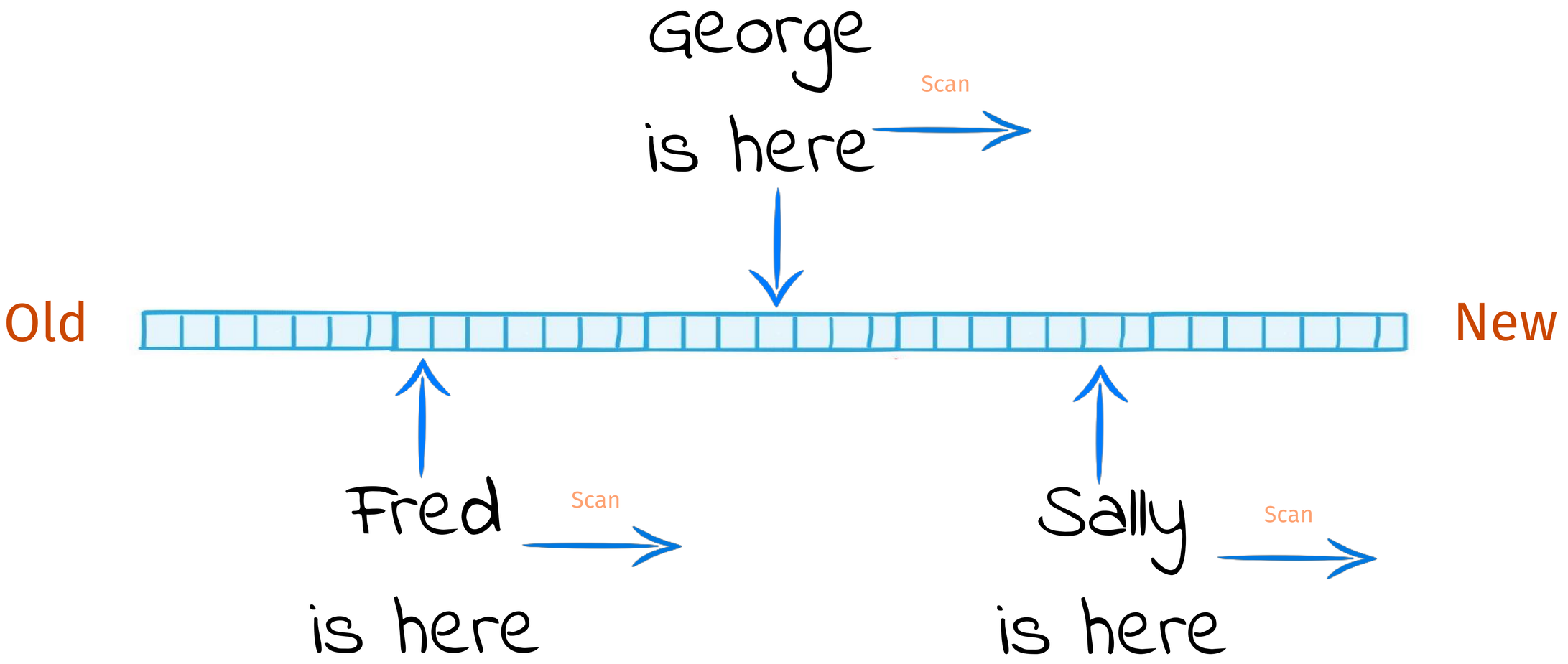
https://twitter.com/157rahul/status/1050505569746841600

The log is a simple idea

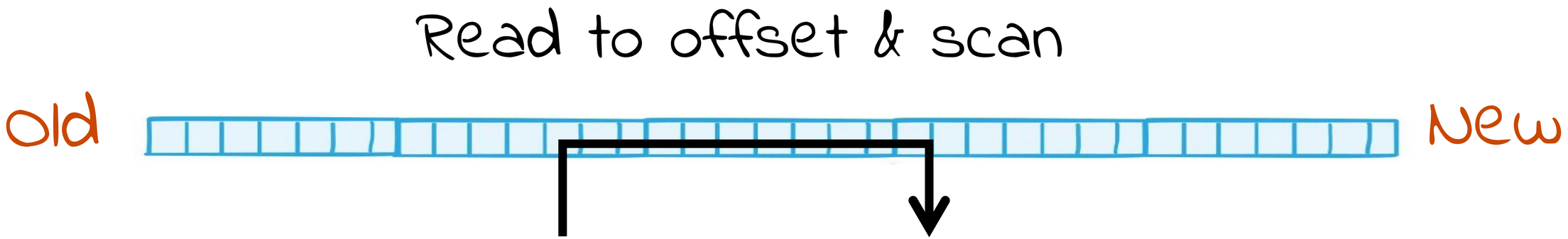


Messages are added at the end of the log

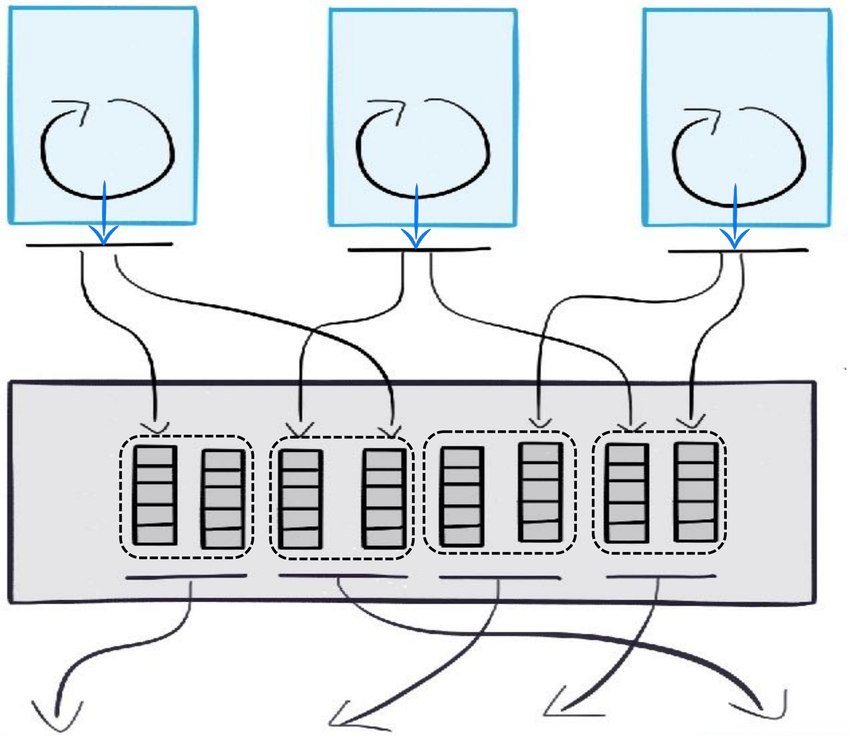
Consumers have a position all of their own



