

Kafka in a nutshell

Webstep AB, 16/11/2018

By Vajo Lukic

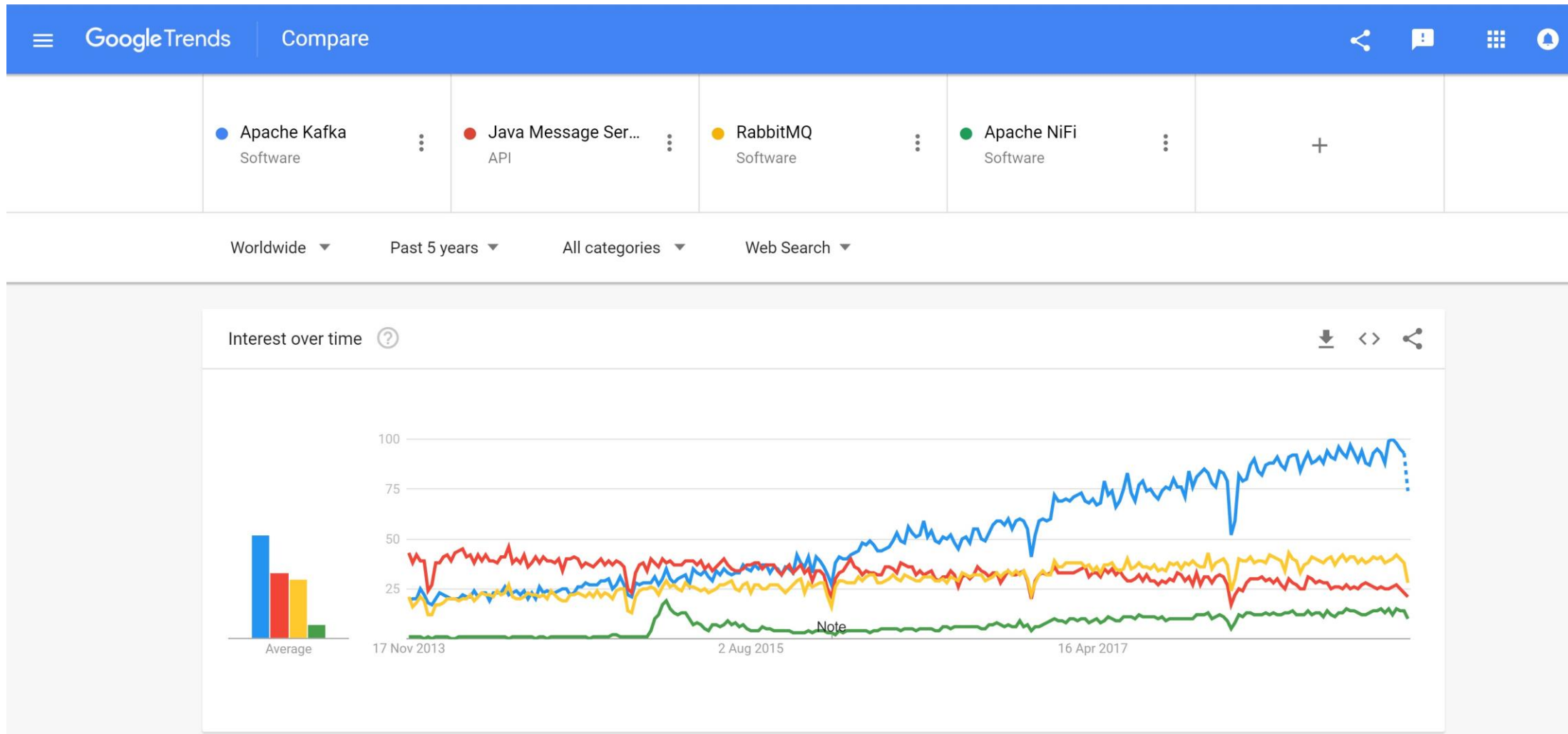
Agenda

- Kafka
- Confluent Schema Registry
- Kafka Connect
- Kafka Streams
- Confluent REST Proxy

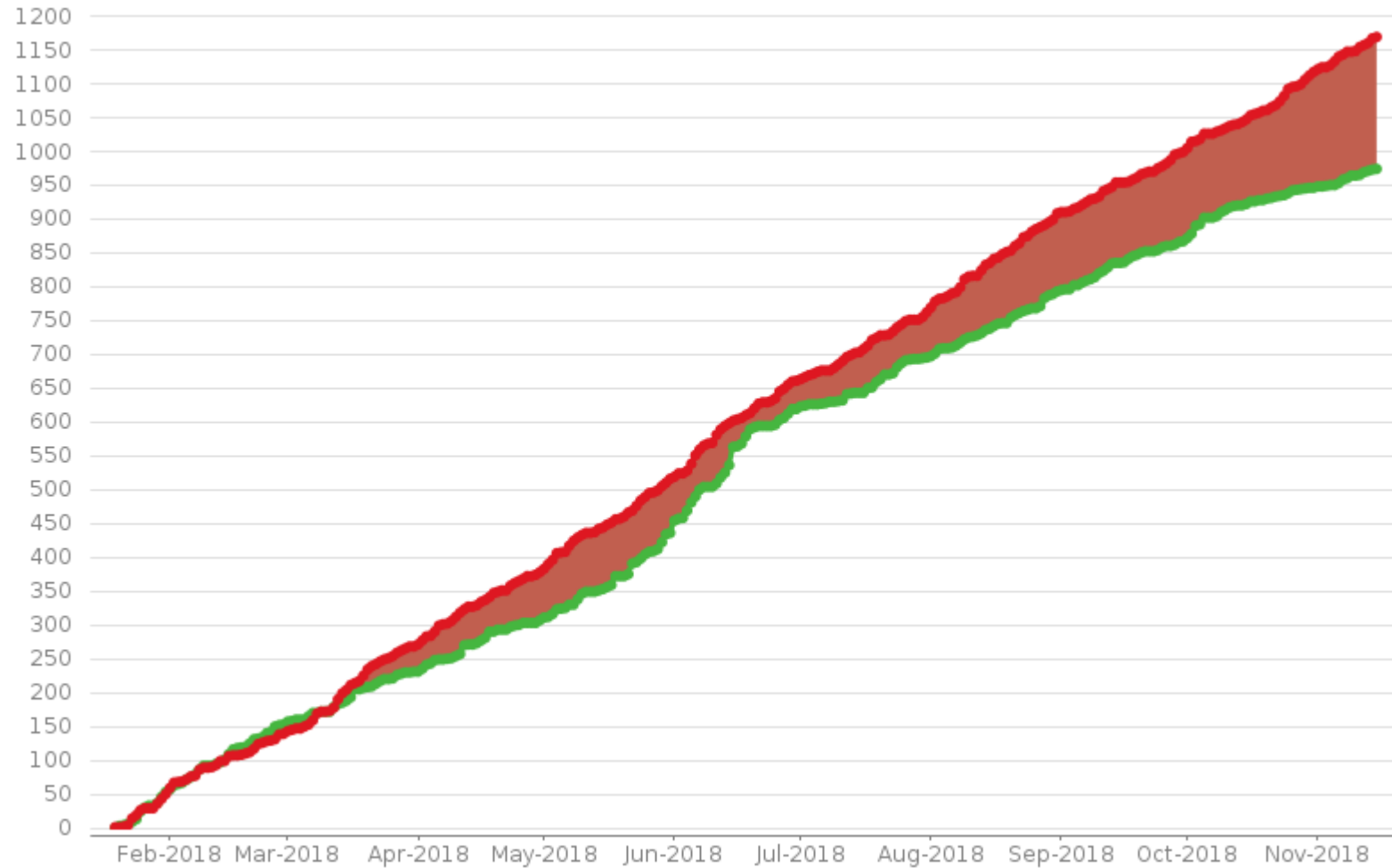
Kafka – a hot topic!



Kafka in Google Trends



Apache Kafka project Open vs. Closed Issues

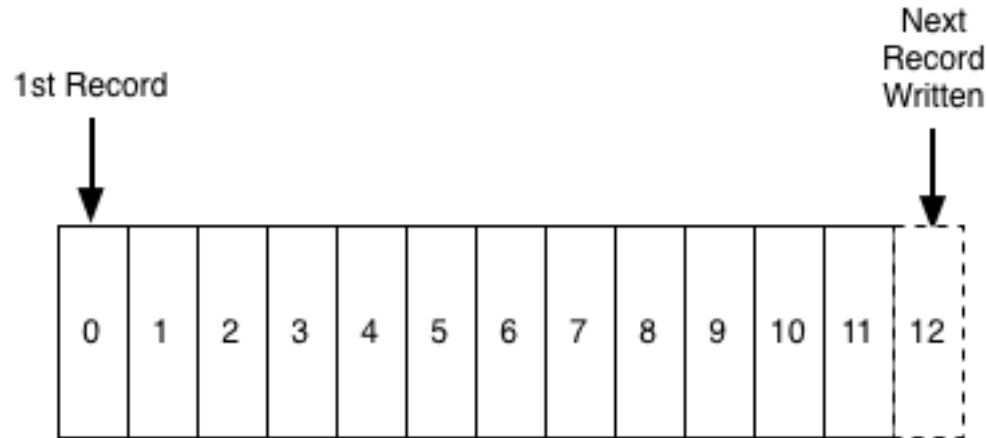


What is Kafka?

- Kafka is a distributed, horizontally-scalable, highly available, fault-tolerant **commit log**
- Kafka is **publish-subscribe** messaging system
- Kafka is **Streaming** Data platform

The Log

A log is perhaps the simplest possible storage abstraction. It is an append-only, totally-ordered sequence of records ordered by time.

