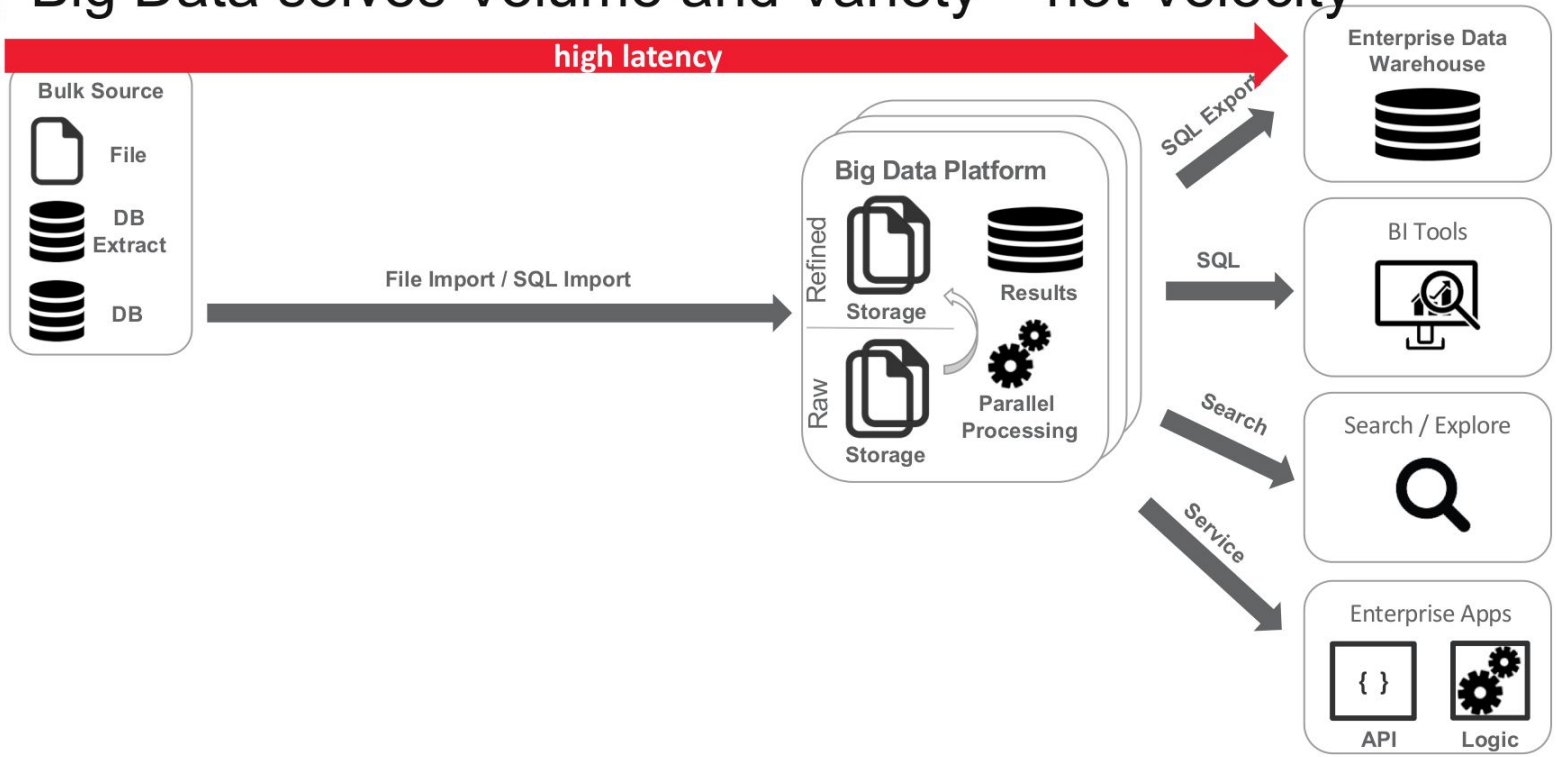
Data Engineering:
Introduction to Stream Processing

Traditional Data Processing - Challenges

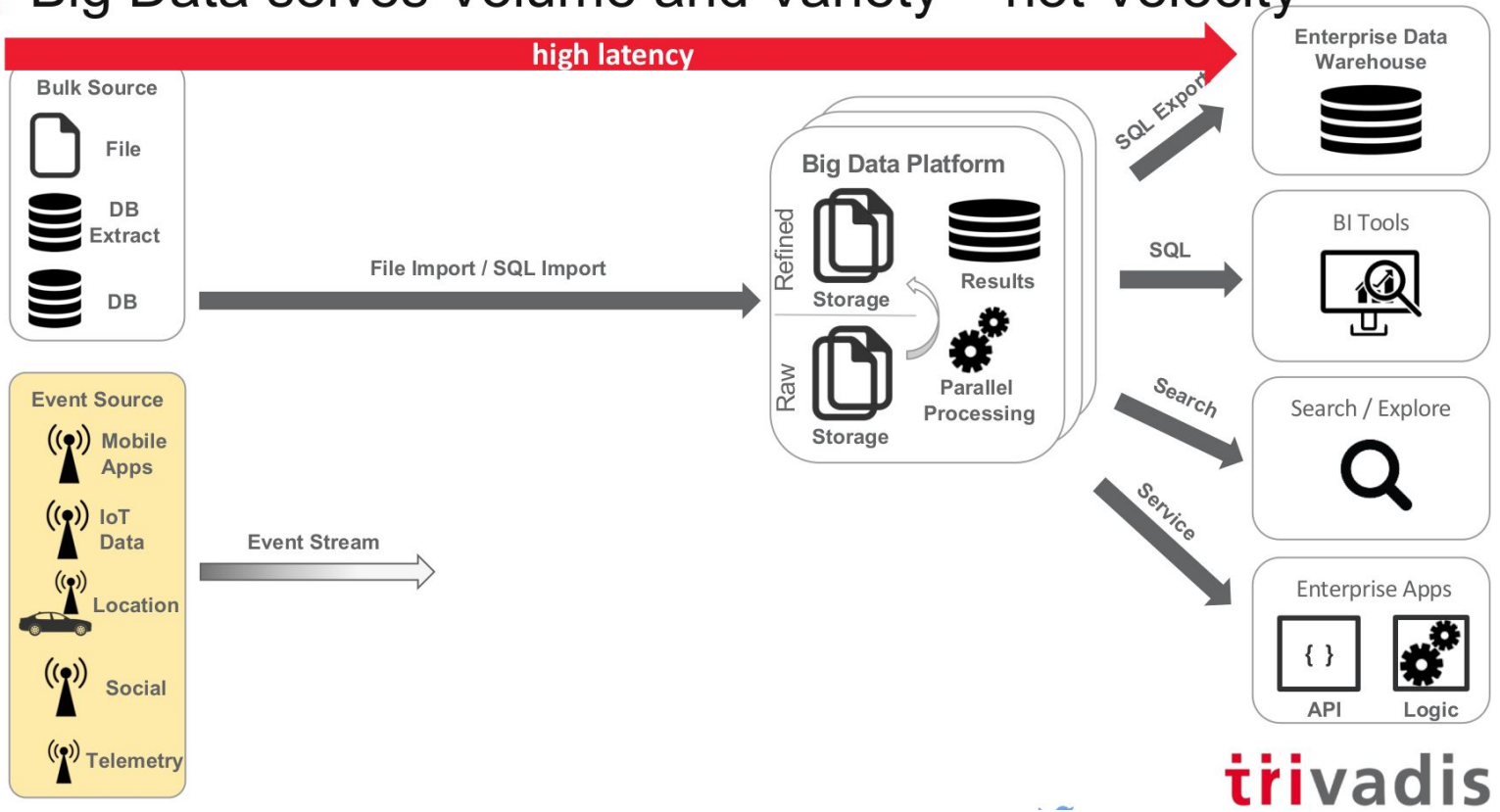
- Introduces too much “decision latency”
- Responses are delivered “after the fact”
- Maximum value of the identified situation is lost

- Decision are made on old and stale data
- “Data a Rest”

Big Data solves Volume and Variety – not Velocity



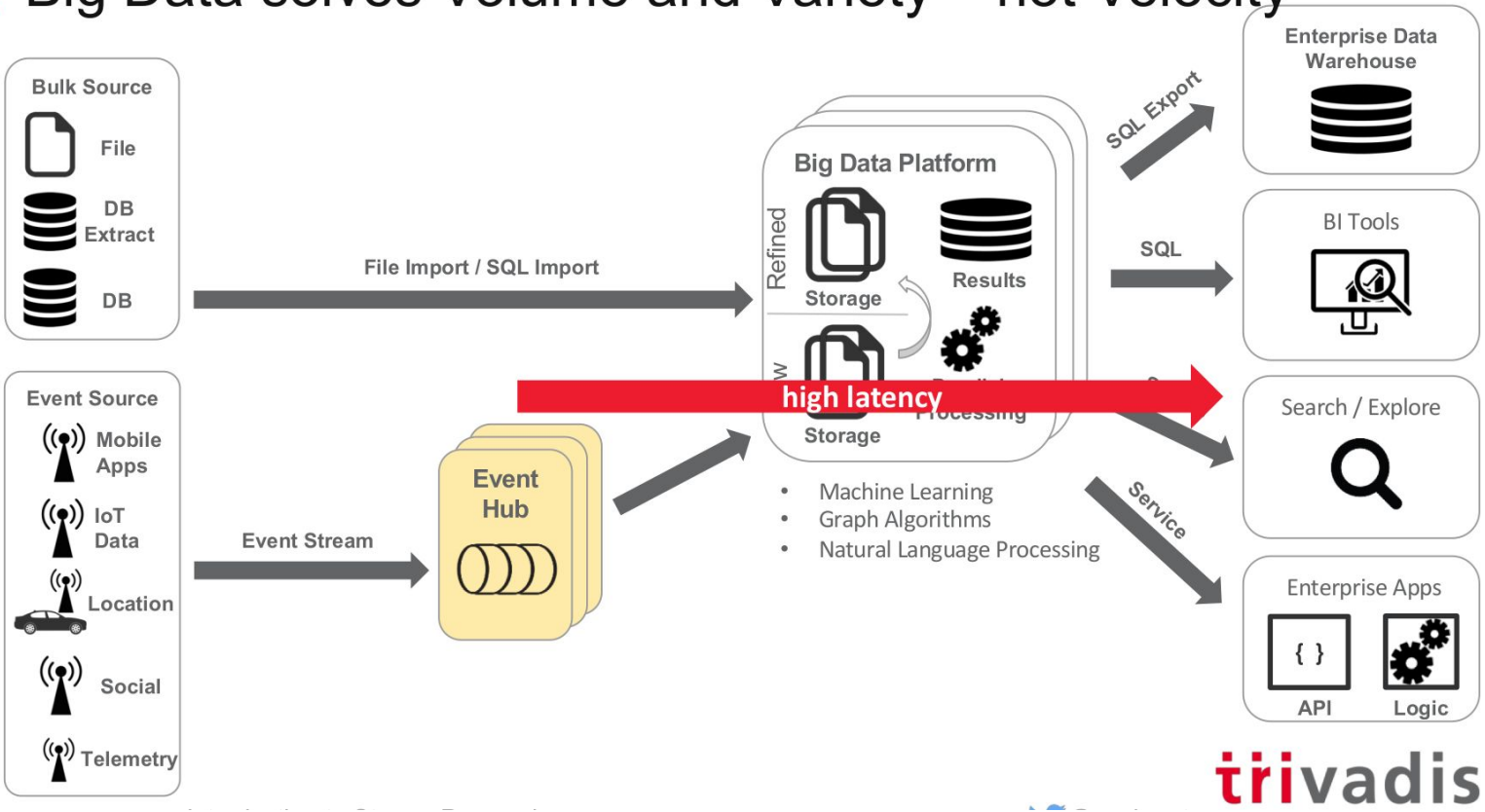
Big Data solves Volume and Variety – not Velocity



Real Time Analytics Use Cases

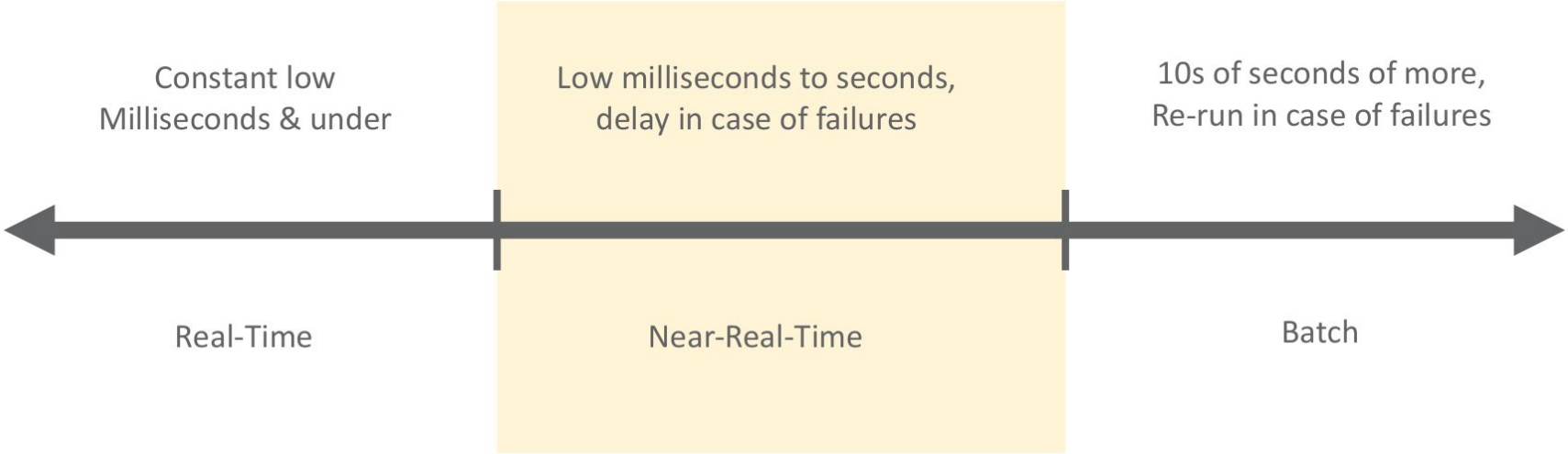
- Algorithmic Trading
- Online Fraud Detection
- Geo Fencing
- Proximity/Location Tracking
- Intrusion detection systems
- Traffic Management
- Recommendations
- Churn detection
- Internet of Things (IoT) / Intelligence Sensors
- Social Media/Data Analytics
- Gaming Data Feed
- …

Big Data solves Volume and Variety – not Velocity

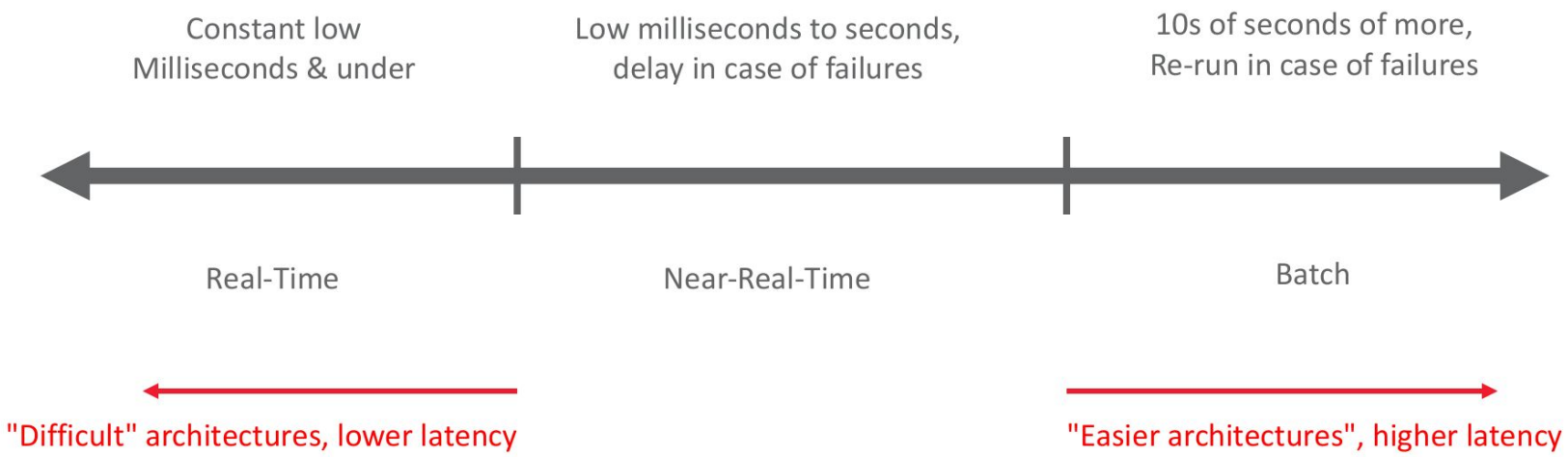


"Data at Rest" vs. "Data in Motion"