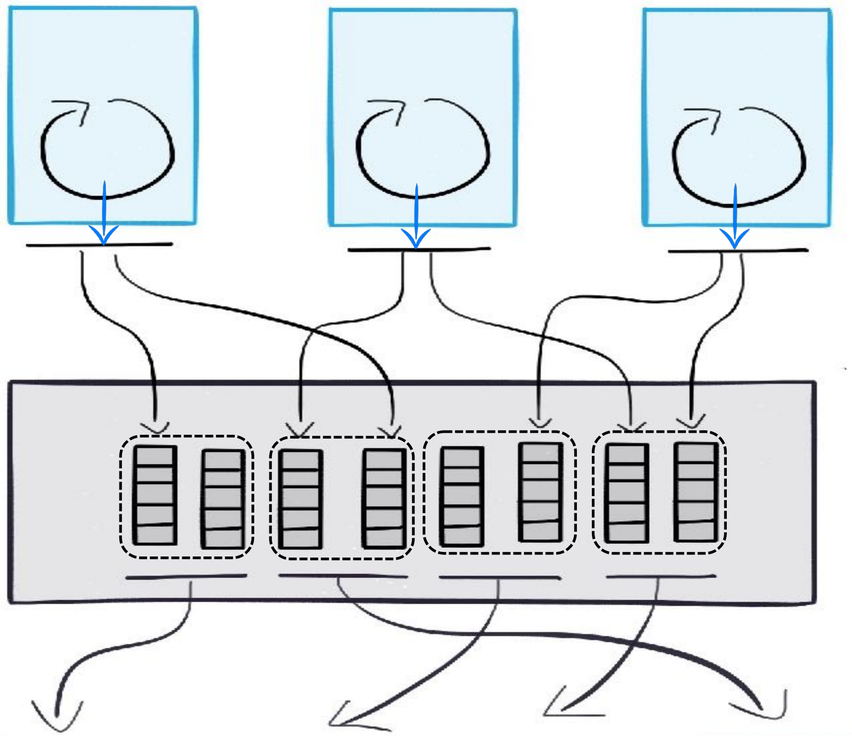
Only Sequential Access



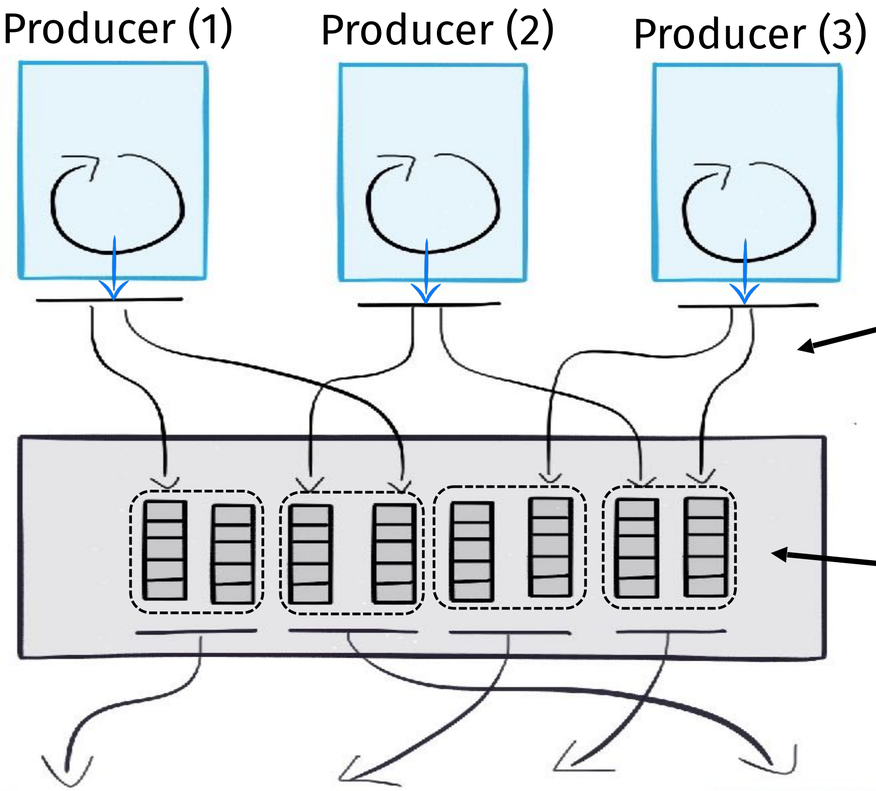
Shard data to get scalability



Shard data to get scalability



Shard data to get scalability



Messages are sent to different partitions

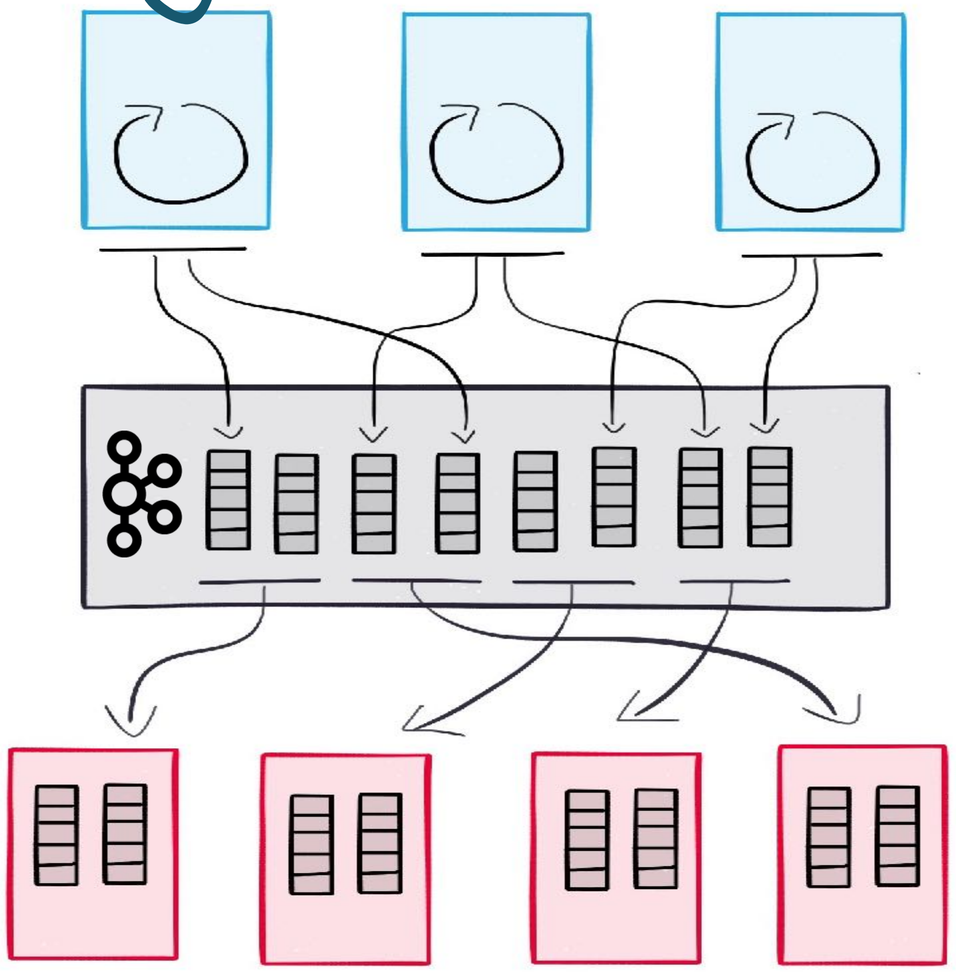
Partitions live on different machines


CONSUMER GROUP
COORDINATOR

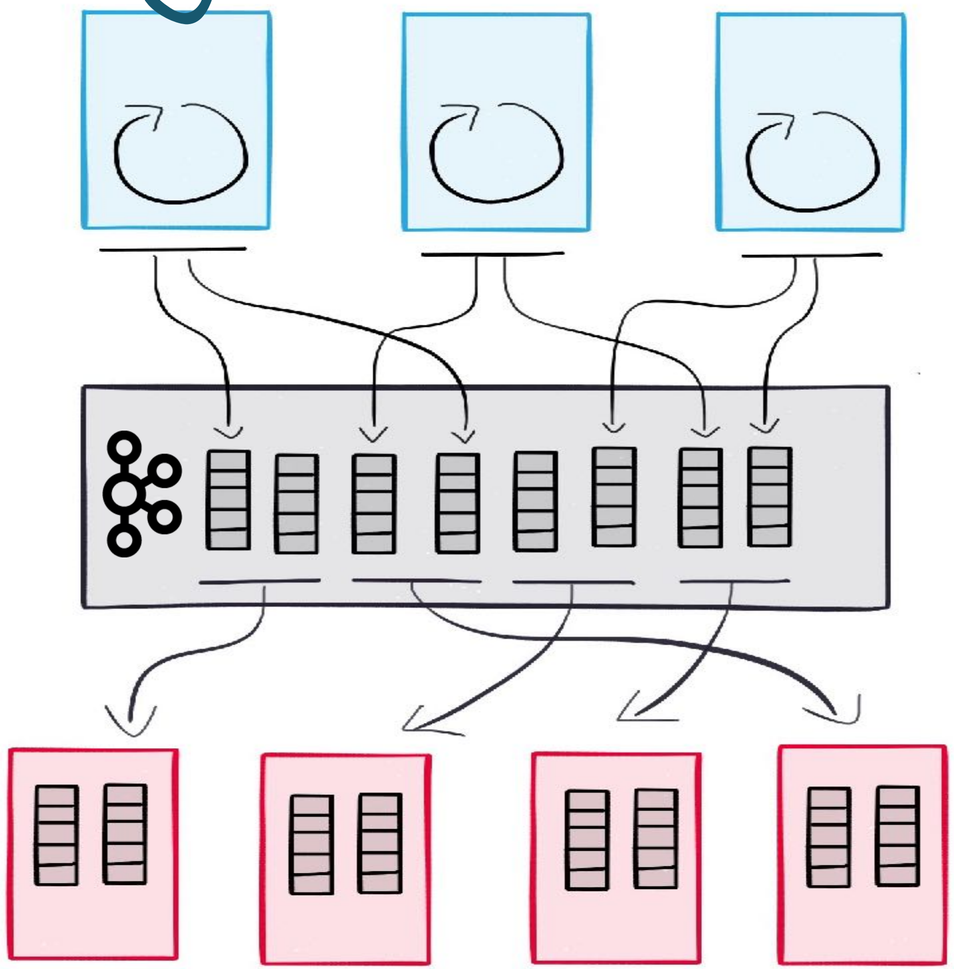

CONSUMERS
CONSUMER GROUP COORDINATOR


CONSUMERS
CONSUMER GROUP COORDINATOR
CONSUMER GROUP

Linearly Scalable Architecture



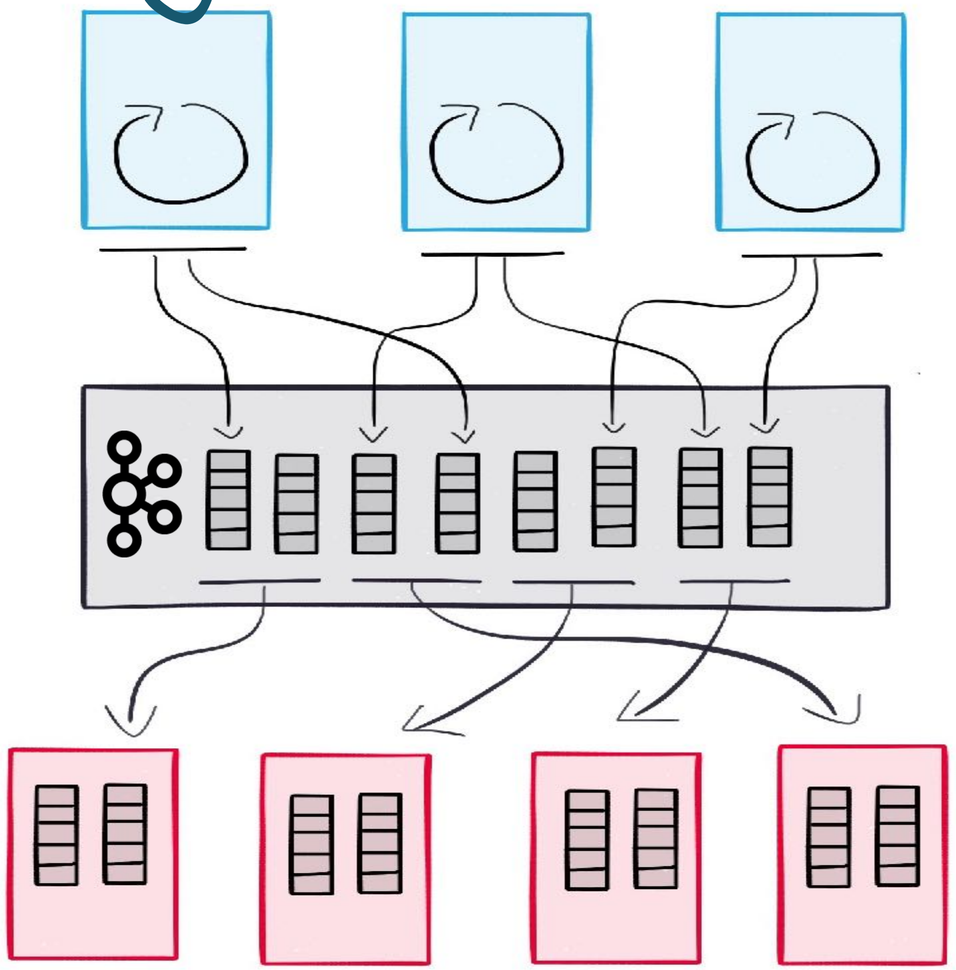
Linearly Scalable Architecture



Single topic:

- Many producers machines
- Many consumer machines
- Many Broker machines

Linearly Scalable Architecture



Single topic:

- Many producers machines
- Many consumer machines
- Many Broker machines

No Bottleneck!!

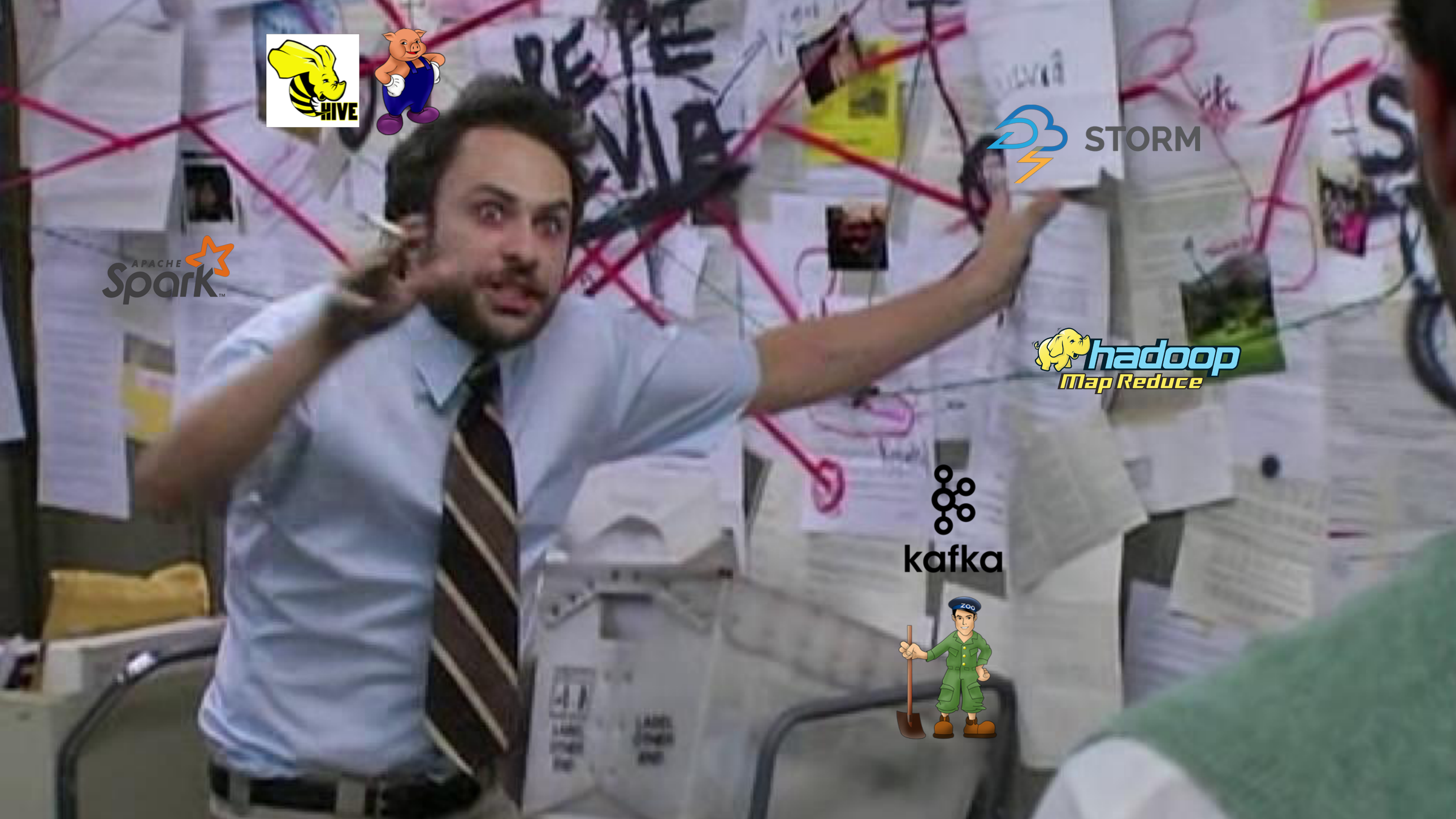
confluent

Talk is cheap! Show me code!