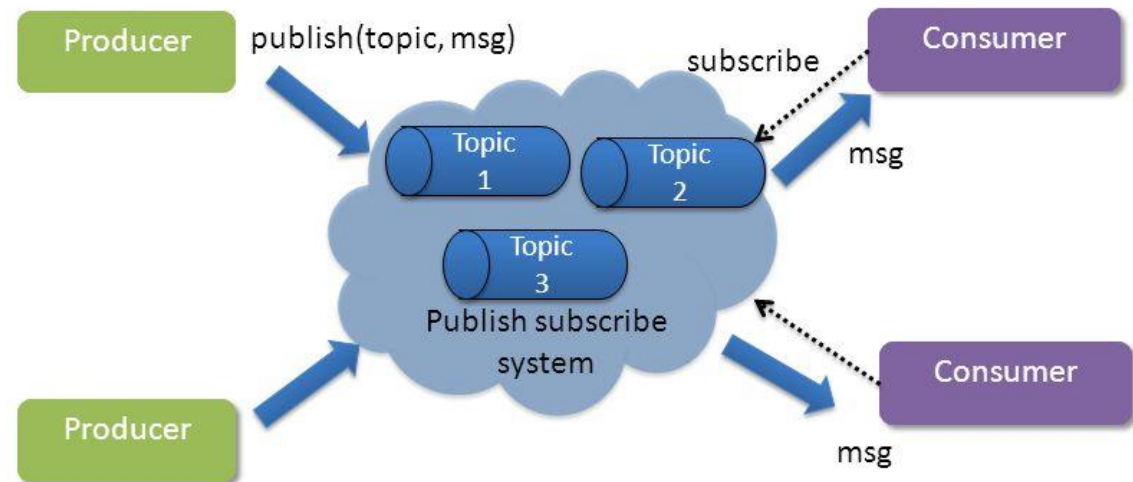


Records are appended to the end of the log, and reads proceed left-to-right. Each entry is assigned a unique sequential log entry number.

Publish-Subscribe pattern

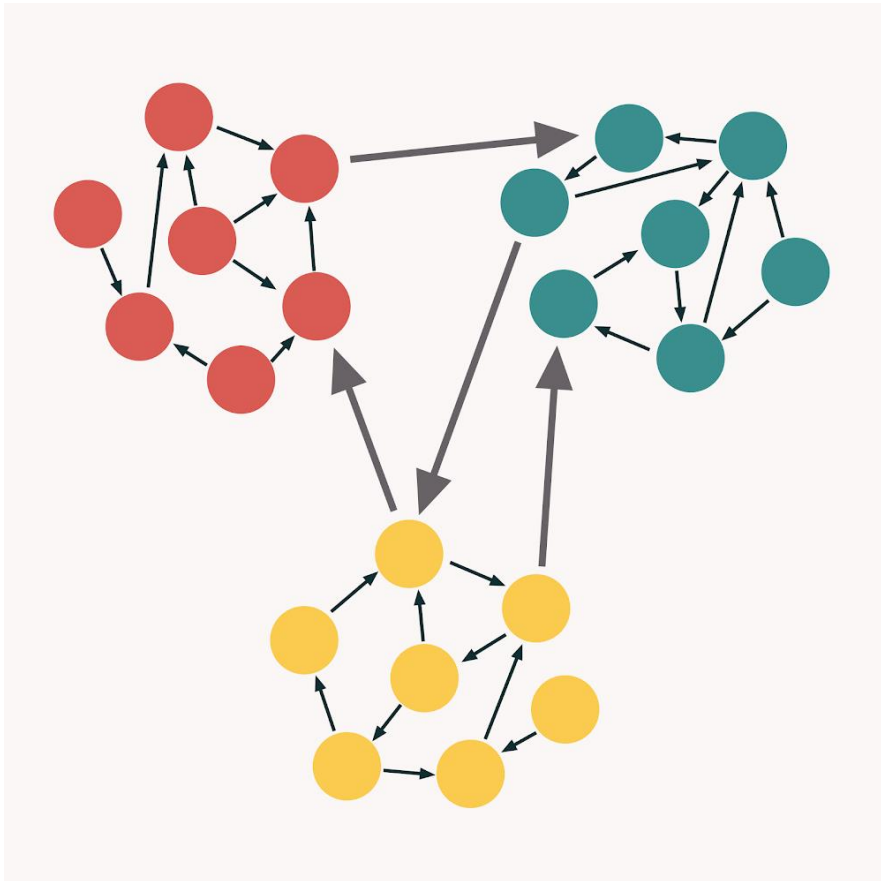
- **Message:** fundamental unit of data. Each message is a key/value pair.
- **Topic:** is just a sharded write-ahead log. A collection of messages that belong to a group
- **Producer:** generates (publishes) messages to the topic
- **Consumer:** consumes (subscribes to) messages in the topic

What is pub sub ?

