# Apache Kafka in a nutshell

**This is first in series of articles about Apache Kafka written by Vajo Lukic. The purpose of this article is to help everyone who is interested in Kafka, to understand what it is and to reduce the learning curve of learning Kafka.**

## Introduction

Whether you are an experienced IT professional or just starting your IT career, it can be a little bit overwhelming to start with some new technology. First learning steps can be difficult if the learning curve is steep, and sometimes people give up to early. My goal here is to help you to climb that first hill and to successfully acquire new skills and learn to use Apache Kafka.

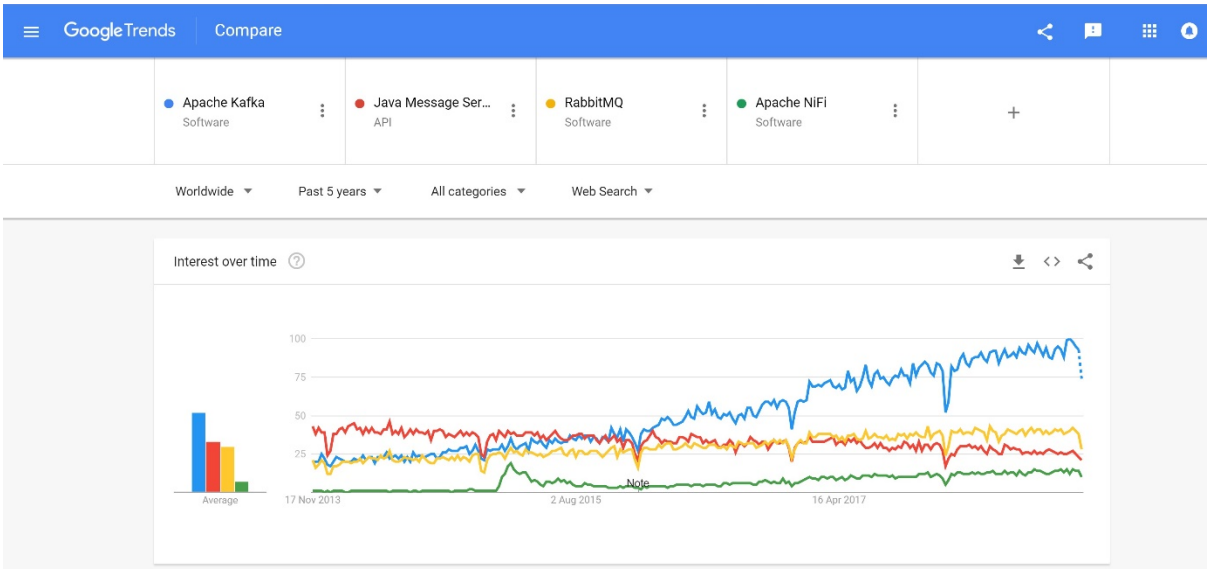
When I started learning about Apache Kafka two years ago, I did not even dream about giving a presentation about it sometime. And that is exactly what has happened recently. My colleagues from Webstep AB here in Stockholm, Sweden, were very kind to invite me to present Apache Kafka to them. It has been an honour to accept the invitation and to get a chance to have a friendly and stimulating discussion about Kafka.

If you want to drop by and take part in one of future Webstep events, you can find the event calendar here: Webstep Facebook events.