https://cnfl.io/streams-movie-demo

As developers,
we want to build APPS

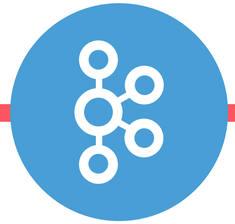
not INFRASTRUCTURE

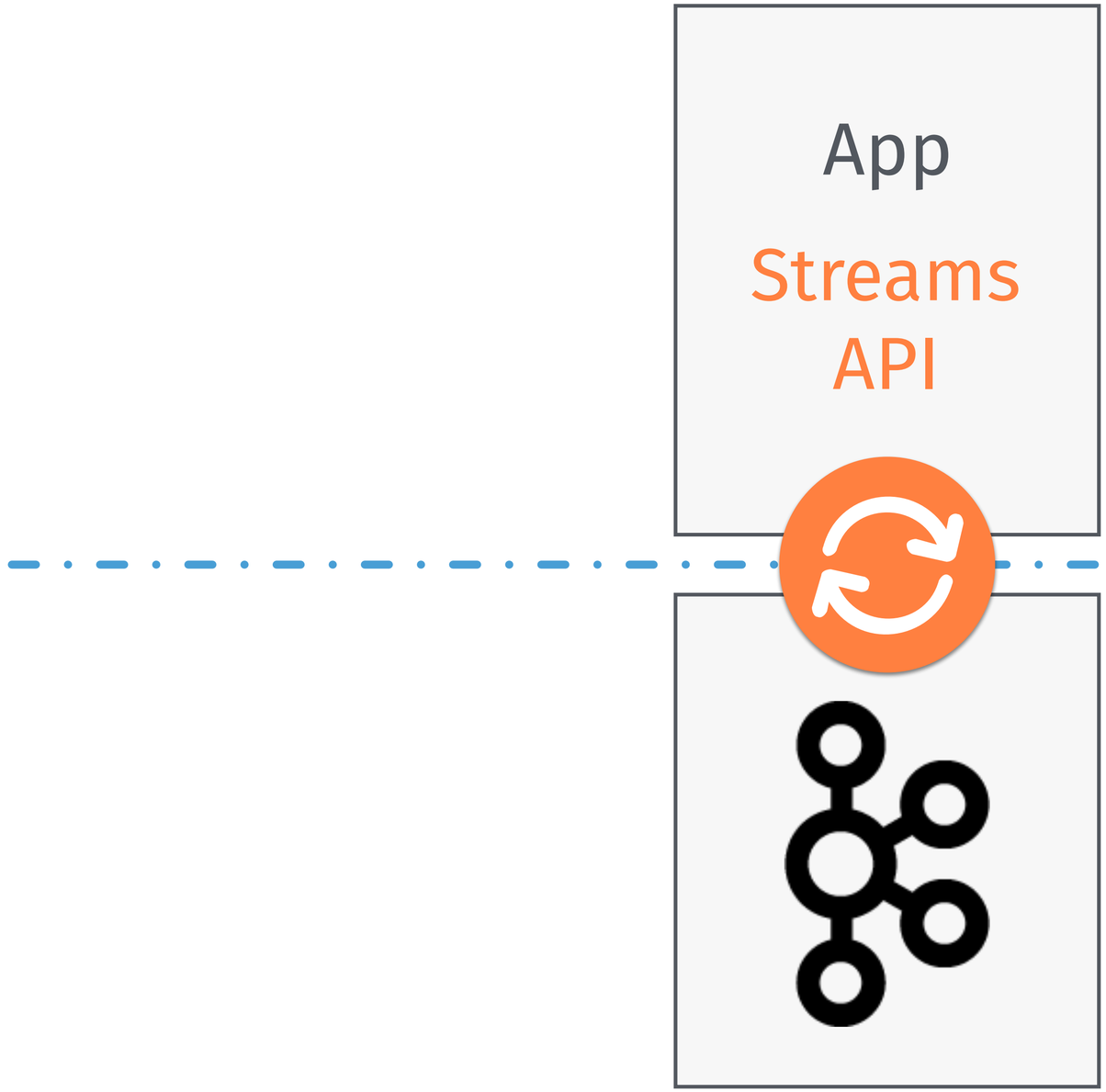

HIVE
STORM
APACHE
Spark
hadoop
Map Reduce
kafka
ZOO

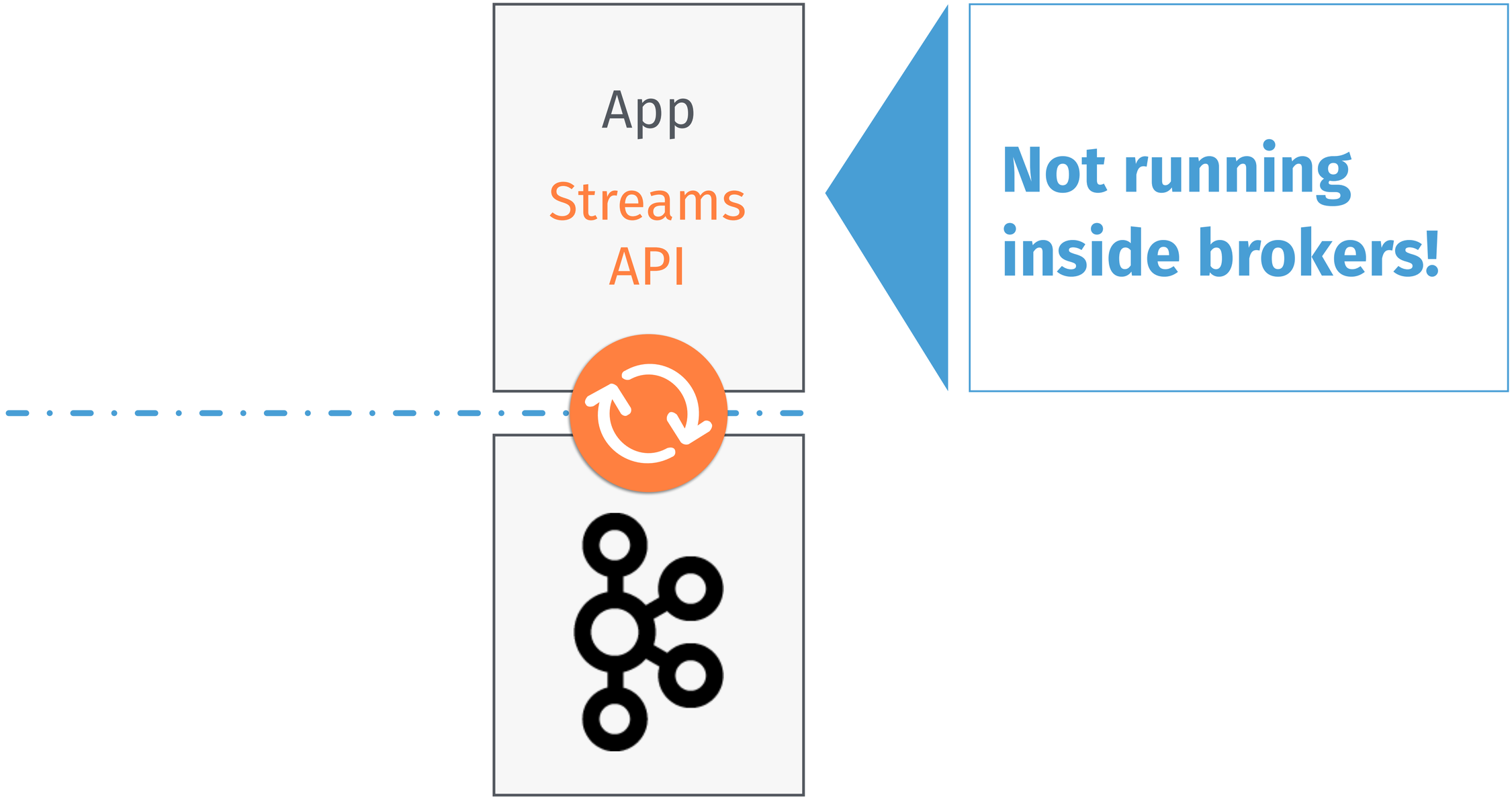
the KAFKA STREAMS **API** is a

JAVA API to

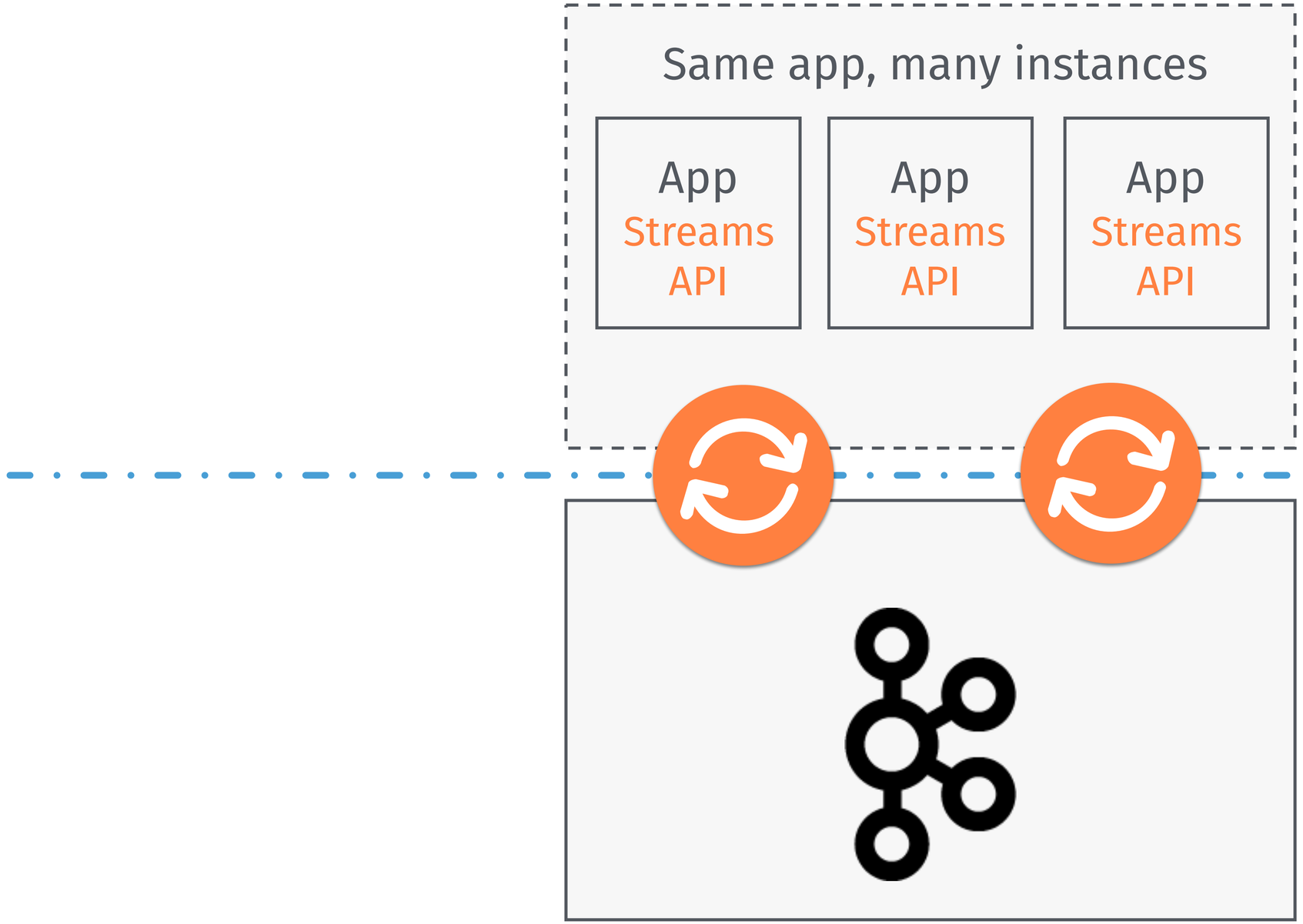
BUILD REAL-TIME **APPLICATIONS**




App
Streams
API

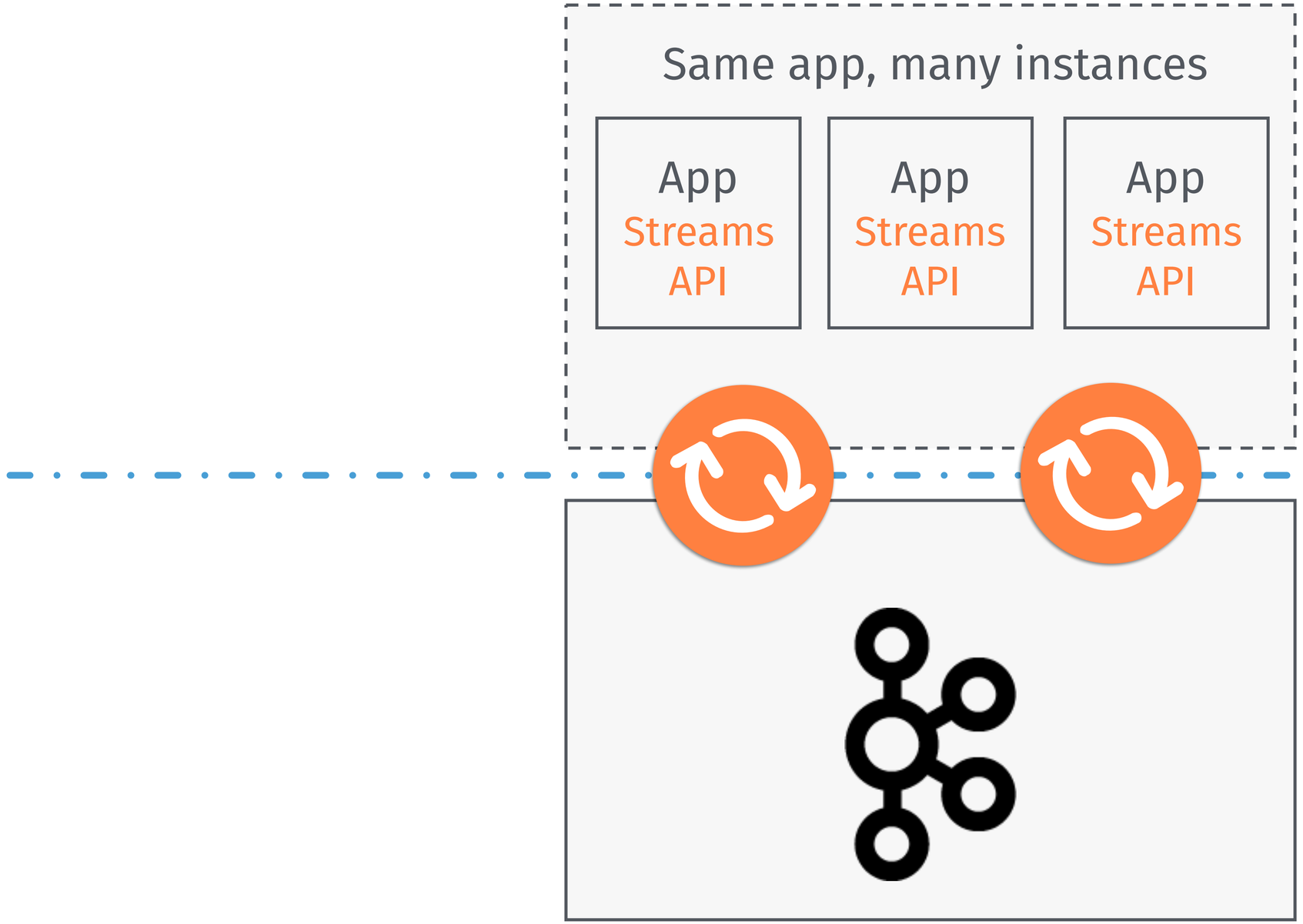


Not running inside brokers!



Brokers?

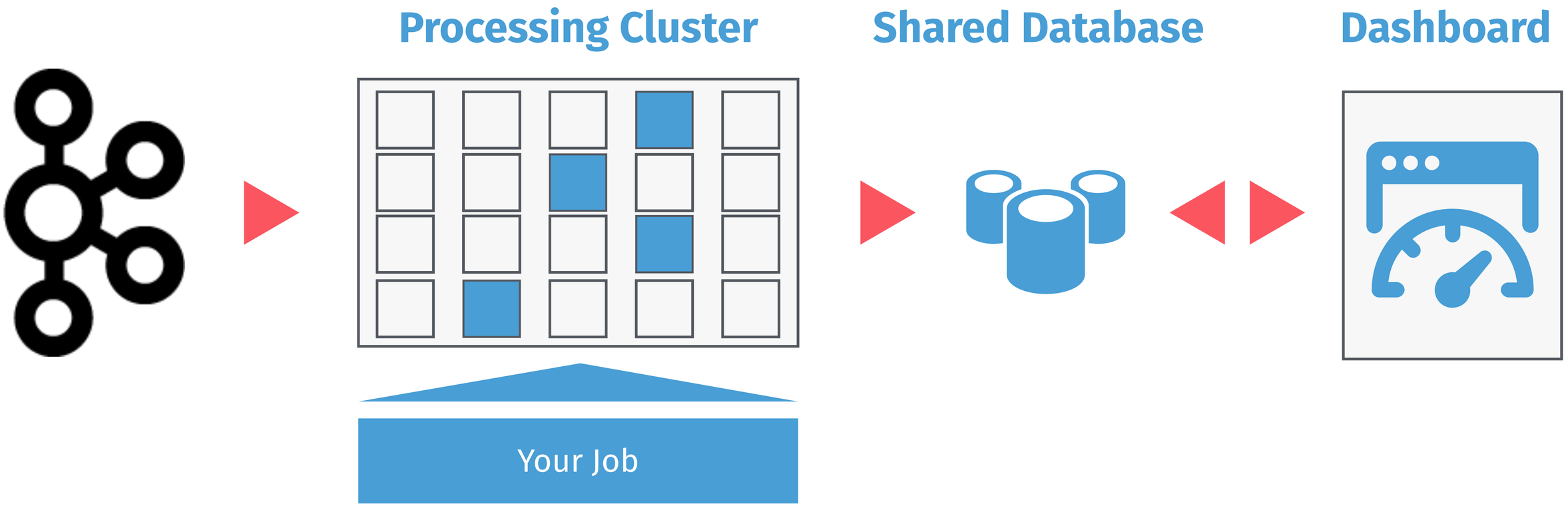
Nope!