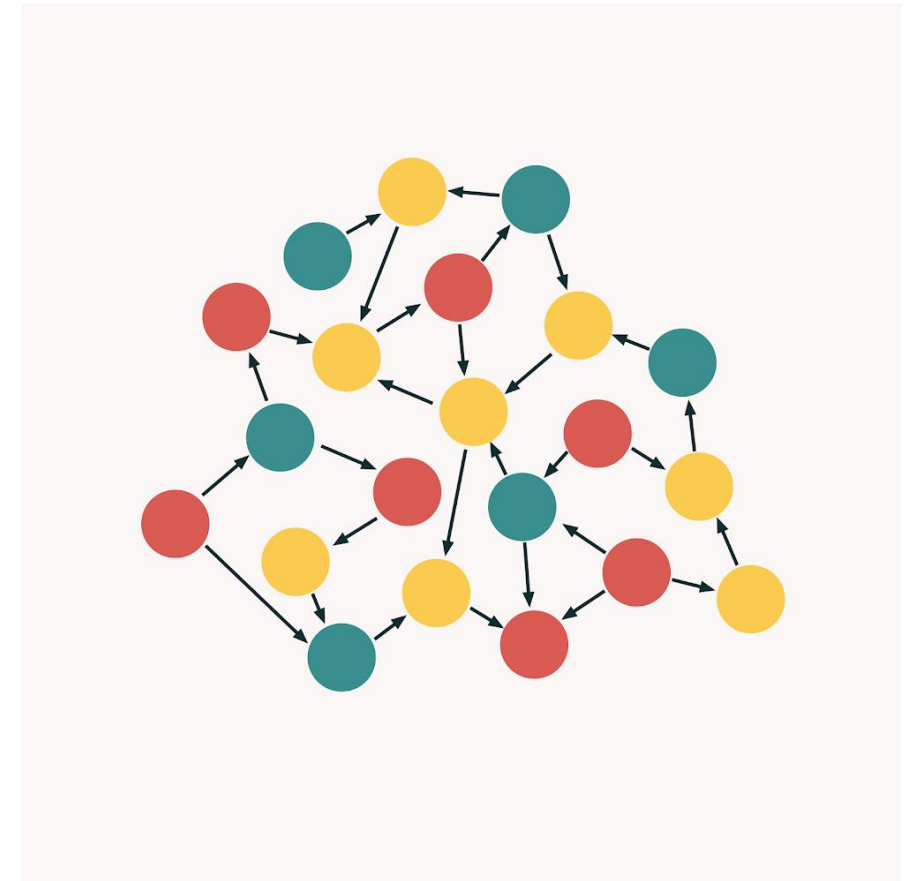


Highly cohesive – Loosely coupled

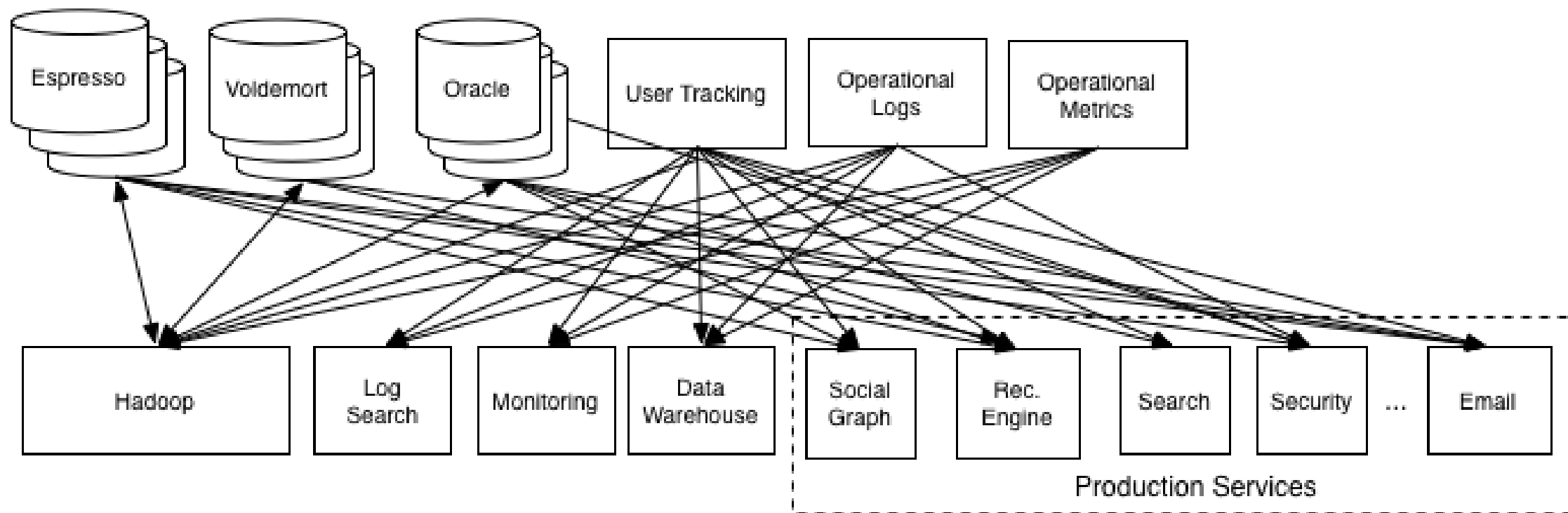
Ideal



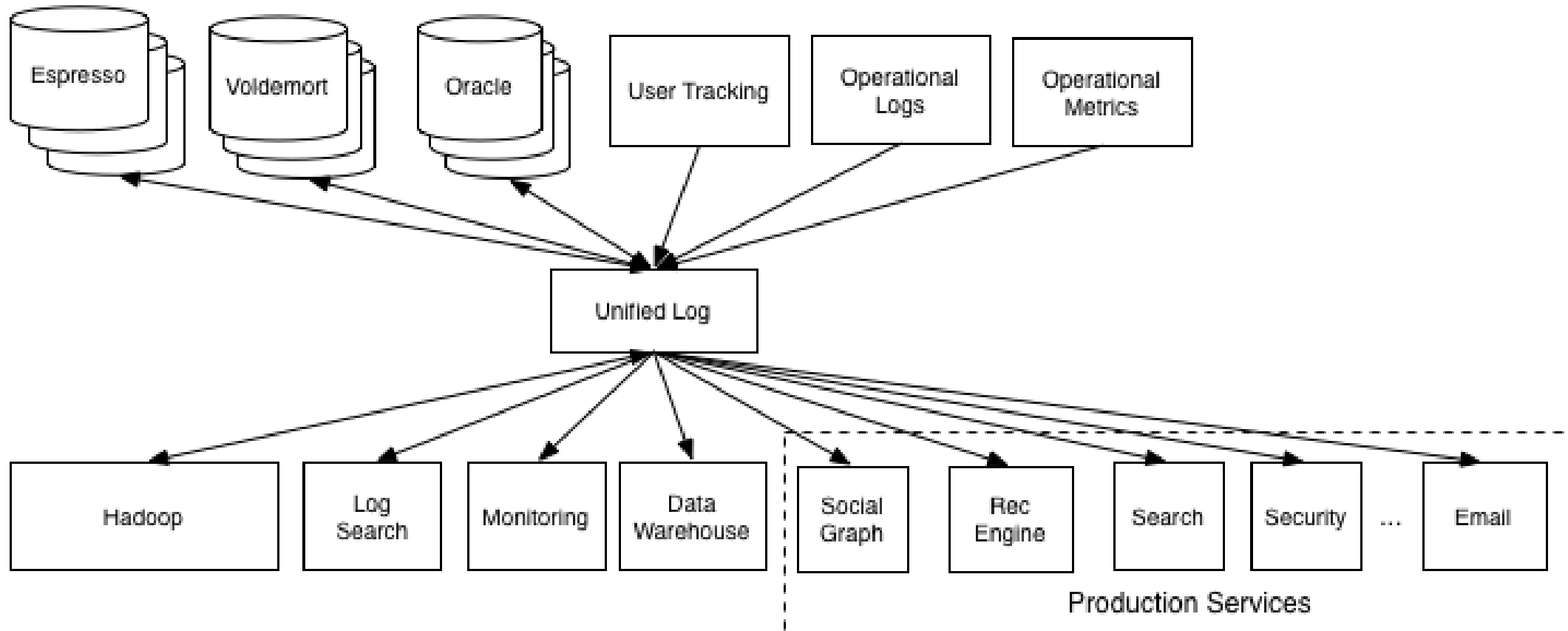
”God object”



Tight coupling

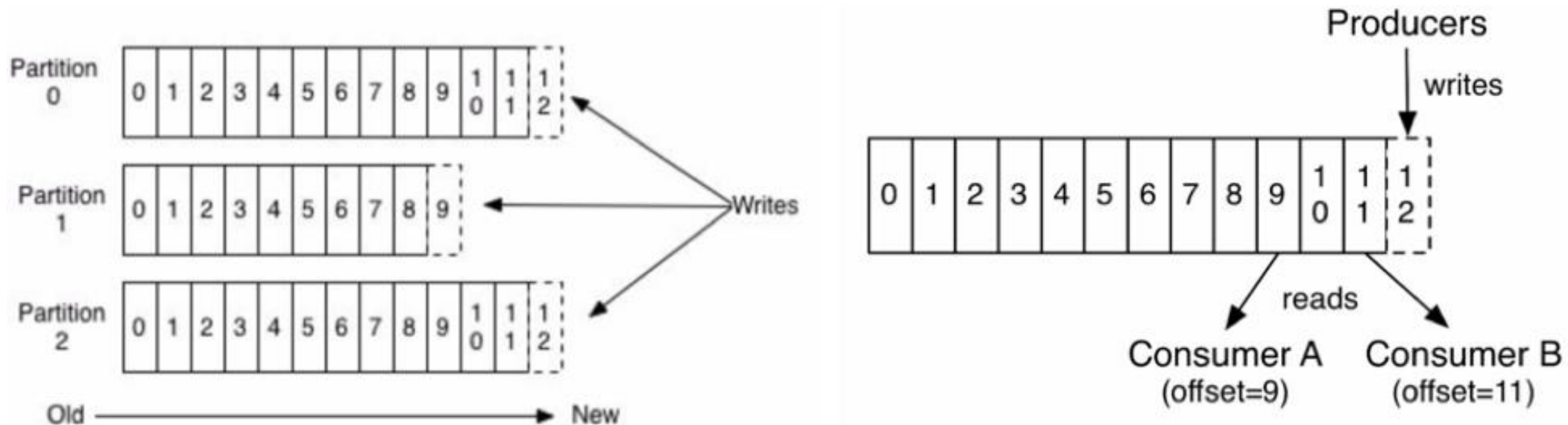


Loose coupling



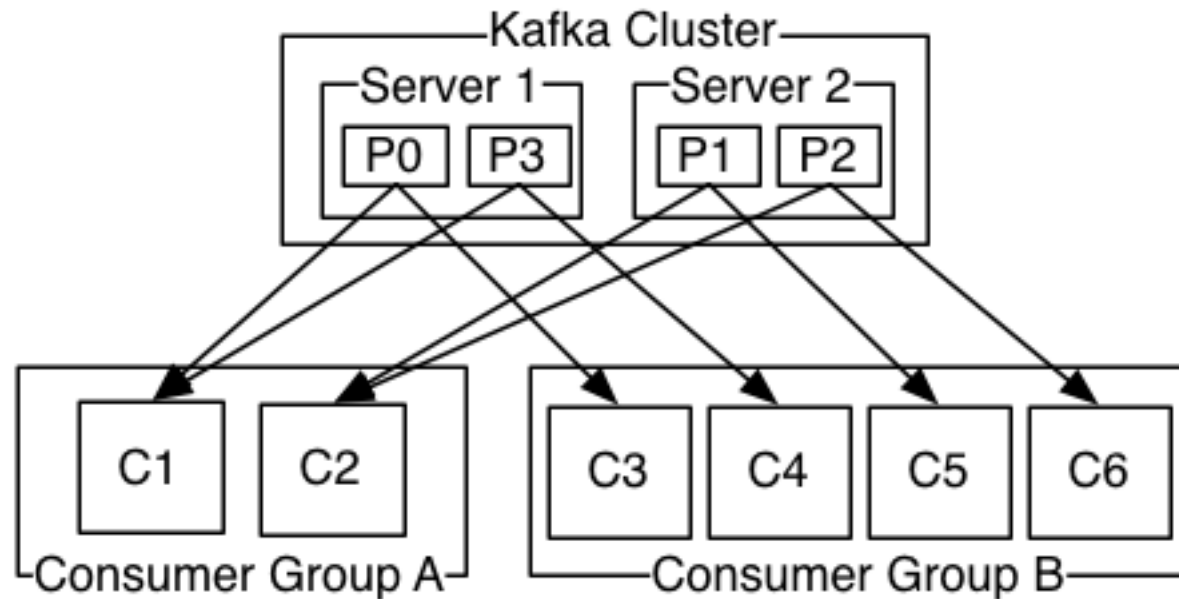
Distributed

Partitions are unique to Apache Kafka that are not seen in the traditional message queuing systems. Each **topic is split** into one or more partitions (logs). Partitions are **redundantly** distributed across the Kafka cluster. Messages are written to one partition but copied to at least two more partitions maintained on different brokers with the cluster.



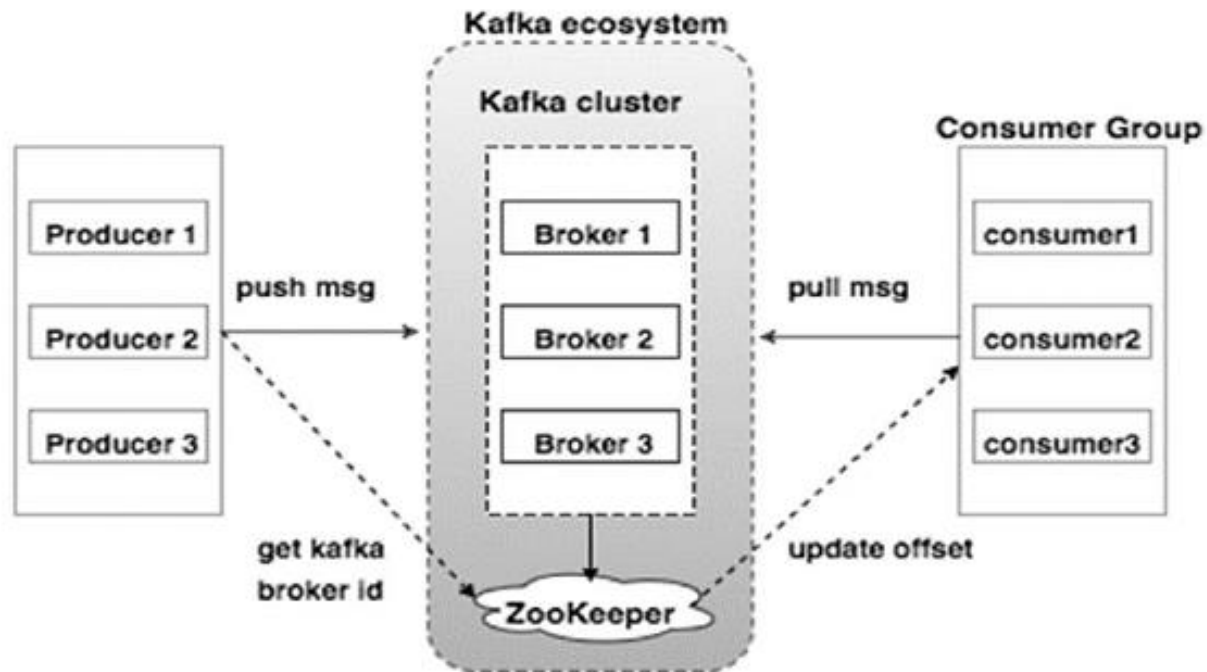
Horizontally Scalable

Consumer groups: consumers belong to at least one consumer group, which is associated with a topic. Each consumer within the group is mapped to one or more partitions of the topic. Kafka will guarantee that a message is only read by a single consumer in the group. Kafka will also ensure that all the messages belonging to the same topic are delivered to all the registered consumers of a group.