## Motivation for learning Kafka

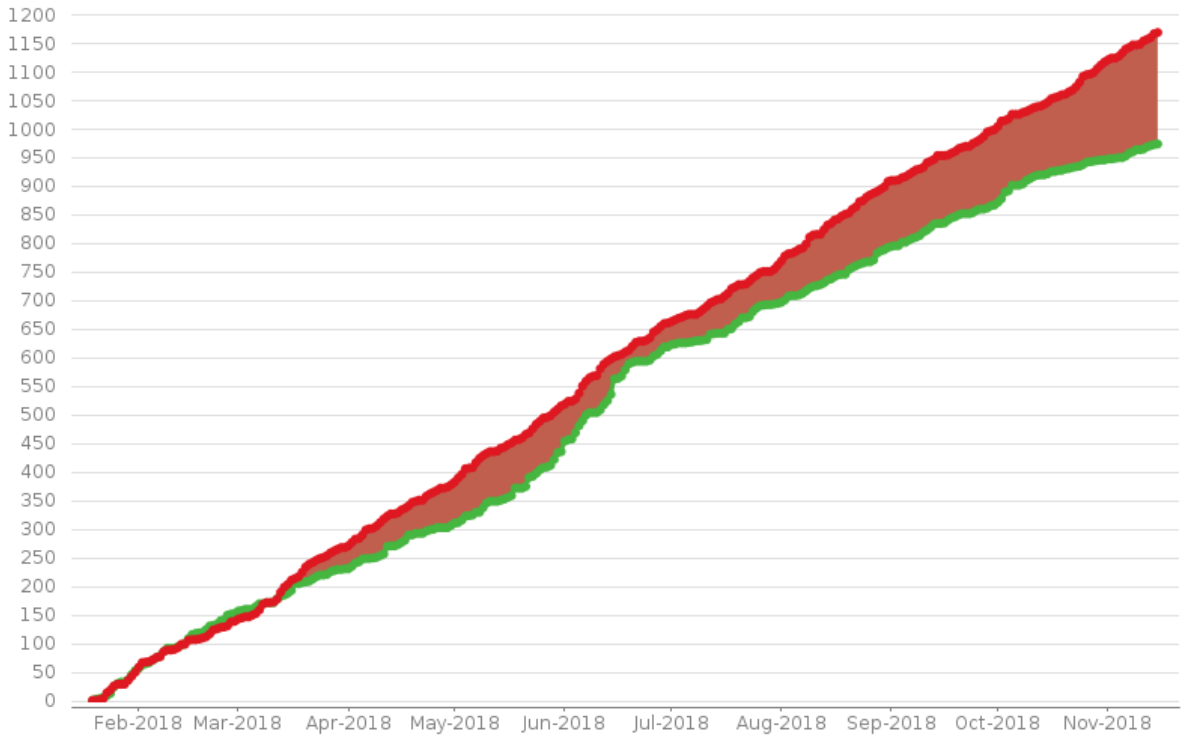
For all of us IT professionals the time is extremely valuable. There is never enough time and sometimes 24 hours are too short to explore, test and learn all exciting new technologies. For that reason, we need to make decisions and to choose wisely what to spend the time on, and what to ignore or leave for later.

Kafka has become one of the most popular Apache projects. It is enough to look at Google trends and see how public interest in Kafka compares with other similar technologies. In the last 5 years interest in apache Kafka has increased more than 300% while some other technologies JMS for example are stagnating.




When you are deciding whether to invest your precious time in learning some new "hot" open-source technology, one very good way to do that is simply to check the interest and time investment by the open-source community.

In Kafka's case, that interest is still growing strong and here for example, you can see number of open and closed issues over time:




Actions speak louder than words and here you can clearly see that there are many people who are putting the time and effort into Kafka, simply because they find it valuable. Strong open-source community is a guarantee that certain technology is going to be there for some time, and therefore is worth learning and investing the time into.

Kafka has gained huge popularity and success in companies like LinkedIn, Netflix, Spotify, Klarna etc. Thanks to its properties for high throughput, horizontal scalability and resilience. While you might not have millions of customers like these companies, Kafka can still provide very valuable services like: increased team agility, quick access to data, real-time data processing, easy data integration and asynchronous service communication.

Here you can find an extensive list of companies which are "powered by Kafka".

# Core Kafka concepts

So, what is Kafka? Depending on how you use it in your enterprise architecture, Kafka can have many faces. Because of the role it plays and its properties, it can be described like this:

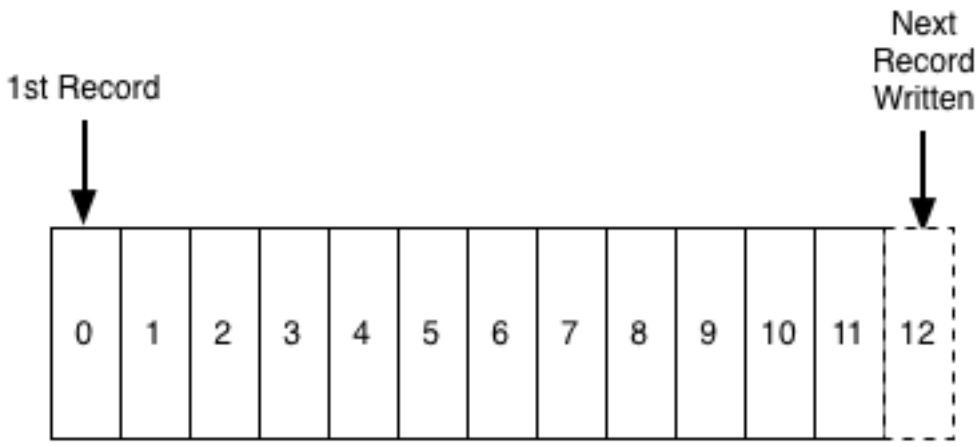
- Kafka is a distributed, horizontally-scalable, highly available, fault-tolerant **commit log**
- Kafka is a publish-subscribe messaging system
- Kafka is a Streaming Data platform