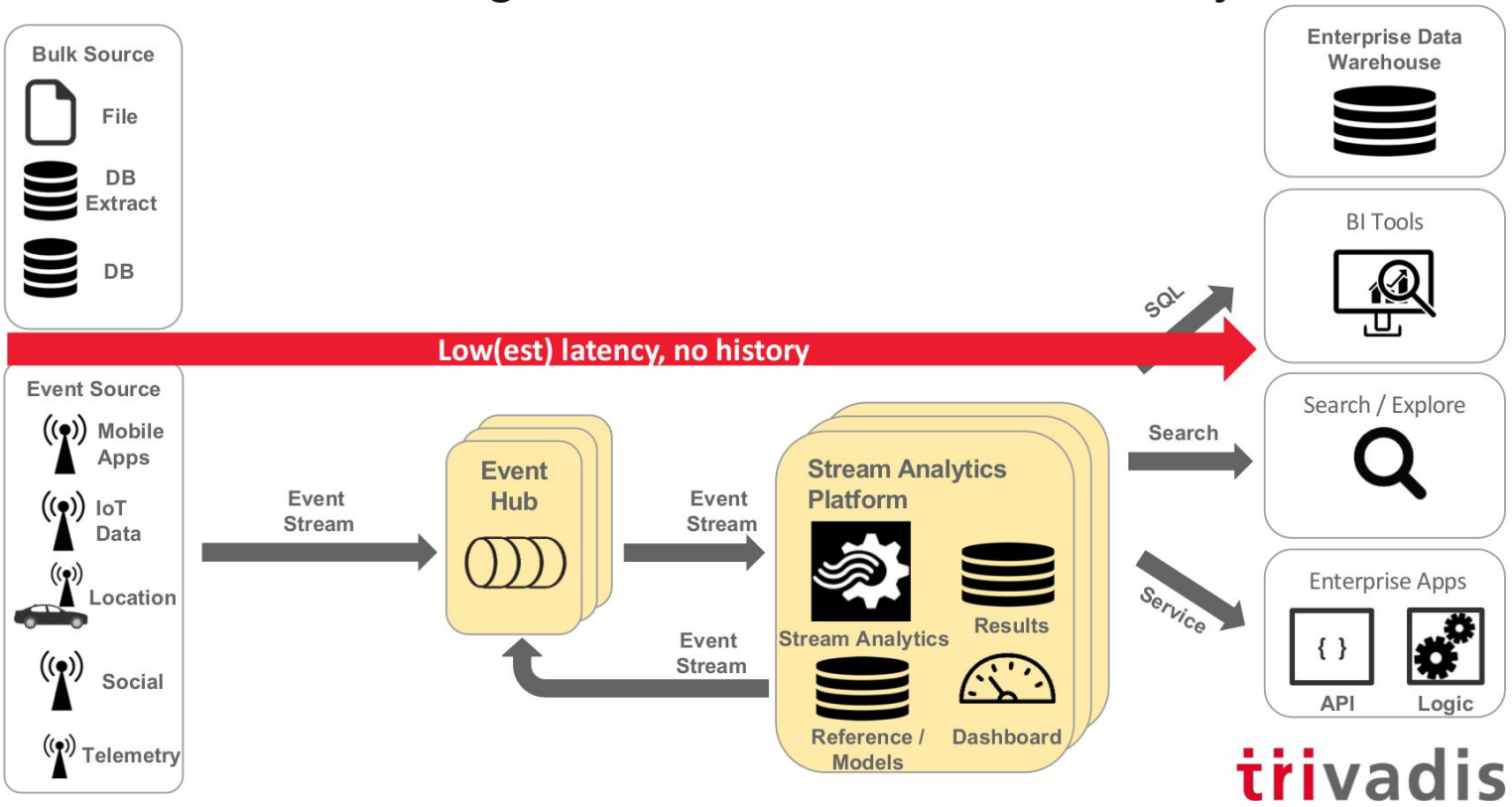
When to Stream / When not?



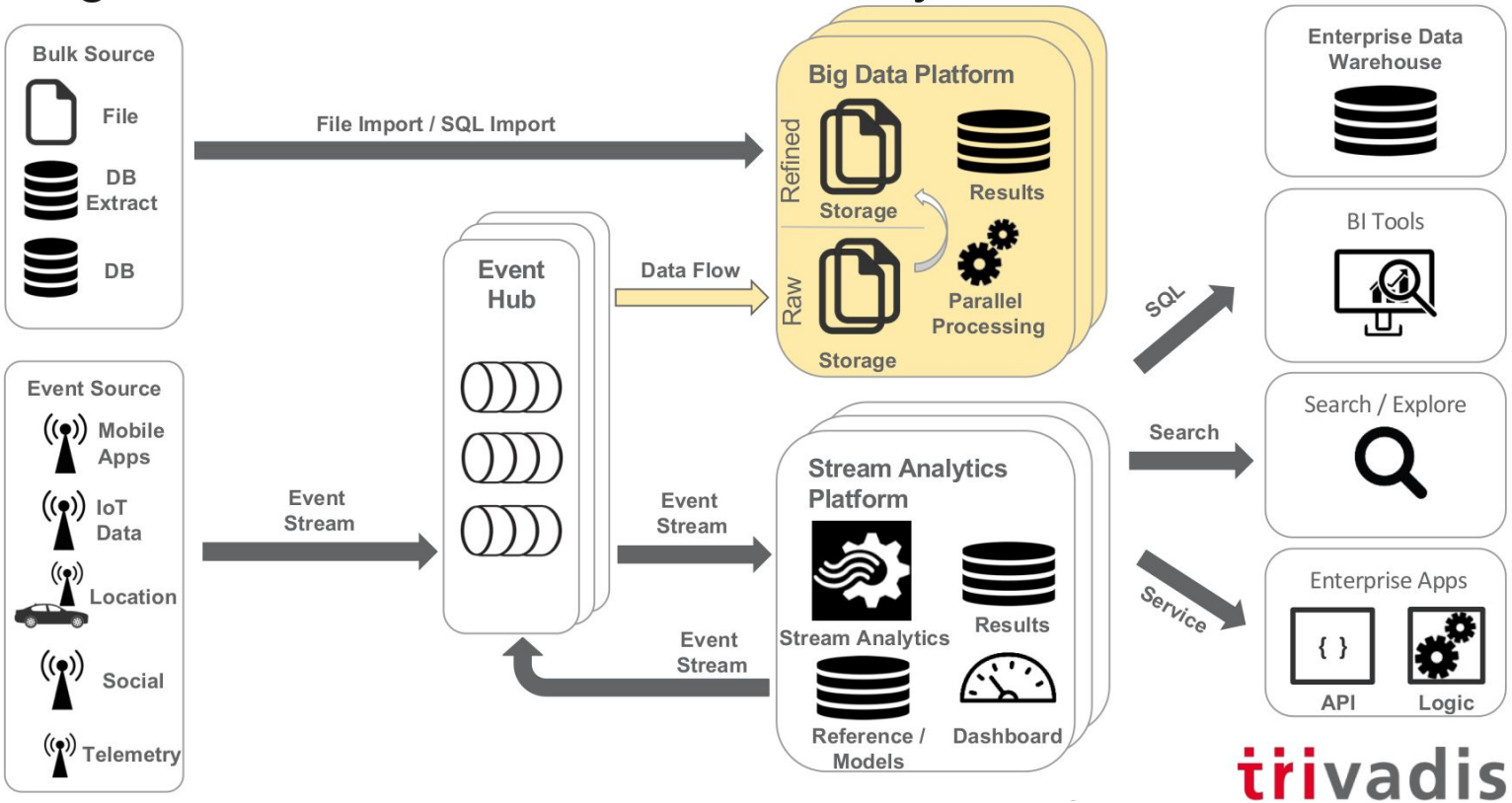
"No free lunch"



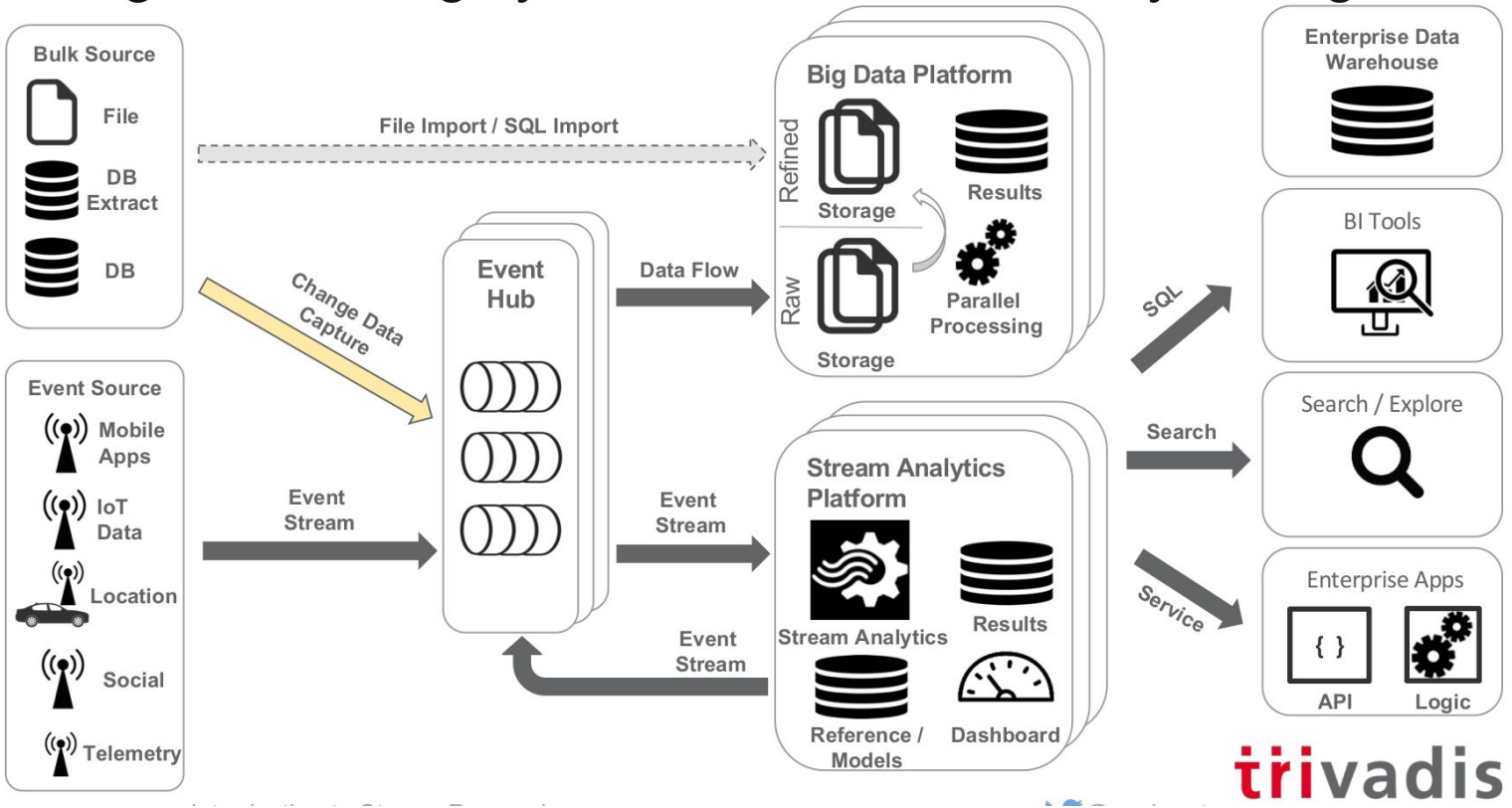
Stream Processing Architecture solves Velocity



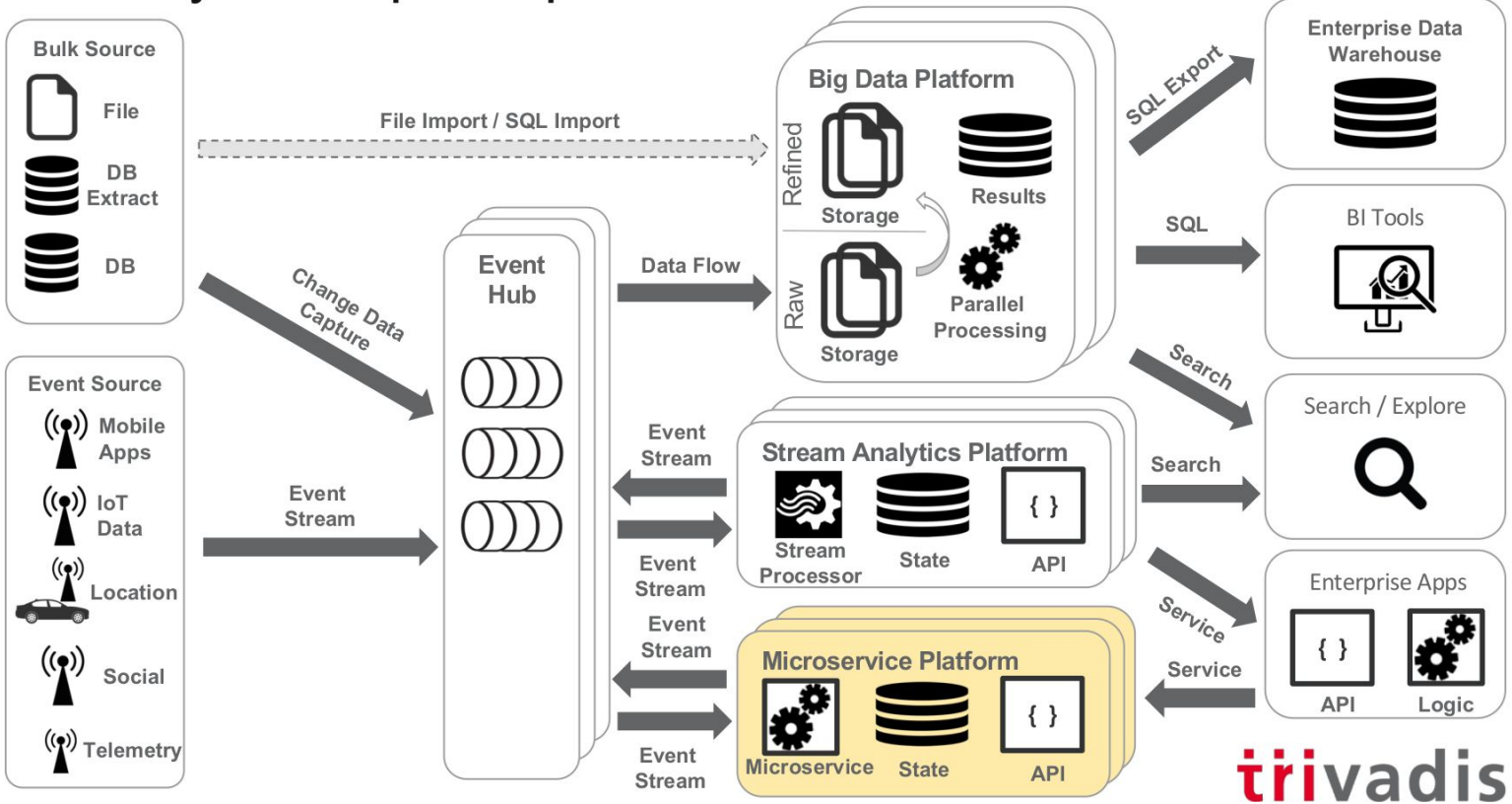
Big Data for all historical data analysis



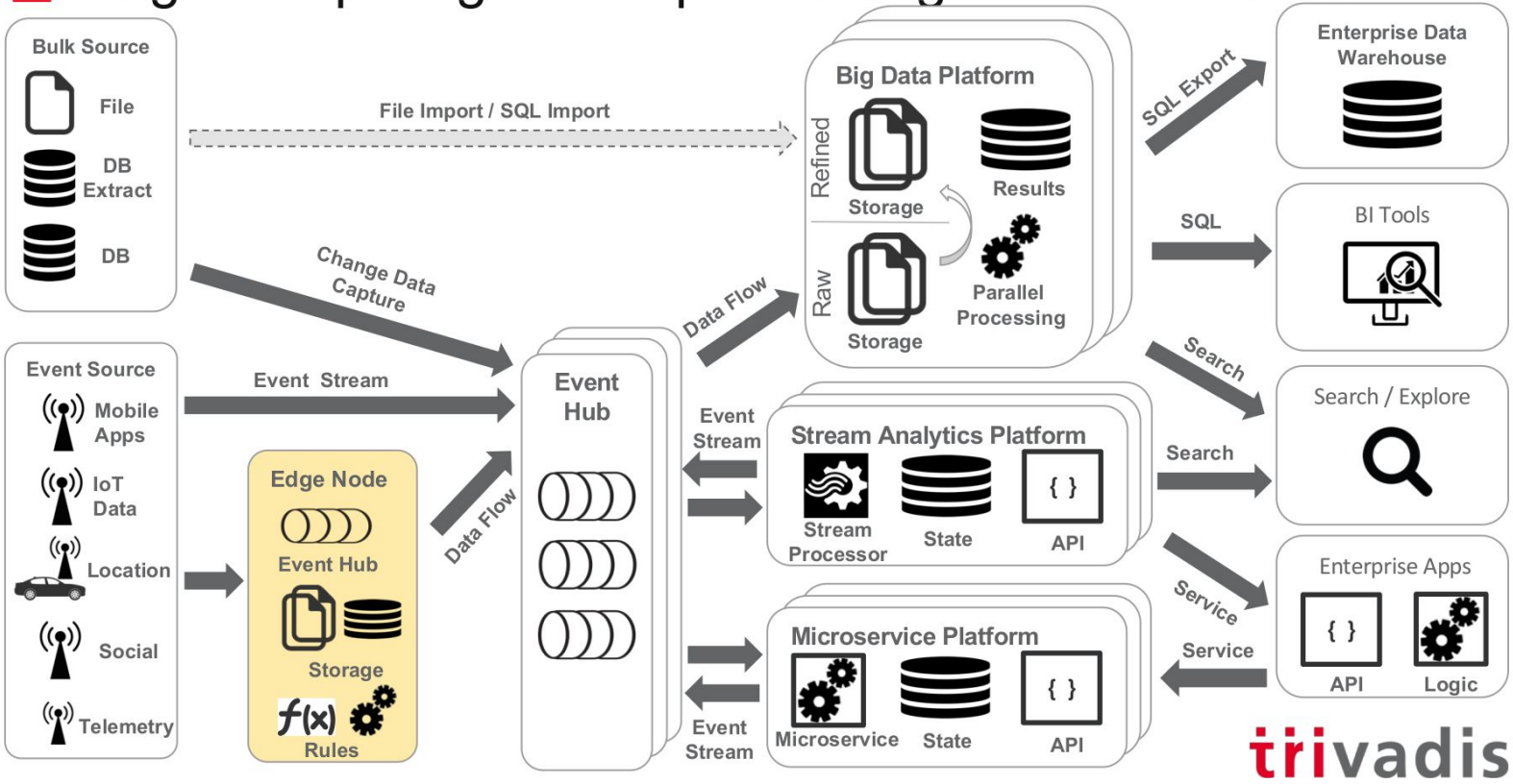
Integrate existing systems with lower latency through CDC



