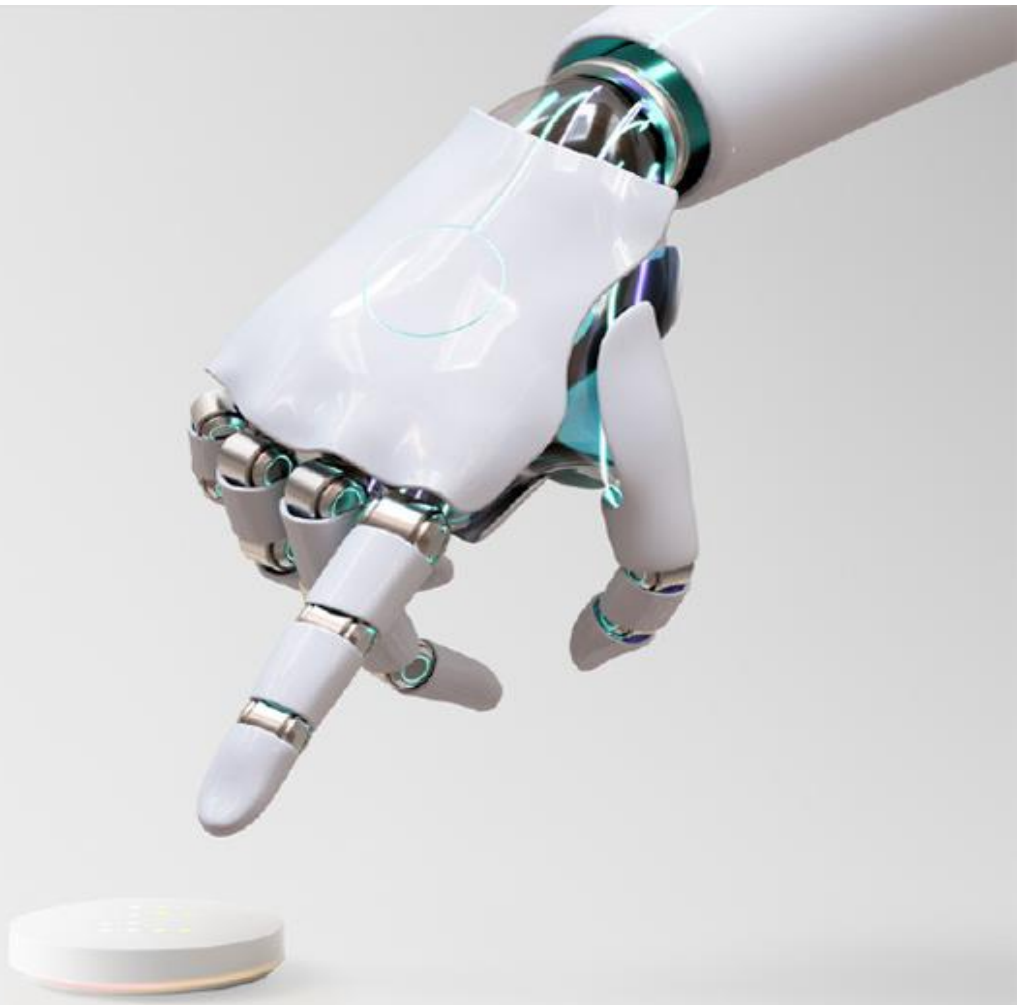
SMART INDUSTRY®
EXCELLENT FUTURE CONDITION

APACHE KAFKA & FLINK
Real-time fleet monitoring



Palermo, 18.12.2023

Speaker: **Nicolò Paganin**

SMART INDUSTRY – engineering services company and industrial consulting

HEADQUARTERS

SMART INDUSTRY **legal and operational headquarter in Ozzano dell'Emilia (BO)**
operational headquarters in Italy and abroad:

- Trento (TN)
- Torino (TO)
- Amaro (UD)
- Schio (VI)
- Mestre (VE)
- Cernusco sul Naviglio (MI)
- Modena (MO)
- Ozzano dell'Emilia (BO)
- Ancona (AN)
- Roma (RM)
- Napoli (NA)
- Palermo (PA)

- Albania
- Bulgaria
- Romania
- Serbia

SMART INDUSTRY®
EXCELLENT FUTURE CONDITION

ENGINEERING SECTORS

SMART INDUSTRY®
EXCELLENT FUTURE CONDITION

STUDENTS OPPORTUNITIES

INTERNSHIPS

DEGREE THESES

WORKSHOPS

SMART INDUSTRY®
EXCELLENT FUTURE CONDITION

Oncode

Ci occupiamo di **progettazione e sviluppo software**
In vari ambiti, con varie competenze, in questo caso parleremo di
Soluzioni di gestione di grandi moli di dati e stream processing

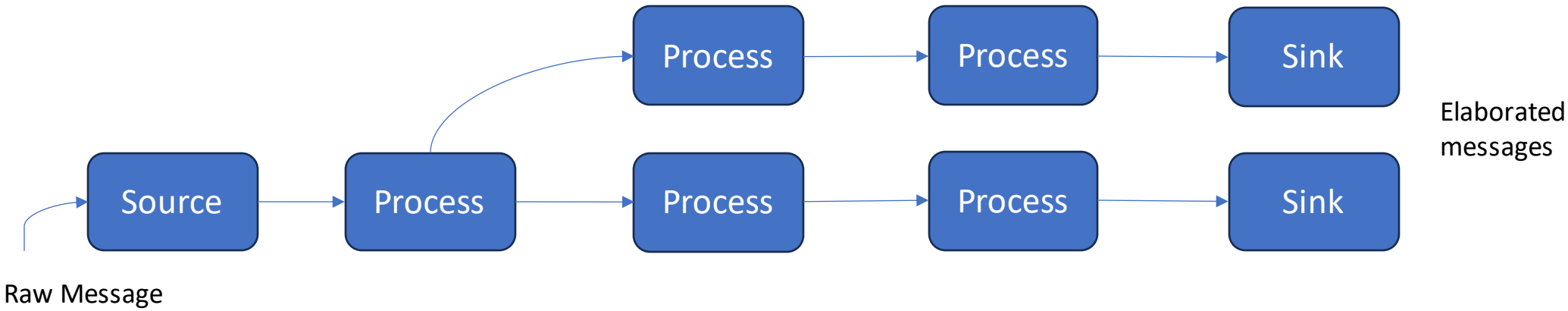
Headquarter: Piazza Maestri del Lavoro 7, Cernusco sul Naviglio (MI)
Sito web: www.oncode.it

Speaker: Ing. Nicolò Paganin
Managing Partner di Oncode – Direttore BU Enterprise Servizi IT

Speaker: Andrea Grazioli
Technical Team Leader BU Enterprise Servizi IT

Stream Processing

Stream processing is the elaboration of data in a flow



- Stateless stream processing: elaboration of date does not have a state, so following processing is independent from the past

- Stateful stream processing: elaboration of date has a related state, the state can be changed by current elaboration (es. mean calculation requires a state)

What we'll talk about

Learn about **Apache Kafka (stateless stream processing)** and **Flink (stateful)** and their usage

View a practical application in a car-sharing architecture

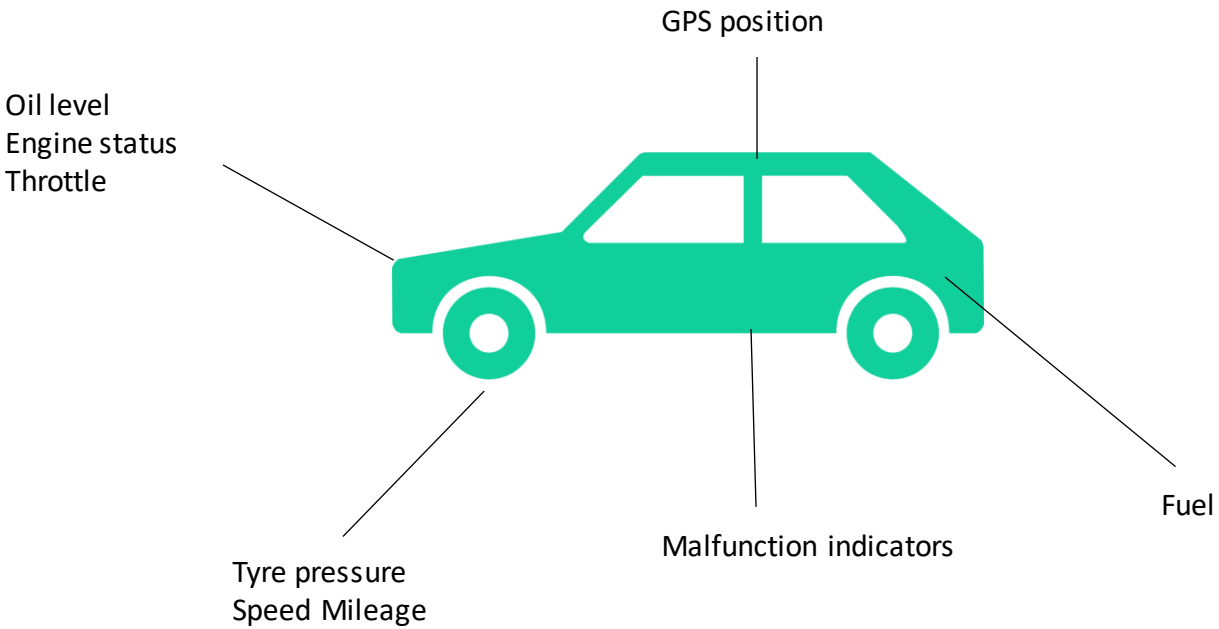
Discover how **car-sharing** services can keep track of their fleets

. . . and know where you are in real time

Use case

Data

Even a simple car can be seen as a system that can provide a huge amount of data



How data are acquired

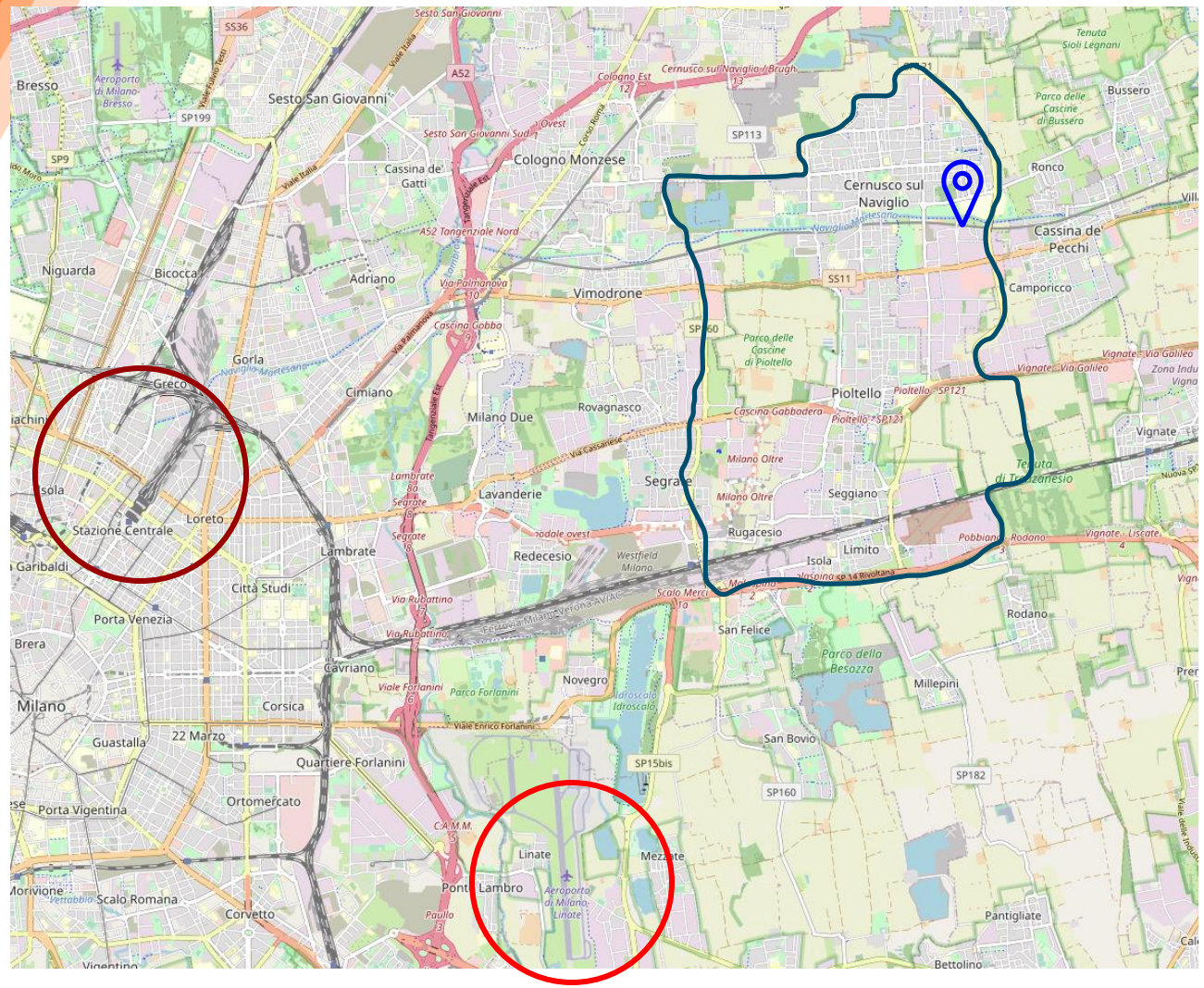
Sensors to be installed in the car

This device collects telemetry data
of the car and the GPS location by sending it
with a SIM card.

If it is connected via OBD2 to the car
It may also collect data from the car
(such as fuel level, car condition,
central locking, etc. etc.)

Use case

Geofence



A virtual perimeter for a real-world geographic area such as point of interest (POI), dangerous areas, ZTL.

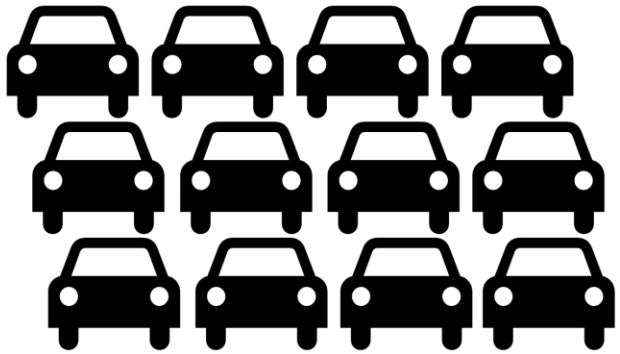
In this example:

- **Headquarter**
- **Near office area**
- **Milano Centrale**
- **Linate Airport**

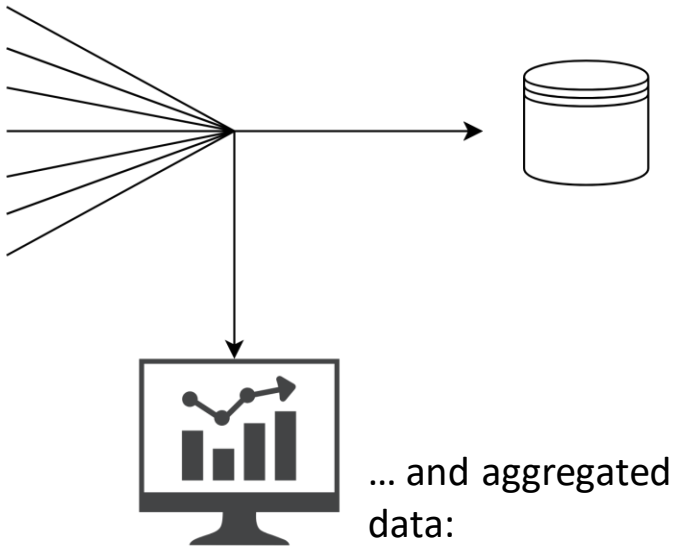
Detect cars movement through these areas

Use case

A simple fleet produces a large amount of such raw data …



… that raw data must be **collected**, **processed** …



… persisted …

… and aggregated to give **value** to data:

- Keep track of cars
- Record trips and events

(Some) Technologies used

Log aggregation
Messaging

Streaming processing
Complex event detection

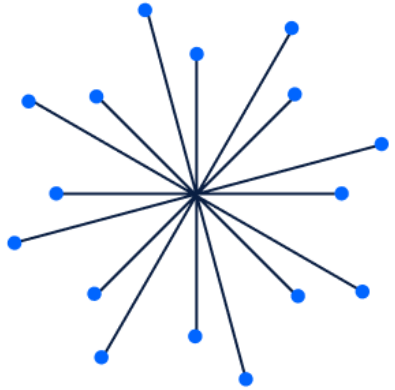
“Collected”: Kafka

Open-source **distributed** **event** **streaming** platform

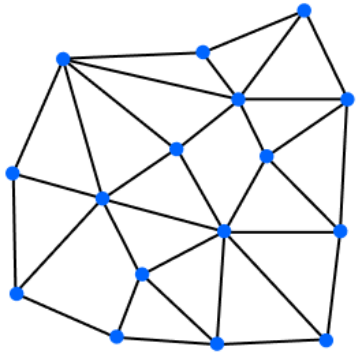
Apache Kafka

“Distributed”

Designed to run in clusters, on multiple **scalable** and **reliable** nodes with a set of **servers** and **clients** communicating via TCP



Centralized



Distributed

Apache Kafka

“Event”

In the data stream, a unit of data represents a specific occurrence (**action**, **change** or **state**) in a system.

- Granularity
- Immutable
- Payload
- Event-driven architecture
- Key, value, timestamp, metadata

Example:

- **Key**: *sensor1*
- **Value:** *pressure 12bar*
- **Timestamp:** *04/12/2023 15:00*

Apache Kafka

“Streaming”

Apache Kafka manages and processes **data streams** continuously, allowing applications to *read/write* in real-time.

Data stream:

- Continuous flow of data generated or received over time
- Sequential and ordered
- *Endless*
- Real-time (almost)
- High-volume

Apache Kafka

Real time streaming

Real time intended as low latency even on large volumes, **NOT** ensuring that a deadline is met!

