Stream processing - part III (real-time data architecture)

Bartosz Łoś
RTB House

Practical Distributed Systems, 2022

What will this lecture be about?

Goal:

- show an example of real-time data architecture at RTB House (different approaches and use cases, design decisions)
- dig deep into data processing frameworks (Apache Storm, Kafka Streams, Kafka Workers etc.)

The context: RTB platform

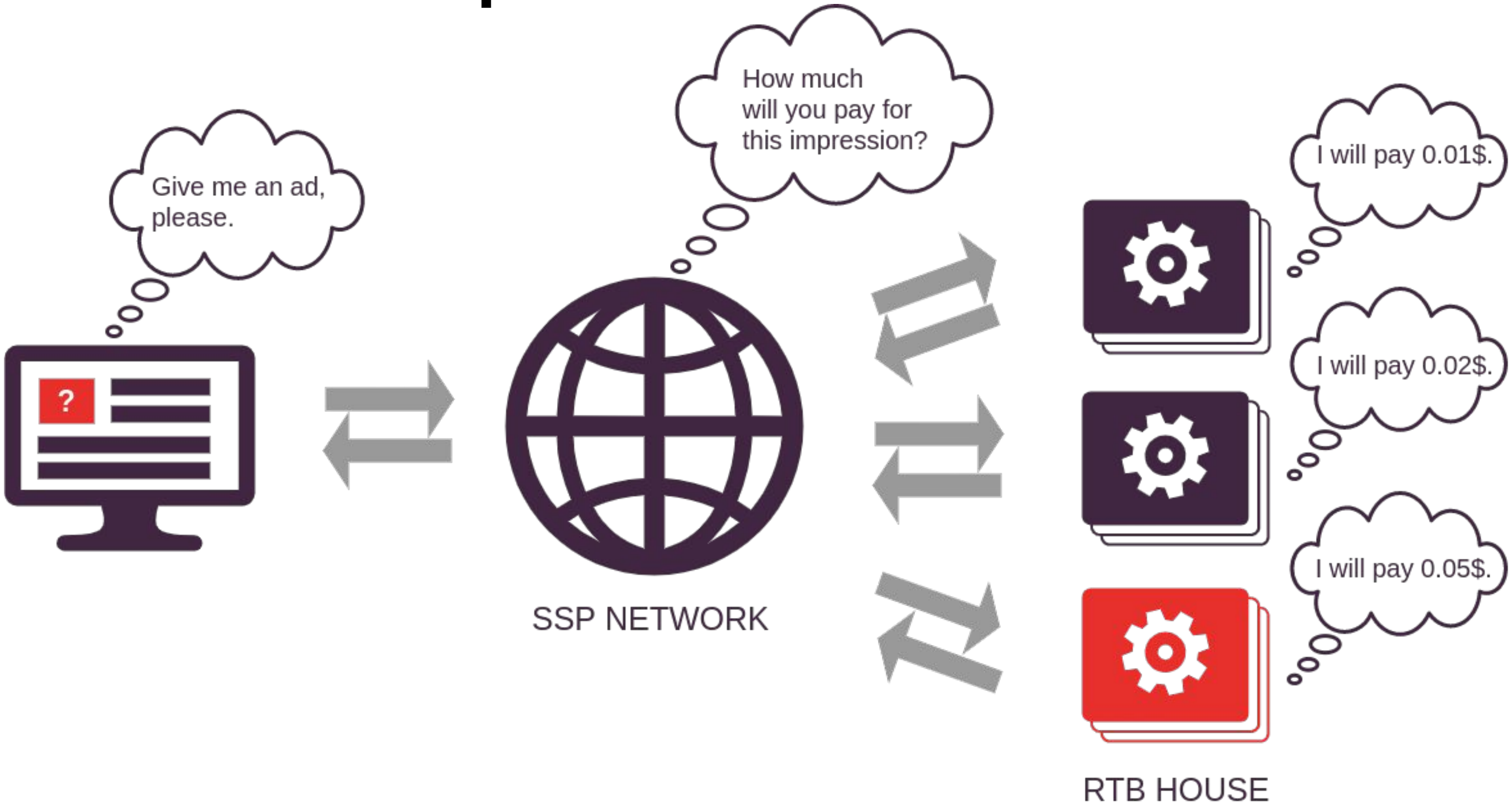
Our platform:

- takes part in auctions, purchases and emits advertisements in the RTB model
- processes **5M bid requests per second** and generates **300K events per second** (**200TB data every day**)

Data processing:

- requirements: machine learning, system monitoring (alerting, ad hoc debugging) and financial settlements (reports, budget limits)
- use cases: filtering, synchronizing, joining, aggregating, storing events and statistics in Hadoop, BigQuery, Postgres or Elasticsearch

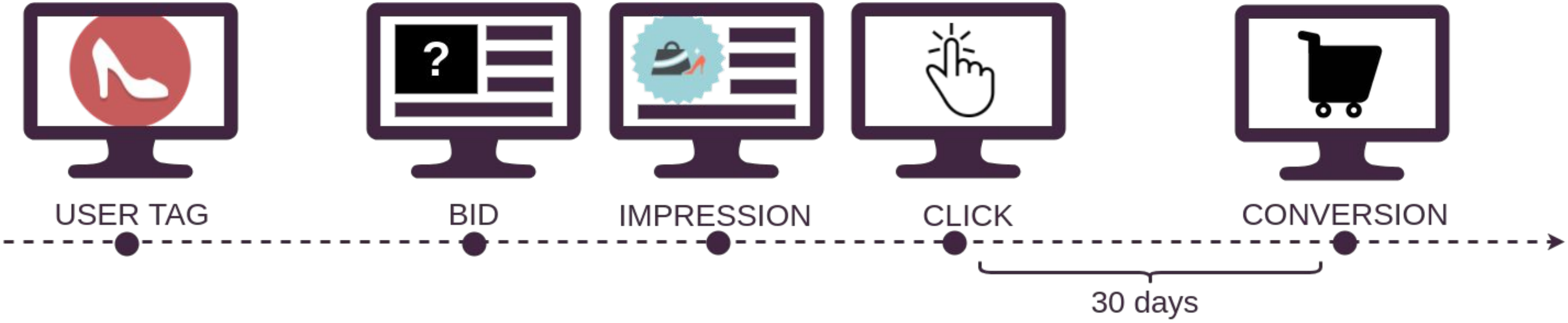
The context: RTB platform



The context: RTB platform

Our platform consists of two types of servlets:

- **bidders** process **bid requests**
- **adservlets** process **user requests** (tags, impressions, clicks and conversions)



The context: RTB platform

To be able to buy advertising space effectively, we needed to store and process data (user info, historical impressions)

We were able to use this data for estimating:

- probability of a click (click-through rate) - **CTR**
- conditional probability of a conversion given that an impression was clicked (conversion rate) - **CR**
- conversion value - **CV**

These estimated values are used for bid pricing:

bid_value = (1-margin) * **CTR** * **CR** * **CV** * rate

Iterations

We have been improving our solution by many iterations:

- at first: end-of-day batch jobs, single-DC, inconsistent data-flows
- finally: real-time data processing, delay reduced from 1 day to 15 seconds, multi-DC architecture, end-to-end exactly-once processing

It was essential to:

- separate data-flow from the core platform
- provide immutable streams of events and data synchronization between DCs
- dig deep into open-source streaming technologies and if needed replace them by better, custom-built components

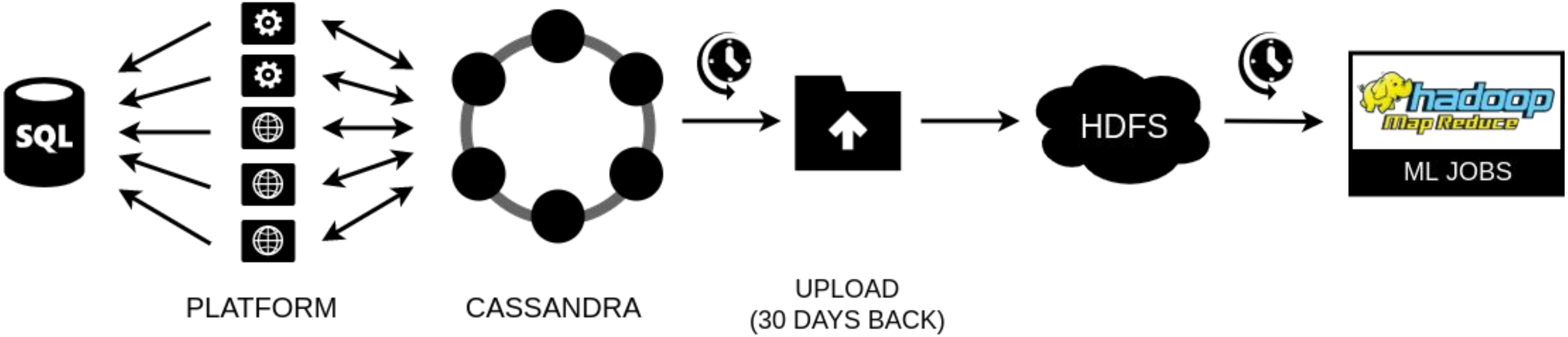
The 1st iteration: mutable impressions

```
{ IMPRESSION:
    IMPRESSION_HASH,
    TIME,
    COOKIE,
    ADVERTISER_ID,
    ...
    CLICKS,
    CONVERSIONS
}
```

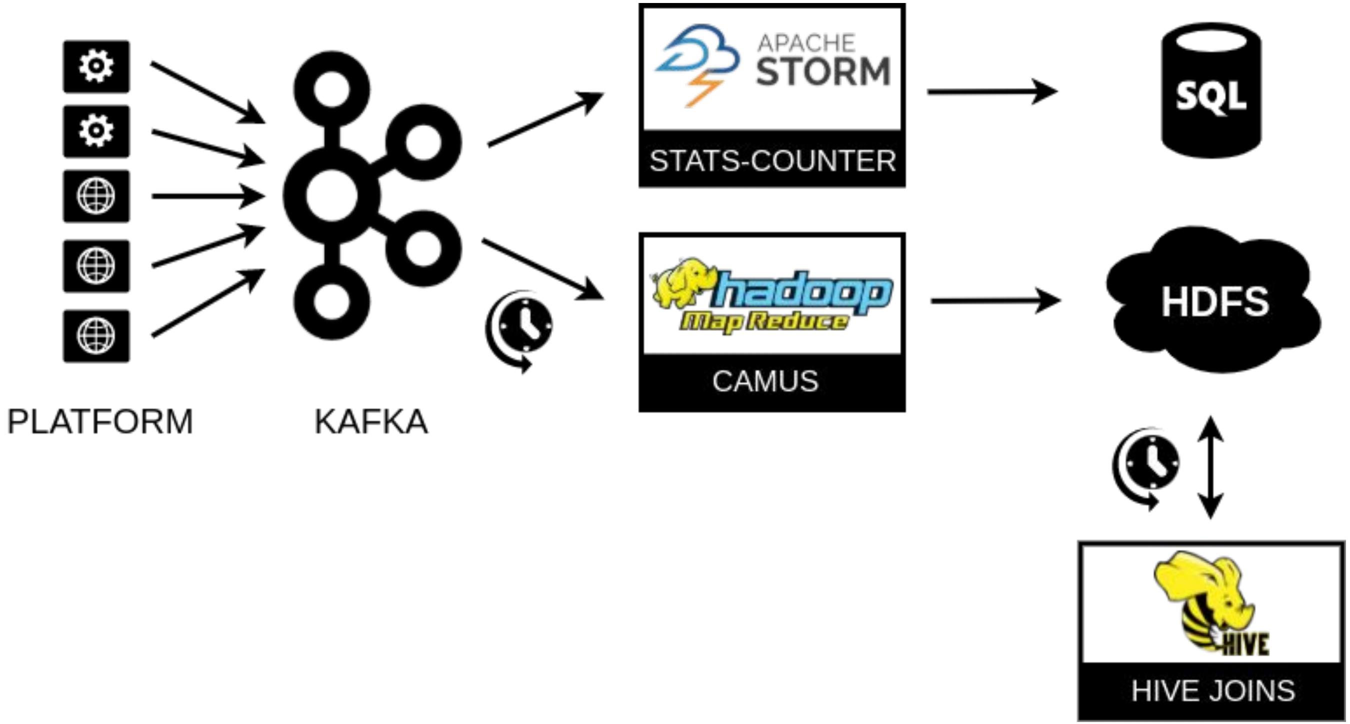
```
{ CLICK:
    CLICK_HASH,
    TIME,
    COOKIE,
    ADVERTISER_ID,
    IMPRESSION_HASH,
    ...
}
```

```
{ CONVERSION:
    CONVERSION_HASH,
    TIME,
    COOKIE,
    ADVERTISER_ID,
    ...
}
```

The 1st iteration: mutable impressions



The 2nd iteration: data-flow



The 3rd iteration: immutable streams of events

```
{ IMPRESSION:
    IMPRESSION_HASH,
    TIME,
    COOKIE,
    ADVERTISER_ID,
    ...
    CLICKS,
    CONVERSIONS
}
```

```
{ CLICK:
    CLICK_HASH,
    TIME,
    COOKIE,
    ADVERTISER_ID,
    IMPRESSION_HASH,
    ...
}
```

```
{ CONVERSION:
    CONVERSION_HASH,
    TIME,
    COOKIE,
    ADVERTISER_ID,
    ...
}
```

