CROSSING THE STREAMS: RETHINKING STREAM PROCESSING WITH KAFKA STREAMS AND KSQL

DEVNEXUS 2019 | @TLBERGLUND | @GAMUSSA

confluent


LIFE
ocean

confluent

HTTPS://CNFL.IO/STREAMS-MOVIE-DEMO

confluent

RAFFLE, YEAH 🚀

- Follow @gamussa @tlberglund
- 📸🖼️👬
- Tag @gamussa @tlberglund
- With #devnexus

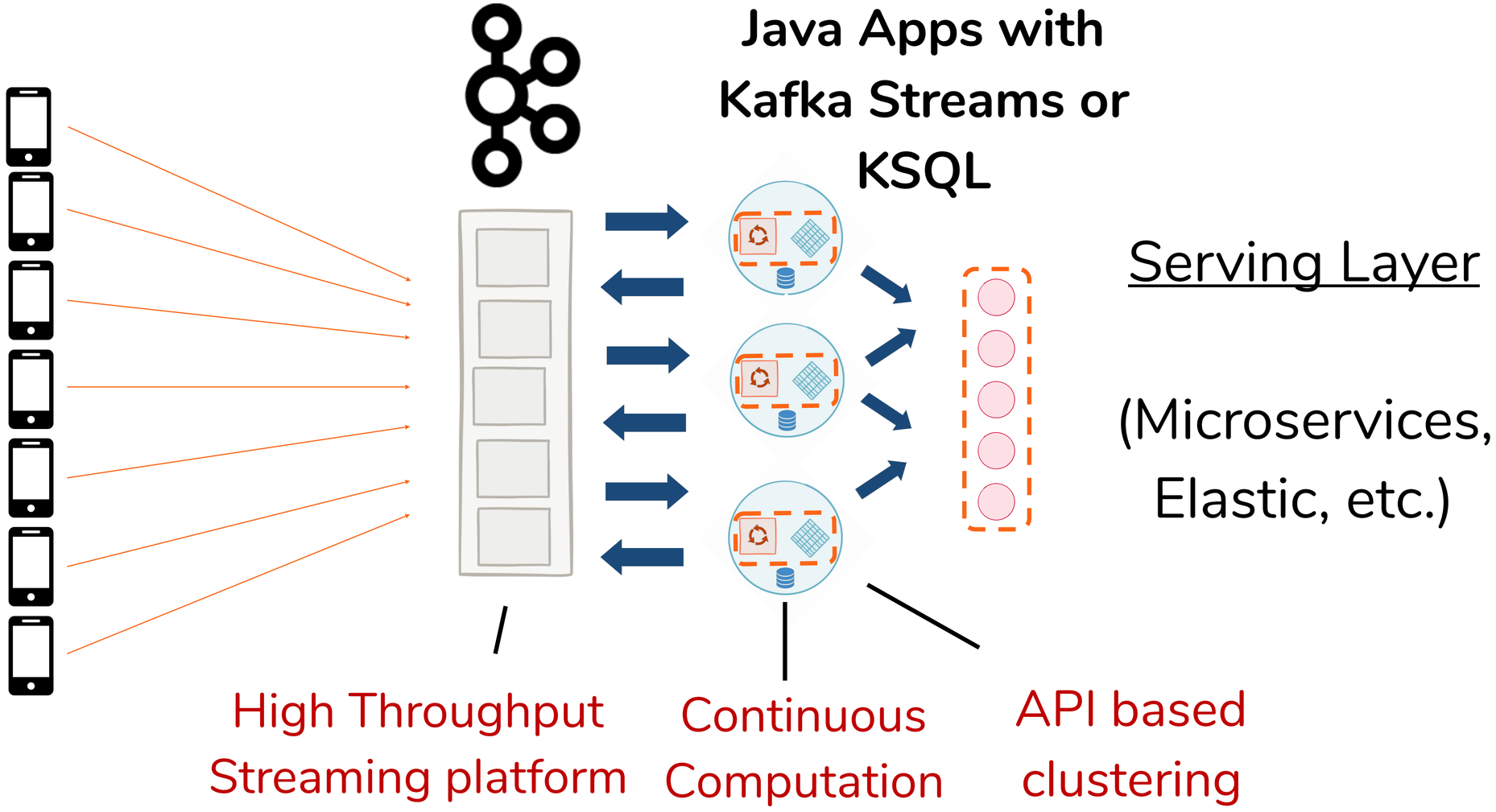
confluent

PREFACE

Streaming

is the toolset for dealing with events

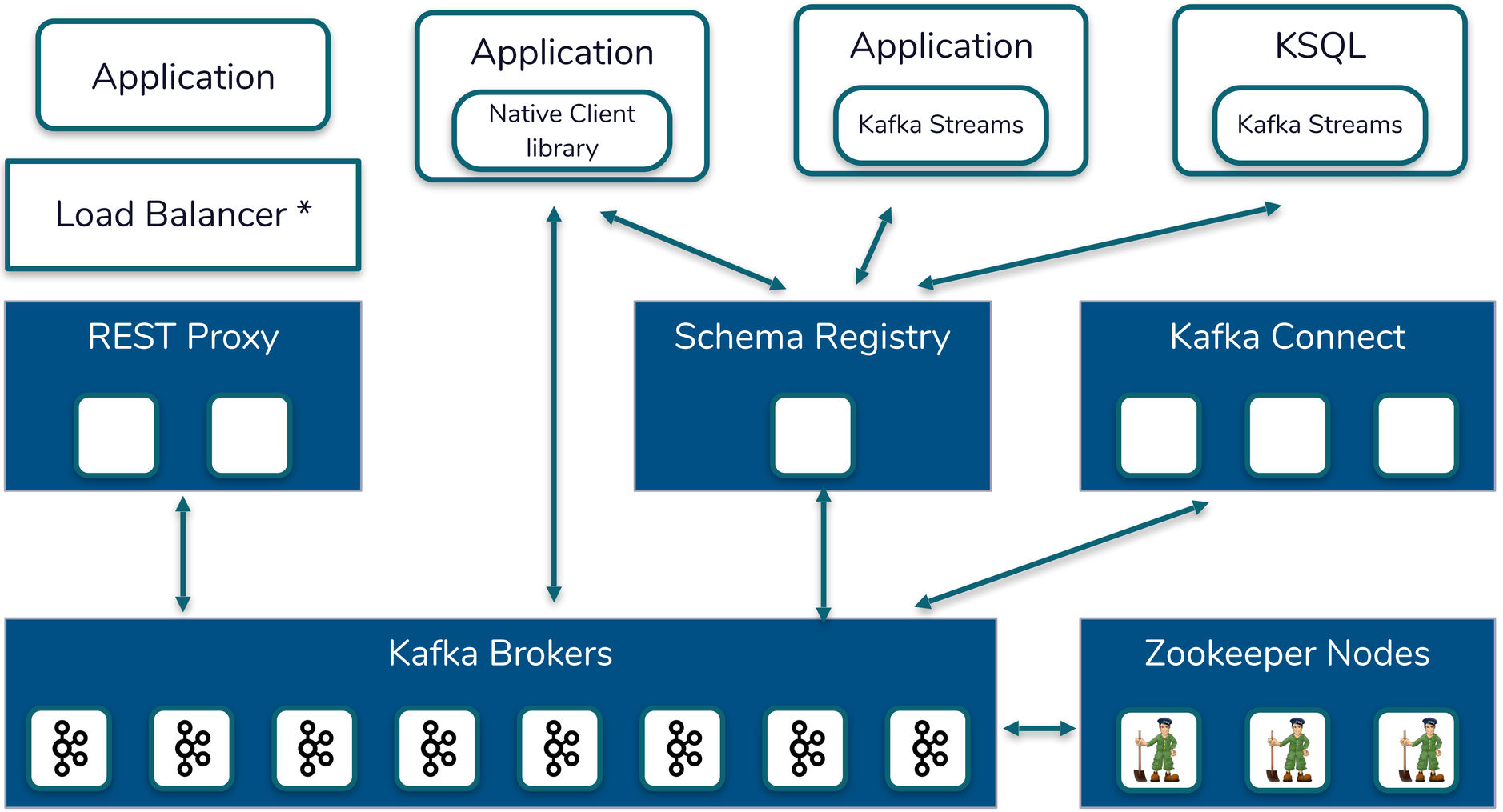
as they move!


