TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Data Acquisition

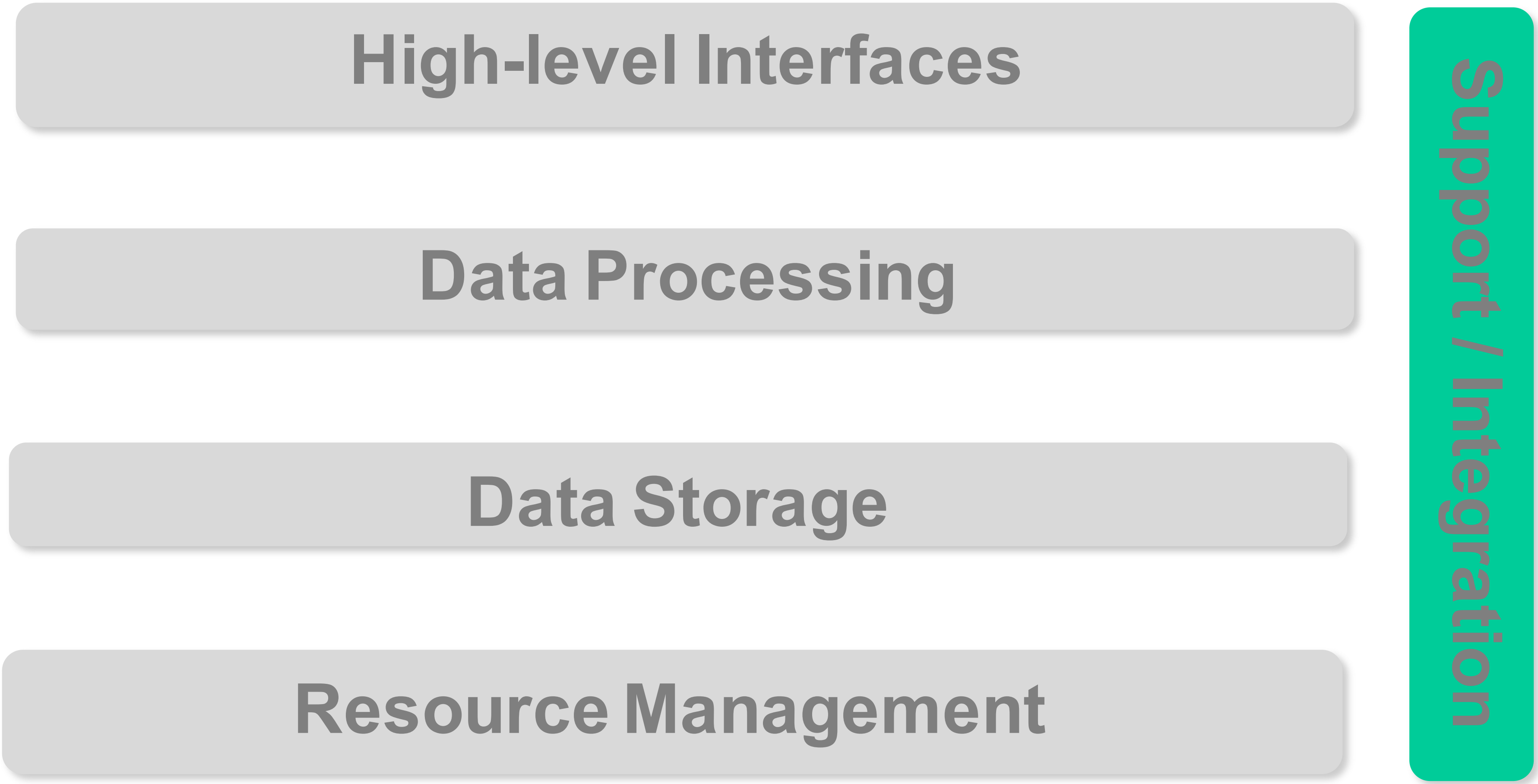
Corso di Sistemi e Architetture per Big Data

A.A. 2018/19

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

The reference Big Data stack



Data acquisition and ingestion

- How to collect data from various data sources and ingest it into a system where it can be stored and later analyzed using batch processing?
  - Distributed file systems (e.g., HDFS), NoSQL data stores (e.g., Hbase), …
- How to connect data sources to stream or in-memory processing systems for immediate use?
- How to perform also some preprocessing, including data transformation or conversion?

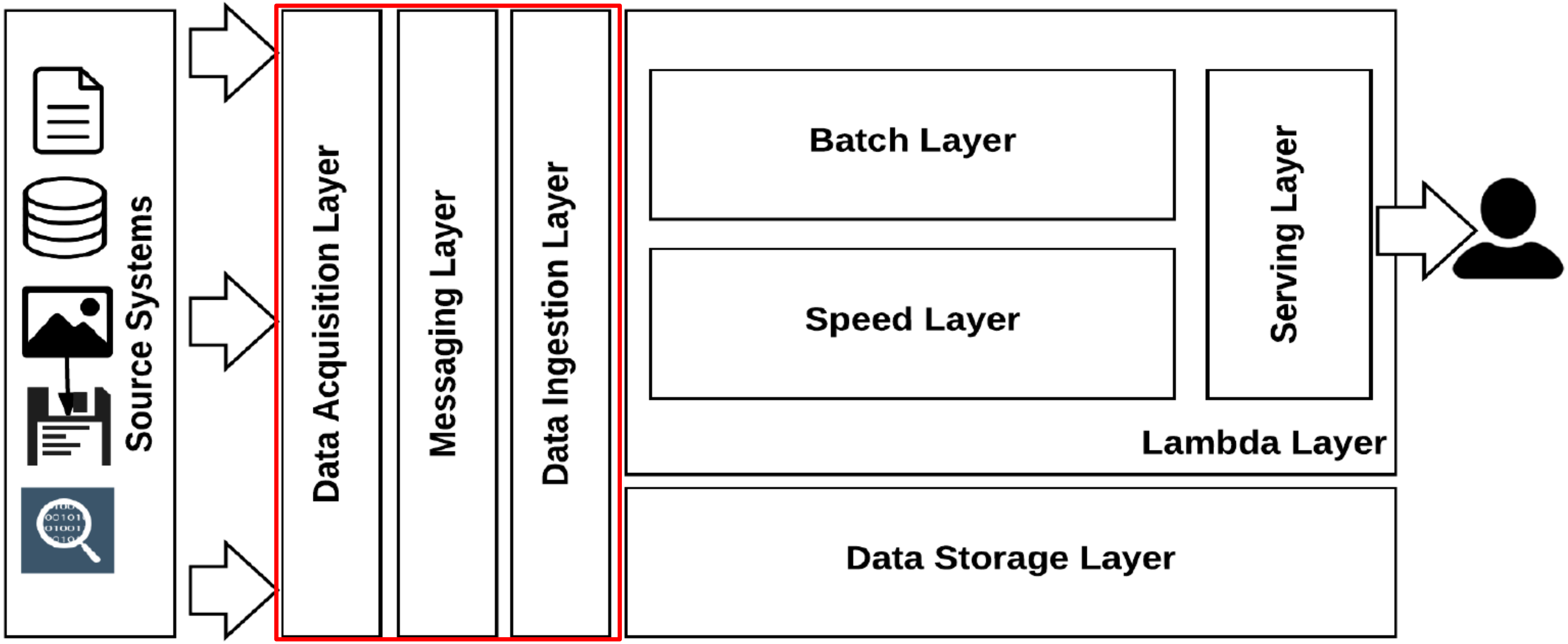
Driving factors

- Source type
  - Batch data sources: files, logs, RDBMS, …
  - Real-time data sources: sensors, IoT systems, social media feeds, stock market feeds, …
- Velocity
  - How fast data is generated?
  - How frequently data varies?
  - Real-time or streaming data require low latency and low overhead
- Ingestion mechanism
  - Depends on data consumers
  - Pull: pub/sub, message queue
  - Push: framework pushes data to sinks

Common requirements

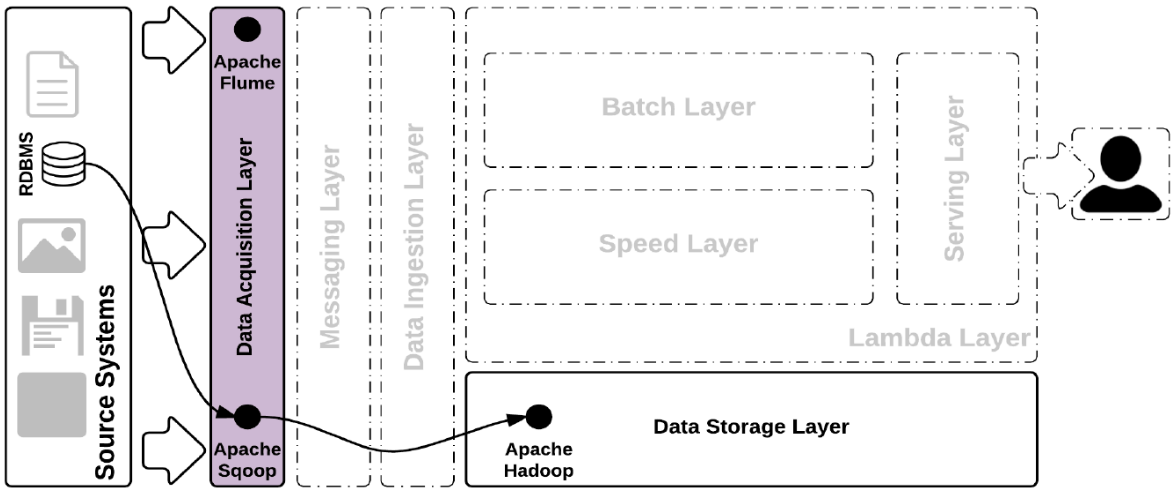
- Ingestion
  - Batch data, streaming data
  - Easy writing to HDFS
- Decoupling
  - Data source should not directly be coupled to analytical backends
- High availability
  - Data ingestion should be available 24x7
  - Data should be buffered (persisted) in case analytical backend is not available
- Scalability
  - Amount of data and number of analytical applications will increase
- Security
  - Authentication and data in motion encryption