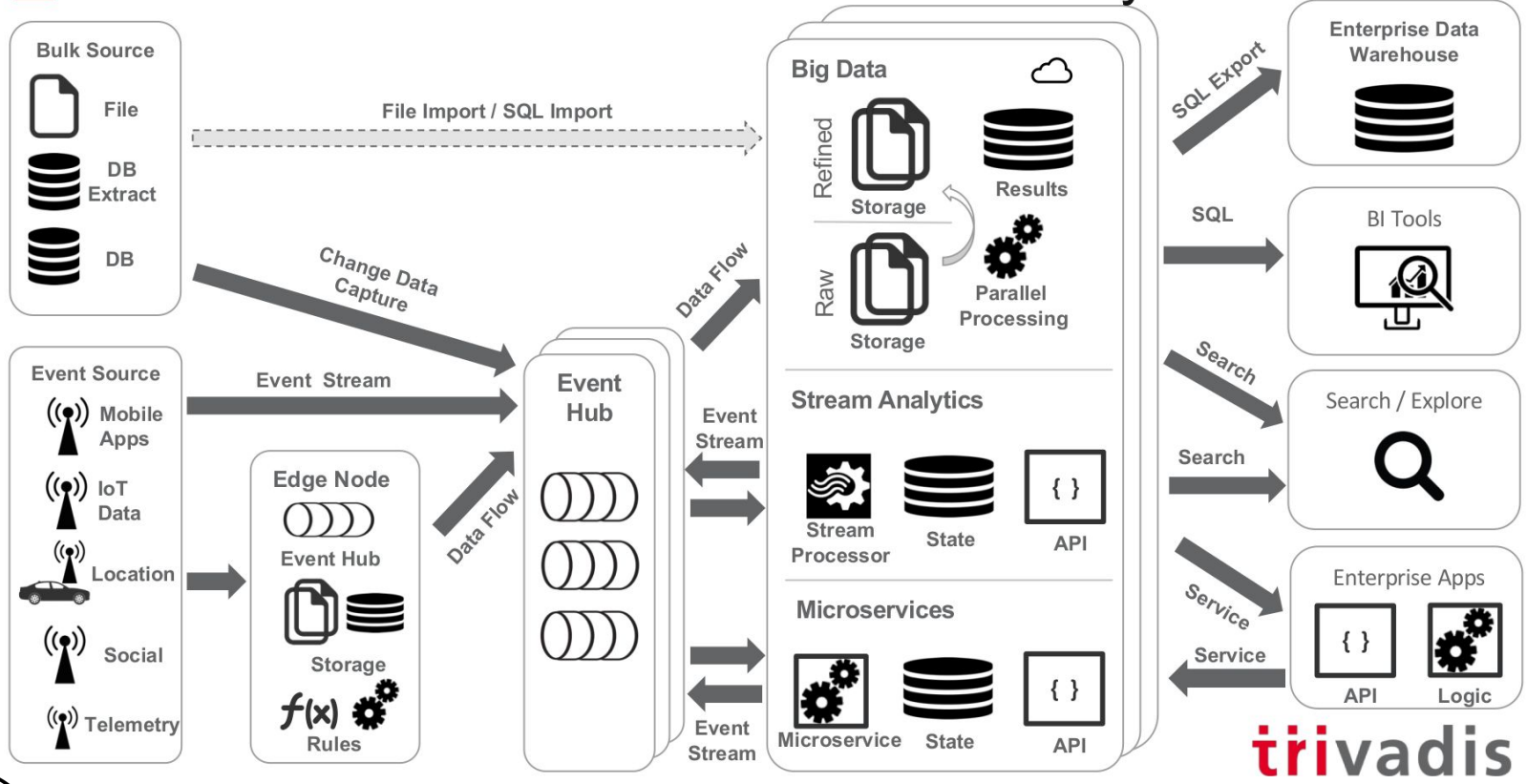
New systems participate in event-oriented fashion



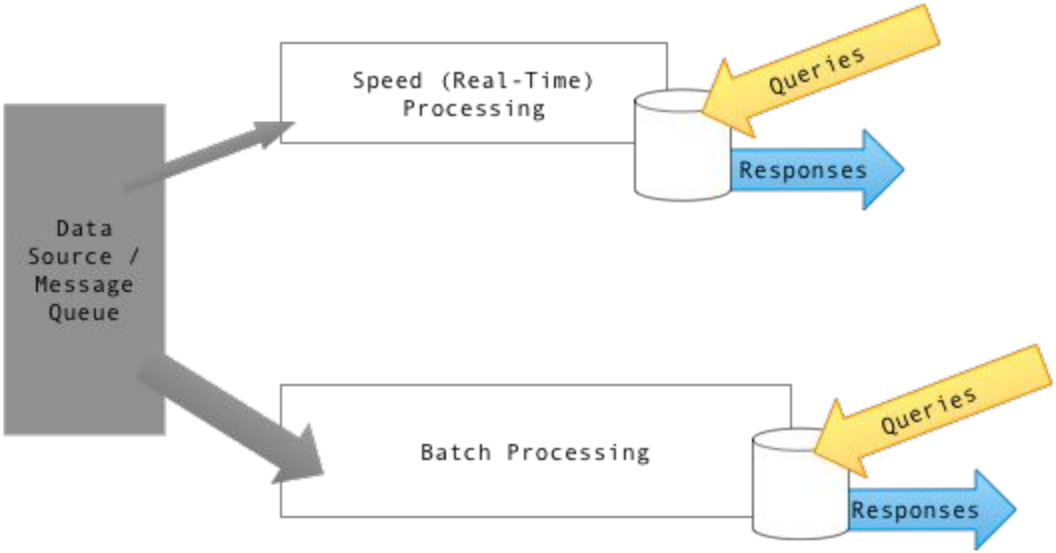
Edge computing allows processing close to data sources



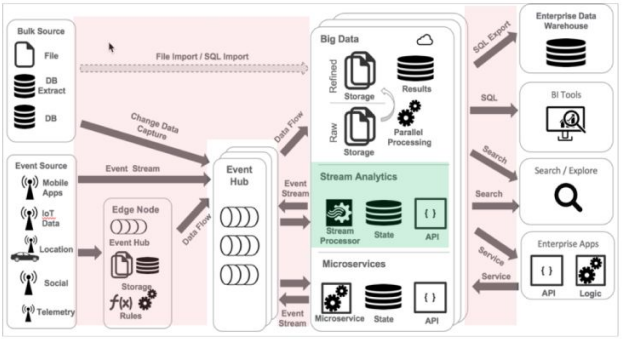
Unified Architecture for Modern Data Analytics Solutions



Lambda Architecture



Two Types of Stream Processing (by Gartner)



Stream Data Integration

- focuses on the ingestion and processing of data sources targeting real-time extract-transform-load (ETL) and data integration use cases
- filter and enrich the data

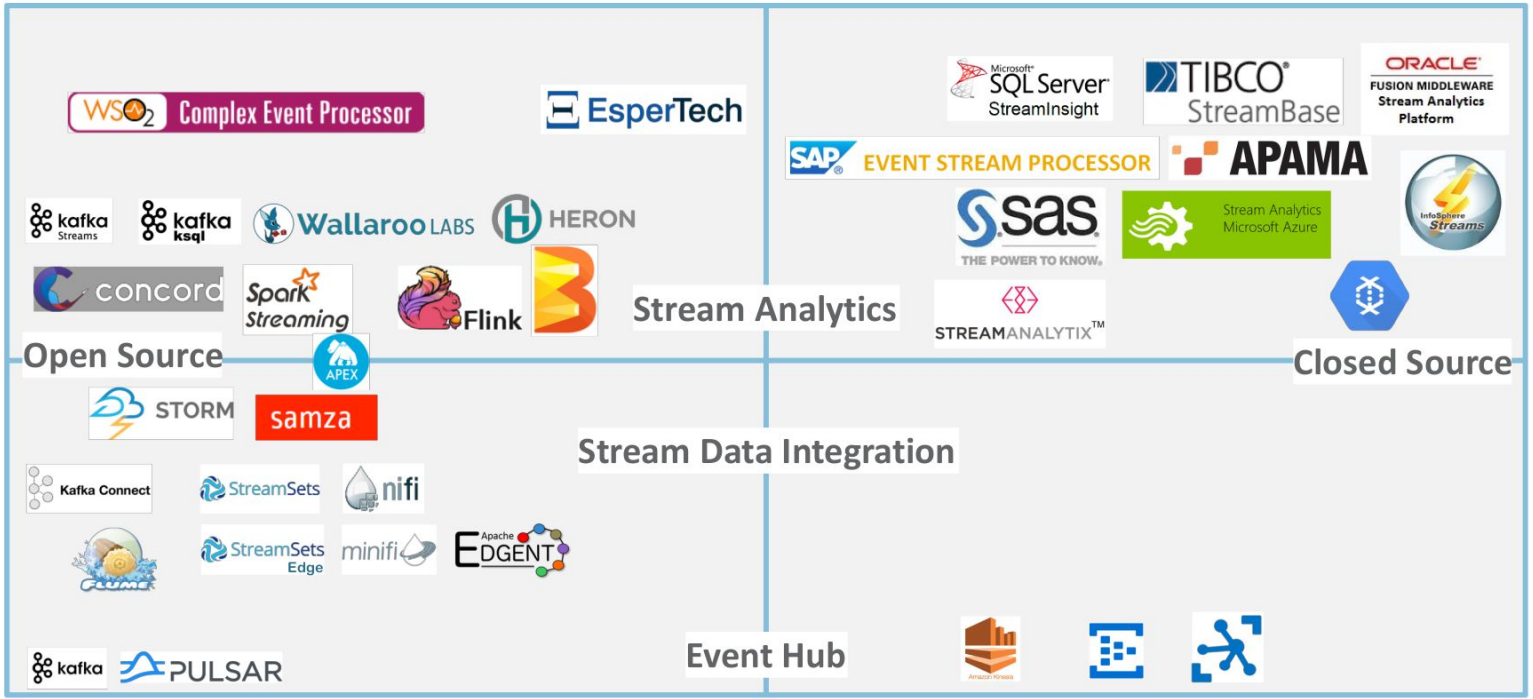
Stream Analytics

- targets analytics use cases
- calculating aggregates and detecting patterns to generate higher-level, more relevant summary information (complex events)
- Complex events may signify threats or opportunities that require a response from the business

Gartner: Market Guide for Event Stream Processing, Nick Heudecker, W. Roy Schulte

trivadis

Stream Processing & Analytics Ecosystem



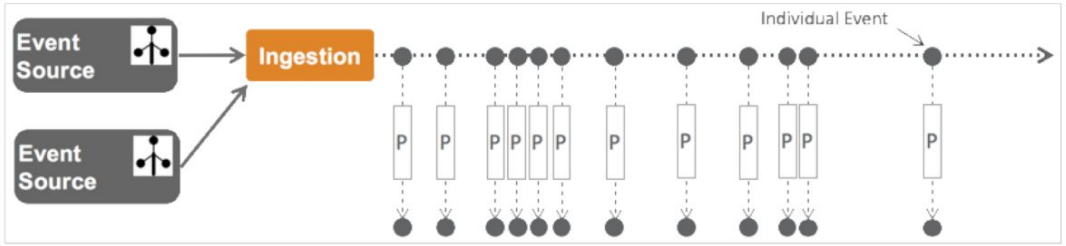
Source: adapted from Tibco

trivadis

Event-at-a-time vs. Micro Batch

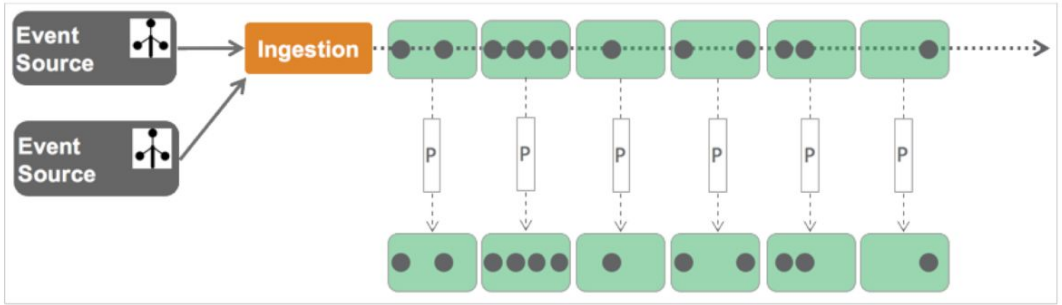
Event-at-a-time Processing

- Events processed as they arrive
- low-latency
- fault tolerance expensive



Micro-Batch Processing

- Splits incoming stream in small batches
- Fault tolerance easier
- Better throughput



Delivery Guarantees

At most once (fire-and-forget)

message is sent, but the sender **doesn't care if it's received or lost**. it is *the easiest and most performant* behavior to support.

At least once

retransmission of a message will occur until an acknowledgment is received. Since a delayed acknowledgment from a receiver could be in flight when the sender retransmits, the message may be received **one or more times**.

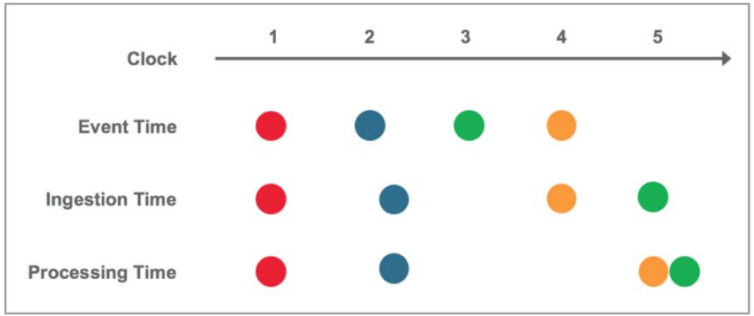
Exactly once

ensures that a message is received ***once and only once***, and is never lost and never repeated. The system must implement whatever mechanisms are required to ensure that a message is received and processed just once

Event Time vs. Ingestion / Processing Time

Event time
the time at which events actually ***occurred***

Ingestion time / Processing Time
the time at which events are ***ingested into*** / ***processed by*** the system



Not all use cases care about event times but many do

Examples
- characterizing user behavior over time
- most billing applications
- anomaly detection

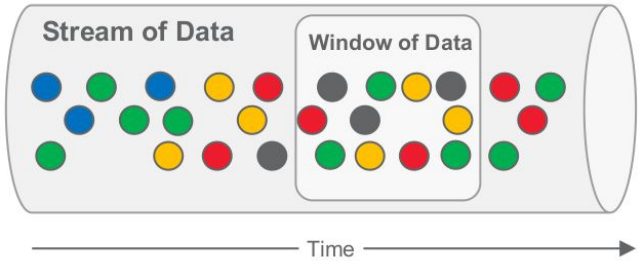
Windowing

Computations over events done using **windows of data**

Due to size and never-ending nature of it, it's not feasible to keep entire stream of data in memory

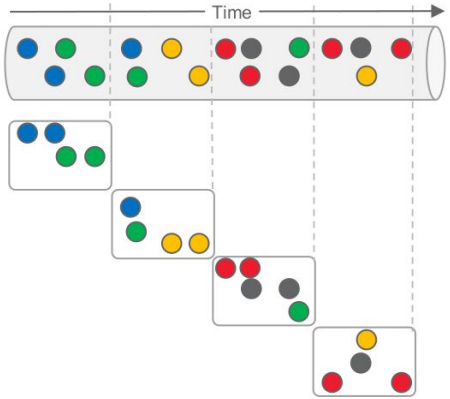
A window of data represents a certain amount of data where we can perform computations on

Windows give the power to keep a working memory and look back at recent data efficiently

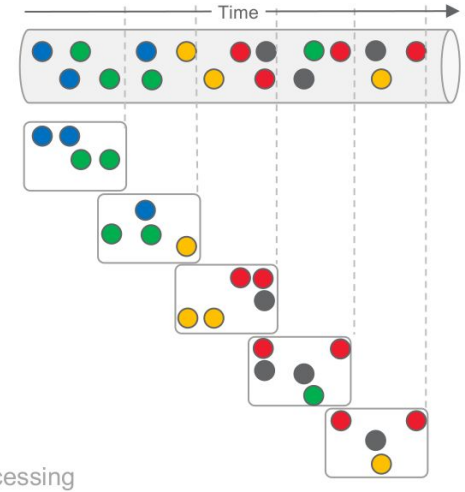


Windowing

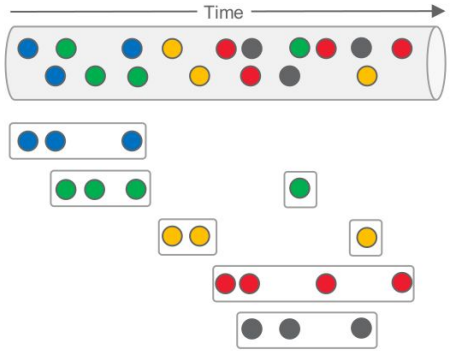
Fixed Window (aka Tumbling Window) - eviction policy always based on the window being full and trigger policy based on either the count of items in the window or time



Sliding Window (aka Hopping Window) - uses eviction and trigger policies that are based on time: *window length* and *sliding interval length*



Session Window – composed of sequences of temporarily related events terminated by a gap of inactivity greater than some timeout



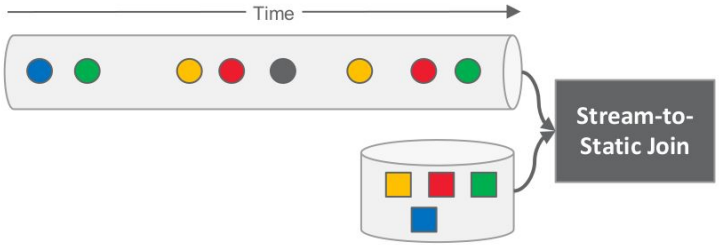
@gschmutz trivadis makes IT easier.