Java Apps with
Kafka Streams or
KSQL
Serving Layer
(Microservices,
Elastic, etc.)
High Throughput
Streaming platform
Continuous
Computation
API based
clustering

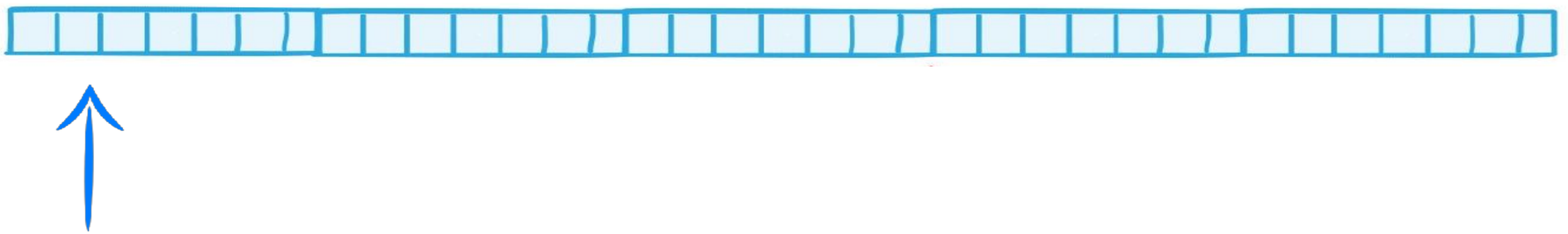
confluent

APACHE KAFKA EVENT STREAMING PLATFORM 101

EVENT STREAMING PLATFORM ARCHITECTURE

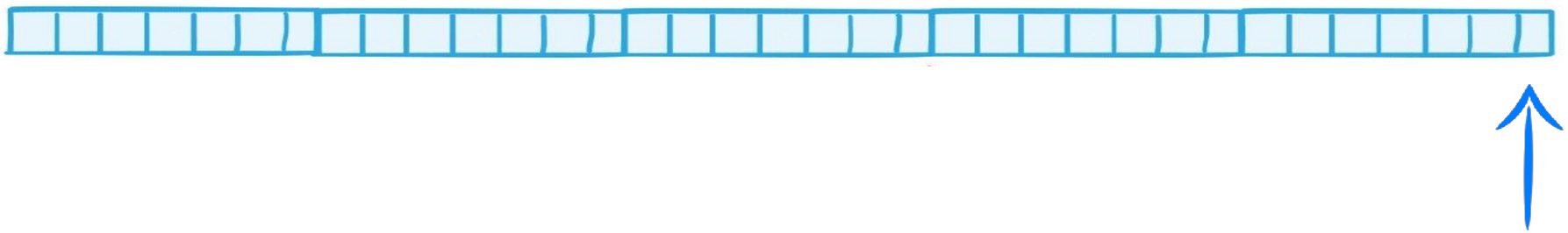


THE LOG IS A SIMPLE IDEA

Old  New

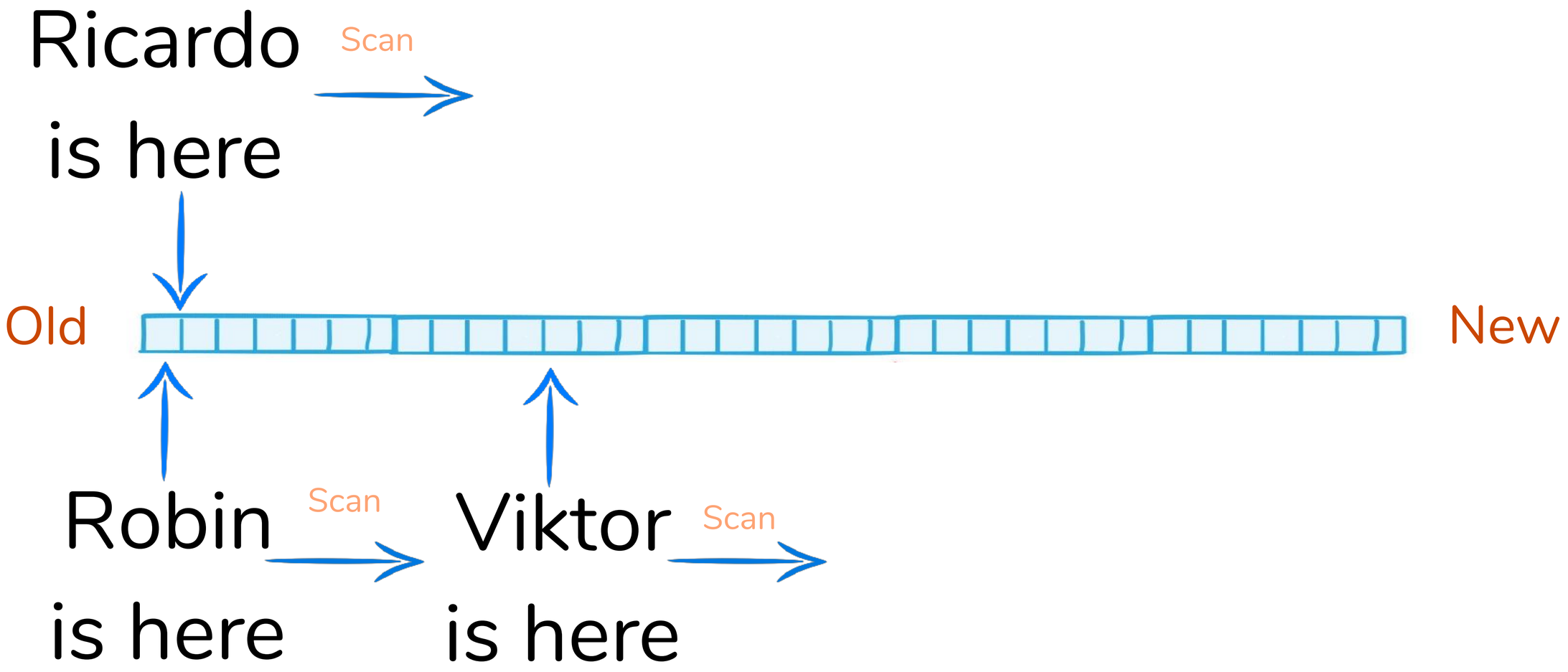
Messages are added at the end of the log