A unifying view: Lambda architecture



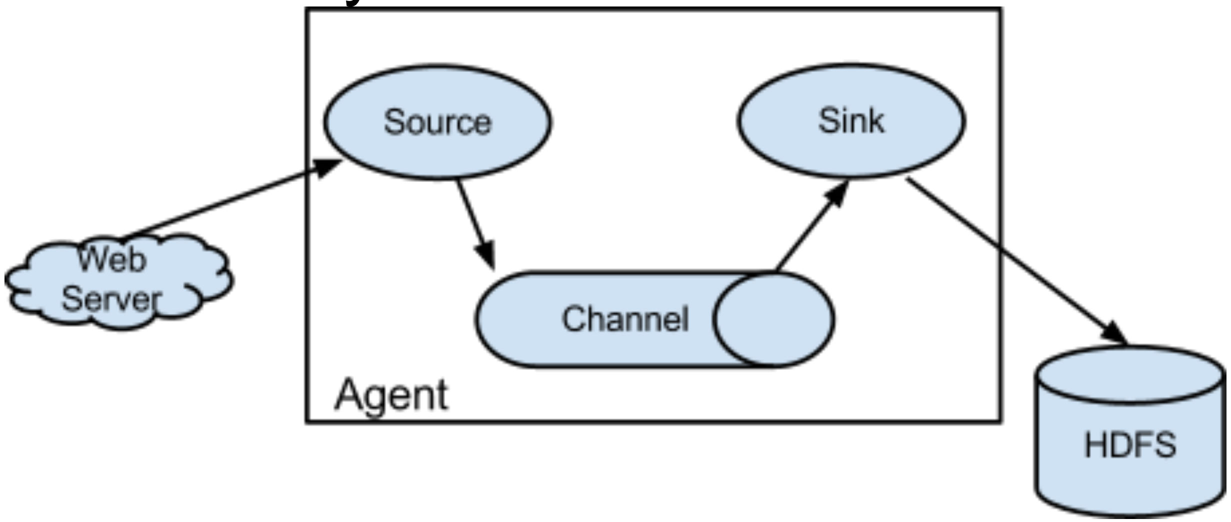
Data acquisition layer

- Allows collecting, aggregating and moving data
- From various sources (server logs, social media, streaming sensor data, …)
- To a data store (distributed file system, NoSQL data store, messaging system)
- We analyze
  - **Apache Flume** for stream data
  - **Apache Sqoop** for batch data

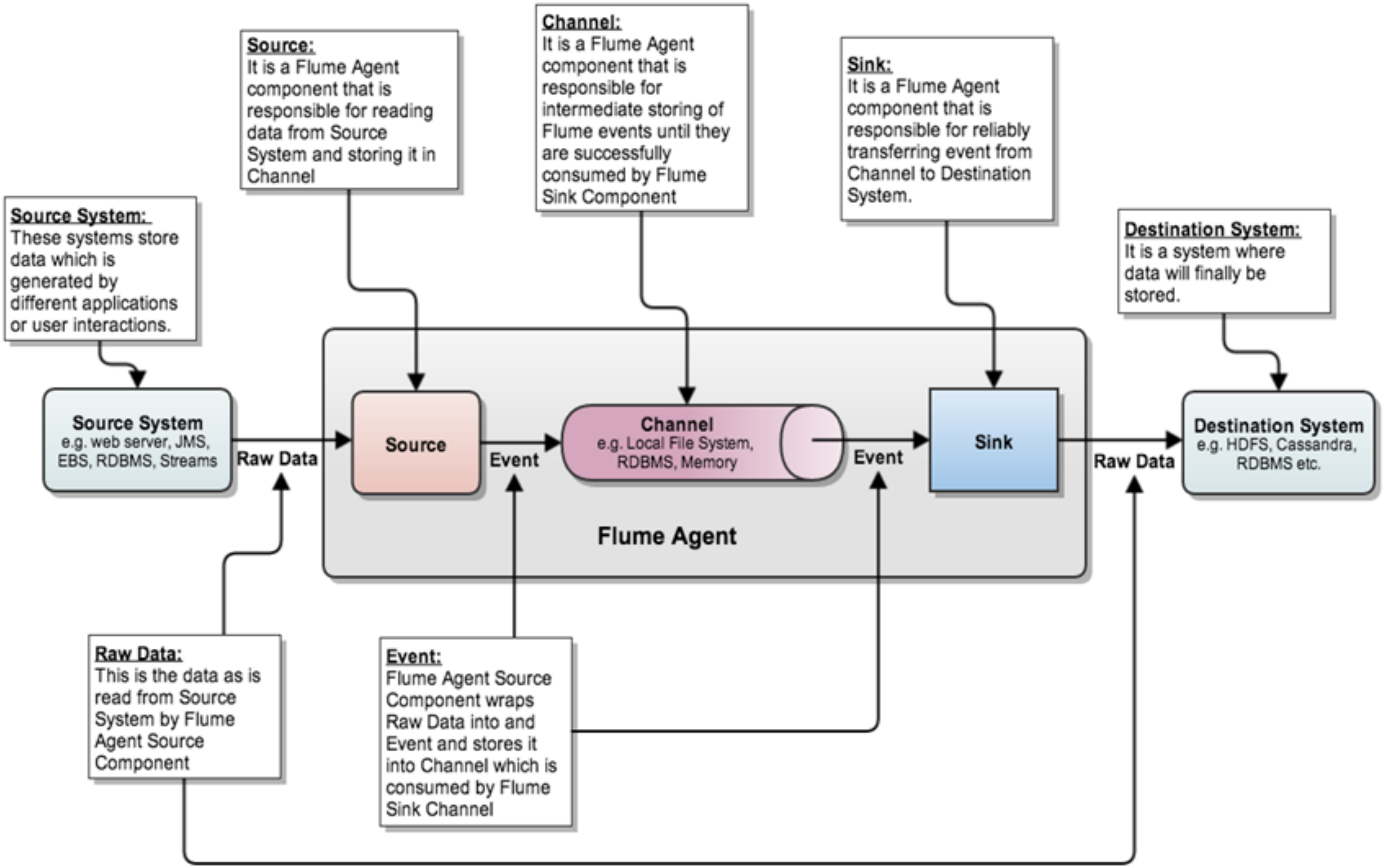


Apache Flume

- Distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of stream data
- Robust and fault tolerant with tunable reliability mechanisms and failover and recovery mechanisms
  - Tunable reliability levels
    - Best effort: “Fast and loose”
    - Guaranteed delivery: “Deliver no matter what”
- Suitable for online analytics



Flume architecture

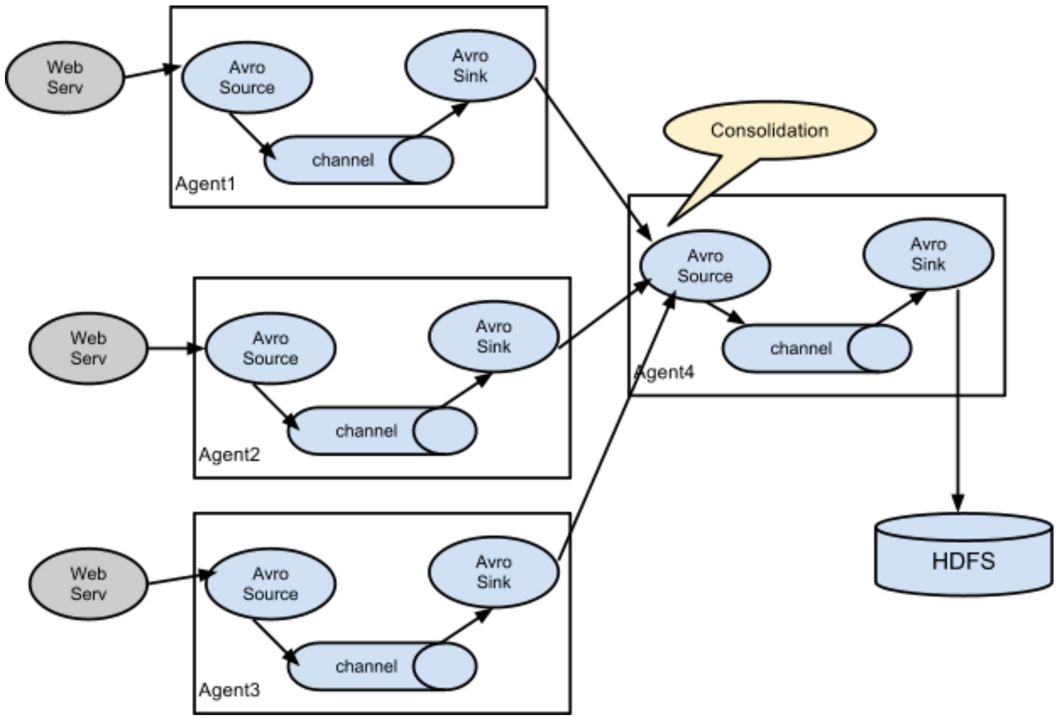


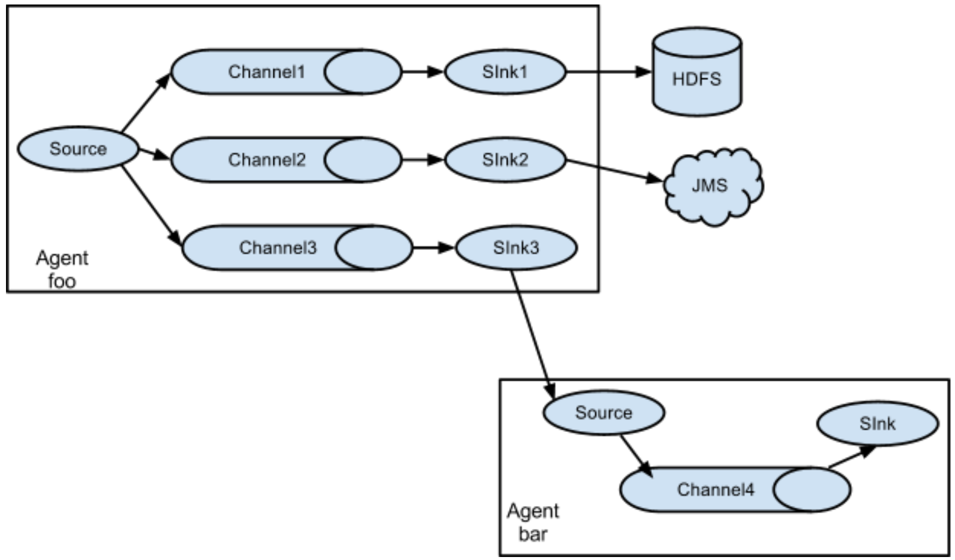
Flume architecture

- Agent: JVM running Flume
  - One per machine
  - Can run many sources, sinks and channels
- Event
  - Basic unit of data that is moved using Flume (e.g., Avro event)
  - Normally ~4KB
- Source
  - Produces data in the form of events
- Channel
  - Connects sources to sinks (like a queue)
  - Implements the reliability semantics
- Sink
  - Removes an event from a channel and forwards it to either to a destination (e.g., HDFS) or to another agent