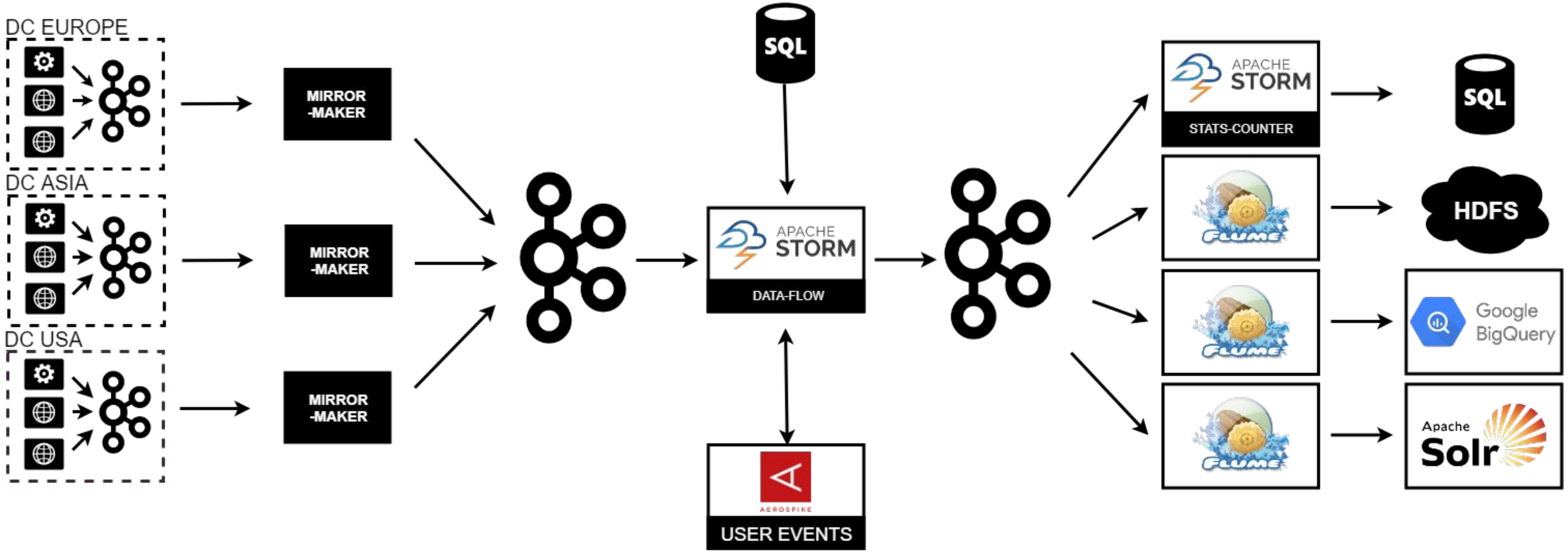
The 3rd iteration: immutable streams of events

{ IMPRESSION:
IMPRESSION_HASH,
TIME,
COOKIE,
ADVERTISER_ID,
...
~~CLICKS~~,
~~CONVERSIONS~~
}

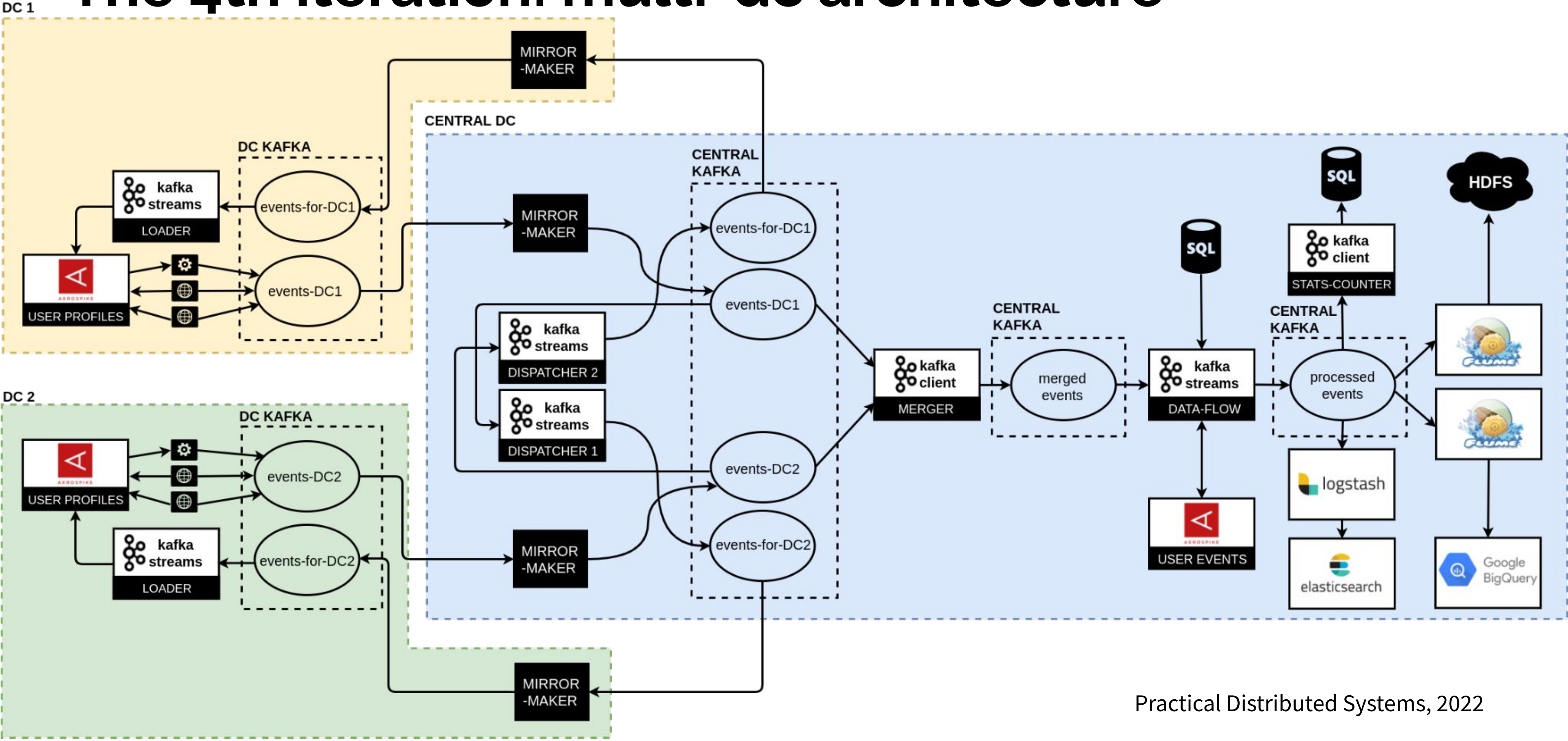
{ CLICK:
CLICK_HASH,
TIME,
COOKIE,
ADVERTISER_ID,
~~IMPRESSION_HASH,~~
...
IMPRESSION
}

{ CONVERSION:
CONVERSION_HASH,
TIME,
COOKIE,
ADVERTISER_ID,
...
IMPRESSION,
CLICK
}

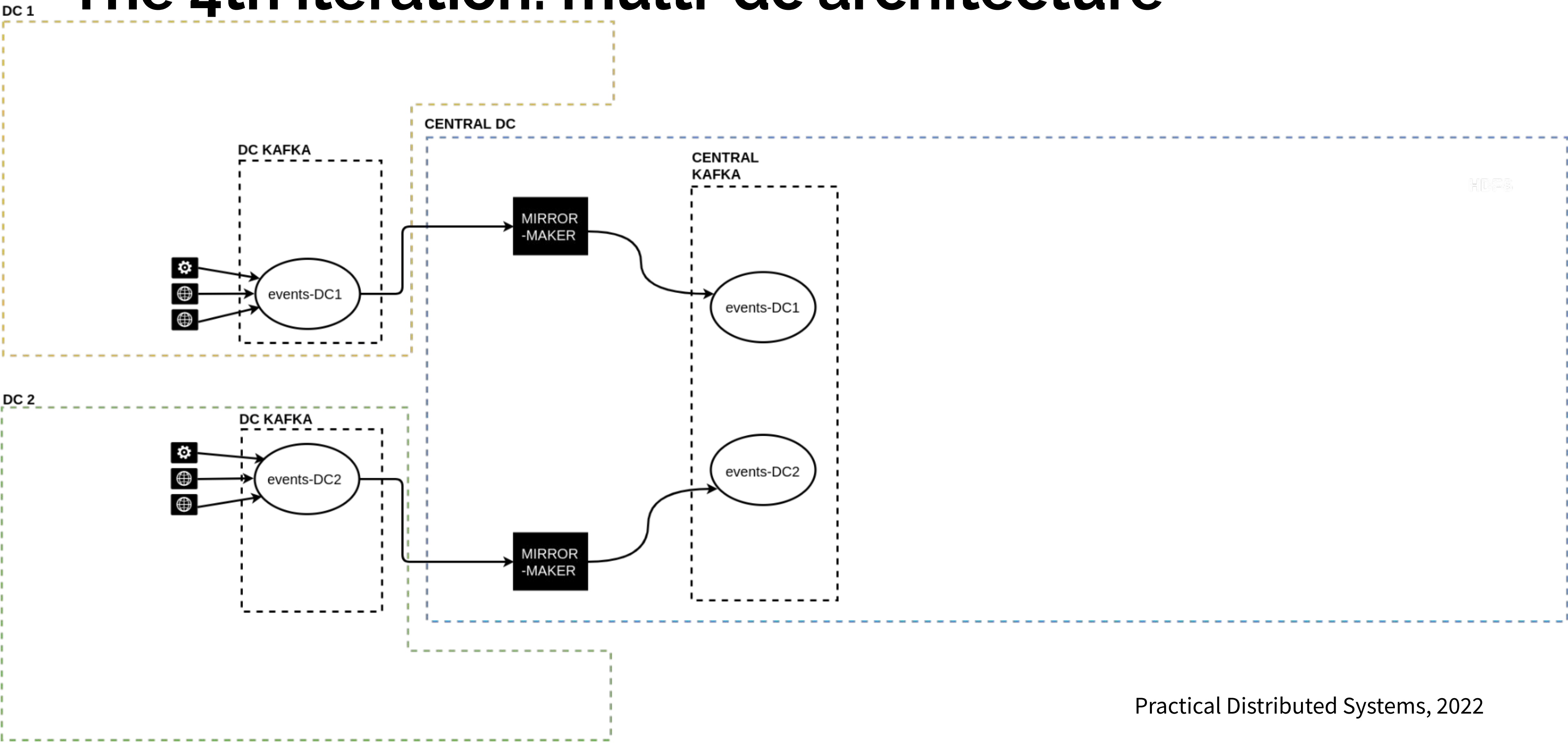
The 3rd iteration: immutable streams of events



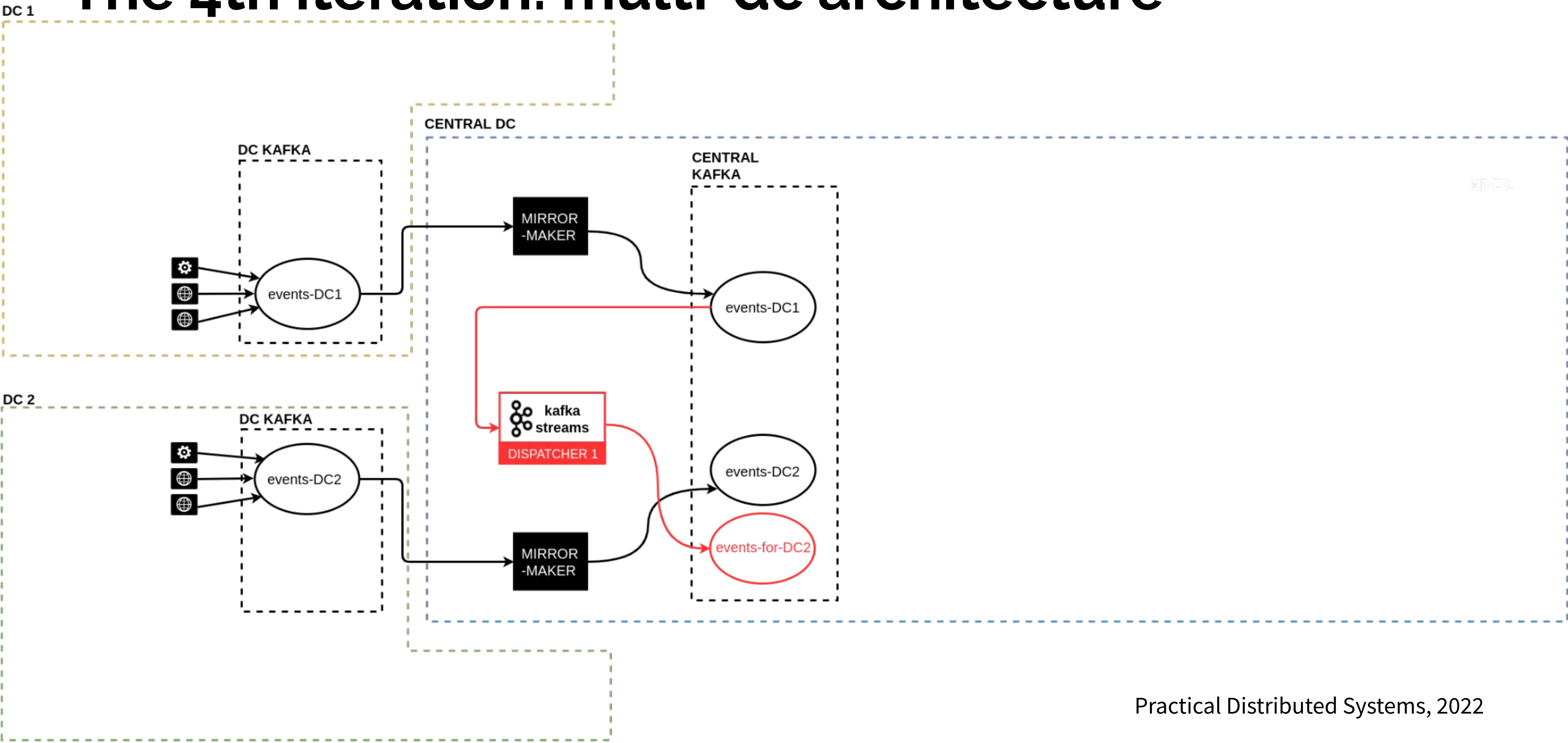
The 4th iteration: multi-dc architecture