Joining – Stream-to-Static and Stream-to-Stream

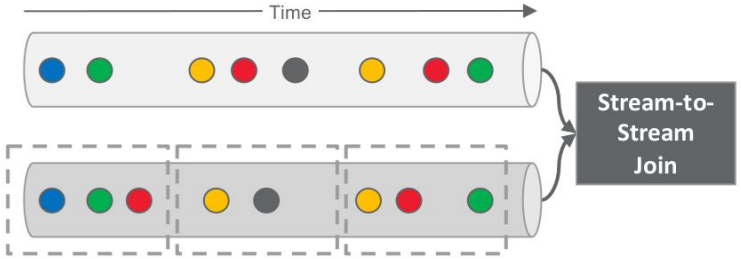
Challenges of joining streams

1. Data streams need to be aligned as they come because they have different timestamps
2. since streams are never-ending, the joins must be limited; otherwise join will never end
3. join needs to produce results continuously as there is no end to the data

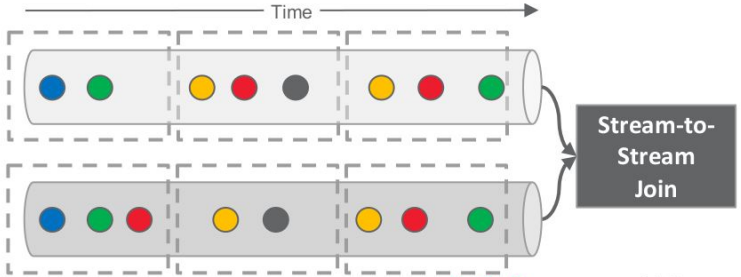
Stream-to-Static (Table) Join



Stream-to-Stream Join (one window join)



Stream-to-Stream Join (two window join)



trivadis

State Management

Necessary if stream processing use case is dependent on previously seen data or external data

Windowing, **Joining** and **Pattern Detection** use State Management behind the scenes

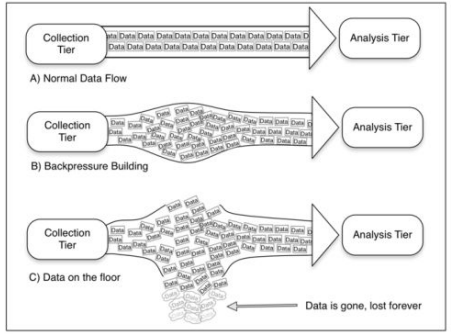
State Management services can be made available for custom state handling logic

State needs to be managed as close to the stream processor as possible

Options for State Management

How does it handle failures? If a machine crashes and the/some state is lost?

7) Back Pressure

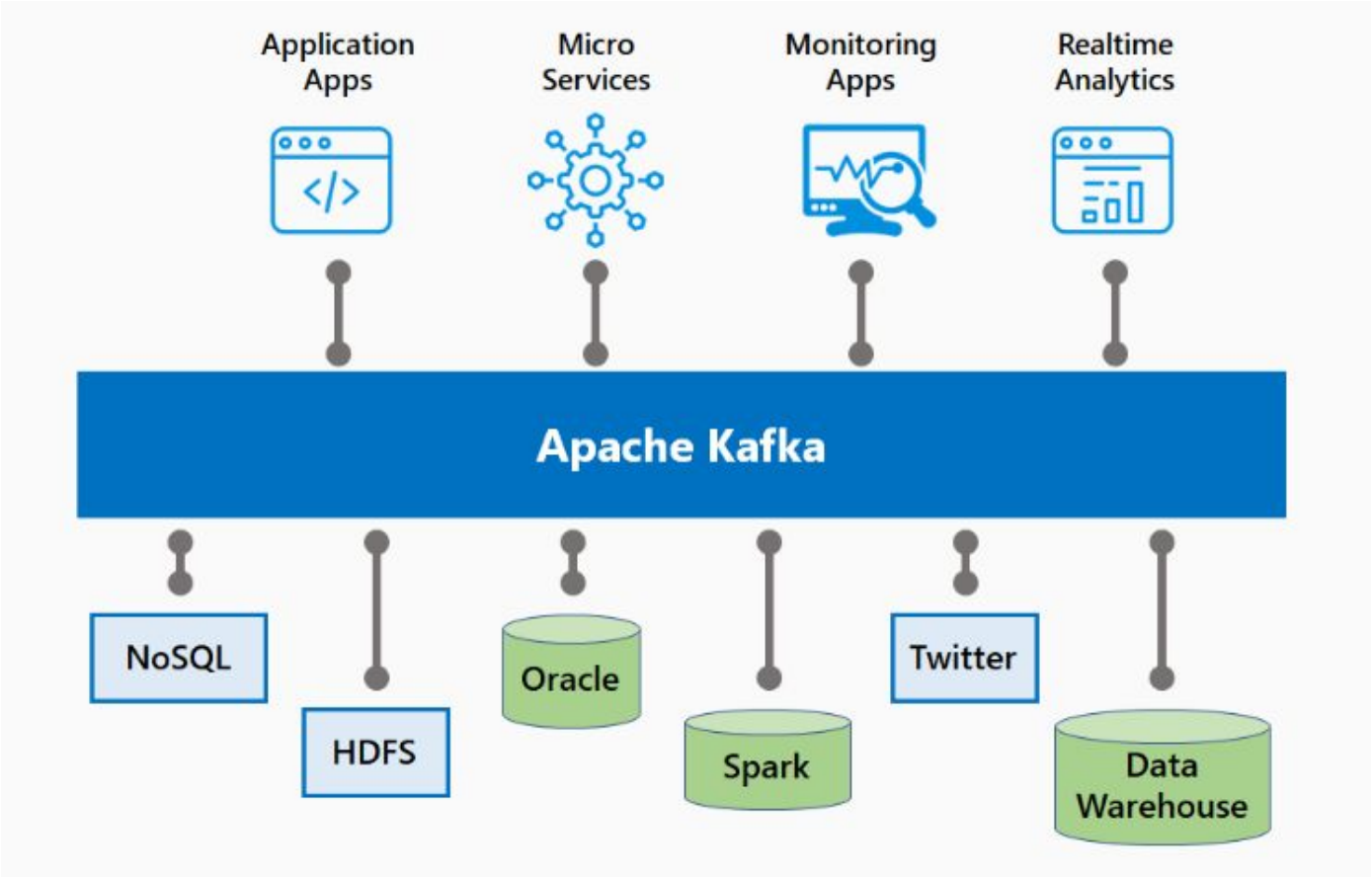


- Backpressure refers to the situation where a system is receiving data at a higher rate than it can process
- Backpressure, if not dealt with correctly, can lead to exhaustion of resources, or even, in the worst case, data loss

- A slow consumer should slow down the producer

trivadis
makes IT easier.


APACHE
kafka®


Application Apps
Micro Services
Monitoring Apps
Realtime Analytics
Apache Kafka
NoSQL
HDFS
Oracle
Spark
Twitter
Data Warehouse

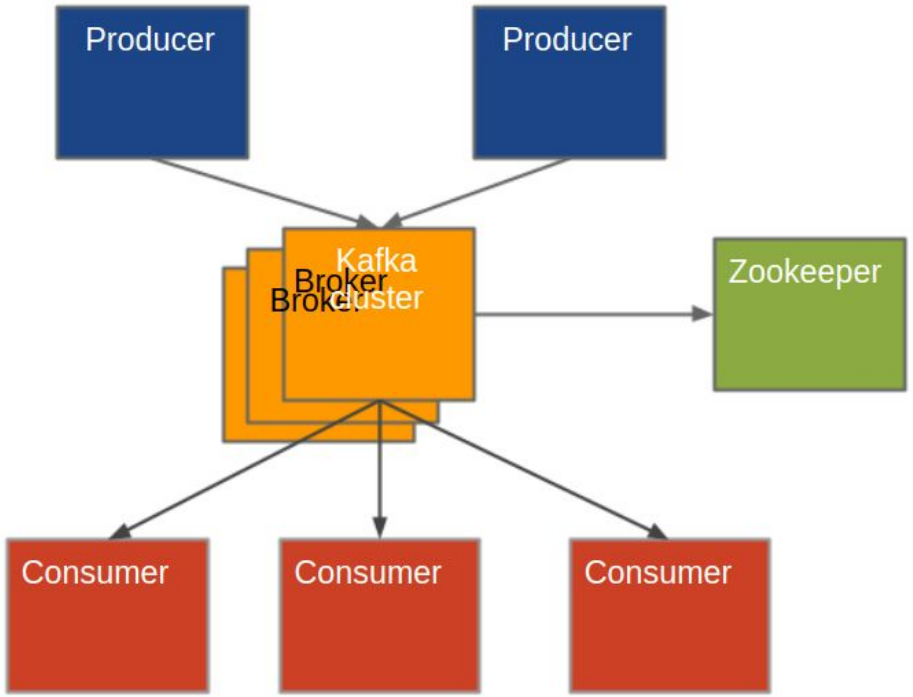
“Kafka is a distributed, partitioned, replicated commit log service. It provides the functionality of a messaging system, but with a unique design”

History

- Apache Kafka was originally developed by LinkedIn, open sourced in early 2011
- Written in Scala

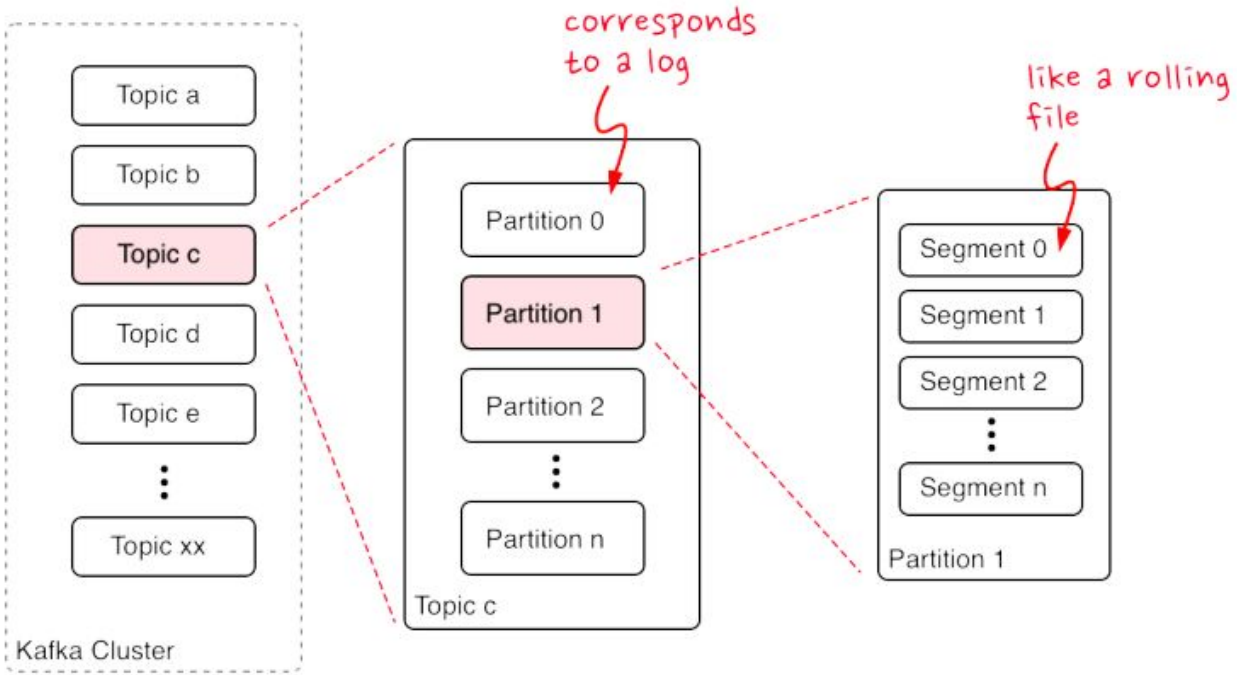
Apache Kafka

- Fast: A single Kafka broker can handle hundreds of megabytes of reads and writes per second from thousands of clients.
- Scalable: Kafka is designed to allow a single cluster to serve as the central data backbone for a large organization. It can be elastically and transparently expanded without downtime. Data streams are partitioned and spread over a cluster of machines to allow data streams larger than the capability of any single machine and to allow clusters of co-ordinated consumers
- Durable: Messages are persisted on disk and replicated within the cluster to prevent data loss. Each broker can handle terabytes of messages without performance impact.
- Distributed by Design: Kafka has a modern cluster-centric design that offers strong durability and fault-tolerance guarantees.

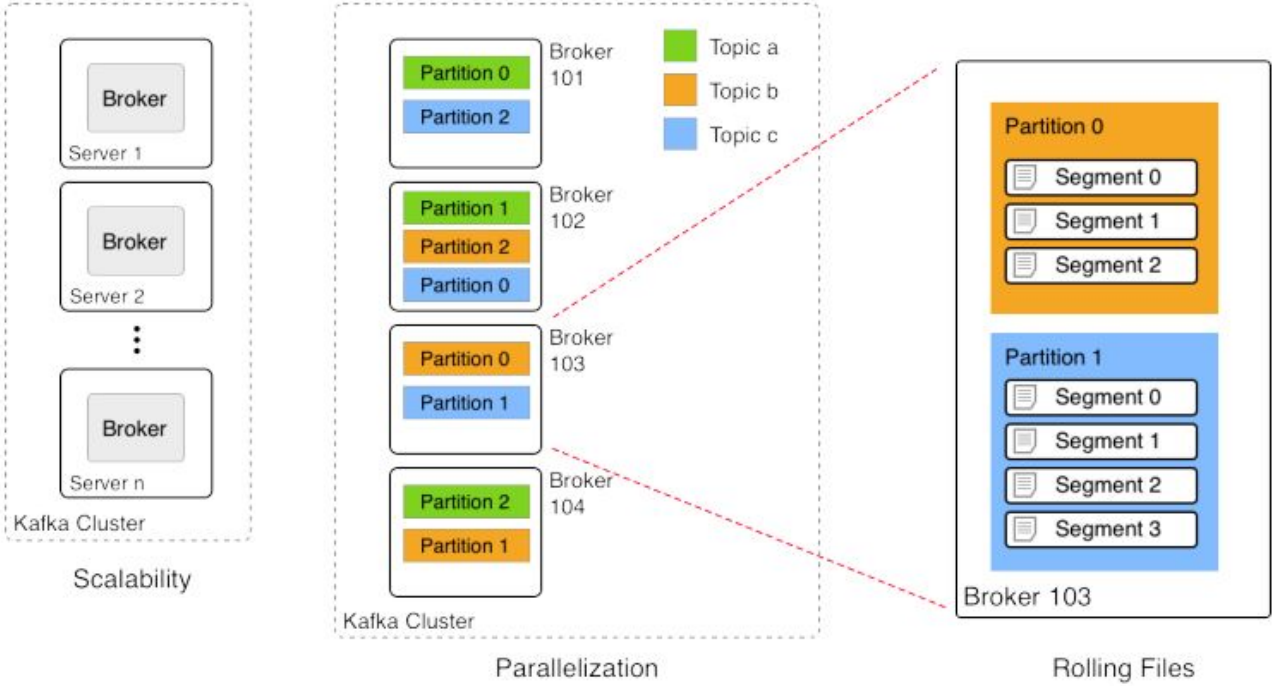


- *Producers* publish messages to a Kafka topic
- *Consumers* process the feed of published messages
- *Cluster* comprises one or more servers (broker)