- **Critical real-time:** microseconds NO
- **Low-latency real-time:** tens/hundreds of microseconds YES
- **Near real-time**: seconds YES

Apache Kafka

High throughput

Apache Kafka handles large volumes of traffic, ideally scaling up to bandwidth saturation.

- Maximizes sequential writes/reads to **disk**
- Uses an in-memory **cache page**
- Uses a **zero-copy** strategy**,** reduces redundant copies between cache and buffer

Apache Kafka

Fault tolerance & Reliability

Apache Kafka provides **reliability**, **consistency** and **redundancy,** thanks to its architecture based on *brokers* and *partitions* (as we will see later).

Kafka is fault tolerant as the failure of a node does not compromise the system thanks to:

- **Redundancy** provided by several replicas (according to the ***replication factor*** desired)
- A mechanism which elects a new node that acts as leader (***leader election mechanism***), guaranteeing *consistency* for consumers and producers without downtime, errors or interruption

Apache Kafka

Durability

Messages are written to **filesystems** and cache pages because they offer better performance than in-memory caches or similar structures.

An in-memory cache would take up too much space (28-30GB on a 32GB machine) and would have to be repopulated in the event of a long shutdown (1 GB/minute).

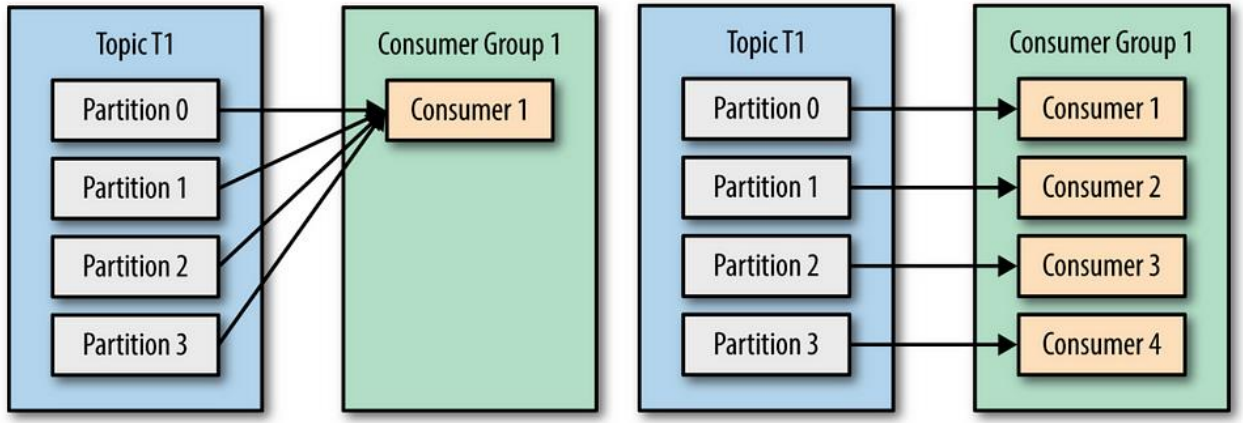
- **Traditional approach**: keep as much as possible in memory, write to disk when it is running out.
- **Kafka approach**: write data to disk, then transfer them to the kernel cache page (pagecache-centric design).

Apache Kafka

Scalability

- **Vertically**
  Adding more resources to the machine, such as CPU, RAM, disk. More expensive approach.

- **Horizontally**
  Adding more brokers to increase processing capacity.
  As we'll see, more is not always better.



Apache Kafka

General use cases

Apache Kafka can replace traditional message brokers (*like MQTT, AMQP*) to achieve:

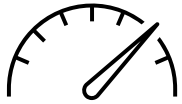
- Better throughput
- Partitioning
- Replication
- Fault-tolerance

which are fundamental characteristics for large-scale solutions.

High throughput is often sufficient, but in some use-cases **low end-to-end latency** and **durability** are also useful.

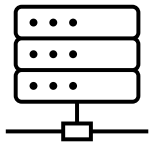
Apache Kafka

General use cases



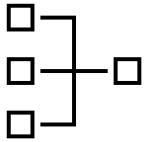
METRICS

Collect monitoring data and merge them into a centralized system.



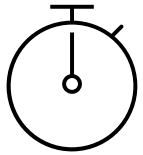
LOGS

Kafka is often used to aggregate log data, typically from files in a central server.



STREAM PROCESSING

Processing data in more complex processing pipelines, where raw input data are consumed by Kafka and are subsequently aggregated, enriched or transformed.



EVENT SOURCING

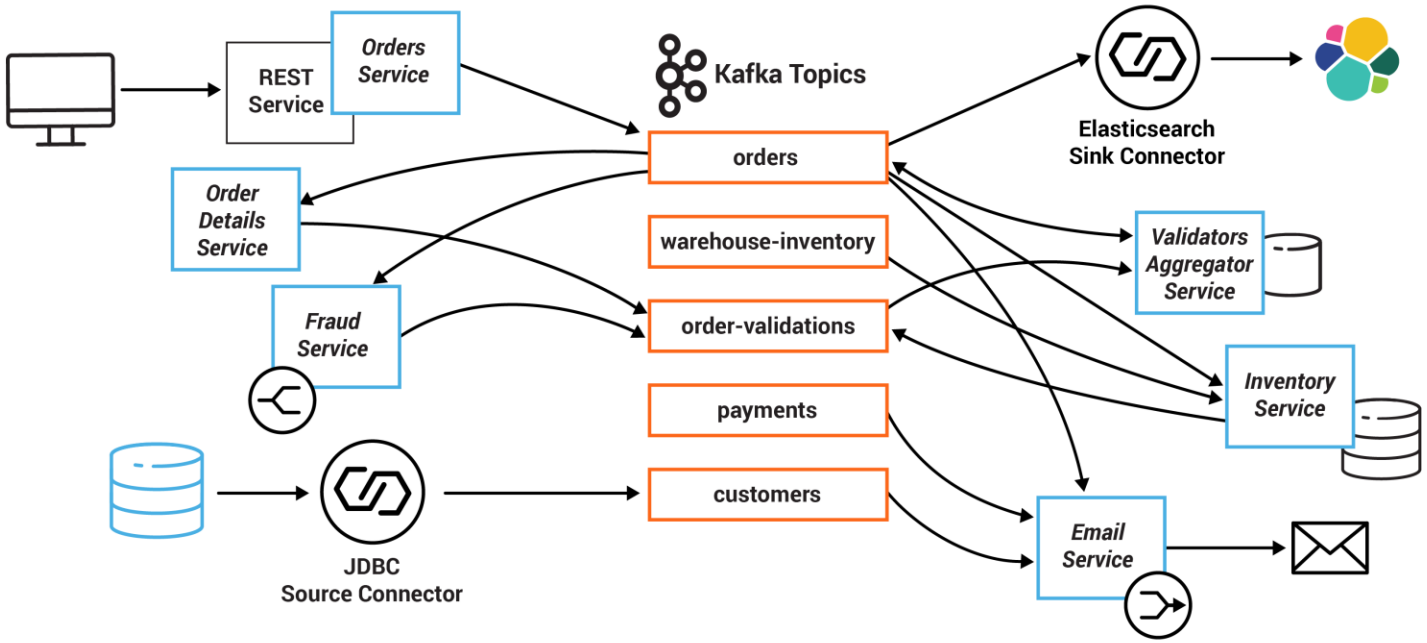
Application design in which state changes are logged as sequences of records in order of time.

BUT ALSO MORE CONCRETELY….

Apache Kafka

General use cases

- Processing **transactions** and payments in real-time
- **Monitoring** vehicle fleets
- Reading and **analyzing** data from IoT sensors
- **Responding** in real time to user interactions
- **Connecting** and saving product data between different divisions of a company
- **Connecting** data platforms and microservices

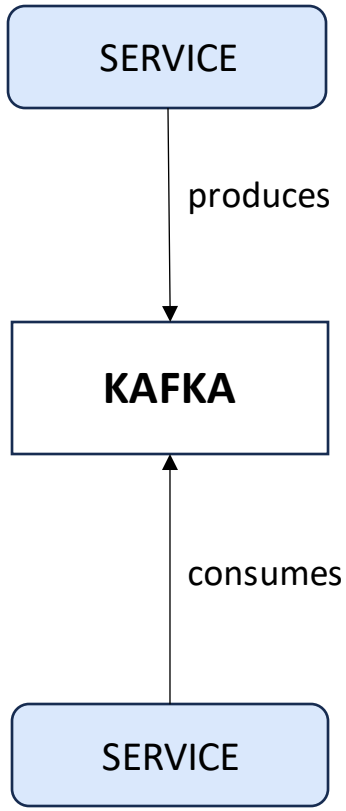


Apache Kafka Architecture

Publish/Subscribe messaging

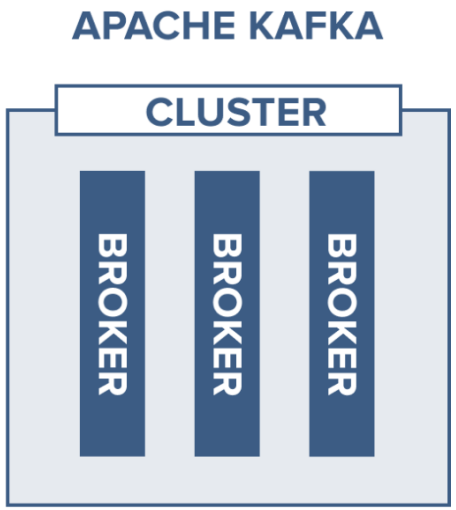
The Pub/Sub messaging architecture:

- Allows to **publish** (write) and **subscribe** (read) on event streams.
- **Decouples** the producer and the consumer of data
- Allows **asynchronous** communication and parallelism



Apache Kafka Architecture

Broker & Cluster



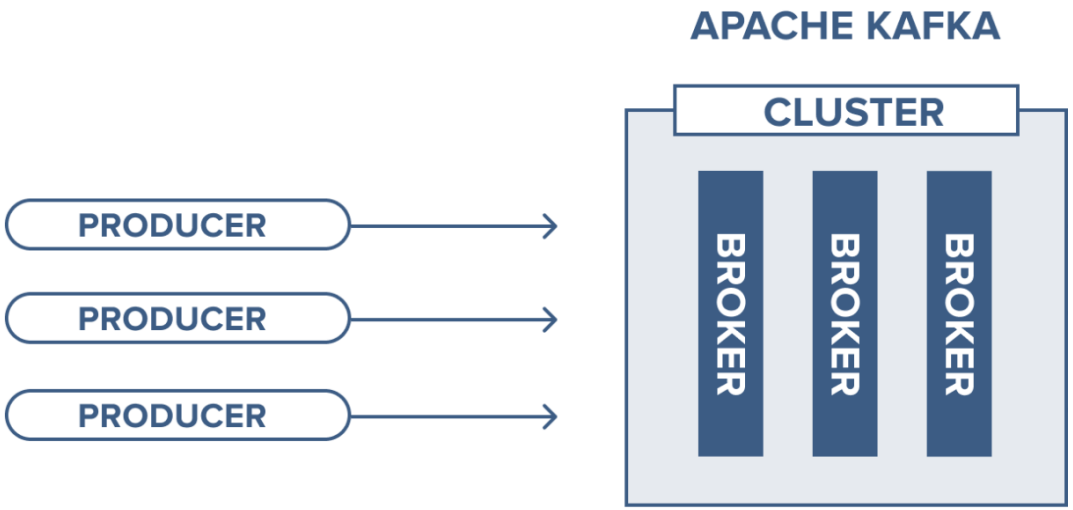
The **cluster**:

- Acts as a **server**
- Manages persistence, replication, durability
- Is scalable
- Maintains Kafka metadata
- Is identified by ID

A single **broker** acts as controller

Apache Kafka Architecture

Producer



A **producer** is any application or service that **sends** messages.

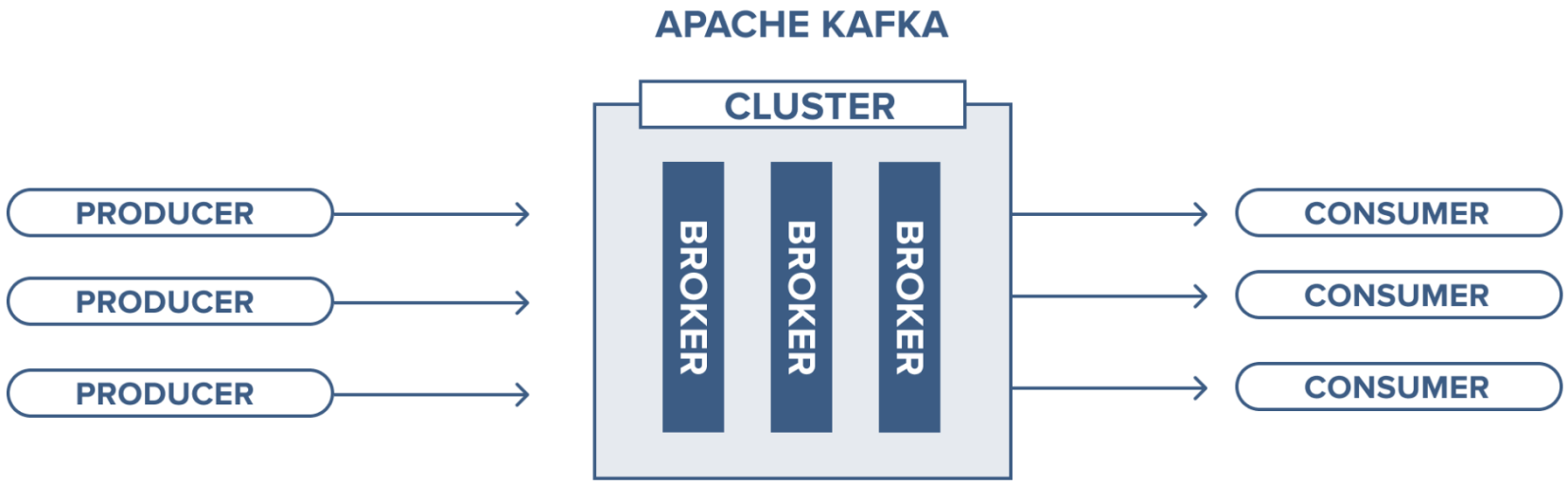
It sends directly to the broker who is the leader for that partition.

To help it, all nodes respond to a request with metadata about which servers are running and where the leader is located.

Finally, the client checks in which partition it is public.
A producer can also send data as **batch**.

Apache Kafka Architecture

Consumer



A **consumer** is any application or service that **receives** messages.

Kafka uses **pull** paradigm, to let the consumer consume at the highest possible rate.

Apache Kafka Architecure

Consumer

A key feature of message brokers is **keeping track** of what has been consumed.
And in Kafka?

- If the broker marks a message as "**consumed**" as soon as it leaves its network?

  If the consumer fails to receive it, **the message is lost.**

- If we add ack from the consumer?

  Not working if consumer receives but **fails** before ack.
  The broker **must keep track** of multiple states for each message

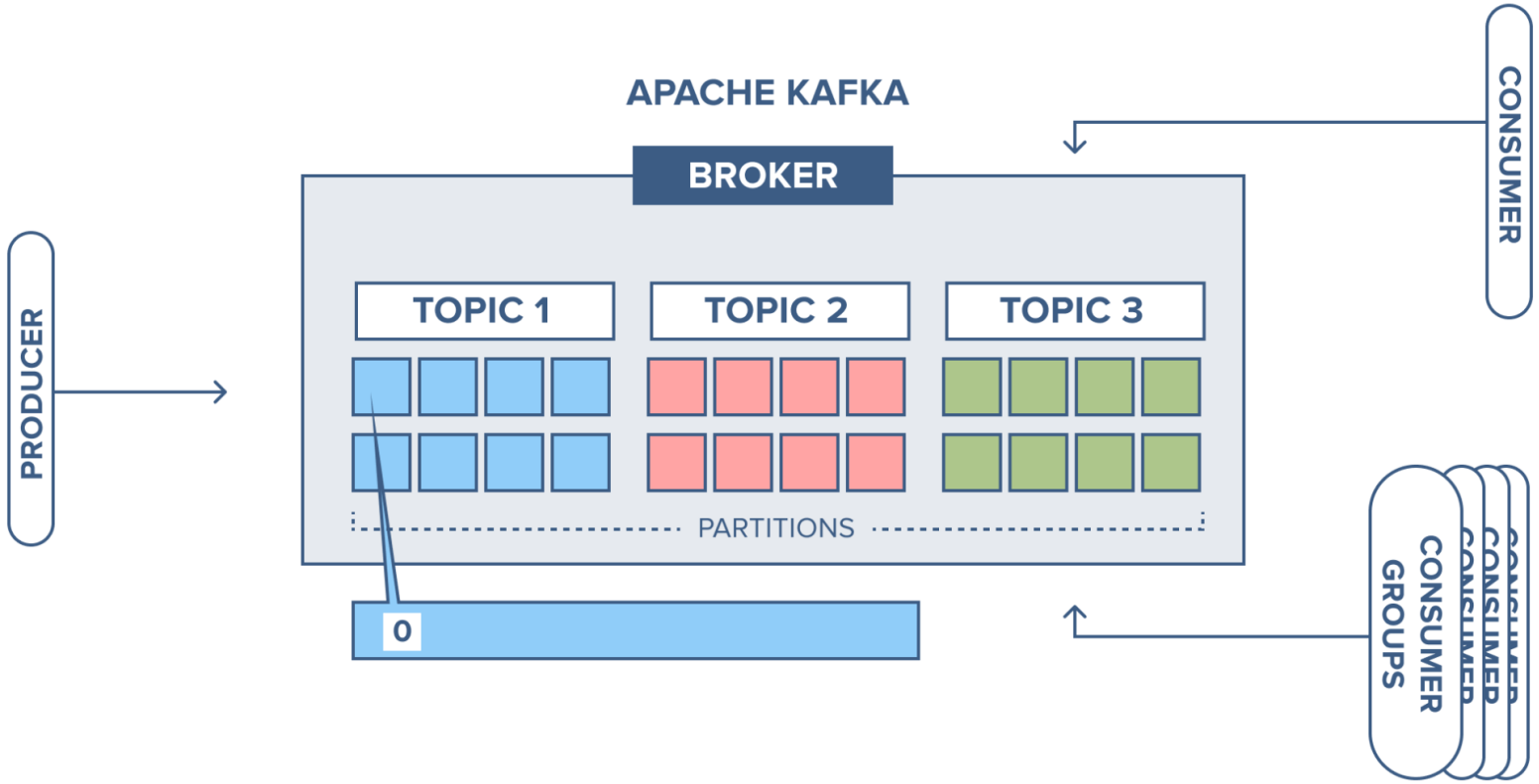
We will see offset strategy

Apache Kafka Architecture

Topic

A message “*category*”