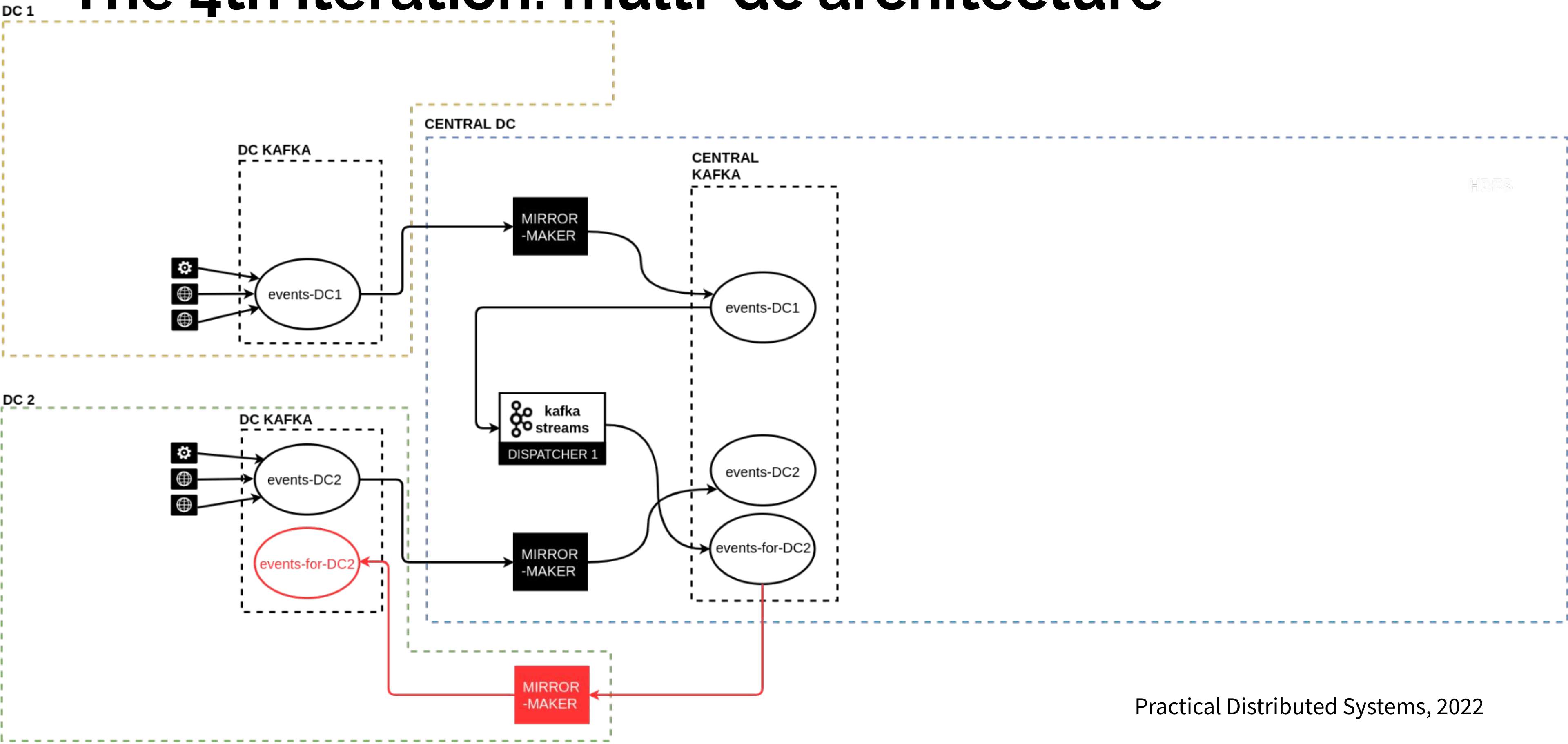
The 4th iteration: multi-dc architecture



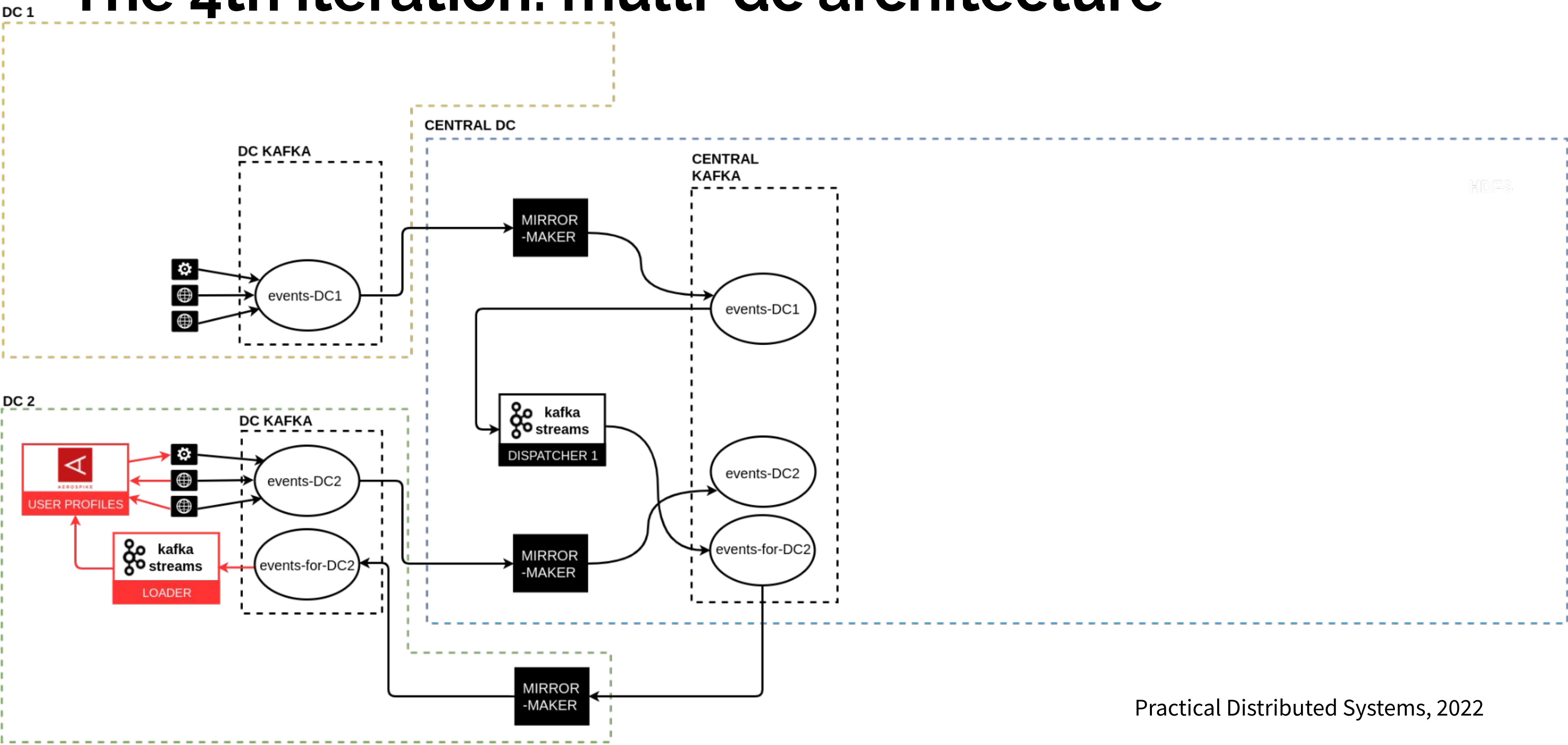
The 4th iteration: multi-dc architecture



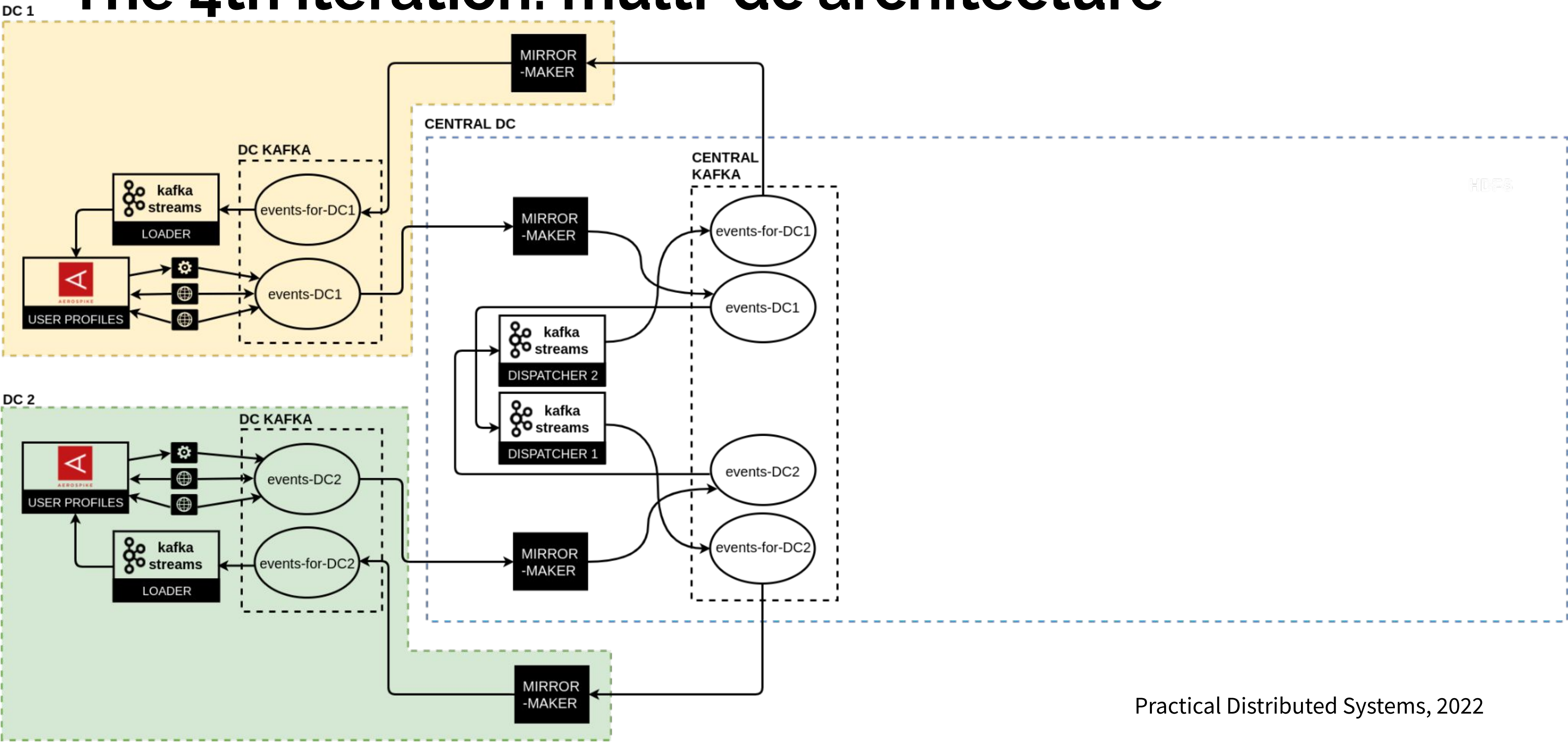
The 4th iteration: multi-dc architecture



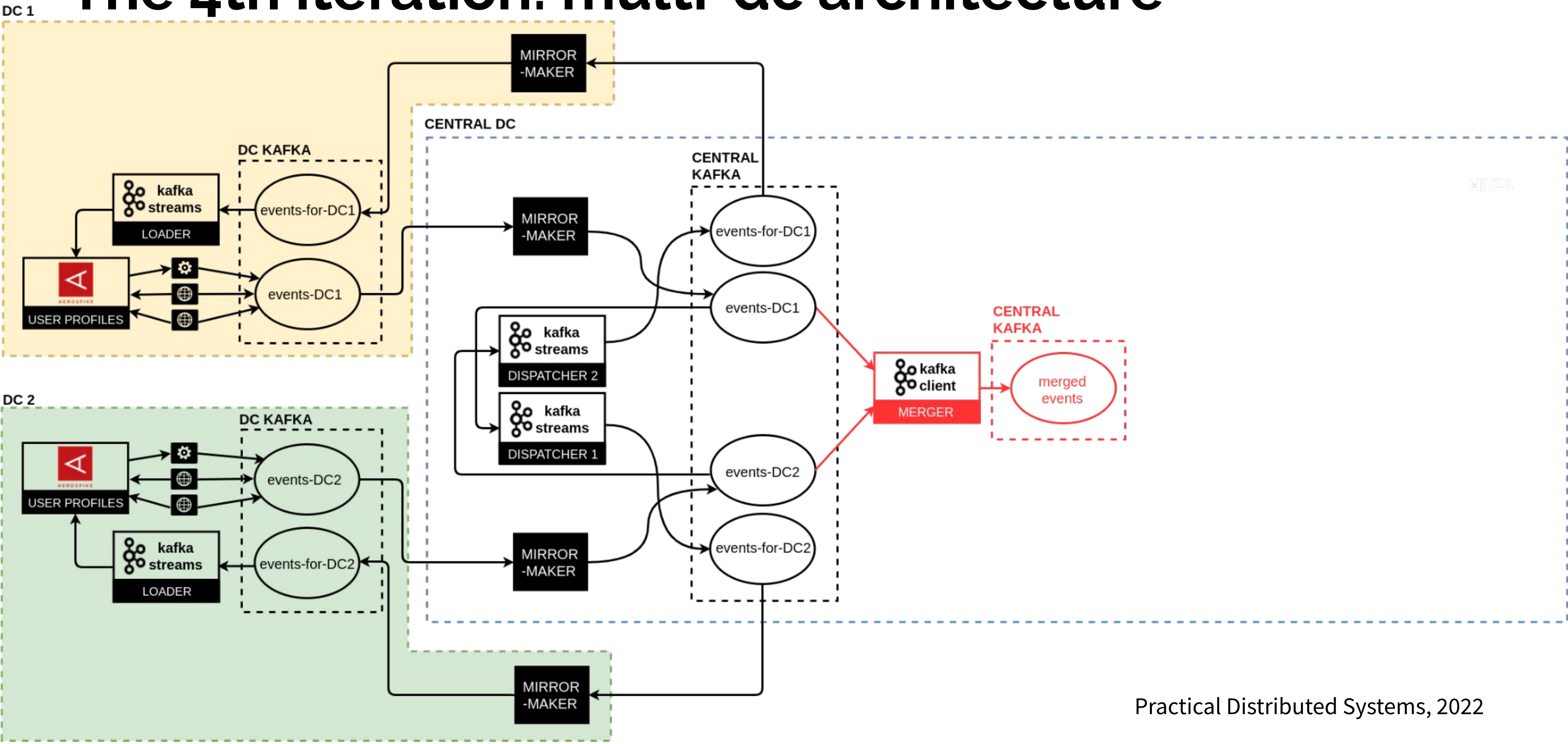
The 4th iteration: multi-dc architecture