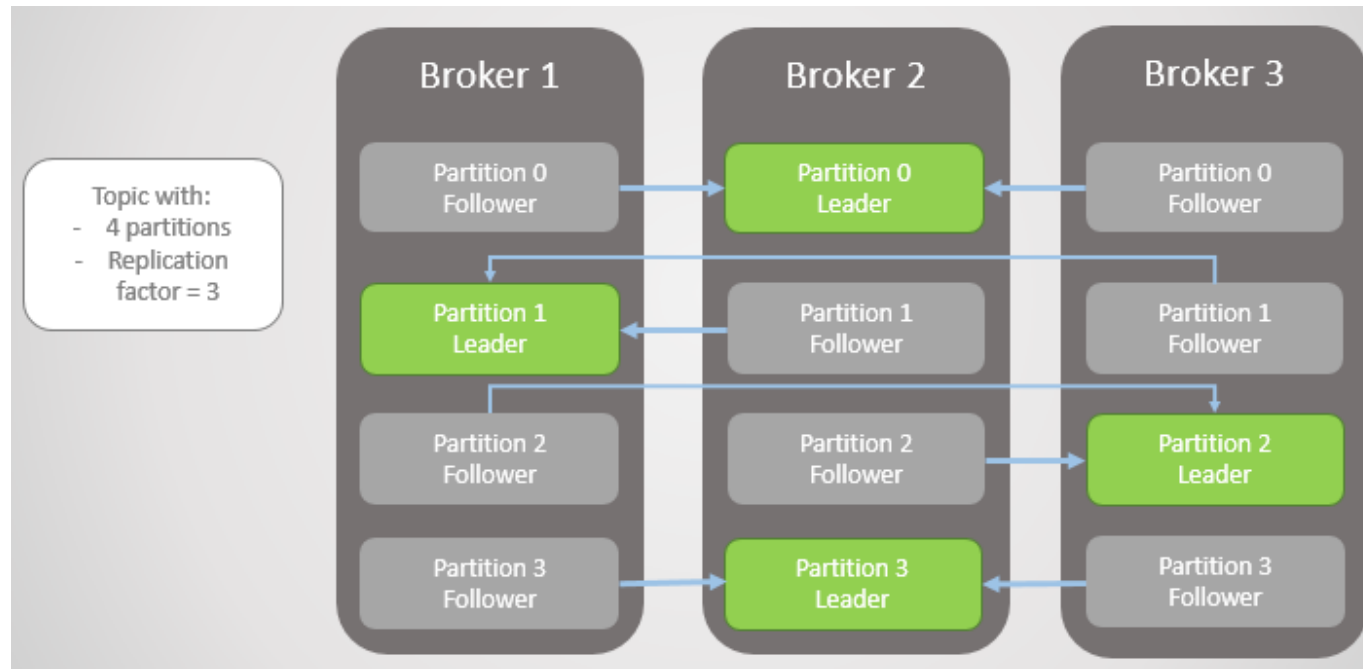
Fault tolerant

Broker: is each Kafka instance belonging to a cluster. It receives messages from producers, assigns offsets, and commits the messages to the disk. Each broker can handle thousands of partitions and millions of messages per second.



Highly Available

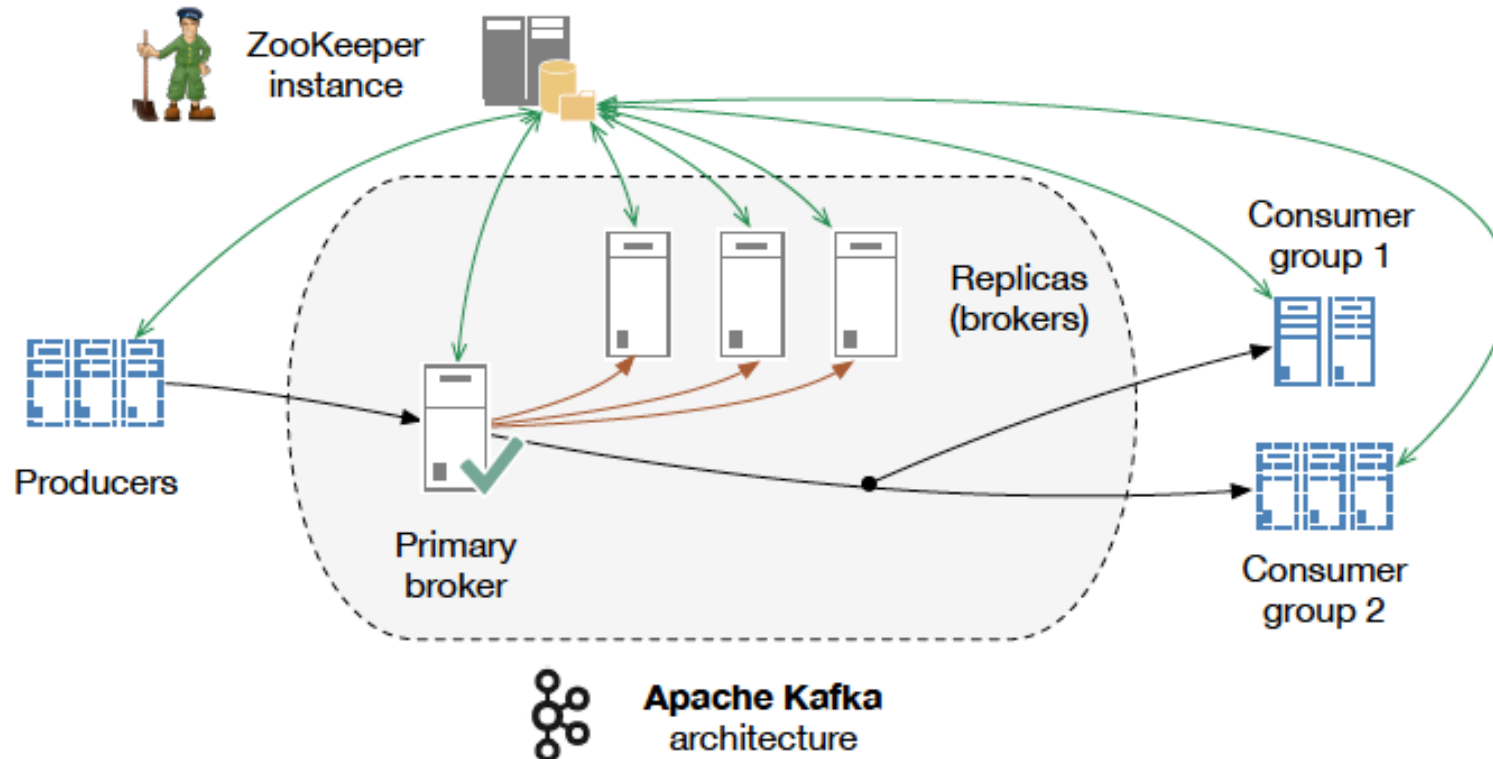
Unit of replication is the partition. Each topic has one or more partitions, each partition has a leader and zero or more followers. You specify the number of partitions and the replication factor for a topic. Factor of three is common - one leader and two followers.



All reads and writes on a partition go to the leader!!!

Distributed and Horizontally Scalable

Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.



Kafka guarantees

Kafka makes the following guarantees about data consistency and availability:

- Messages sent to a topic partition will be appended to the commit log **in the order** they are sent,
- A single consumer instance will see messages **in the order** they appear in the log,
- A message is 'committed' when **all in sync replicas** have applied it to their log,
- Any committed message **will not be lost**, as long as at least one in sync replica is alive.

It's Okay To Store Data In Apache Kafka

The first two rules of message queues in a traditional messaging system are “you do not store messages in the message queue.”

- Because reading the message also **removes** it
- Because messaging systems **scale poorly** as data accumulates beyond what fits in memory.
- Because messaging systems generally **lack robust replication** features (so if the broker dies, your data may well be gone too).

In Kafka:

- Data is appended to the log and persisted on the disk
- Logs are replicated
- Data retention can be defined **per topic** and **by time** or **by size**

Log Compaction

- Log compaction retains **at least the last known value** for each record key for a single topic partition.
- Compacted logs are useful for **restoring state** after a crash or system failure.
- Kafka log compaction allows downstream consumers to restore their state from a log compacted topic.
- Kafka log compaction also allows for deletes. A message with a key and a null payload acts like a tombstone, a delete marker for that key.


Log Compaction

Before
Compaction

Offset	13	17	19	20	21	22	23	24	25	26	27	28
Keys	K1	K5	K2	K7	K8	K4	K1	K1	K1	K9	K8	K2
Values	V5	V2	V7	V1	V4	V6	V1	V2	V9	V6	V22	V25

Cleaning

Only keeps latest version
of key. Older duplicates not
needed.