Producer → who sends
Consumer → who reads
Topic → the subject

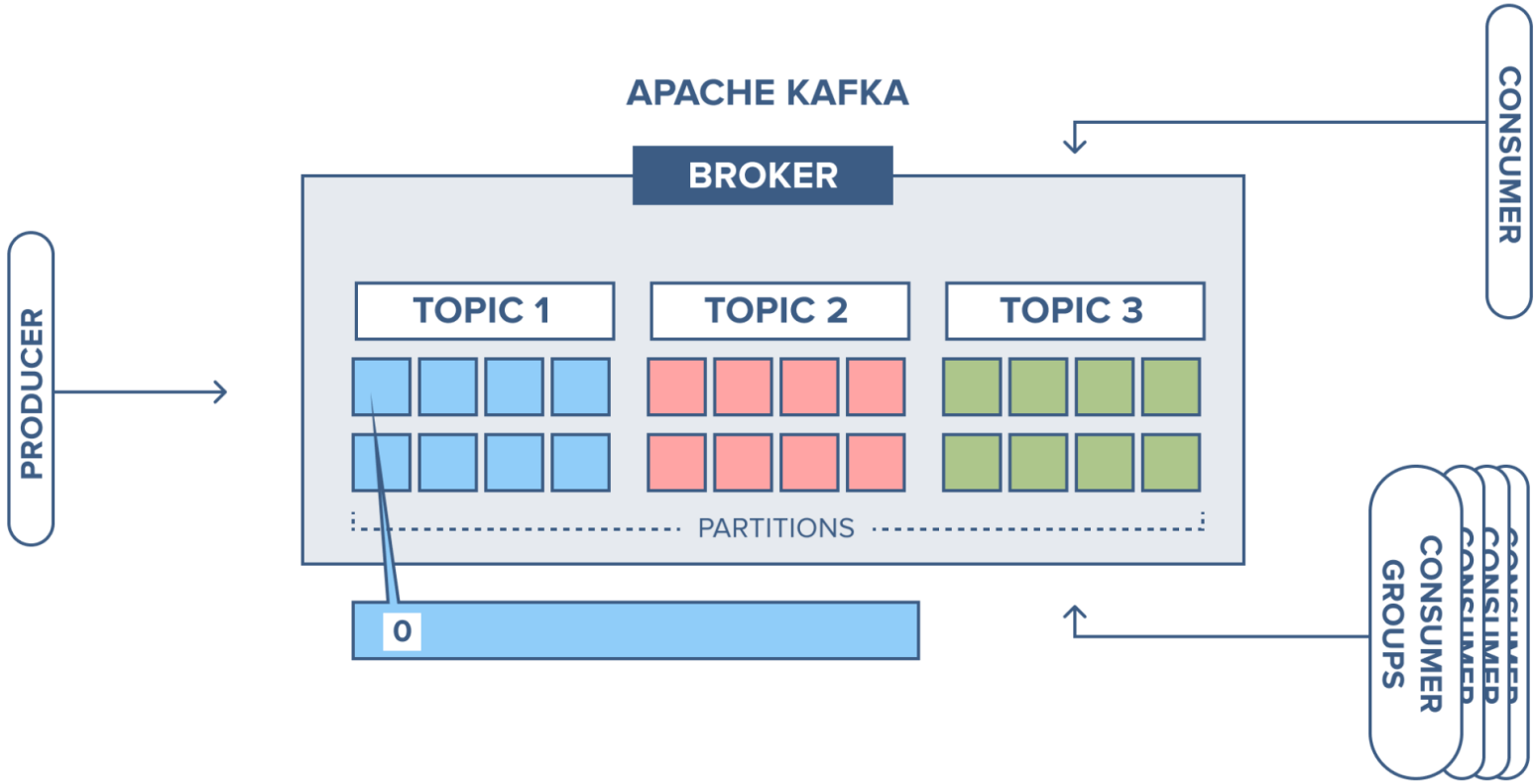


Apache Kafka Architecture

Partitions

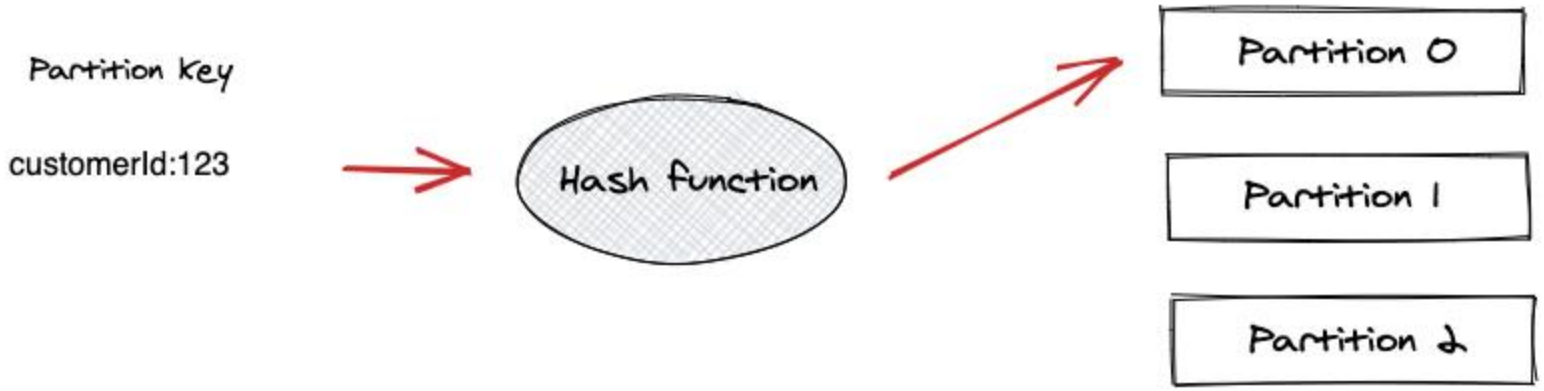
A **partition** is an ordered set of messages with a unique sequence number called **offset**.

Partitions allow greater scalability and fault tolerance



Apache Kafka Architecture

Partition key

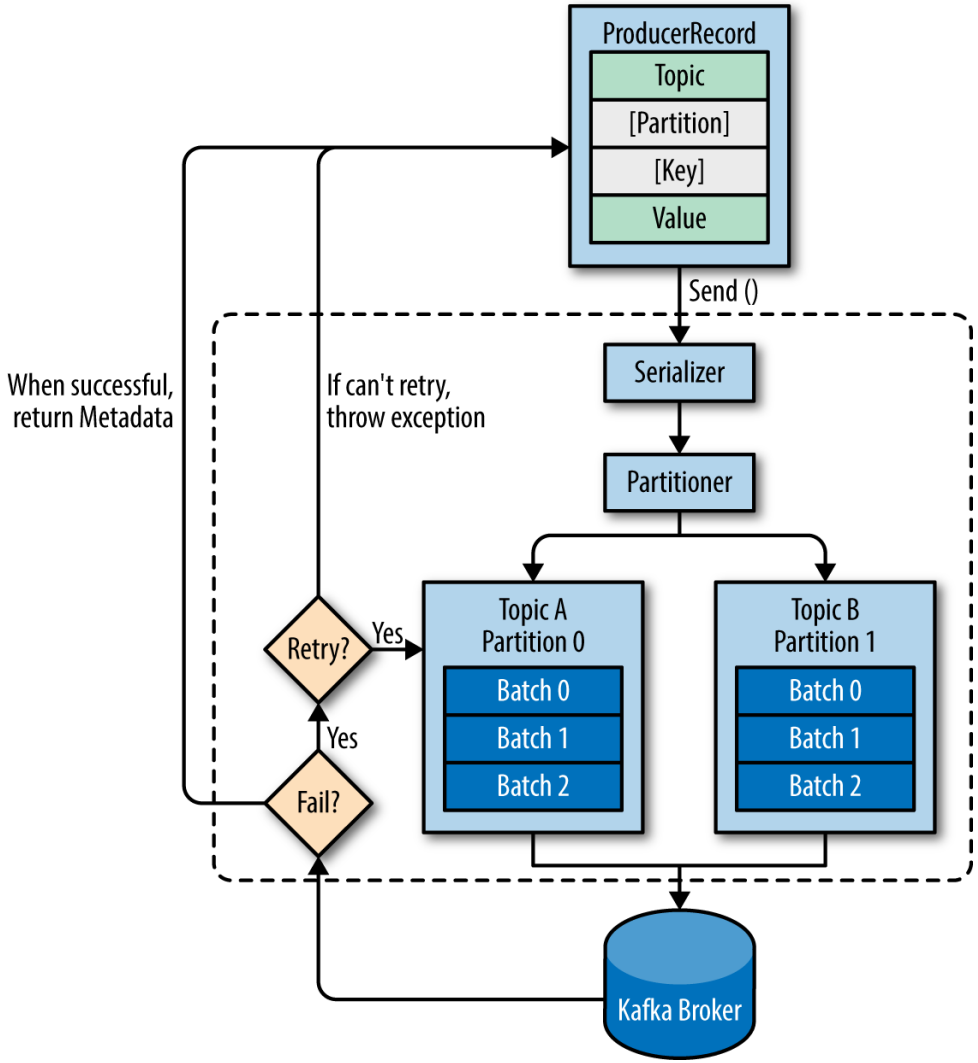


You can choose a partition key to distribute messages.
NB: choose well!

- Default Kafka: If not specified, Kafka uses round-robin
- Custom partitioner: implement your own rules

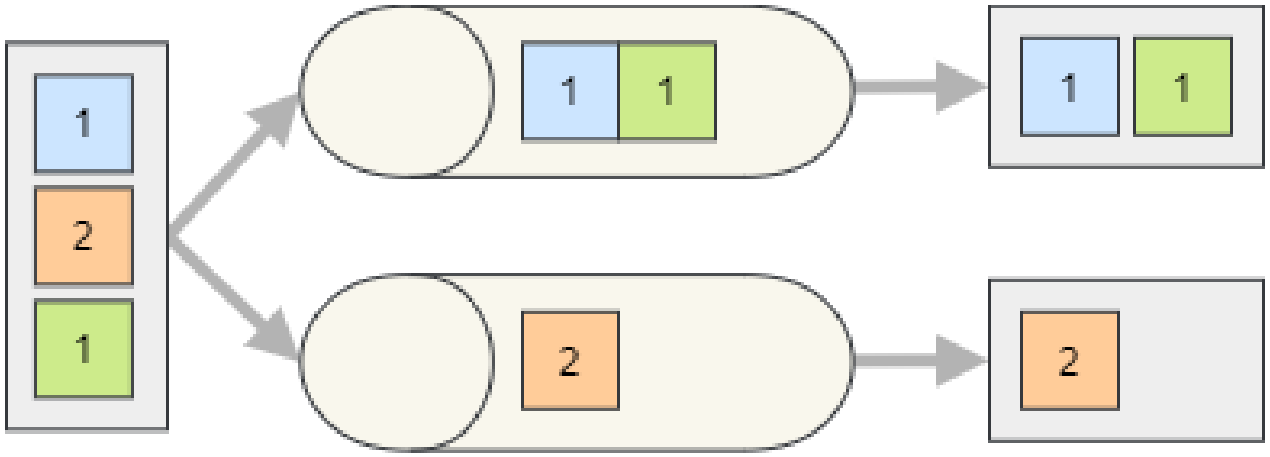
Apache Kafka Architecture

Partition key



Apache Kafka Architecture

Partition key



Kafka ensures that messages with the same key are written to the same partition

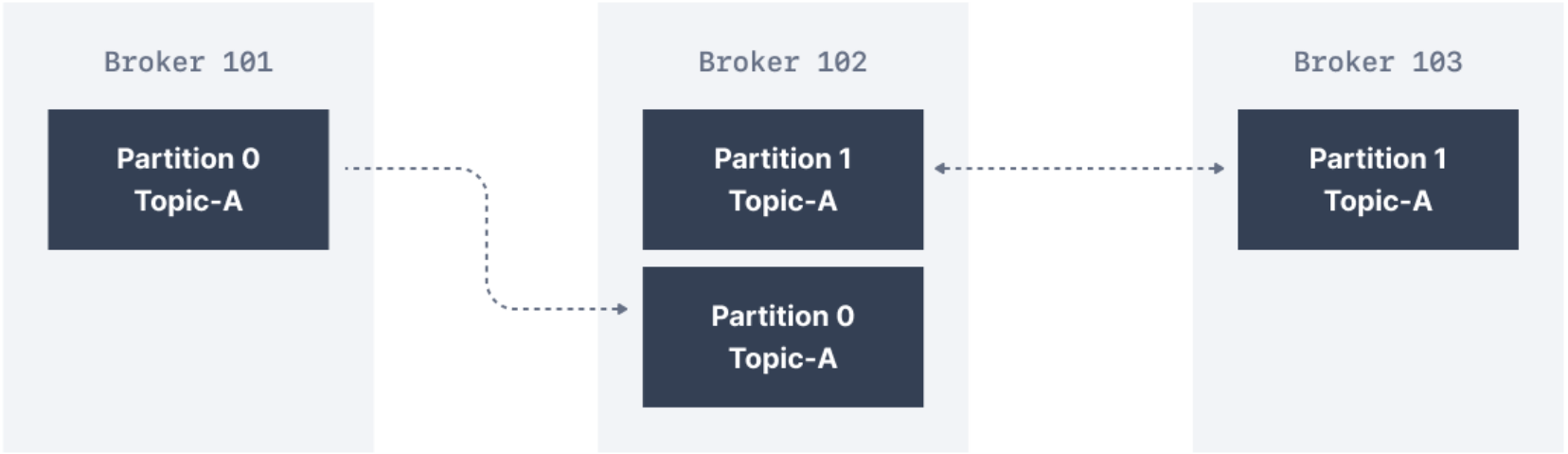
Apache Kafka

Replication factor

Factor that regulates how much data replication there will be between brokers.

Prevents data loss and preserves HA (*high availability*) in the event of broker failure.

Example with Topic-A and replication factor = 2



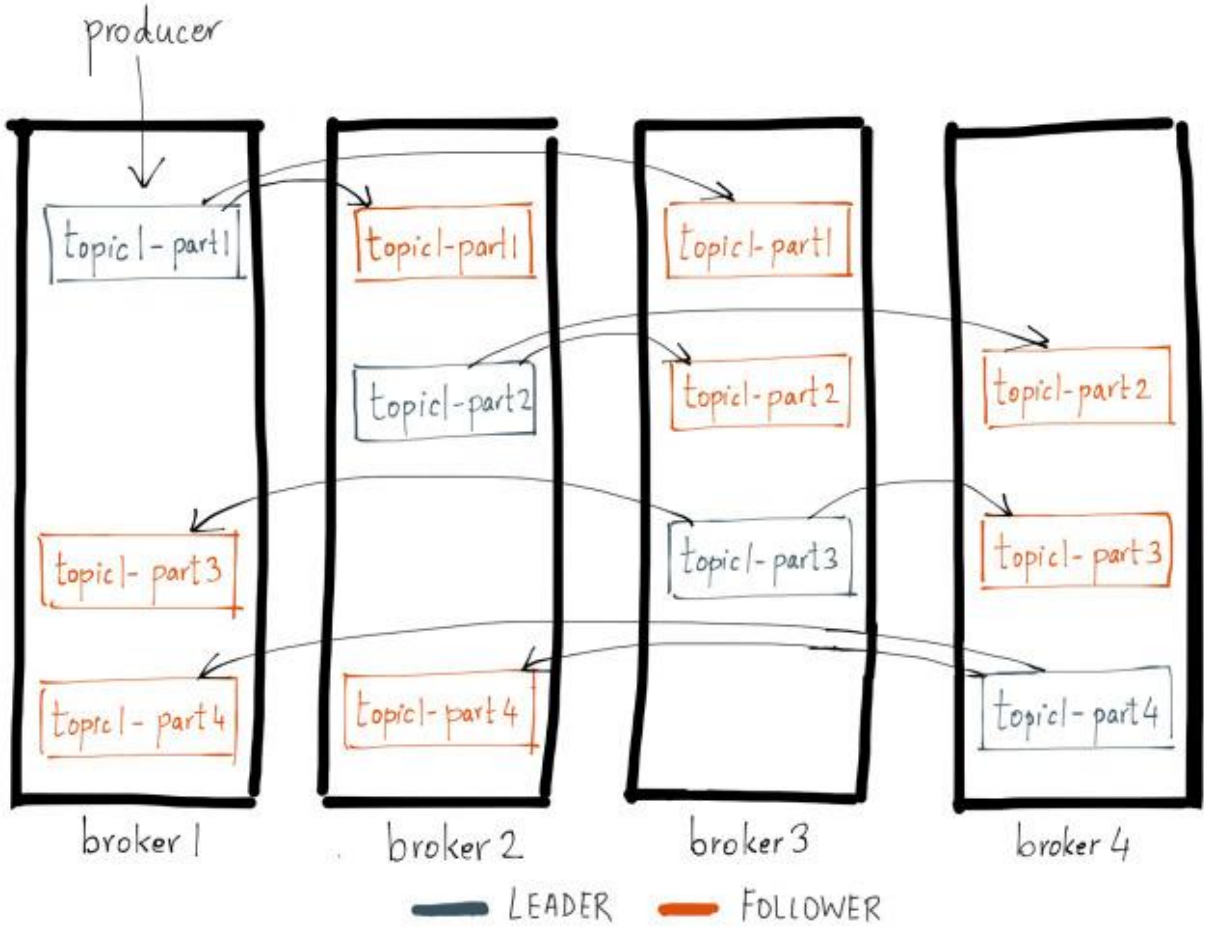
Apache Kafka

Replica leader

Leader is one of the replicas of a topic partition that is responsible for managing the writing and reading of messages.