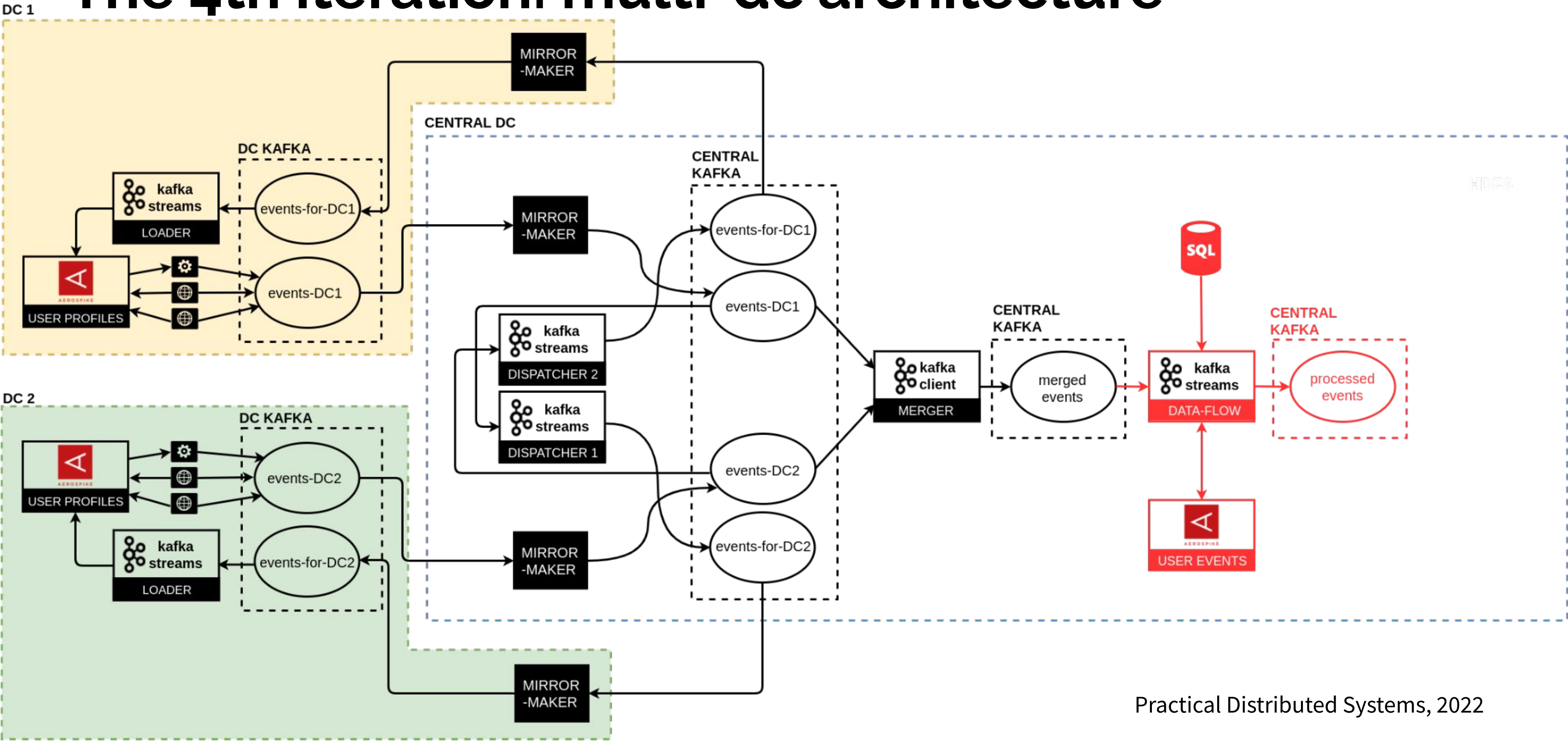
The 4th iteration: multi-dc architecture



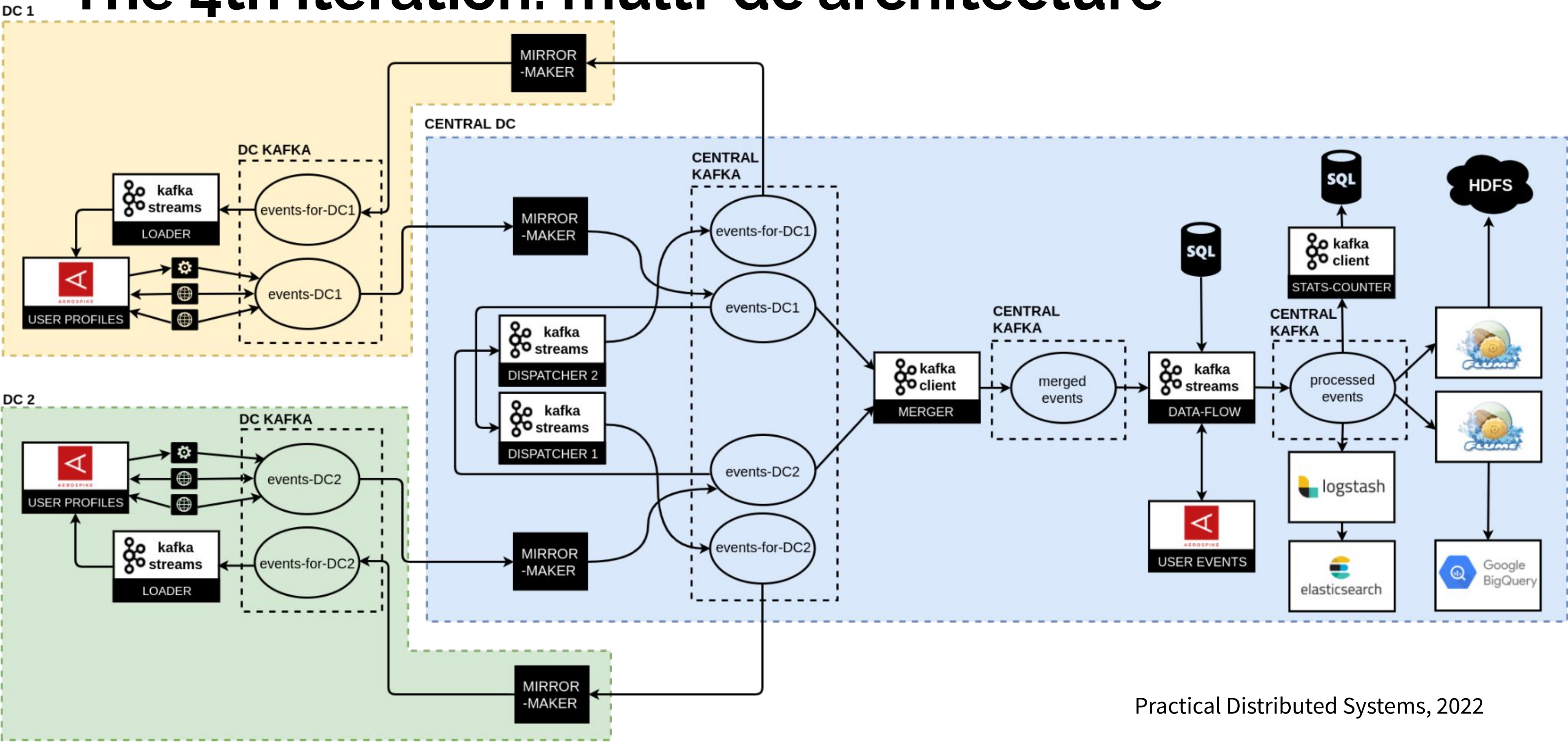
The 4th iteration: multi-dc architecture



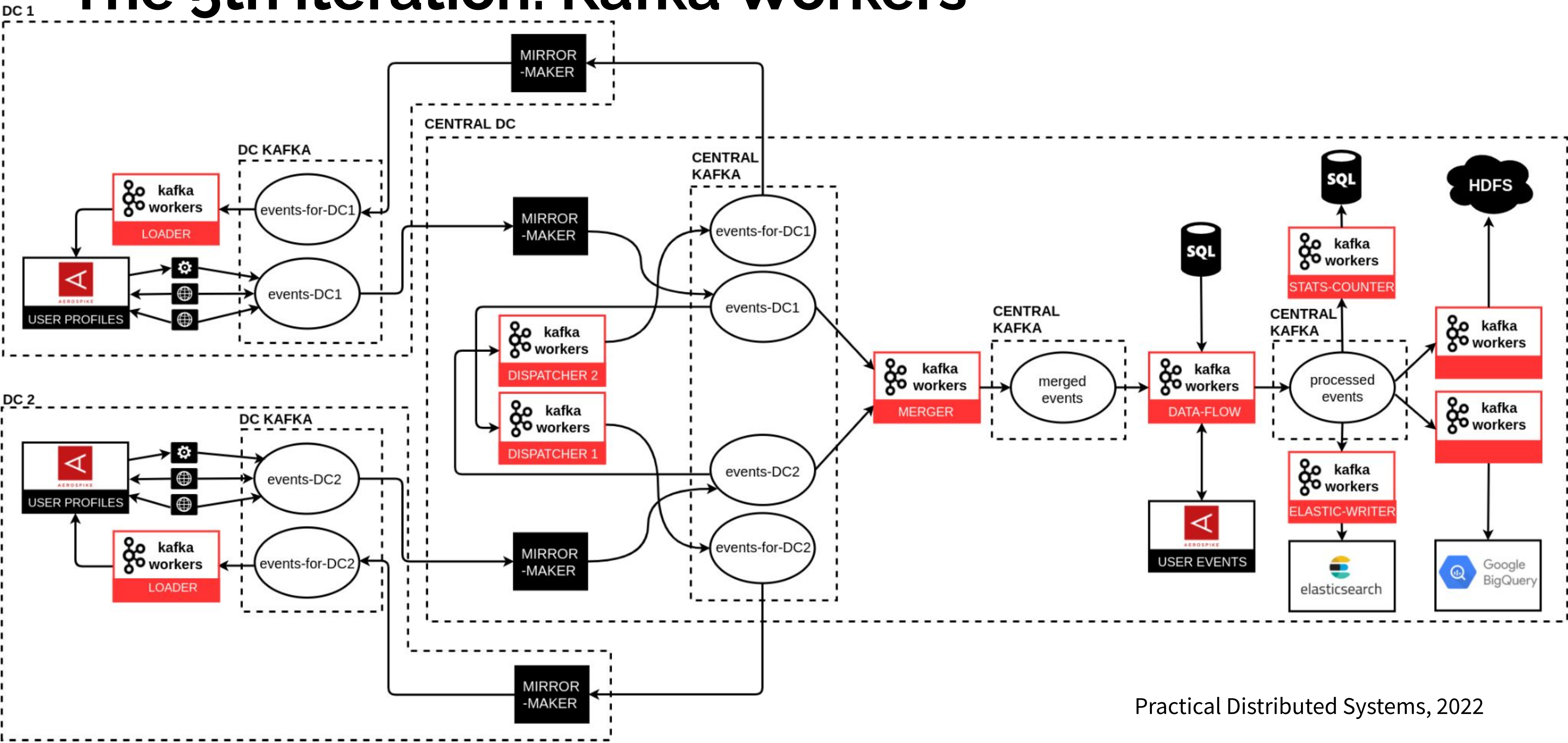
The 4th iteration: multi-dc architecture



The 4th iteration: multi-dc architecture



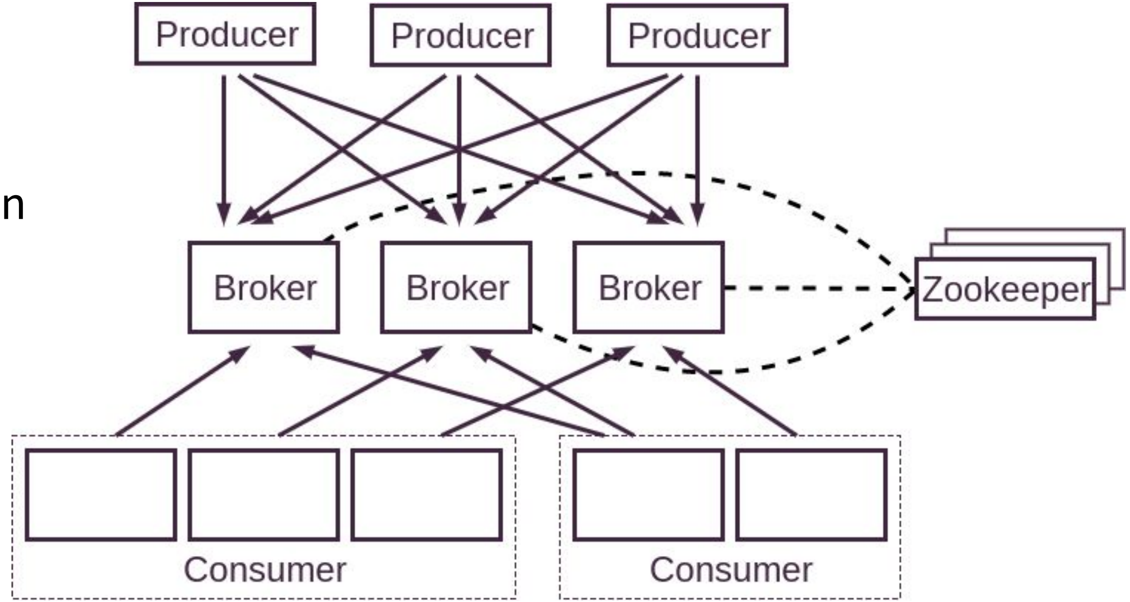
The 5th iteration: Kafka Workers



Apache Kafka

Why Kafka:

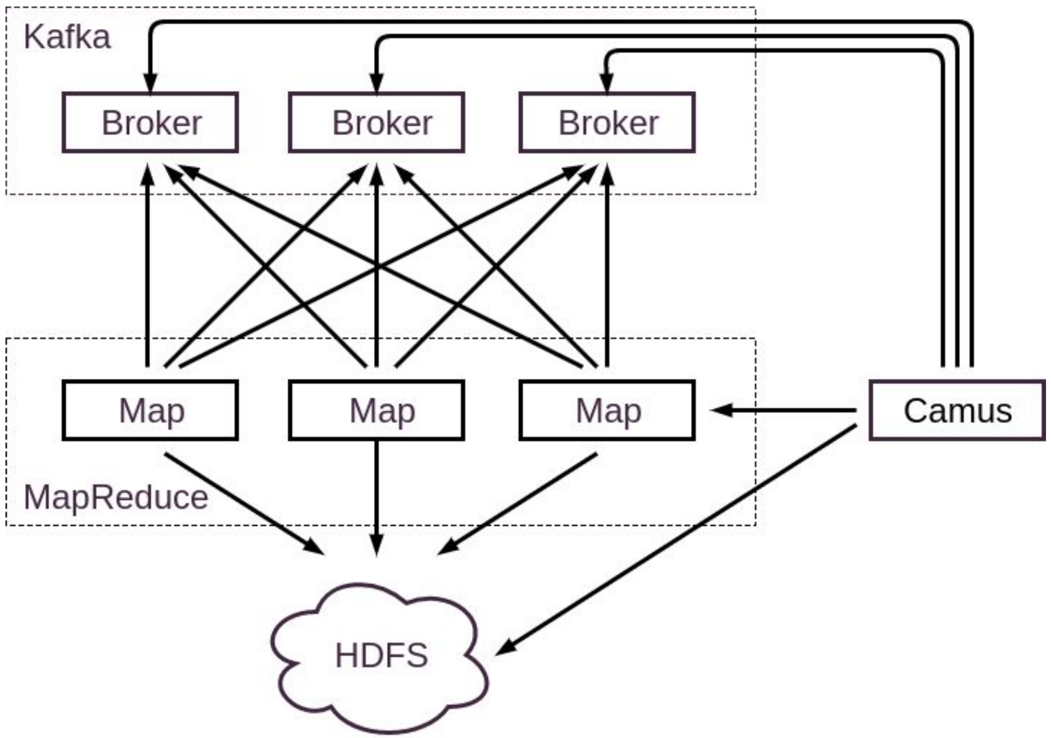
- event streaming platform (**distributed log)**
- producer-consumer separation
- fault-tolerance (**replication**)
- scalability and distribution (**topics partitioning**)
- log retention, statelessness
- efficient data consumption



Apache Camus

Why Camus:

- **MapReduce** job that incrementally **loads data from Kafka into HDFS**
- fetches topics from Zookeeper and latest offsets from Kafka
- **partitions the output** based on the timestamp of each record
- **stores offsets** in log files in HDFS



Apache Avro

Why Apache Avro:

- data serialization framework
- stores data in a compact, efficient **binary format**
- schema (JSON) could define rich data structures using various complex types
- schema is stored with data in one **Avro file** (self-describing container files)
- supports **schema changes** (old schema could be deserialized by a new program)

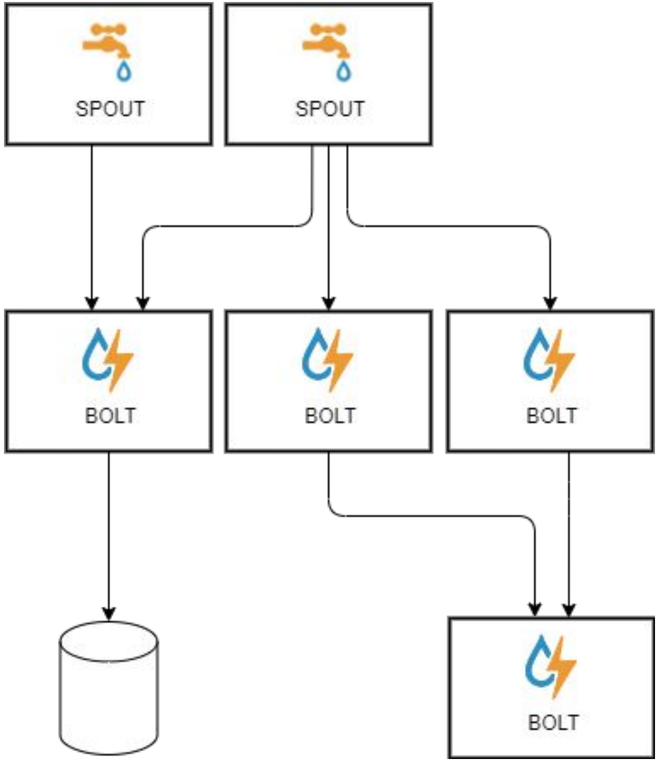
Our approach:

- Kafka's messages and HDFS files
- schema registry, historical schemas for Avro deserialization
- avro-fastserde (github.com/RTBHOUSE/avro-fastserde)

Apache Storm

Why Apache Storm:

- real-time computation system
- processes **streams** of **tuples** and runs user-defined **topologies** with processing nodes:
  - **spouts** emit new tuples
  - **bolts** receive tuples, do processing and generate tuples (**states** persist information)
- guarantees that every spout tuple will be fully processed (fault-tolerance)
- executes spouts and bolts as individual **tasks** that run in parallel on multiple machines



Apache Storm

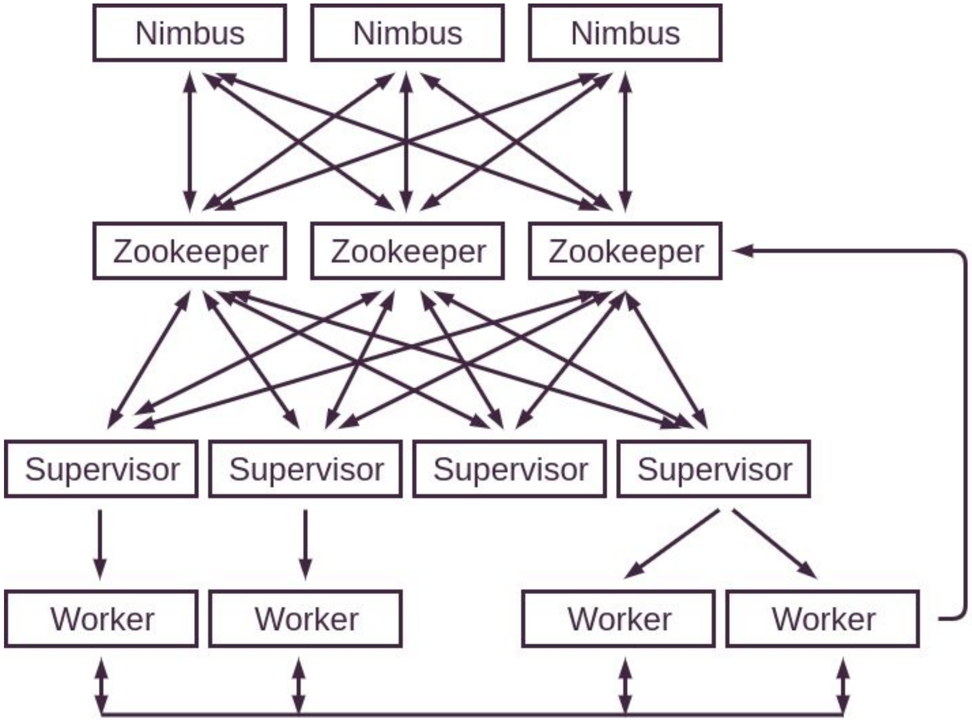
Master runs a daemon - **Nimbus**:

- responsible for distributing code around the cluster, assigning tasks to machines, and monitoring

Worker runs a daemon - **Supervisor**:

- listens for work assigned to its machine and starts and stops worker processes

Each **worker process** is a physical JVM and executes a subset of all the tasks for the topology

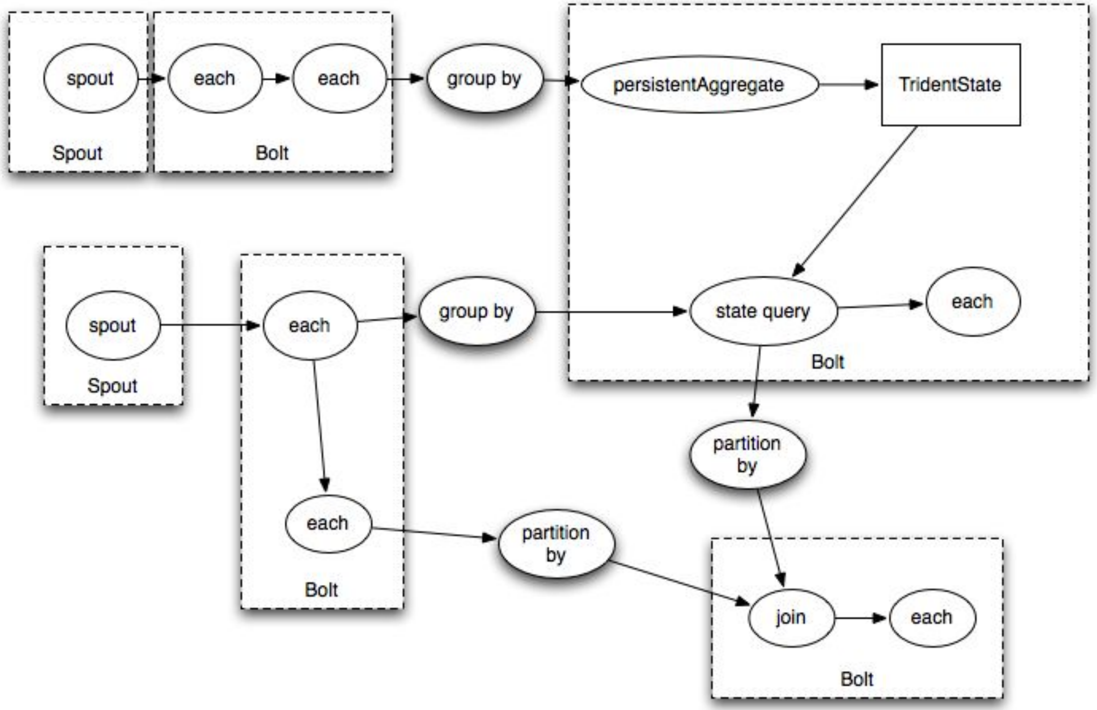


Apache Storm: Trident API

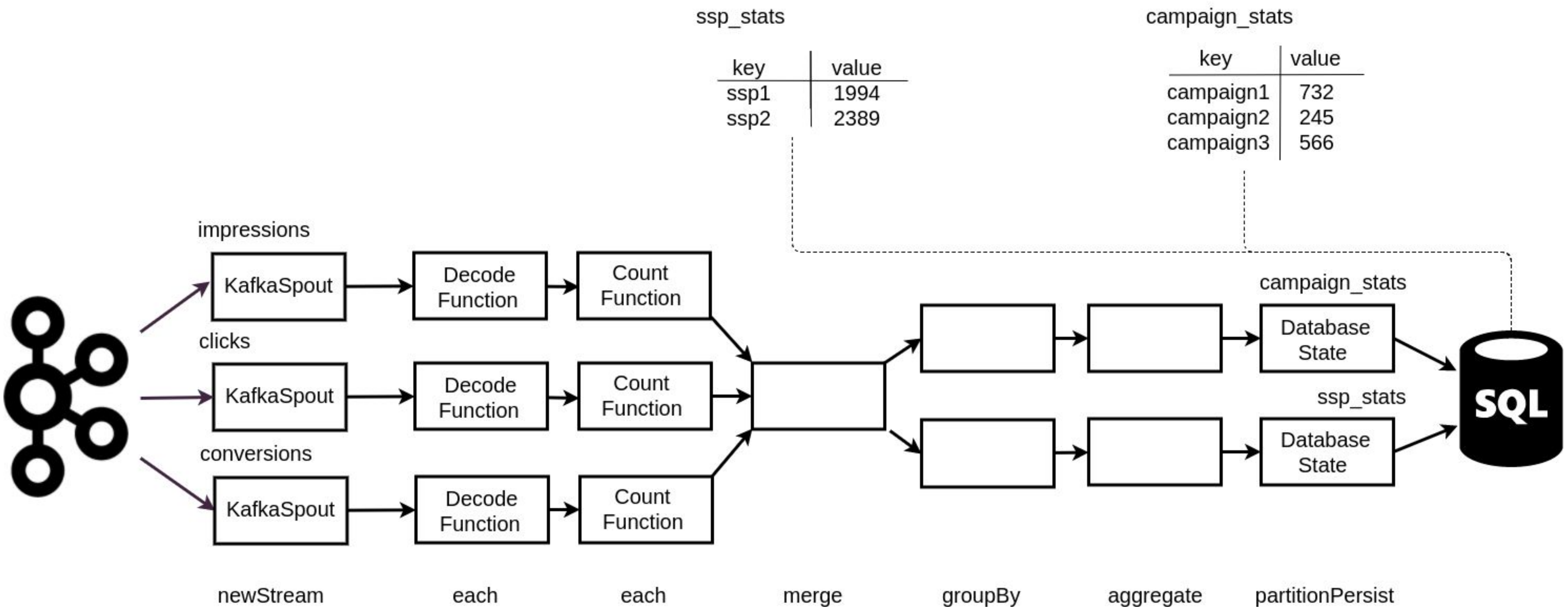
Source: storm.apache.org

Why Trident:

- high-level declarative API
- provides **functions, filters, joins, groupings, and aggregations**
- supports stateful, incremental processing on top of persistence stores
- processes **microbatches** (transactions) and supports exactly-once processing