Kafka's architecture is built around concepts like: log, message, schema, retention, partition, topic, broker, producer, consumer, offset. Let's describe some of these concepts in more details.

## The Log

Log is probably the simplest possible abstraction of a storage of any kind. Fundamentally it is just an append-only, totally-ordered sequence of recorded changes. Records are only being appended to the end of the log and each entry is assigned a unique sequential number.

Any reads are being executed from left to right. By replaying all the records from the log exactly in the same order as they have happened, we should be able to re-create current state at any time we want, again and again if we have the log.



Order of records in the log is indicating the time flow. Records on the left are considered as "older" than those on the right. Logs are very important in Kafka because that is where all the messages are stored.

This blog by LinkedIn is one of the best explanations of logs and their purpose.

## Message and Schema

The unit of data in Kafka is a **message**. A message is like a record or a row in the database. It is important to know that for Kafka, any message is simply an array of bytes. Kafka is completely agnostic to the message content. It does not care what the message is. Kafka does not interpret it or validate it, it just stores it. Each message can also contain a piece of meta data which is considered as the *key*. With this, each message can be considered as key-value pair.
While message content is meaningless to Kafka, it is advised that a **schema** is enforced on each message so that it can be understood and interpreted by any services or people who need it. A schema can be created with one of many available serialization formats like JSON, XML or Avro, depending on your application preferences.

Formats like JSON and XML are often used simply because they are convenient, human-readable formats, but it is much more efficient to use **binary formats** like Apache Avro, Google's Protocol Buffers and Apache Thrift. These formats are much better fit for inter-service communication and data serialization.

## Apache Avro

Is a recommended format for Kafka because of its properties. Apache Avro is binary data serialization system which provides rich data structures. Avro needs less encoding since it stores names and types in the schema reducing duplication. One of its most important properties is that it supports the evolution of the schema.

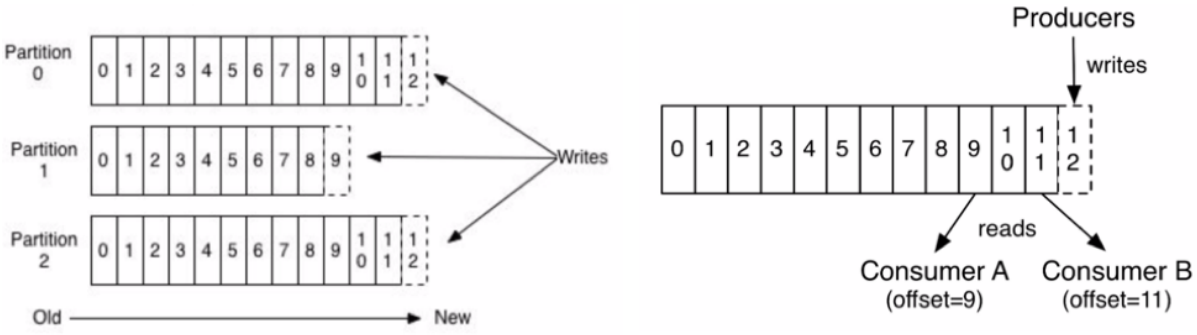
Avro does not require code generation. It integrates well with Java Script, Python, Ruby, C, C#, C++ and Java. Avro gets used in the Hadoop ecosystem as well as by Kafka. It is very similar to Thrift, Protocol Buffers, JSON, etc.

## Topics and Partitions

In Kafka messages are organized in *topics*. You can think of topics as tables in the database or as folders in the file system. Topics are further split into several *partitions*. Partitions are unique to Apache Kafka and are not seen in the traditional message queuing systems.