It is the **contact point** for producers and consumers.

Follower replicas copy the data.



Apache Kafka

Leader election

The **leader** is fundamental!

Elected by:

- Network latency
- Broker workload
- Specific configurations

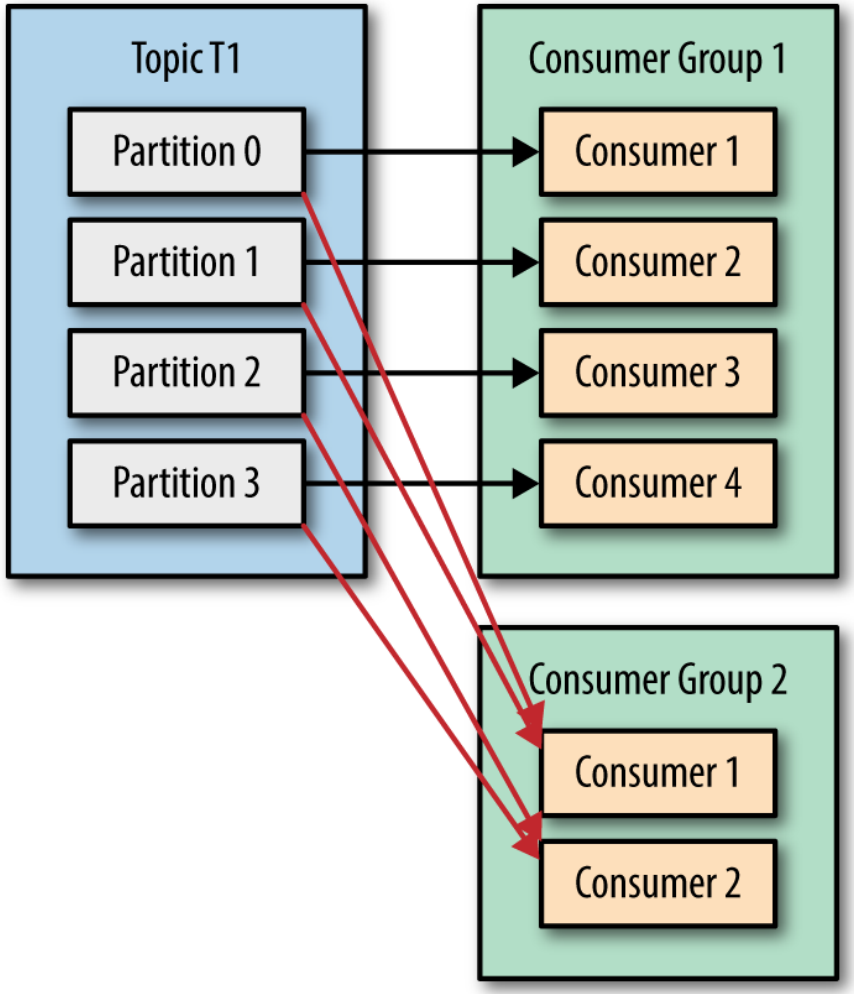
But also:

- Quorum consensus: determined by *replication factor*
- ZooKeeper or topic metadata
- Epoch number
- ISR (In-Sync Replica)
- Performance

Apache Kafka Architecture

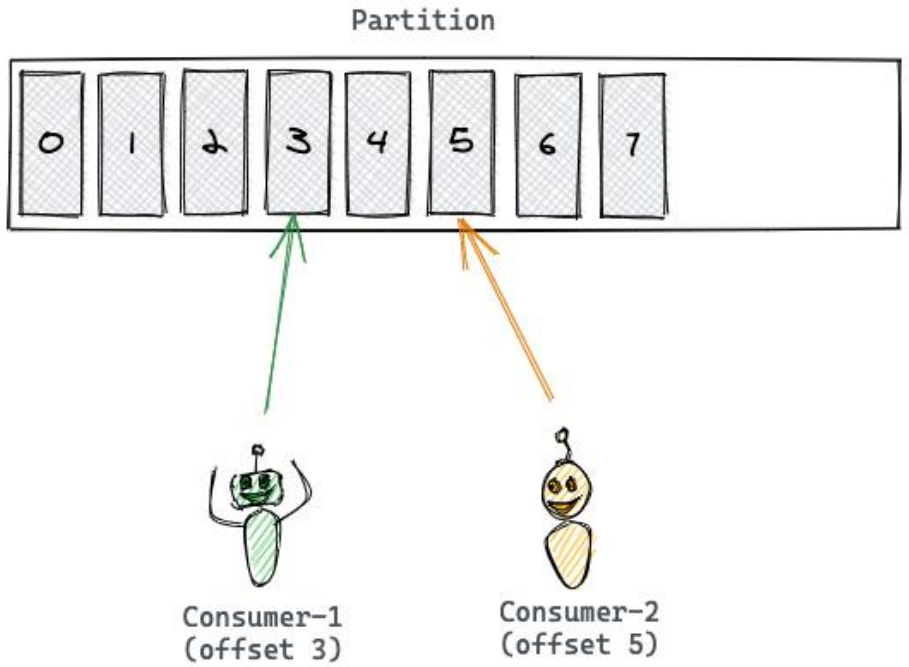
Consumer Group

- Logical group of consumers united to read messages from one or more topics.
- Unique name
- They allow you to distribute the load among various consumers
- They guarantee that messages are read only **once (per topic, per partition)**



Apache Kafka Architecture

Offset



Kafka does not send messages, but consumers read them from the topic.

Remembering the offset is therefore the consumer's task.

Apache Kafka Architecture

Offset

Each consumer has his own “**view**" of the partition and the topic and can **independently** manage the consumption strategy:

- Starting again from the last offset you saved
- Always reprocessing all messages from the beginning
- Starting from a specific offset

Apache Kafka Architecture

Acks - Producer

The producer sends a message to a Kafka broker and receives an asynchronous ack when the broker has confirmed writing the message to the log.

Using the acks you choose the level of reliability between **at most once, at least once and exactly once:**

1. **acks=0**: the producer does not wait for confirmation, proceeds to the next message

2. **acks=1**: Default, the producer considers a message delivered when leader sends an acknowledgment. **It does not** mean that the consumer has received!

3. **acks=all or acks=-1**: the producer interprets a message as delivered only when all replicating brokers have received the message and written to the logs. More latency!

Apache Kafka Architecture

Commit - Consumer

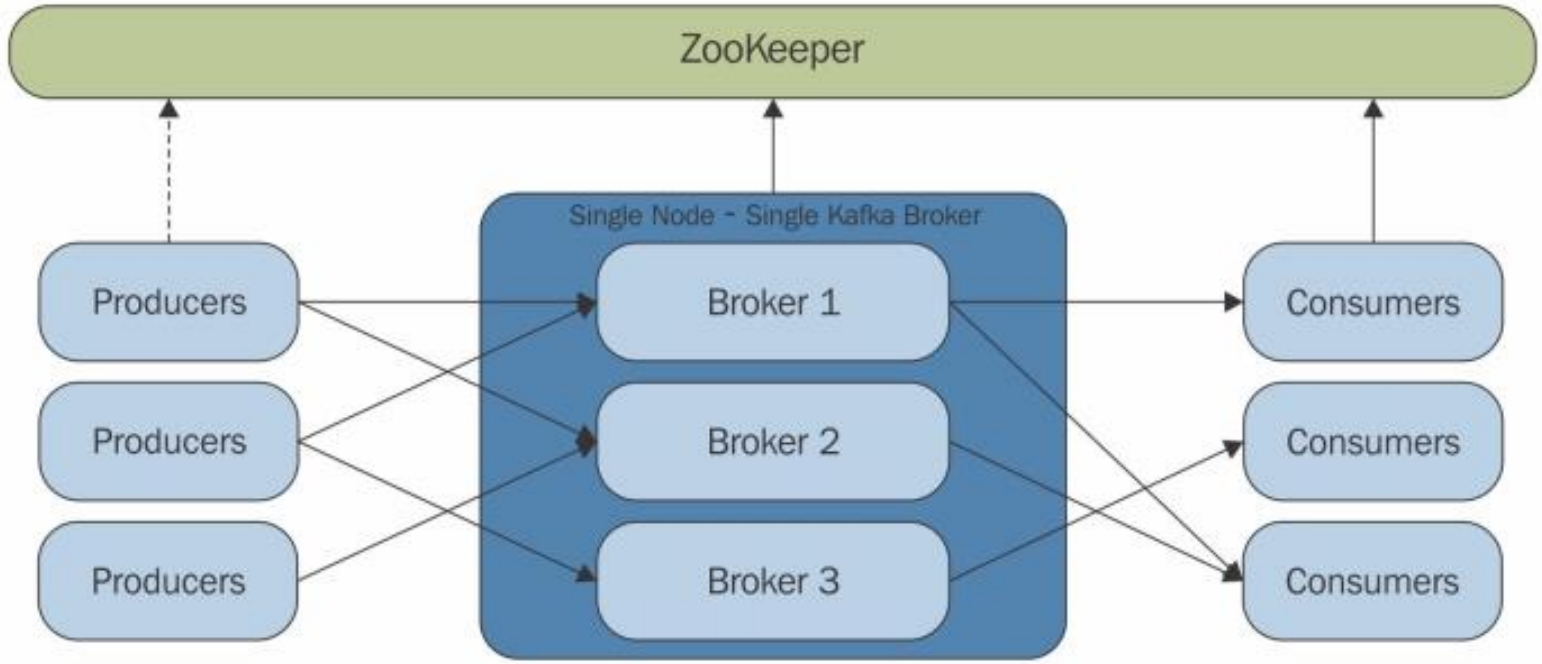
The consumer can acknowledge receipt of the message and confirm its correct processing.

1. **Auto commit:** managed by Kafka automatically when the message has been delivered to the consumer. Configurable time and frequency.

2. **Explicit commit**: sent manually by the consumer, more control. For example, I await the actual processing of the message.

Apache Kafka Architecture

ZooKeeper



Centralized service used to maintain definitions and configurations in a synchronized and flexible way within a distributed system (such as Kafka between brokers).

Apache Kafka Architecture

ZooKeeper

ZooKeeper is useful BUT:

- It adds an extra layer of management
- It makes Kafka more complicated, having to manage two "services" instead of one
  (Instead, o*ther distributed systems have similar mechanisms but internally managed)*
- It has few hardware needs, but they are still present

Apache Kafka

Exercises

The repo: https://tinyurl.com/fleetMonitor

Who has Docker Desktop installed

Can run the command

docker compose up

to start the containers

In order to compile the Java project and execute use this command

mvn compile exec:java

Apache Kafka

Exercise 1

- Create a topic “speeds”
- Write a producer that sends simple random numeric speed values between 0 and 150, for ten random cars every 250 ms.

- JSON Example:

Apache Kafka

Exercise 1

```
public static void main(String[] args) {
    try (var producer = createKafkaProducer()) {
        var objectMapper = new ObjectMapper();


        while (true) {
            var random = new Random();
            var vin = random.nextInt(11);
            var speed = random.nextInt(151);

            CarSpeed carSpeed = new CarSpeed(vin, speed);
            String carSpeedJson;

            try {
                carSpeedJson = objectMapper.writeValueAsString(carSpeed);
                Thread.sleep(250);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
            logger.info("Sending: {}", carSpeedJson);



            ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, carSpeedJson);
            producer.send(record);
            producer.flush();
        }
    }
}
```

Apache Kafka

Exercise 1b

- Write a consumer that reads message and print them with metadata (*key, value, offset, partition ecc.)*

Apache Kafka

Exercise 1b

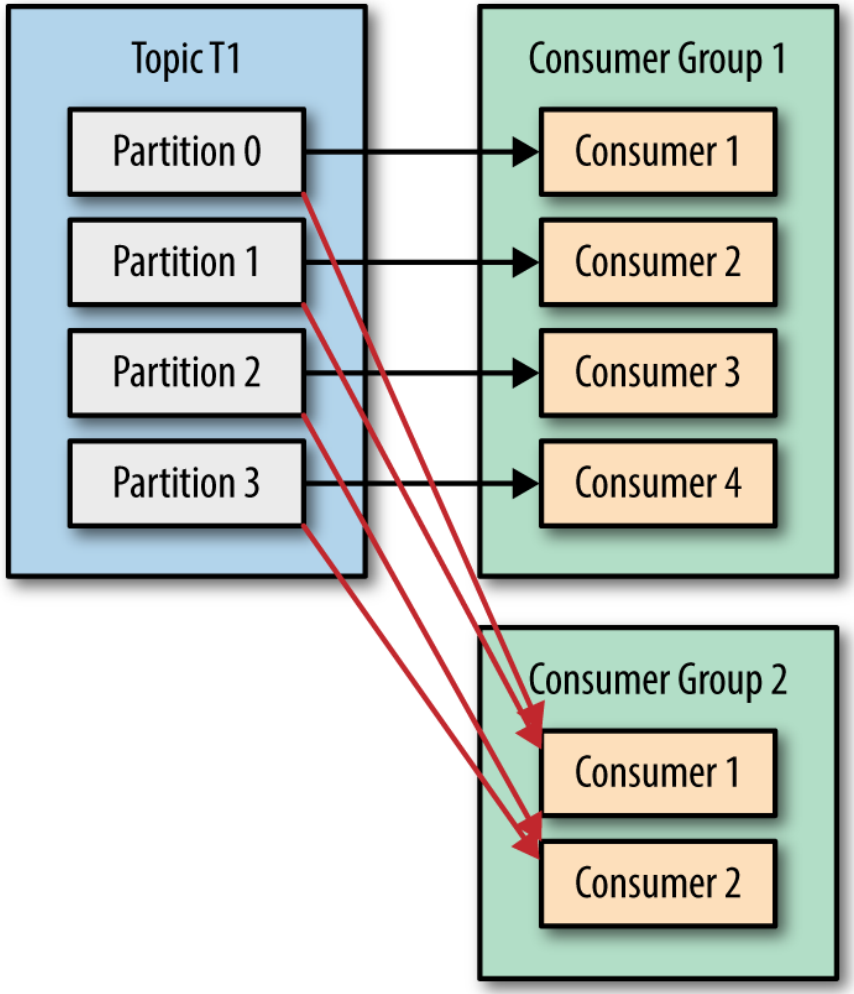
```
public static void main(String[] args) {
    try (var consumer = createKafkaConsumer(consumerGroup)) {
        consumer.subscribe(List.of(topic));

        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(1));
            for (ConsumerRecord<String, String> record : records) {
                logger.info("offset = {}, key = {}, value = {}", record.offset(), record.key(), record.value());
            }
        }
    }
}
```

Apache Kafka Architecture

Consumer Group

- Logical group of consumers united to read messages from one or more topics.
- Unique name
- They allow you to distribute the load among various consumers
- They guarantee that messages are read only **once (per topic, per partition)**

