Apache Kafka Fundamentals

Student Handbook

Version 7.1.1-v1.0.0

CONFLUENT

Table of Contents

01 Introduction

Copyright & Trademarks

Agenda

1. Introduction ...
2. Motivation & Customer Use Cases
3. Apache Kafka Fundamentals
4. How Kafka Works
5. Integrating Kafka into your Environment
6. The Confluent Platform
7. Conclusion

---

2. Motivation & Customer Use Cases
   - Why are learners here to listen?
   - Why is it relevant?
   - Some interesting use cases our customers solve with Kafka
3. Apache Kafka Fundamentals

   This module gives motivation for stream processing, and a high level overview of the following:

   - Broker
   - Producer
   - Consumers, consumer offsets
   - Topics, Partitions, Replication
   - The (commit) log and streams
   - High level architecture
   - ZooKeeper
4. How Kafka Works
   - Introduction to development
   - Partition Leadership & Replication
   - Data Retention Policy

- Producer Design & Guarantees
- Delivery Guarantees, Idempotent Producers, EOS
- Partition Strategies
- Consumer Groups and Consumer Rebalances
- Topic Compaction
- Troubleshooting
- Security Overview

5. Integrating Kafka into your Environment
   - REST-Proxy
   - Schema-Registry
   - Kafka Connect
   - Kafka Streams
   - KSQL
   - Confluent Control Center
6. Confluent Platform
   Explaining the added benefit of CP over bare Apache Kafka
   - Central Nervous System
   - The maturity Model
   - Confluent Platform
   - Confluent Cloud
   - Confluent Control Center
   - Confluent CLI
   - RBAC
   - Confluent Operator
   - Confluent Hub and certified connectors

Course Objectives

After this course, you will be able to:

- Explain to an interested lay person why "**Event-driven**" is important
- Explain the concepts of **Topics** and **Partitions**
- List the main responsibilities of a **Broker**
- Describe how **Producers** and **Consumers** work
- Setup a minimal **Kafka Cluster**
- List 3 to 4 features of **Confluent Platform** that enterprises require

Accessing Lab & Course Materials

Trainer, please demo the access of the CR environment, including the VM and how to best access the exercise book.

Class Logistics

- Start and end times
- Can I come in early/stay late?
- Breaks
- Lunch
- Restrooms
- Wi-Fi and other information
- Emergency procedures

⚠	No recordings please (audio, video)

Introductions

- About you:
  - Your Name, Company and Role
  - Your Experience with Kafka
  - Other Messaging or Big Data Systems, you use
  - OS, Programming Languages
  - Your Course Expectations
- About your instructor

02 Motivation & Customer Use Cases

Agenda

1. Introduction
2. Motivation & Customer Use Cases ...
3. Apache Kafka Fundamentals
4. How Kafka Works
5. Integrating Kafka into your Environment
6. The Confluent Platform
7. Conclusion

Why Are We Here?

Why are we all here? Why do we care about Kafka? What's in it for us? These are questions that we hope to address in this module. We will provide some use cases from real customers of Confluent that have solved mission critical problems by using Kafka to power their real time event streaming platforms.

Stay tuned!

Motivation

The Shift to Event-driven Systems has Already Begun...

From a static snapshot... **...to a continuous stream of events**

Occasional call to a friend

A constant feed about the activities of all your friends

Daily news reports

Real time news feeds, accessible online anytime, anywhere

Motivation

This leads us to...

Single platform to connect everyone to every event

Real-time stream of events

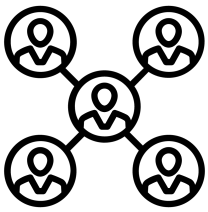
All events stored for historical view

This slide summarizes what makes real-time systems for the scenarios shown on the last slide possible.

Motivation

Successful Digital Businesses are Inherently Event-driven

Born cloud-native...

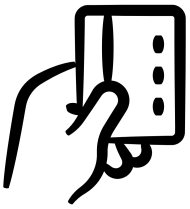


Social Networks
Enabling Event Sharing

Streaming Provider
On-demand Digital Content

Traditional ones that adapt...

Newspaper
Provide a single Source of Truth



Credit Card Payments
Microservices Architecture

Motivation

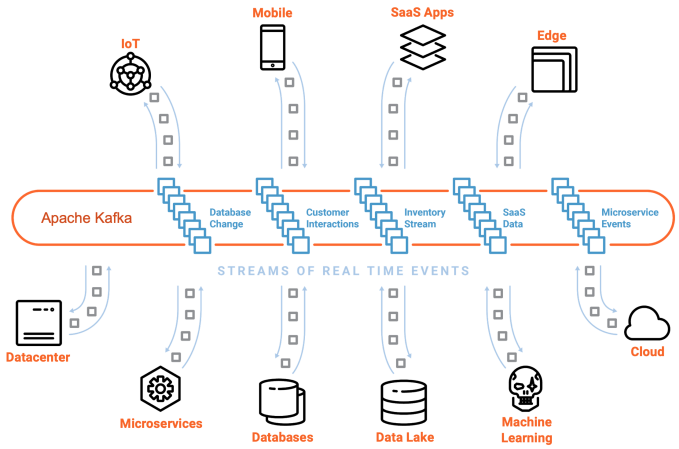
Born cloud-native...

Ride Hailing
Connecting Provider with Consumer in real-time

Traditional ones that adapt...

Connected Cars
IoT - Real-time Traffic Routing

Motivation



Apache Kafka®: the De-facto Standard for Real-Time Event Streaming

- Global-scale
- Real-time
- Persistent Storage
- Stream Processing

The event streaming paradigm enables enterprises to transform into data-driven businesses.

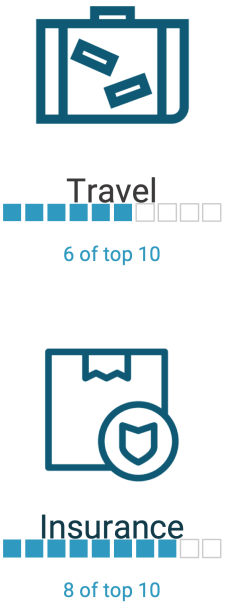
Motivation

Thousands of Companies Worldwide trust Kafka for their Journey towards "**Event-driven**"

https://kafka.apache.org/powered-by

Motivation

Over 70% of Fortune 500's Already Trust Kafka for Mission Critical Apps



Reference:

Over 70% Fortune 500's

Real-time Fraud Detection

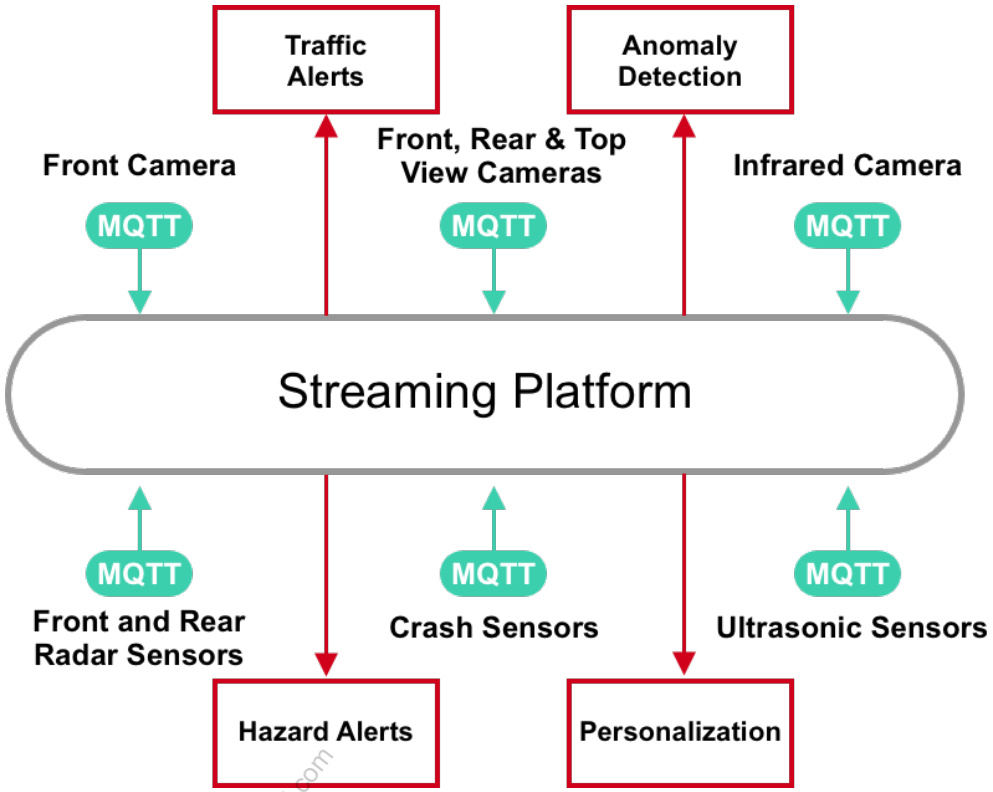
- Act in real-time
- Detect fraud
- Minimize risk
- Improve customer experience

Leading Credit Card Company

The company needed a streaming platform to move transactional and business-critical data between all of their apps in real-time to not only ensure high-quality customer experience, but also to minimize risk by detecting fraud and managing approval and rejection status for card services. These applications also needed to seamlessly interact with the company's loyalty platform in real-time, as membership rewards management impacts both card holders and merchants alike.

Automotive

The Future of the Automotive Industry is a Real Time Data Cluster



CHALLENGE

To increase revenue and support their mission of "Advancement through Technology," Audi developed a platform to collect and analyze the vast amounts of data from its fleet. An autonomous vehicle driven for just a few hours generates over four terabytes of data, which is useless if that data cannot be organized and used to generate business insights. To make use of this data and bring their "Swarm Intelligence" vision to life, Audi deployed a streaming platform to reveal once undiscoverable and actionable business insights. They can now process data about the car's sensors, and components and use "Predictive A/I," which is the ability for a car to anticipate changes in its state and surrounding.

SOLUTION

Confluent Platform was chosen to be the streaming platform for their **Automotive Cloud Data Collector** (ACDC). By using the insights gained in real-time via the platform, Audi will be able to leverage the platform and use it to avoid accidents, suggest vacant parking spots, save people a lot of time and maneuver thousands of cars around obstacles in the road. Kafka is also used to connect the data from the ACDC to other approved systems requesting the data from the platform to guide business decisions. Audi selected Kafka as the streaming backbone for a platform that unifies the entire Volkswagen Group of

connected vehicles for its scalability, light-weight features, and ability to connect to all their internal systems. Data will be collected from the cars and streamed in real time to several regional hubs around Germany.

RESULTS

- Worldwide tracking and data gathering of Audi cars to improve user experience
- Providing sales, engineering and post-sales departments with the data required to provide additional or better services to customers
- Increased availability and scalability

Real-time e-Commerce

Rewards Program

- Onboarding new merchants faster
- Increased speed at which mobile applications are delivered to customers
- Enabled a full 360 view of customers
- Enhanced performance and monitoring
- Projected savings of millions of dollars

CHALLENGE

The **Bank** sought to launch an eCommerce digital platform, a rewards program that allows customers to earn points and cash when they use their **Bank** card to make purchases from hundreds of bank sponsored merchants, such as Starbucks and Walmart. As their first use case, the platform would allow the **Bank** to have exponentially more checkpoints with their customer and improved merchant-analytics, and would require the real-time management of large amounts of data. Batch processes, using Informatica, were part of their architecture so data was not updated in real time. Ultimately, the **Bank** was having difficulty in delivering meaningful mobile applications to their customers.

SOLUTION

To help manage the vast amounts of data coming from the eCommerce platform program, Confluent Platform is now the standard event streaming platform for Mobile Commerce and Technology within the Consumer Bank.

RESULTS

- Ability to onboard new merchants into the eCommerce platform program faster
- Increased speed at which mobile applications are delivered to their customers
- Enabled a full 360 view of their customers that was not possible before
- Enhanced performance and monitoring
- Projected savings of millions of dollars by reducing reliance on IBM MQ, Informatica and

their preexisting mainframe

Customer 360

- Improved data integration
- Increased up-sell and cross-sell opportunities
- Increased scalability and flexibility
- Saved costs

CHALLENGE

With rising expectations from customers to leverage mobile, email and SMS communications, **The Insurance Company** was struggling to keep up with the ever changing ways to interact with their retail insurance customers. They also struggled to achieve a true 360 view of customer interactions and wanted to leverage omnichannel marketing best practices.

SOLUTION

The Insurance Company implemented Confluent Platform to get a real time view of their customer interactions, email history and interaction, as well as advertisement impressions and success rates. Confluent enabled **The Insurance Company** to gain a complete view of what products a customer had purchased, which enabled the identification of up-sell and cross-sell opportunities.

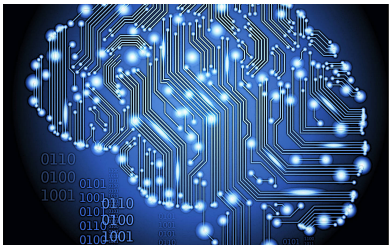
RESULTS

- Improved data integration across the organization
- Increased up-sell and cross-sell opportunities
- Increased scalability and flexibility of their mission-critical applications

- Saved costs by reducing the reliance on costly IBM Unica and IBM MQ solutions

Core Banking

- Empowered successful core banking platform relaunch
- Met HA and SLA needs
- Improved scalability



- Power AI for Chat Bots

CHALLENGE

The Bank built a new core banking platform to support their 150+ million customers. The bank replies on Apache Kafka to process their 200,000 transactions per day, and were in dire need of enterprise-wide support for the January 2018 go-live. They sought to mitigate risk of the platform going down and wanted to improve the time to market, efficiency, analytics, personalization, and the speed of the new core banking platform.

SOLUTION

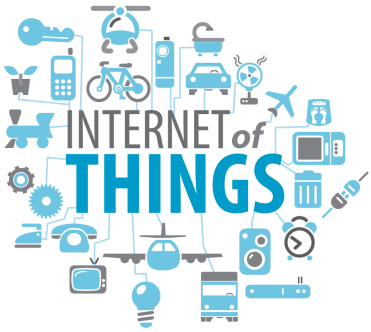
Apache Kafka with Confluent support enables **The Bank** to use Kafka to build a new Core Banking Platform that run all their existing and new business models on. These range from standard banking applications, such a fraud mitigation, to online payments, cashless smart cities, and A/I for chat bots.

RESULTS

- Empowered successful core banking platform relaunch
- Met high availability and SLA needs
- Improved scalability, as the bank anticipates that transactions will triple or quadruple in the in coming years

Health Care

- Microservices



- Internet of Things

CHALLENGE

The **Health Care Company** needed a streaming data platform to support their "Clinics Without Walls" initiative. This initiative aimed to enstate a dedicated server in each of their clinics to capture IoT sensor data at frequent intervals for analysis. They also wanted to improve their steaming analytics capabilities to connect their 3,000 outpatient dialysis centers globally.

SOLUTION

The **Health Care Company** first launched Apache Kafka for their event sourcing pub/sub scenarios. They then adopted Confluent Platform to integrate with the in-clinic hardware to stream patient data every 22 seconds to their central databases.

RESULTS

- Improved ability to more accurately customize medications and dosages for each unique patient
- Achieved stability and agility via microservices event-sourcing architecture

Online Gaming

Data Pipelining

- Increased reliability
- Accurate, real-time data
- Ability to process data at scale
- Faster ramp time

CHALLENGE

The Gaming Company needs to deliver smooth game launches and understand when usage was going to spike. Prior to deploying Kafka, **The Gaming Company** was using a database as a data pipeline, but as the number of players increased from year to year they began running into high IO problems; different applications were all reading and writing to the same few tables and the databases could not keep up with the demand as the data volumes grew. This solution was breaking at the seams. **The Gaming Company** started looking for a solution that could handle a high volume of data at scale, and could provide a fundamental platform for real-time data access.