THE LOG IS A SIMPLE IDEA

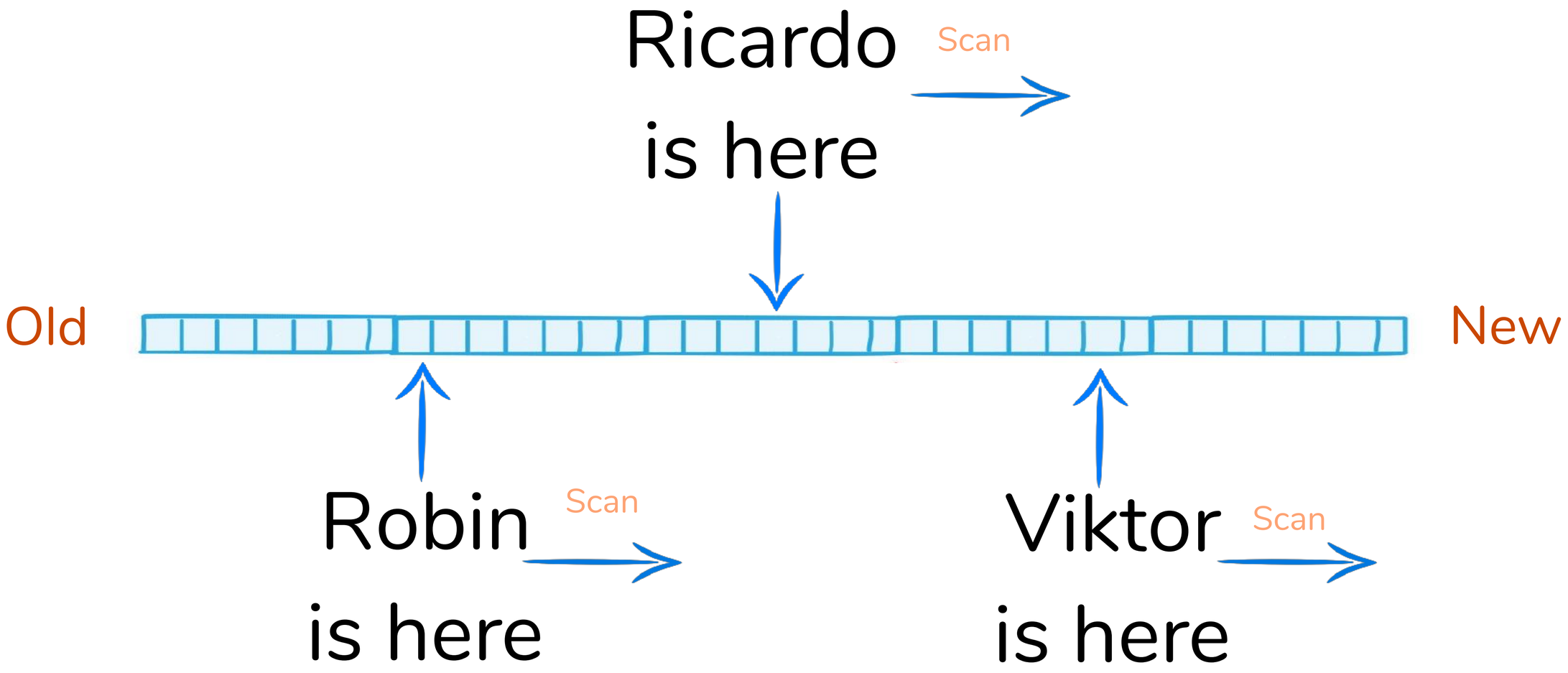
Old  New

Messages are added at the end of the log

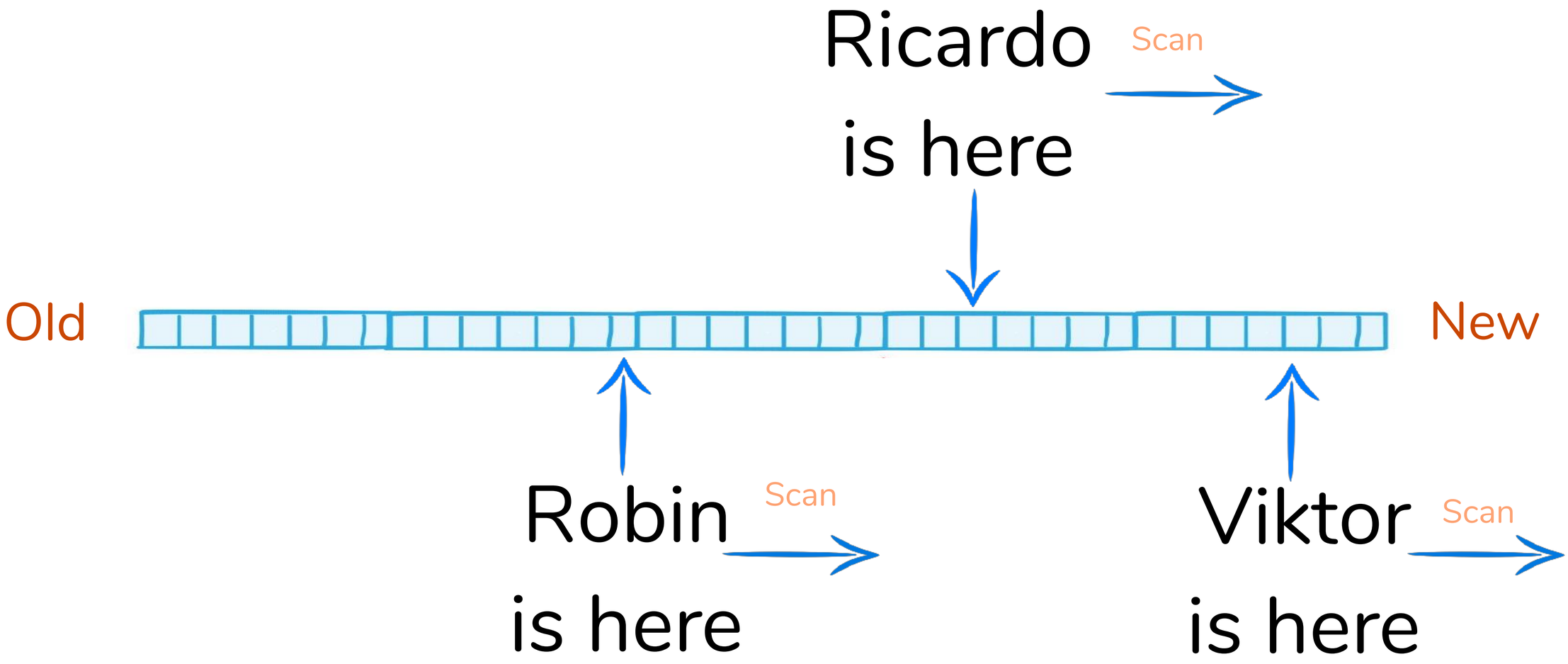
CONSUMERS HAVE A POSITION ALL OF THEIR OWN



CONSUMERS HAVE A POSITION ALL OF THEIR OWN



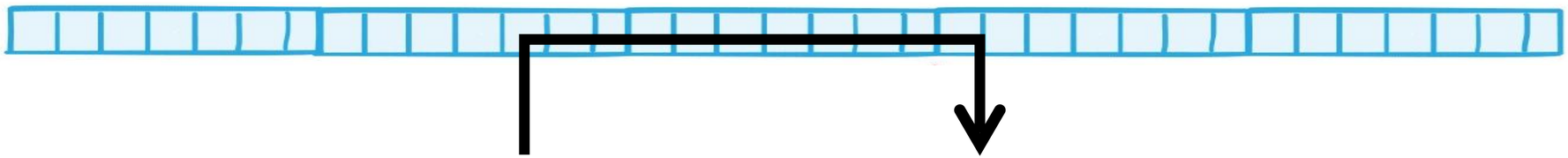
CONSUMERS HAVE A POSITION ALL OF THEIR OWN