Use case: stats-counter



Use case: stats-counter (exactly-once state)

zookeeper

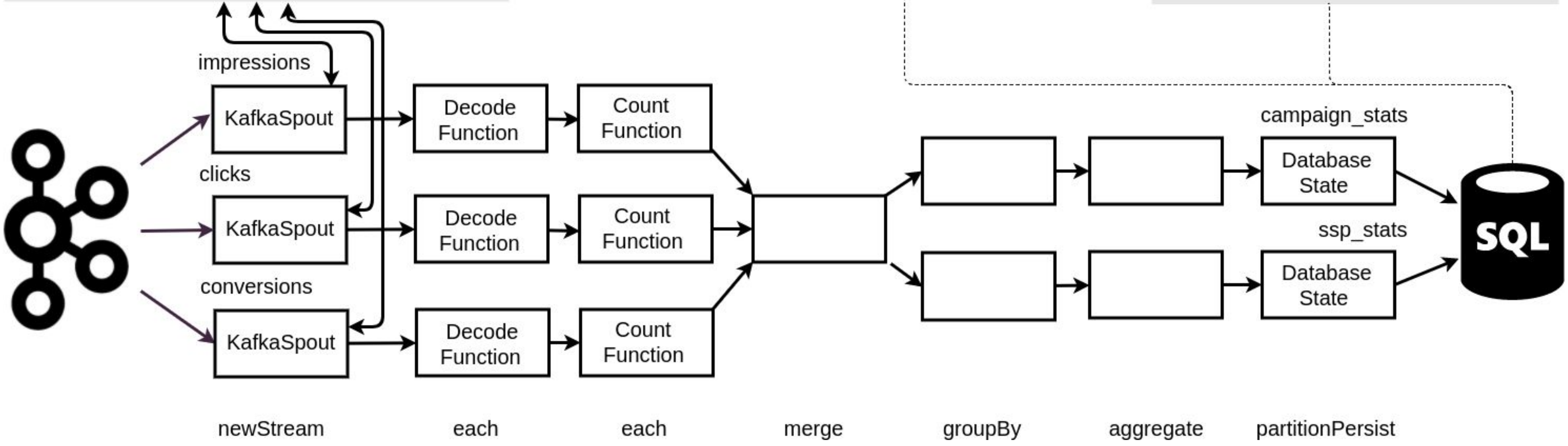
topic	<offset1, offset2>	transaction_id
impressions	<5831, 5917>	19
clicks	<623, 680>	19
conversions	<423, 442>	19

ssp_stats

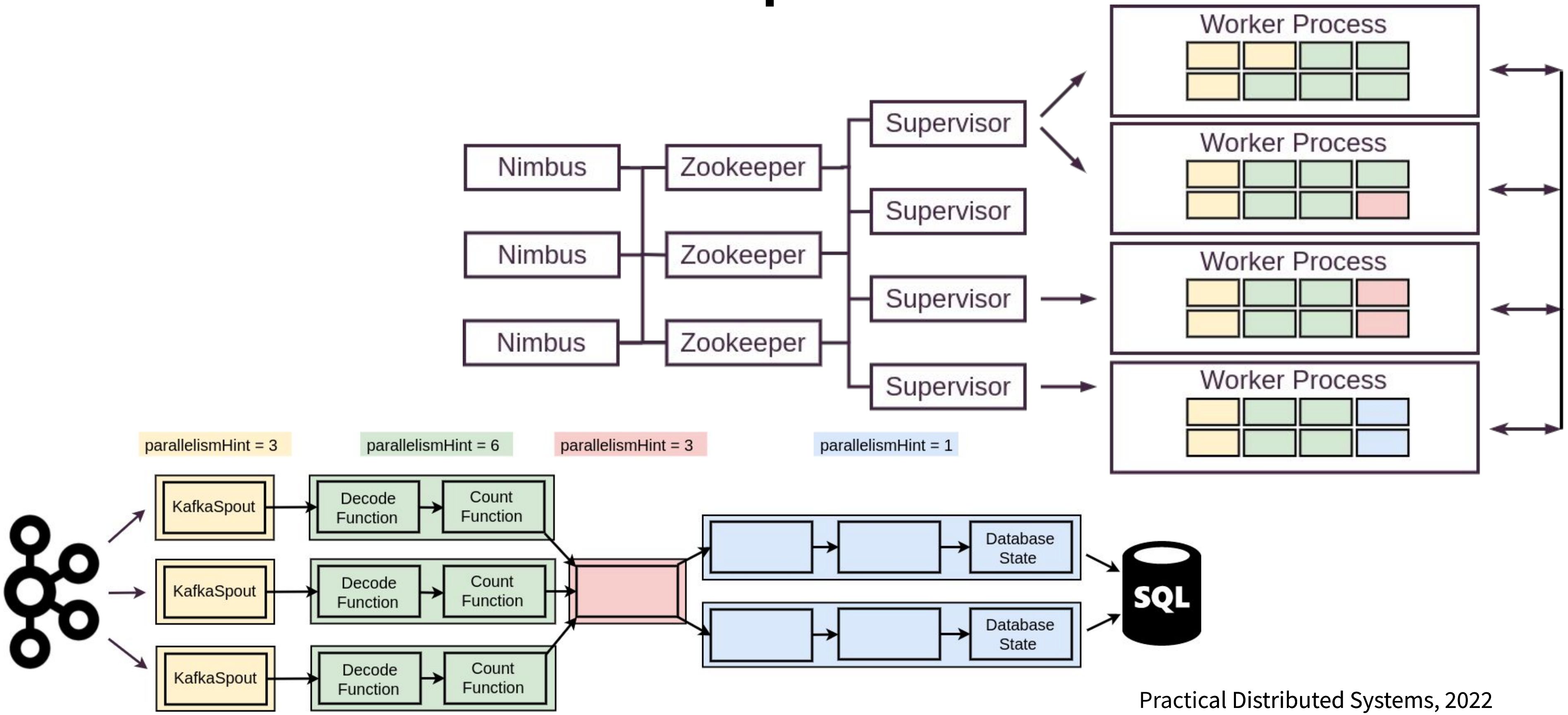
key	value	last_transaction_id
ssp1	1994	17
ssp2	2389	18

campaign_stats

key	value	last_transaction_id
campaign1	732	18
campaign2	245	17
campaign3	566	18



Use case: stats-counter (parallelism)

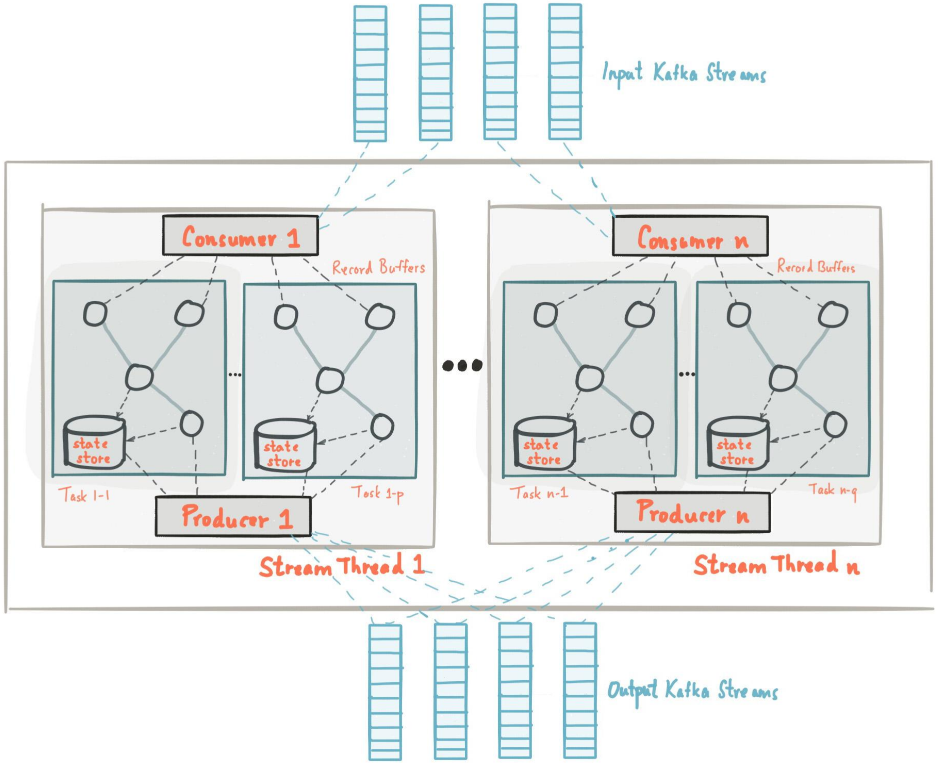


Kafka Streams

Source: kafka.apache.org

Why Kafka Streams:

- Java library (based on Kafka **producer and consumer APIs)** run as a standard application
- **no processing cluster** and no external dependencies
- uses Kafka's **parallelism model** and group membership mechanism (scalability and fault-tolerance)
- does **event-at-a-time** processing (no batching)
- supports **exactly-once** processing



Kafka Streams: topology

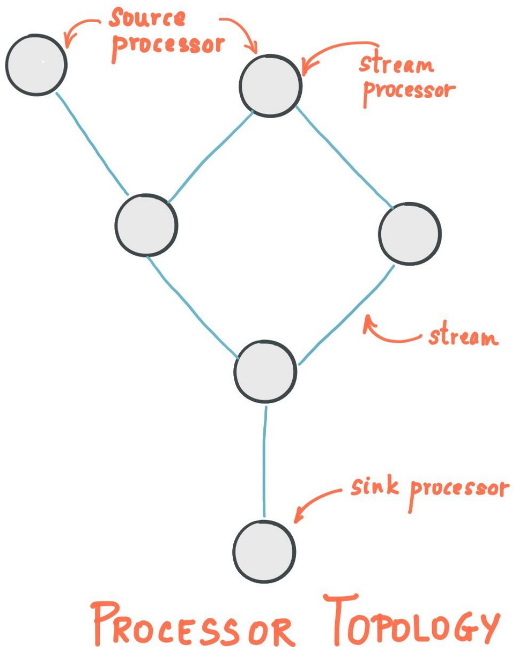
Source: kafka.apache.org

Topology is a graph of **stream processors** that are connected by **streams**:

- consumes records from one or more input Kafka topics (source processors)
- sends records to output Kafka topic (sink processors)

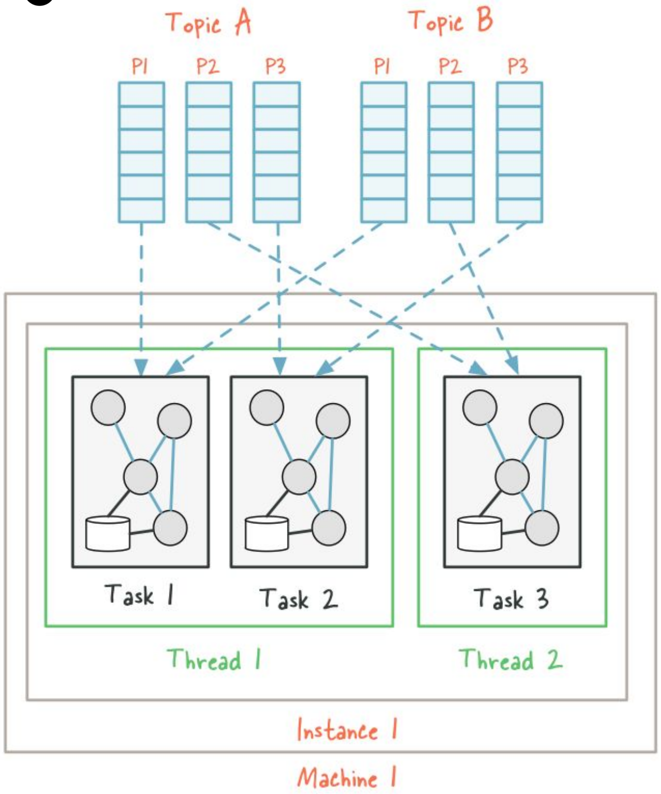
Kafka Streams uses Kafka concepts:

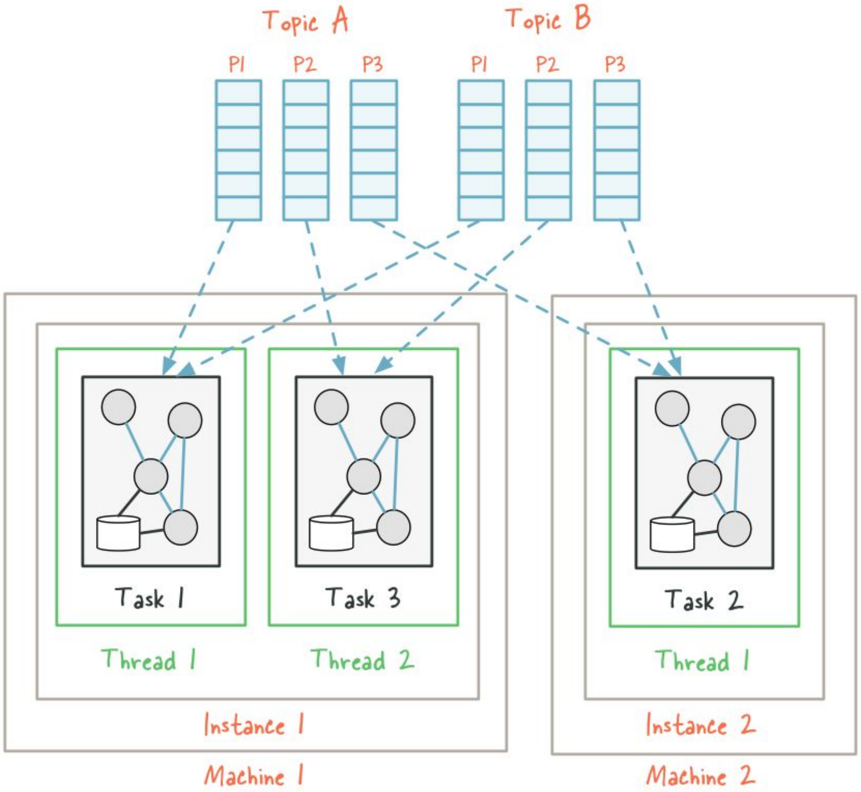
- **data record** - Kafka message
- **stream partition** - Kafka topic partition
- **keys** determine the partitioning