Partition can be considered as a log. When writing, messages are appended to the end of the partition preserving the order. Messages are written to one partition but copied to at least two more partition replicas for redundancy. When reading, they are read from "left" to "right".

Since each topic is split into multiple partitions, it is important to remember that order is only guaranteed inside the partition, but total order across entire topic and all partitions is not.




## Producers and Consumers

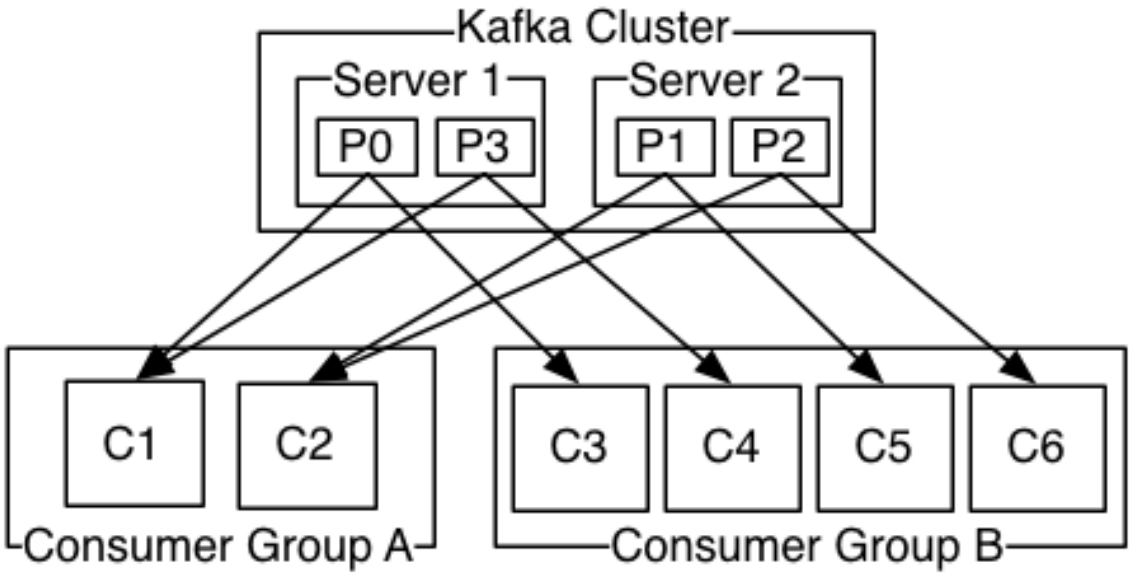
There are two types of client users of Kafka: **producers** and **consumers**. Producers are creating new messages. In other messaging system they might be called *publishers*. Message is generally produced to a specific *topic*. Producers do not write to a specific partition, they evenly distribute messages to all partitions of the topic. Default partitioning strategy for messages without a key is round-robin. For messages with the key, producer will always send messages with the same key to the same partition.

It is also possible for producers to direct messages to a single partition. To achieve this, producer relies on the message key and on a partitioner to generate the hash of the key, and to map it to a partition.
Consumers read messages and in other system they can be called *subscribers* or *readers*. Consumers subscribe to one or more topics and read messages in the same order hey have been written.

**Offset** is a piece of meta data that Kafka adds to each message that is being produced. Offsets are integer values that continually increase.

Consumers are using offsets to keep the track of messages which have been read. Each message has a unique offset in some partition. By storing the offset of the last consumed message, consumer can be stopped and started again, without losing the track of data it has processed. Offset are usually stored in Kafka, in special topics reserved for that.
One or more consumers that read together from the same topic are forming a *consumer group*. This makes sure that each partition is being read by one member of the group. One consumer can read from one or more partitions, but one partition can only be consumed by one consumer, to avoid duplication of read messages.

Consumer groups are enabling easy horizontal scaling. Also, if some consumer fails, remaining members of its group will re-balance the partitions they are consuming, to make sure that all the messages are being read.