ONLY SEQUENTIAL ACCESS

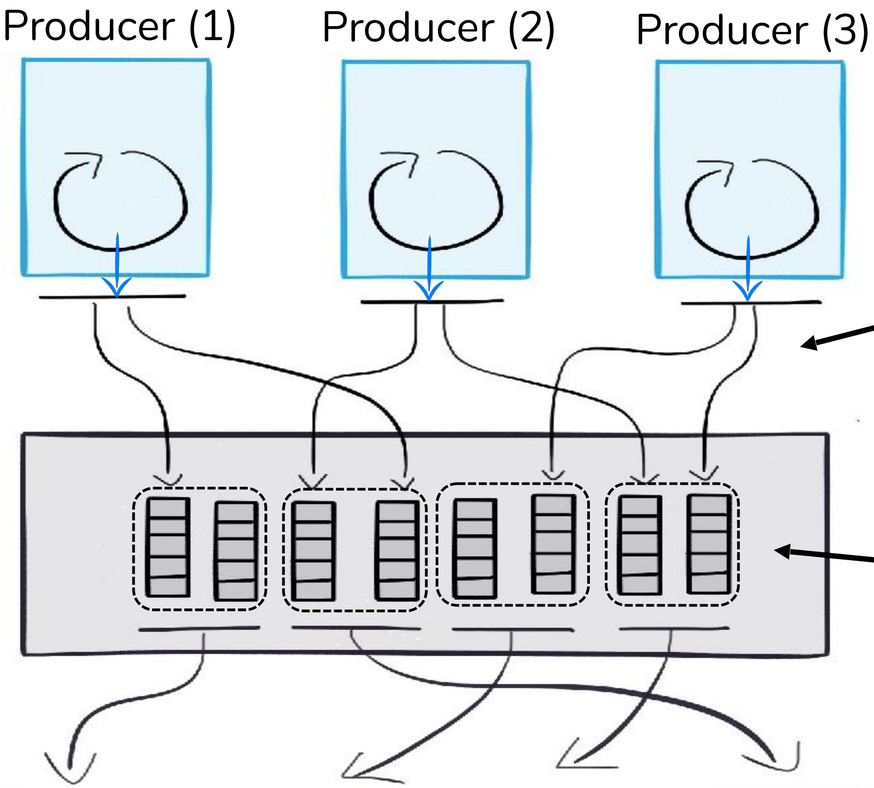
Read to offset & scan

Old



New

SHARD DATA TO GET SCALABILITY



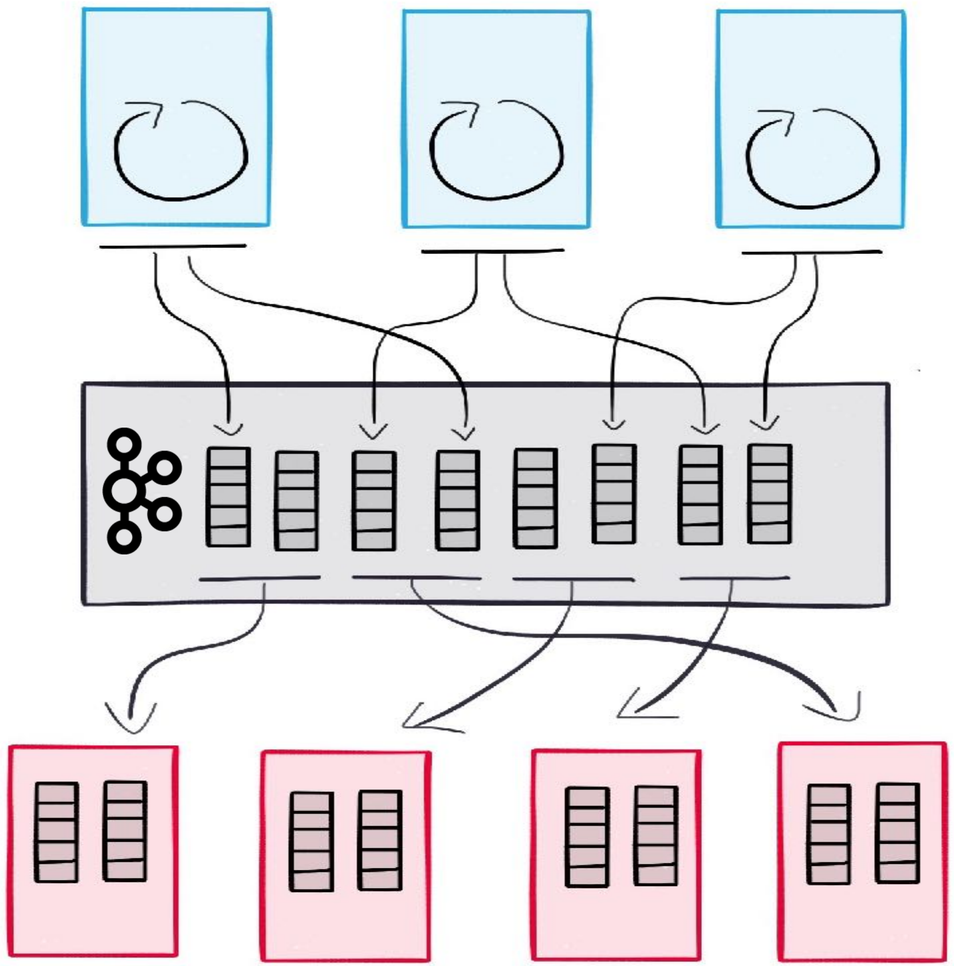
Messages are sent to different partitions

Partitions live on different machines


CONSUMERS
CONSUMER GROUP COORDINATOR
CONSUMER GROUP

LINEARLY SCALABLE ARCHITECTURE

Producers



Consumers

Single topic:

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

No Bottleneck!!

```
// 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();
    }
  }
}
```

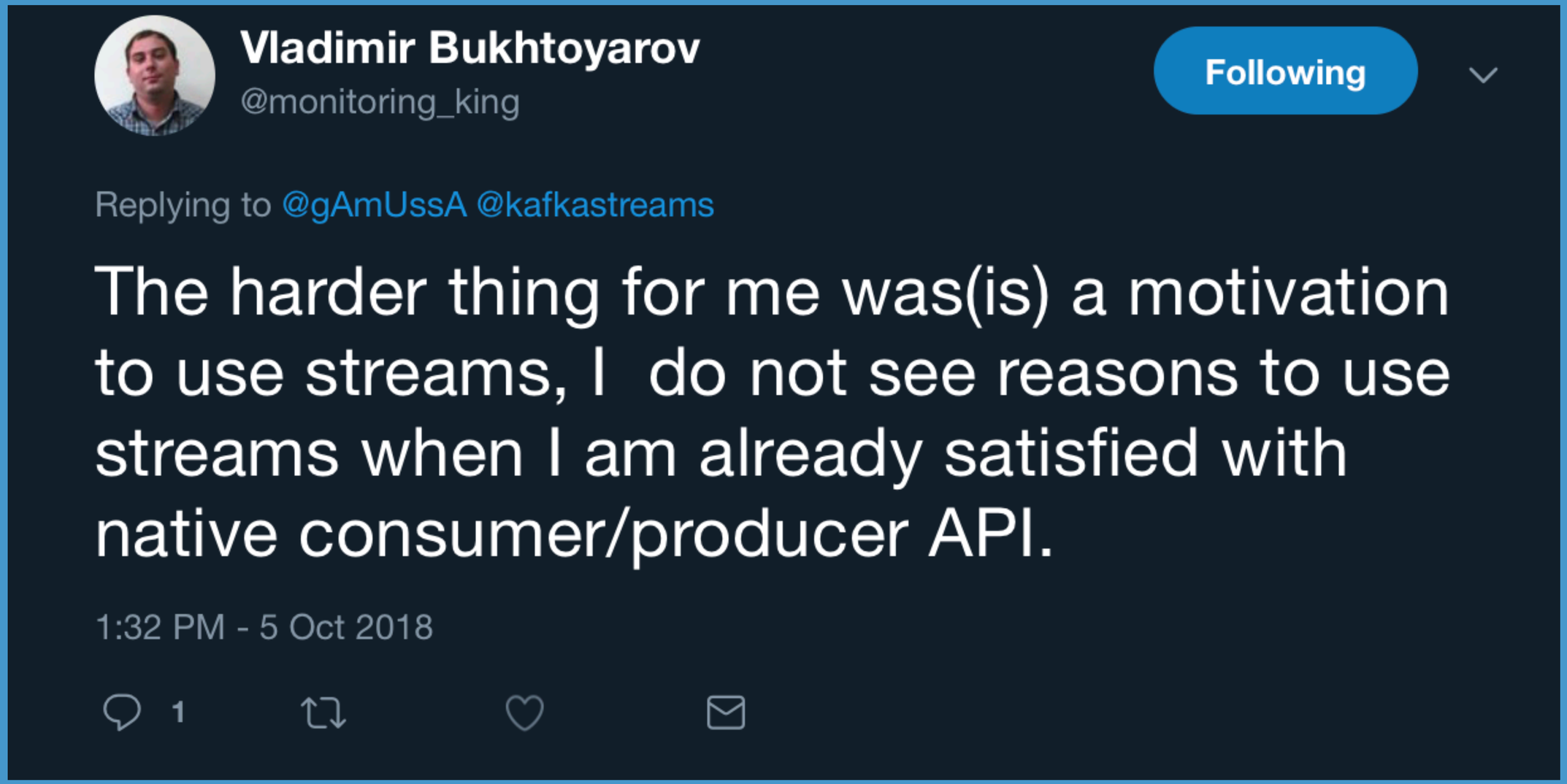
```
    if (counter++ % sendInterval == 0) {
      for (Map.Entry<String, Integer> groupedEntry : groupByCounts.entrySet()) {

        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();
    }
  }
}
```

confluent



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


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

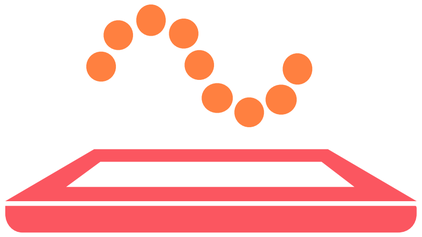

SO LET ME SHOW KAFKA STREAMS
THAT WAY IT MIGHT BE REALLY GOOD

confluent

TALK IS CHEAP!
SHOW ME CODE!