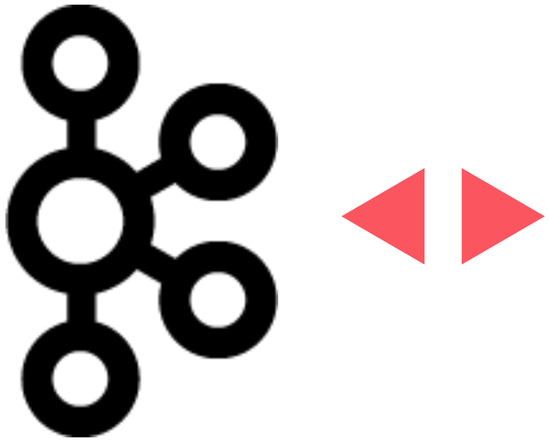
Brokers?

Nope!

Before



After



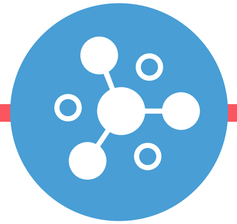
Dashboard

APP

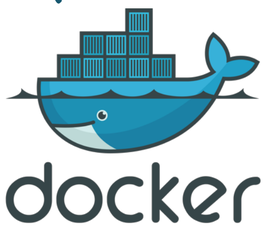
Streams API

this means you can

DEPLOY your app ANYWHERE using WHATEVER TECHNOLOGY YOU WANT



So many places to run you app!

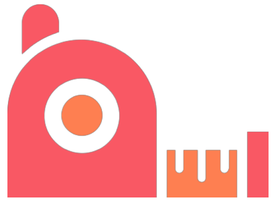



Physical

...and many more...

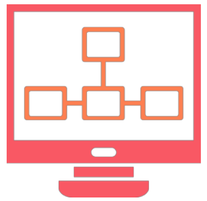
Things Kafka Stream Does

Enterprise Support



Open Source

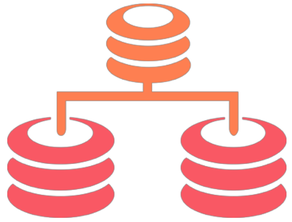
Runs Everywhere



Elastic, Scalable, Fault-tolerant

Kafka Security Integration

Powerful Processing incl. Filters, Transforms, Joins, Aggregations, Windowing



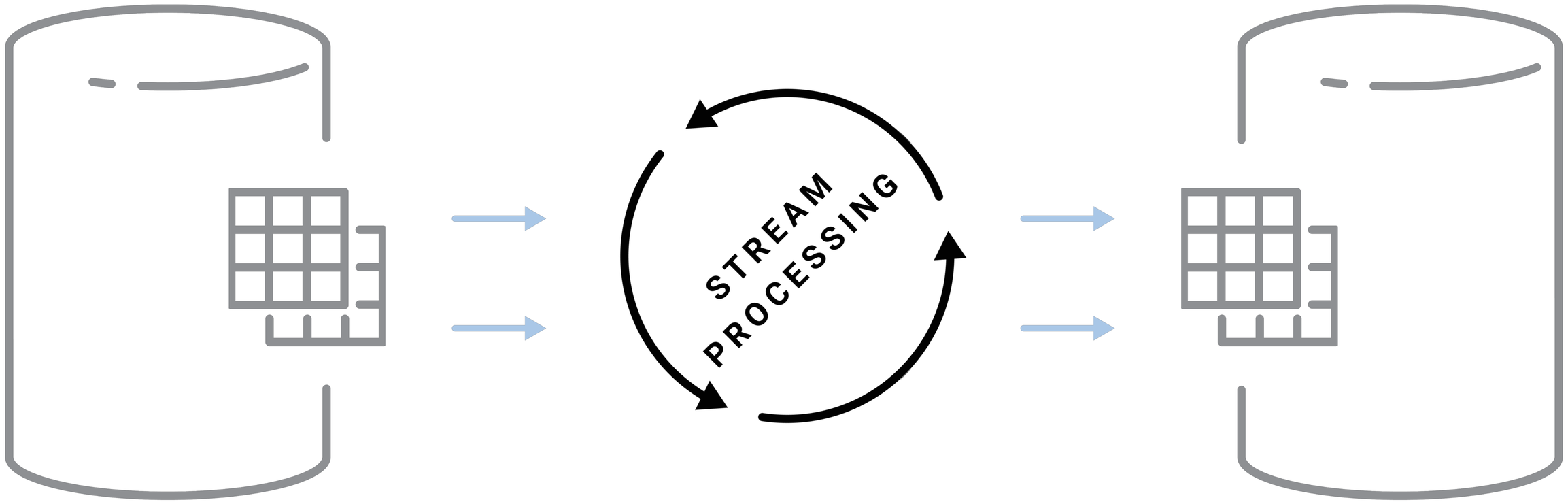
Supports Streams and Tables

Exactly-Once Processing

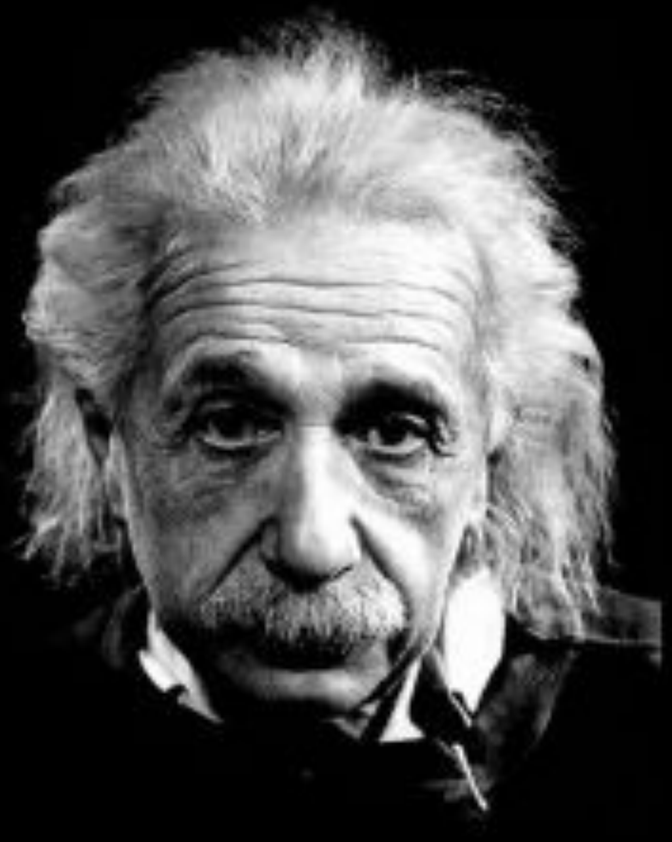
Event-Time Processing

Table-Stream Duality

Table-Stream Duality



SPECIAL RELATIVITY



OF STREAMS AND TABLES

TABLE

STREAM

TABLE

TABLE

Gwen	1

STREAM

TABLE

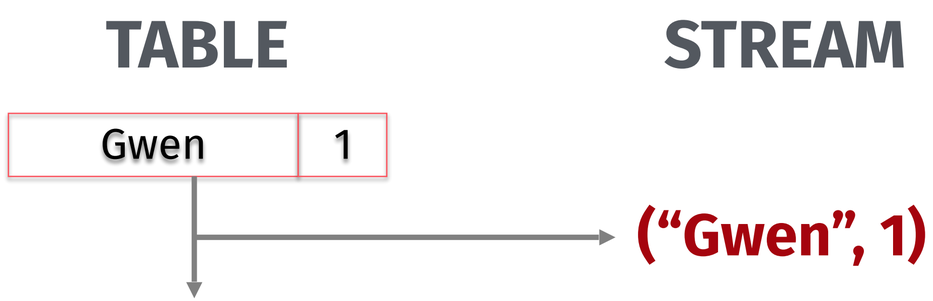


TABLE
STREAM
TABLE
Gwen
1
("Gwen", 1)


TABLE
STREAM
TABLE
Gwen
1
("Gwen", 1)
Gwen
1

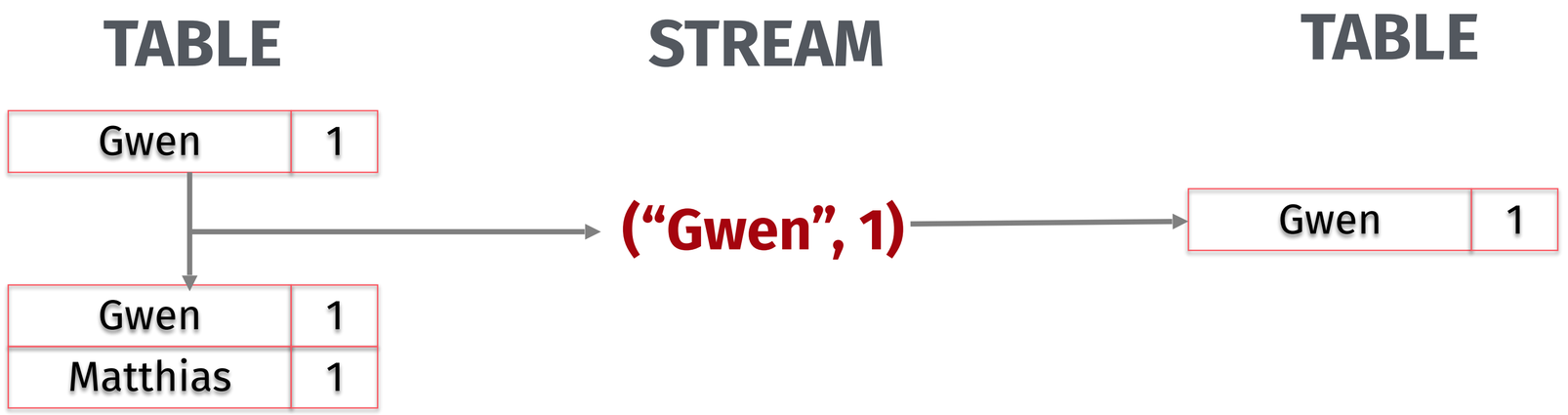


TABLE
STREAM
TABLE
Gwen
1
("Gwen", 1)
Gwen
1
Gwen
1
Matthias
1

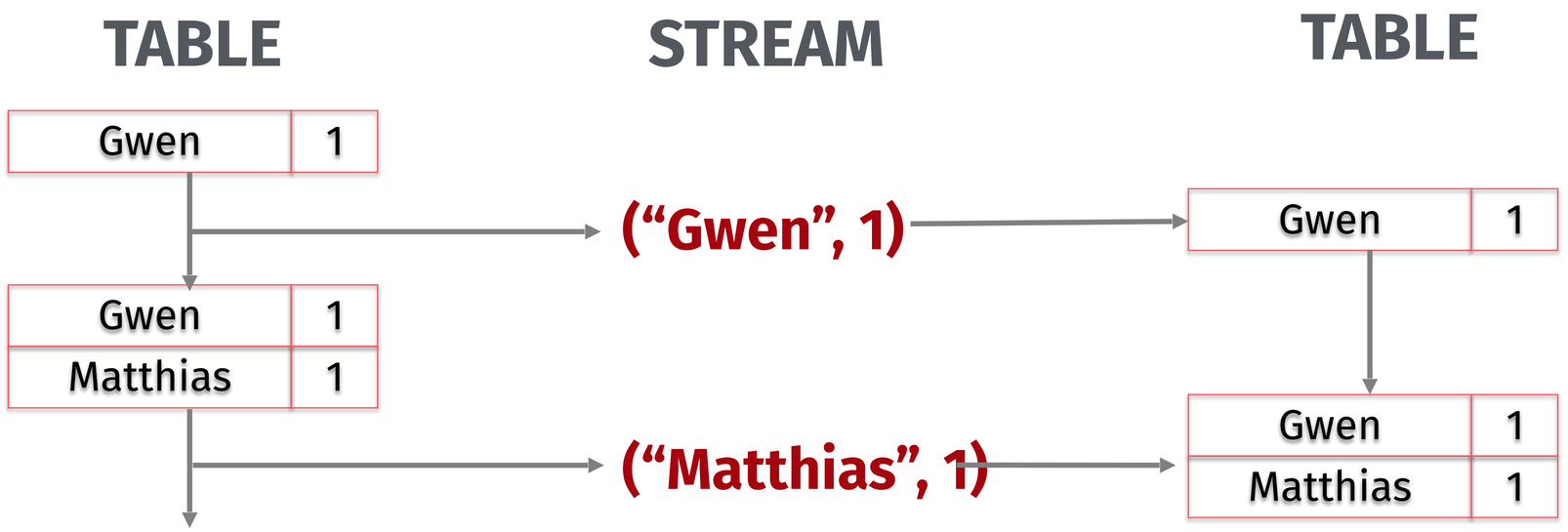


TABLE
STREAM
TABLE
Gwen
1
Gwen
1
Matthias
1
(“Gwen”, 1)
(“Matthias”, 1)
Gwen
1
Gwen
1
Matthias
1