Flume data flows

- Flume allows a user to build multi-hop flows where events travel through multiple agents before reaching the final destination
- Supports multiplexing the event flow to one or more destinations
- Multiple built-in sources and sinks (e.g., Avro)



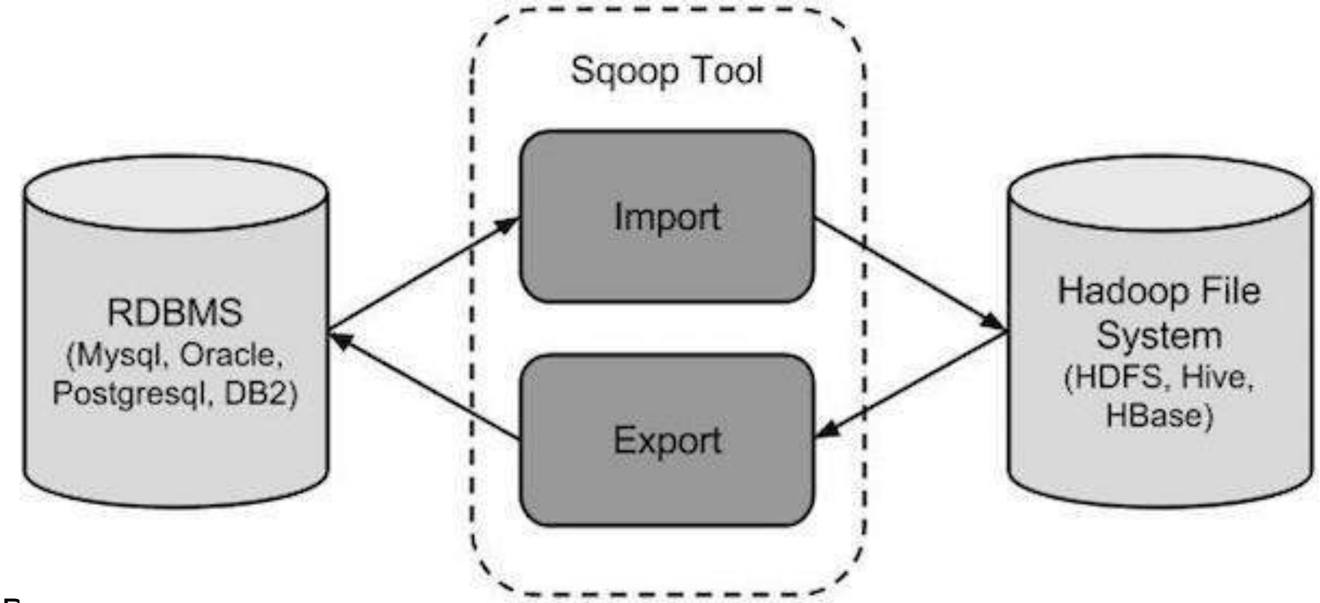


Flume reliability

- Events are staged in a channel on each agent
- Events are then delivered to the next agent or final repository (e.g., HDFS) in the flow
- Events are removed from a channel *only after* they are stored in the channel of next agent or in the final repository
- Transactional approach to guarantee the reliable delivery of events
  – Sources and sinks encapsulate in a transaction the storage/retrieval of the events placed in or provided by a transaction provided by the channel

Apache Sqoop

- A commonly used tool for SQL data transfer to Hadoop
  – SQL to Hadoop = SQOOP
- To import bulk data from structured data stores such as RDBMS into HDFS, HBase or Hive
- Also to export data from HDFS to RDBMS
- Supports a variety of file formats such as Avro



Data serialization formats for Big Data

- Serialization is the process of converting structured data into its raw form
- Some serialization formats you already know
  - JSON
  - XML
- Other serialization formats
  - Protocol buffers
  - Thrift
  - Apache Avro

Apache Avro

- Data serialization format part developed by the Apache Software Foundation
- Key features
  - Compact, fast, binary data format
  - Supports a number of data structures for serialization
  - Neutral to programming language
  - Code generation is optional: data can be read, written, or used in RPCs without having to generate classes or code
  - JSON-based schema segregated from data
    - Data is always accompanied by a schema that permits full processing of that data
- Comparing their performance https://bit.ly/2qrMnOz
  - Avro should not be used from small objects (large serialization and deserialization times)
  - Interesting for very big objects

Apache NiFi

- Powerful and reliable system to process and distribute data over several resources
- Mainly used for data routing and transformation
- Highly configurable
  - Flow specific QoS (loss tolerant vs guaranteed delivery, low latency vs high throughput)
  - Prioritized queueing and flow specific QoS
  - Flow can be modified at runtime
    - Useful for data pre-processing
  - Back pressure
- Ease of use: visual command and control
  - UI based platform where to define the sources from where to collect data, processors for data conversion, destination to store the data
- Multiple NiFi servers can be clustered for scalability

Apache NiFi: use case

- Use NiFi to fetch tweets by means of NiFi's processor 'GetTwitter'
  - It uses Twitter Streaming API for retrieving tweets
- Move data stream to Apache Kafka using NiFi's processor 'PublishKafka'

Messaging layer: Architecture choices

- Message queue (MQ)
  - ActiveMQ
  - RabbitMQ
  - ZeroMQ
  - Amazon SQS

- Publish/subscribe (pub/sub)
  - Kafka
  - NATS http://www.nats.io
  - Apache Pulsar
    - Geo-replication of stored messages
  - Redis

Messaging layer: use cases

- Mainly used in the data processing pipelines for data ingestion or aggregation
- Envisioned mainly to be used at the beginning or end of a data processing pipeline
- Example
  - Incoming data from various sensors: ingest this data into a streaming system for real-time analytics or a distributed file system for batch analytics

Message queue pattern

- Allows for persistent asynchronous communication
  - How can a service and its consumers accommodate isolated failures and avoid unnecessarily locking resources?
- Principles
  - Loose coupling
  - Service statelessness
    - Services minimize resource consumption by deferring the management of state information when necessary

Message queue API

- Basic calls:
  - **put**: non-blocking send
    - Append a message to a specified queue
  - **get**: blocking receive
    - Block until the specified queue is nonempty and remove the first message
    - Variations: allow searching for a specific message in the queue, e.g., using a matching pattern
  - **poll**: non-blocking receive
    - Check a specified queue for message and remove the first
    - Never block
  - **notify**: non-blocking receive
    - Install a handler (callback function) to be automatically called when a message is put into the specified queue

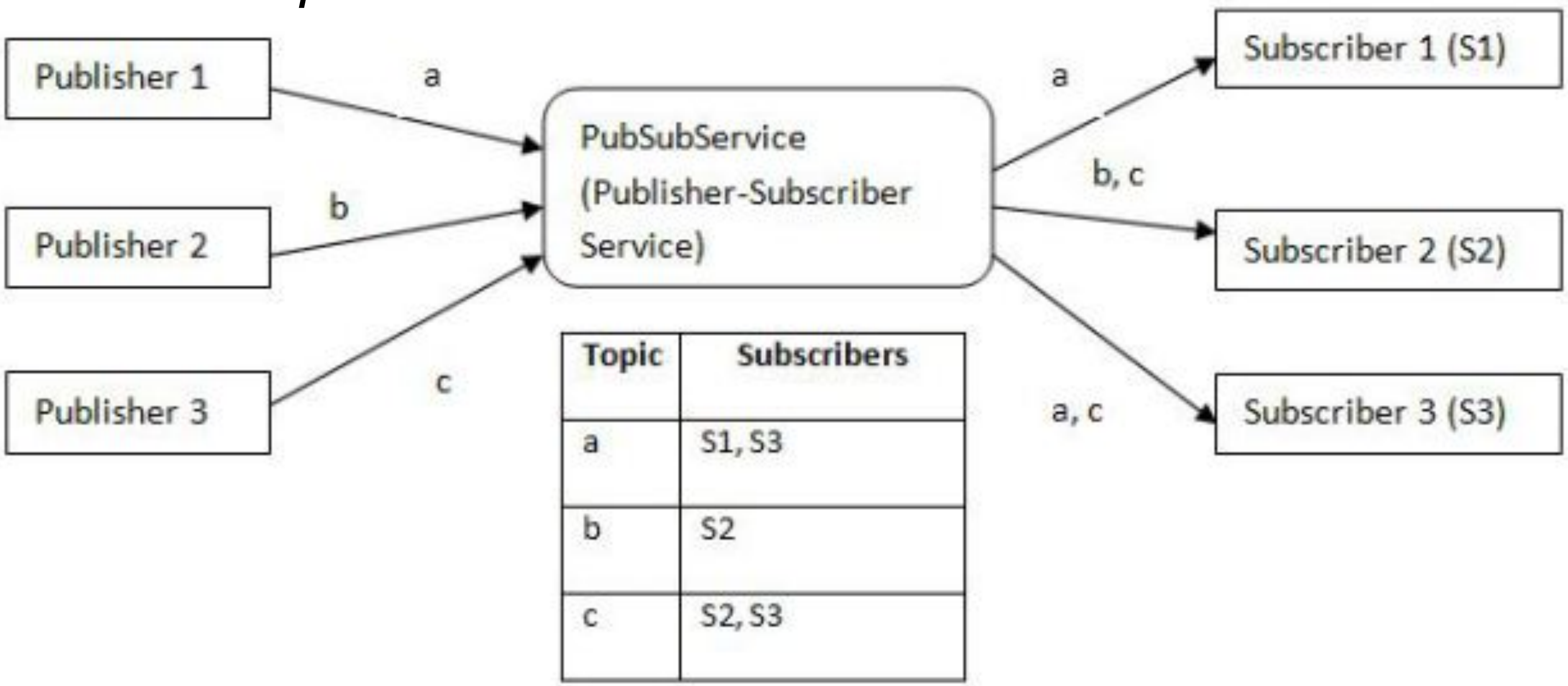
Message queue systems

- Can be used for push-pull messaging
  – Producers push data to queue
  – Consumers pull data from queue