SOLUTION

The Gaming Company deployed Confluent Platform and Apache Kafka to scale their data pipeline infrastructure depending on usage as they capture and process the various types of data coming in every second from user activity. The data is then sent to teams within **The Gaming Company** who use it to power the various services including diagnostics and optimizing and improving player experiences, which enables designers to adjust and make improvements in the games.

RESULTS

- Increased reliability
- Accurate, real-time data
- Ability to process data at scale
- Faster ramp time with access to Confluent team

Government

Mainframe offload

- Near real-time events and better data quality
- Increased efficiency
- Ability to change their organization
- Produce & store population data from several sources
- Reduce welfare crime through strengthened identity management
- Provide better privacy and meet GDPR requirements

CHALLENGE

NAV has established a vision that "Life is a Stream of Events," and wanted to leverage streaming data technology to associate every major life event of its citizen with an event in a streaming platform. NAV had legacy systems that operated on their mainframe, which led to slow development speeds, technological and organizational dependencies. This left them with no single holistic perspective on individual citizens, which led to a lack of verifiable process outcomes, and inability to track changes to data throughout history.

SOLUTION

NAV has established a vision that each life event triggers an event in Kafka. By doing that, instead of a citizen having to apply for welfare or apply for pensions, the event is automatically triggered and NAV is able to proactively send the benefit to the taxpayer.

RESULTS

- Client-oriented tasks and processes supported by near real-time events and better data quality
- Increased efficiency and ability to change our organization
- Produce and store population data from several sources
- Reduce welfare crime through strengthened identity management

- Provide better privacy and meet GDPR requirements

Financial Services

Customer communications

- Enhanced the customer experience
- Enabled "One Bank" strategy

Payments engine

- Improved fraud detection engine, saving millions of euros
- Grew topics in production by 600 percent

CHALLENGE

Faced with regulatory uncertainty, changing customer behavior and an evolving global banking competitive landscape, **The Company** needed to differentiate through customer experience and technology.

SOLUTION

The company chose Confluent Platform and Apache Kafka to re-architect their technology stack. **The Company** leveraged Confluent Platform to improve fraud detection company wide, to develop of standard of on-time customer communication where customers are only communicated when and via the methods they prefer, and to power their future payments engine.

RESULTS

- Improved fraud detection engine, saving **The Company** millions of euros
- Grew topics in production by 600 percent
- Enhanced the customer experience
- Enabled **The Company**'s "One Bank" strategy

Module Review

Question:

Which area in your company could benefit most from having their apps transformed into an event driven, near real-time applications powered by Kafka?

Hands-On Lab

Refer to the lab **01 - Exploring Apache Kafka** in the Exercise Book.

03 Apache Kafka Fundamentals

Agenda

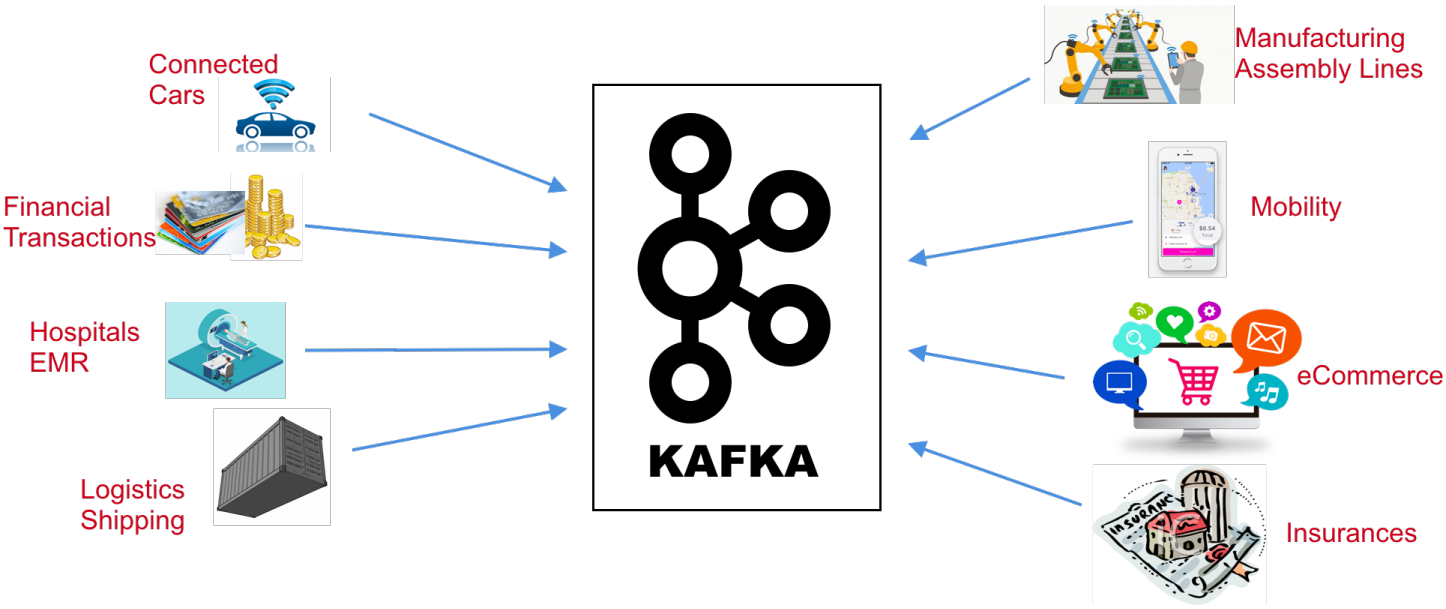
1. Introduction
2. Motivation & Customer Use Cases
3. Apache Kafka Fundamentals ...
4. How Kafka Works
5. Integrating Kafka into your Environment
6. The Confluent Platform
7. Conclusion

Learning Objectives

After this module you will be able to:

- Identify the key elements in a Kafka cluster
- Name the essential responsibilities of each key element
- Explain what a Topic is and describe its relation to Partitions and Segments

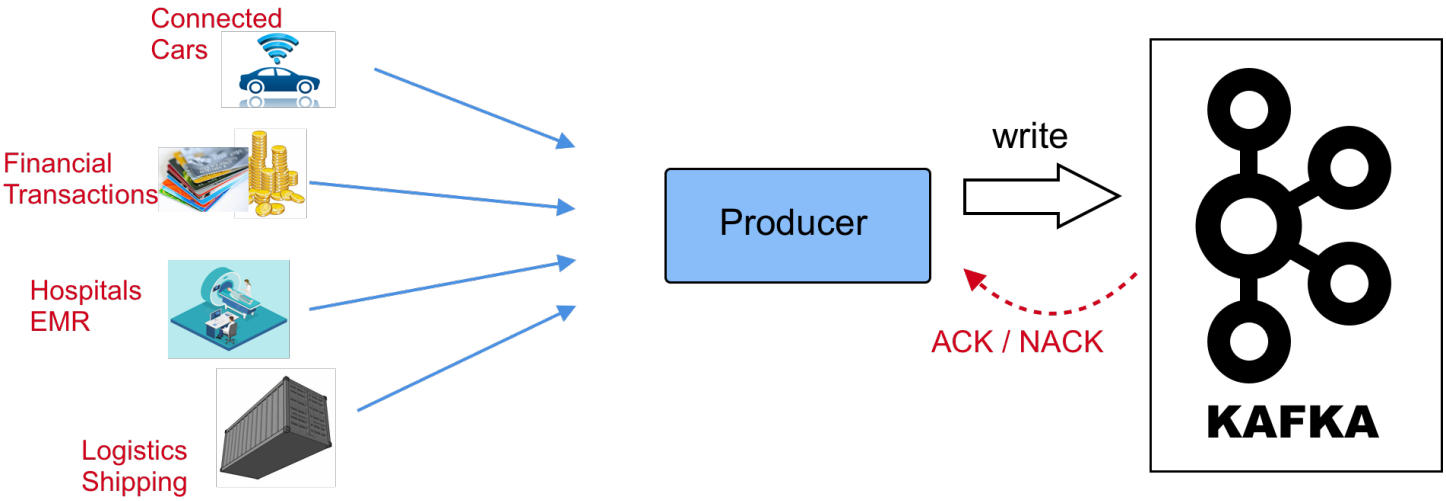
The World Produces Data



The world produces data, loads of it. And we want to collect and store all that data in Kafka. We have

- connected cars whose many detectors emit streams of readings
- hospitals producing a wealth of health related data
- insurances collecting a wealth of customer and environmental data
- banks and other financial institutions producing streams of transactions
- logistics tracking all the goods shipped world wide
- manufacturing automating their whole assembly lines

Producers



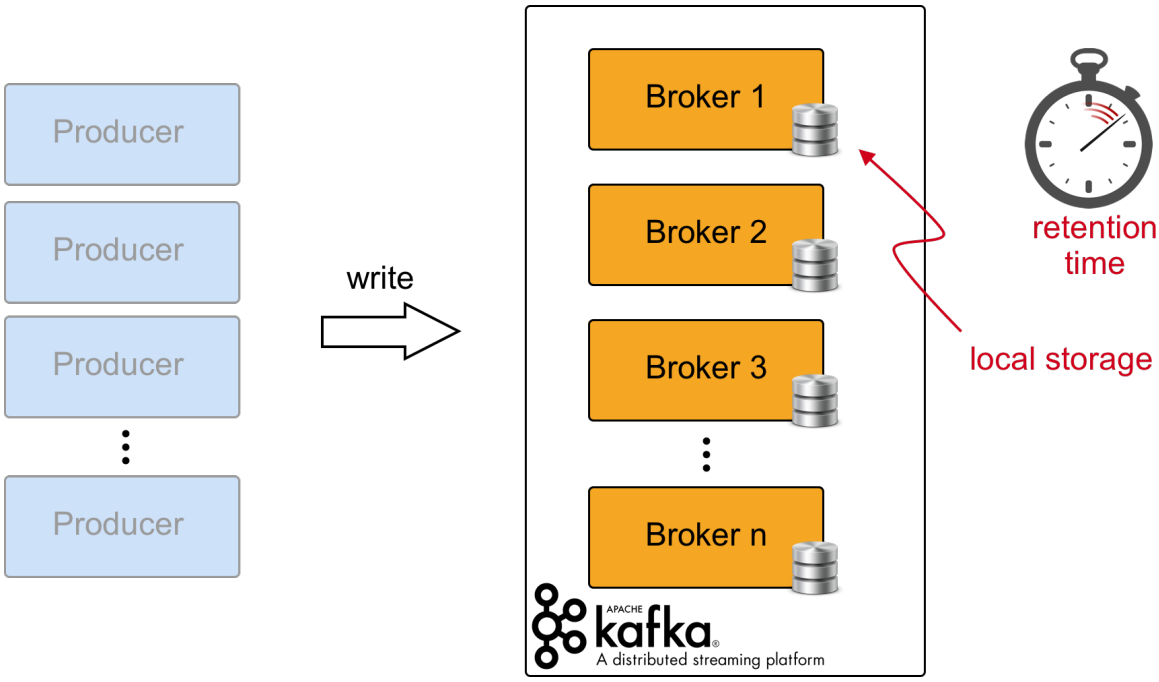
The applications that forward or write all this data to Kafka are called **producers**.

- A producer **sends** data to Kafka
- The producer (optionally) receives an **ACK** or **NACK** from Kafka
- If an **ACK** (acknowledged) is received then all is good
- If a **NACK** (not acknowledged) is received then the producer knows that Kafka was not able to accept the data for whatever reason. In this case the producer automatically retries to send the data.

	The producer also automatically retries if the request times out.

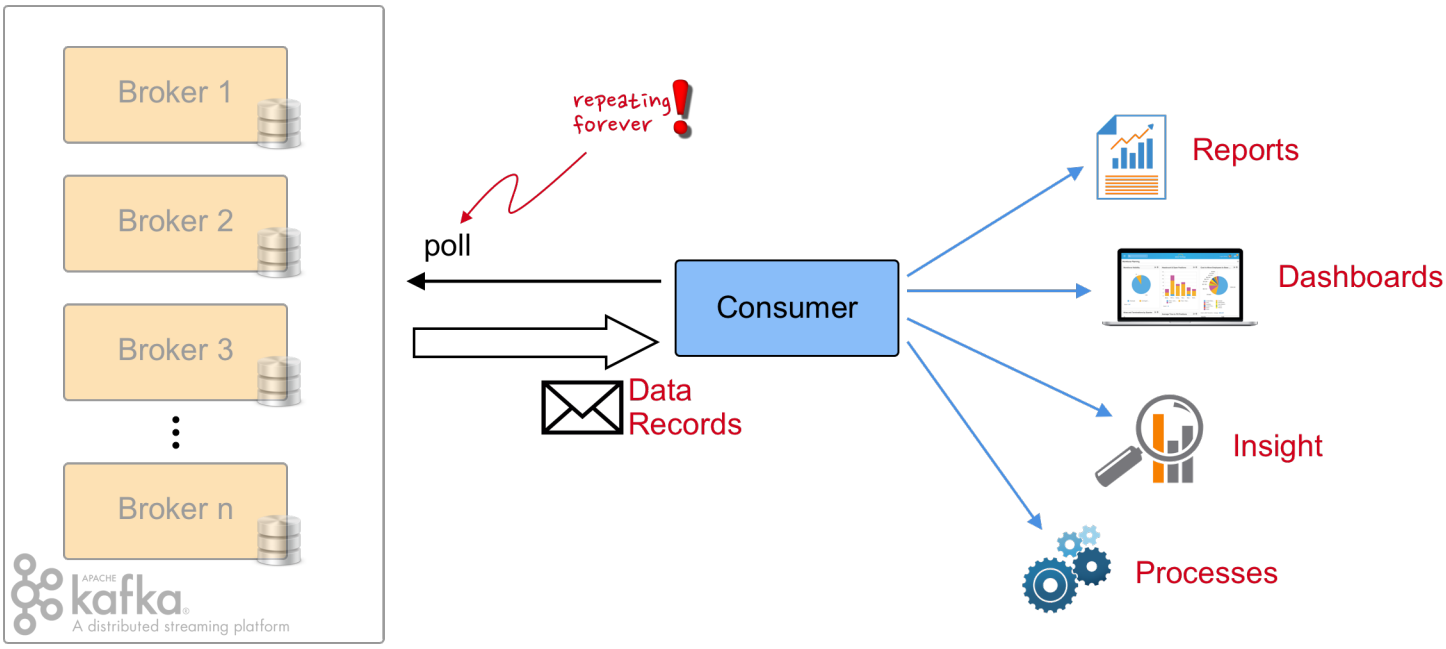
- Many producers can **send data** to Kafka **concurrently**

Kafka Brokers



- Kafka consists of a bunch of what we call **brokers**. More formally Kafka is **a cluster of brokers**
- Brokers receive the data from producers and **store it temporarily** in the page cache, or permanently on disk after the OS flushes the page cache
- Brokers keep the data ready for down stream consumers
- How long the data is kept around is determined by the so called **retention time** (1 week by default)

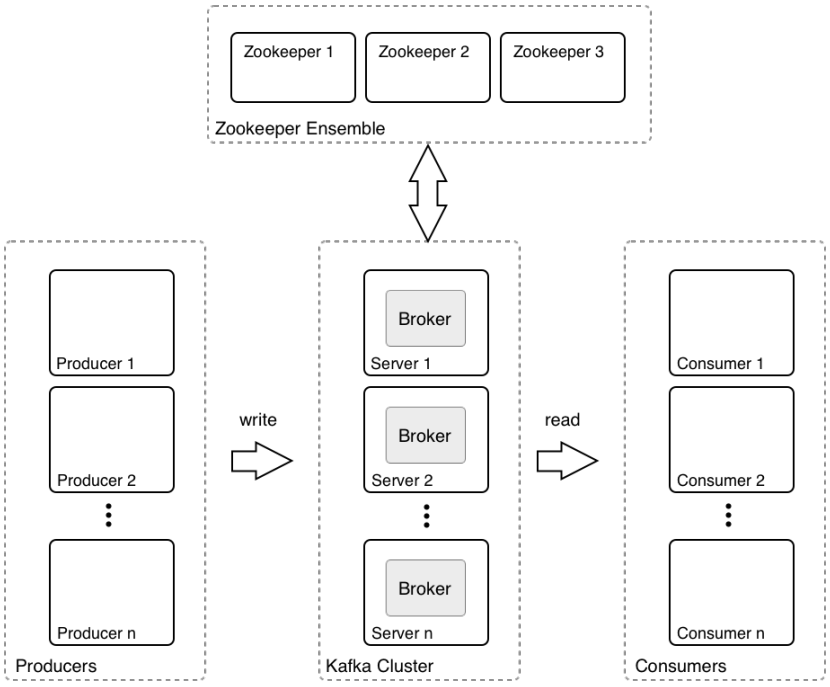
Consumers



Just storing the data in Kafka in most cases makes no sense. The real value comes into play if we have some down stream consumers that use this data and create business value out of it.