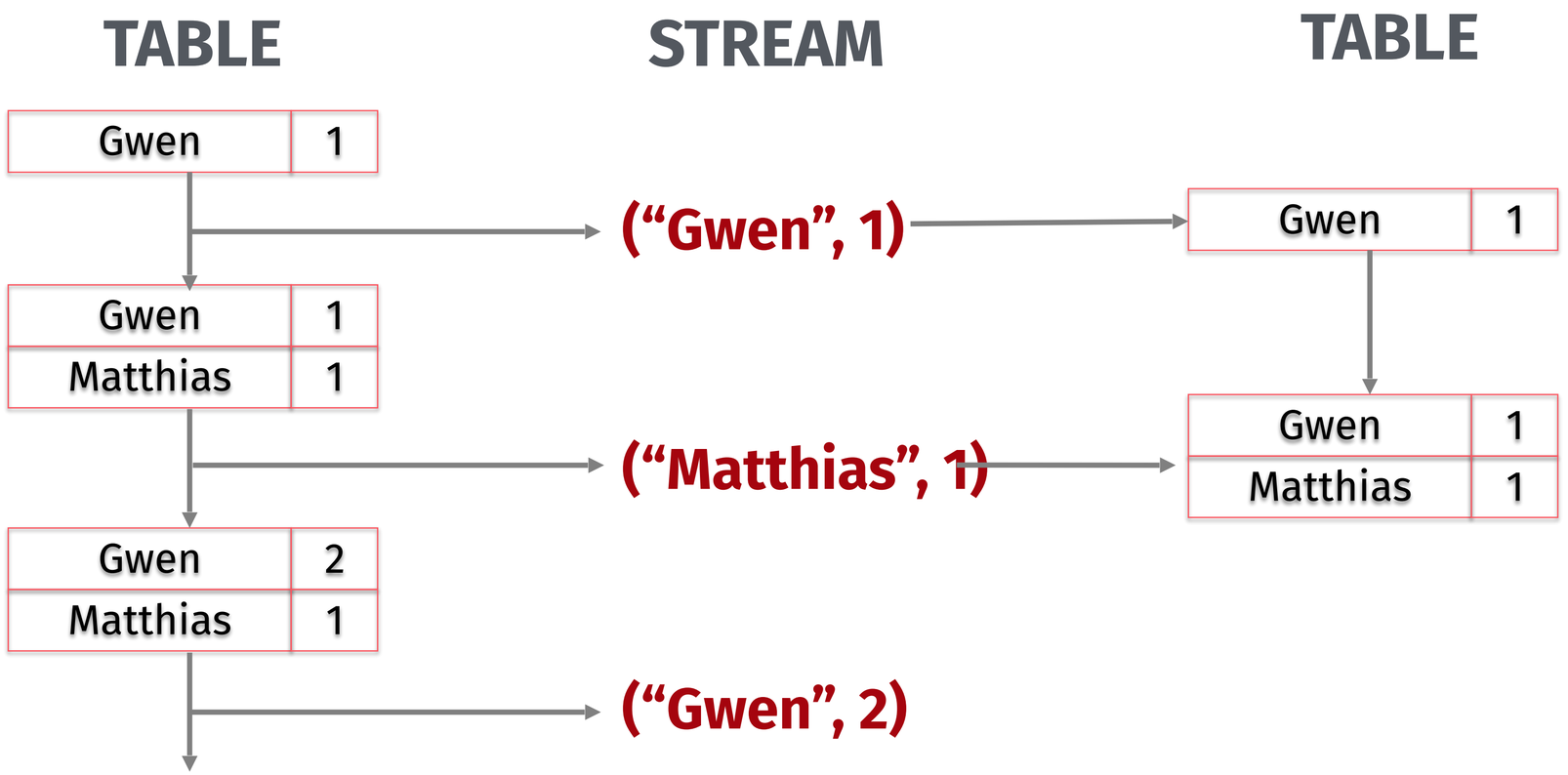


TABLE
STREAM
TABLE
Gwen 1
("Gwen", 1)
Gwen 1
Gwen 1
Matthias 1
("Matthias", 1)
Gwen 1
Matthias 1
Gwen 2
Matthias 1
("Gwen", 2)

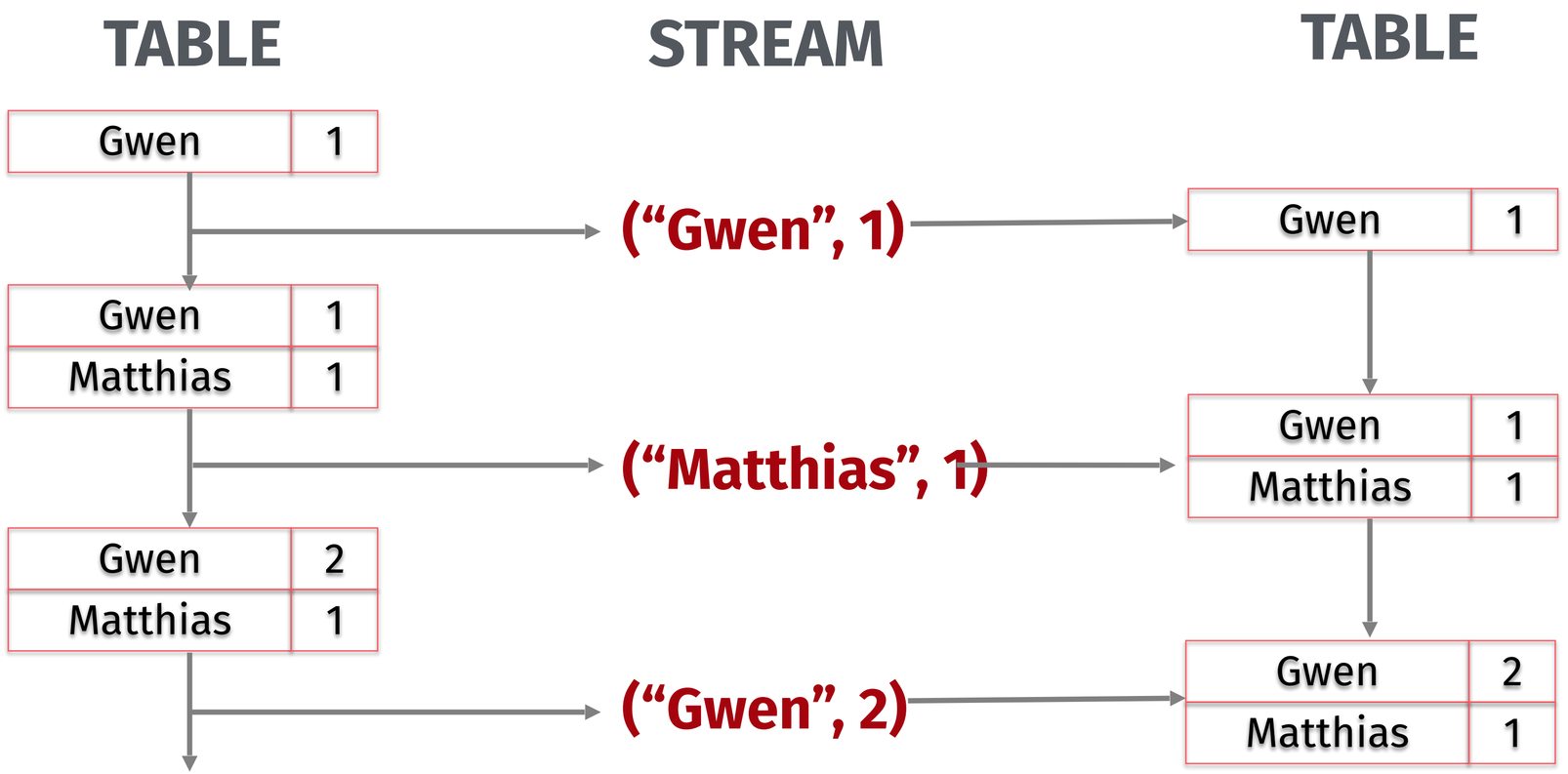


TABLE
STREAM
TABLE
Gwen 1
Gwen 1
Matthias 1
Gwen 2
Matthias 1
(“Gwen”, 1)
(“Matthias”, 1)
(“Gwen”, 2)
Gwen 1
Gwen 1
Matthias 1
Gwen 2
Matthias 1

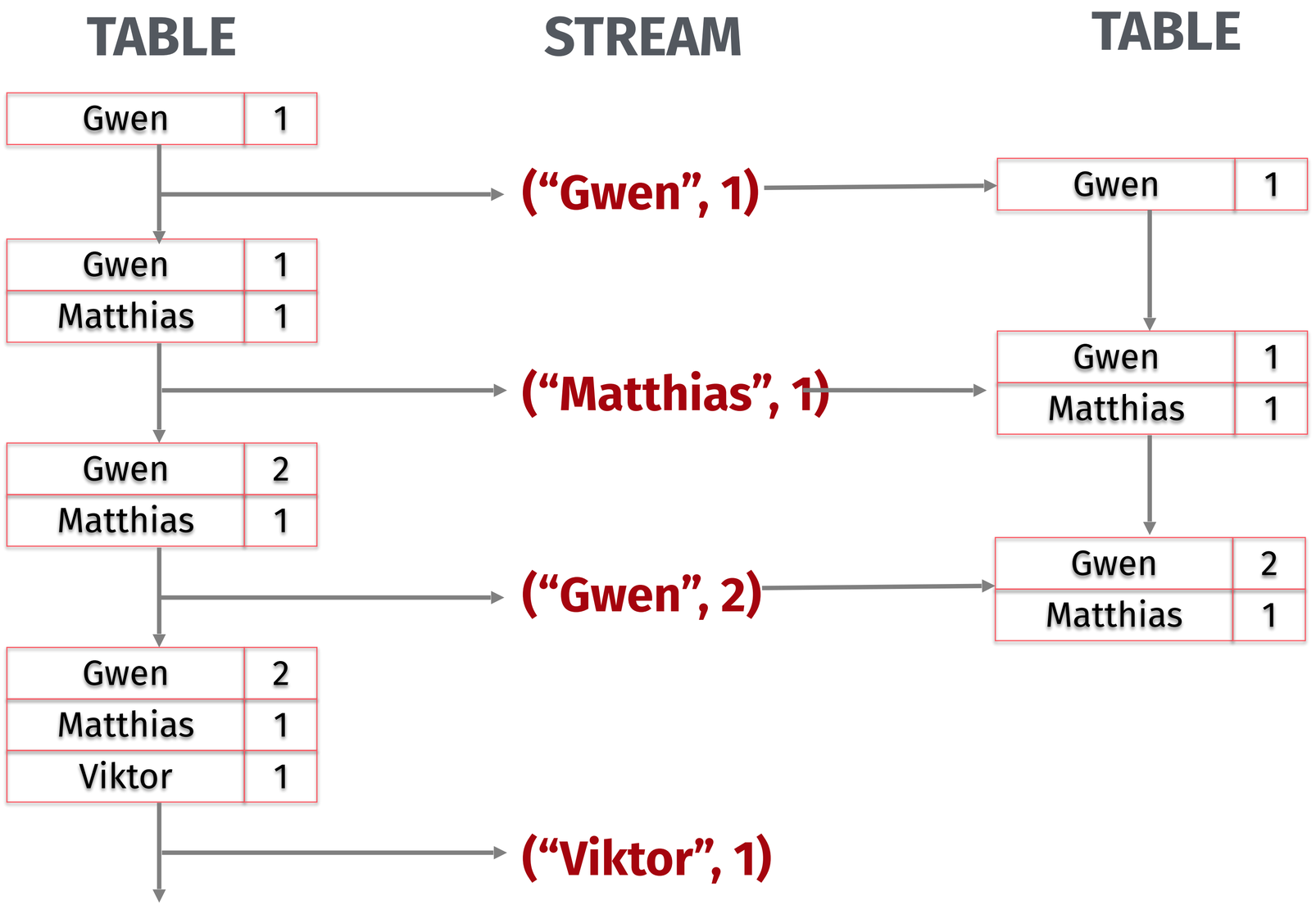


TABLE
STREAM
TABLE
Gwen 1
("Gwen", 1)
Gwen 1
Gwen 1
Matthias 1
("Matthias", 1)
Gwen 1
Matthias 1
Gwen 2
Matthias 1
("Gwen", 2)
Gwen 2
Matthias 1
Gwen 2
Matthias 1
Viktor 1
("Viktor", 1)