- Message queue systems based on protocols
  – RabbitMQ https://www.rabbitmq.com
    - Implements AMQP and relies on a broker-based architecture
  – ZeroMQ http://zeromq.org
    - High-throughput and lightweight messaging library
    - No persistence
  – Amazon SQS

Publish/subscribe pattern

- Application components can publish asynchronous messages (e.g., event notifications), and/or declare their interest in message topics by issuing a *subscription*



Publish/subscribe pattern

- Multiple consumers can subscribe to topics with or without filters
- Subscriptions are collected by an *event dispatcher* component, responsible for routing events to <u>all</u> matching subscribers
  - For scalability reasons, its implementation is usually distributed
- High degree of decoupling
  - Easy to add and remove components
  - Appropriate for dynamic environments

Publish/subscribe API

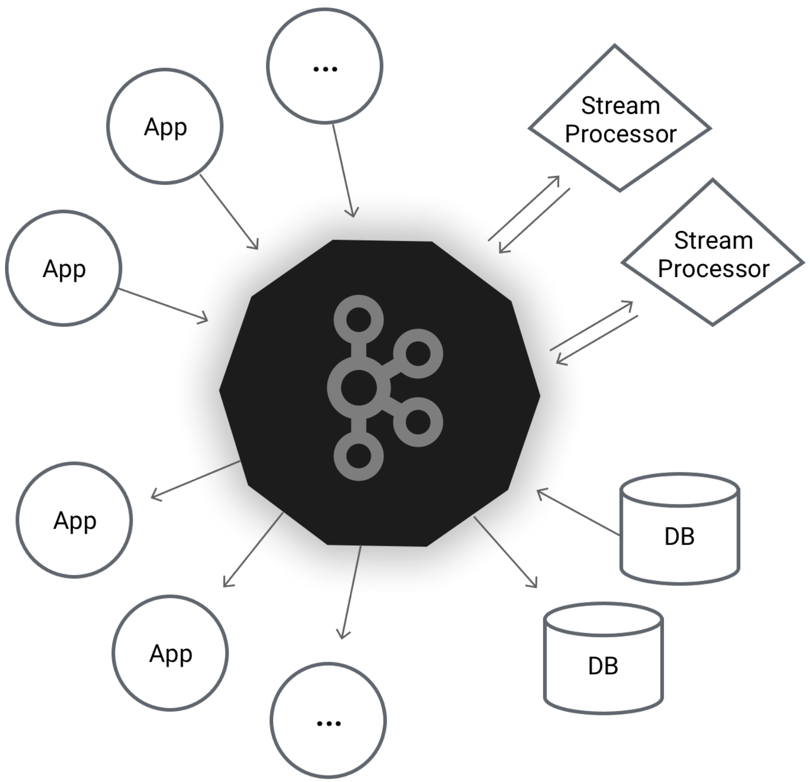
- Basic calls:
  - publish(event): to publish an event
    - Events can be of any data type supported by the given implementation languages and may also contain meta-data
  - subscribe(filter_expr, notify_cb, expiry) → sub_handle: to subscribe to an event
    - Takes a filter expression, a reference to a notify callback for event delivery, and an expiry time for the subscription registration.
    - Returns a subscription handle
  - unsubscribe(sub_handle)
  - notify_cb(sub_handle, event): called by the pub/sub system to deliver a matching event

Pub/sub vs. message queue

- A sibling of message queue pattern but further generalizes it by delivering a message to multiple consumers
  - Message queue: delivers messages to *only one* receiver, i.e., one-to-one communication
  - Pub/sub: delivers messages to *multiple* receivers, i.e., one-to-many communication
- Some frameworks (e.g., RabbitMQ, Kafka, NATS) support both patterns

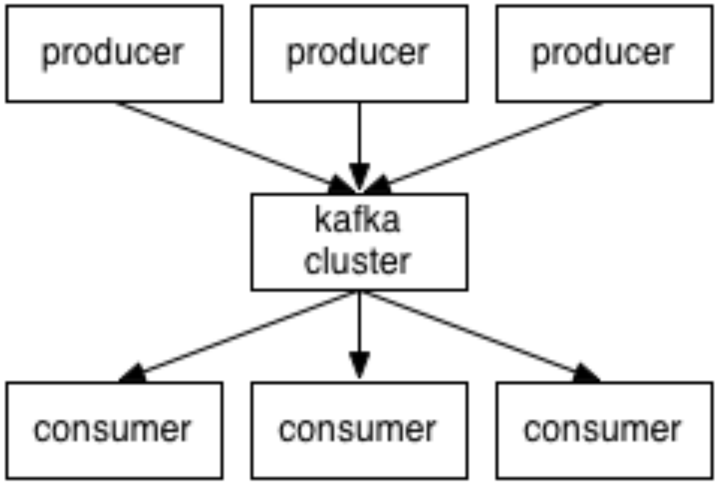
Apache Kafka

- General-purpose, distributed pub/sub system
- Originally developed in 2010 by LinkedIn
- Written in Scala
- Horizontal scalability
- High throughput
  - Thousands of messages per sec
- Fault-tolerant
- Delivery guarantees
  - **At least once**: guarantees no loss, but duplicated messages, possibly out-of-order
  - **Exactly once**: guarantees no-loss and no duplicates, but requires expensive end-to-end 2PC



Kreps et al., "Kafka: A Distributed Messaging System for Log Processing", 2011

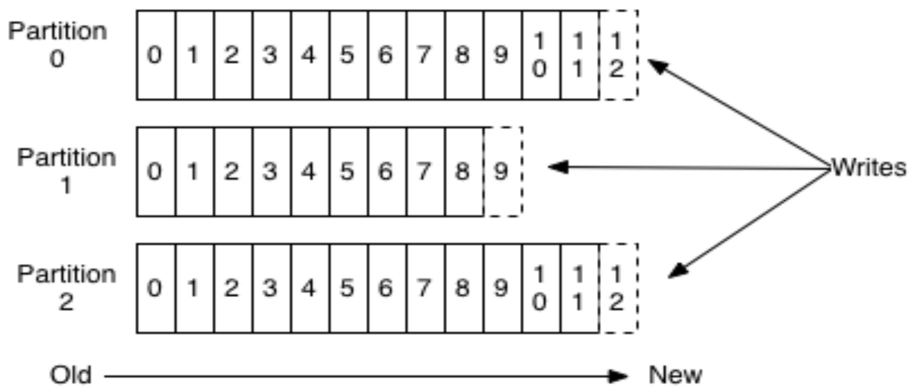
Kafka at a glance



- Kafka maintains feeds of messages in categories called **topics**
- Producers: publish messages to a Kafka topic
- Consumers: subscribe to topics and process the feed of published message
- Kafka cluster: distributed log of data over servers known as **brokers**
  – Brokers rely on Apache Zookeeper for coordination

Kafka: topics

- Topic: category to which the message is published
- For each topic, Kafka cluster maintains a partitioned log
  - Log (data structure!): append-only, totally-ordered sequence of records ordered by time
- Topics are split into a pre-defined number of partitions
  - Partition: unit of parallelism of the topic
- Each partition is replicated in multiple brokers with some replication factor

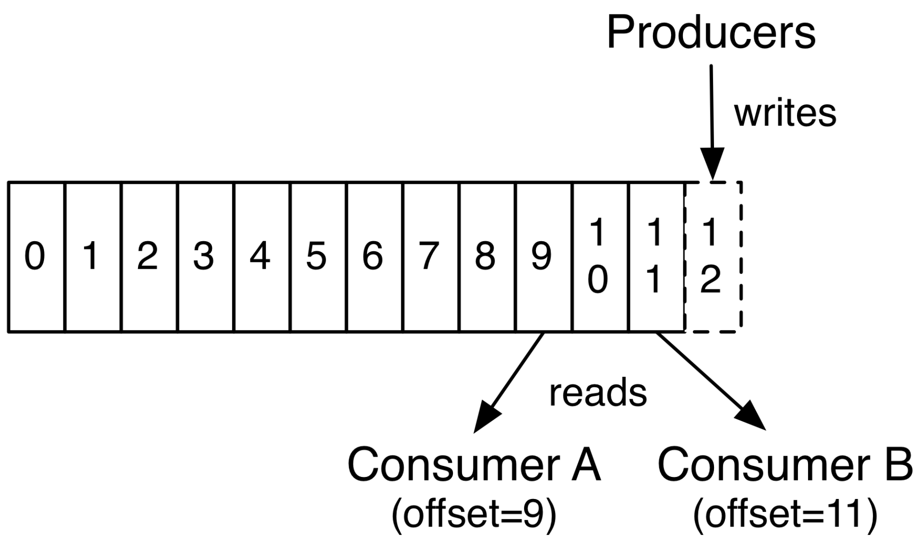


- CLI command to create a topic with a single partition and one replica

```
> bin/kafka-topics.sh --create --zookeeper
localhost:2181 --replication-factor 1 --partitions 1 --
topic test
```