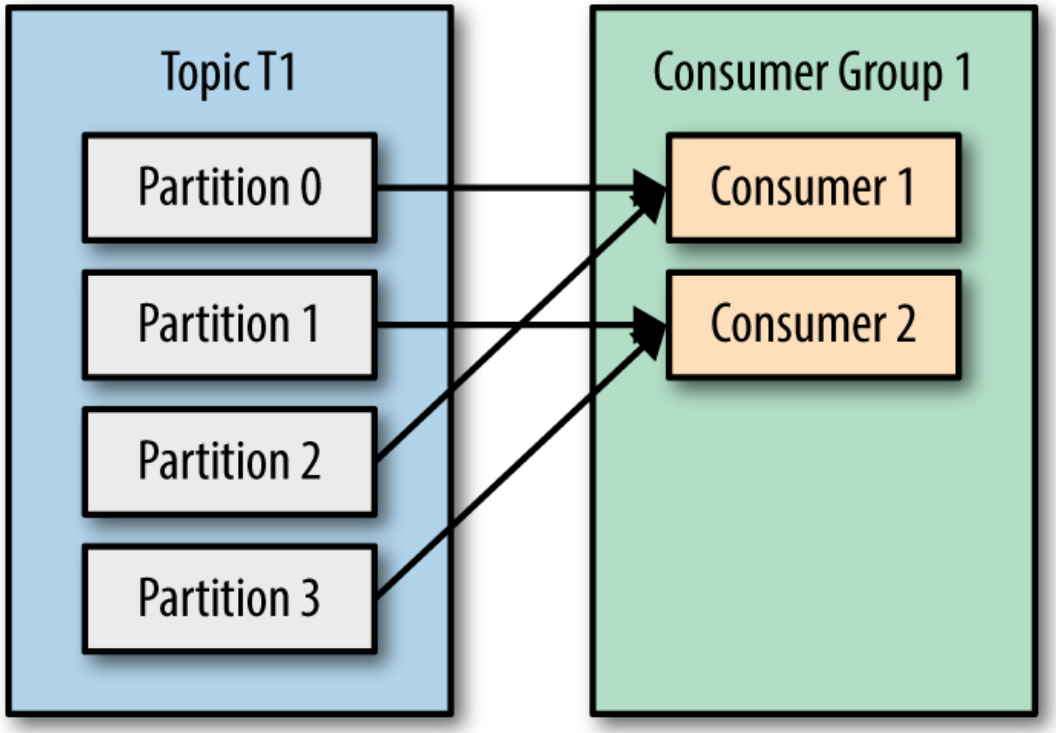


Apache Kafka

Parallelism

P = number of partitions, C = number of consumers

P > C

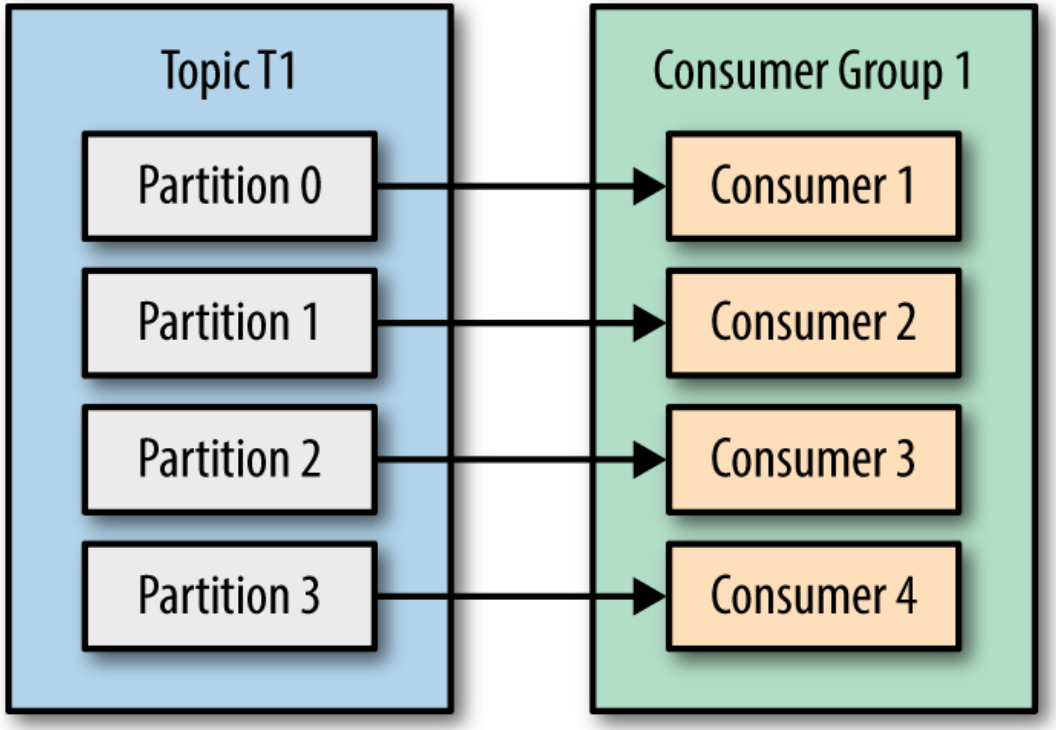


Apache Kafka

Parallelism

P = number of partitions, C = number of consumers

P = C

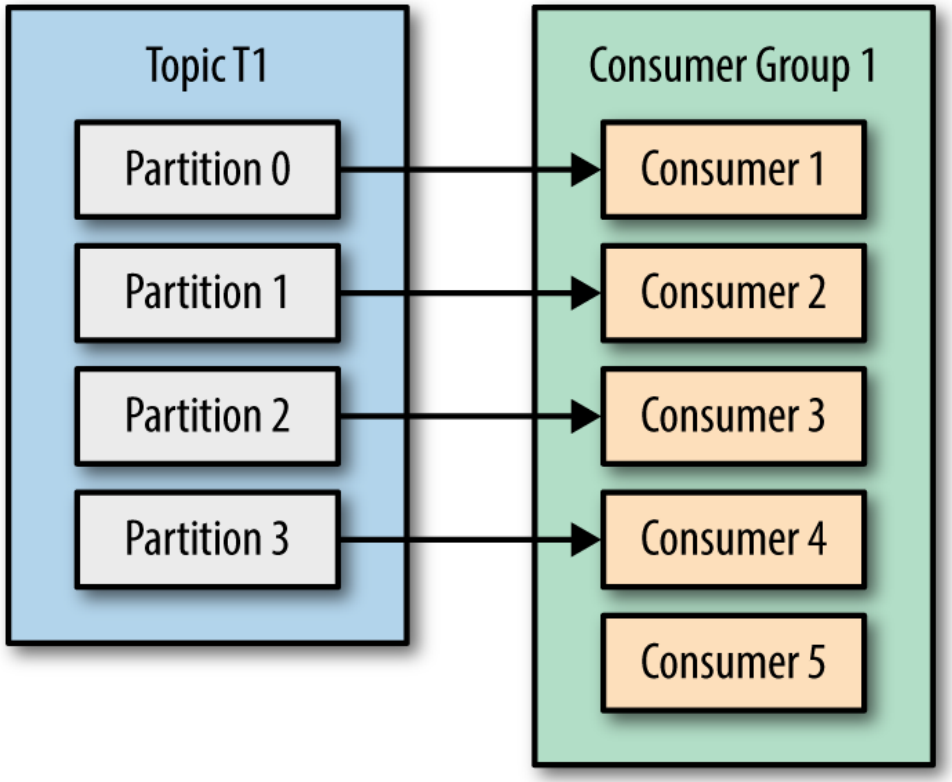


Apache Kafka

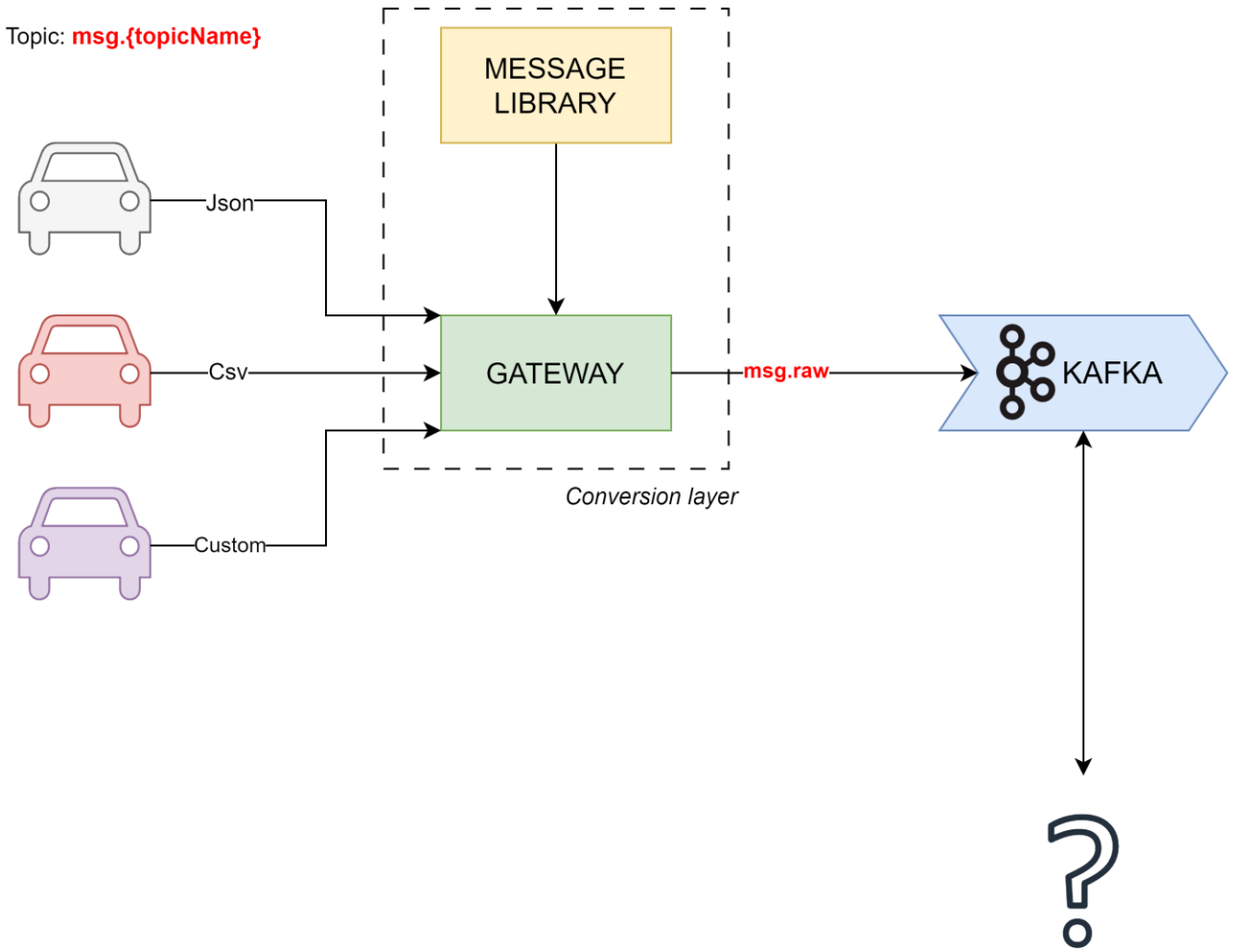
Parallelism

P = number of partitions, C = number of consumers

P < C



Architecture so far



What have we achieved?
Tracking and logs cars, just a stateless elaboration

Not enough!
We need to remember devices state to get *geofencing*

Stateless VS Stateful

STATELESS	STATEFUL	HYBRID
Process each event independently	Maintains historical data	Combines both trade-offs
No memory of prior events	Complex computations, pattern recognition	Balance low latency with analytics
Suitable for low-latency tasks	Fraud detection, recommendation systems	Choose based on specific use case needs

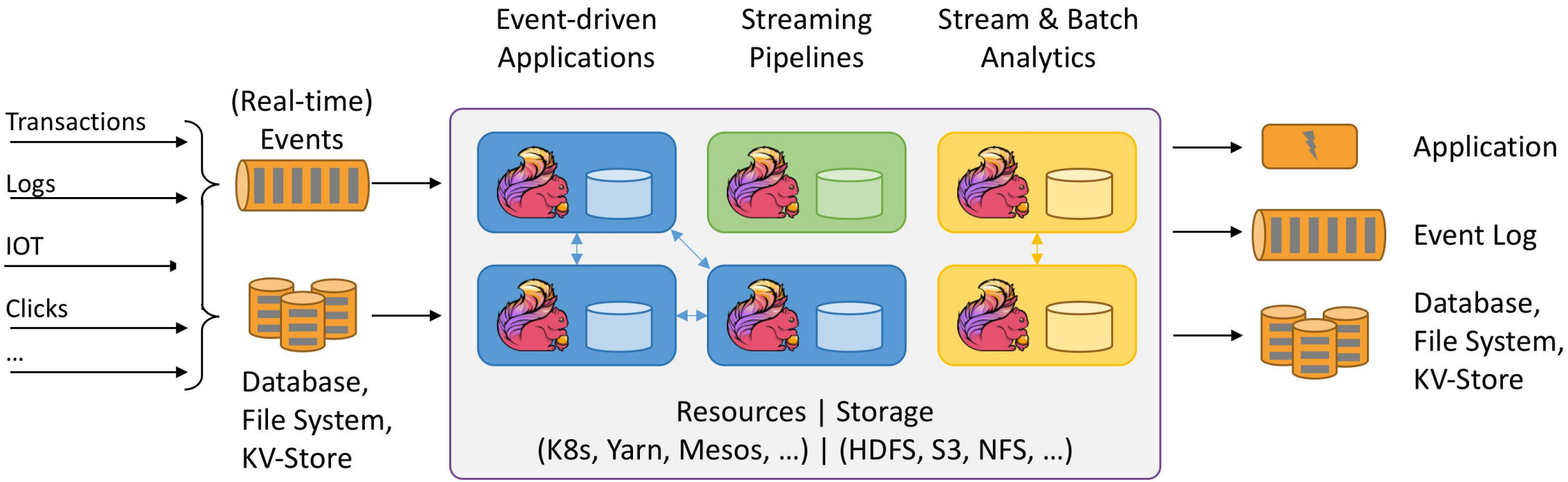
Apache Flink

Processed

Framework for **stateful** computations over *unbounded* and bounded data streams

Apache Flink

Framework…



Apache Flink

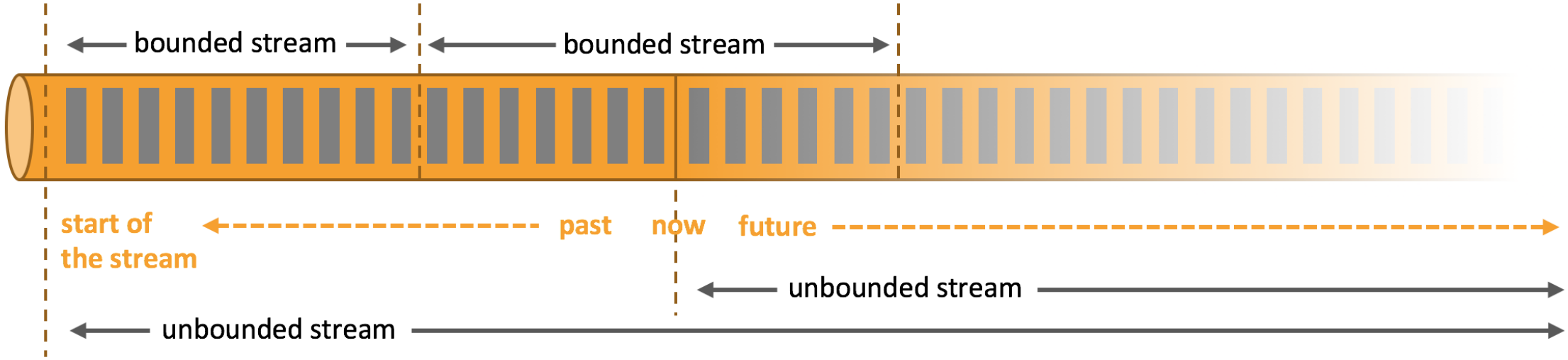
… for stateful computations…

Examples of stateful operations:

- Application searching for certain event patterns → sequence of events encountered so far
- Aggregating events per hour → pending aggregates
- Training a machine learning model → current version of the model parameters

Apache Flink

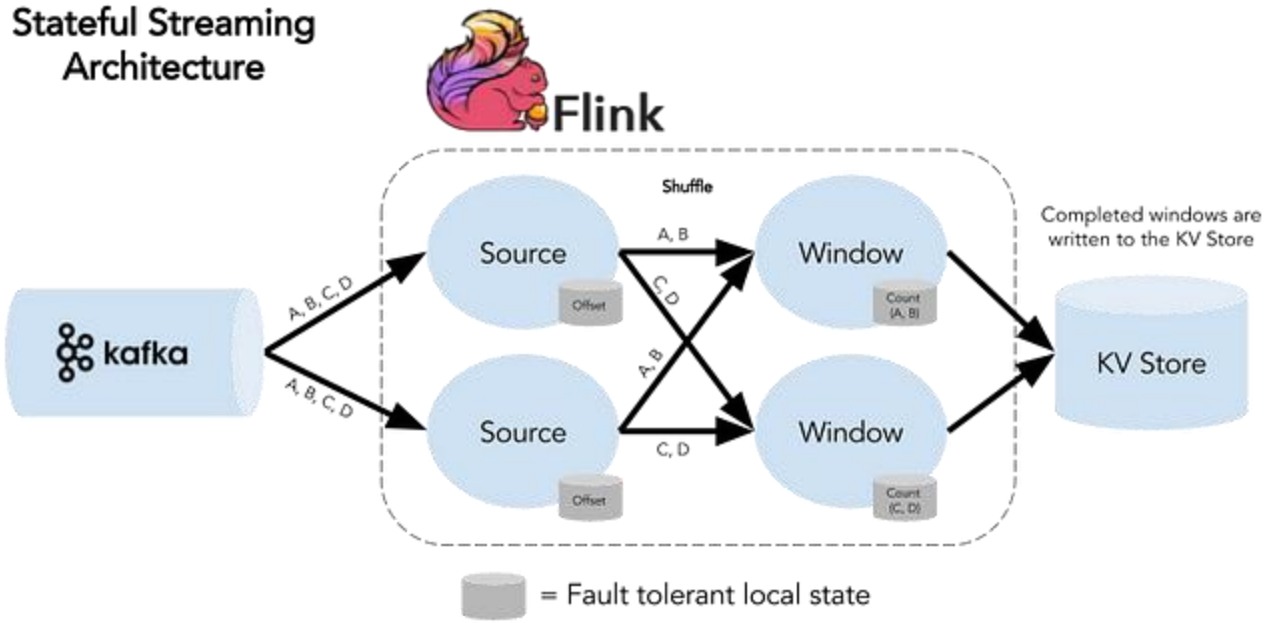
… over *unbounded* and *bounded* data streams



Apache Flink

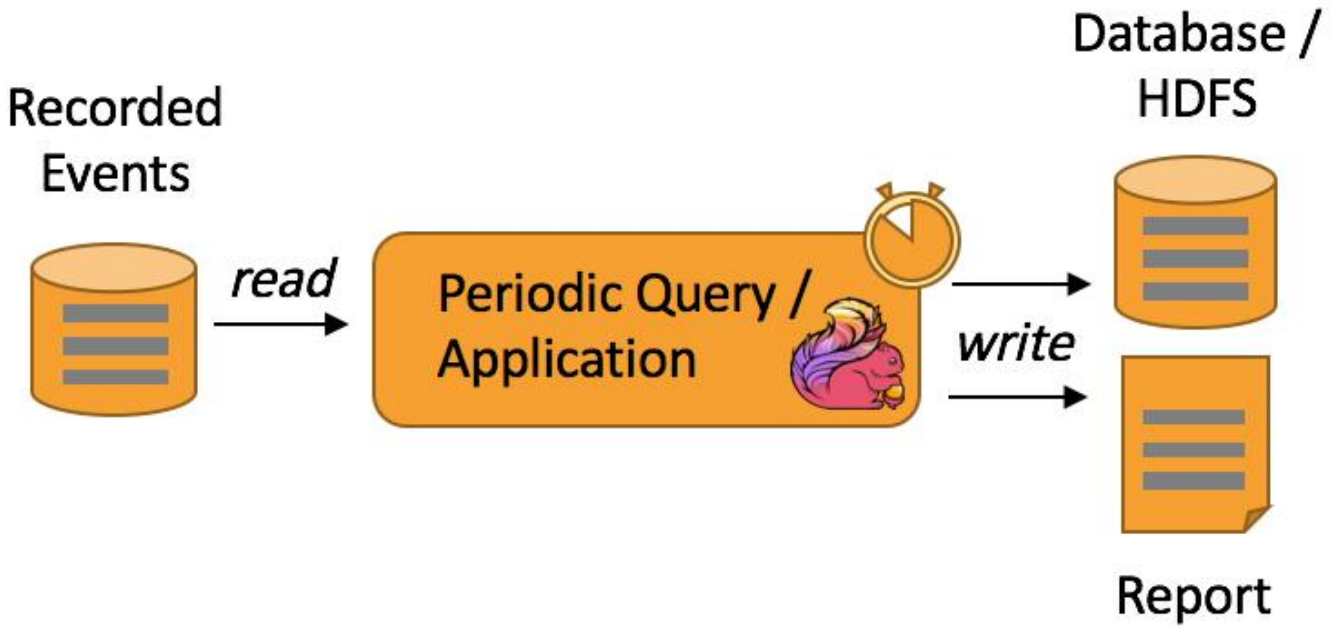
Features & advantages

- Low latency
- Exactly-once processing
- Stateful processing
- Event time processing
- Fault tolerance
- Scalability
- Integration with Kafka
- Stream-Batch unification