EVERY FRAMEWORK WANTS TO BE WHEN IT GROWS UP

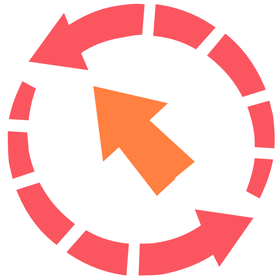
Scalable



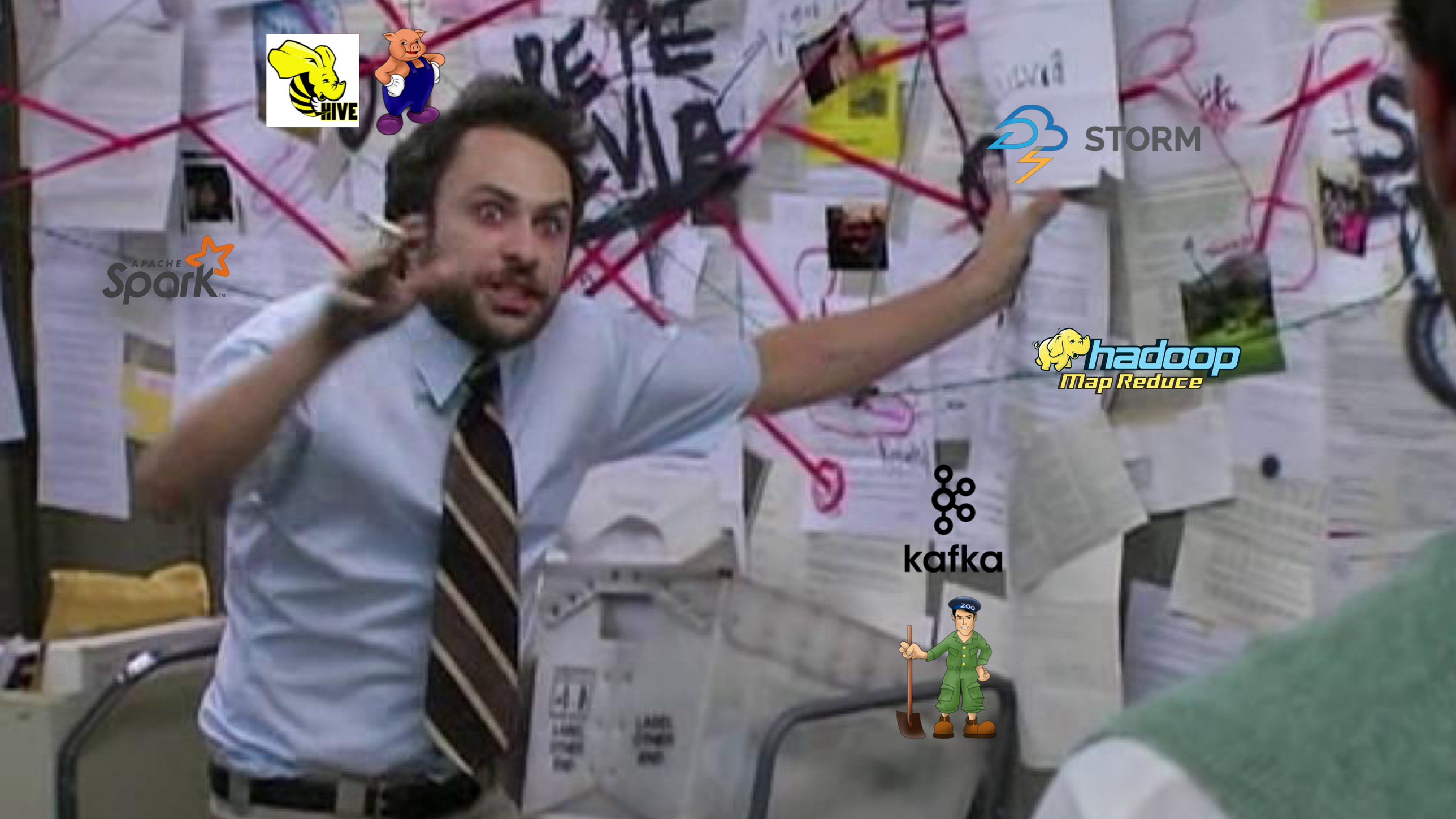
Elastic

Fault-tolerant

Stateful



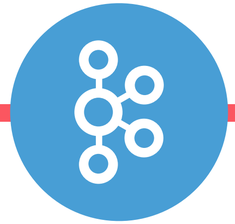
Distributed

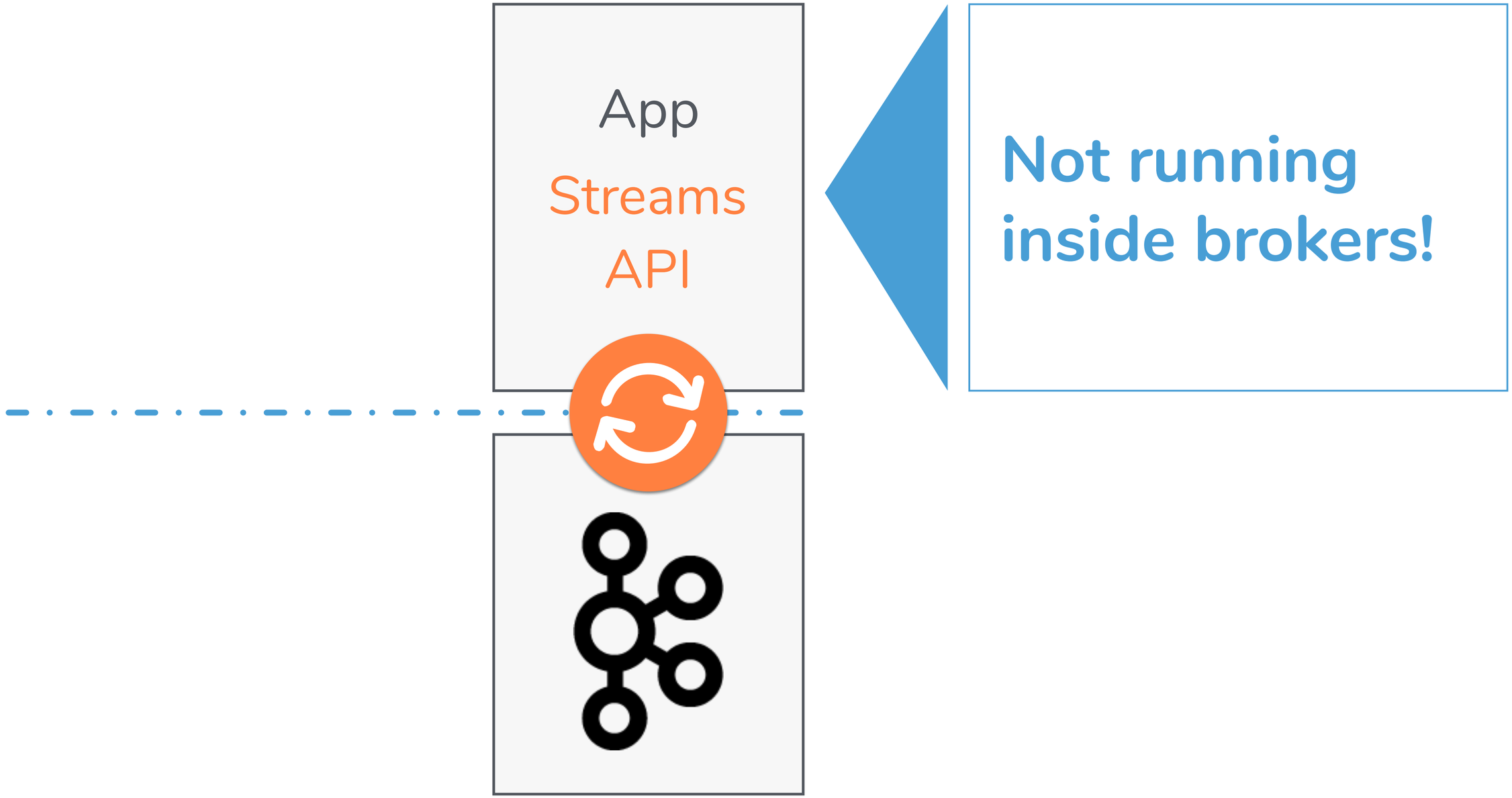

HIVE
STORM
APACHE
Spark
hadoop
Map Reduce
kafka
ZOO

the KAFKA STREAMS **API** is a

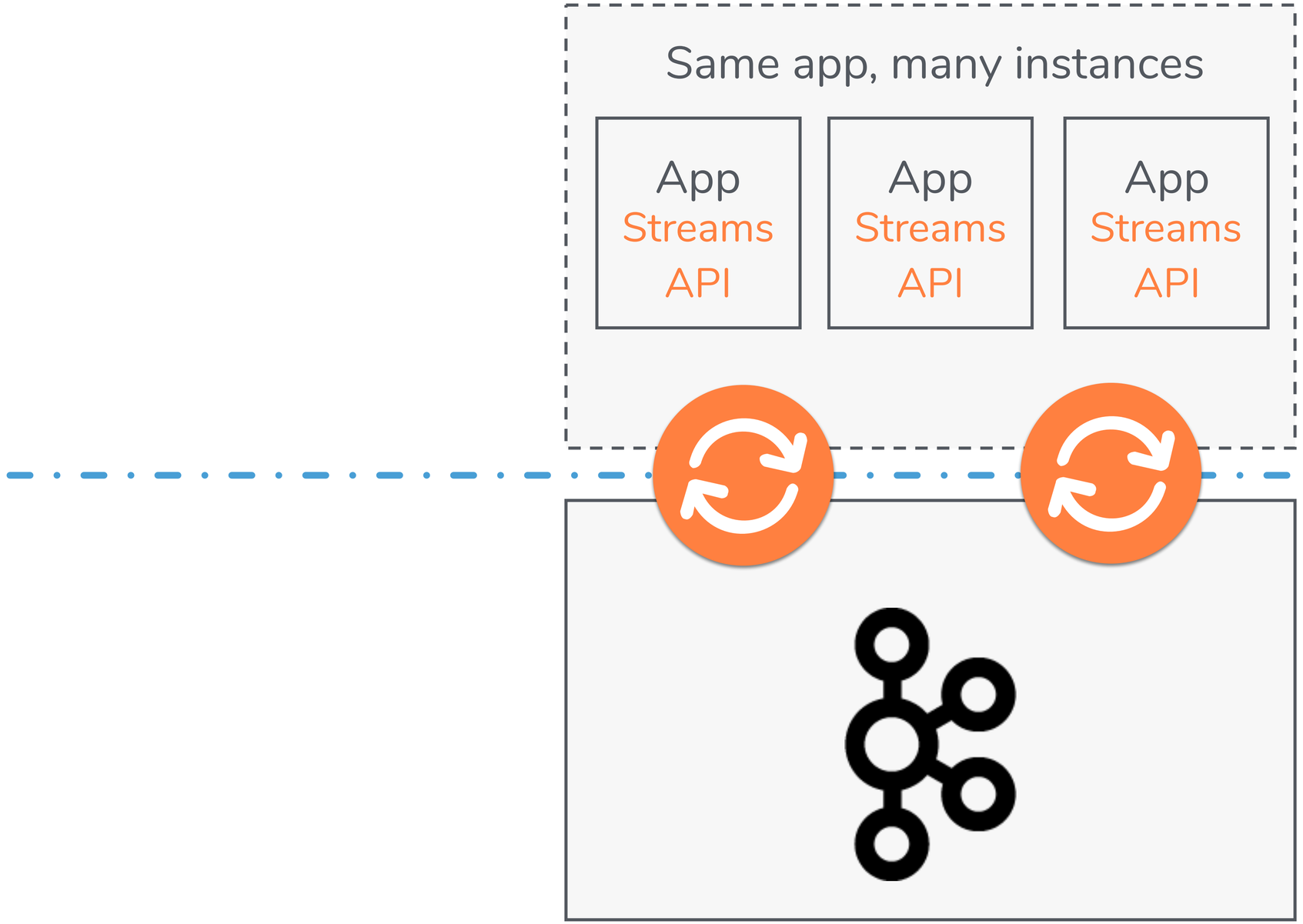
JAVA API to

BUILD REAL-TIME **APPLICATIONS**





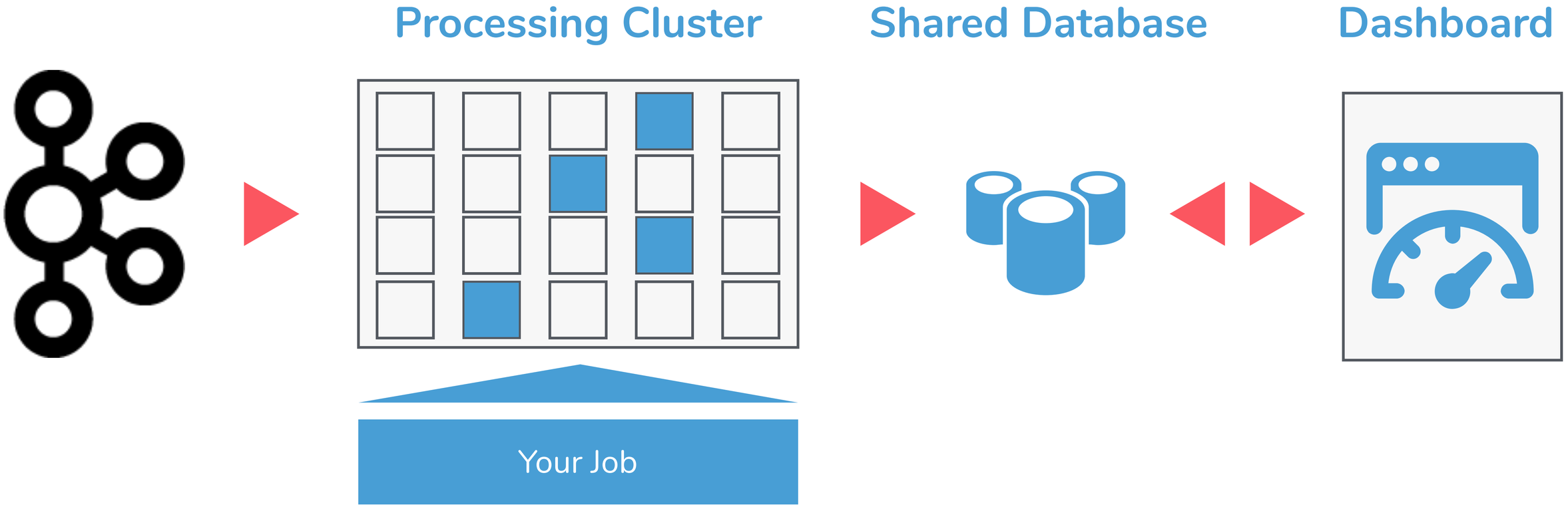
Not running inside brokers!



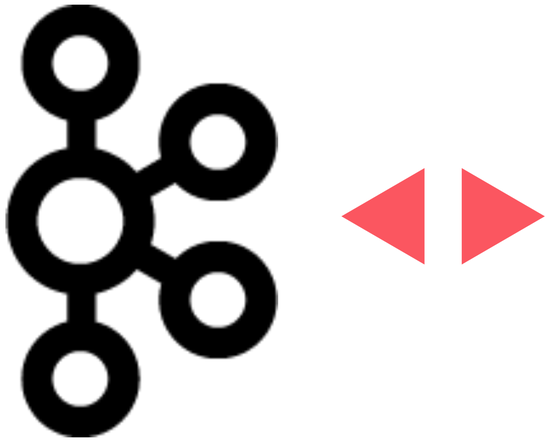
Brokers?

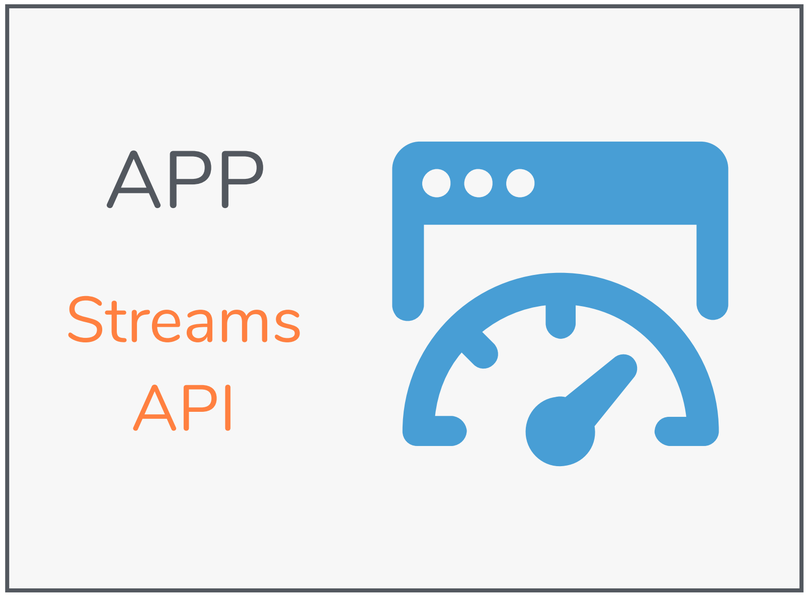
Nope!

BEFORE



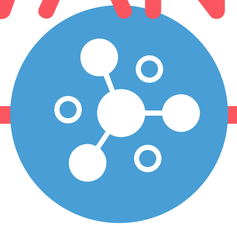