Kafka: partitions

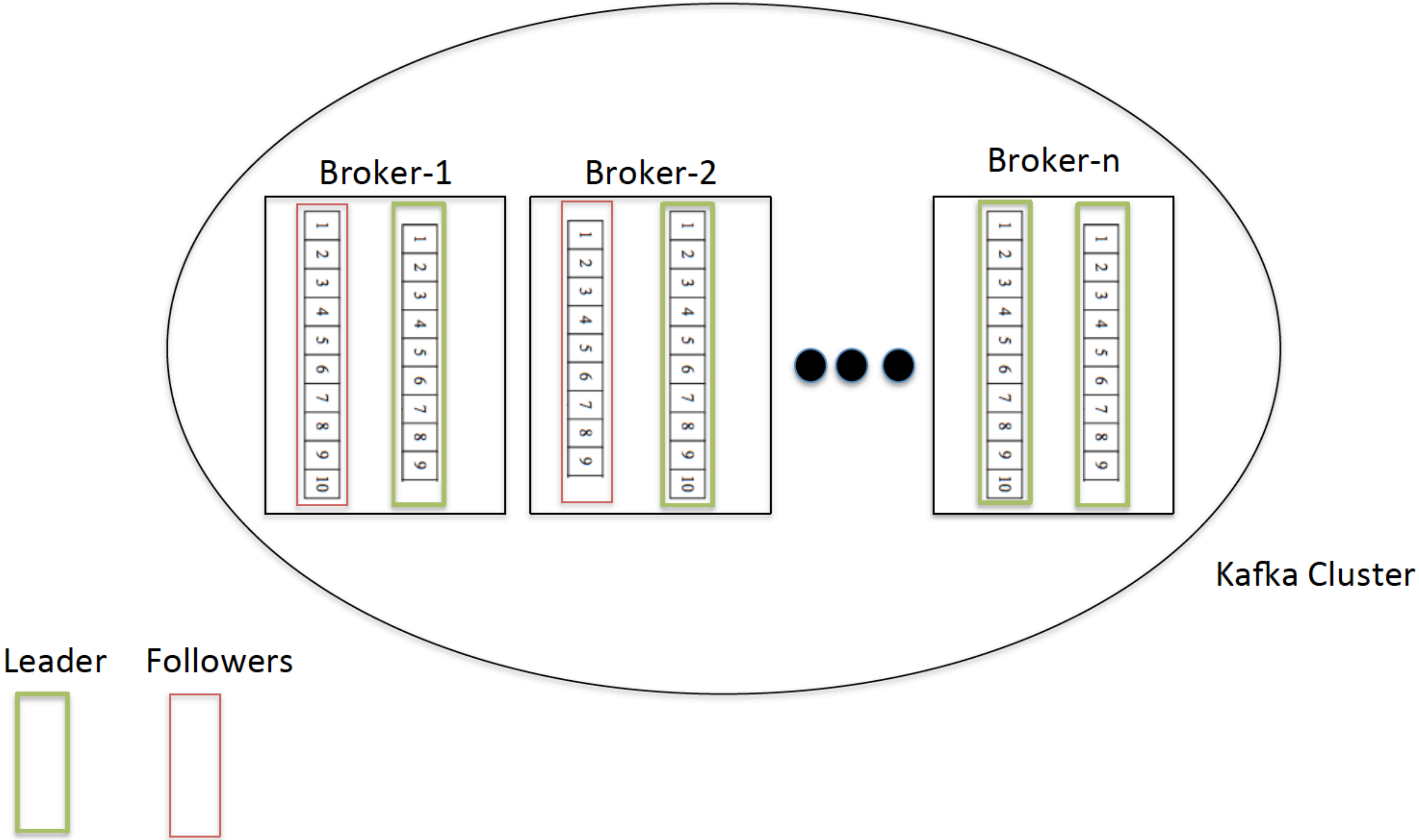
- Producers publish their records to partitions of a topic (round-robin or partitioned by keys), and consumers consume the published records of that topic
- Each partition is an ordered, numbered, immutable sequence of records that is continually appended to
  - Like a commit log
- Each record is associated with a monotonically increasing sequence number, called offset



Kafka: partitions

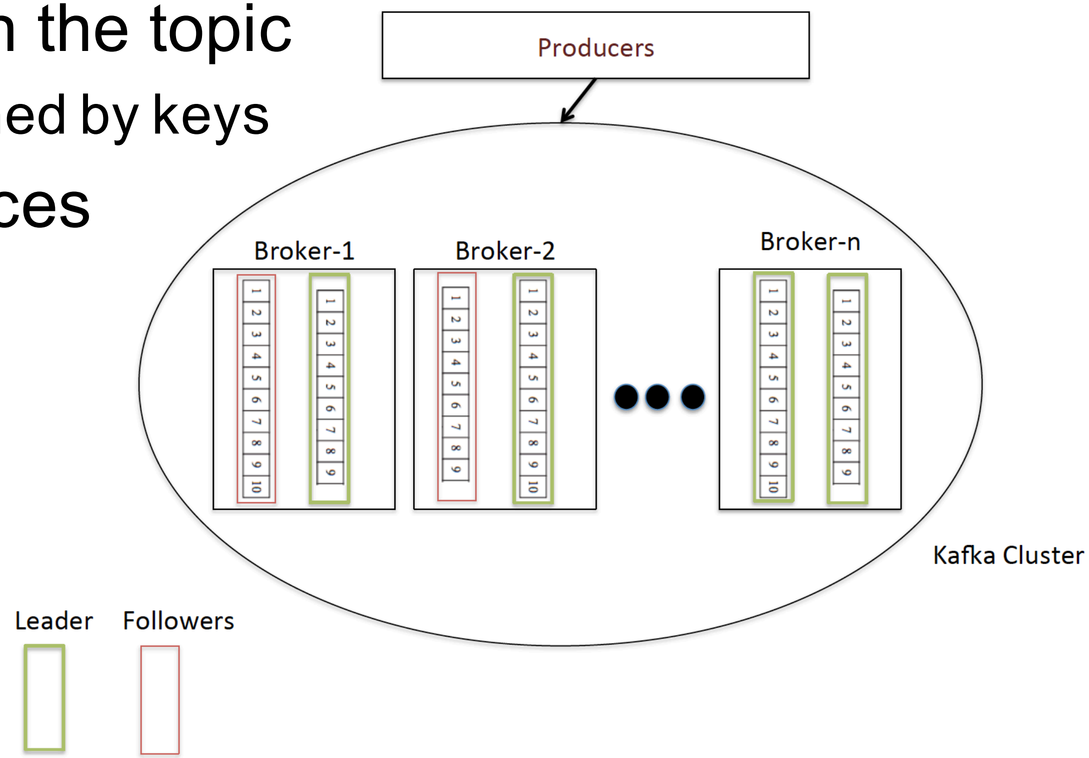
- Partitions are distributed across brokers for scalability
- Each partition is replicated for fault tolerance across a configurable number of brokers
- Each partition has one leader broker and 0 or more followers
- The leader handles read and write requests
  - Read from leader
  - Write to leader
- A follower replicates the leader and acts as a backup
- Each broker is a leader for some of it partitions and a follower for others to load balance
- ZooKeeper is used to keep the brokers consistent

Kafka: partitions



Kafka: producers

- Publish data to topics of their choice
- Also responsible for choosing which record to assign to which partition within the topic
  – Round-robin or partitioned by keys
- Producers = data sources



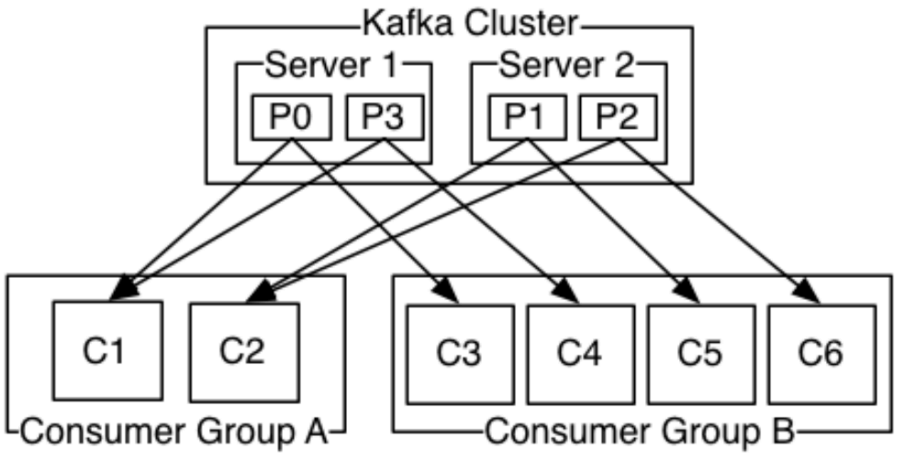
- To run the producer

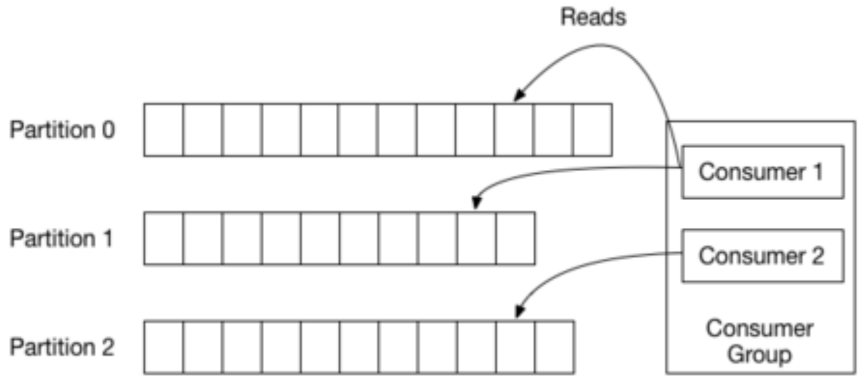
```
> bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic test

This is a message

This is another message
```

Kafka: consumers





- Consumer Group: set of consumers sharing a common group ID
  - A Consumer Group maps to a logical subscriber
  - Each group consists of multiple consumers for scalability and fault tolerance
- Consumers use the offset to track which messages have been consumed
  - Messages can be replayed using the offset
- To run the consumer

```
> bin/kafka-console-consumer.sh --bootstrap-server
localhost:9092 --topic test --from-beginning
```

Kafka: design choice for consumers

- Push vs. pull model for consumers

- Push model
  - Challenging for the broker to deal with different consumers as it controls the rate at which data is transferred
  - Need to decide whether to send a message immediately or accumulate more data and send

- **Pull** model
  - Pros: scalability, flexibility (different consumers can have diverse needs and capabilities)
  - Cons: in case broker has no data, consumers may end up busy waiting for data to arrive

Kafka: ordering guarantees

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent
- Consumer sees records in the order they are stored in the log
- Strong guarantees about ordering only within a partition
  – Total order over messages within a partition, but Kafka cannot preserve order between different partitions in a topic
- Per-partition ordering combined with the ability to partition data by key is sufficient for most applications