Offset	17	20	22	25	26	27	28
Keys	K5	K7	K4	K1	K9	K8	K2
Values	V2	V1	V6	V9	V6	V22	V25

After
Compaction

Some benchmarks

This is from **April 2014** ([Source link](#)):

Setup (3 machines): Intel Xeon 2.5 GHz processor with six cores, Six 7200 RPM SATA drives, 32GB of RAM, 1Gb Ethernet

- Single producer thread, no replication: 821,557 records/sec, (78.3 MB/sec)
- Single producer thread, 3x asynchronous replication: 786,980 records/sec, (75.1 MB/sec)
- Single producer thread, 3x synchronous replication: 421,823 records/sec, (40.2 MB/sec)
- Three producers, 3x async replication: **2,024,032** records/sec, (193.0 MB/sec)

- Single Consumer: 940,521 records/sec, (89.7 MB/sec)
- Three Consumers: **2,615,968** records/sec, (249.5 MB/sec)

- End-to-end Latency: 2 ms (median), **3 ms** (99th percentile), 14 ms (99.9th percentile)

How is it so fast?

1. Partitioning the topic (scales horizontally)
2. Consumer groups (scales horizontally)
3. Optimizing throughput by batching reads and writes
4. Using log: data is only appended to the system, and reads are simple. All such operations are $O(1)$
5. Zero Copy ([link](#)) - calls the OS kernel directly rather than at the application layer to move data fast
6. Uses standardized binary data format for producers, brokers and consumers (so data can be passed without modification) - [link](#)

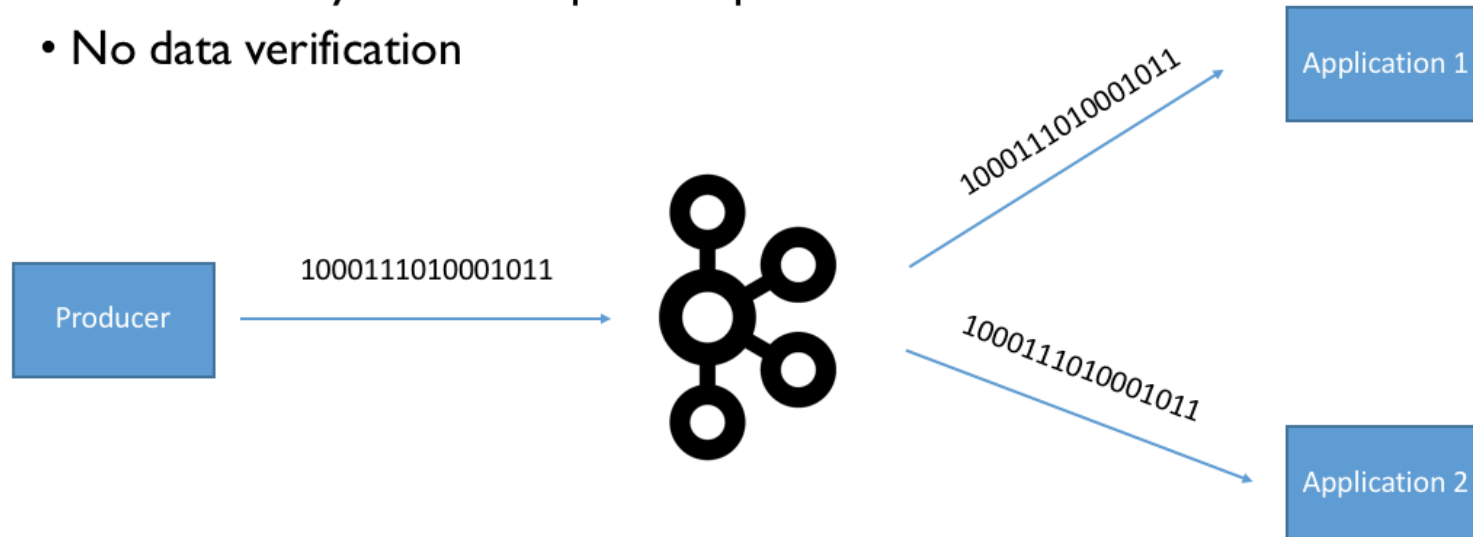
Confluent Schema Registry

- Schema Registry provides a serving layer for your **metadata**.
- It provides a **RESTful interface** for storing and retrieving Avro schemas.
- It stores a **versioned history** of all schemas, provides multiple compatibility settings and allows evolution of schemas according to the configured compatibility settings and expanded Avro support.
- It provides **serializers** that plug into Kafka clients that handle schema storage and retrieval for Kafka messages that are sent in the Avro format.

Consumer data parsing errors

- The field you're looking for doesn't exist anymore
- The type of the field has changed (e.g. what used to be a String is now an Integer)

- Kafka takes bytes as an input and publishes them
- No data verification



Options to prevent these issues

1. Catch exception on parsing errors. Your code becomes ugly and very hard to maintain.
2. Never ever change the data producer and triple check your producer code will never forget to send a field. That's what most companies do. But after a few key people quit, all your "safeguards" are gone.
3. Adopt a data format and enforce rules that allow you to perform schema evolution while guaranteeing not to break your downstream applications. (Sounds too good to be true ?)

Avro to the rescue!



CSV

Probably the worst data format for streaming, all-time favorite of everyone who doesn't deal with data on a daily basis

Pros:

- Easy to parse... with Excel,
- Easy to read... with Excel,
- Easy to make sense of... with Excel

Cons:

- The data types of elements have to be inferred and are not a guarantee
- Parsing becomes tricky when data contains the delimiting character
- Column names (header) may or may not be present in your data

XML — The Dinosaur

- XML is heavyweight, CPU intense to parse and completely outdated, so don't use it for data streaming.
- It has schemas support, but unless you take pleasure in dealing with XSD files, XML is not worth considering.
- Additionally you would have to send the XML schema with each payload, which is very wasteful of resources.

The relational database format???

- Pros: Data is fully typed, Data fits in a table format
- Cons: Data has to be flat, data is stored in a database, and data definition, storage, and serialization will be different for each database technology.
- No schema evolution protection mechanism. Evolving a table can break applications

JSON — Everyone's favorite

Pros:

- Data can take any form (arrays, nested elements)
- JSON is a widely accepted format on the web
- JSON can be read by pretty much any language
- JSON can be easily shared over a network

Cons:

- JSON has no native schema support (JSON schema is not a spec of JSON)
- JSON objects can be quite big in size because of repeated keys
- No comments, metadata, documentation

Apache Avro

- Apache Avro is binary data serialization system.
- Avro provides data structures, binary data format, container file format to store persistent data, and provides RPC capabilities.
- Avro does not require code generation
- It integrates well with JavaScript, Python, Ruby, C, C#, C++ and Java.
- Avro gets used in the Hadoop ecosystem as well as by Kafka.
- Avro is similar to Thrift, Protocol Buffers, JSON, etc.
- Avro needs less encoding as part of the data since it stores names and types in the schema reducing duplication.
- Avro supports the evolution of schemas.

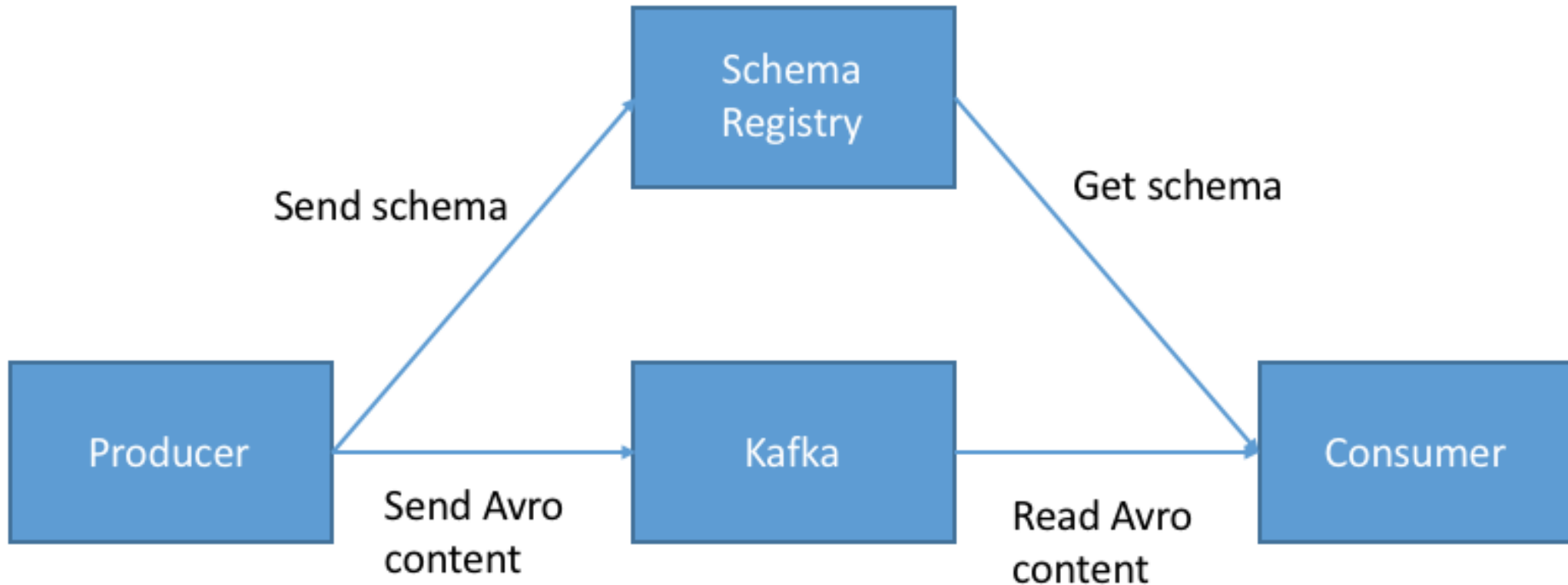
Apache Avro

- **Avro schemas** are defined using JSON. Because every developer knows or can easily learn JSON, there's a very low barrier to entry.
- **An Avro object** contains the **schema and the data**. The data without the schema is an invalid Avro object. That's a big difference with say, CSV, or JSON.
- You can make your schemas **evolve over time**. Apache Avro has a concept of projection which makes evolving schema seamless to the end user.