- A consumer **polls** data from Kafka
- Each consumer **periodically** asks brokers of the Kafka cluster: "Do you have more data for me?". Normally the consumer does this in an endless loop.
- Many consumers can poll data from Kafka at the same time
- Many different consumers can poll the same data, each **at their own pace**
- To allow for **parallelism**, consumers are organized in consumer groups that **split up the work**

Architecture

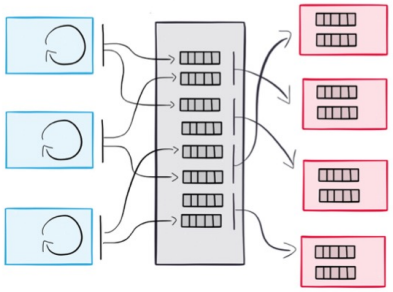


Up to this point, we have learned about the producer, broker, and consumer separately. Now, let's see how these all fit together in an architecture diagram.

- On the left you see producers that **write** data to Kafka
- In the middle you have the Kafka cluster consisting of many brokers that receive and **store** the data
- On the right you see consumers that poll or **read** data from Kafka for downstream processing
- On top there is a cluster of ZooKeeper instances that form a so called **ensemble**. We will talk more about the latter on the subsequent slides

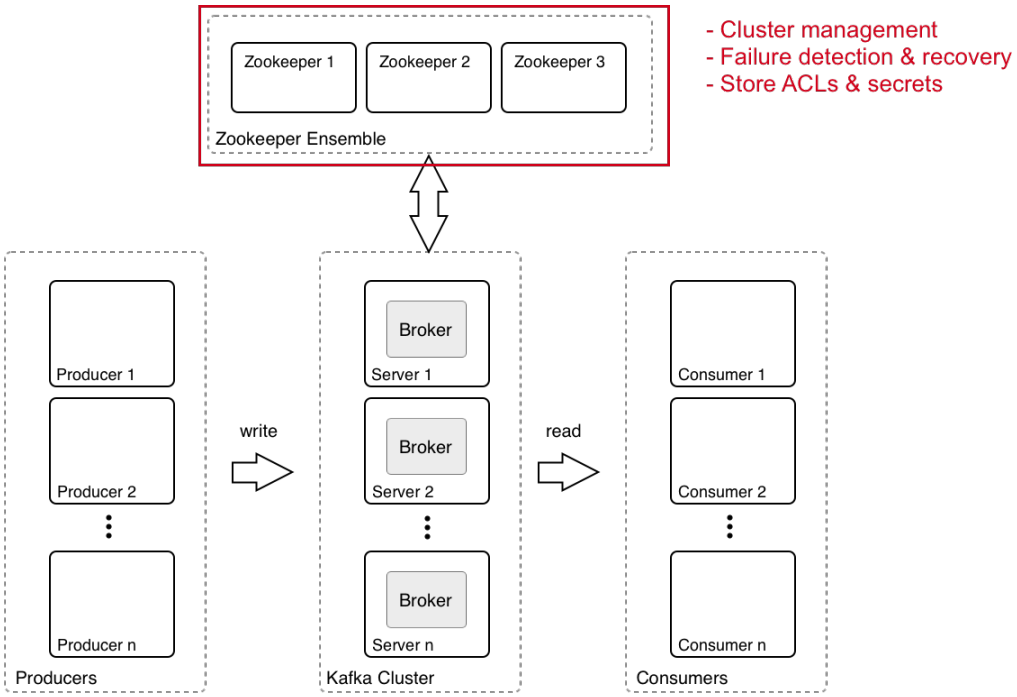
Decoupling Producers and Consumers

- Producers and Consumers are decoupled
- Slow Consumers do not affect Producers
- Add Consumers without affecting Producers
- Failure of Consumer does not affect System



- a key feature of Kafka is that Producers and Consumers are **decoupled**, that is,
  - they do not need to know about the existence of each other.
  - the internal logic of a producer does never depend on any of the downstream consumers
  - the internals of a consumer do not depend on the upstream producer
- Producers and Consumers simply need to agree on the data format of the records produced and consumed
- a slow Consumer will not affect Producers and
- more Consumers can be added without affecting Producers.
- failure of a Consumer will not affect the upstream system.

How Kafka Uses ZooKeeper



- Kafka Brokers use ZooKeeper for a number of important internal features such as
  - Cluster management
  - Failure detection and recovery (e.g. when a broker goes down)
  - To store Access Control Lists (ACLs) used for authorization in the Kafka cluster

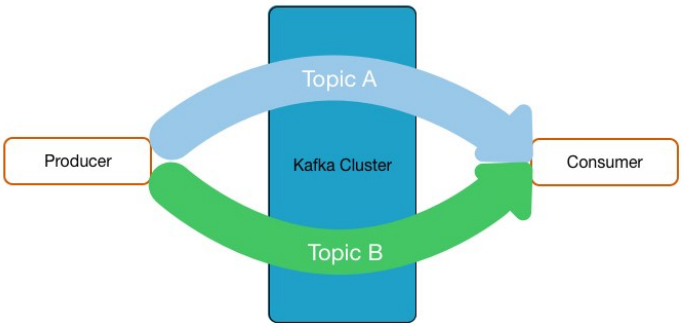
ZooKeeper Basics

- **Open Source** Apache Project
- Distributed **Key Value Store**
- Maintains **configuration information**
- Stores **ACLs** and **Secrets**
- Enables highly reliable **distributed coordination**
- Provides **distributed synchronization**
- 3 or 5 servers form an **ensemble**

- ZooKeeper is a centralized service used by many distributed applications or infrastructure such as Kafka, Akka, Neo4j and Mesos (https://cwiki.apache.org/confluence/display/ZOOKEEPER/PoweredBy)
- For **resiliency** ZK is typically run in a cluster of 3 or 5 servers, called an **ensemble**
- The number needs to be odd due to the **ZAB** consensus algorithm used to achieve a quorum
- It is not recommended to run more than 5 instances in an ensemble for performance reasons (ZK instances need to communicate synchronously with each other to achieve a quorum)
- ACL = Access Control List

Topics

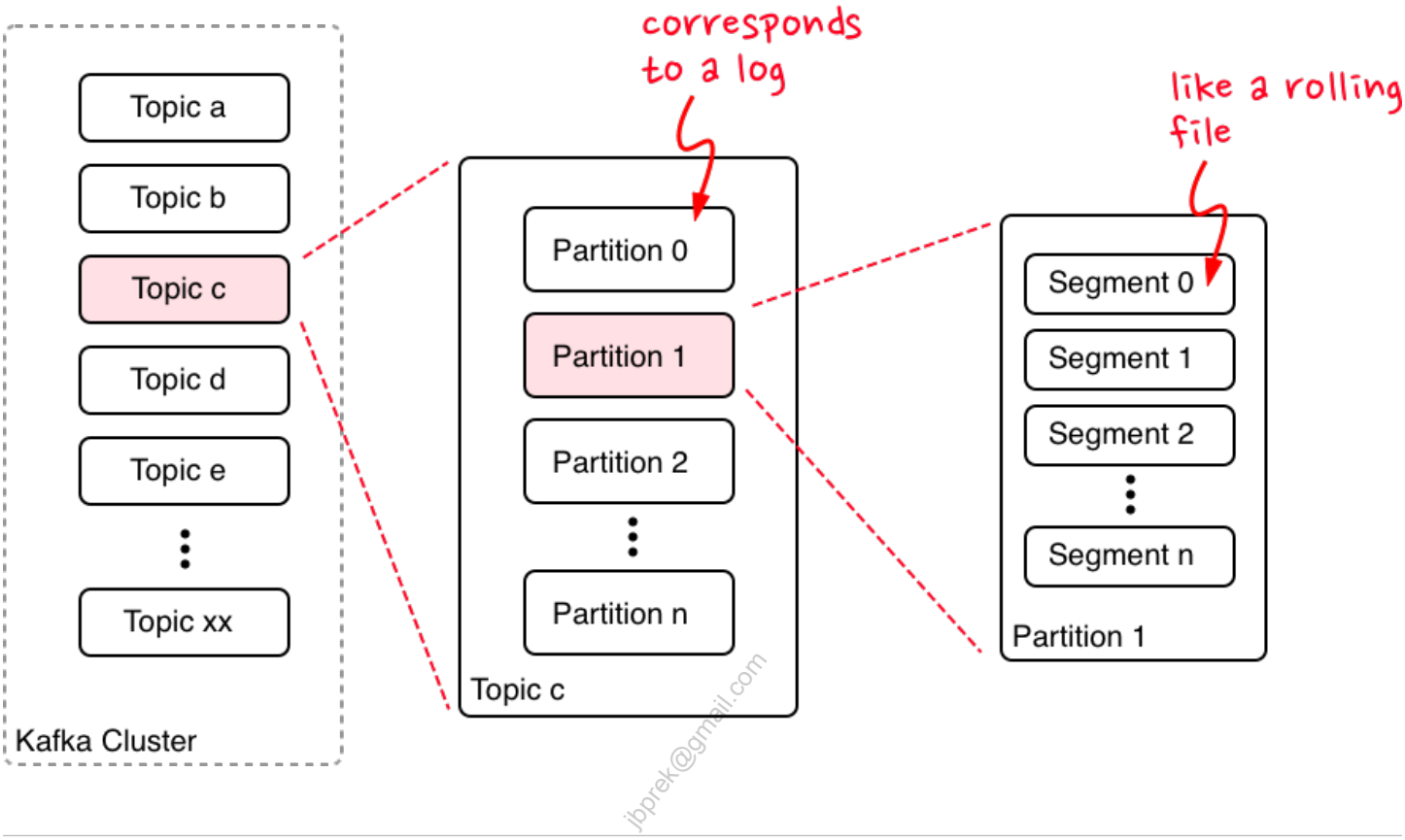
- **Topics**: Streams of "related" Messages in Kafka
  - Is a **Logical Representation**
  - **Categorizes Messages** into Groups
- Developers define Topics
- Producer ←→ Topic: N to N Relation
- Unlimited Number of Topics



The Topic is a logical representation that spans across Producers, Brokers, and Consumers

- Developers decide which Topics exist
  - By default, a Topic is auto-created when it is first used
- One or more Producers can write to one or more Topics
- There is no limit to the number of Topics that can be used

Topics, Partitions and Segments



Let's now transfer what we have learned about the log to Kafka. In Kafka we have the three terms **Topic**, **Partition** and **Segment**. Let's review each of them:

- **Topic**: A topic comprises all messages of a given category. We could e.g. have the topic "temperature_readings" which would contain all messages that contain a temperature reading from one of the many measurement stations the company has around the globe.
- **Partition**: To parallelize work and thus increase the throughput Kafka can split a single topic into many partitions. The messages of the topic will then be split between the partitions. The default algorithm used to decide to which parition a message goes uses the hash code of the message key. A partition is handled in its entirety by a single Kafka broker. A partition can be viewed as a "log".

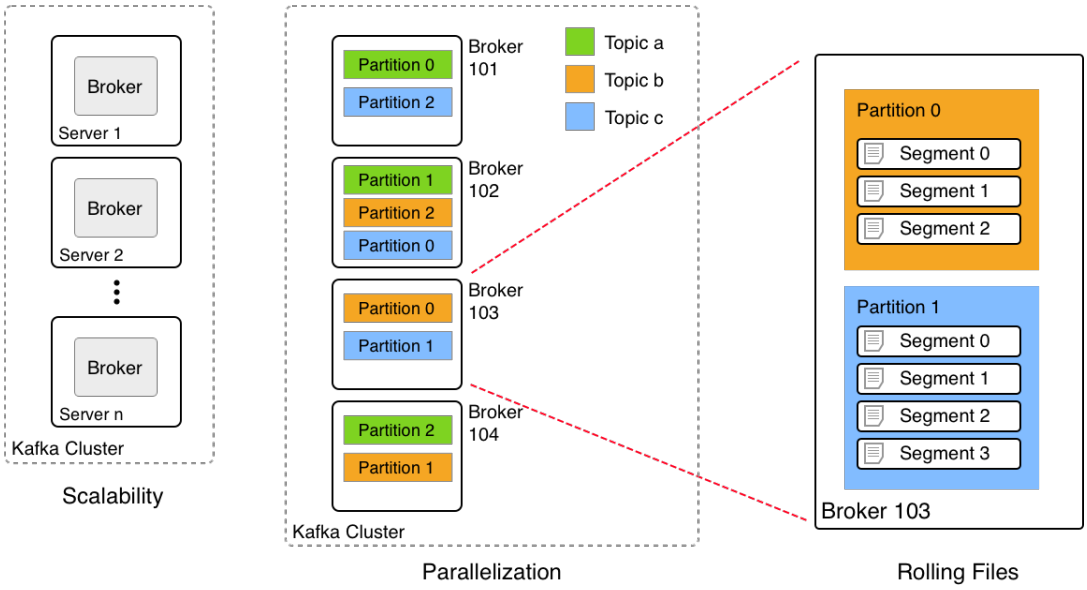
Topics, Partitions and Segments

- **Segment**: The broker stores the messages as they come in memory (page cache), then periodically flushes them to a physical file. Since the data can potentially be endless the broker is using a "rolling-file" strategy. It creates/allocates a new file and fills it with

messages. When the segment is full (or the given max. time per segment expires), the next one is allocated by the broker. Kafka rolls over files in segments to make it easier to manage **data retention** and **remove old data**

On the slide we annotate the segment and say that it "corresponds to a log". The **Log** will be introduced on a subsequent slide...