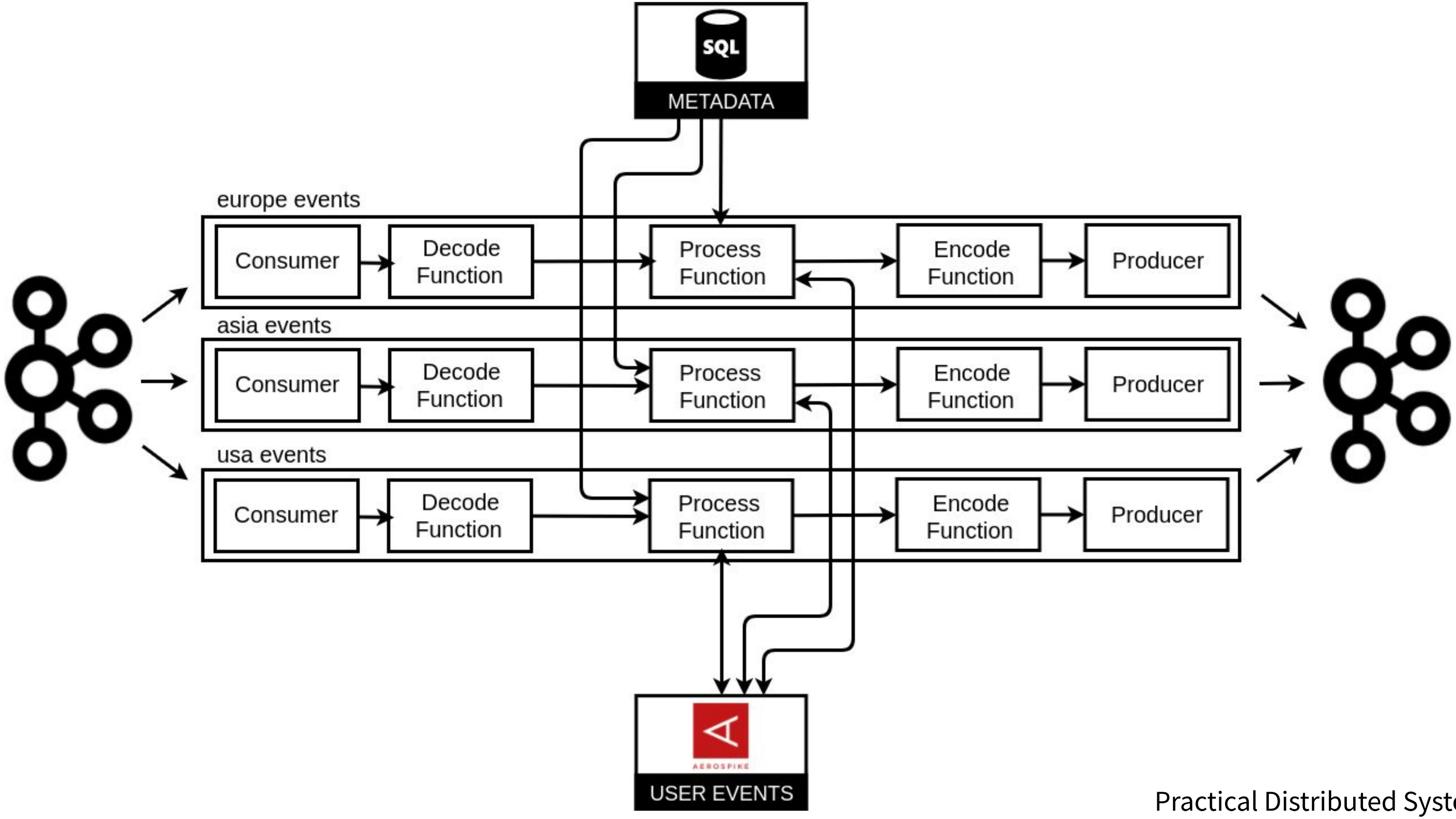
Kafka Streams: threading model

Source: kafka.apache.org

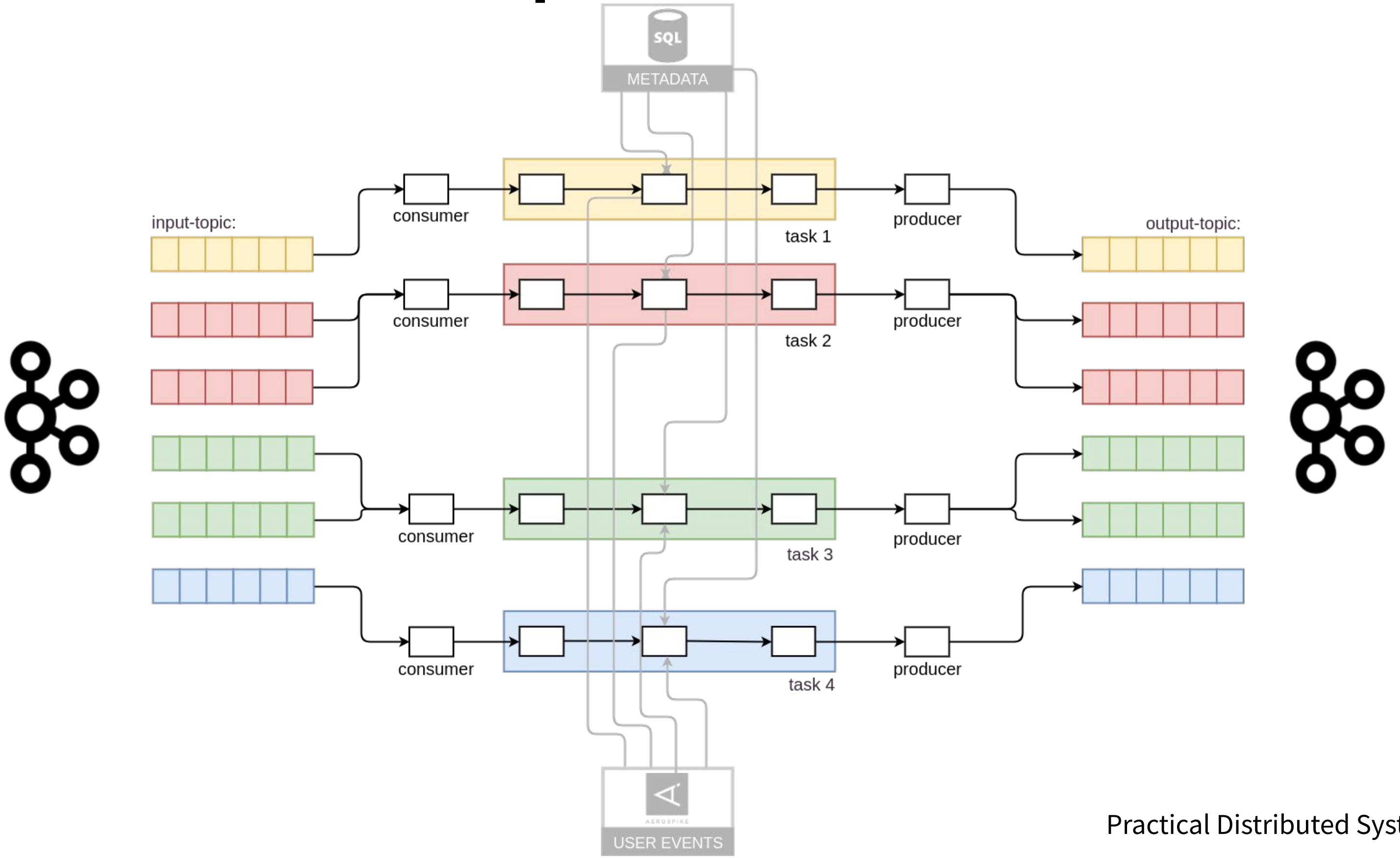




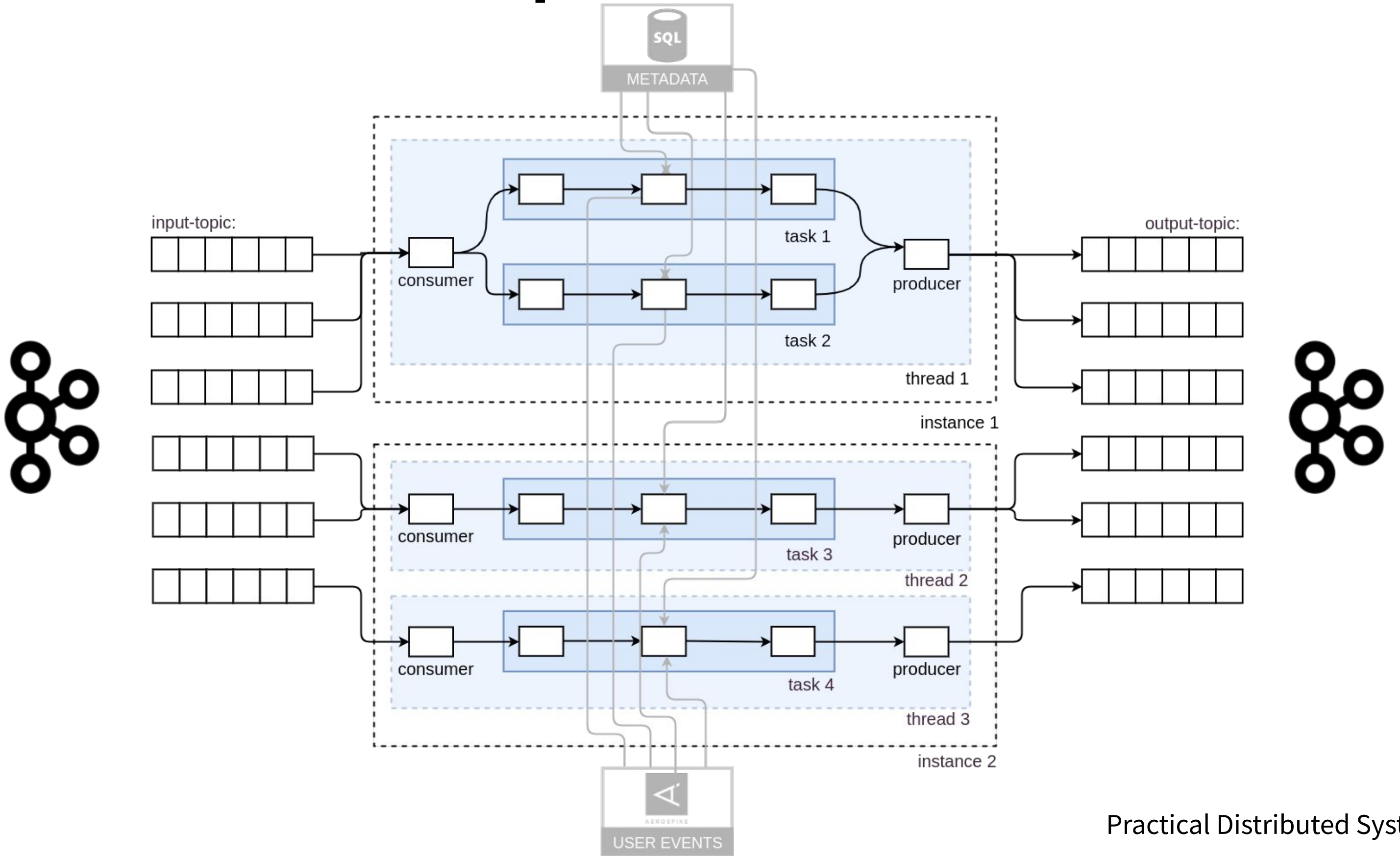
Use case: data-flow



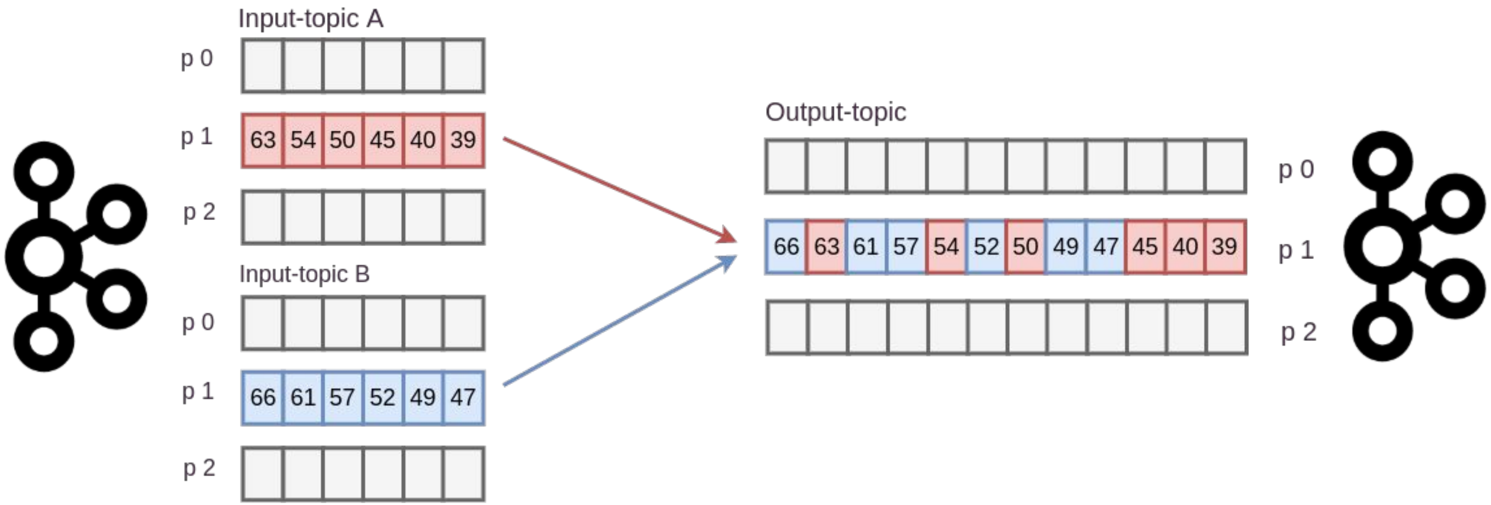
Use case: data-flow (parallelism)