Avro example - Java

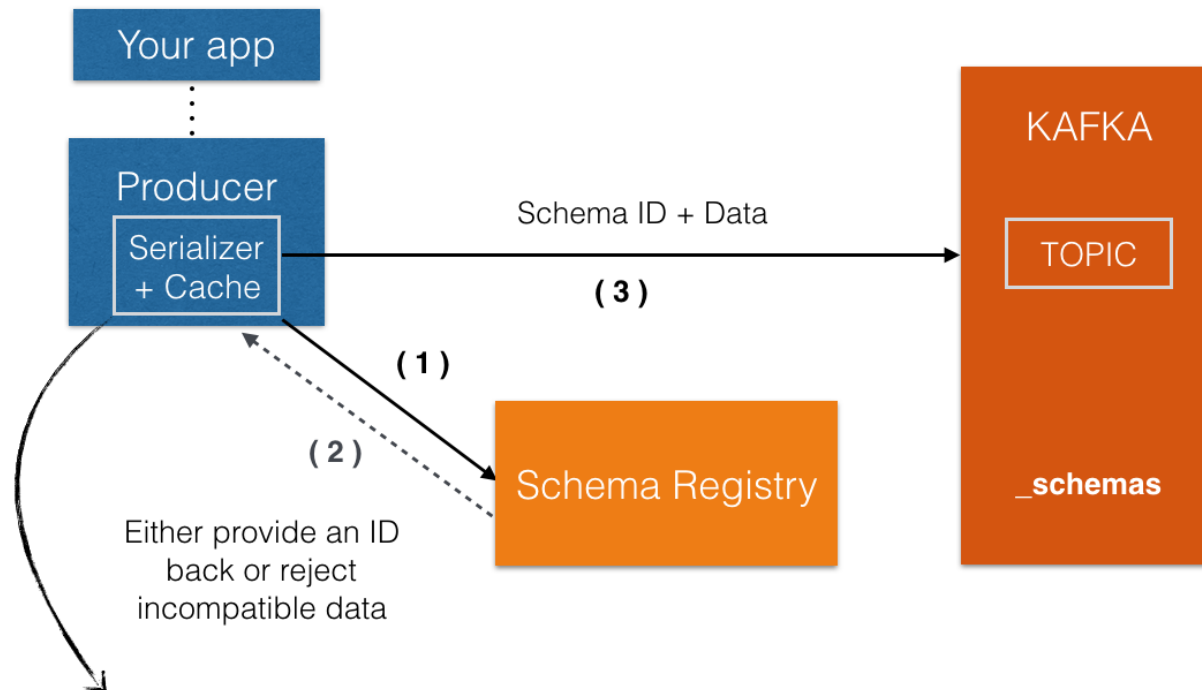
```
{ "namespace": "example.avro",  
  "type": "record",  
  "name": "User",  
  "fields": [  
    { "name": "name", "type": "string" },  
    { "name": "favorite_number", "type": ["int", "null"] },  
    { "name": "favorite_color", "type": ["string", "null"] }  
  ]  
}
```


Schema Registry Architecture



Schema Registry Example

Writing to a topic with Schema Registry

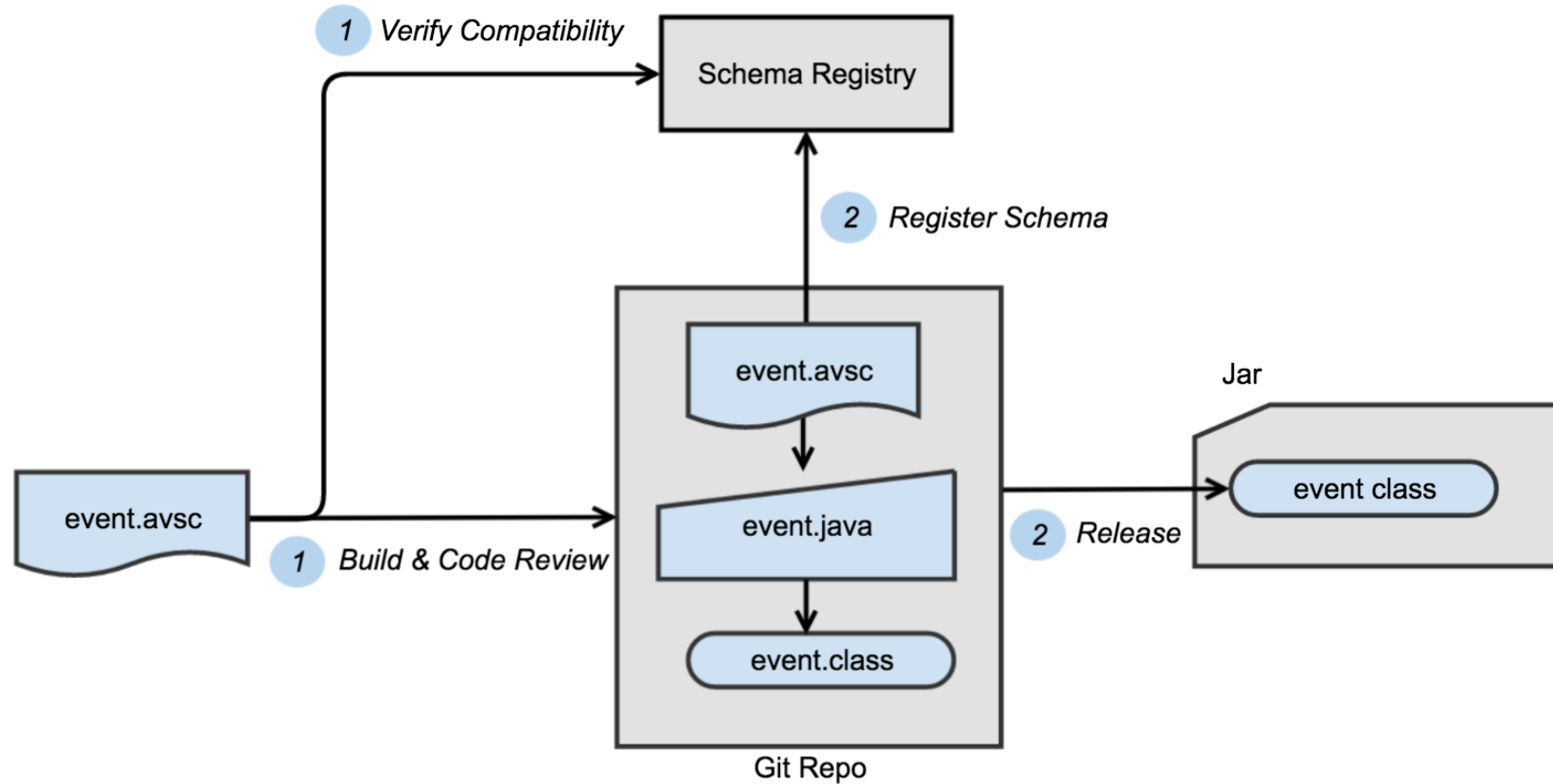


```
producerProps.put("key.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer");  
producerProps.put("value.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer");
```

Schema Registry Example

- Developers create or edit .avsc files locally, and Gradle/Maven will check the schema compatibility against Schema Registry using its REST interface
- If the schema is compatible, generate Java classes of the corresponding event schema with avro-tools.
- Next, the developer will be able to create a pull request for the new schema, and once the code change is approved and merged, our build tool will use Gradle/Maven to conduct one more compatibility check against Schema Registry, and then actually update the schema in Schema Registry, and release a respective Jar to our local Maven repository.

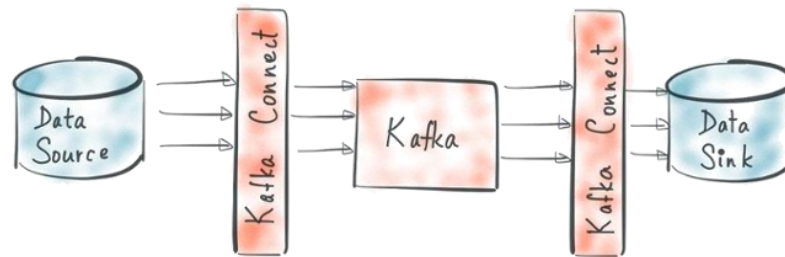
Schema Registry Example



Kafka Connect

Kafka Connect use cases:

- Log and metric collection, processing, and aggregation
- ETL for data warehousing
- Data pipelines management