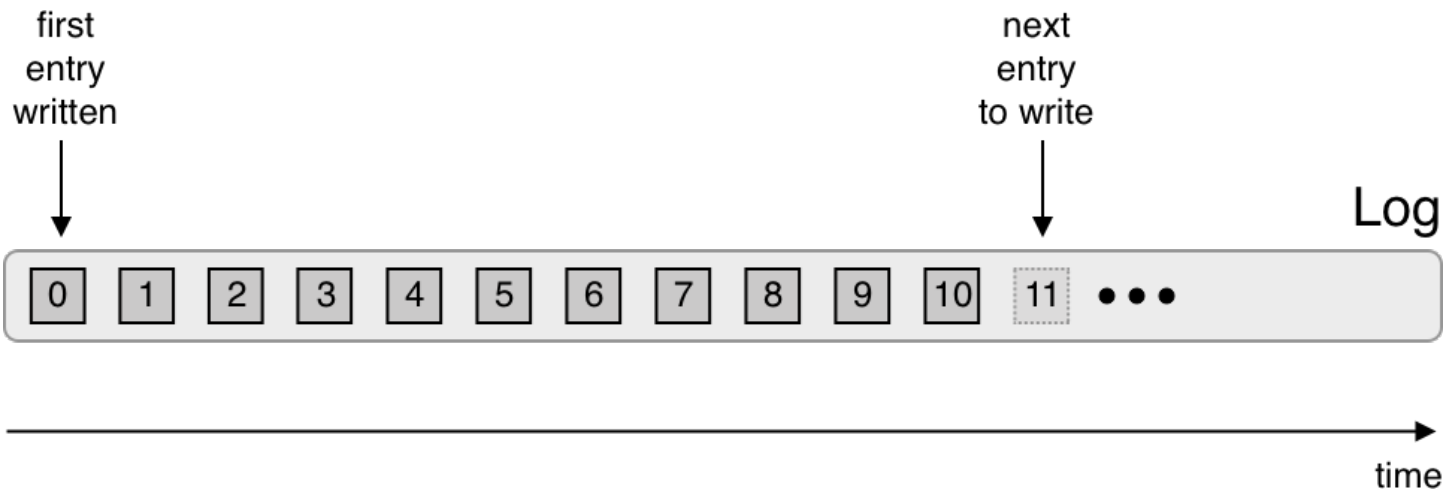
Topics, Partitions and Segments



Normally one uses several Kafka brokers that form a cluster to scale out. If we have now three topics (that is 3 categories of messages) then they may be organized physically as shown in the graphic.

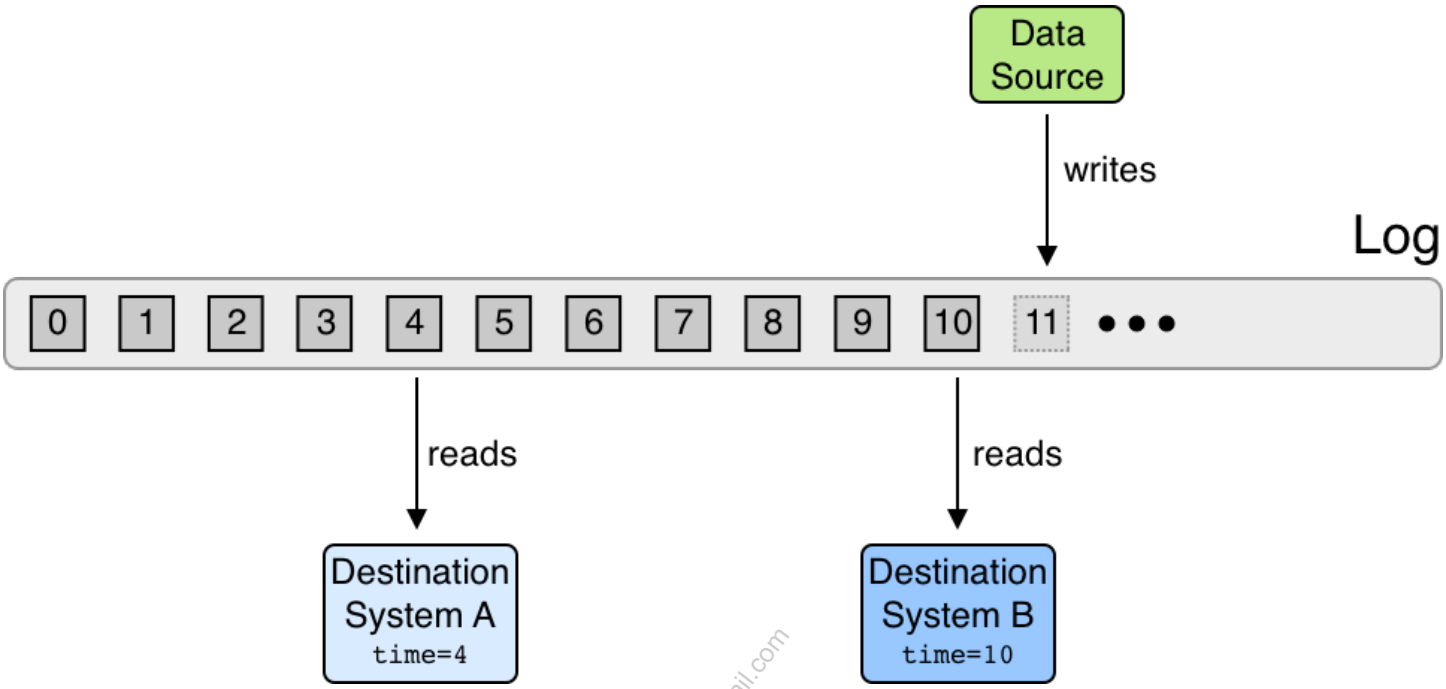
- Partitions of each topic are distributed among the brokers
- Each partition on a given broker results in one to many physical files called segments
- Segments are treated in a rolling file strategy. Kafka opens/allocates a file on disk and then fills that file sequentially and in append only mode until it is full, where "full" depends on the defined max size of the file, (or the defined max time per segment is expired). Subsequently a new segment is created.

The Log



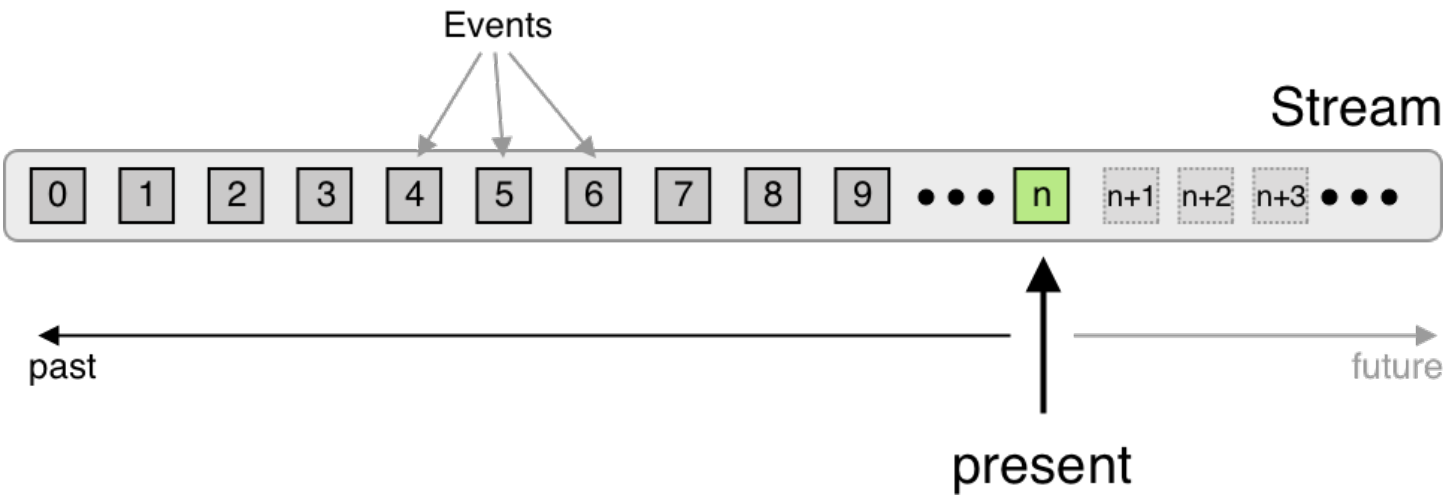
- To understand Kafka as a streaming platform it is important that we first discuss a few basics. One central element that enables Kafka and stream processing is the so called "log".
- A log is a data structure that is like a queue of elements. New elements are always appended at the end of the log and once written they are never changed. In this regard one talks of an append only, write once data structure.
- Elements that are added to the log are strictly ordered in time. The first element added to the log is older than the second one which in turn is older than the third one.
- In the image the time axis reflects that fact. The offset of the elements in that sense can be viewed as a time scale.

Log Structured Data Flow



- The log is produced by some data source. In the image the data source has already produced elements with offset 0 to 10 and the next element the source produces will be written at offset 11.
- The data source can write at its own speed since it is totally decoupled from any of the destination systems. In fact, the source system does not know anything about the consuming applications.
- Multiple destination systems can independently consume from the log, each at its own speed. In the sample we have two consumers that read from different positions at the same time. This is totally fine and an expected behavior.
- Each destination system consumes the elements from the log in temporal order, that is from left to right in our image.

The Stream



In this course we're often going to hear about **streaming data**. Let's thus give a very simple definition of what a **stream** is:

As show in the image, a stream is a sequence of events. The sequence has a beginning somewhere in the past. The first event has offset `0`. "Past" in this context can mean anything from seconds to hours to weeks or even longer periods.

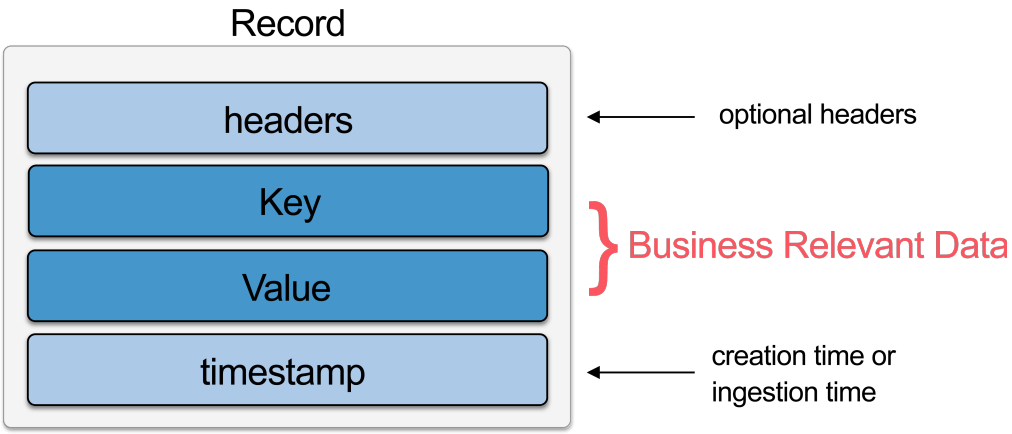
In the image we have denoted the event with offset `n` to be "now". Everything that comes after event `n` is considered the "future".

A stream is open ended. There is not necessarily an end-time when streaming of events stops (although there can be one of course).

Given the fact that a stream is open ended we cannot know how much data to expect in the future and we can certainly not wait until all data has arrived before we start doing something with the stream.

Also very important to note is that a stream is "immutable". In stream processing one never modifies an existing stream but always generates a new output stream.

Data Elements



A data element in a log (or topic; to be introduced later) is called a **record** in the Kafka world. Often we also use equivalent words for a record. The most common ones are **message** and **event**.

A record in Kafka consists of **Metadata** and a **Body**.

- The metadata contains offset, compression, magic byte, timestamp and an optional **headers** collection of 0 to many key value pairs.
- The body consists of a **Key** and a **Value** part
- The value part is usually containing the **business relevant** data
- The key by default is used to decide into which partition a record is written to. As a consequence all records with identical an key go into the same partition. This is important in the downstream processing, since ordering is (only) guaranteed on a partition and **not** on a topic level!

There is also a timestamp in the message but we will talk later about the concept of time...

From the perspective of Kafka (Brokers) it doesn't really matter what is in the key and value. Any data type is possible that can be serialized by the producers. The broker only sees arrays of bytes.

Brokers Manage Partitions

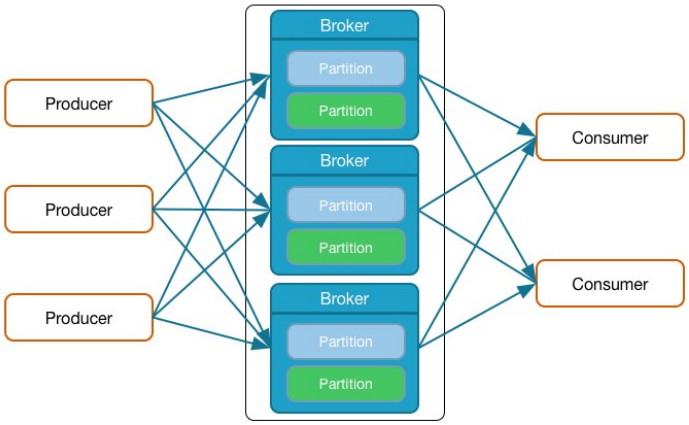
- Messages of Topic spread across Partitions
- Partitions spread across Brokers
- Each Broker handles many Partitions
- Each Partition stored on Broker's disk
- Partition: 1..n `log` files
- Each message in Log identified by *Offset*
- Configurable Retention Policy

- Messages in a Topic are spread across Partitions in different Brokers
- Typically, a Broker will handle many Partitions
- Each Partition is stored on the Broker's disk as one or more `log` files
- Each message in the log is identified by its *offset* which is a monotonically increasing value
- Kafka provides a configurable retention policy for messages to manage log file growth

Brokers share metadata with Producers and Consumer, *e.g.* mapping of partitions to Brokers, ISR information, where the leaders are, etc.

Broker Basics

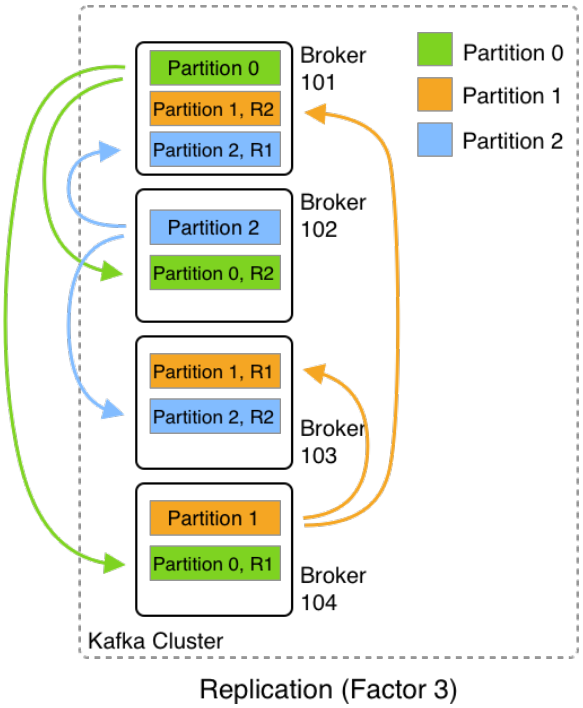
- Producer sends Messages to Brokers
- Brokers receive and store Messages
- A Kafka Cluster can have many Brokers
- Each Broker manages multiple Partitions



A typical Kafka cluster has many brokers for high availability and scalability reasons. Each broker handles many partitions from either the same topic (if the number of partitions is bigger than the number of brokers) or from different topics.

Each broker can handle hundreds of thousands, or millions, of messages per second

Broker Replication



Replication (Factor 3)

Kafka can **replicate** partitions across a configurable number of Kafka servers which is used for **fault tolerance**. Each partition has a leader server and zero or more follower servers. Leaders handle all read and write requests for a partition.