Kafka: ZooKeeper

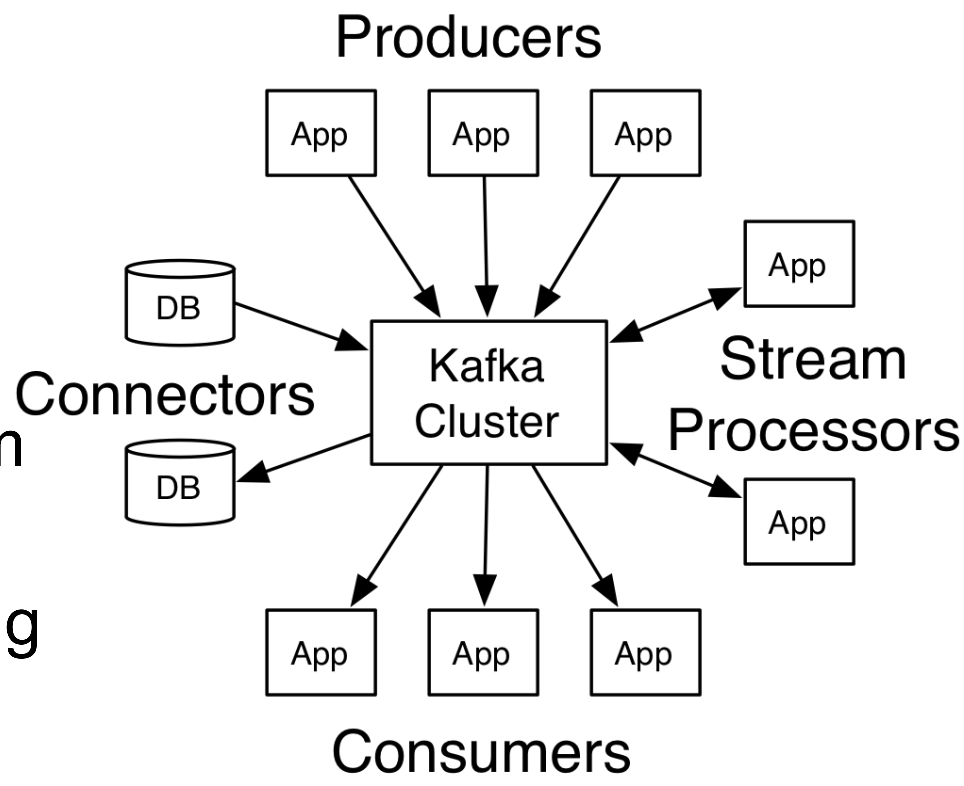
- Kafka uses ZooKeeper to coordinate among the producers, consumers and brokers
- ZooKeeper stores metadata
  - List of brokers
  - List of consumers and their offsets
  - List of producers
- ZooKeeper runs several algorithms for coordination between consumers and brokers
  - Consumer registration algorithm
  - Consumer rebalancing algorithm
    - Allows all the consumers in a group to come into consensus on which consumer is consuming which partitions

Kafka: fault tolerance

- Replicates partitions for fault tolerance
- Kafka makes a message available for consumption only after all the followers acknowledge to the leader a successful write
  - Implies that a message may not be immediately available for consumption
- Kafka retains messages for a configured period of time
  - Messages can be “replayed” in case a consumer fails

Kafka APIs

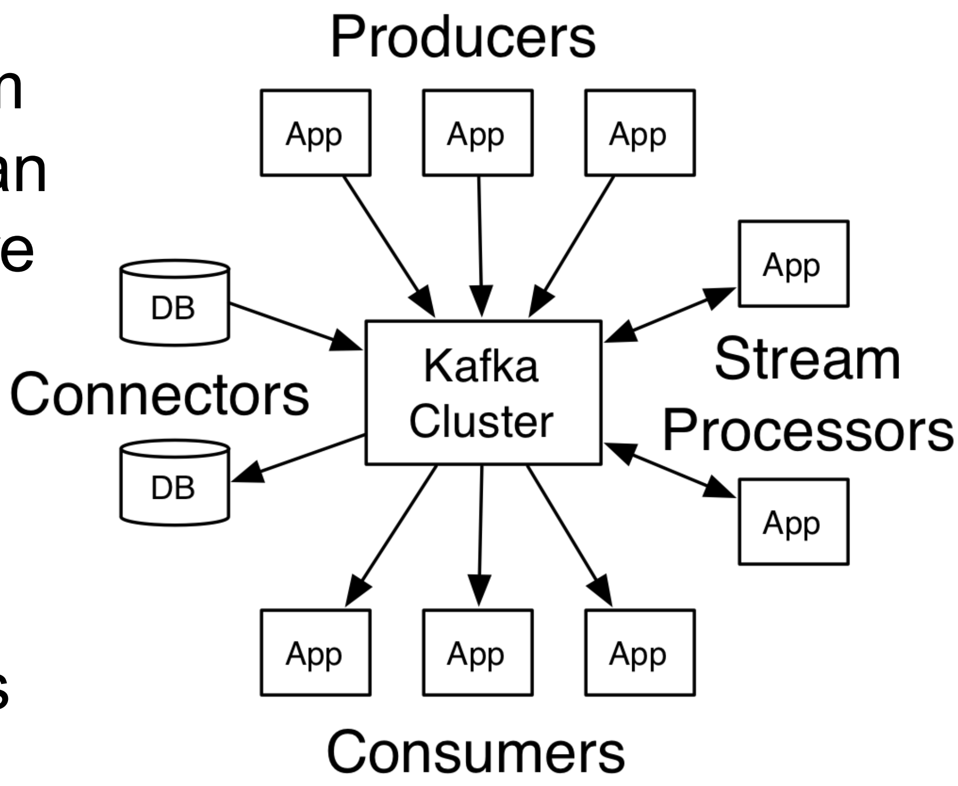
- Four core APIs
- Producer API: allows app to publish streams of records to one or more Kafka topics
- Consumer API: allows app to subscribe to one or more topics and process the stream of records produced to them
- Connector API: allows building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems so to move large collections of data into and out of Kafka



Kafka APIs

- Streams API: allows app to act as a stream processor, transforming an input stream from one or more topics to an output stream to one or more output topics

- Can use Kafka Streams to process data in pipelines consisting of multiple stages



Kafka clients

- JVM internal client
- Plus rich ecosystem of clients, among which:
  - Sarama: Go library
    https://shopify.github.io/sarama/
  - Python library
    https://github.com/confluentinc/confluent-kafka-python/
  - NodeJS client
    https://github.com/Blizzard/node-rdkafka

Performance comparison: Kafka versus RabbitMQ

- Both guarantee millisecond-level low-latency

	mean	max
with and without replication	1–4 ms	2–17 ms

(a) RabbitMQ

	50 percentile	99.9 percentile
without replication	1 ms	15 ms
with replication	1 ms	30 ms

(b) Kafka

  – At least once delivery guarantee more expensive on Kafka (latency almost doubles)
- Replication has a drastic impact on the performance of both
  – Performance reduced by 50% (RabbitMQ) and 75% (Kafka)
- Kafka is best suited as scalable ingestion system
- The two systems can be chained

Dobbelaere and Esmaili, "Kafka versus RabbitMQ", ACM DEBS 2017

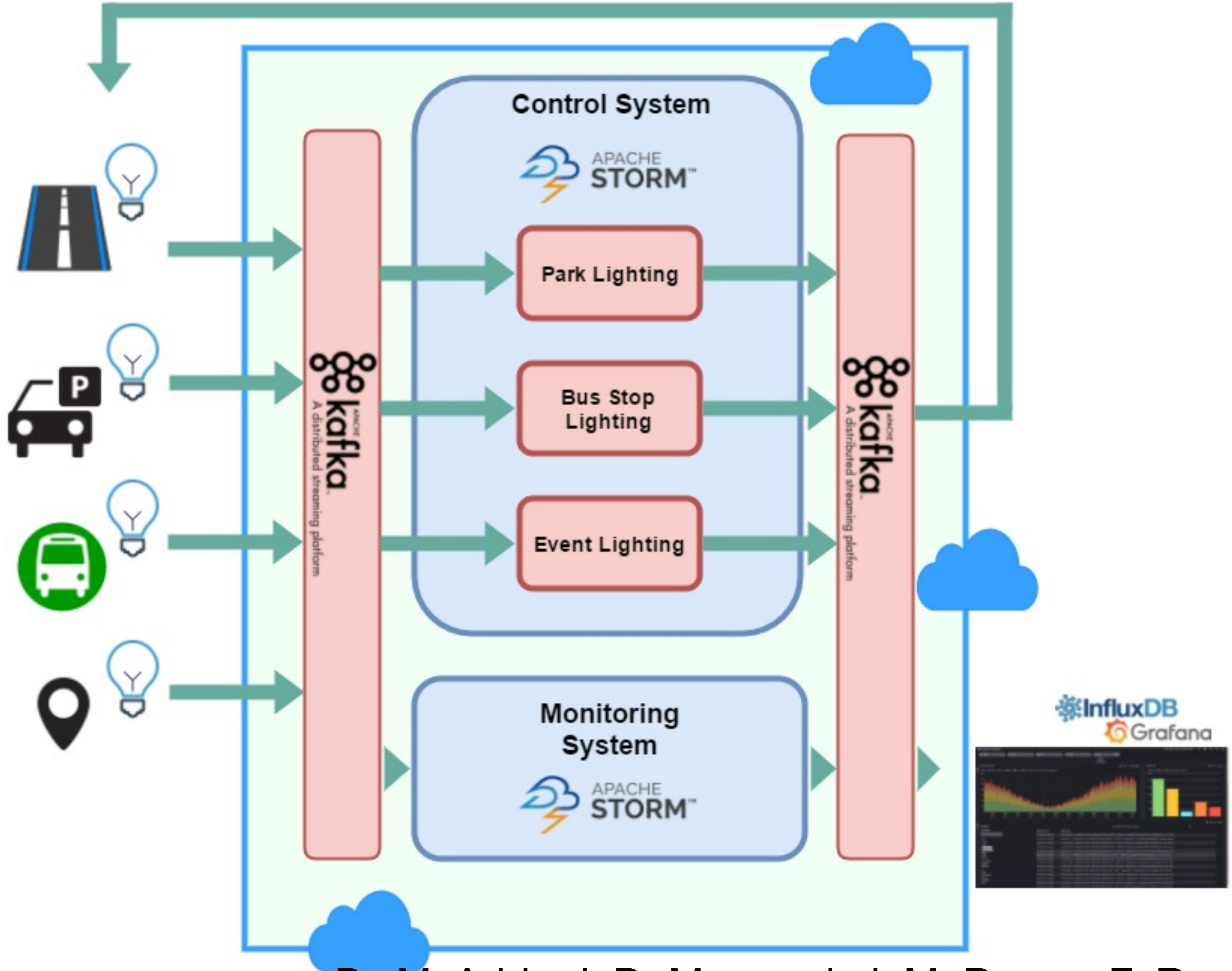
Kafka: limitations

- No complete set of monitoring and management tools
- No support for wildcard topic selection
- No geo-replication