Kafka Connect

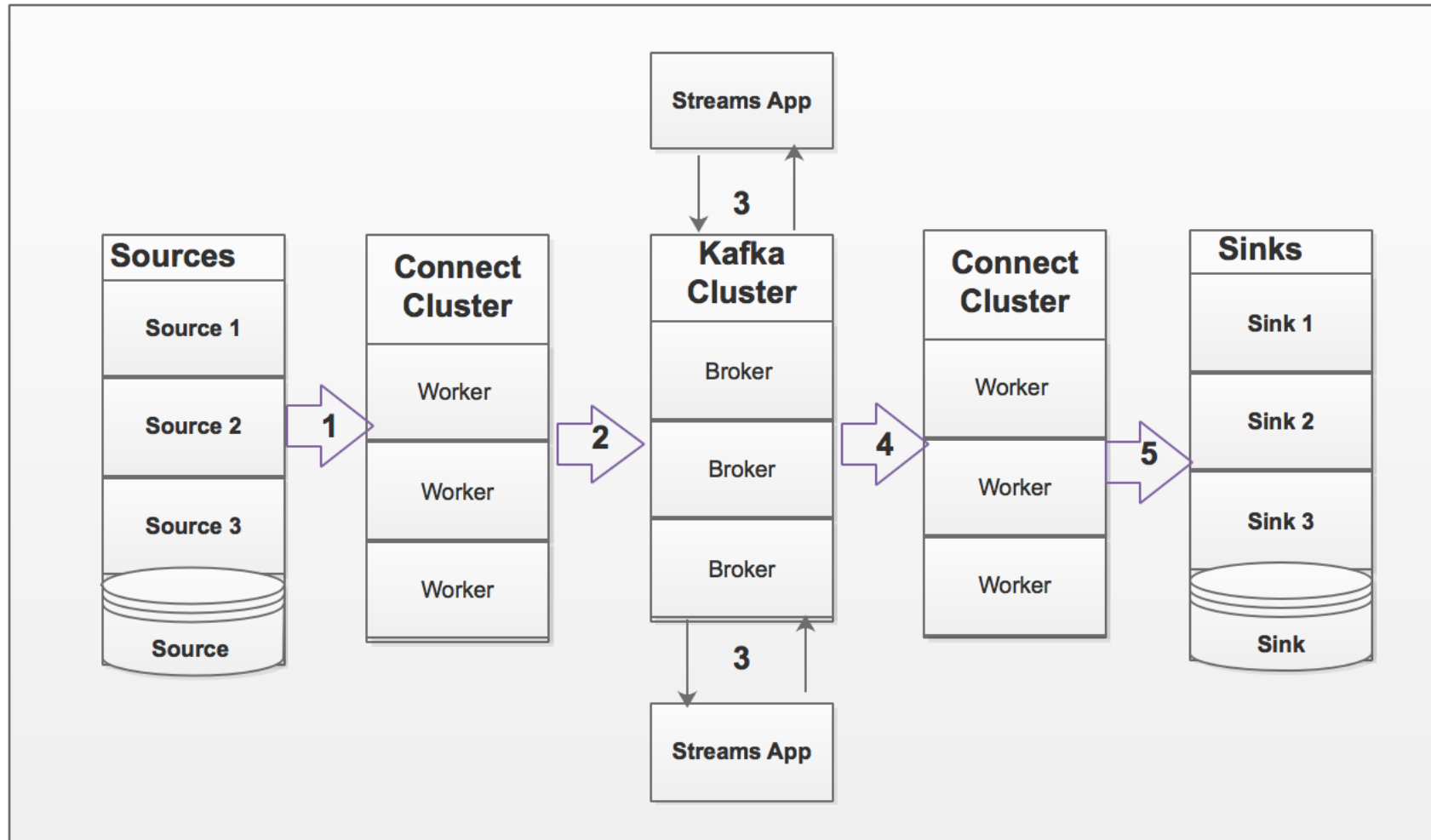
- **Source Connector:** imports data from any external systems (called Source) like mysql,hdfs,etc to Kafka broker cluster
- **Sink Connector:** exports data from Kafka cluster to any external system (called Sink) like hdfs,s3, Oracle etc.

Kafka Connect has its own dedicated cluster where on each machine daemons are running (java processes called **Workers**).

Each worker instance is stateless and does not share info with other workers. Instead, they coordinate with each other, belonging to same group id, via the internal kafka topics.

Key properties: Broad copying by default, Streaming and batch, Scales to the application, Focus on copying data only, Parallelism, Accessible connector API

Kafka Connect



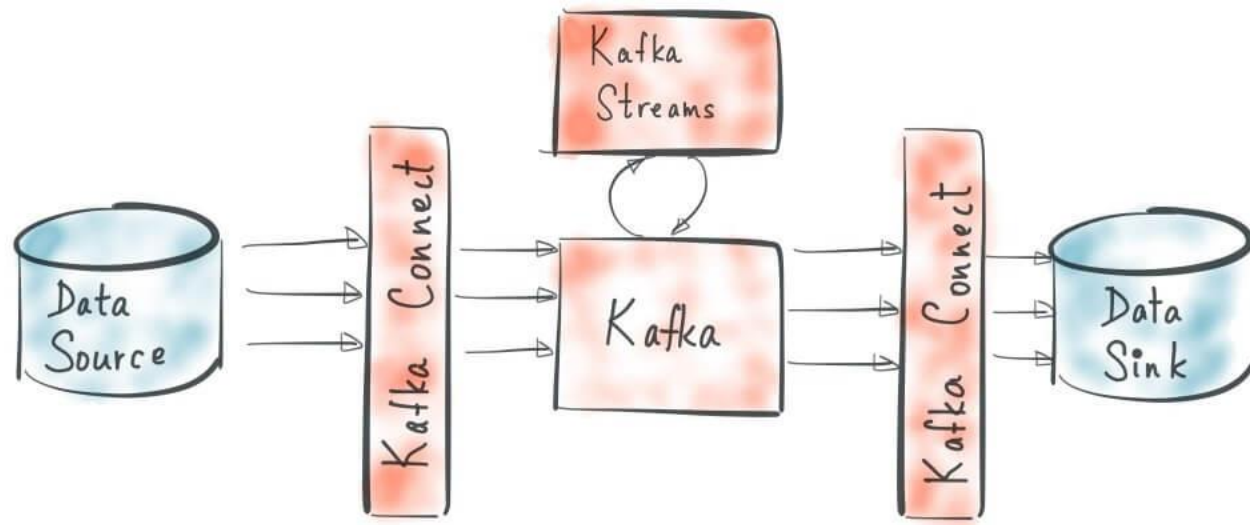
Kafka Streams

Despite being a humble Java library, Kafka Streams directly addresses a lot of the hard problems in stream processing:

- Event-at-a-time processing (not microbatch) with millisecond latency
- Stateful processing including distributed joins and aggregations
- A convenient DSL
- Windowing with out-of-order data using a DataFlow-like model
- Distributed processing and fault-tolerance with fast failover
- Reprocessing capabilities so you can recalculate output when your code changes
- No-downtime rolling deployments

Kafka Streams Architecture

KAFKA CONNECT + STREAMS



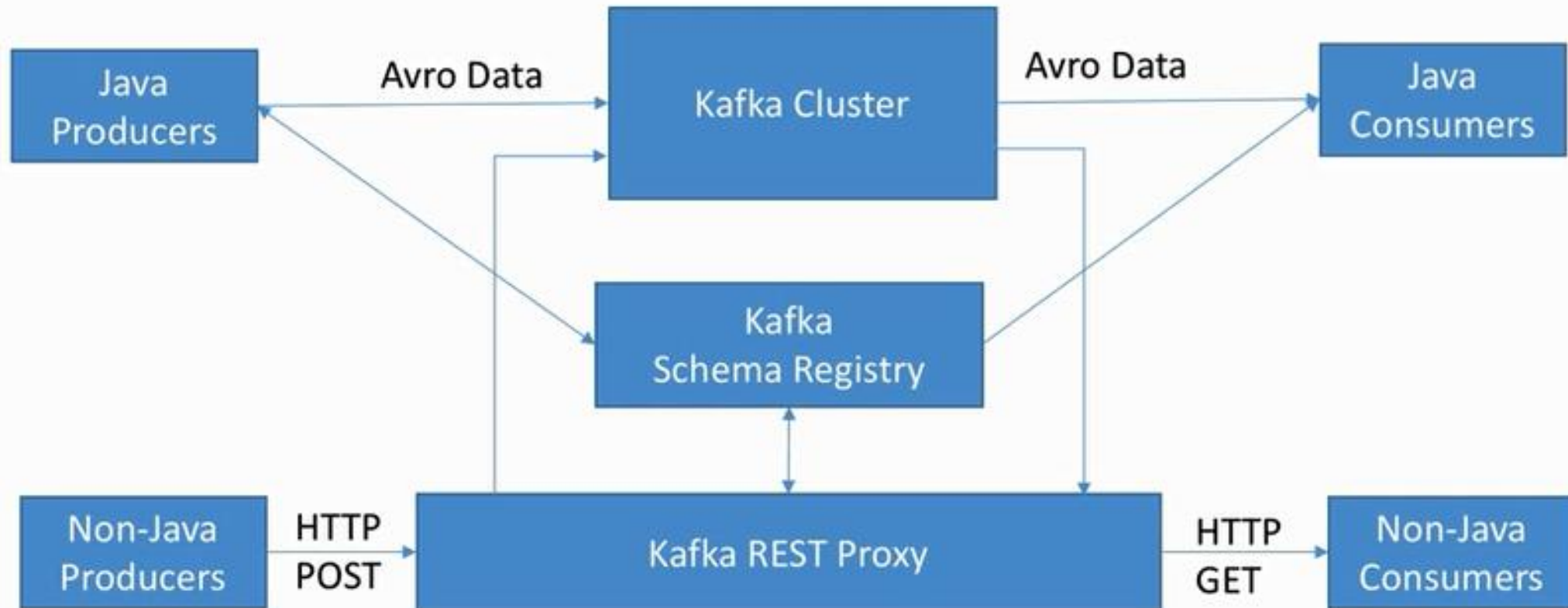
Confluent REST Proxy

The Confluent REST Proxy provides a RESTful interface to a Kafka cluster, making it easy to produce and consume messages, view the state of the cluster, and perform administrative actions without using the native Kafka protocol or clients.