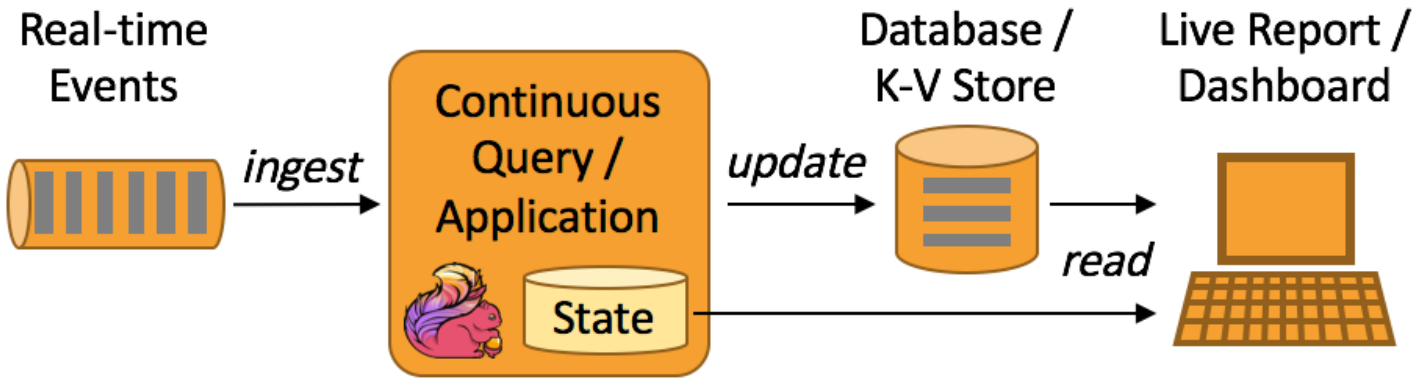
Apache Flink

Batch processing



Apache Flink

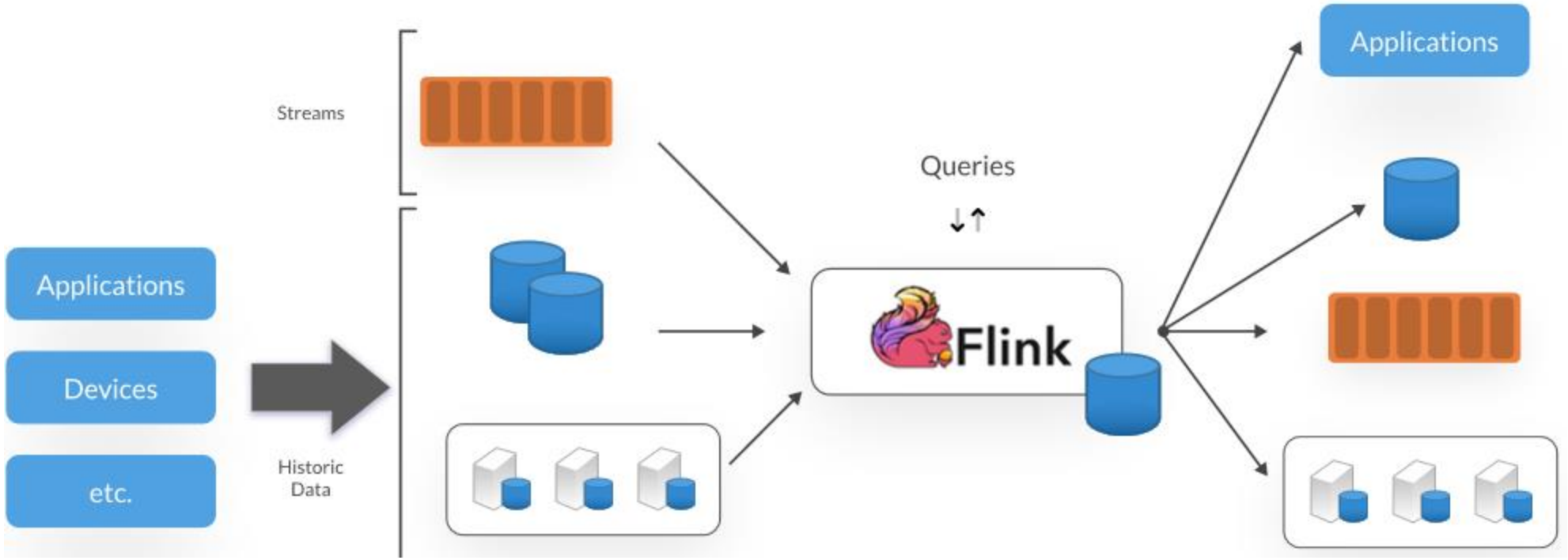
Stream processing



Apache Flink

Processed

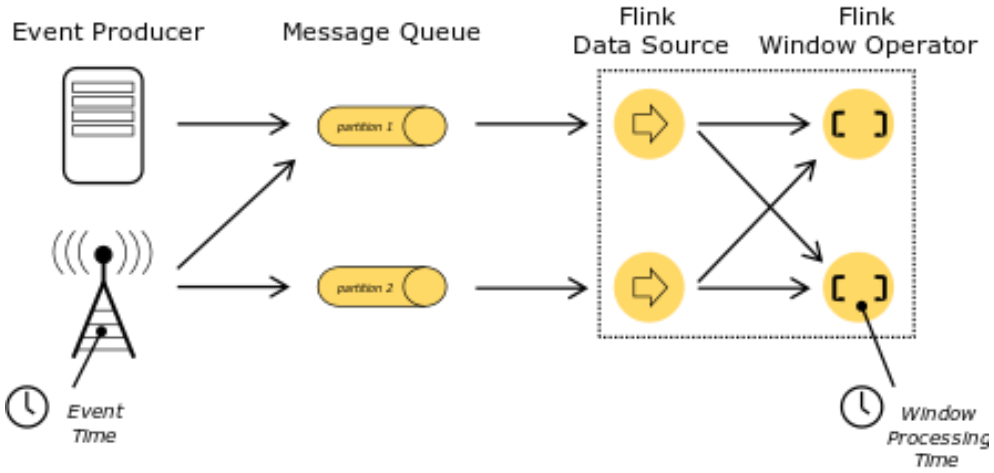
State on data-streams → Stateful → Remember device's state



Apache Flink

Timely stream processing

Extension of stream **processing** in which **time** plays some role in the computation



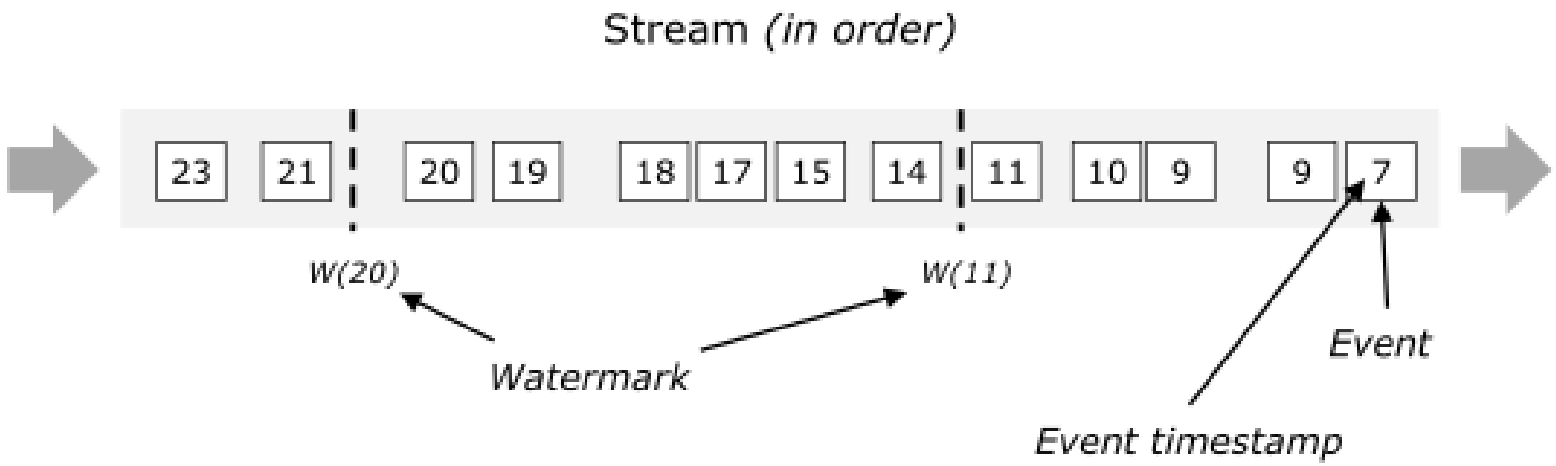
Processing time: refers to the system time of the machine that is executing the operation

Event time: the time when each individual event occurred

Apache Flink

Watermark and Time windows

A **Watermark** declares that event time has reached time t in that stream, meaning that there should be no more elements from the stream with an event time t' <= t



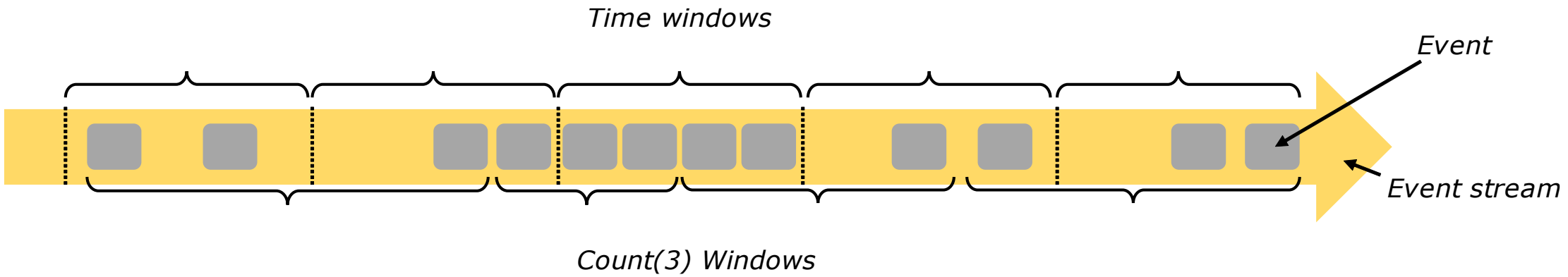
E.G. W(11) → I'll process only records with event time >= 11

Apache Flink

Watermark and Time windows

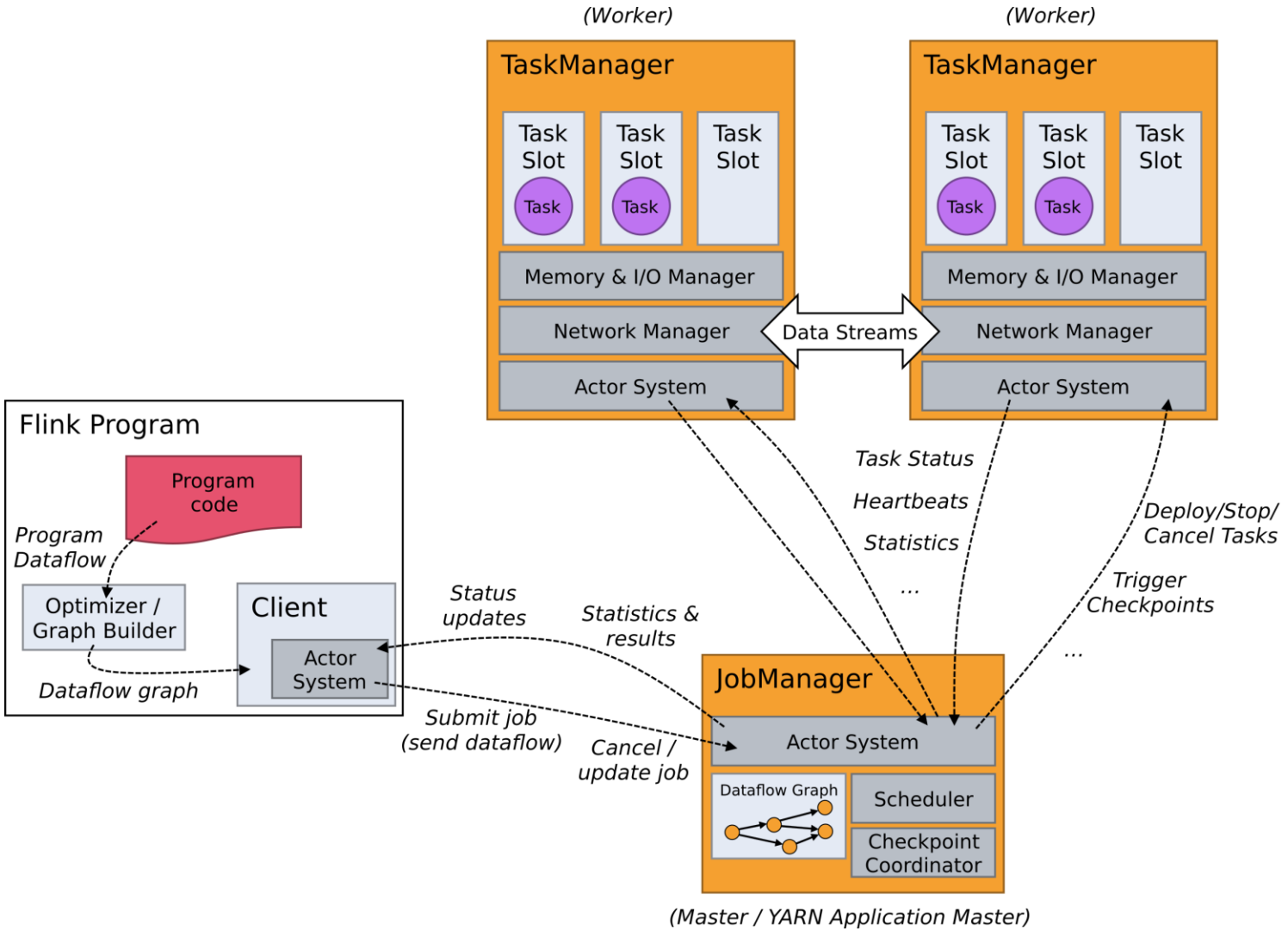
A **Window** can be:

- Time driven or data driven
- Tumbling, sliding or session



Apache Flink

Components: JobManager and TaskManager

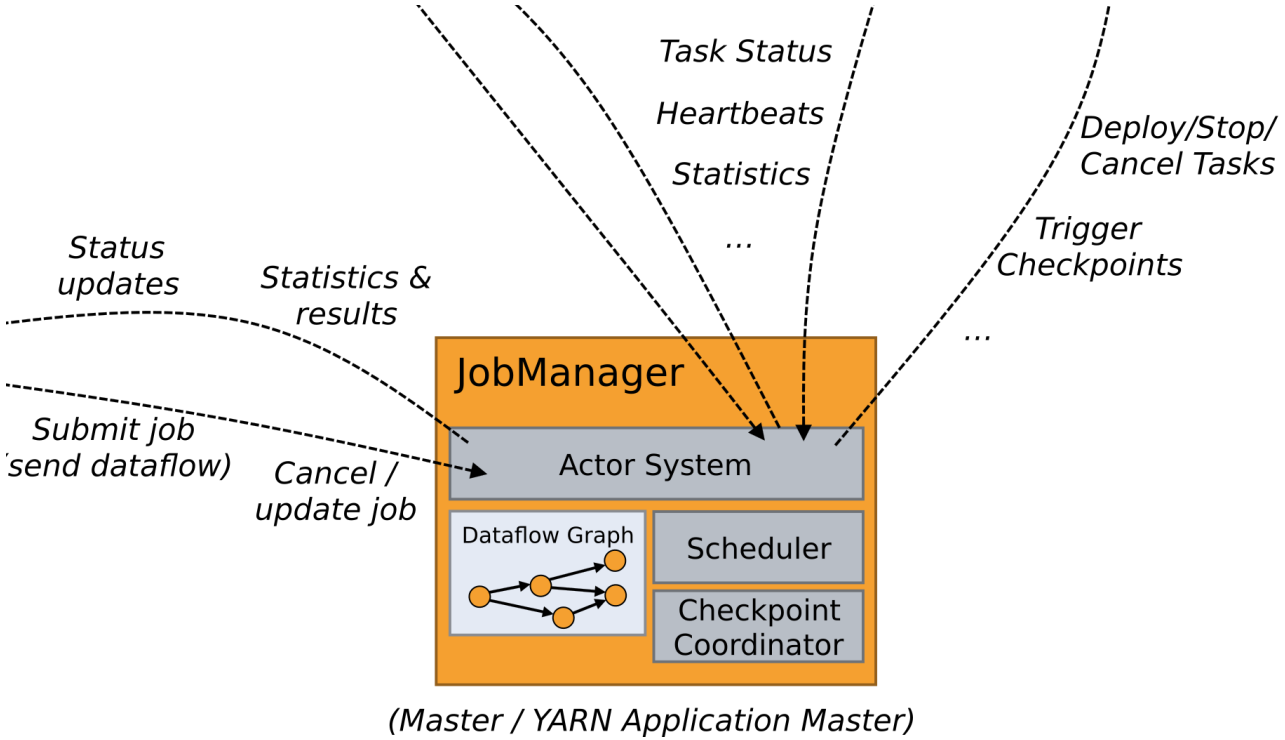


Apache Flink

Components: JobManager and TaskManager

JobManager

In charge of coordinating ad distributing execution of Flink application.