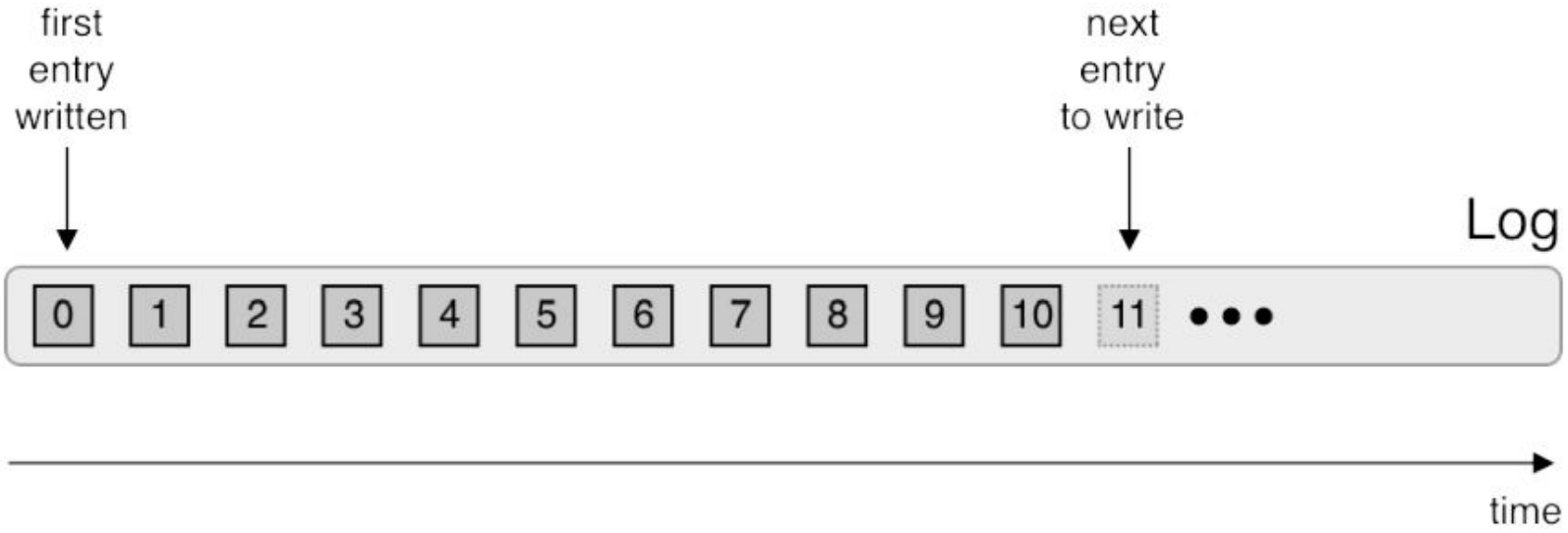
Topics, Partitions, and Segments



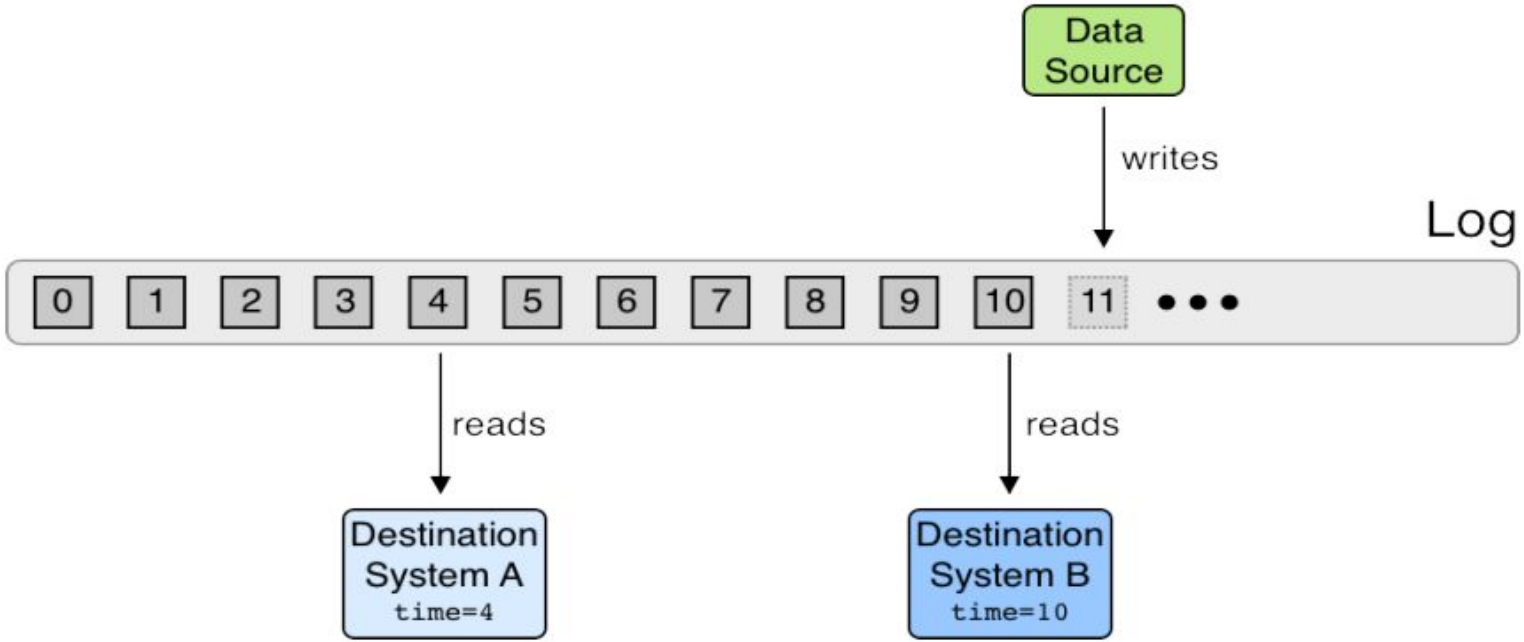
Topics, Partitions, and Segments



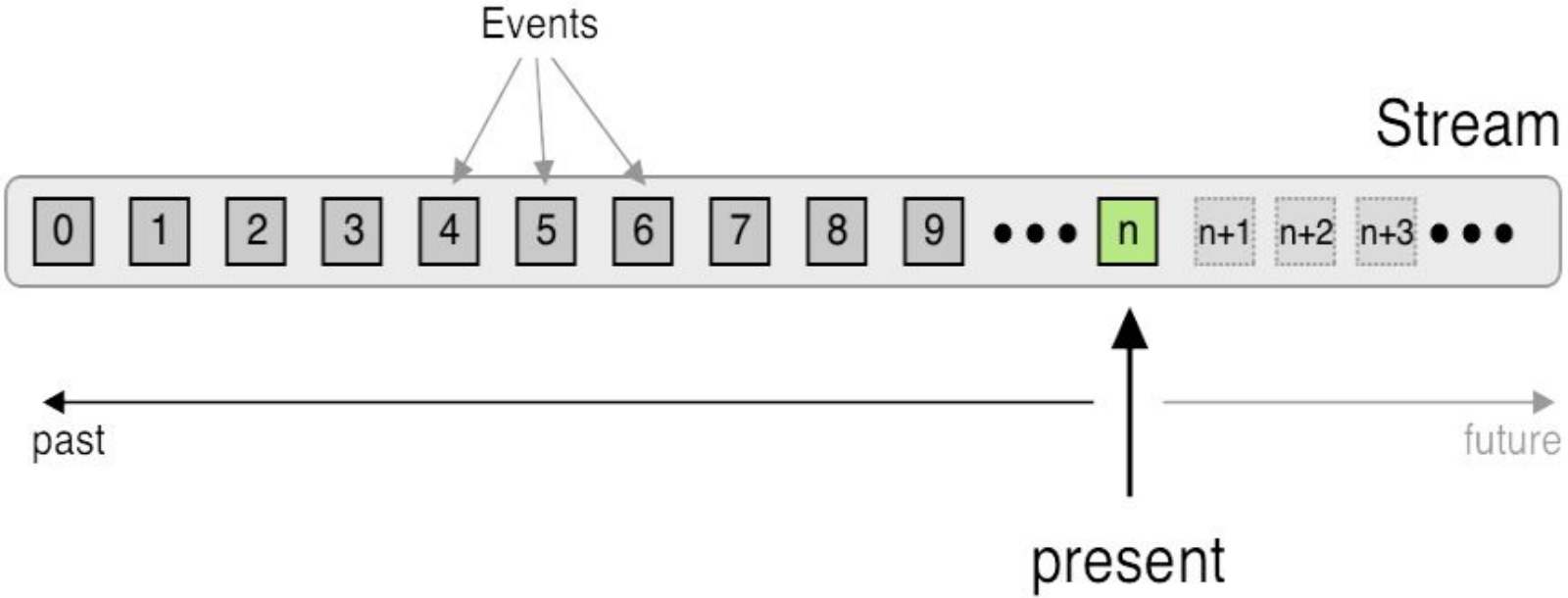
The Log


first
entry
written
next
entry
to write
Log
0
1
2
3
4
5
6
7
8
9
10
11
time

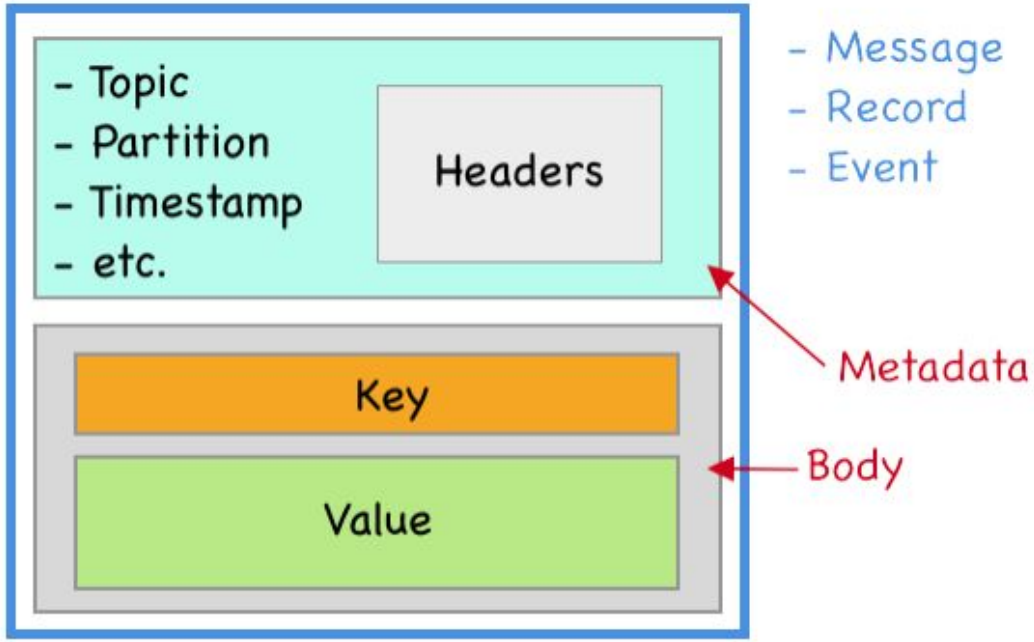
Log Structured Data Flow



The Stream



Data Elements

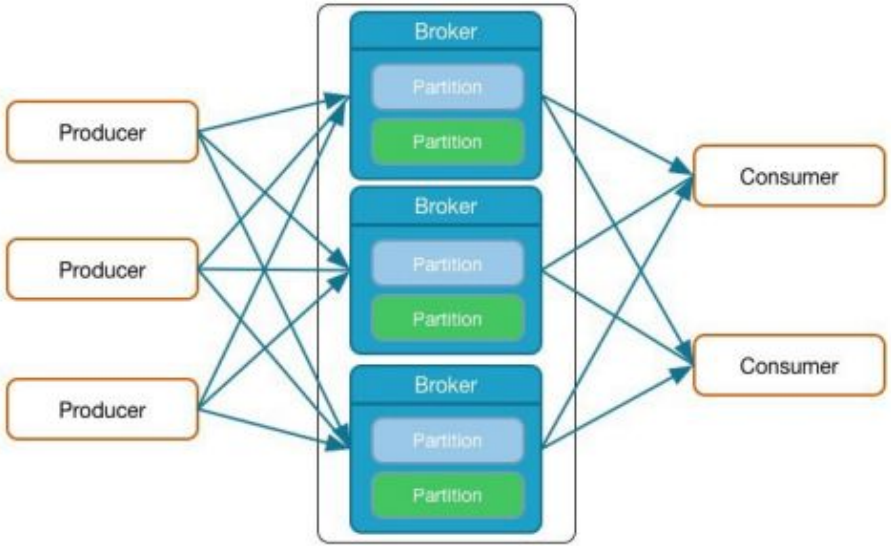


Brokers Manage Partitions

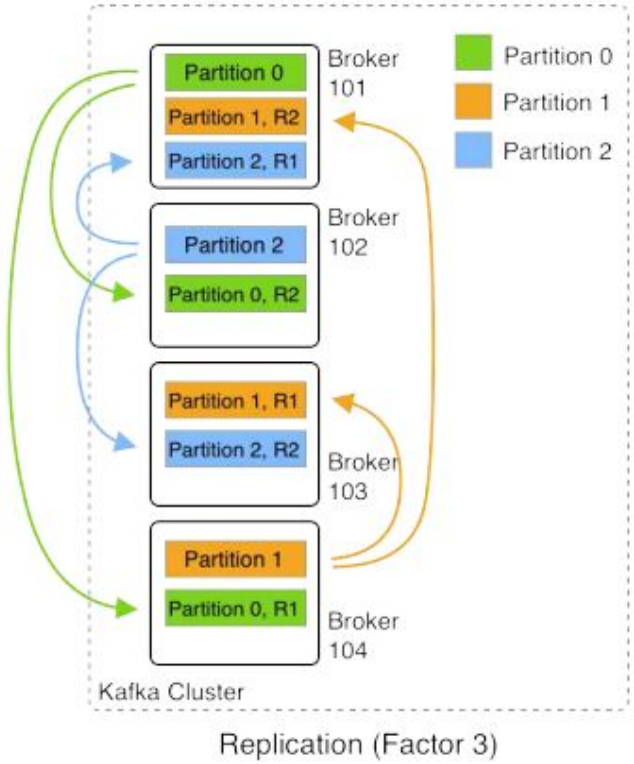
- Messages of Topic spread across Partitions
- Partitions spread across Brokers
- Each Broker handles many Partitions
- Each Partition stored on Broker's disk
- Partition: 1..n log files
- Each message in Log identified by *Offset*
- Configurable Retention Policy

Broker Basics

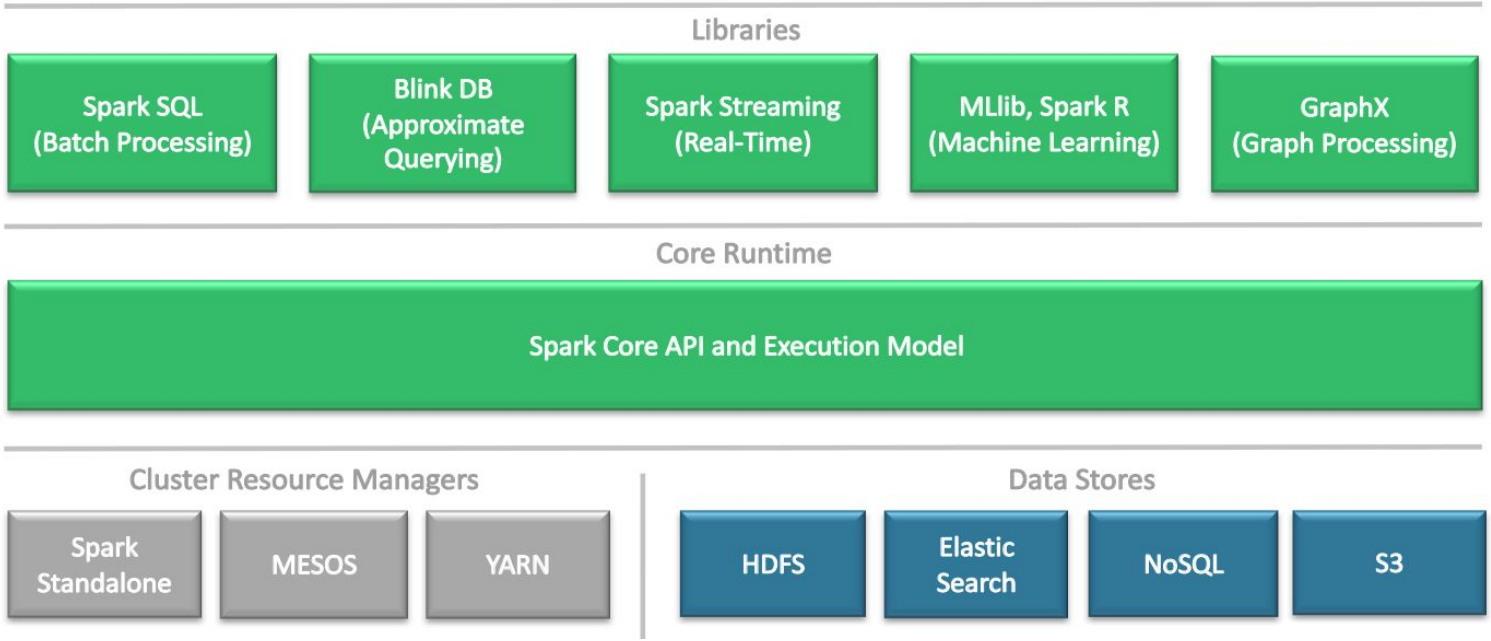
- Producer sends Messages to Brokers
- Brokers receive and store Messages
- A Kafka Cluster can have many Brokers
- Each Broker manages multiple Partitions