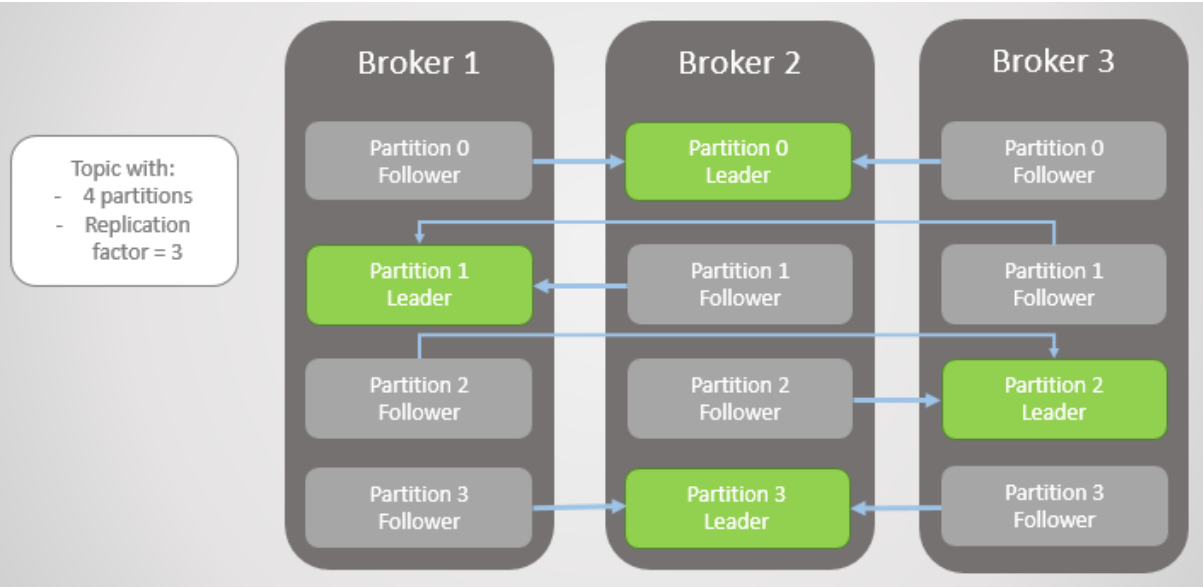

## Brokers and Clusters

A **broker** is a single Kafka server. They receive messages from producers, assign offset to them and store them to disk. They also serve consumers, serving them messages on their request. Depending on its hardware, a single broker can easily process millions of messages per second and manage thousands of partitions.

Kafka brokers usually operate as part of the *cluster* where one broker is acting as a *controller*. The controller assigns partitions to brokers, monitors broker failures and performs other administrative tasks. Although a single partition can be assigned to many brokers, only one broker at the time is considered as the owner of partition or as a *leader*. All read and write operations on some partition must go through the leader of that partition.

Other brokers provide the redundancy of messages in a partition, so that some other broker can take leader's role in the case of current leader failure. Those brokers that have the same data (messages) as the leader, are called *in sync replicas*.

When setting up a Kafka cluster for the first time, one of the most important parameters is a "number of in sync replicas". With this parameter you choose data consistency or data availability in the case of failures. The higher the number of in sync replicas, the lower the chance of losing any data, but higher time when writing to Kafka, and vice versa.




## Retention

One of the key properties of Kafka is the data *retention*. This is where Kafka differs from other messaging technologies which usually do not persist the data for longer periods. With other messaging technologies, message read is usually a destructive operation. Meaning that message gets deleted after read operation has been confirmed.

This is not the case with Kafka. Kafka brokers have a default setting for topics to retain the data either for some period (e.g., 14 days) or when the topic reaches certain size in bytes (e.g., 1 GB). When these predefined limits are reached, messages are expired and deleted. Also, individual topics can be configured to keep the data if it provides value.