In this image we have four brokers and three replicated partitions. The replication factor is also 3. For each partition a different broker is the leader, for **optimal resource usage**.

When we say `replication-factor=3` then that includes the leader partition. Thus in this case we have the leader and 2 followers.

Producer Basics

- Producers write Data as Messages
- Can be written in any language
  - Native: Java, C/C++, Python, Go, .NET, JMS
  - More Languages by Community
  - REST Proxy for any unsupported Language
- Command Line Producer Tool

- Producers write data in the form of messages to the Kafka cluster
- Producers can be written in any language
  - Native Java, C/C++, Python, Go, .NET, JMS clients (for legacy or enterprise Java apps) are supported by Confluent
  - Clients for many other languages exist that are supported by the community
  - Confluent develops and supports a REST server which can be used by clients written in any language for which a native client does not exists
- A command-line Producer tool exists to send messages to the cluster which is useful for experimenting, testing and debugging.

The Confluent JMS Client is part of **Confluent Enterprise**

Load Balancing and Semantic Partitioning

- Producers use a Partitioning Strategy to assign each Message to a Partition
- Two Purposes:
  - Load Balancing
  - Semantic Partitioning
- Partitioning Strategy specified by Producer
  - Default Strategy: `hash(key) % number_of_partitions`
  - No Key → Round-Robin
- Custom Partitioner possible

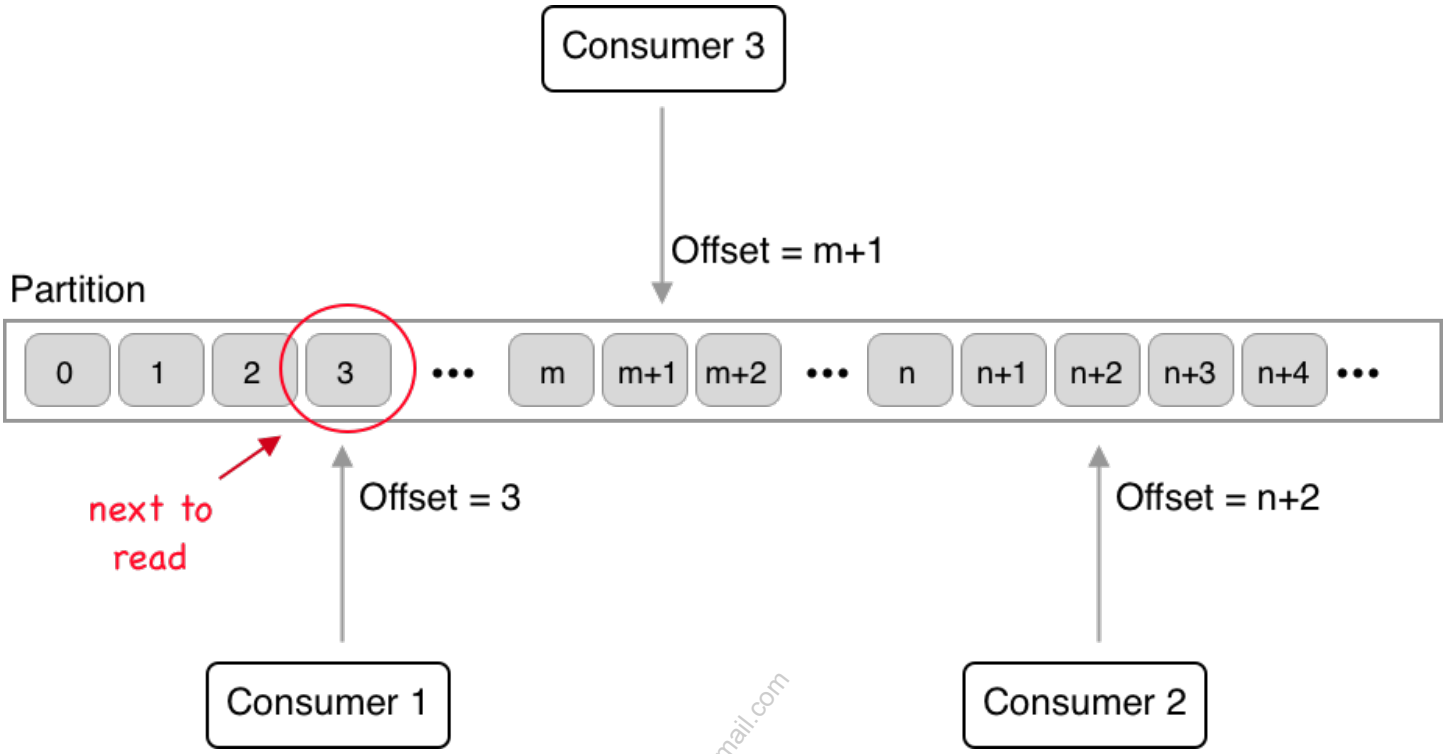
- Producers use a partitioning strategy to assign each message to a Partition
- Having a partitioning strategy serves two purposes
  - Load balancing: shares the load across the Brokers
  - Semantic partitioning: user-specified key allows locality-sensitive message processing
- The partitioning strategy is specified by the Producer
  - Default strategy is a hash of the message key
    - `hash(key) % number_of_partitions`
  - If a key is not specified, messages are sent to Partitions on a round-robin basis
- Developers can provide a custom partitioner class
- Load balancing: for example, round robin just to do random load balancing
- Semantic partitioning: for example, user-specified key is the user id, allows Consumers to make locality assumptions about their consumption. This style of partitioning is explicitly designed to allow locality-sensitive processing in Consumers.

Consumer Basics

- Consumers **pull** messages from 1...n topics
- New inflowing messages are automatically retrieved
- Consumer offset
  - keeps track of the last message read
  - is stored in special topic
- CLI tools exist to read from cluster

- A Consumers pulls messages from one or more Topics in the cluster. As messages are written to a Topic, the Consumer will automatically retrieve them
- The **Consumer Offset** keeps track of the latest message read and it is stored in a special Kafka Topic. If necessary, the Consumer Offset can be changed, for example, to reread messages
- A command-line Consumer tool exists to read messages from the Kafka cluster, which might be useful for experimenting, testing and debugging.

Consumer Offset



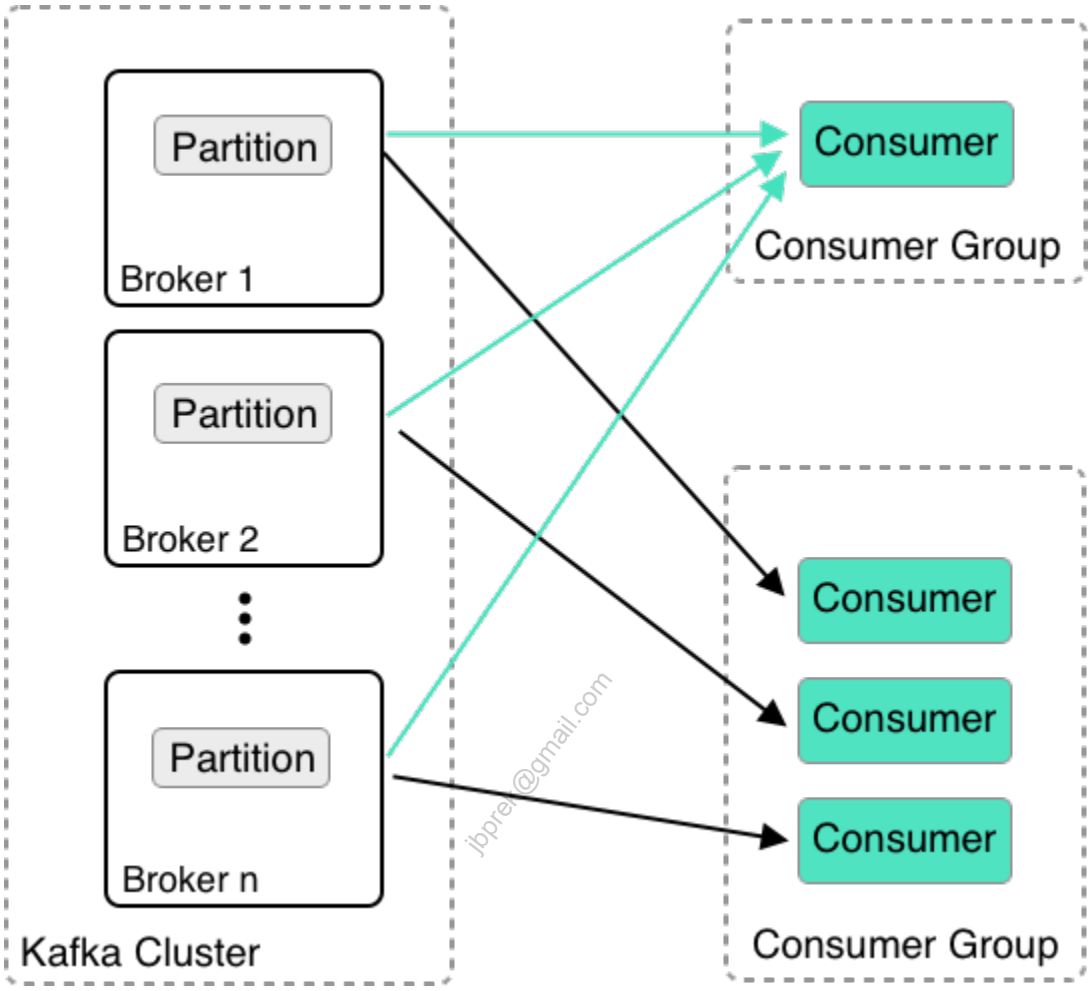
Kafka stores messages in topics for a pre-defined amount of time (by default 1 week), such as that many consumers (or more precisely consumer groups) can access the data. Most consumers will process the incoming data in near-real time, but others may consume the data at a later time.

On the slide we see 3 consumers that all poll data from the same partition of a given topic.

- consumer 1 will be reading from offset `3`
- consumer 2 is currently reading from offset `n+2`
- consumer 3 is reading from offset `m+1`

Consumers, by default, store their offsets in a special topic called `__consumer_offsets` in Kafka. The reason being that if a consumer crashes, another consumer can take its place, read the next to read offset from Kafka and proceed the task where the crashed consumer left off.

Distributed Consumption

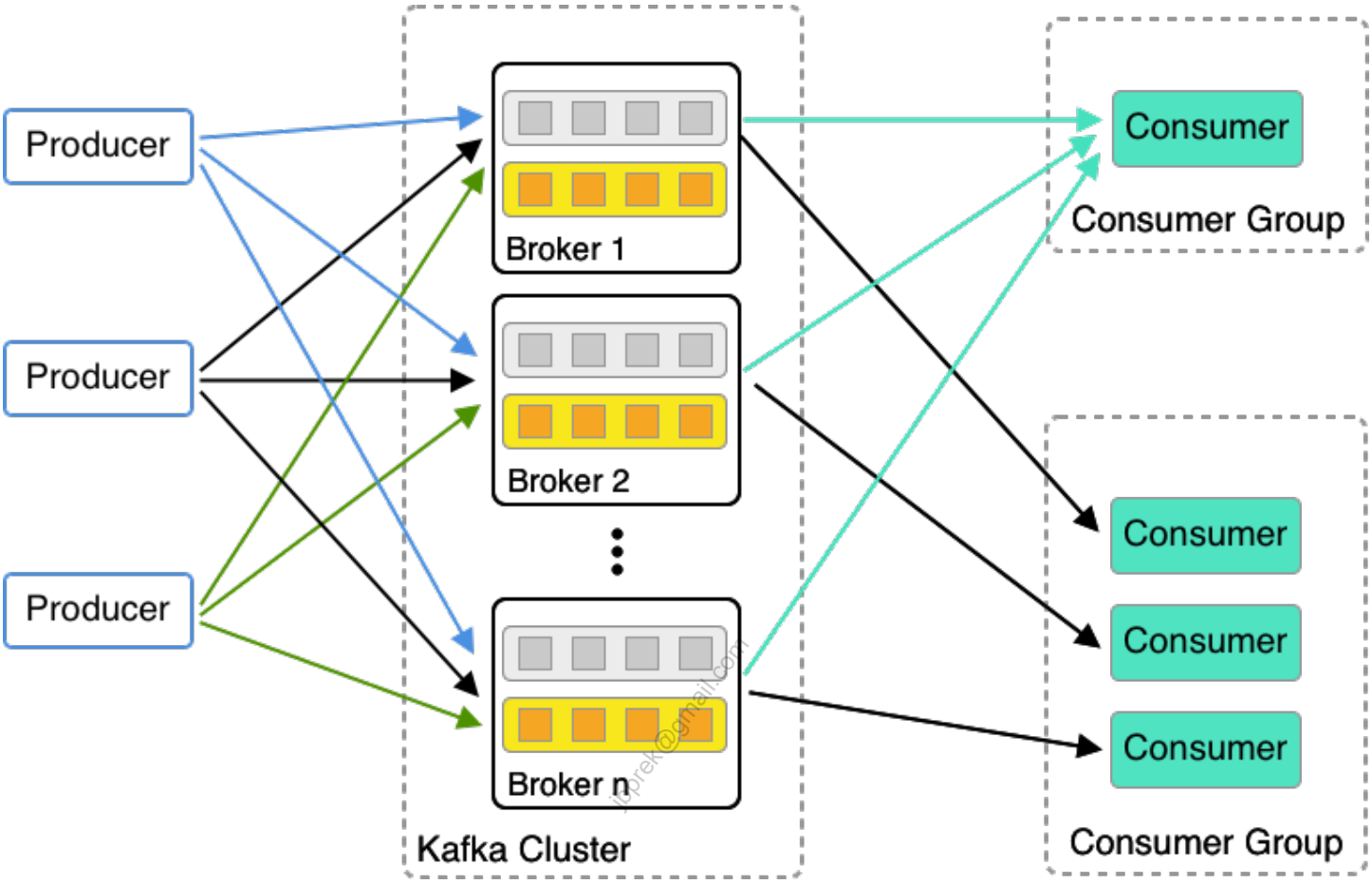


I mentioned consumer group as a concept in the previous slide already. To allow to increase the throughput in downstream consumption of data flowing into a topic, Kafka introduced **Consumer Groups**.

- A consumer group consists of 1 to many consumer instances
- All consumer instances in a consumer group are identical clones of each other
- To be transparently added to a consumer group an instance needs to use the same `group.id`
- A consumer group can scale out and thus parallelize work until the number of consumer instances is equal to the number of topic partitions

Each partition is consumed in its entirety by a single consumer of a given consumer group. On the other hand, each consumer instance can consume from multiple partitions.

Scalable Data Pipeline



Putting everything together we get a highly scalable system of data pipelines.

- Conceptually the Kafka Cluster can grow infinitely. The only limit is the failover time after a catastrophic failure. This limits the reasonable max size of a Kafka cluster to approximately 50 brokers with up to 4000 partitions each.
- Downstream work can also be parallelized infinitely by creating topics with many partitions and by running an equal number of consumers as topics in a consumer group. Here the limits of parallelism lies in the number of distinct keys (if using the default partitioner on the producers). It doesn't make sense to have more partitions in a topic than distinct keys in the messages that are sent to this topic.

Chapter Review

- A Kafka system is made up of Producers, Consumers, and Brokers
- ZooKeeper provides coordination Services for the Brokers
- Producers write messages to Topics
- Topics are broken down into Partitions for Scalability
- Consumers read Data from one or more Topics

Q&A

Questions:

- Why do we need an odd number of ZooKeeper nodes?
- How many Kafka brokers can a cluster maximally have?
- How many Kafka brokers do you minimally need for high availability?
- What is the criteria that two or more consumers form a consumer group?