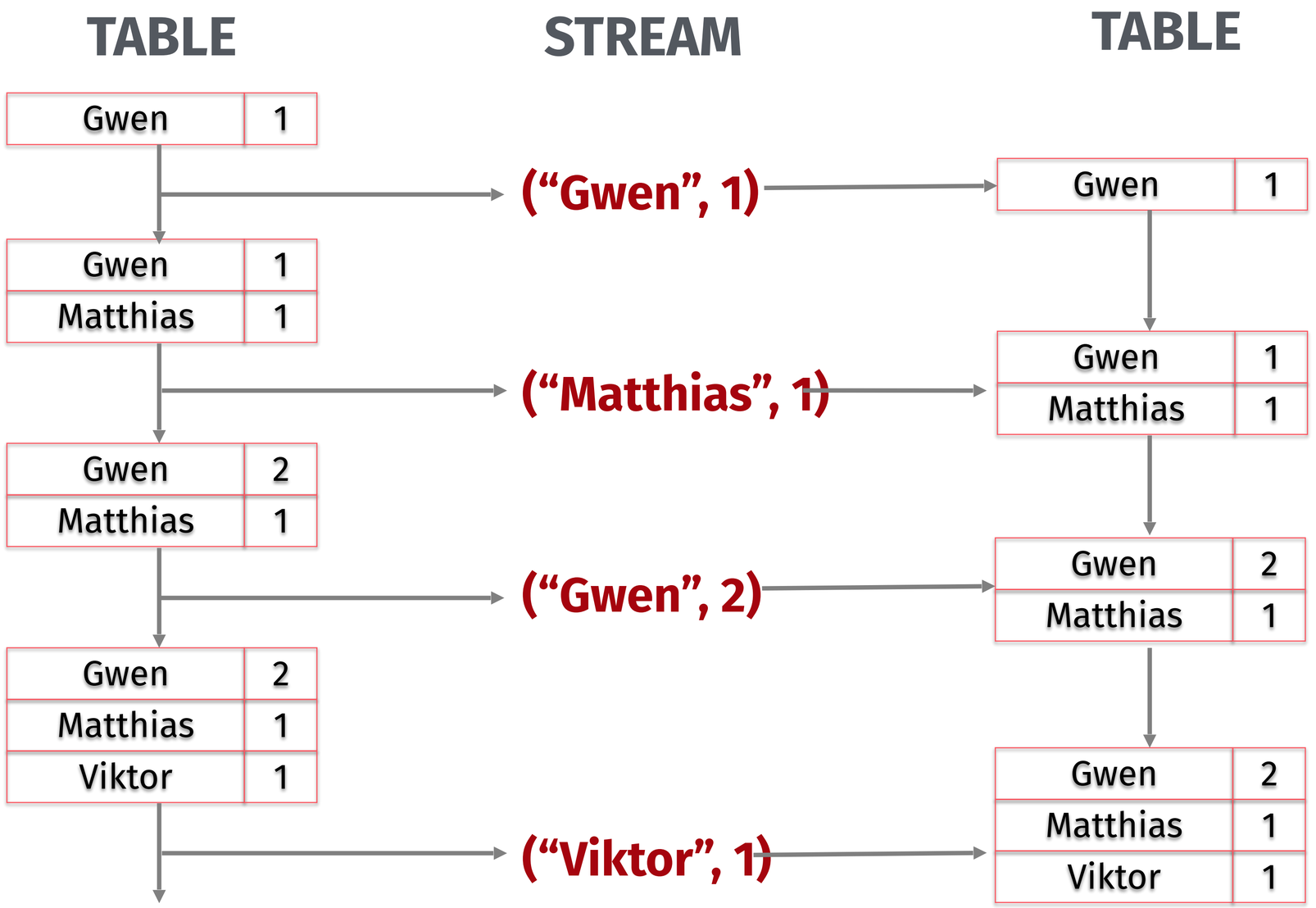
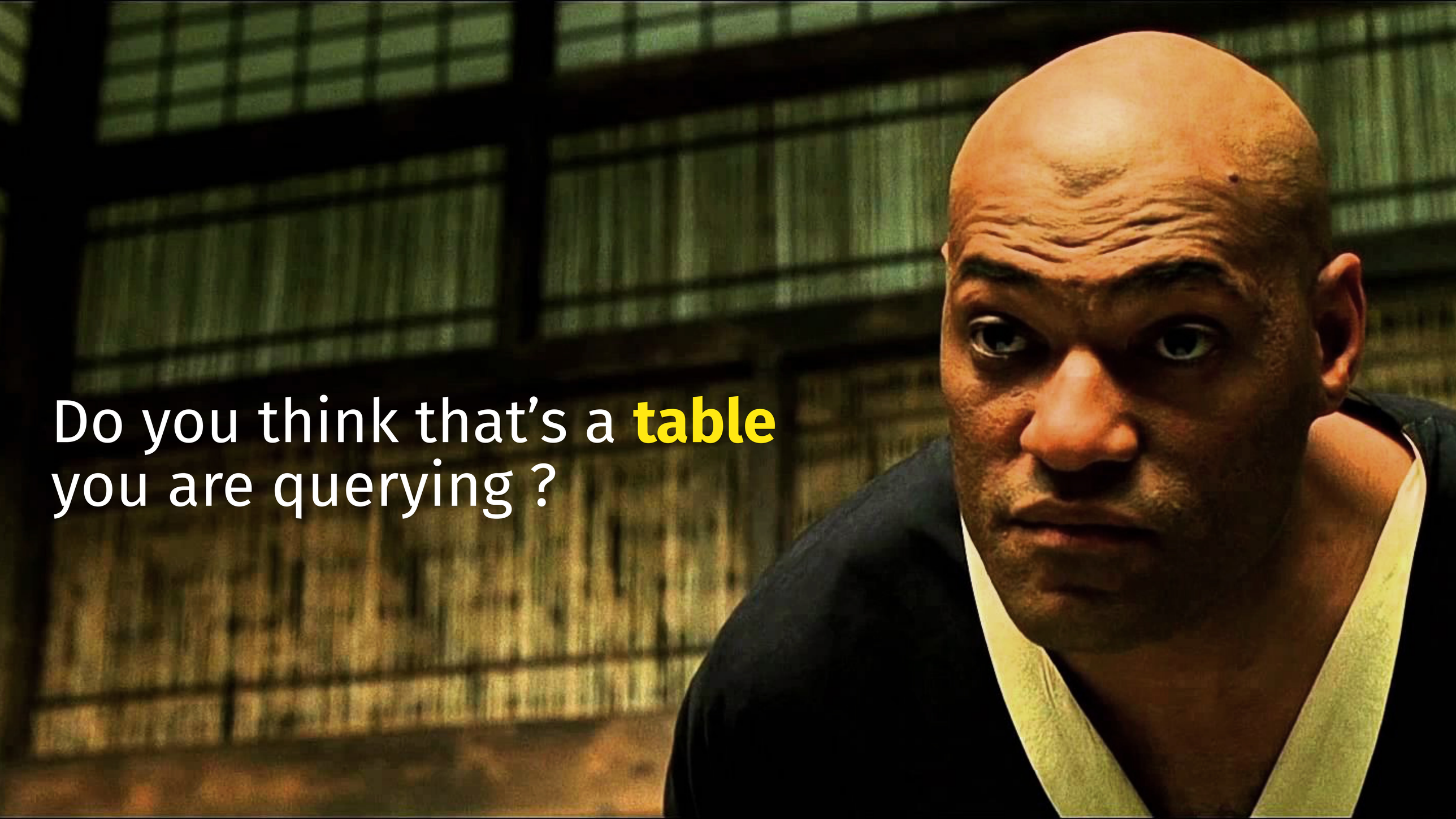


TABLE
STREAM
TABLE
Gwen 1
("Gwen", 1)
Gwen 1
Matthias 1
("Matthias", 1)
Gwen 2
Matthias 1
("Gwen", 2)
Gwen 2
Matthias 1
Viktor 1
("Viktor", 1)


Do you think that’s a **table**
you are querying ?

confluent

Talk is cheap! Show me code!

confluent

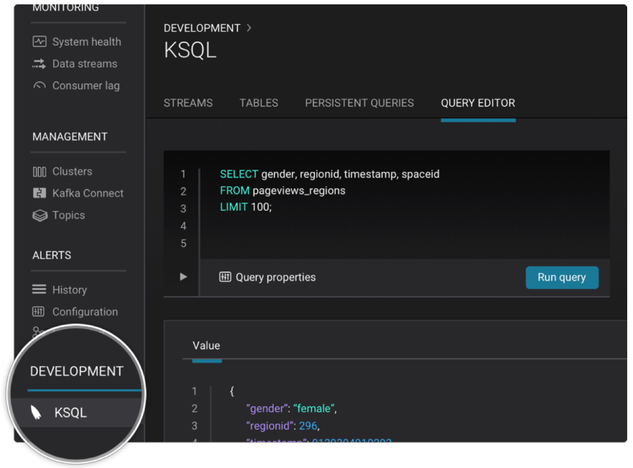
What’s next?

confluent

https://twitter.com/IDispose/status/1048602857191170054

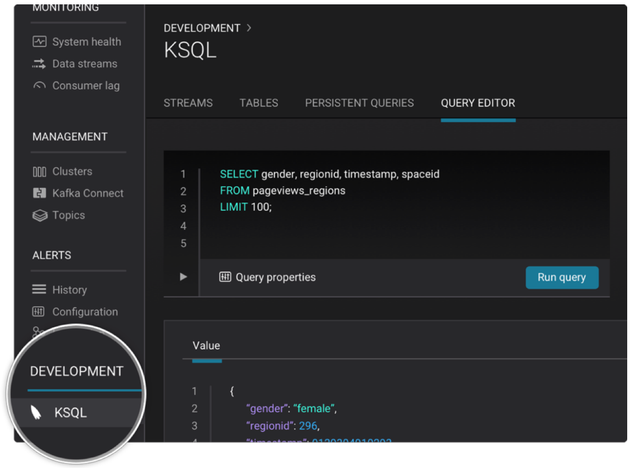
KSQL #FTW

KSQL #FTW


MONITORING
System health
Data streams
Consumer lag
MANAGEMENT
Clusters
Kafka Connect
Topics
ALERTS
History
Configuration
DEVELOPMENT
KSQL
DEVELOPMENT ›
KSQL
STREAMS
TABLES
PERSISTENT QUERIES
QUERY EDITOR
SELECT gender, regionid, timestamp, spaceid
FROM pageviews_regions
LIMIT 100;
Query properties
Run query
Value
{
"gender": "female",
"regionid": 296,

1 UI

KSQL #FTW

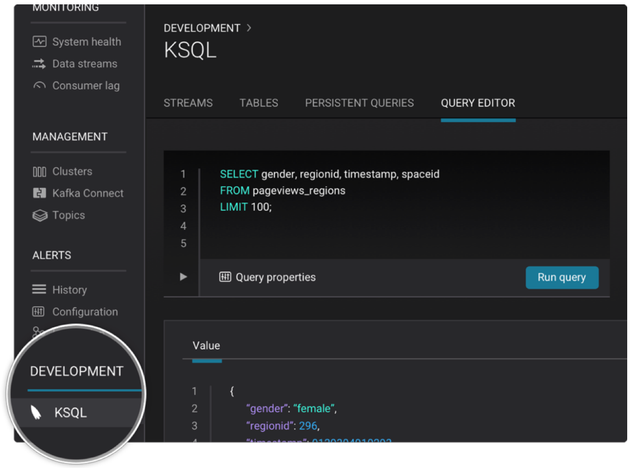



```
ksql>
```

1 UI

2 CLI

KSQL #FTW




```
ksql>
```

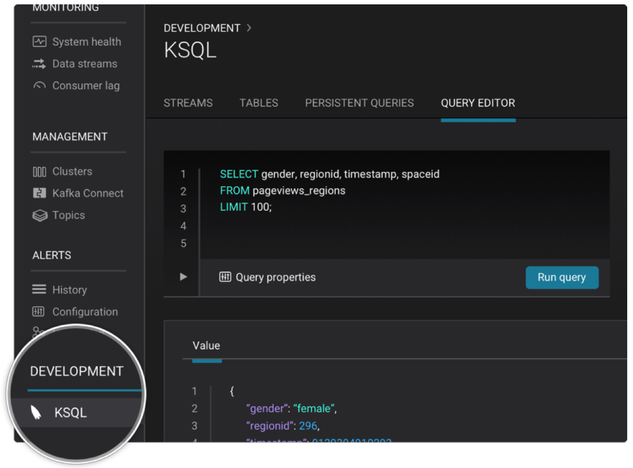
```
POST /query
```

1 UI

2 CLI

3 REST

KSQL #FTW



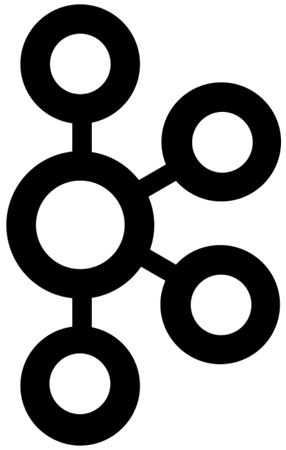

```
ksql>
```

```
POST /query
```

1. UI
2. CLI
3. REST
4. Headless

Interaction with Kafka

Kafka
(data)

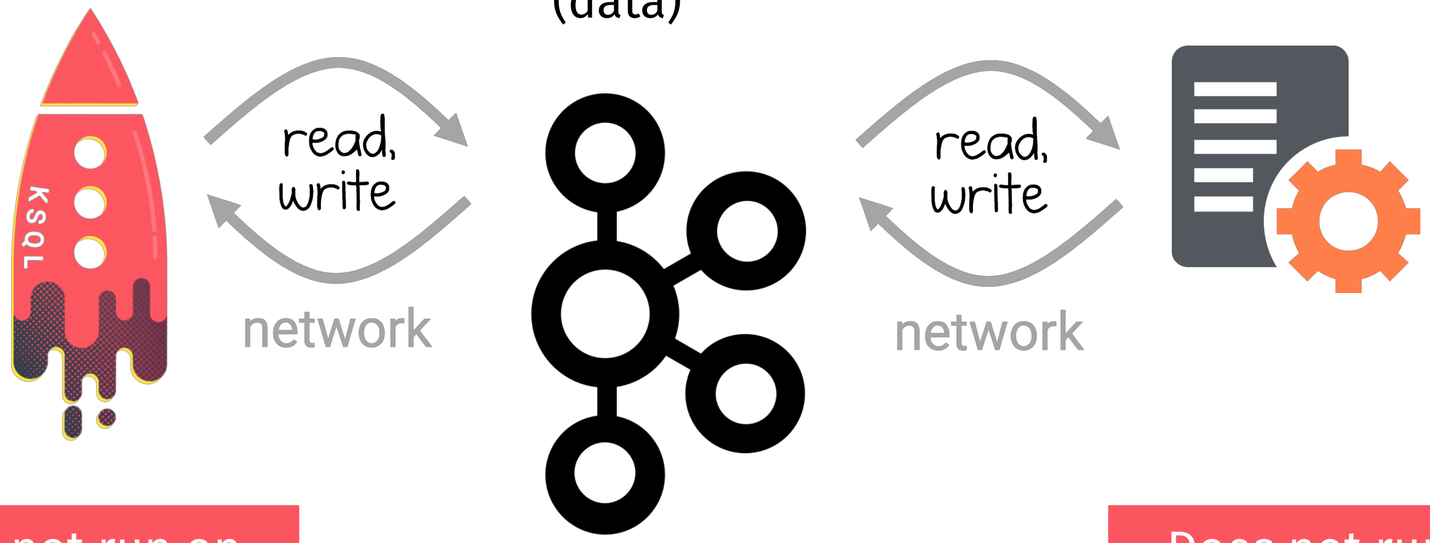


Interaction with Kafka

KSQL
(processing)