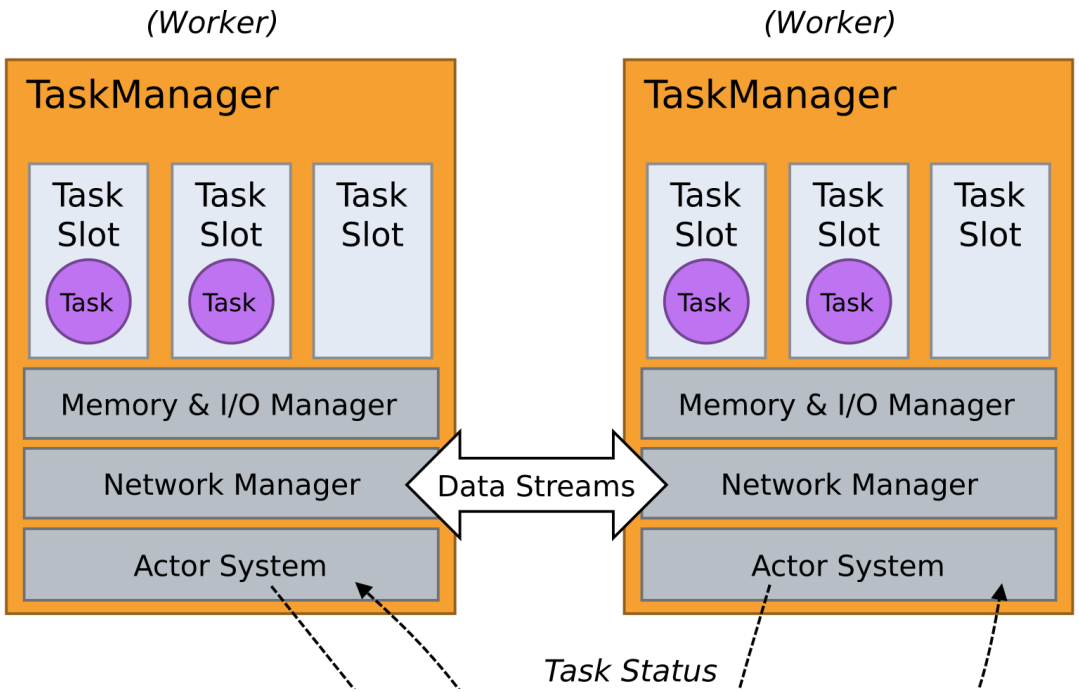


Apache Flink

Components: JobManager and TaskManager

TaskManagers

Execute tasks of a dataflow, and buffer and exchange the data streams.

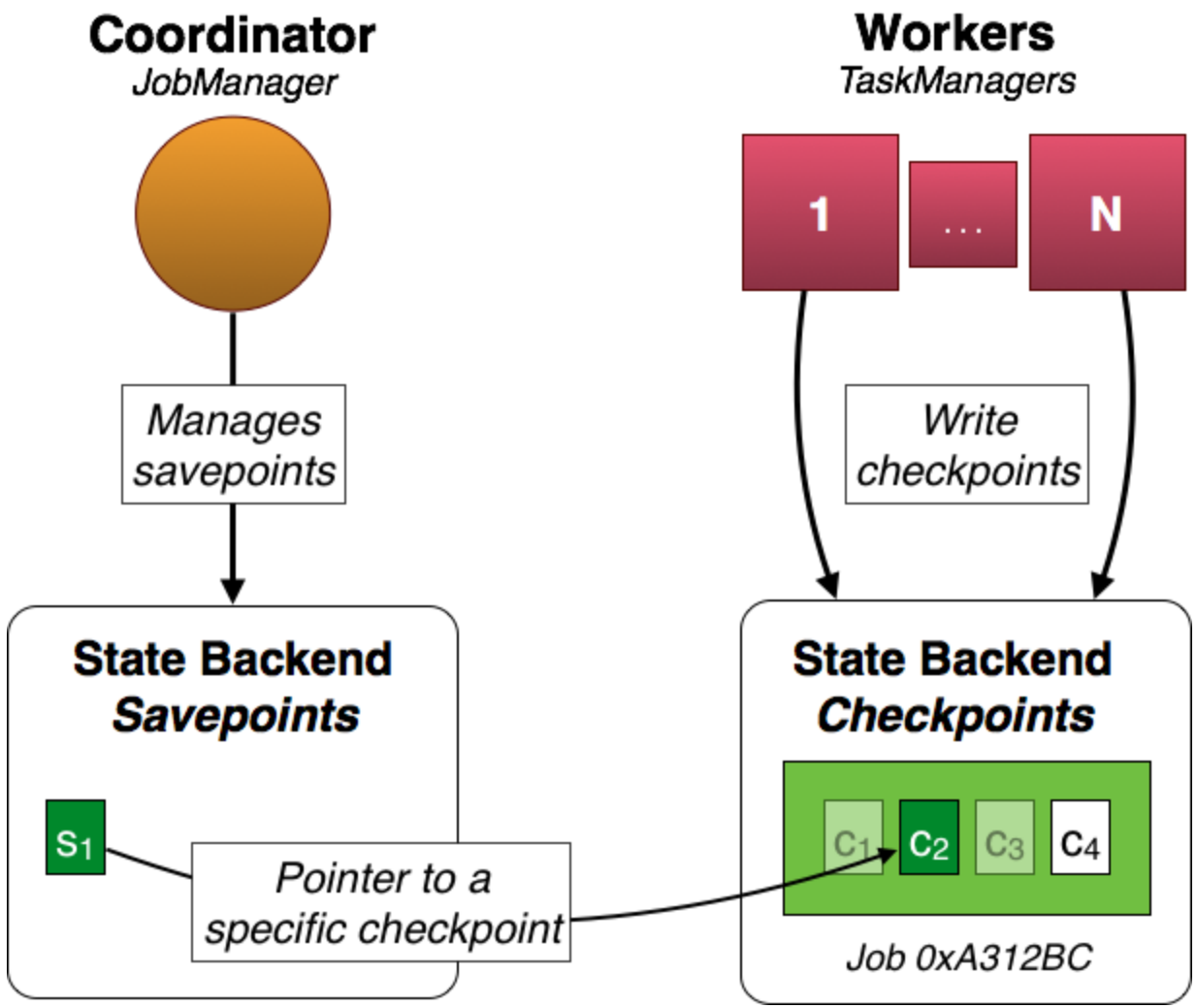


Apache Flink

Savepoint - Checkpoint

Checkpoint: managed by Flink, relied upon for failure recovery, triggered often. Lightweight and fast.

Savepoint: planned manual operations, as for example an update of Flink version.



Apache Flink

Checkpoint recovery

Checkpointing needs:

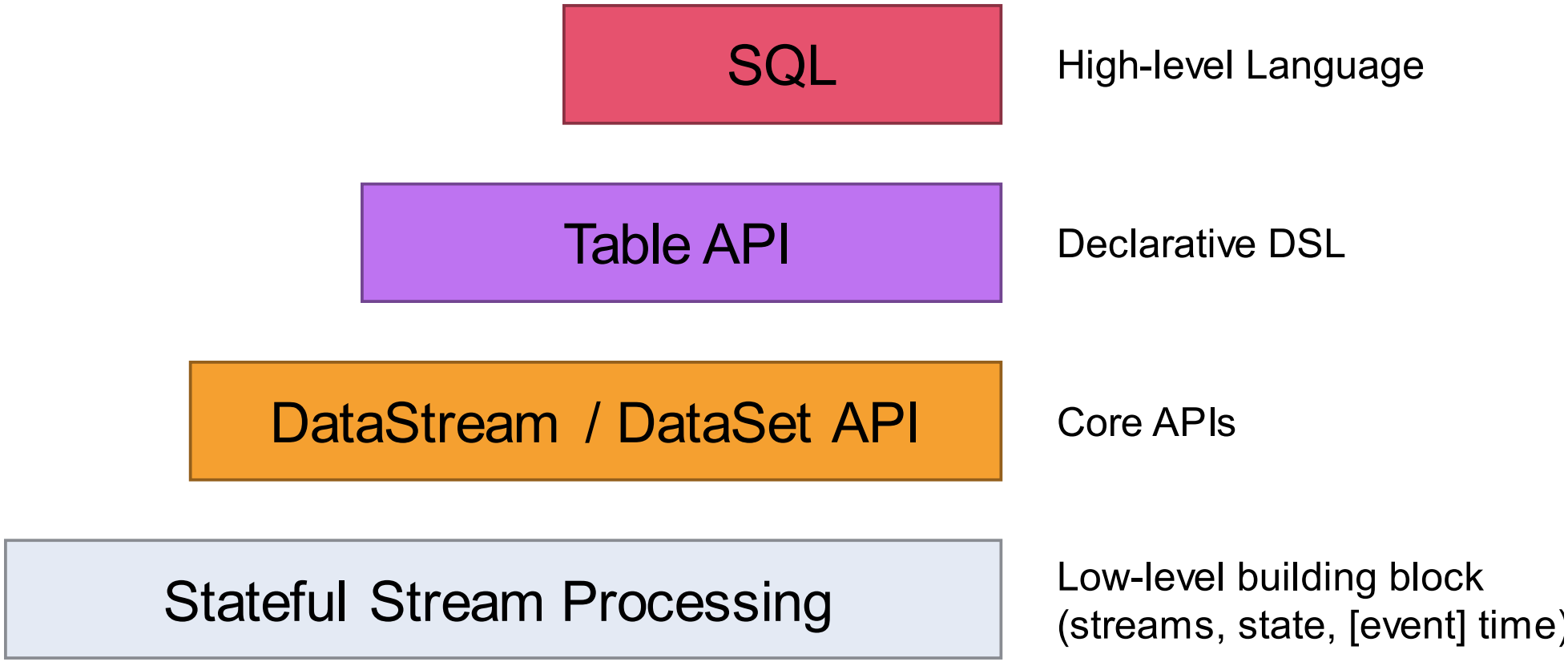
- Persistent data source that can replay records for a certain amount of time
  Kafka, RabbitMQ or file systems (HDFS, S3, Ceph)

- Persistent storage for state
  HDFS, S3, Ceph..

- Explicit activation

Apache Flink

SQL Abstraction

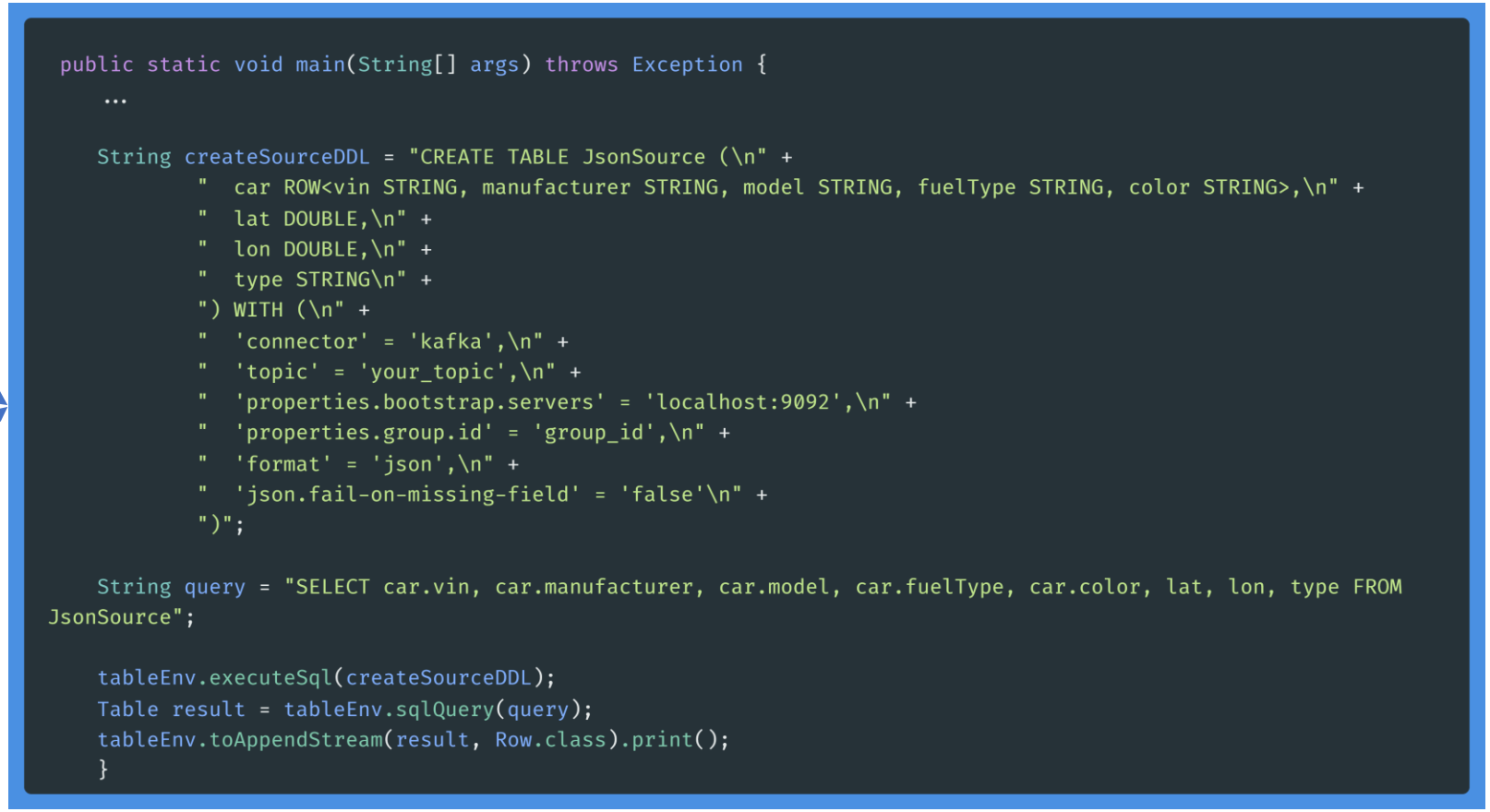


Apache Flink

SQL Example

```
{
   "car":{
      "vin":"PMWR7SPNJAMP38047",
      "manufacturer":"Acura",
      "model":"500X",
      "fuelType":"Compressed Natural Gas",
      "color":"Beige"
   },
   "lat":45.51322,
   "lon":9.30374,
   "type":"MOVE"
}
```

```
{
   "car":{
      "vin":"PELP3NP5CPPZ00076",
      "manufacturer":"Nissan",
      "model":"G9",
      "fuelType":"Compressed Natural Gas",
      "color":"Red"
   },
   "lat":45.51323,
   "lon":9.30381,
   "type":"MOVE"
}
```



```
public static void main(String[] args) throws Exception {
    ...

   String createSourceDDL = "CREATE TABLE JsonSource (\n" +
           "  car ROW<vin STRING, manufacturer STRING, model STRING, fuelType STRING, color STRING>,\n" +
           "  lat DOUBLE,\n" +
           "  lon DOUBLE,\n" +
           "  type STRING\n" +
           ") WITH (\n" +
           "  'connector' = 'kafka',\n" +
           "  'topic' = 'your_topic',\n" +
           "  'properties.bootstrap.servers' = 'localhost:9092',\n" +
           "  'properties.group.id' = 'group_id',\n" +
           "  'format' = 'json',\n" +
           "  'json.fail-on-missing-field' = 'false'\n" +
           ")";

   String query = "SELECT car.vin, car.manufacturer, car.model, car.fuelType, car.color, lat, lon, type FROM
JsonSource";

    tableEnv.executeSql(createSourceDDL);
    Table result = tableEnv.sqlQuery(query);
    tableEnv.toAppendStream(result, Row.class).print();
    }
```

Apache Flink

Integration with Apache Kafka

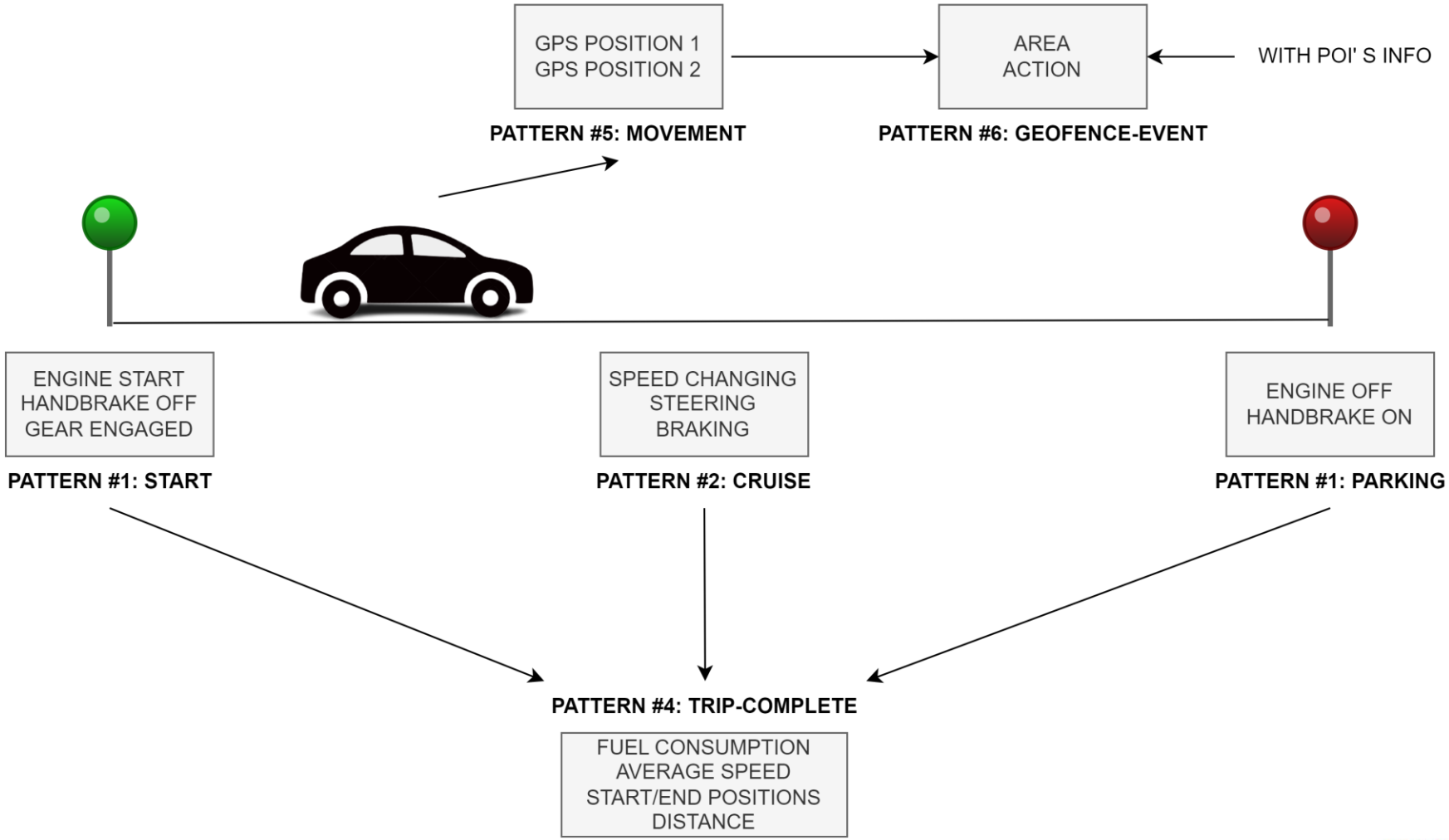
```
KafkaSource<String> source = KafkaSource.<String>builder()
    .setBootstrapServers(brokers)
    .setTopics("input-topic")
    .setGroupId("my-group")
    .setStartingOffsets(OffsetsInitializer.earliest())
    .setValueOnlyDeserializer(new SimpleStringSchema())
    .build();

env.fromSource(source, WatermarkStrategy.noWatermarks(), "Kafka Source");
```