AFTER





this means you can

DEPLOY your app ANYWHERE using WHATEVER TECHNOLOGY YOU WANT



SO MANY PLACES TO RUN YOU APP!

amazon web services

Google Cloud Platform

Microsoft Azure

docker

kubernetes

Physical

openstack.

MESOS

vmware®

VAGRANT

TERRAFORM

ANSIBLE

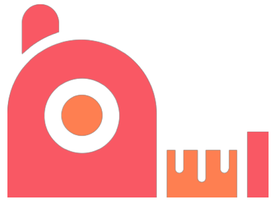
puppet labs

Jenkins

...and many more...

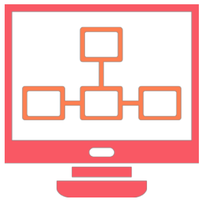
THINGS KAFKA STREAM DOES

Enterprise Support



Open Source

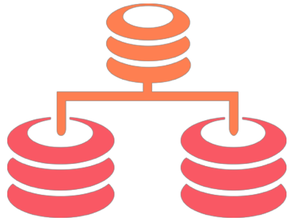
Runs Everywhere



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

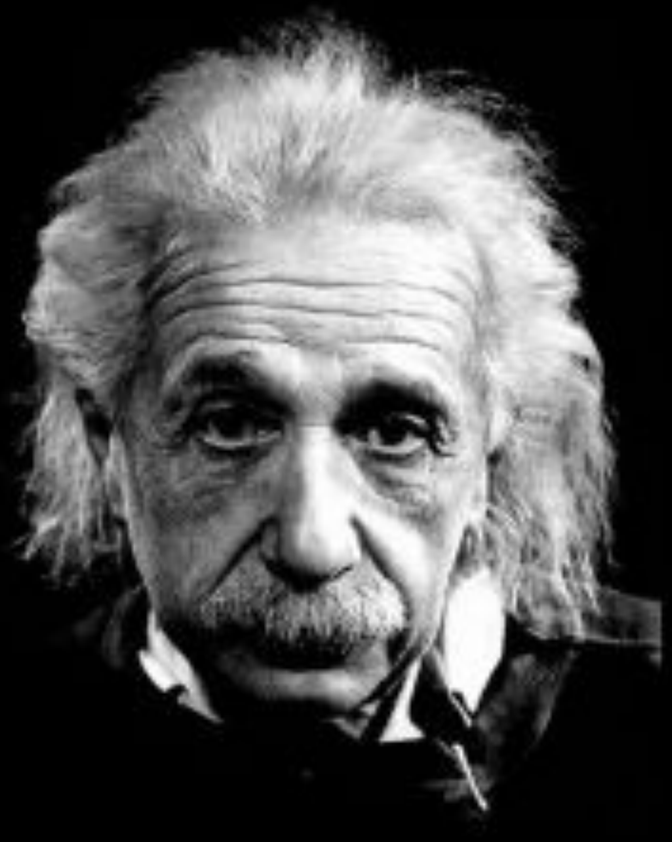
confluent

TALK IS CHEAP!
SHOW ME CODE!