Some example use cases are:

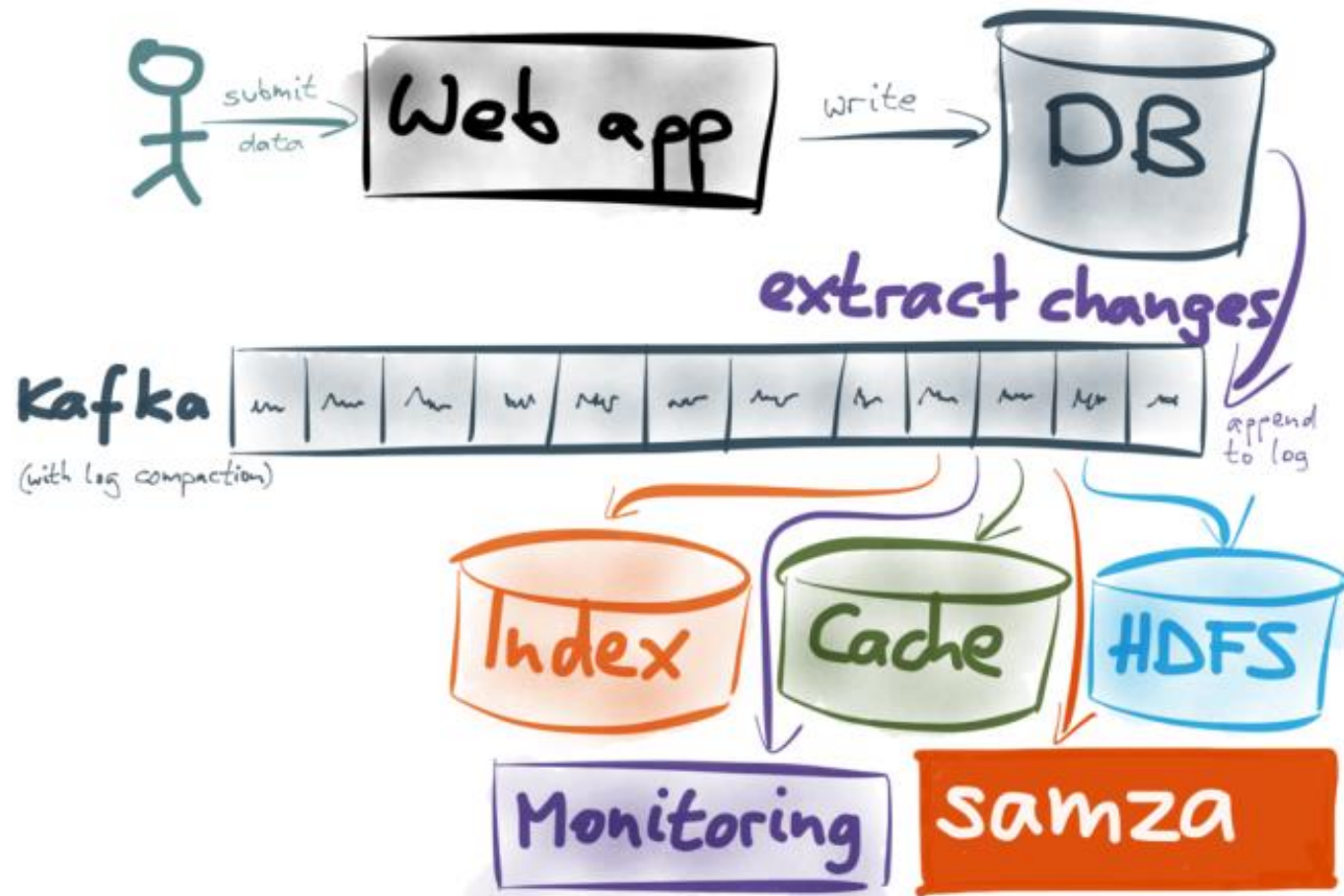
- Reporting data to Kafka from any frontend app built in any language not supported by official Confluent clients
- Ingesting messages into a stream processing framework that doesn't yet support Kafka
- Scripting administrative actions

Confluent REST Proxy Architecture

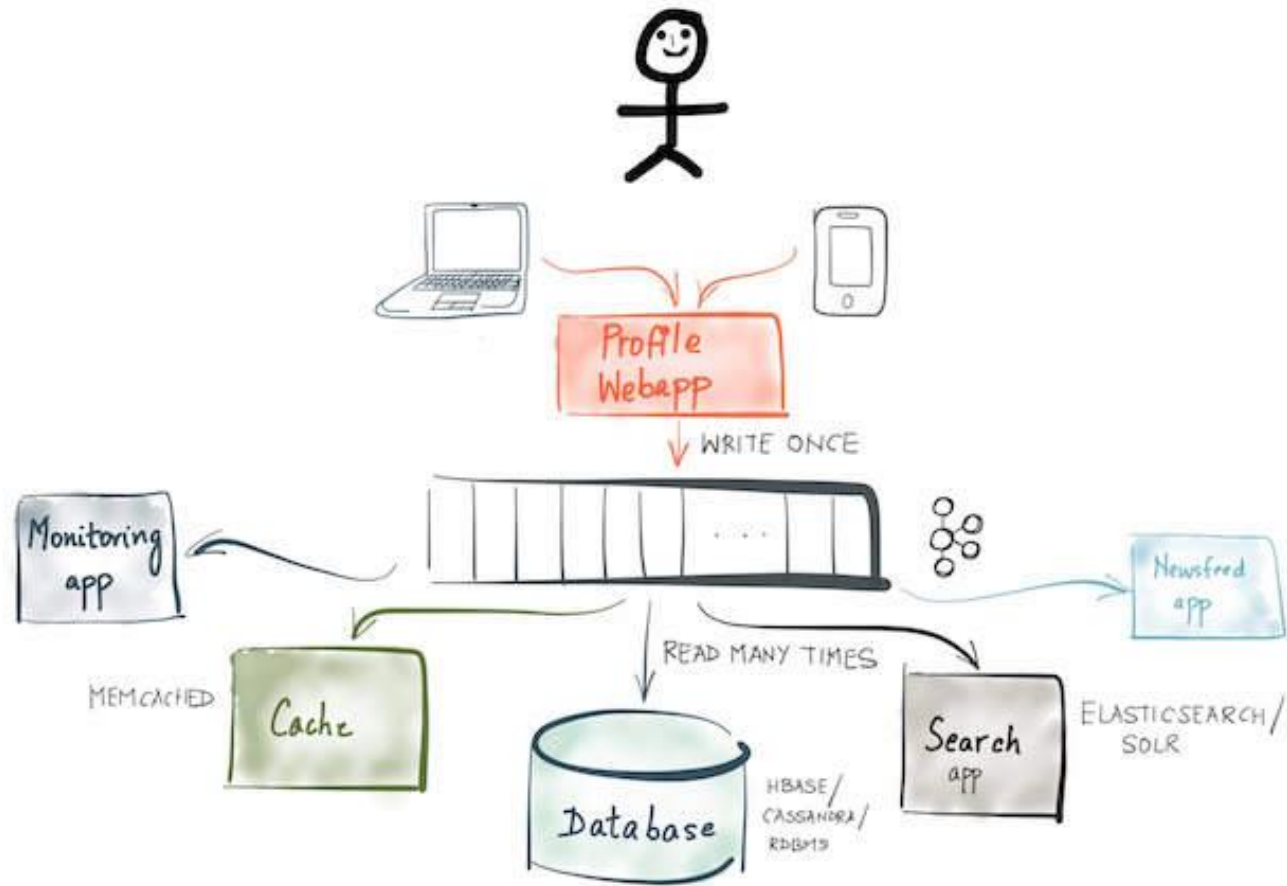


Change Data Capture use case

USING CHANGE CAPTURE



Event Sourcing use case



Kafka Reliability - Be Safe, Not Sorry

Disable Unclean Leader Election: **unclean.leader.election.enable = false**

Set replication factor: **default.replication.factor >= 3**

Set minimum ISRs: **min.insync.replicas = 2**

Leader will wait for the full set of in-sync replicas to acknowledge the record: **Acks = all**

Make sure producer doesn't just throw messages away: **block.on.buffer.full = true**

Consumer Recommendations: **autocommit.enable = false**

Manually commit offsets after the message data is processed / persisted: **consumer.commitOffsets();**

Run each consumer in its own thread

Retries = MAX_INT

Max.inflight.requests.per.connect = 1

Producer.close()

Auto.offset.commit = false

Monitor!

Try Kafka – Mac OS summary

Install brew if needed: `/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"`

Download and Setup Java 8 JDK:

- **brew tap caskroom/versions**
- **brew cask install java8**

Download & Extract the Kafka binaries from <https://kafka.apache.org/downloads>

Install Kafka commands using brew: **brew install kafka**

Try Kafka commands using **kafka-topics** (for example):

Edit Zookeeper & Kafka configs using a text editor

- `zookeeper.properties: dataDir=/your/path/to/data/zookeeper`
- `server.properties: log.dirs=/your/path/to/data/kafka`

Start Zookeeper in one terminal window: **zookeeper-server-start config/zookeeper.properties**

Start Kafka in another terminal window: **kafka-server-start config/server.properties**

Try Kafka - Linux summary

Download and Setup Java 8 JDK: **sudo apt install openjdk-8-jdk**

Download & Extract the Kafka binaries from <https://kafka.apache.org/downloads>

Try Kafka commands using **bin/kafka-topics.sh** (for example)

Edit PATH to include Kafka (in **~/.bashrc** for example) **PATH="\$PATH:/your/path/to/your/kafka/bin"**

Edit Zookeeper & Kafka configs using a text editor

- **zookeeper.properties:** `dataDir=/your/path/to/data/zookeeper`
- **server.properties:** `log.dirs=/your/path/to/data/kafka`

Start Zookeeper in one terminal window: **zookeeper-server-start.sh config/zookeeper.properties**

Start Kafka in another terminal window: **kafka-server-start.sh config/server.properties**

Try Kafka – Windows summary

Download and Setup Java 8 JDK

Download the Kafka binaries from <https://kafka.apache.org/downloads>

Extract Kafka at the root of C:\

Setup Kafka bins in the Environment variables section by editing Path

Try Kafka commands using **kafka-topics.bat** (for example)

Edit Zookeeper & Kafka configs using NotePad++ <https://notepad-plus-plus.org/download/>

- zookeeper.properties: dataDir=**C:/kafka_2.12-2.0.0/data/zookeeper** (yes the slashes are inversed)
- server.properties: log.dirs=**C:/kafka_2.12-2.0.0/data/kafka** (yes the slashes are inversed)

Start Zookeeper in one command line: **zookeeper-server-start.bat config\zookeeper.properties**

Start Kafka in another command line: **kafka-server-start.bat config\server.properties**

Q&A