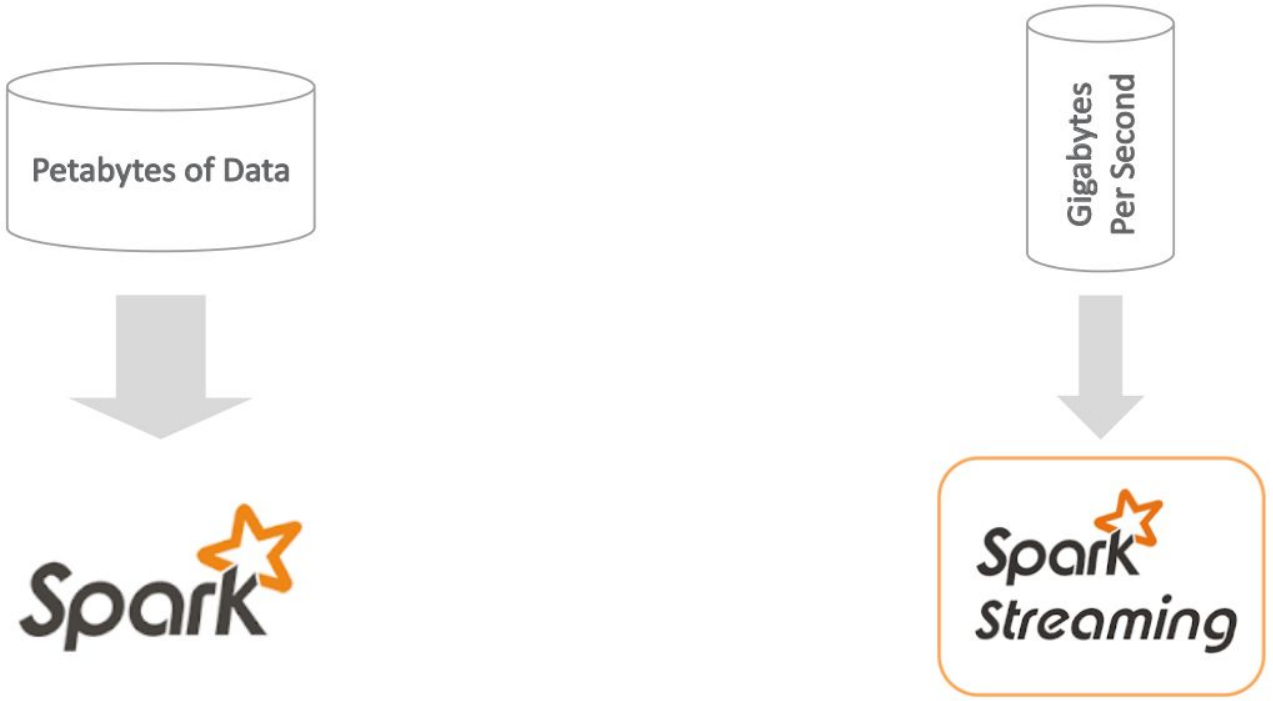
Broker Replication



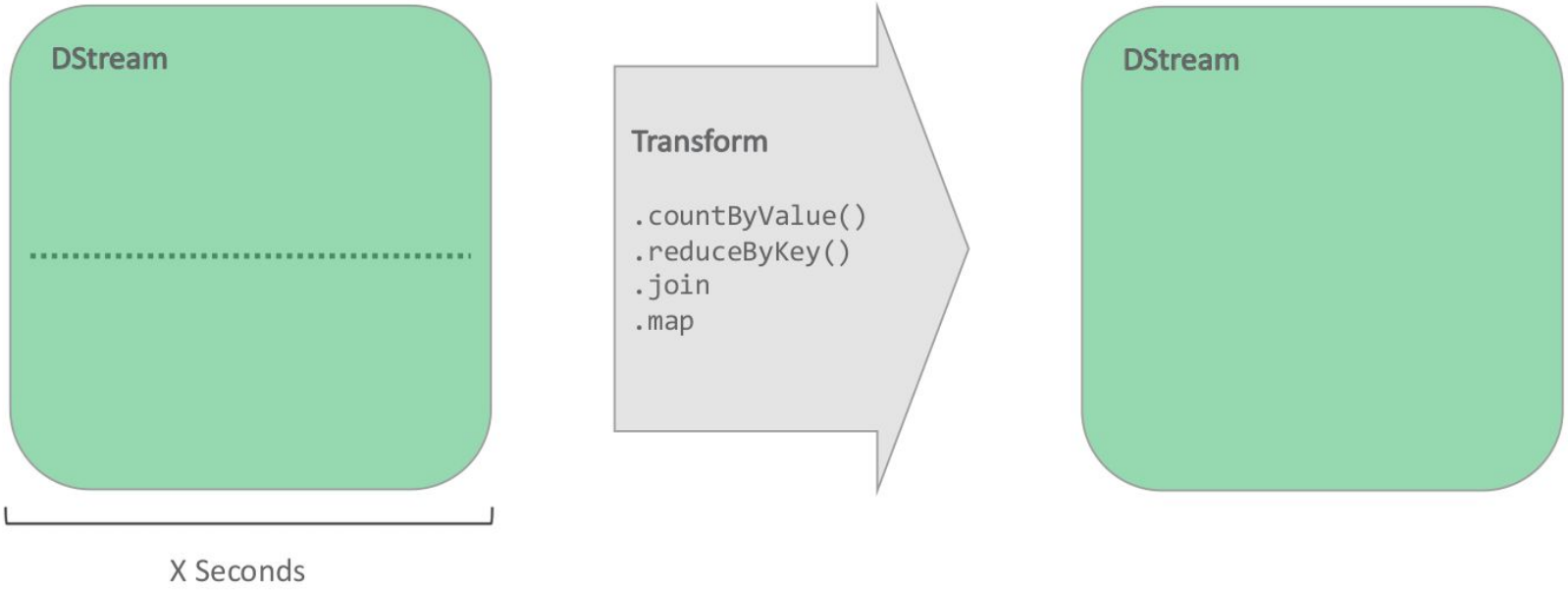
Apache Spark – Stack



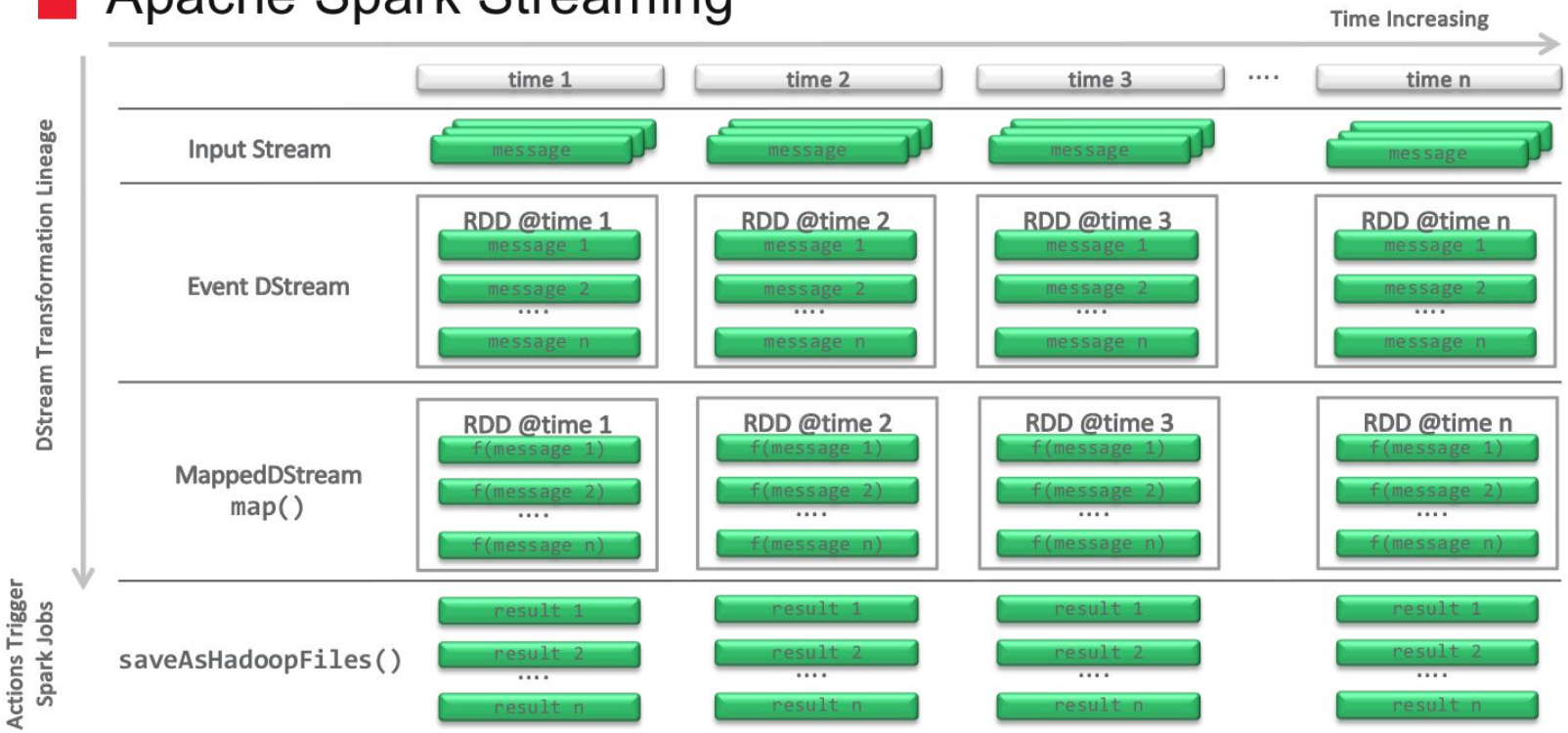
Apache Spark - Batch vs. Real-Time Processing



Apache Spark Streaming



Apache Spark Streaming

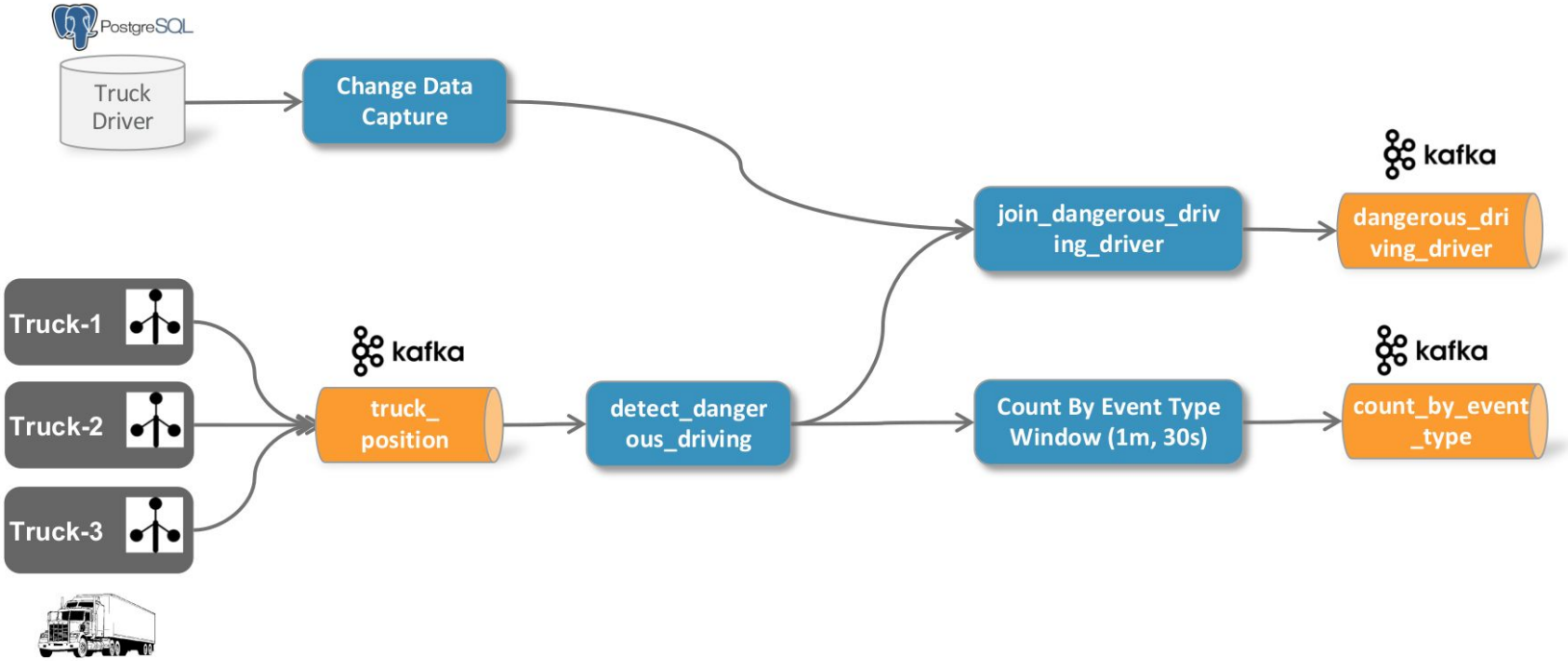


Apache Spark Streaming

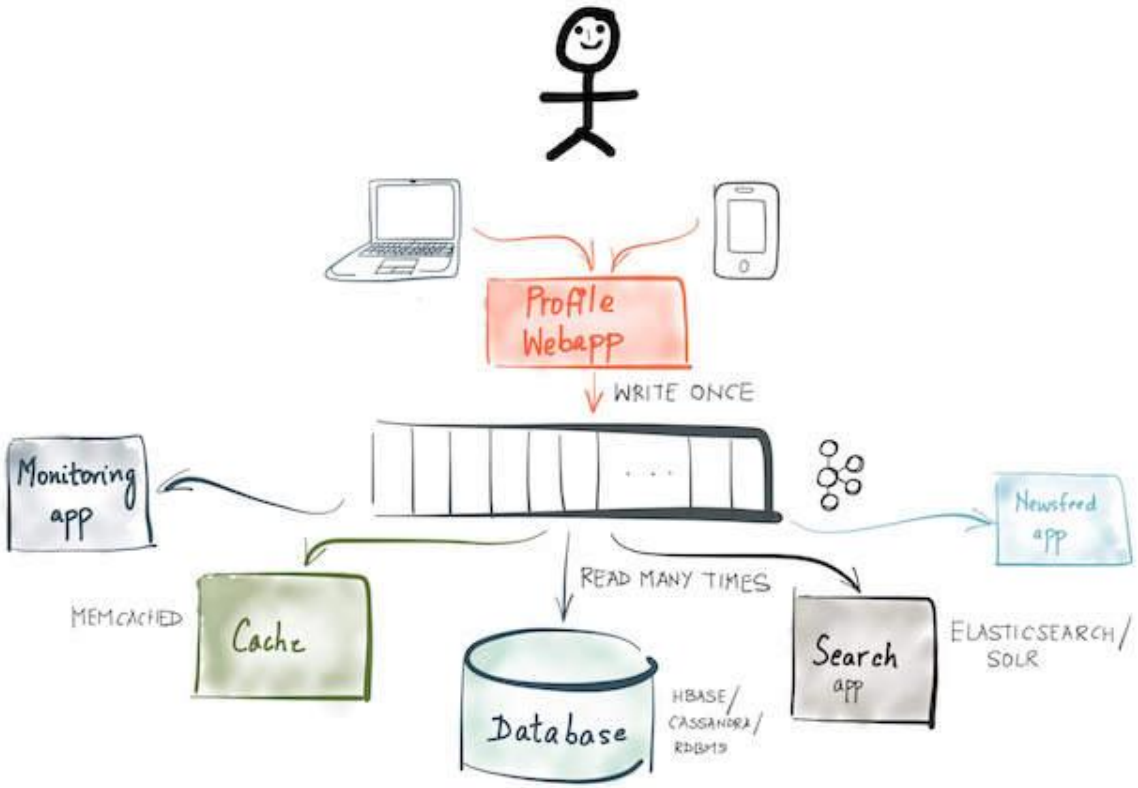
```
class TwitterStreamingHashTagsByInterval extends Serializable {

  def start(auth: Option[Authorization], ssc: StreamingContext, filters: Regex, keyspace: String, table: String): Unit = {

    val transform = (cruft: String) => filters.findAllIn(cruft).flatMap(_.stripPrefix("#"))

    val stream = TwitterUtils.createStream(ssc, auth, Nil, StorageLevel.MEMORY_ONLY_SER_2)

    /** Note that Cassandra is doing the sorting for you here. */
    stream.flatMap(_.getText.toLowerCase.split("""\s+"""))
      .map(transform)
      .countByValueAndWindow(Seconds(5), Seconds(5))
      .transform((rdd, time) => rdd.map { case (term, count) => (term, count, now(time))})
      .saveToCassandra(keyspace, table, SomeColumns("hashtag", "mentions", "interval"))

    ssc.checkpoint("./checkpoint")
    ssc.start()
    ssc.awaitTermination()
  }

  private def now(time: Time): String =
    new DateTime(time.milliseconds, DateTimeZone.UTC).toString("yyyyMMddHH:mm:ss.SSS")
}
```

```
val oderDf = spark
   .readStream
   .format("kafka")
   .option("kafka.bootstrap.servers",
"broker-1:9092")
   .option("subscribe", ”order")
   .load()

val orderFilteredDf =
    orderDf.where(
         ”address.county = 'Switzerland'")
```