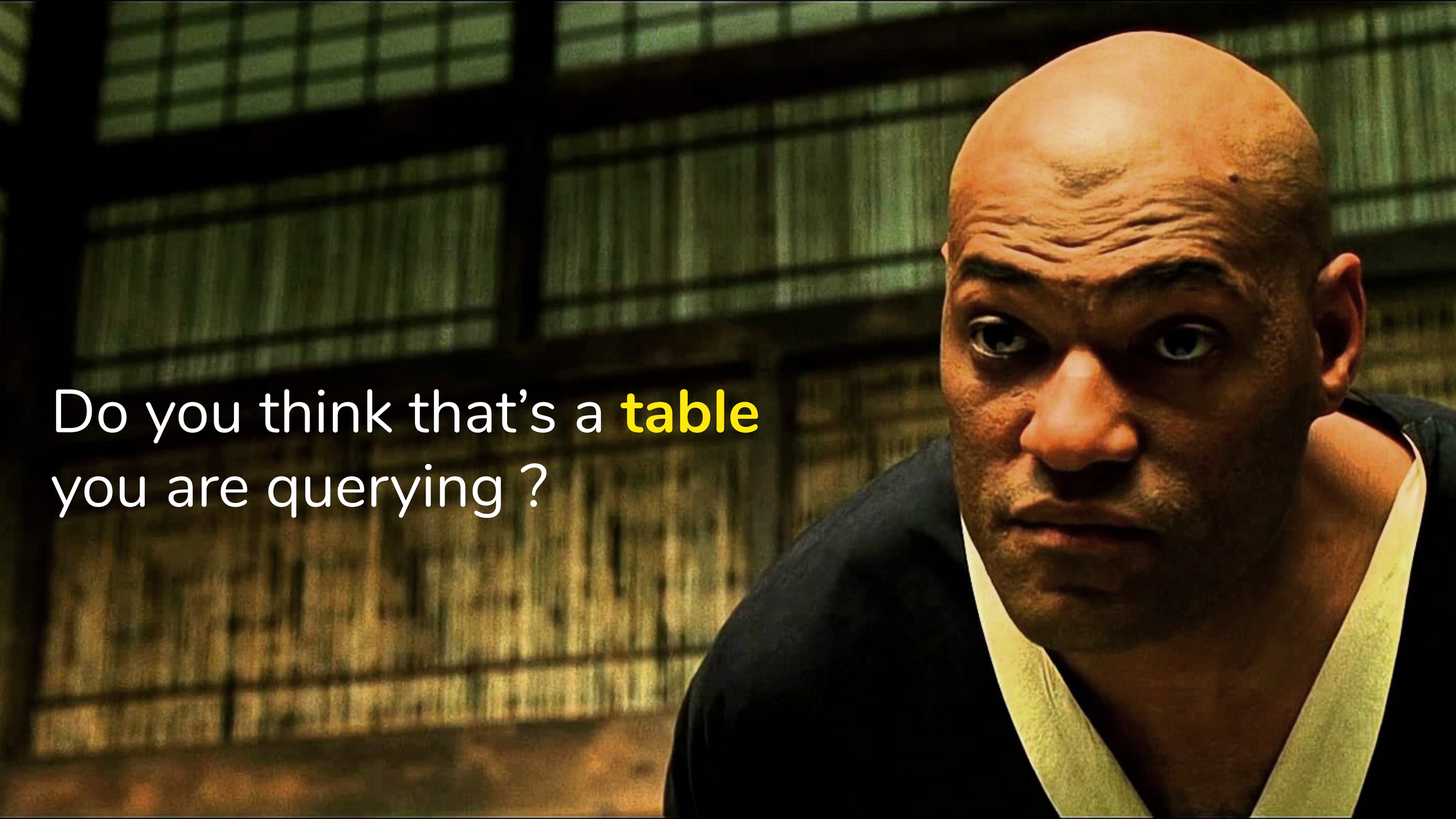
confluent

TALK IS CHEAP!
SHOW ME CODE!

SPECIAL RELATIVITY

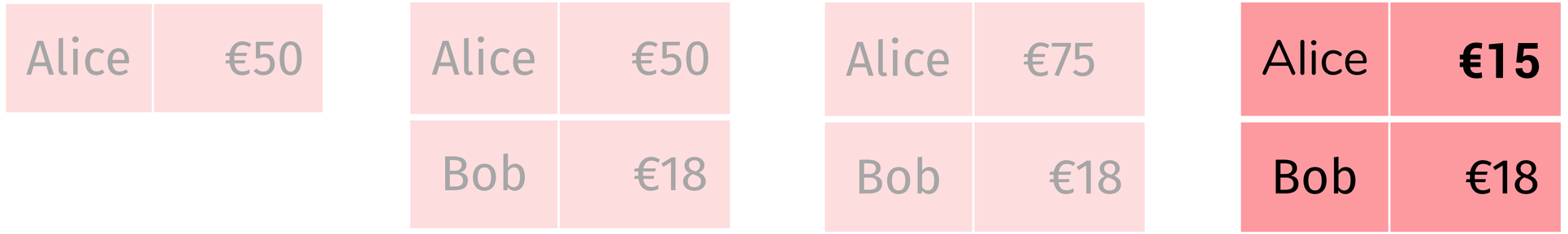


OF STREAMS AND TABLES

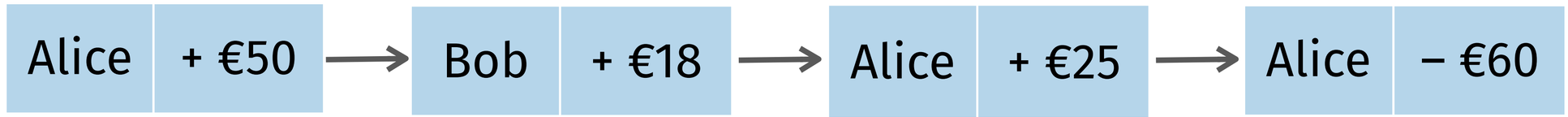

Do you think that’s a table
you are querying ?

THE STREAM-TABLE DUALITY

Table
(balance)



Stream
(payments)



confluent

TALK IS CHEAP!
SHOW ME CODE!