Answers:

- ZooKeeper forms an ensemble which requires an odd number of nodes to be able to form a quorum
- Technically an unlimited number. Practically the number is limited by the dependency on ZooKeeper
- The minimum number of brokers for limited high availability is 2. Each partition can then be replicated once.
- The group ID of all consumer instances must be the same

Hands-On Lab

Refer to the lab **02 - Fundamentals of Apache Kafka** in the Exercise Book.

Further Reading

- Apache Kafka Quickstart: https://kafka.apache.org/quickstart
- Using Apache Kafka as a Scalable, Event-Driven Backbone for Service Architectures https://www.confluent.io/blog/apache-kafka-for-service-architectures/
- The Changing Face of ETL: https://www.confluent.io/blog/changing-face-etl
- How to choose the number of topics/partitions in a Kafka cluster?: https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster
- ZooKeeper `v3.6.3` Administrator's Guide: https://zookeeper.apache.org/doc/r3.6.3/zookeeperAdmin.html

---

These are are few links to material diving into the topics of this module in more depth.

04 How Kafka Works

Agenda

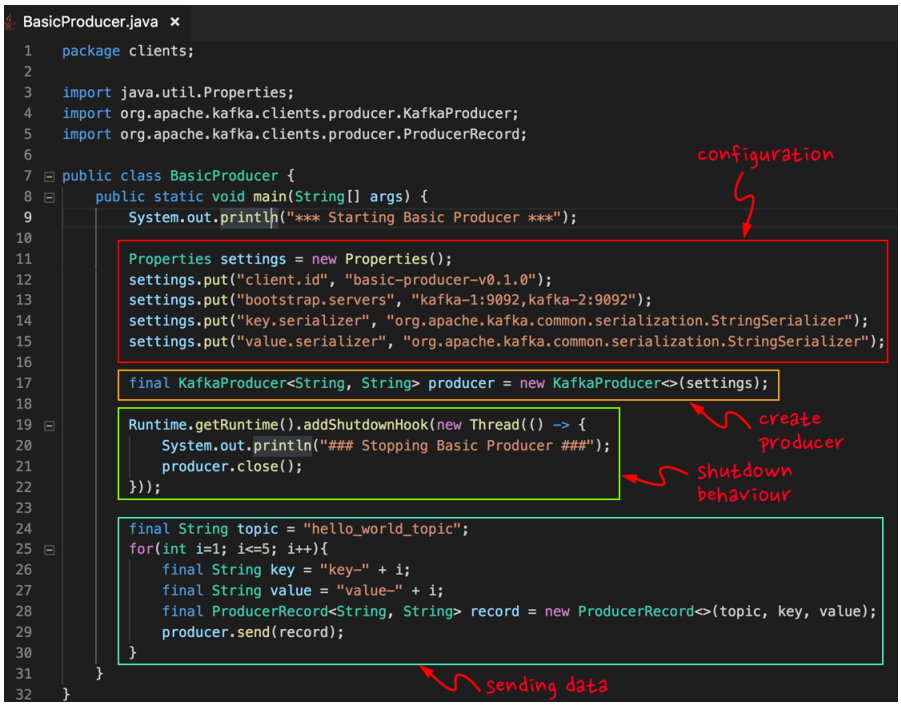
1. Introduction
2. Motivation & Customer Use Cases
3. Apache Kafka Fundamentals
4. How Kafka Works ...
5. Integrating Kafka into your Environment
6. The Confluent Platform
7. Conclusion

Learning Objectives

After this module you will be able to:

- Give a high level description of the programming logic in a Kafka consumer client
- Explain how EOS works to an interested lay person
- List the means with which Kafka provides durability and HA
- Illustrate on a high level, how you can secure your Kafka cluster

Development: A Basic Producer in Java




```java
package clients;

import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

public class BasicProducer {
    public static void main(String[] args) {
        System.out.println("*** Starting Basic Producer ***");

        Properties settings = new Properties();
        settings.put("client.id", "basic-producer-v0.1.0");
        settings.put("bootstrap.servers", "kafka-1:9092,kafka-2:9092");
        settings.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        settings.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        final KafkaProducer<String, String> producer = new KafkaProducer<>(settings);

        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            System.out.println("### Stopping Basic Producer ###");
            producer.close();
        }));

        final String topic = "hello_world_topic";
        for(int i=1; i<=5; i++){
            final String key = "key-" + i;
            final String value = "value-" + i;
            final ProducerRecord<String, String> record = new ProducerRecord<>(topic, key, value);
            producer.send(record);
        }
    }
}
```


This slide is **not about the code details**, but just wants to give a high level overview over a "hello-world" type producer written in Java!

When writing a basic producer in Java we can distinguish four parts:

- Configuration: this is the part where we define all non-default properties of our producer
- Constructor: Here we construct a producer object using the configuration values
- Shutdown: In this section we define how the application shall behave in case it receives a `SIG_TERM` or `SIG_KILL` signal
- Sending: In this part we define the logic that actually sends messages to the respective topic in the Kafka cluster

Development: A Basic Consumer in .NET/C#


```
namespace consumer_net {
    0 references
    class Program {
        0 references
        static void Main (string[] args) {
            Console.WriteLine ("Starting Consumer!");
            var config = new Dictionary<string, object> {
                { "group.id", "dotnet-consumer-group" },
                { "bootstrap.servers", "kafka-1:9092" },
                { "auto.commit.interval.ms", 5000 },
                { "auto.offset.reset", "earliest" }
            };

            var deserializer = new StringDeserializer (Encoding.UTF8);
            using (var consumer = new Consumer<string, string> (config, deserializer, deserializer)) {
                consumer.OnMessage += (_, msg) =>
                  Console.WriteLine ($"Read ('{msg.Key}', '{msg.Value}') from: {msg.TopicPartitionOffset}");

                consumer.OnError += (_, error) =>
                  Console.WriteLine ($"Error: {error}");

                consumer.OnConsumeError += (_, msg) =>
                  Console.WriteLine ($"Consume error ({msg.TopicPartitionOffset}): {msg.Error}");

                consumer.Subscribe ("hello_world_topic");

                while (true) {
                    consumer.Poll (TimeSpan.FromMilliseconds (100));
                }
            }
        }
    }
}
```

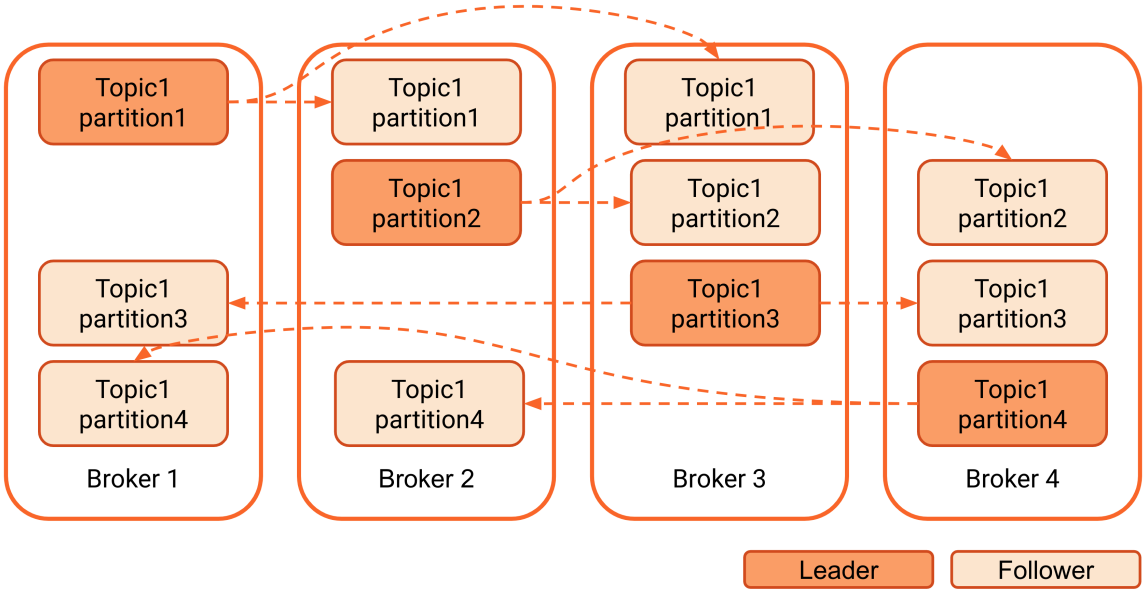
This slide is **not about the code details**, but just wants to give a high level overview over a "hello-world" type consumer written in .NET/C#

When writing a basic consumer (here in C#, .NET) we can distinguish the following parts:

- **Configuration**: In this section we define the non-default values of the various configuration parameters used when creating a consumer object.
- **Message Callback:** In this callback, which is triggered for each message we define what shall happen with the particular message.
- **Error Callbacks:** Whenever an error during the handling of a message or an unexpected error happens, one of these callback methods is called. Here we just report the exception to `STDOUT`.
- **Subscription:** Here the consumer subscribes to the desired topic(s)
- **Polling:** Here we define how the consumer shall be polling. In this sample in an endless loop with a 100 ms wait time between polls.

If you're not familiar with C# please note that the `using(…)` statement makes sure that the consumer (resource) is disposed orderly even in case of an expected or unexpected killing of the application. Thus we avoid leaving any orphaned resources behind us.

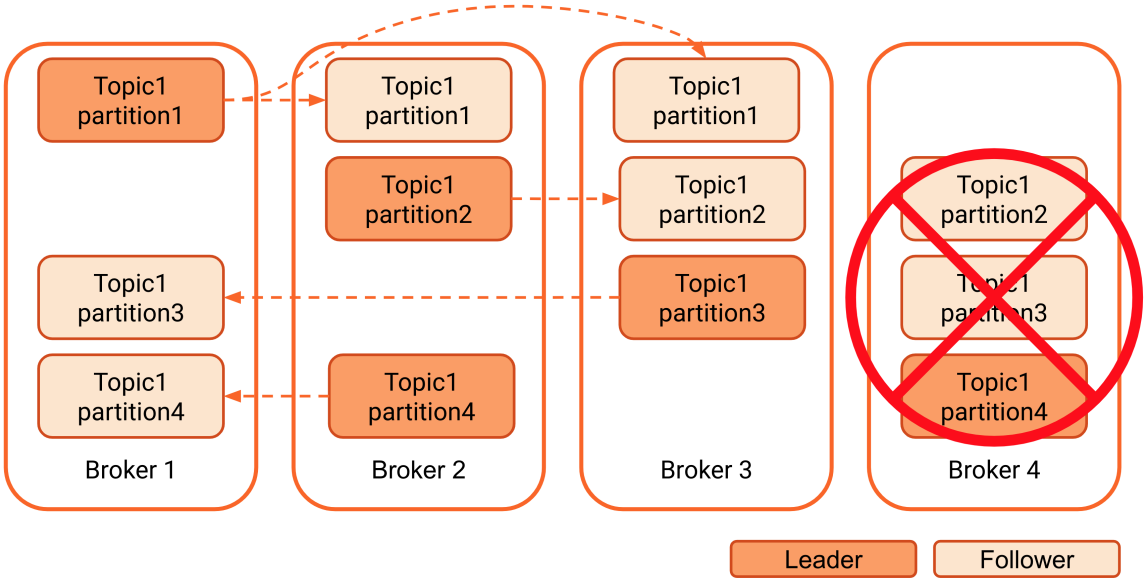
Partition Leadership & Replication



In this sample we see that:

- we have a single topic called `topic1`
- the topic has 4 partitions 1 to 4
- each partition is replicated 3 times
- each partition has a leader and 2 followers
- Kafka clients, e.g. producers and consumers only access the leader of each topic
- Kafka does its best to equally distribute the work load among available brokers, thus in this case each broker is leader for one partition
- Followers poll the leader periodically for new data and then write it to their own local commit log

Partition Leadership & Replication



On this slide we see what happens if broker 4 fails.

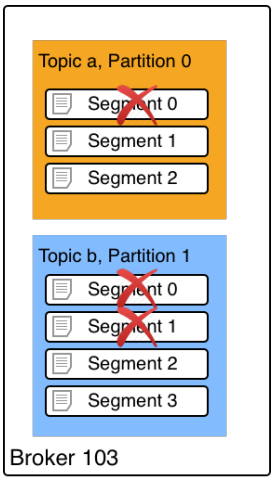
- The controller of the Kafka cluster reassigns the partition leadership for topic1/partition4 to one of the remaining replicas. In this case to broker 2
- All partitions that had replicas on Broker 4 will now be under-replicated
- If the number of replicas is lower than the minimal requested number of ISRs (in-sync replicas), producers will not be able to write to this topic anymore
- Consumers will now consume from broker 2 when accessing topic1/partition4

	The above happens totally transparently to the users. Note that if broker 4 is brought up again and added to the cluster, the leadership for partition 4 might go back to it eventually (after a few minutes), since Kafka tries to balance the leadership across all available brokers

Data Retention Policy

How long do I want or can I store my data?

- How long (default: `1 week`)
- Set **globally** or **per topic**
- Business decision
- Cost factor
- Compliance factor → GDPR



Retention Policy

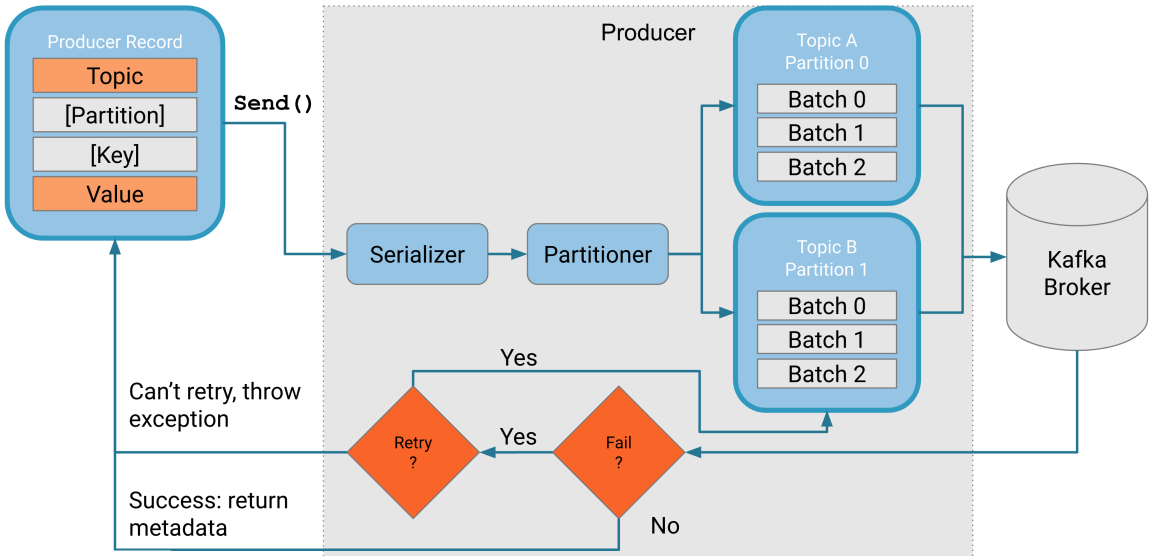
	Data purged per segment

- Contrary to a normal enterprise service bus such as IBM MQ Series or Rabbit MQ, Kafka stores messages for some time
- The time **how long** messages are to be stored in their respective topics is called **retention time**
- By **default** the retention time is set to **1 week**, but it can be changed from 0 to infinite
- **Retention time** can be set globally or **per topic**
- **How long** you want to store your data is a **business decision** and an important cost factor. It is also a compliance question, e.g. GDPR determines how long (sensitive) customer data can be maximally stored

	The data is always purged per segment. When all the messages in a segment are older than the retention time then the segment is deleted.