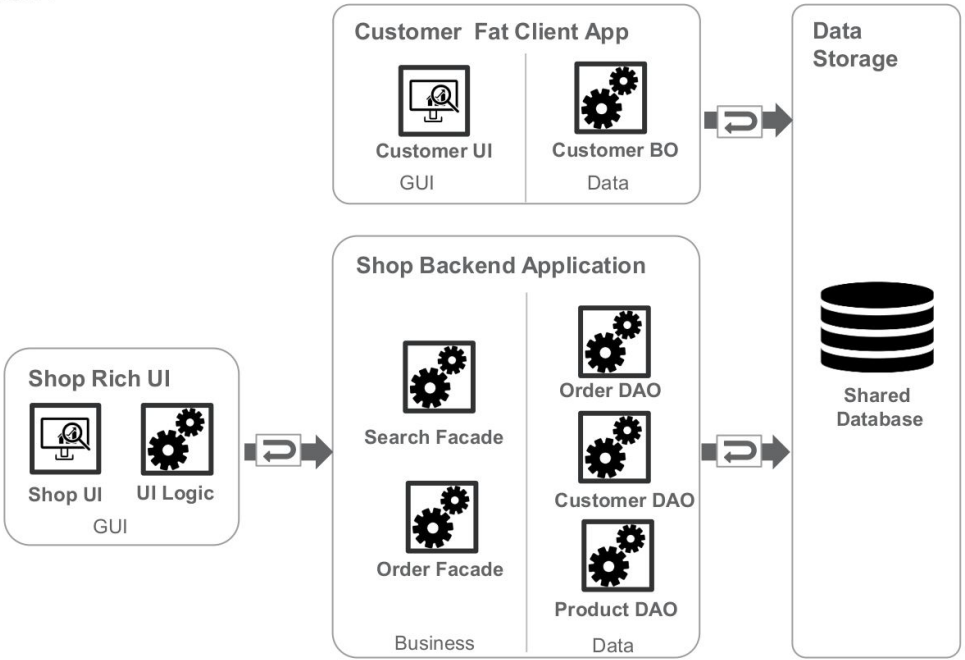
Sample Use Case



Event Sourcing


Profile Webapp
WRITE ONCE
Monitoring app
Newsfeed app
MEMCACHED
Cache
READ MANY TIMES
Database
HBASE/ CASSANDRA/ RDBMS
Search app
ELASTICSEARCH/ SOLR

Traditional Approach



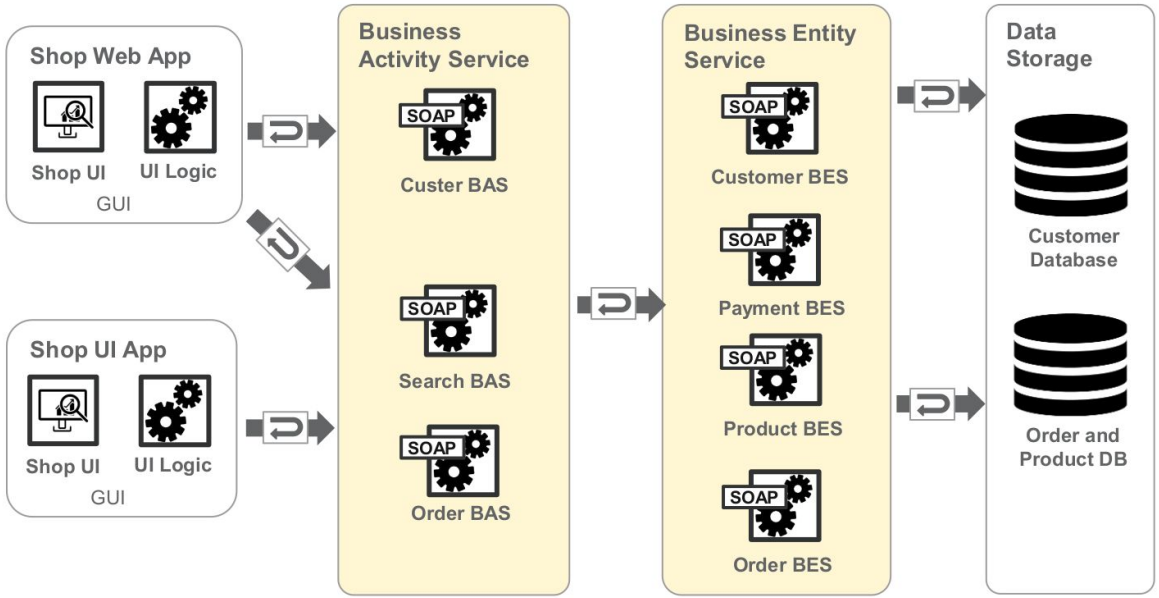
SOA Approach

Contract-first Web Services

Technical layers offer their own interfaces

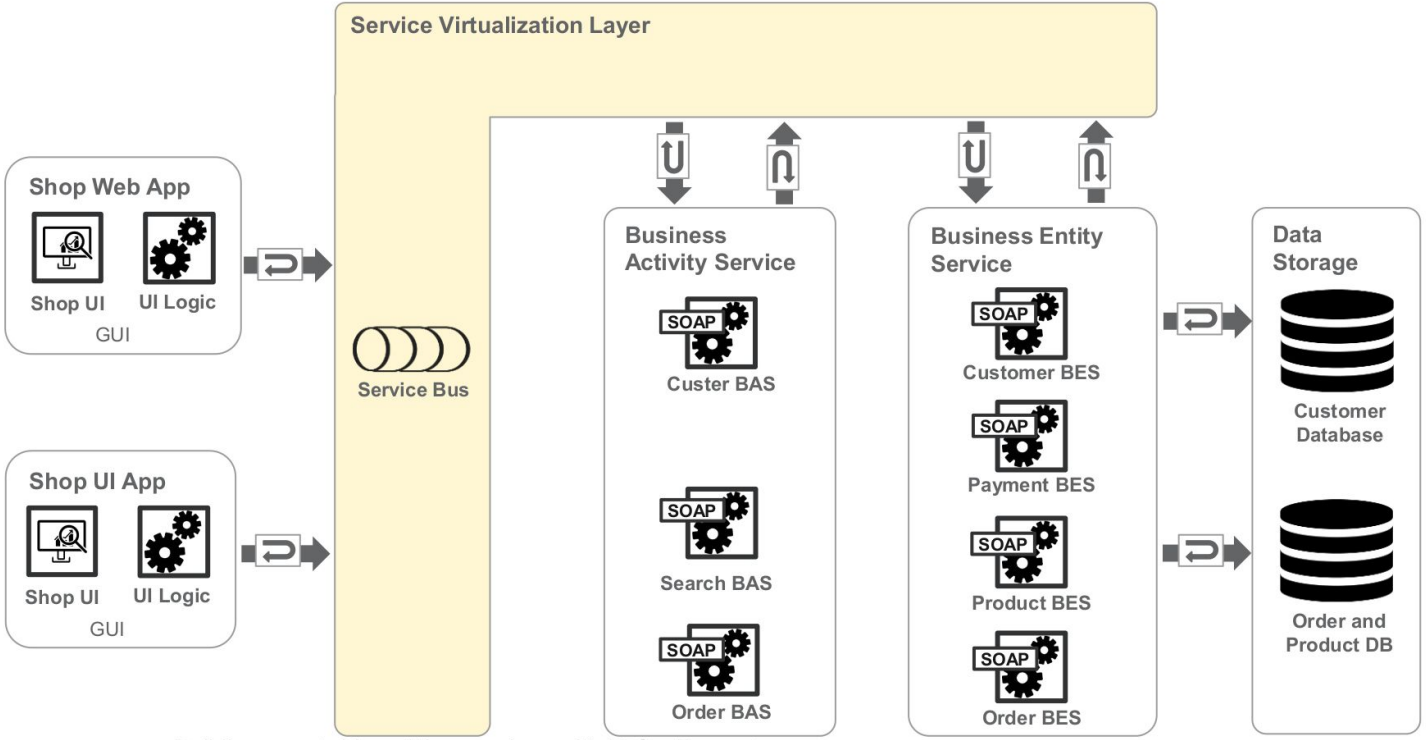
Reuse on each level

Lower layer often wraps legacy code



trivadis makes IT easier.

Virtualized SOA Approach



Building event-driven Microservices with Kafka Ecosystem

Microservice Approach

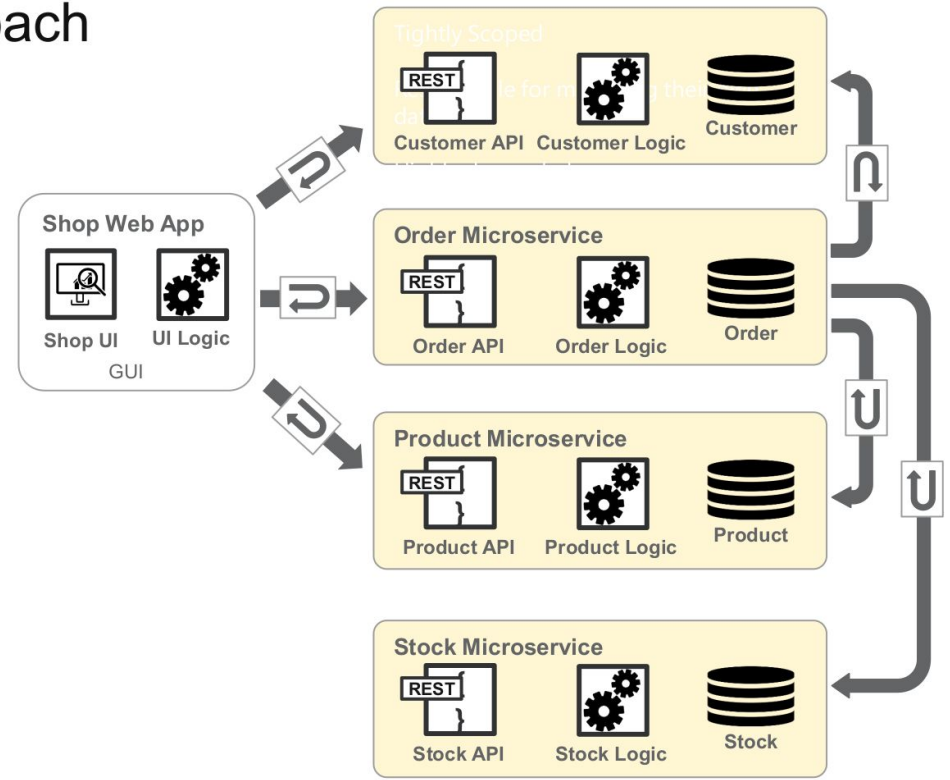
Tightly Scoped behind clear interfaces

Responsible for managing their own data (not necessarily the infrastructure)

Should be highly decoupled

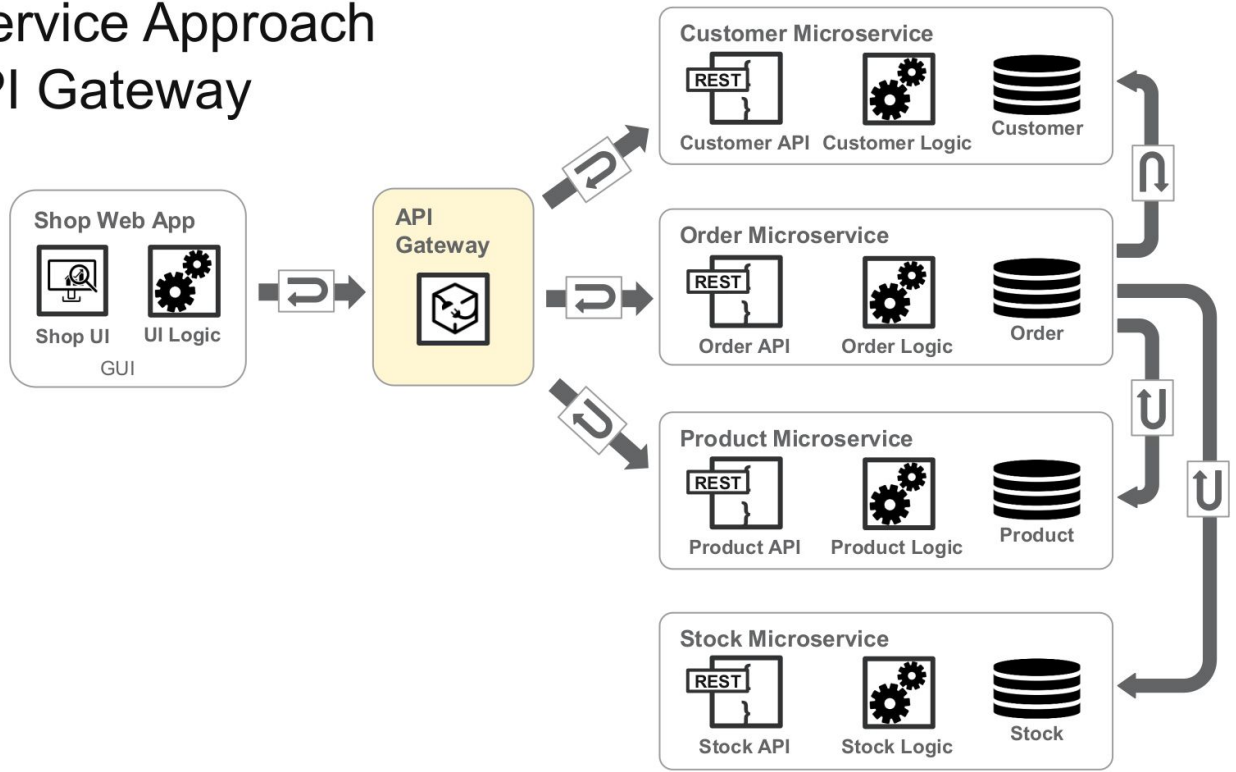
Independently deployable, self-contained and autonomous

SOA done right ?!



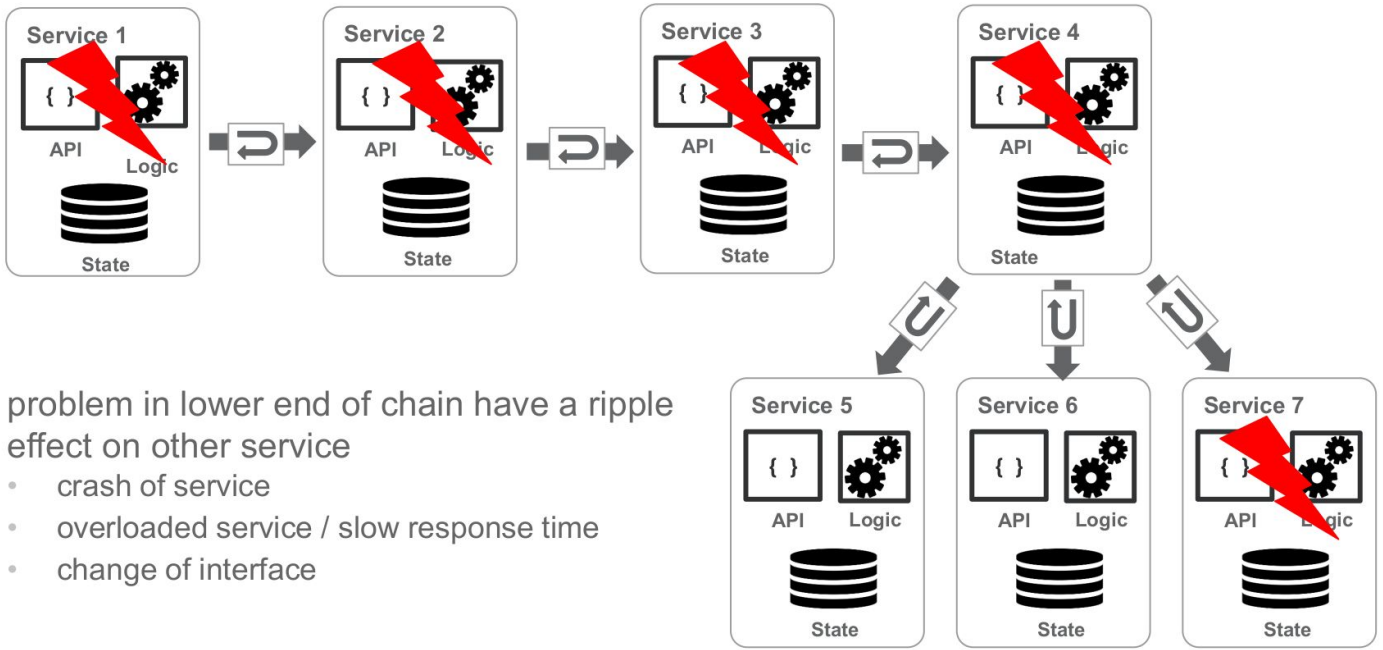
Building event-driven Microservices with Kafka Ecosystem