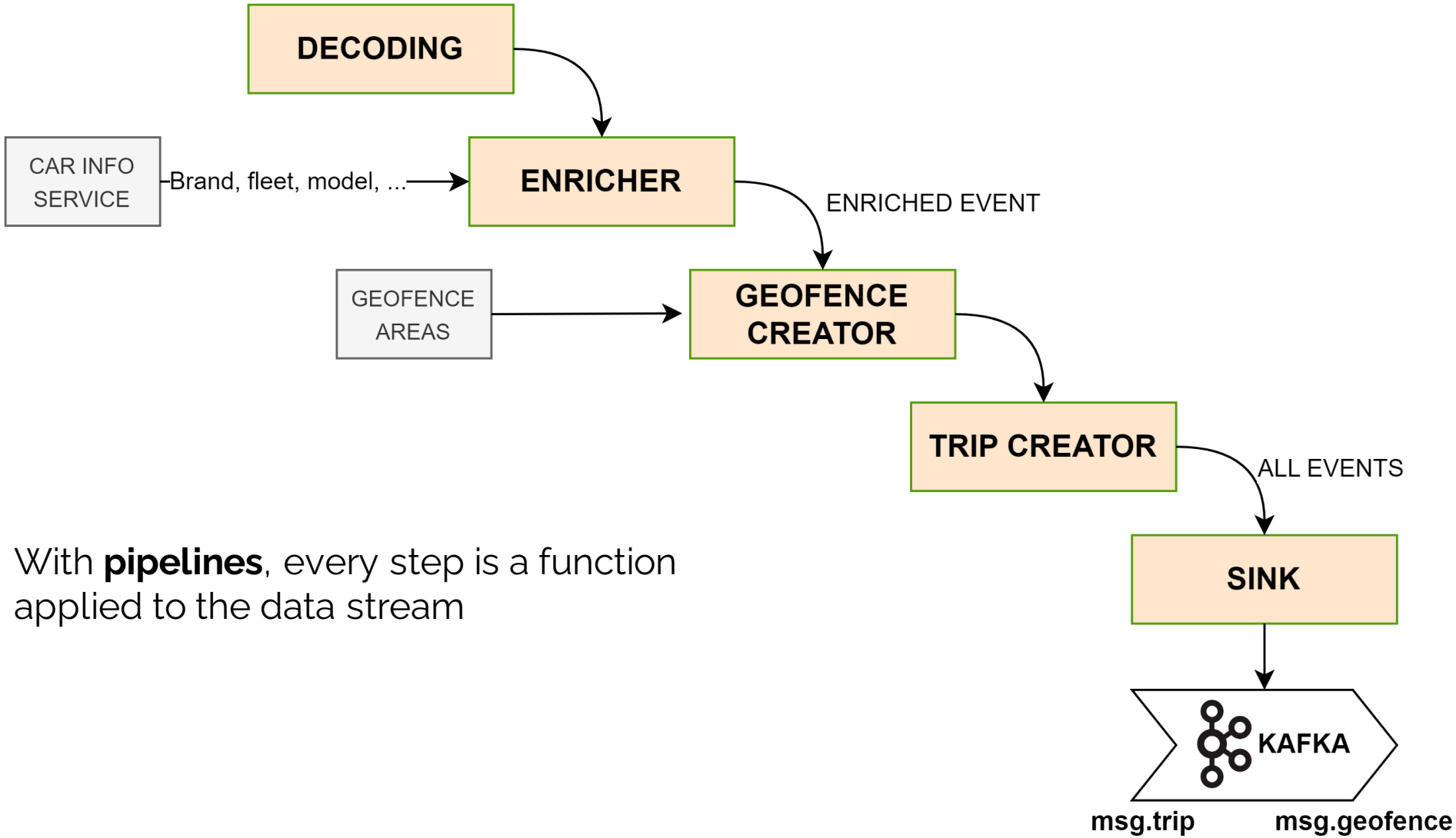
Use case

Meaning pattern

A set of data that, once aggregated, forms more structured information



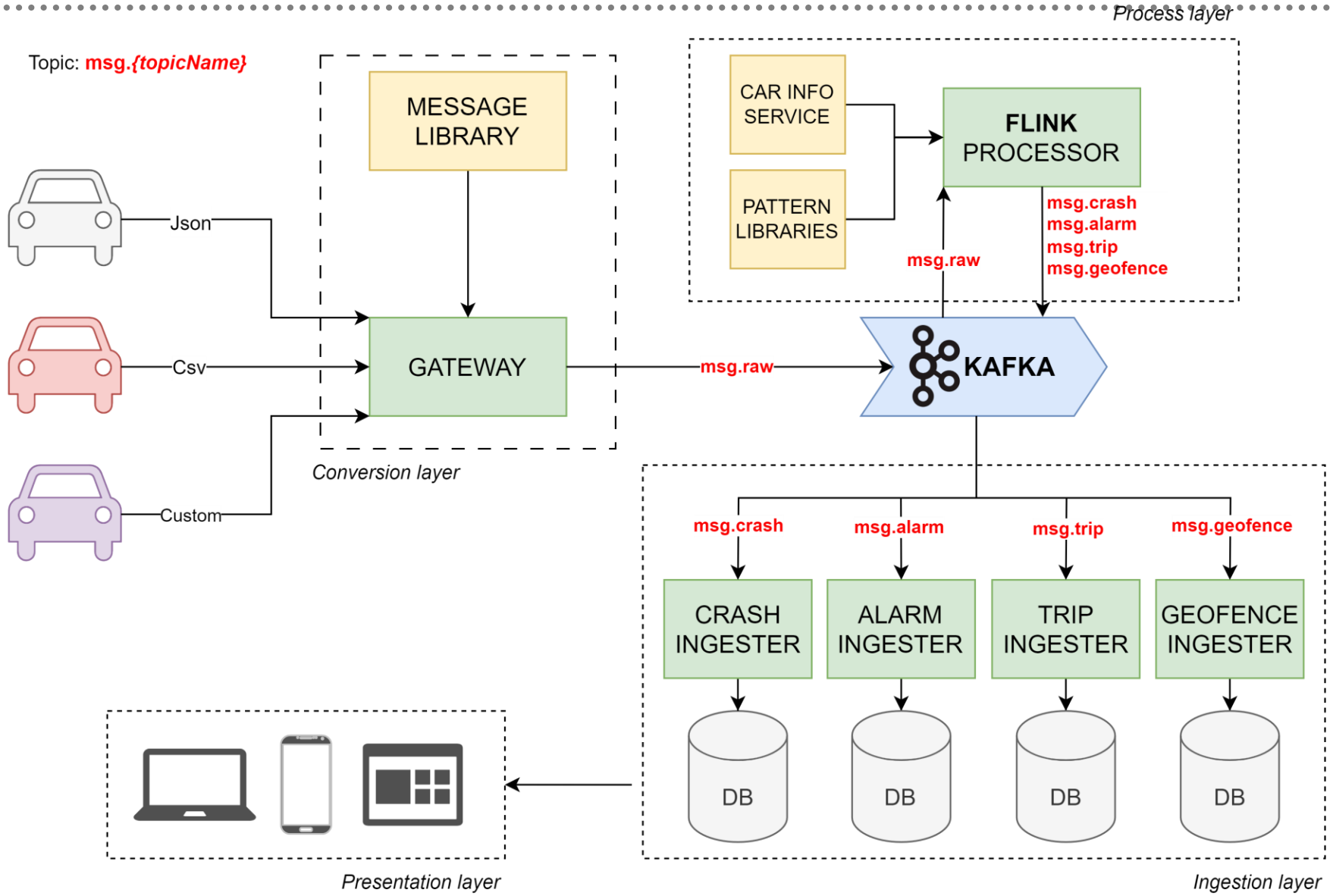
Build of a pattern



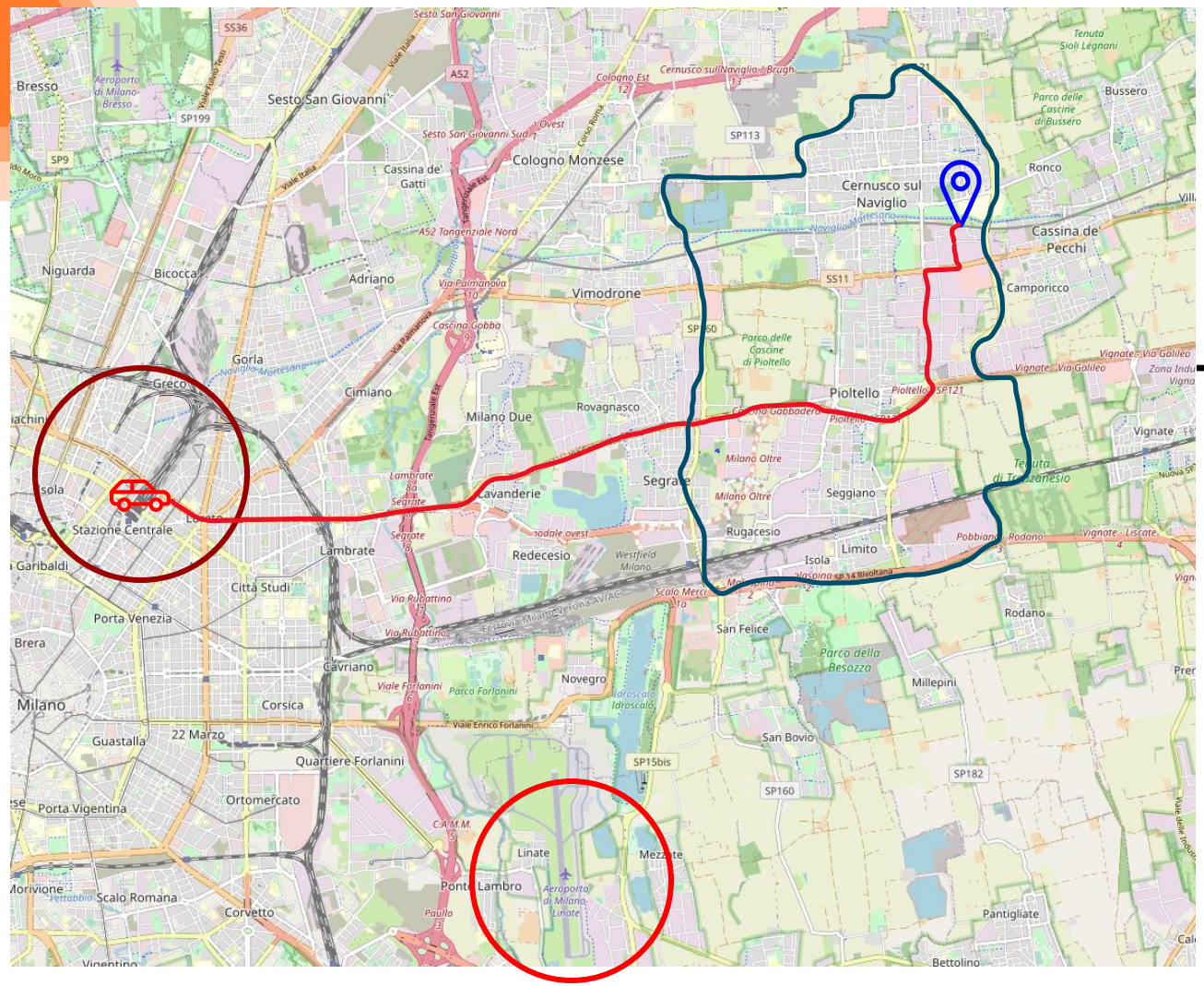
With **pipelines**, every step is a function applied to the data stream

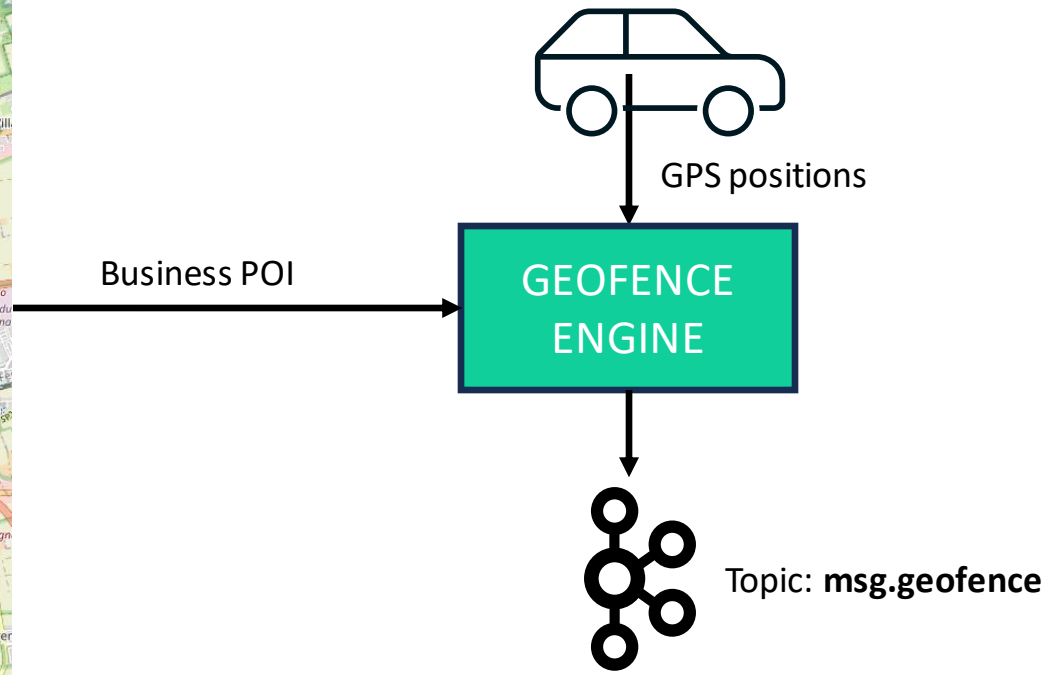
Architecture so far #2

We refer to whole architecture as “Geofence engine”



“Value”: from raw data to geofencing





{device: 1 , action: start, poi: milano_centrale}

{device: 1 , action: exit, poi: milano_centrale}

. . .

{device: 1 , action: enter, poi: office_area}

{device: 1 , action: end, poi: headquarter}

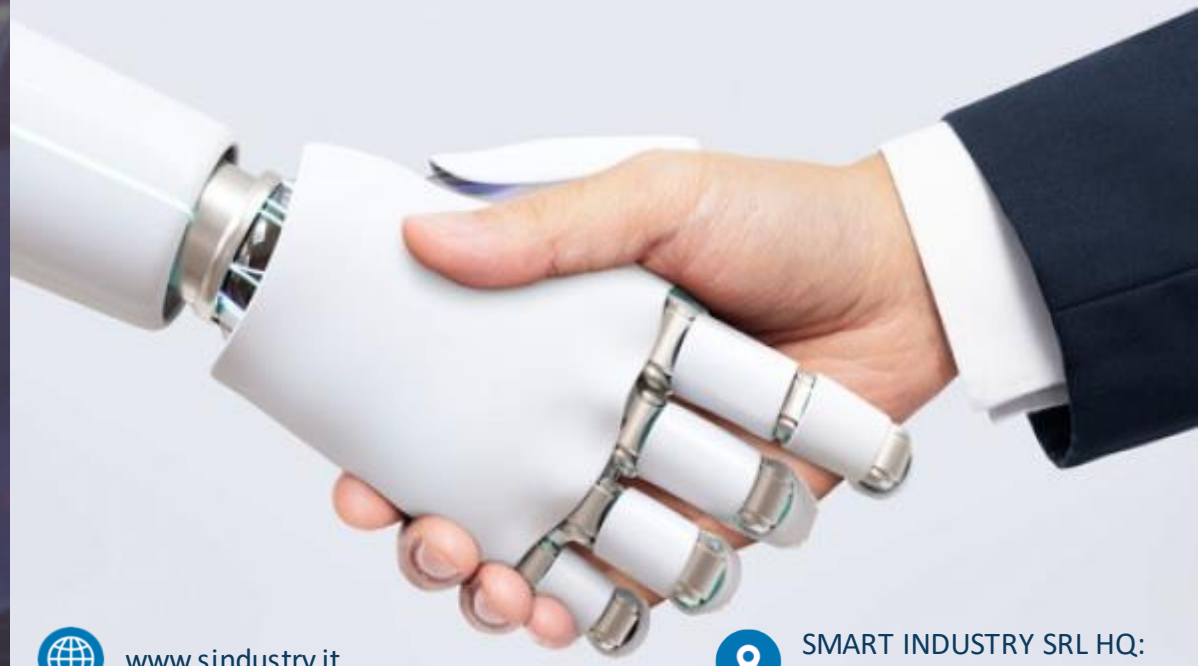
SMART INDUSTRY®
EXCELLENT FUTURE CONDITION

Questions about collaborations or opportunities?

Feel free to ask!

SMART INDUSTRY®
EXCELLENT FUTURE CONDITION

REAL TIME VEHICLE MONITORING AND ANALYSIS



www.sindustry.it

SMART INDUSTRY

fabio.riccicurbastro@sengineering.it

SMART INDUSTRY SRL HQ:
VIA PROGRESSO, 40/42
40064 OZZANO EMILIA (BO)

051-6511574 (int 2205)