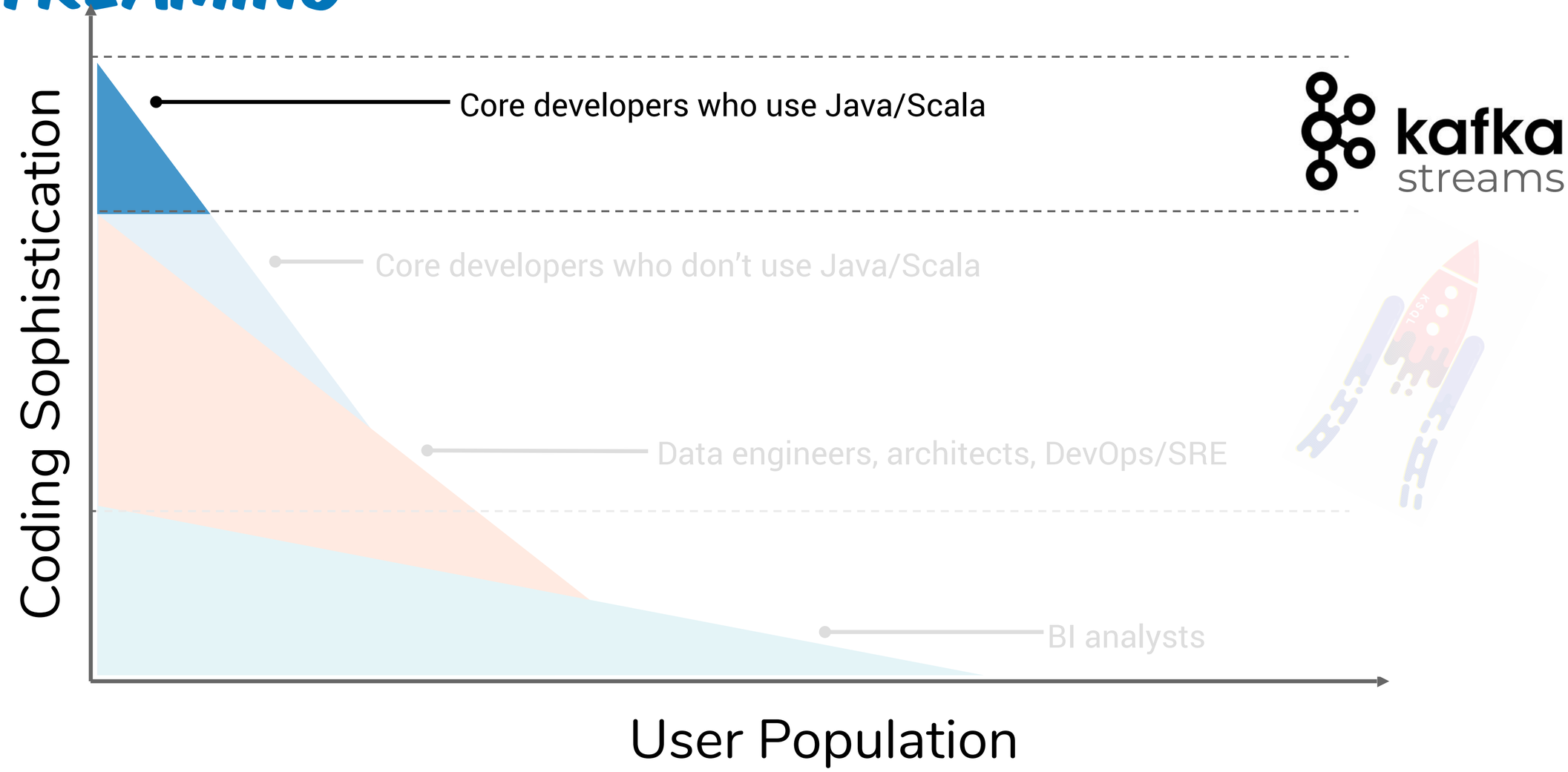
confluent

WHAT'S NEXT?

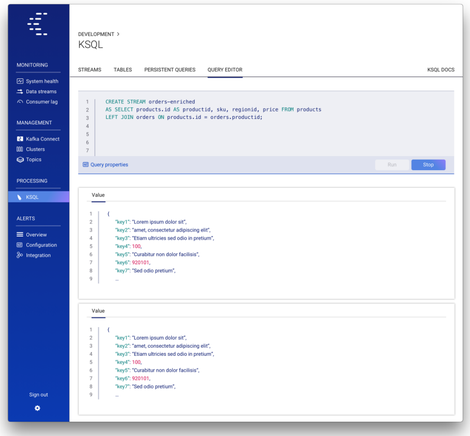
confluent

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

LOWER THE BAR TO ENTER THE WORLD OF STREAMING



KSQL #FTW



1 UI

2 CLI

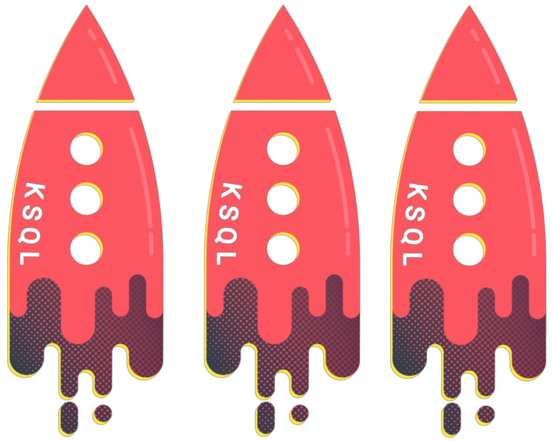
3 REST

4 Headless

INTERACTION WITH KAFKA

KSQL

(processing)

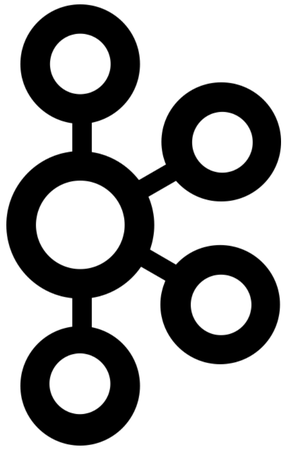


read, write

network

Kafka

(data)



read, write

network

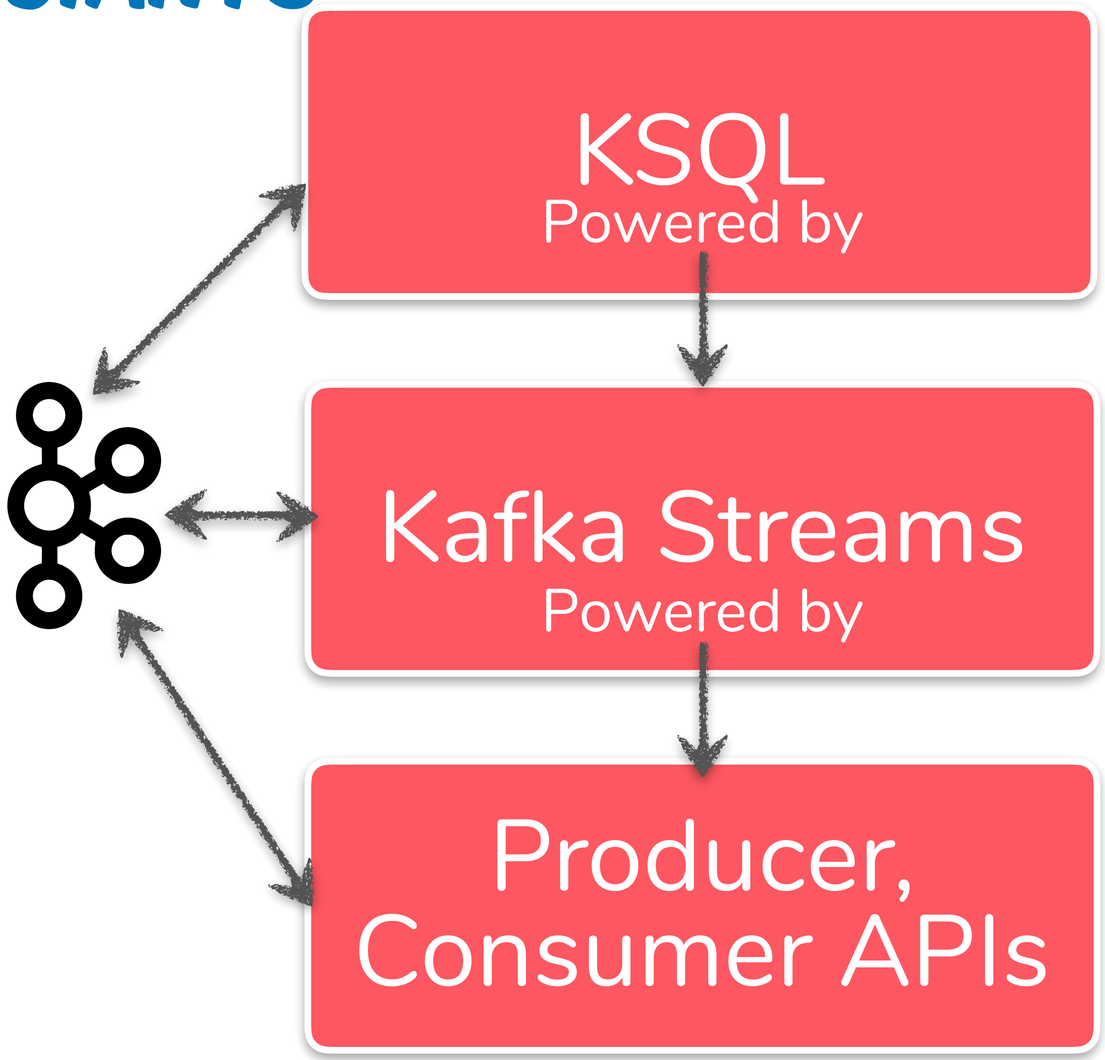
JVM application

with Kafka Streams (processing)

Does not run on Kafka brokers

Does not run on Kafka brokers

STANDING ON THE SHOULDERS OF STREAMING GIANTS



Ease of use

Flexibility

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

KSQL

```
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

@tlberglund
tim@confluent.io

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