Kafka
(data)

JVM application
with Kafka Streams (processing)

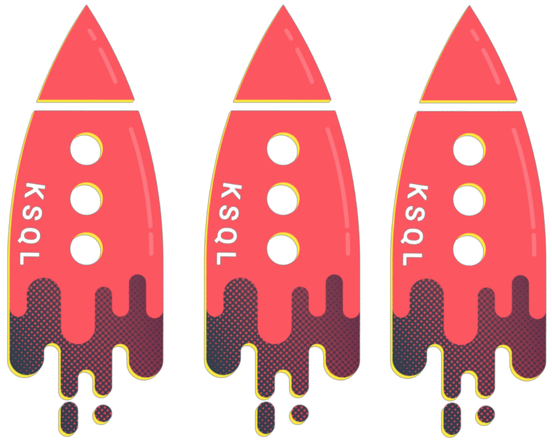


Does not run on Kafka brokers

Does not run on Kafka brokers

Interaction with Kafka

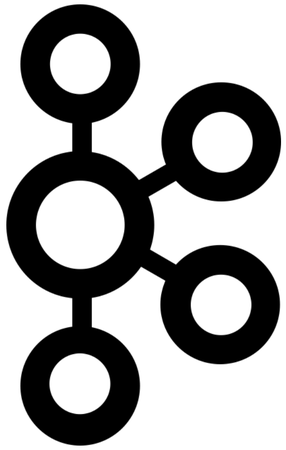
KSQL
(processing)



read, write

network

Kafka
(data)



read, write

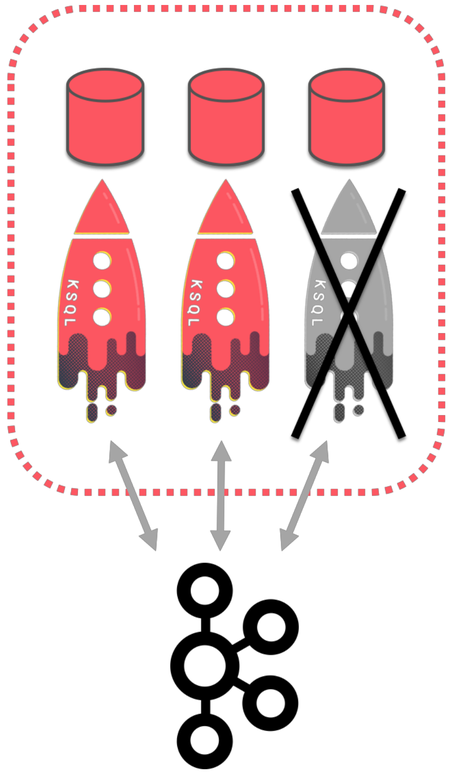
network

JVM application
with Kafka Streams (processing)

Does not run on Kafka brokers

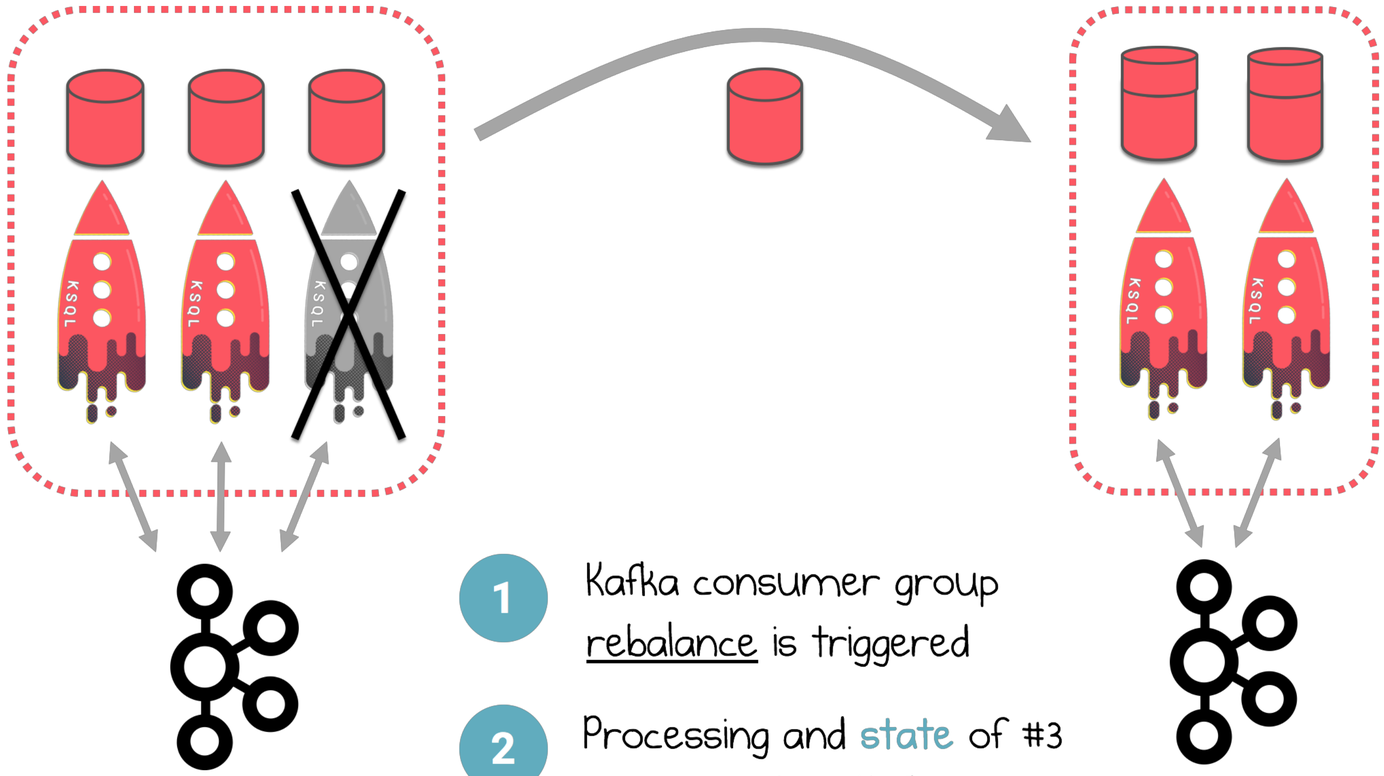
Does not run on Kafka brokers

Fault-Tolerance, powered by Kafka



Fault-Tolerance, powered by Kafka

#3 died so #1 and #2 take over

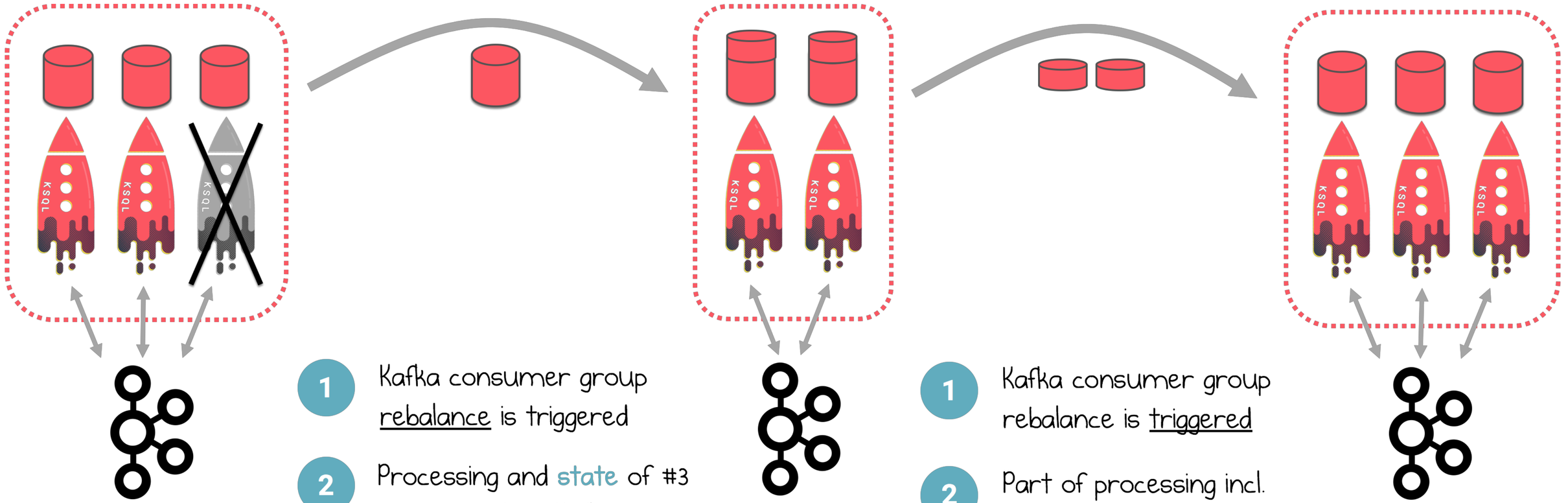


1. Kafka consumer group rebalance is triggered
2. Processing and state of #3 is migrated via Kafka to remaining servers #1 + #2

Fault-Tolerance, powered by Kafka

#3 died so #1 and #2 take over

#3 is back so the work is split again

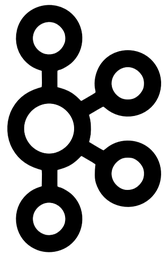


1. Kafka consumer group rebalance is triggered
2. Processing and state of #3 is migrated via Kafka to remaining servers #1 + #2

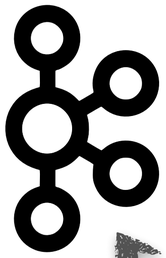
1. Kafka consumer group rebalance is triggered
2. Part of processing incl. state is migrated via Kafka from #1 + #2 to server #3

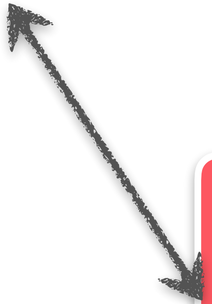
Standing on the shoulders of Streaming Giants

Standing on the shoulders of Streaming Giants



Standing on the shoulders of Streaming Giants

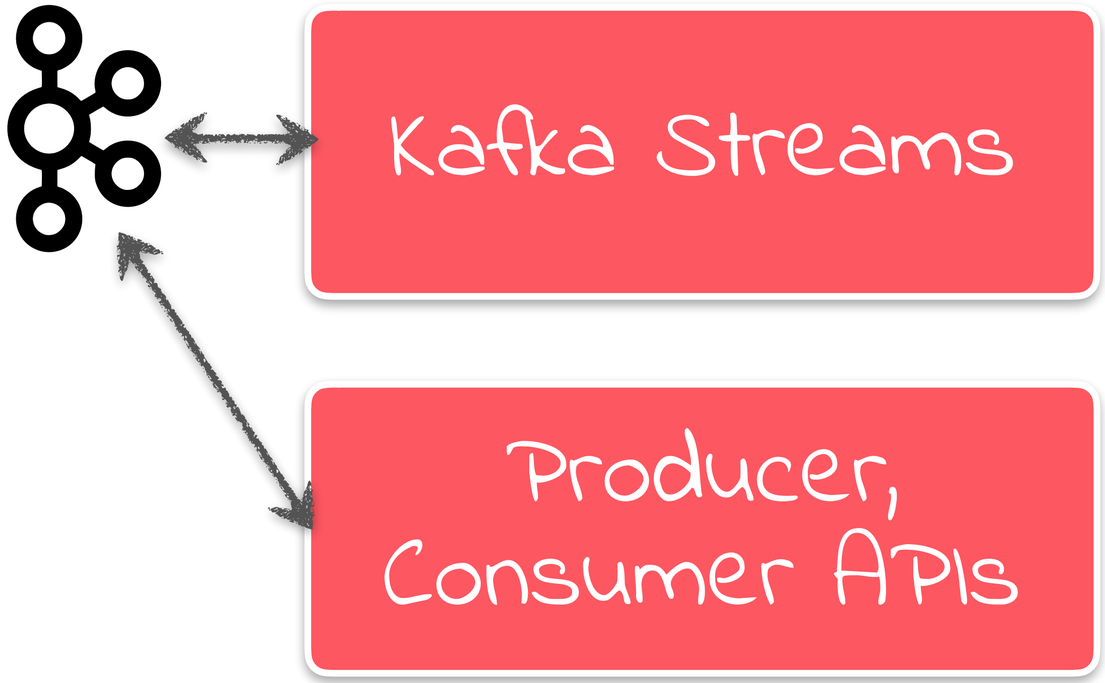




Producer,
Consumer APIs

```
subscribe(), poll(), send(),
flush(), beginTransaction(), …
```

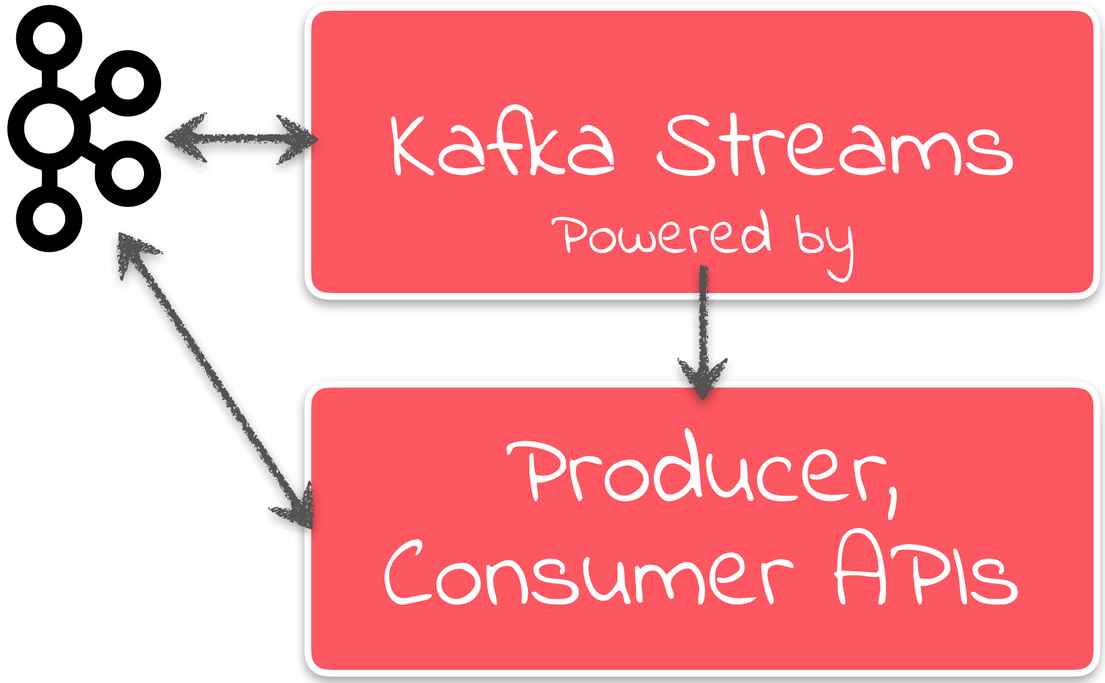
Standing on the shoulders of Streaming Giants




```
KStream<>, KTable<>, filter(), map(),
flatMap(), join(), aggregate(),
transform(), …
```

```
subscribe(), poll(), send(),
flush(), beginTransaction(), …
```