	The exception here are **compacted topics**. Log/topic compaction means that Kafka will keep the latest version of a record (with a given key) and delete the older versions during a log compaction.

Producer Design

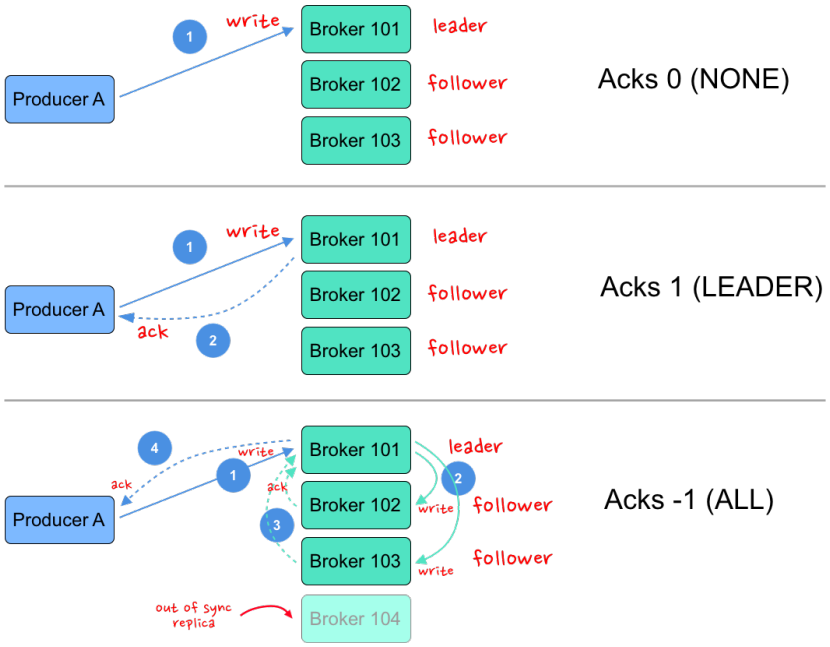


On this slide we dive a bit more into the high level internals of a producer. No worries, we will remain pretty high level.

- The area with the grey underlay represents the code of the Kafka Client library - to be more precise, of it's producer part. Thus this is not the code that you write, but that is available to you with the Kafka client library
- On the left upper side you can see a single record. It contains elements such as the record key and value as well as other meta data such as topic name and partition number
- Your application code will `send()` the message to Kafka
- First your record is serialized into an array of bytes, using the pre-configured serializer
- Then the record is passing through a partitioner. By default the partitioner looks at the (serialized) key of the record and decides to which partition of the topic this message will be sent
- No often messages are not sent directly to the respective broker but first batched. This depends on some settings the developer can define in their code
- Once a batch is "full" the producer library code flushes the batch to the respective Kafka broker
- The broker tries to store the batch in its local commit log
- If the broker was successful it will answer with an ACK and some additional meta data. All is good in this case!

- If the broker encounters a failure while trying to save the batch, it will respond with a NACK. The producer then automatically tries to resend the same batch of messages again, until it succeeds
- If the sending is not successful and the number of retries have been exhausted, then the producer triggers an exception which needs to be handled by your code

Producer Guarantees

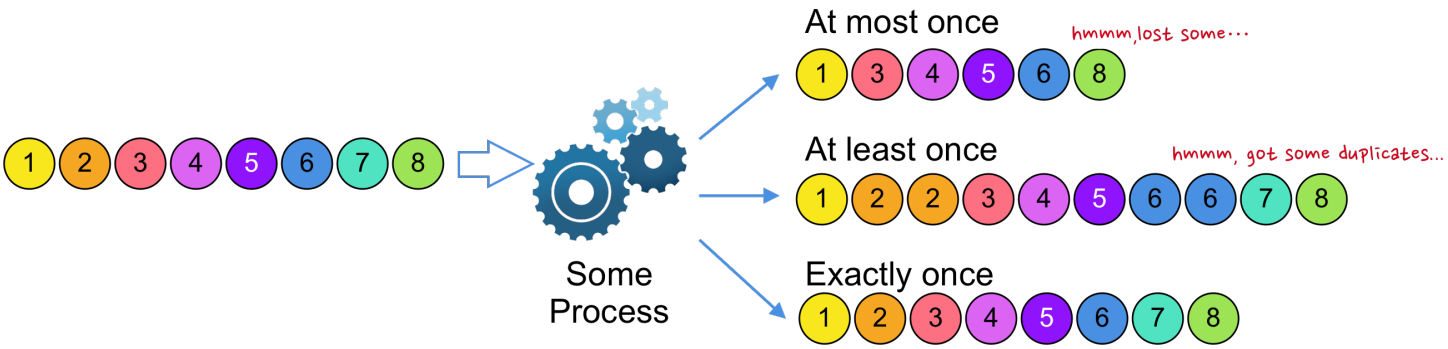


When using a producer, you can configure its acks (Acknowledgments) which default to `all`. The `acks` config setting is the write-acknowledgment received count required from partition leader before the producer write request is deemed complete. This setting controls the producer's durability which can be very strong (`all`) or none. Durability is a tradeoff between throughput and consistency. The acks setting is set to "all" (`-1`), "none" (`0`), or "leader" (`1`).

- **Acks 0 (NONE):** The `acks=0` is none meaning the Producer does not wait for any ack from Kafka broker at all. The records added to the socket buffer are considered sent. There are no guarantees of durability. The record offset returned from the send method is set to -1 (unknown). There could be record loss if the leader is down. There could be use cases that need to maximize throughput over durability, for example, log aggregation.
- **Acks 1 (LEADER):** The `acks=1` is leader acknowledgment. The means that the Kafka broker acknowledges that the partition leader wrote the record to its local log but responds without the partition followers confirming the write. If leader fails right after sending ack, the record could be lost as the followers might not have replicated the record yet. Record loss is rare but possible, and you might only see this used if a rarely missed record is not statistically significant, log aggregation, a collection of data for machine learning or dashboards, etc.
- **Acks -1 (ALL):** The `acks=all` or `acks=-1` is all acknowledgment which means the leader gets write confirmation from the full set of ISRs before sending an ack back to the producer. This guarantees that a record is not lost as long as one ISR remains alive. This `ack=all` setting is the strongest available guarantee that Kafka provides for durability.

This setting is even stronger with broker setting `min.insync.replicas` which specifies the minimum number of ISRs that must acknowledge a write. Most use cases will use `acks=all` and set a `min.insync.replicas` > `1`.

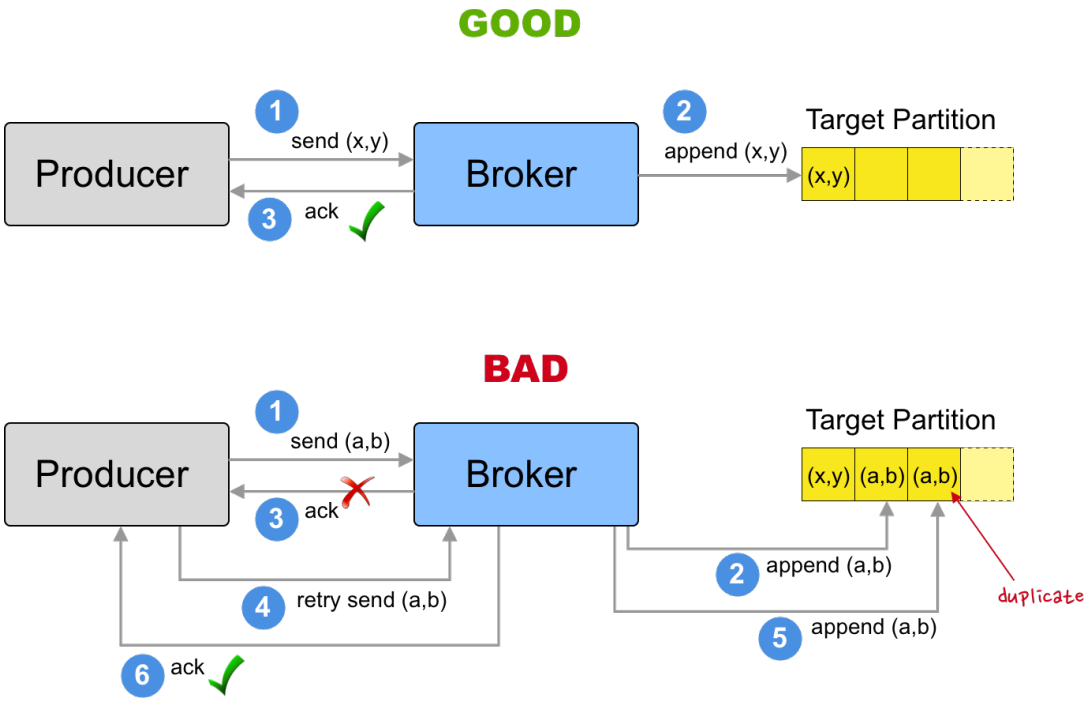
Delivery Guarantees



Kafka supports 3 delivery guarantees:

- **At most once:** From all records written to Kafka it is guaranteed that there will never be a duplicate. Under certain bad circumstances some record may be lost
- **At least once:** From all records written to Kafka none is ever lost. Under certain bad situations there could be duplicates in the log
- **Exactly once:** Every single record written to Kafka will be found in the Kafka logs exactly once. There is no situation where either a record is lost or where a record is duplicated

Idempotent Producers



To achieve exactly once semantics (EOS) the first step is to establish **idempotent** producers. Idempotence in producers is enabled by default (`enable.idempotence = true`) from CP 7.0 / AK 3.0.

On the slide I have shown what happens if a producer is not idempotent. Upon failure one gets duplicates in the commit log.

But, an idempotent producer guarantees, in collaboration with the respective broker, that:

- all messages written to a specific partition are maintaining their relative order in the commit log of the broker
- each message is only written once. No duplicates ever are to be found in the commit log
- together with `acks=all` we also make sure that no message is ever lost

Exactly Once Semantics

What?

- Strong **transactional guarantees** for Kafka
- Prevents clients from processing duplicate messages
- Handles failures gracefully

Use Cases

- Tracking ad views
- Processing financial transactions
- Stream processing

For the longest time it seemed an impossibility to have transactional guarantees in a highly distributed and asynchronous system such as Kafka.

But EOS gives us exactly this behavior.

Exactly Once Semantics (EOS) bring strong transactional guarantees to Kafka, preventing duplicate messages from being processed by client applications, even in the event of client retries and Broker failures

Use cases:

- tracking ad views,
- processing financial transactions,
- stream processing with e.g. aggregates only really makes sense with EOS

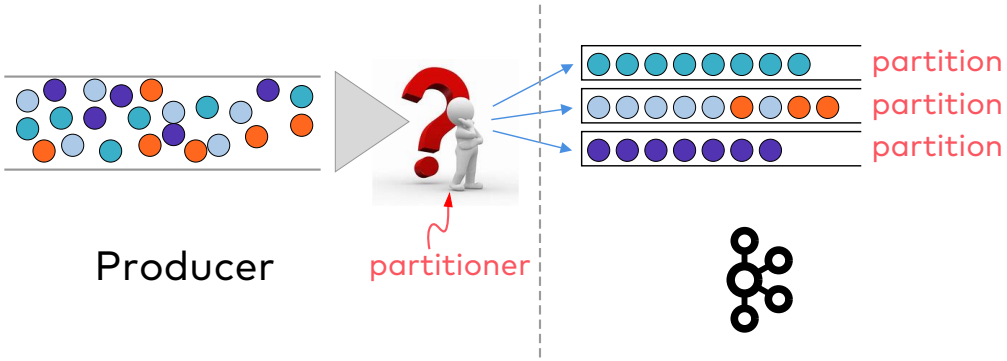
Process:

- A producer can start a transaction
- The producer then writes several records to multiple partitions on different brokers
- The producer can then commit the transaction
- If the TX succeeds, then the producer has the guarantee, that all records have been written exactly once and maintaining the local ordering to the Kafka brokers
- If the TX fails, then the producer knows that none of the record written will be showing up in the downstream consumers. That is, the aborted TX will leave no unwanted side effects
- NOTE: To have this downstream guarantee the consumers need to set their reading behavior to **read committed**

Partition Strategies

Why partitioning?

- Consumers need to **aggregate** or **join** by some key
- Consumers need **ordering guarantee**
- Concentrate data for **storage efficiency** and/or **indexing**



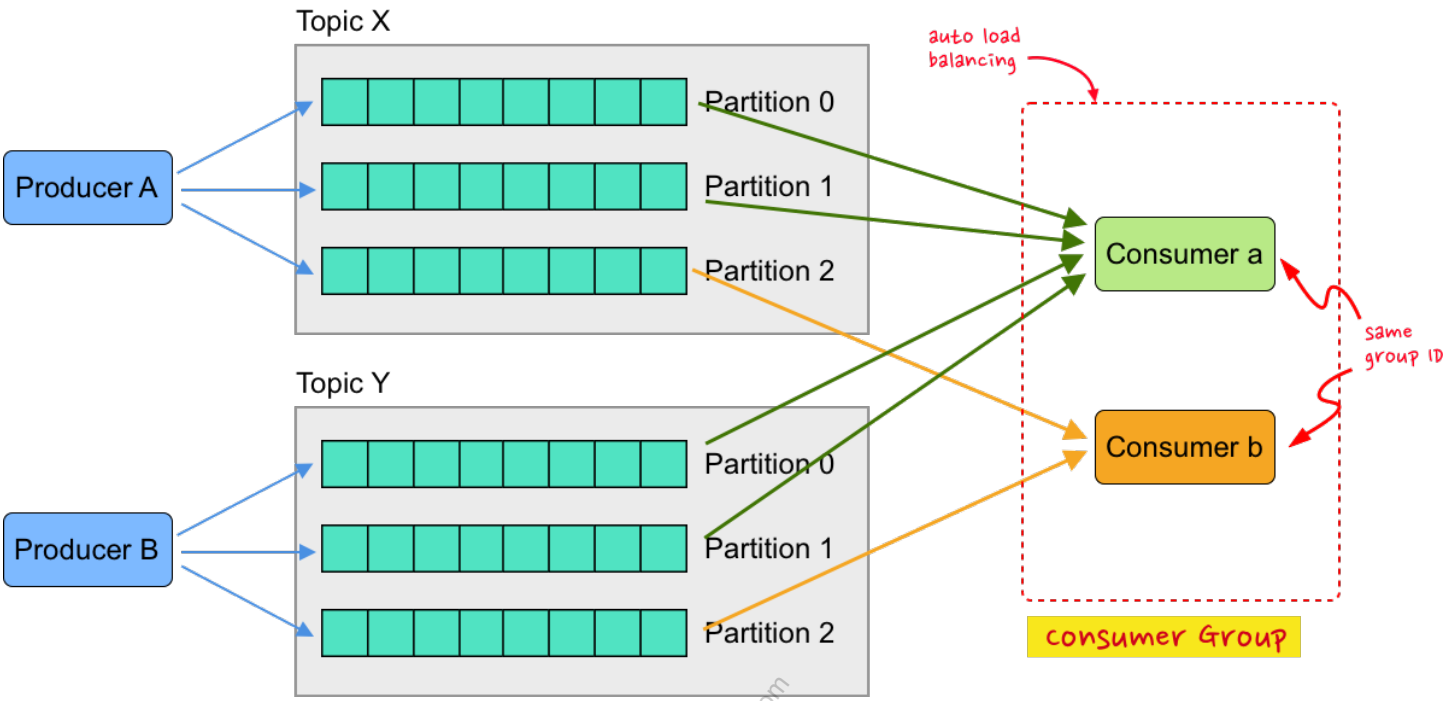
If you have enough load that you need more than a single instance of your application, you need to partition your data. The producer clients decide which topic partition data ends up in, but it's what the consumer applications will do with that data that drives the decision logic. If possible, the best partitioning strategy to use is random.