Kafka: evolution

- Kafka as a streaming platform
- In upcoming hands-on lesson

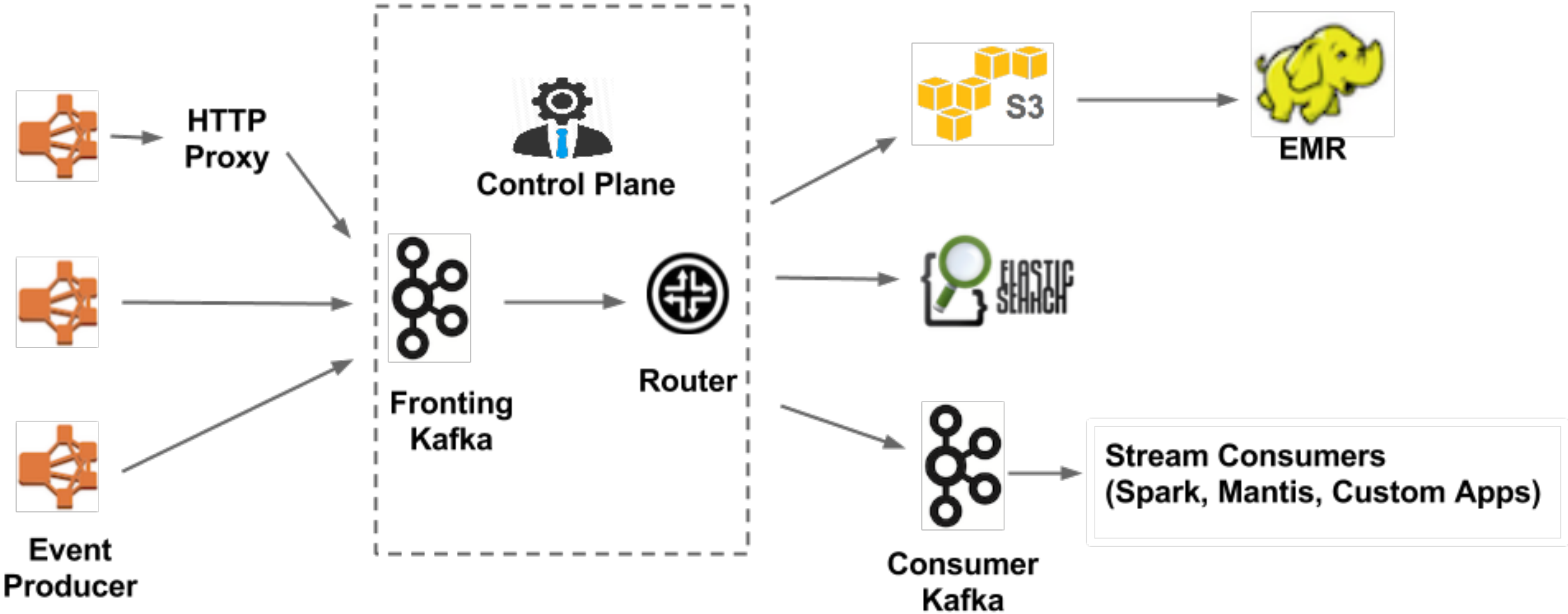
Kafka @ CINI Smart City Challenge '17



By M. Adriani, D. Magnanimi, M. Ponza, F. Rossi

Kafka @ Netflix

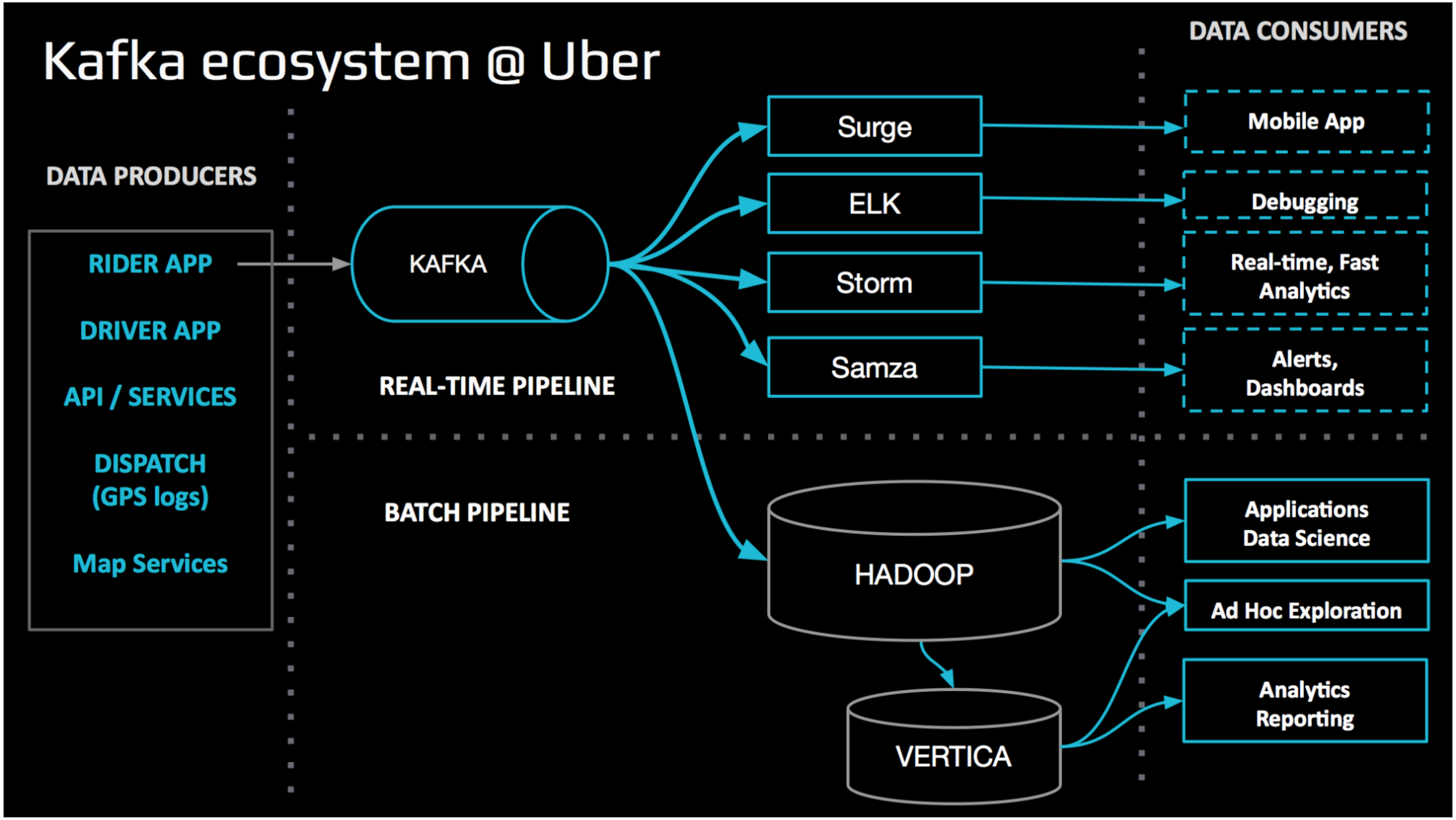
- Netflix uses Kafka for data collection and buffering so that it can be used by downstream systems



http://techblog.netflix.com/2016/04/kafka-inside-keystone-pipeline.html

Kafka @ Uber

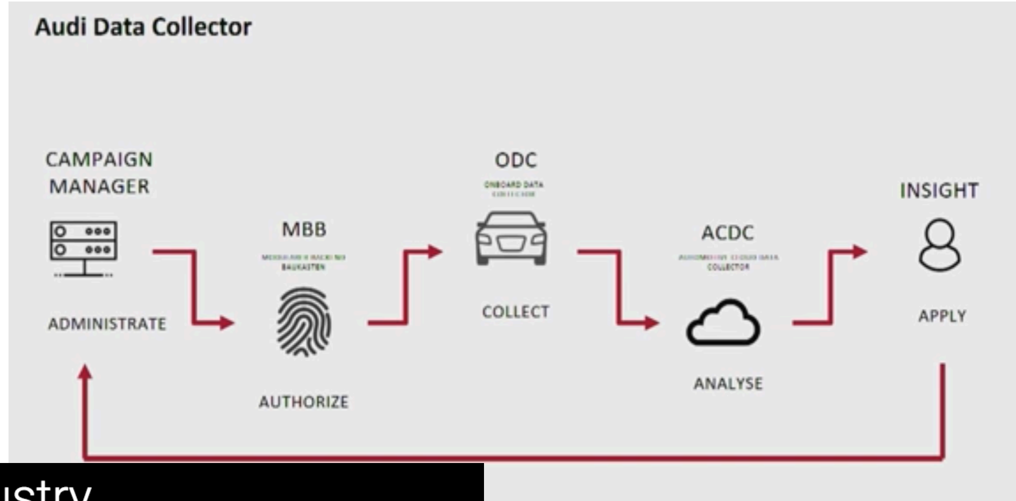
- Uber uses Kafka for real-time business driven decisions