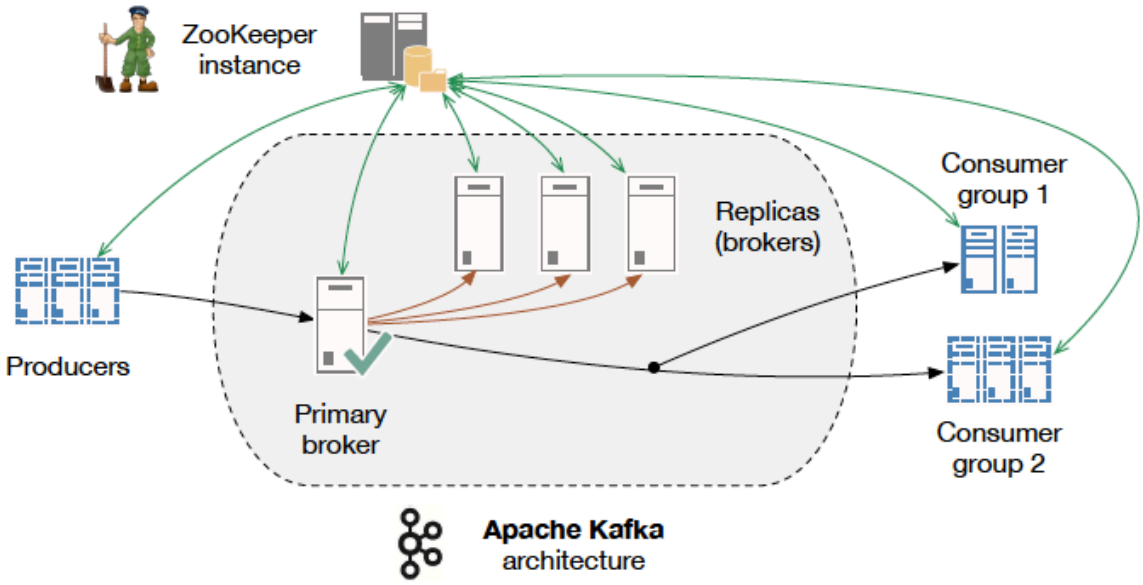
This property can be extremely useful in certain situations. For example, if the source system is serving messages much faster than the target system can process them, Kafka who sits in the middle, can retain the messages until target system gets upgraded or gets freed of its current big processing load. With this Kafka acts as a *buffer* between different systems and prevents load crashes or big delays.

Another such case is data reprocessing for Business intelligence and reporting purposes. Since Kafka retains the data for longer periods, any late arriving messages can easily be added to current snapshots, or any errors in those snapshots can be fixed simply by replaying the same messages once again, through updated version of the data processing module.

## Zookeeper

One other important part of Kafka architecture is an Apache Zookeeper. Zookeeper is a centralized service for distributed systems like Kafka is. Zookeeper is **necessary for Kafka to**

**run**. It keeps all meta-data information for Kafka: broker information, topic configuration, cluster membership and performs leader election.




# Kafka benchmarks

These numbers are a little bit old, but still very relevant. The best thing about this is that with newer versions of Kafka, these numbers just got improved even more. Benchmark 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)

## Why is Kafka so fast?

It uses some clever tricks combined with some good engineering principles:

- Partitioning of the topic (writes scale horizontally)
- Consumer groups (reads scale horizontally)
- Optimizing throughput by batching reads and writes
- Using log to store the data: data is only appended to the system, and reads are simple. All such operations are O(1)

- Zero Copy link - calls the OS kernel directly rather than at the application layer to move data fast
- Uses standardized binary data format for producers, brokers and consumers (so data can be passed without modification)

### Kafka guarantees

To summarize - when considering all its properties, Kafka provides some very valuable guarantees:

- 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, if at least one "in sync replica" is alive.

With these guarantees, one can design different types of very valuable systems: real time stream processing systems like fraud detection, customer recommendations, legacy systems data extraction to Kafka through CDC (Change Data Capture), ETL (Extract - Transform - Load) data pipelines for Business intelligence and Reporting systems, "event-sourcing" systems like ledger type of applications, micro-services applications etc.

## Kafka ecosystem

There are some other exciting technologies which have emerged around Kafka. Some are part of regular Kafka installation, some must be installed separately.

Although they are very much worth looking into, we are just mentioning them now and we will describe them in more detail in one of following articles:

- Kafka Streams: a client library for building Java and Scala applications where input and output are written to Kafka. More about it here.
- Kafka Connect: is a framework for connecting Kafka with external systems such as databases, key-value stores, search indexes, and file systems (documentation).
- Confluent Schema Registry: is a distributed storage layer for Avro Schemas which uses Kafka as its underlying storage mechanism (documentation).
- Confluent REST Proxy: provides a RESTful interface to a Kafka cluster, making it easy to produce and consume messages, and perform administrative actions without using the native Kafka protocol or clients (documentation).

# Final words

I personally believe that one of the biggest benefits of Kafka is that it decreases "time-to-value" for **data**. By being a back-bone for all the data in organization, removing "silos" around data from different domains, enabling Data Scientist and data engineers to get quick access to data - all this provides a tremendous competitive advantage to any company in this "age of data" that we are living in.

That is why I wholeheartedly recommend to every IT professional to dive more into Kafka, learn how to use it and build some exciting applications.

***By Vajo Lukic***