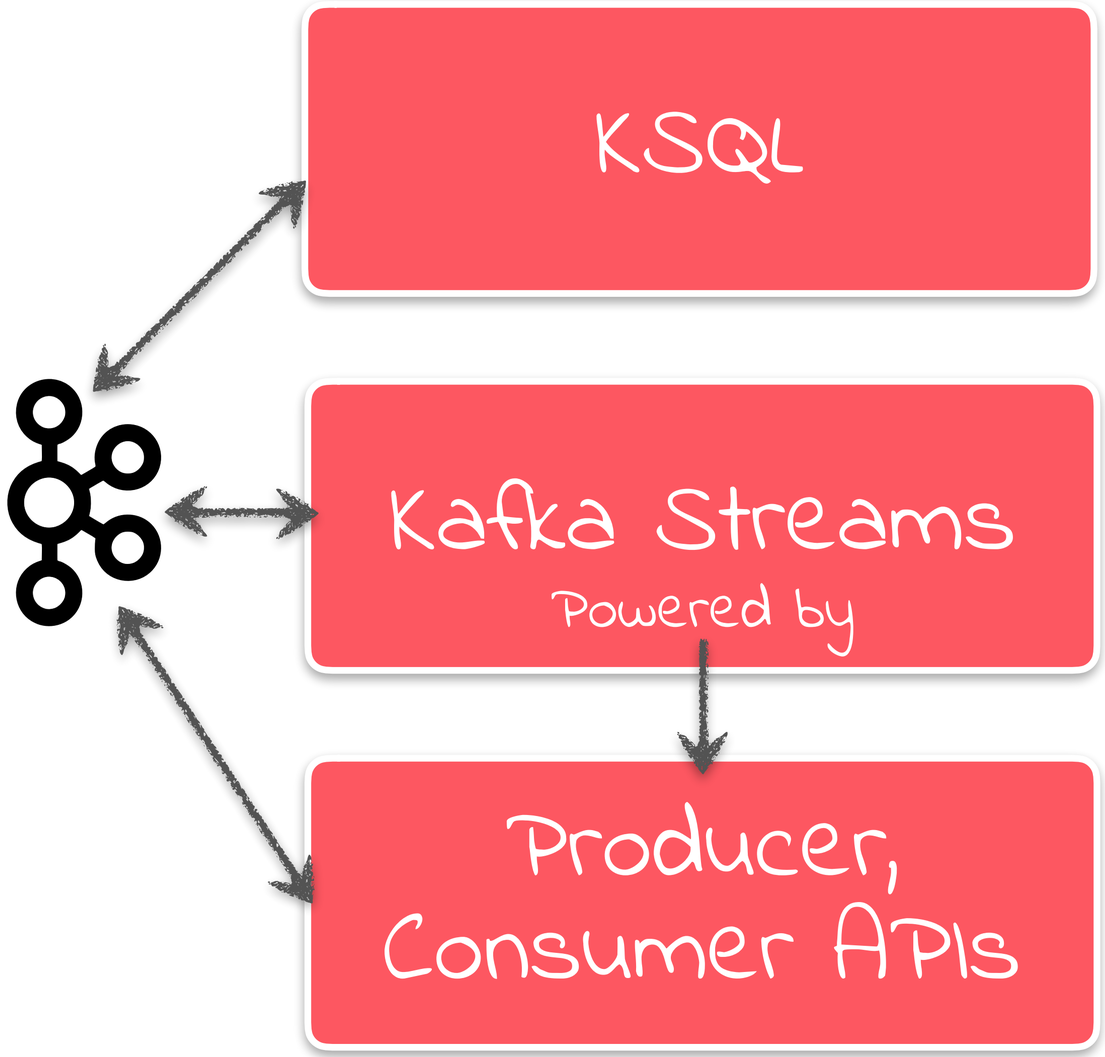
Standing on the shoulders of Streaming Giants




```
KStream<>, KTable<>, filter(), map(),
flatMap(), join(), aggregate(),
transform(), …
```

```
subscribe(), poll(), send(),
flush(), beginTransaction(), …
```

Standing on the shoulders of Streaming Giants



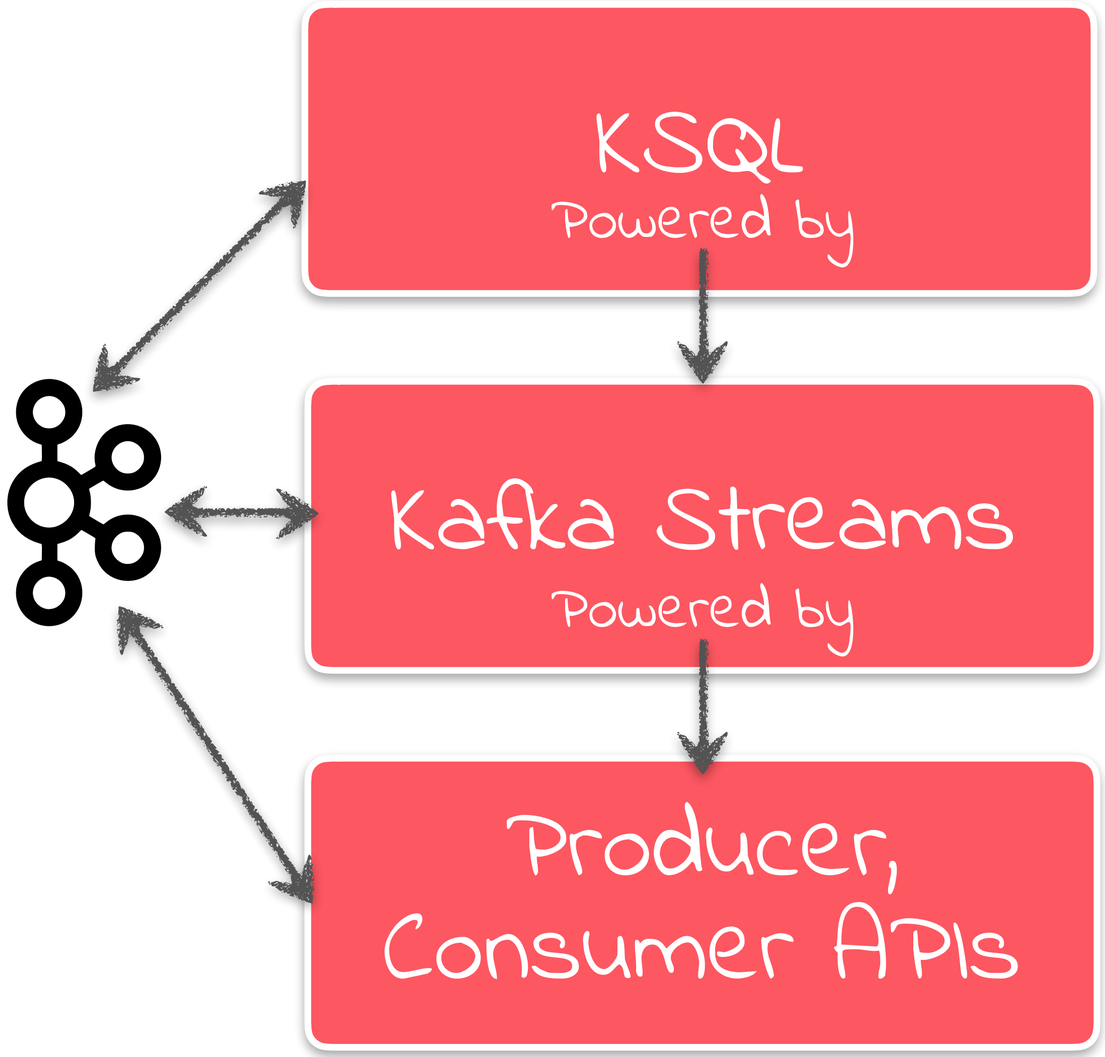

```
CREATE STREAM,
CREATE TABLE, SELECT, JOIN,
GROUP BY, SUM, …
```

KSQL UDFs

```
KStream<>, KTable<>, filter(), map(),
flatMap(), join(), aggregate(),
transform(), …
```

```
subscribe(), poll(), send(),
flush(), beginTransaction(), …
```

Standing on the shoulders of Streaming Giants



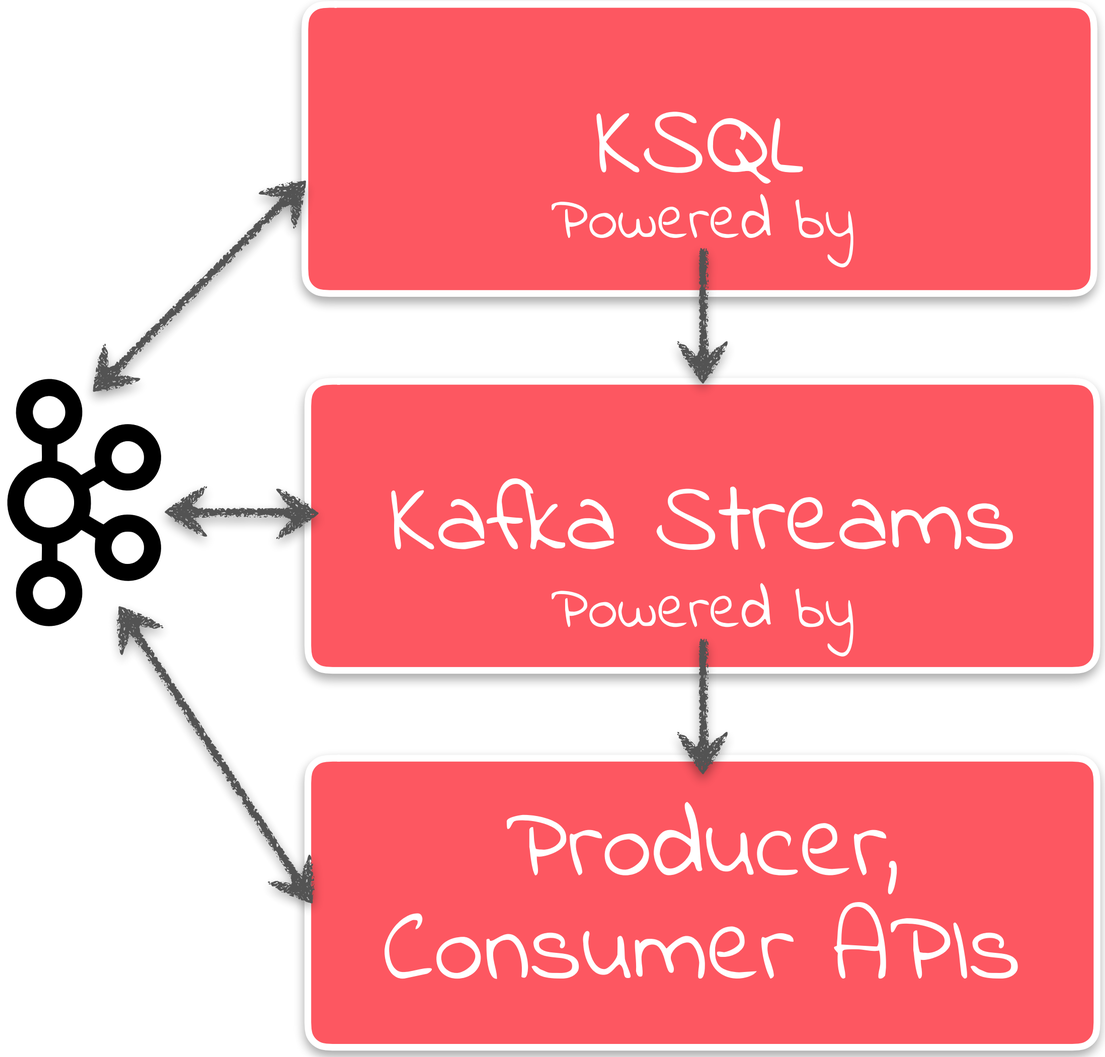

```
CREATE STREAM,
CREATE TABLE, SELECT, JOIN,
GROUP BY, SUM, …
```

KSQL UDFs

```
KStream<>, KTable<>, filter(), map(),
flatMap(), join(), aggregate(),
transform(), …
```

```
subscribe(), poll(), send(),
flush(), beginTransaction(), …
```

Standing on the shoulders of Streaming Giants



Ease of use

Flexibility

```
CREATE STREAM,
CREATE TABLE, SELECT, JOIN,
GROUP BY, SUM, …
```

KSQL UDFs

```
KStream<>, KTable<>, filter(), map(),
flatMap(), join(), aggregate(),
transform(), …
```

```
subscribe(), poll(), send(),
flush(), beginTransaction(), …
```

confluent

One last thing...


kafka summit
https://kafka-summit.org
Gamov30
APRIL 2, 2019
NEW YORK CITY
REGISTER
LEARN MORE

THANKS!

@gamussa
viktor@confluent.io

We are hiring!
https://www.confluent.io/careers/