Microservice Approach with API Gateway



Building event-driven Microservices with Kafka Ecosystem

Synchronous World of Request-Response leads to tight, point-to-point couplings

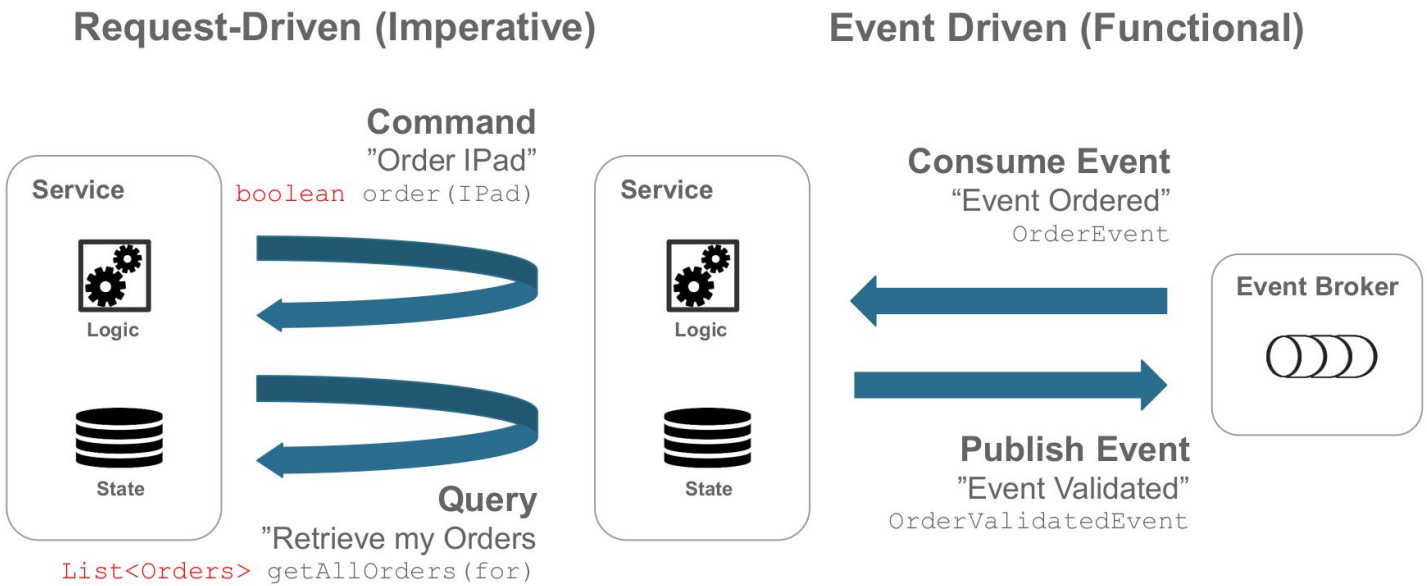


problem in lower end of chain have a ripple effect on other service

- crash of service
- overloaded service / slow response time
- change of interface

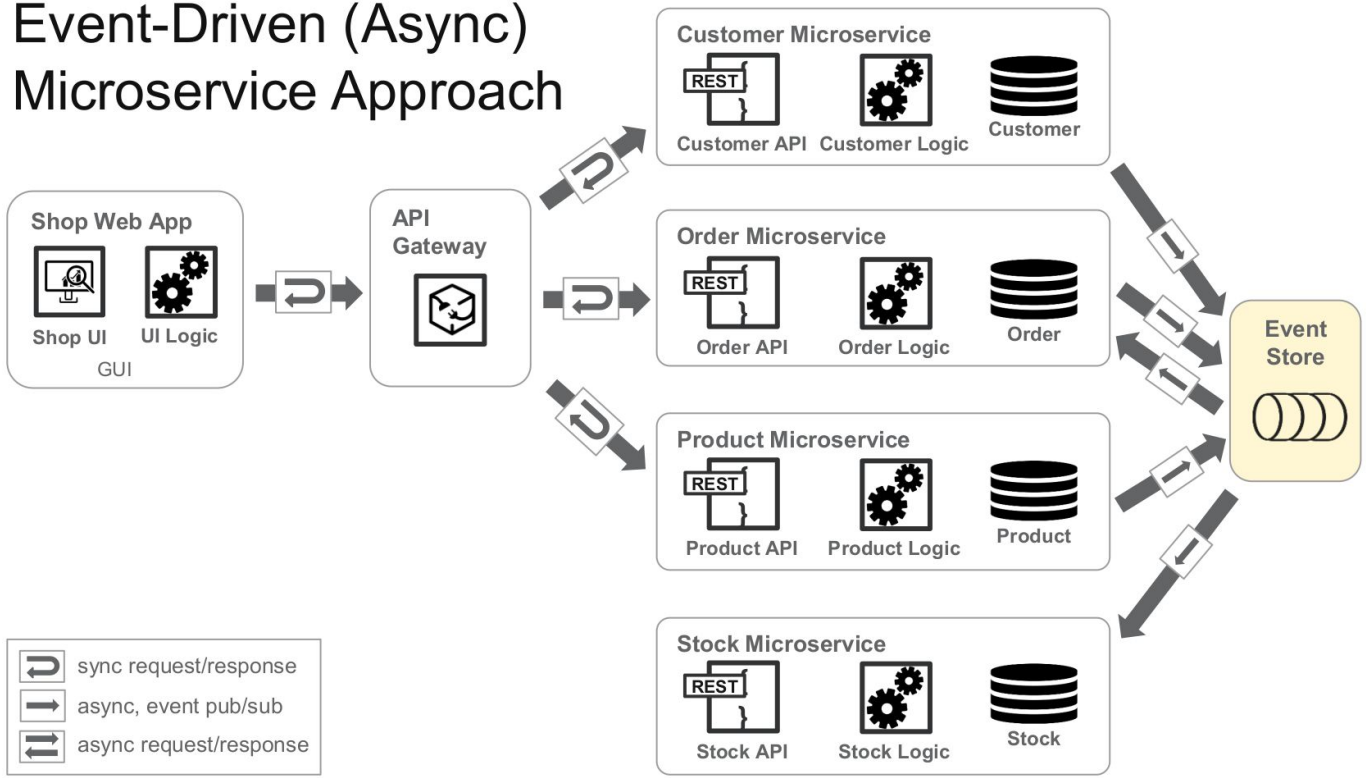
Building event-driven Microservices with Kafka Ecosystem

Three mechanisms through which services can interact



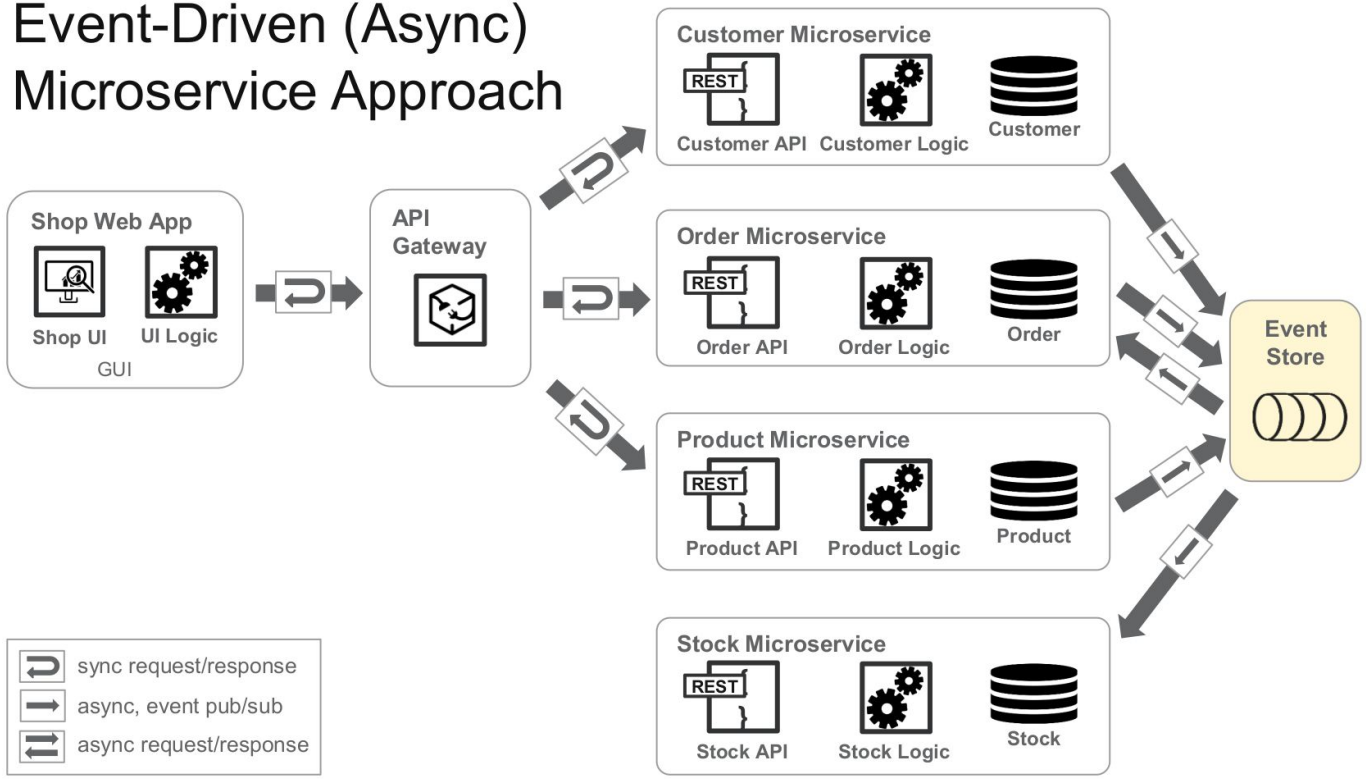
trivadis makes IT easier.

Event-Driven (Async) Microservice Approach

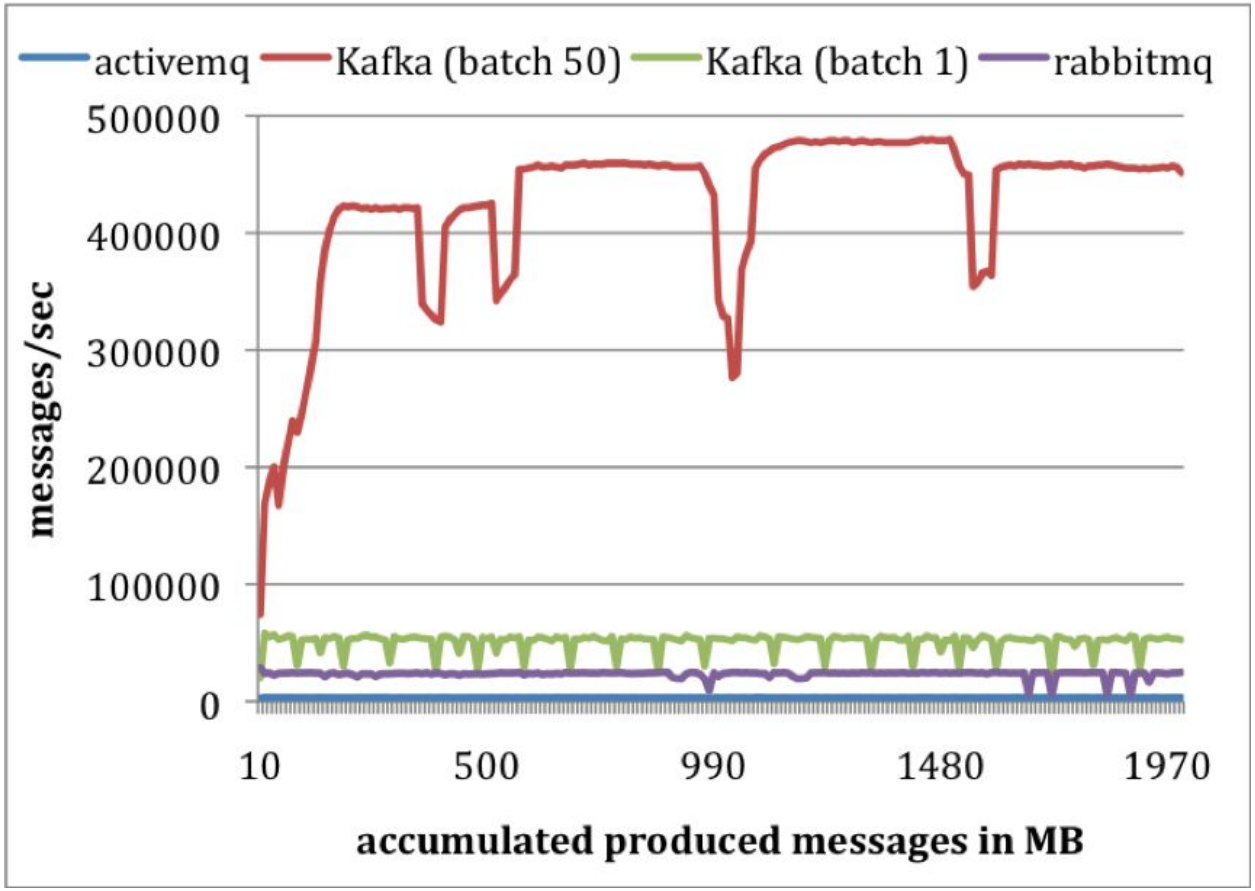


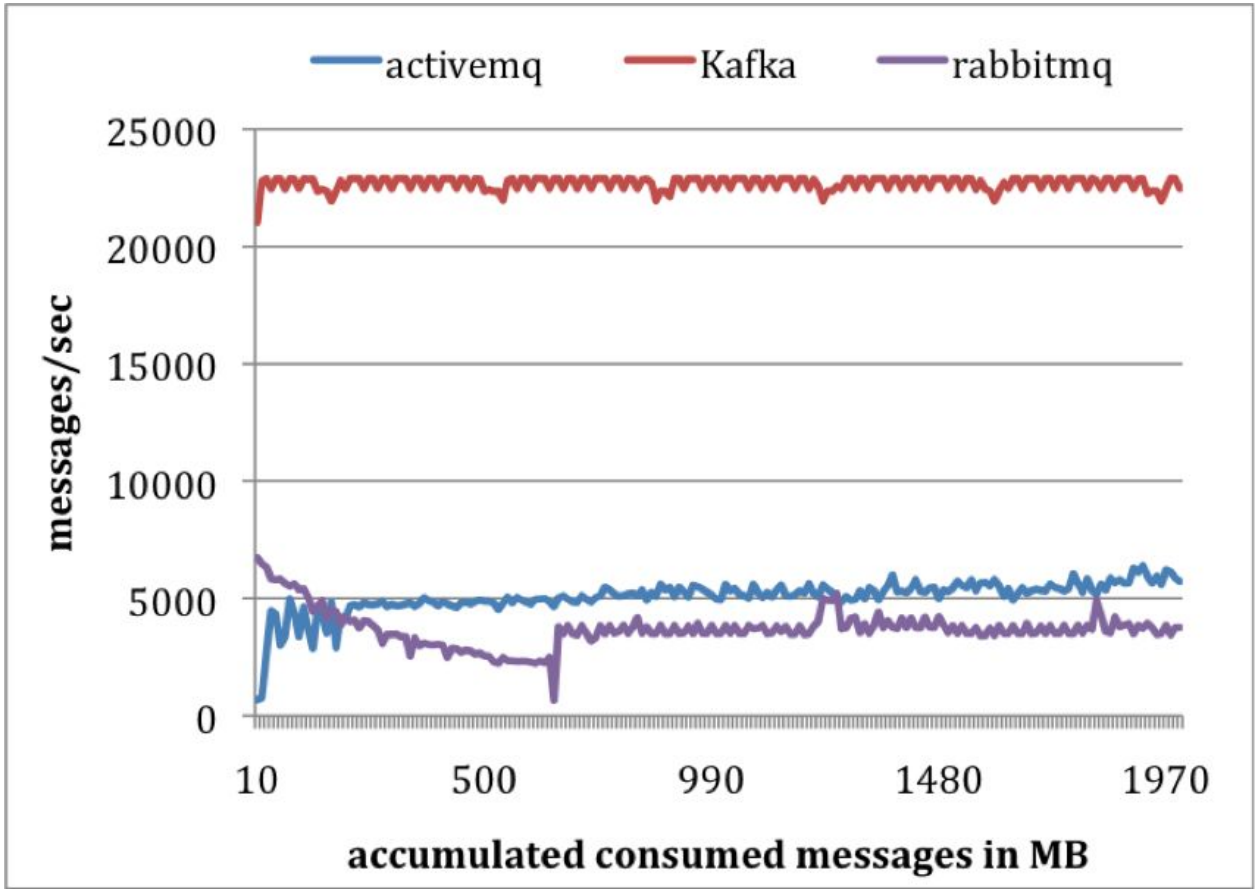
Building event-driven Microservices with Kafka Ecosystem

Event-Driven (Async) Microservice Approach



Building event-driven Microservices with Kafka Ecosystem


activemq
Kafka (batch 50)
Kafka (batch 1)
rabbitmq
500000
400000
300000
200000
100000
0
messages/sec
10
500
990
1480
1970
accumulated produced messages in MB


activemq
Kafka
rabbitmq
25000
20000
15000
10000
5000
0
messages/sec
10
500
990
1480
1970
accumulated consumed messages in MB

References

- A Gentle Introduction to Stream Processing
- Event Sourcing, CQRS, Stream Processing and Apache Kafka: What's the Connection?
- Introduction to Streaming Analytics-Guido Schmutz
- Microservices mit Apache Kafka-Guido Schmutz
- Streaming Visualization-Guido Schmutz