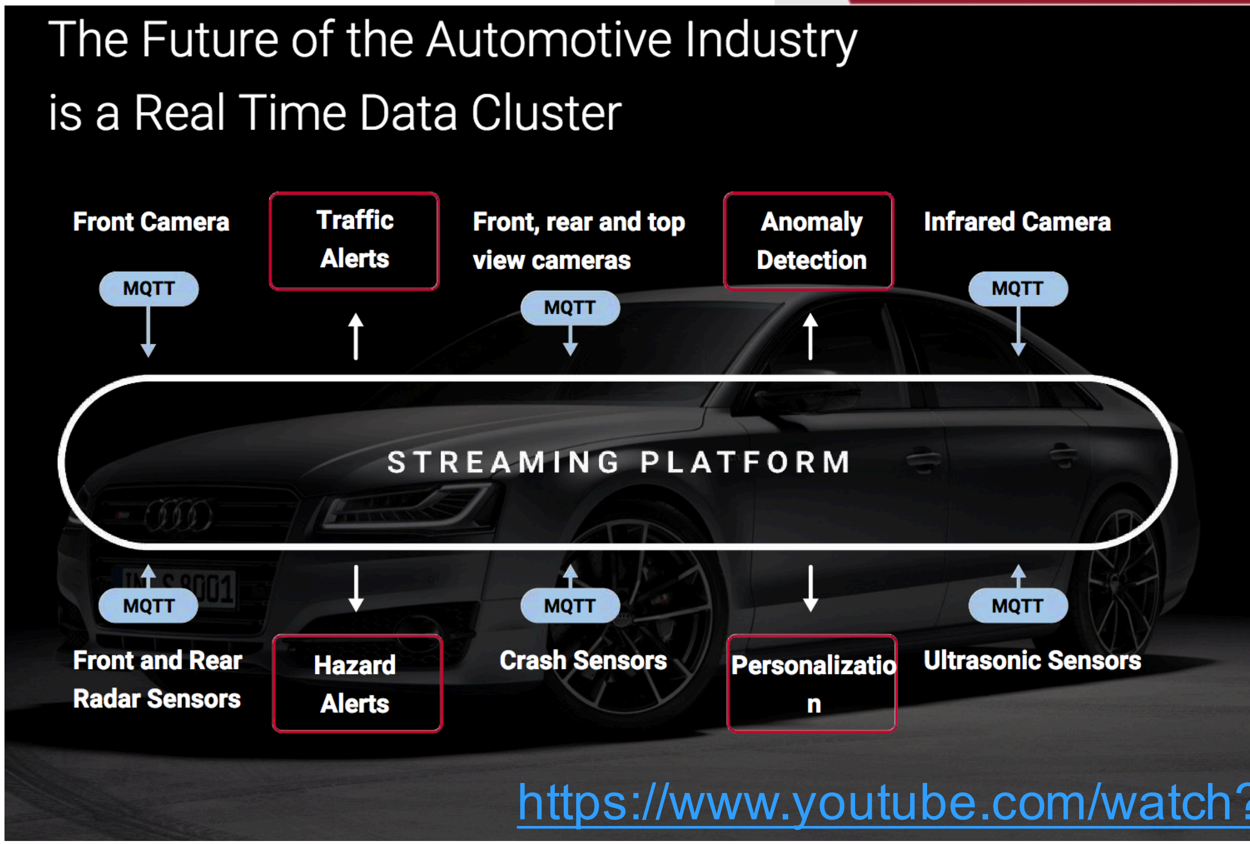


https://eng.uber.com/ureplicator/

Kafka @ Audi

- Audi uses Kafka for real-time data processing





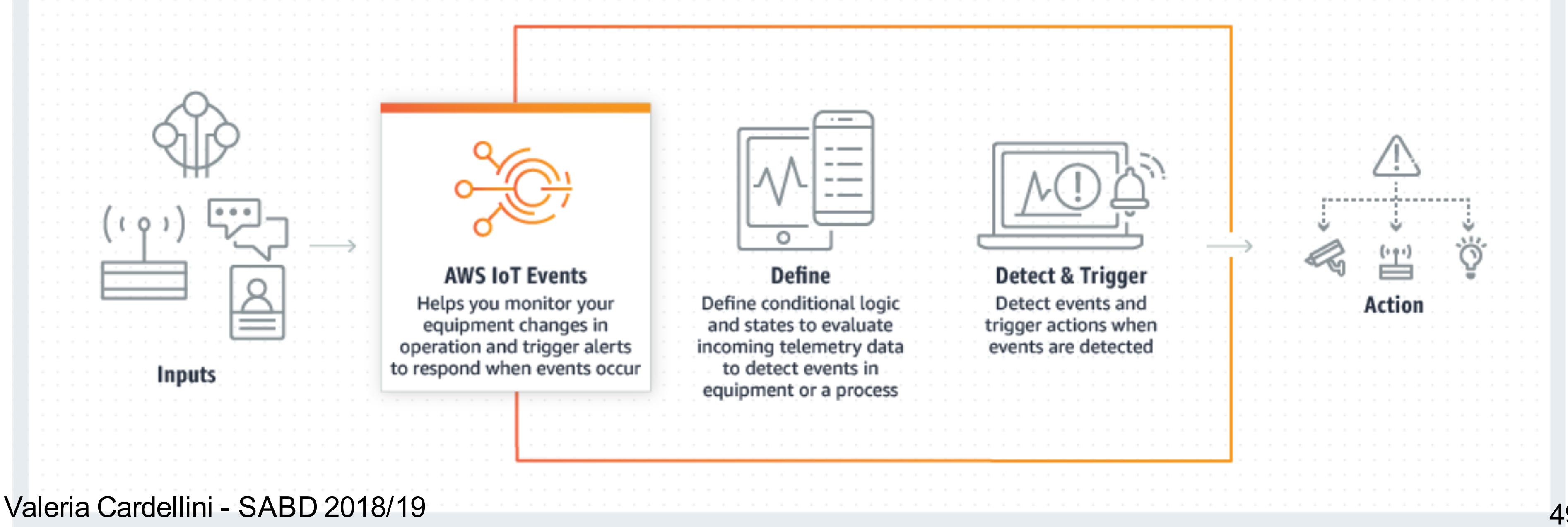
https://www.youtube.com/watch?v=yGLKi3TMJv8

Cloud services for IoT data ingestion and analysis

- Let’s consider AWS cloud services devoted to Internet of Things data ingestion and analysis
  - AWS IoT Events
  - AWS IoT Core
  - AWS IoT Analytics

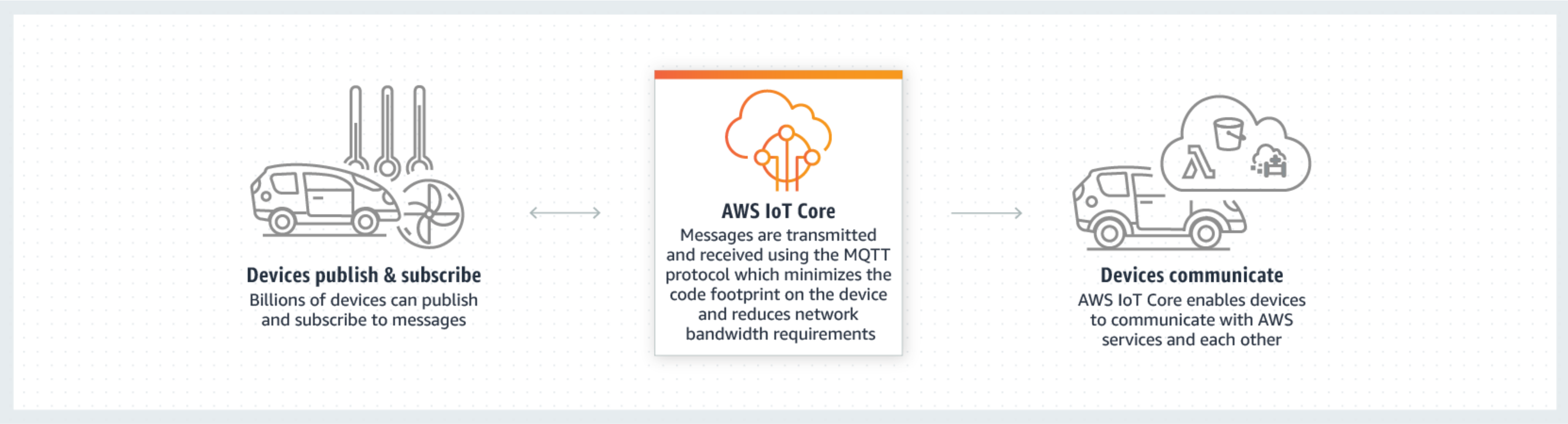
AWS IoT Events

- IoT service to detect and respond to events from IoT sensors and applications
  - Select the data sources to ingest, define the logic for each event using if-then-else statements, and select the alert or custom action to trigger when an event occurs
  - Integrated with other services, such as AWS IoT Core and AWS IoT Analytics, to enable detection and insights into events



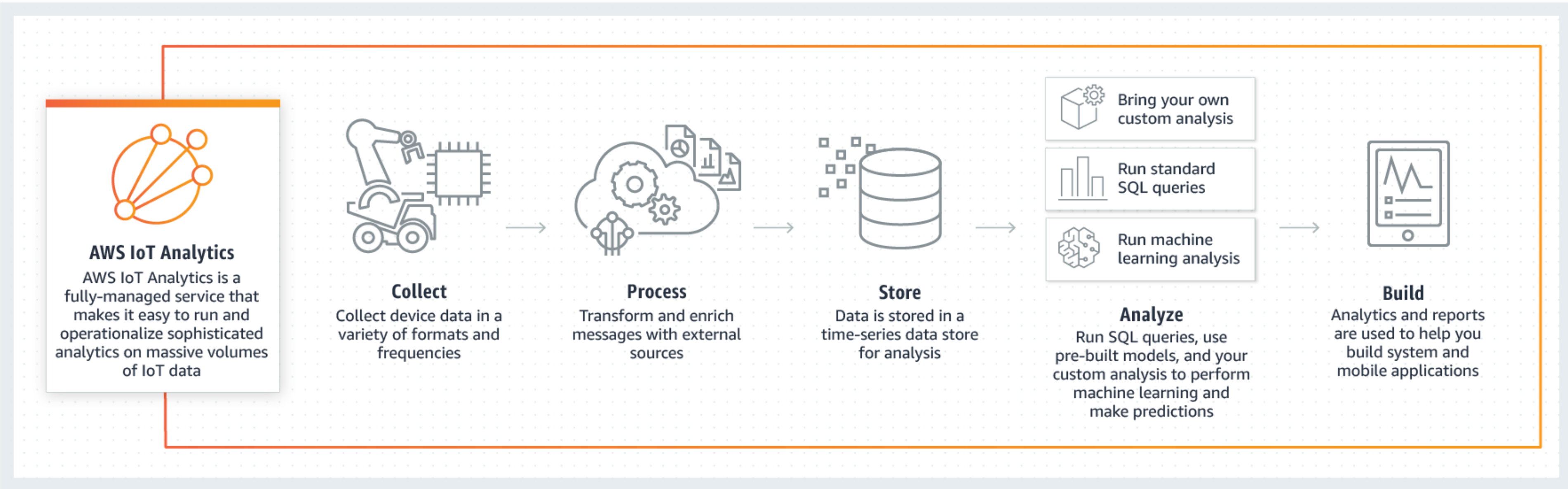
AWS IoT Core

- Managed cloud service that lets connected devices interact with cloud applications and other devices



AWS IoT Analytics

- Fully-managed Cloud service to run analytics on massive volumes of IoT data
- Filters, transforms, and enriches IoT data before storing it in a time-series data store for analysis



References

- Apache Flume documentation, http://bit.ly/2qE5QK7
- Apache NiFi documentation, https://nifi.apache.org/docs.html
- Kreps et al., "Kafka: A Distributed Messaging System for Log Processing", *NetDB 2011*. http://bit.ly/2oxpael
- Apache Kafka documentation, http://bit.ly/2ozEY0m