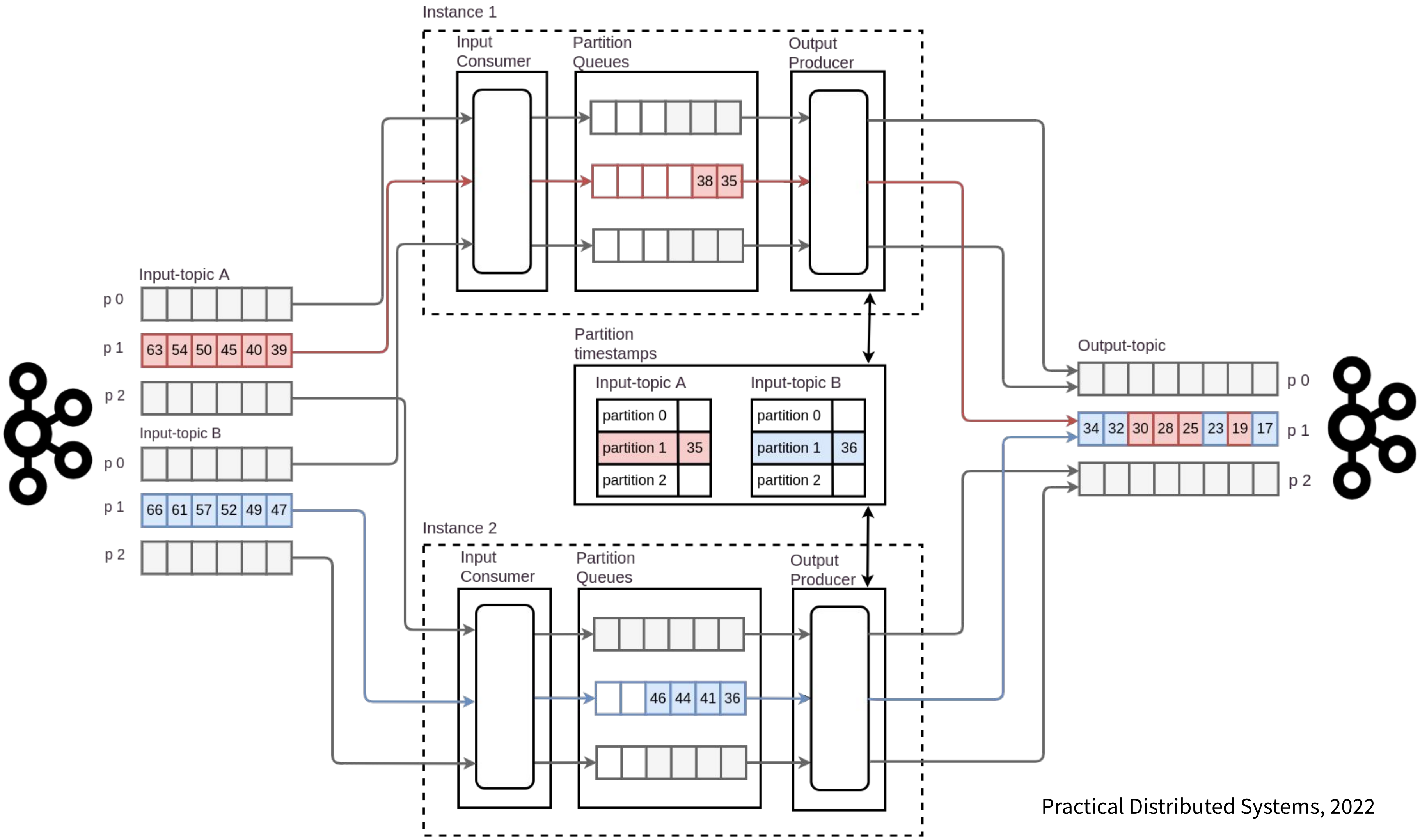


Use case: data-flow (parallelism)



Use case: merger

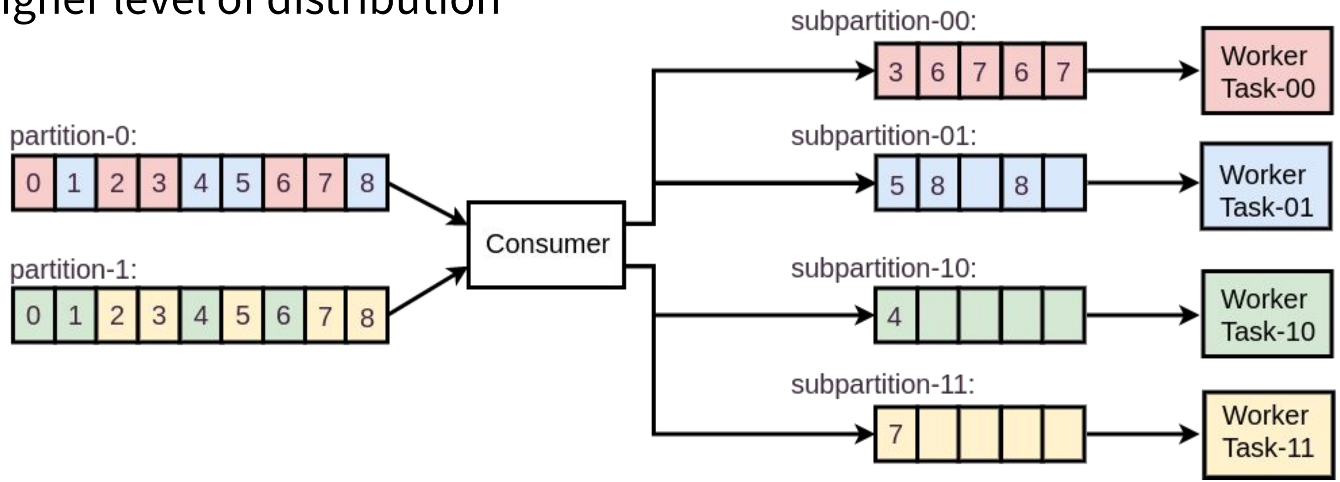



Instance 1
Input Consumer
Partition Queues
Output Producer
38 35
Input-topic A
p 0
p 1
63 54 50 45 40 39
p 2
Input-topic B
p 0
p 1
66 61 57 52 49 47
p 2
Partition timestamps
Input-topic A
partition 0
partition 1 35
partition 2
Input-topic B
partition 0
partition 1 36
partition 2
Output-topic
p 0
p 1
34 32 30 28 25 23 19 17
p 2
Instance 2
Input Consumer
Partition Queues
Output Producer
46 44 41 36

Kafka Workers: main features

Why Kafka Workers (github.com/RTBHOUSE/kafka-workers):

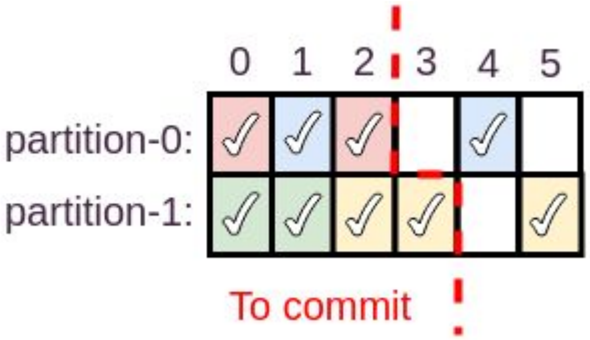
- better threading model with better resources utilization
  - separating processing from consumption
  - higher level of distribution



Kafka Workers: main features

Why Kafka Workers (github.com/RTBHOUSE/kafka-workers):

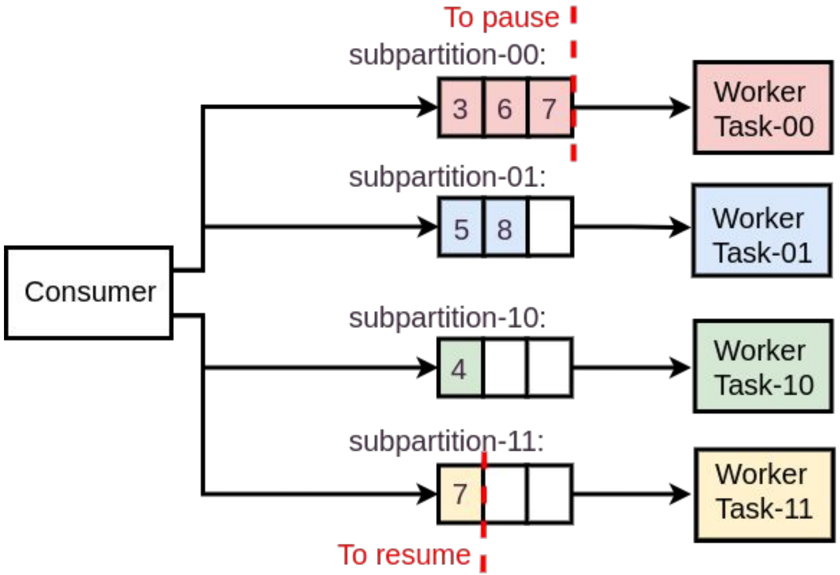
- asynchronous processing
    - processing timeouts
    - tighter control of offset commits



Kafka Workers: main features

Why Kafka Workers (github.com/RTBHOUSE/kafka-workers):

- backpressure



Kafka Workers: main features

Why Kafka Workers (github.com/RTBHOUSE/kafka-workers):

- possibility to pause and resume processing for a given partition
- at-least-once semantics
  - handling failures
- simplicity
  - Kafka Consumer API
  - no processing cluster, no external dependencies
  - without translating messages to/from its internal data format
  - no interprocess communication
- kafka-to-kafka, hdfs, bigquery, elasticsearch connectors

Kafka Workers: API (subpartitions)

```java
public interface WorkerPartitioner<K, V> {

    int subpartition(ConsumerRecord<K, V> consumerRecord);

}
```

Kafka Workers: API (tasks)

```java
public interface WorkerTask<K, V> {

    boolean accept(WorkerRecord<K, V> record);

    void process(WorkerRecord<K, V> record, RecordStatusObserver observer);

}
```

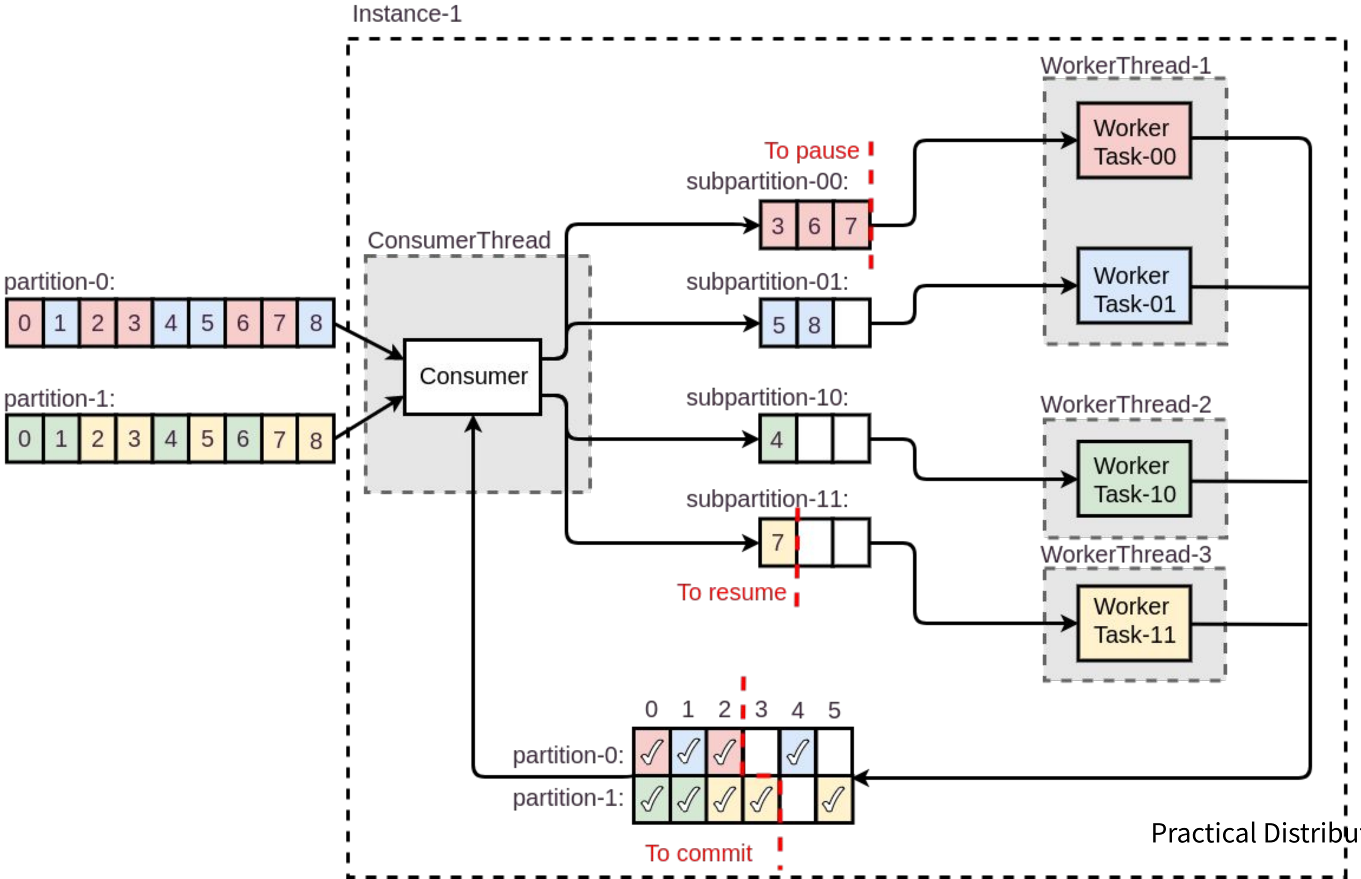
```java
public interface RecordStatusObserver {

    void onSuccess();

    void onFailure(Exception exception);

}
```

Kafka Workers: threading model



Summary

What we have achieved:

- platform monitoring
- much more stable platform
- higher quality of data processing
- HDFS & BigQuery & Elasticsearch streaming
- multi-DC architecture and data synchronization
- high scalability
- better data-flow monitoring, deployment & maintenance