- The default partitioner of the Kafka client library is hashing the key of the message and taking the hash code (modulo number of partitions) as the partition number (`hash(key) % number_of_partitions`).
- If a key is not specified, messages are sent to partitions on a round-robin basis
- The partitioner is pluggable and thus we are free to implement a custom partitioner that uses any scenario specific algorithm to partition the data, e.g. any field of the value object.

	The number of partitions is specified per topic, but partitioning strategy is a **per-producer setting**. For example, a producer could use the same partitioning strategy across multiple topics using the same key.

	The guarantee of order from key-based allocation only applies if all messages with the same key are sent by the same producer.

Consumer Groups



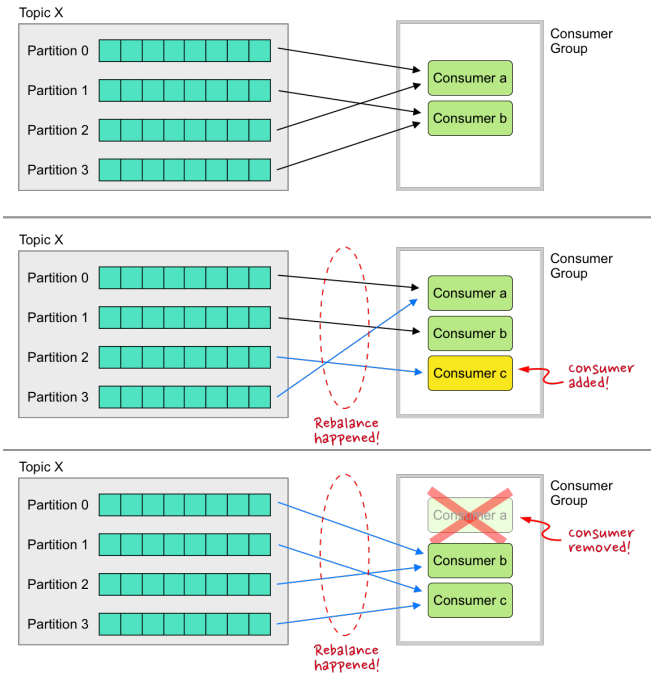
Kafka Topics allow the same message to be consumed multiple times by different Consumer Groups.

A Consumer Group binds together multiple consumers for parallelism. Members of a Consumer Group are subscribed to the same topic(s). The partitions will be divided among the members of the Consumer Group to allow the partitions to be consumed in parallel. This is useful when processing of the consumed data is CPU intensive or the individual Consumers are network-bound.

Within a Consumer Group, a Consumer can consume from multiple Partitions, but a Partition is only assigned to one Consumer to prevent repeated data.

Consumer Groups have built-in logic which triggers partition assignment on certain events, e.g., Consumers joining or leaving the group.

Consumer Rebalances



- Automatic rebalancing of partitions in a consumer group is triggered when either a new consumer is added to the group or a consumer is removed from a group, e.g. to scale down or because it crashes.
- The consumption is **stopped** during the rebalancing
- No data is lost during a rebalancing
- The partitions are automatically assigned (by the consumer group protocol) to the consumers using the specified strategy.
- There exist multiple partition assignment strategies:

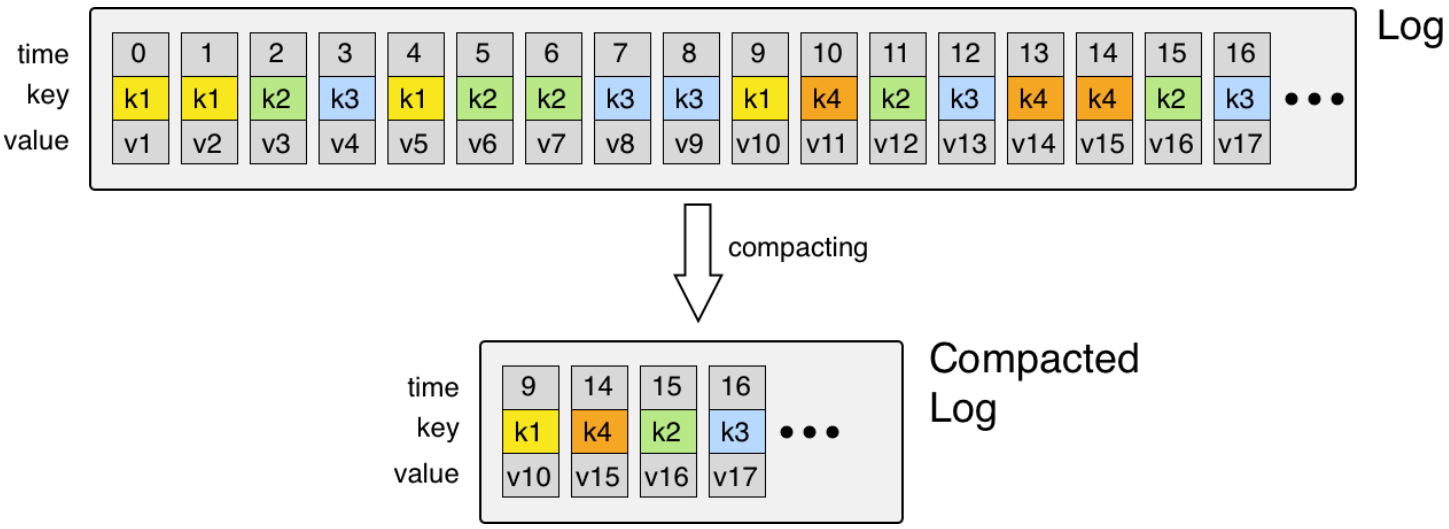
Partition assignment strategies are:

- **Range (default):** In Range, the Partition assignment assigns matching partitions to the same Consumer. The Range strategy is useful for "co-partitioning", which is particularly useful for Topics with keyed messages. Imagine that these two Topics are using the same key - for example, a userid.
- **Range (continued):** Topic A is tracking search results for specific user IDs; Topic B is tracking search clicks for the same set of user IDs. By using the same user IDs for the key in both Topics, messages with the same key would land in the same numbered Partition in both Topics (assuming both topics had the same number of Partitions) and so will land in the same Consumer.

- **Round robin:** The RoundRobin strategy is much simpler. Partitions are assigned one at a time to Consumers in a rotating fashion until all the Partitions have been assigned. This provides much more balanced loading of Consumers than Range.
- **Sticky:** An important note about Range and RoundRobin: Neither strategy guarantees that Consumers will retain the same Partitions after a reassignment. In other words, if a Consumer 1 is assigned to Partition A-0 right now, Partition A-0 may be assigned to another Consumer if a reassignment were to happen. Most Consumer applications are not locality-dependent enough to require that Consumer-Partition assignments be static.

  If the application requires Partition assignments to be preserved across reassignments, we can use the Sticky strategy. Sticky is RoundRobin with assignment preservation.
- **Cooperative Sticky:** Follows the same StickyAssignor logic, but allows for cooperative rebalancing; which means that only consumers that change assignments pause the consumption during the rebalance, the other consumers keep consuming.

Compacted Topics



The log on top contains all events/records that are produced by the data source (each event in this slide has an offset [or a time], a key and a value). Events with the same key are color coded with the same color for easier readability.

If we create a new stream that only ever contains the last event per key of the full log then we call this "log compaction". The lower part of the slide shows the result of such a compaction.

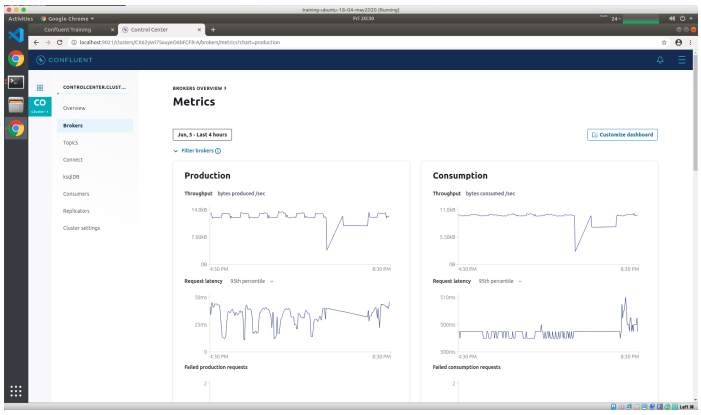
Compacted logs are useful for restoring state after a crash or system failure. Kafka log compaction also allows downstream consumers to restore their state from a log compacted topic.

They are useful for in-memory services, persistent data stores, reloading a cache, etc. An important use case of data streams is to log changes to keyed, mutable data changes to a database table or changes to object in in-memory microservice.

Log compaction is a granular retention mechanism that retains the last update for each key. A log compacted topic log contains a full snapshot of final record values for every record key not just the recently changed keys.

Troubleshooting

- Confluent Control Center
- Log Files
- Special Config Settings
  - SSL logging
  - Authorizer debugging



- C3 provides many ways to help troubleshoot your Kafka cluster
  - Compare broker configuration files (like Git diff)
  - Monitor end-to-end message latency
  - Monitor consumer lag
  - define alerts for when thresholds are exceeded
- Log Files should be centrally aggregated e.g. in DataDog to avoid SSH access to brokers (security risk!)
  - create dashboards for anomalies
  - create & run queries for anomalies
  - define alerts for when thresholds are exceeded
- To troubleshoot e.g. misconfigured security settings use special config settings
  - **SSL logging:** Enable SSL debug logging at the JVM level by starting the Kafka broker and/or clients with the `javax.net.debug` system property.
  - **Authorizer Debugging:** It's possible to run with authorizer logs in DEBUG mode by making some changes to the log4j.properties file.

Security Overview

- Kafka supports Encryption in Transit
- Kafka supports Authentication and Authorization
- No Encryption at Rest out of the box!
- Clients can be mixed with & without Encryption & Authentication

Using Security is optional - BUT!

- Kafka supports cluster wide encryption in transit via TLS and authentication (SASL or TLS) as well as authorization via ACLs
- Kafka does not support encryption at rest out of the box but can be combined with other data encryption methods such as volume encryption or client based encryption.
- Kafka supports a mix of authenticated and unauthenticated, and encrypted and non-encrypted clients.

Using security is optional but highly recommended in any production or production like system.

Client Side Security Features

- Encryption of Data in Transit
- Client Authentication
- Client Authorization

Encryption in transit

SSL

authn & authz

authn: SASL or SSL
authz: ACLs

Some of the relevant security features are:

- **Encrypt data-in-transit between Kafka client applications and Kafka brokers:** You can enable the encryption of the client-server communication between your applications and the Kafka brokers. For example, you can configure your applications to always use encryption when reading and writing data to and from Kafka. This is critical when reading and writing data across security domains such as internal network, public internet, and partner networks.
- **Client authentication:** You can enable client authentication for connections from your application to Kafka brokers. For example, you can define that only specific applications are allowed to connect to your Kafka cluster.
- **Client authorization:** You can enable client authorization of read and write operations by your applications. For example, you can define that only specific applications are allowed to read from a Kafka topic. You can also restrict write access to Kafka topics to prevent data pollution or fraudulent activities.

Q&A

Question: Why is it important to secure your Kafka cluster? Name ways that you can use to secure parts of, or the whole cluster.

The idea is that the learner understand that security is of utmost importance when running a Kafka cluster in production or in a production like environment. Various means can be used to achieve this goal:

- run the cluster on a secured private network with access only through secured gateways
- use TLS to encrypt data in transit between the brokers themselves and the clients and brokers
- use either MTLS or SASL to authenticate access to the brokers from the clients
- use ACLs to lock down access to resources (use the least privileges principle)
- run components of the cluster (brokers, clients, etc.) in containers in e.g. Kubernetes to harden your components even more (according to Gartner applications running in containers are more secure than their natively running counterparts)

Hands-On Lab

Refer to the lab **03 - How Kafka Works** in the Exercise Book.

Further Reading

- Kafka Clients: https://docs.confluent.io/current/clients/index.html
- Optimizing your Kafka Deployment:
  - https://www.confluent.io/blog/optimizing-apache-kafka-deployment/
  - https://www.confluent.io/white-paper/optimizing-your-apache-kafka-deployment/
- Is it OK to store data in Kafka? https://www.confluent.io/blog/okay-store-data-apache-kafka/
- Security Tutorial: https://docs.confluent.io/current/tutorials/security_tutorial.html
- Consumer Rebalances | Monitoring Kafka in Confluent Control Center: https://youtu.be/2Egh3l0q4dE
- Apache Kafka and Stream Processing O'Reilly Book Bundle: https://www.confluent.io/apache-kafka-stream-processing-book-bundle

---

These are are few links to material diving into the topics of this module in more depth.

05 Integrating Kafka into your Environment

Agenda

1. Introduction
2. Motivation & Customer Use Cases
3. Apache Kafka Fundamentals
4. How Kafka Works
5. Integrating Kafka into your Environment …
6. The Confluent Platform
7. Conclusion

Learning Objectives

After this module you will be able to:

- Explain how the **Confluent REST Proxy** works
- Justify why **Confluent Schema Registry** is an essential piece of a streaming platform powered by Kafka
- Sketch where and how **Kafka Connect** is used in streaming ETL
- Name 3 to 4 main features of **Kafka Streams**
- Elaborate on 3 to 4 main goals of **Confluent ksqlDB**

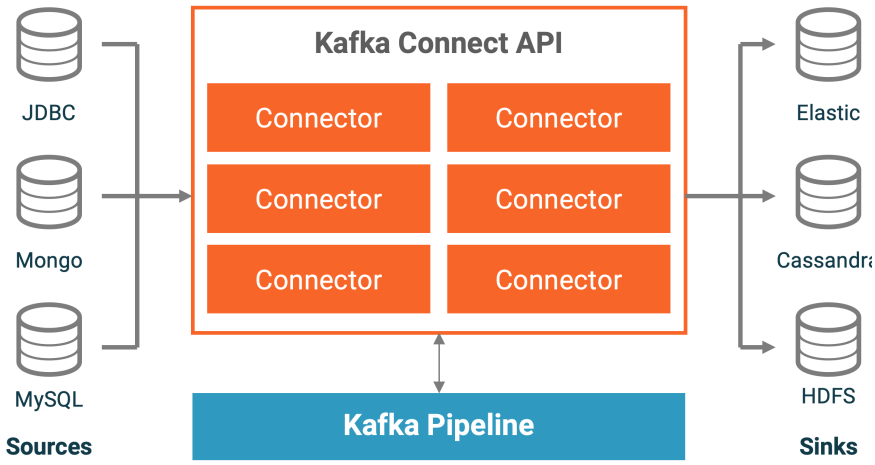
Development & Connectivity

How do I get streams of data into and out of Kafka?

Apache Kafka Connect

Import and Export Data In & Out of Kafka

- Fault tolerant
- Manage hundreds of data sources and sinks
- Preserves data schema
- Integrated within Confluent Control Center

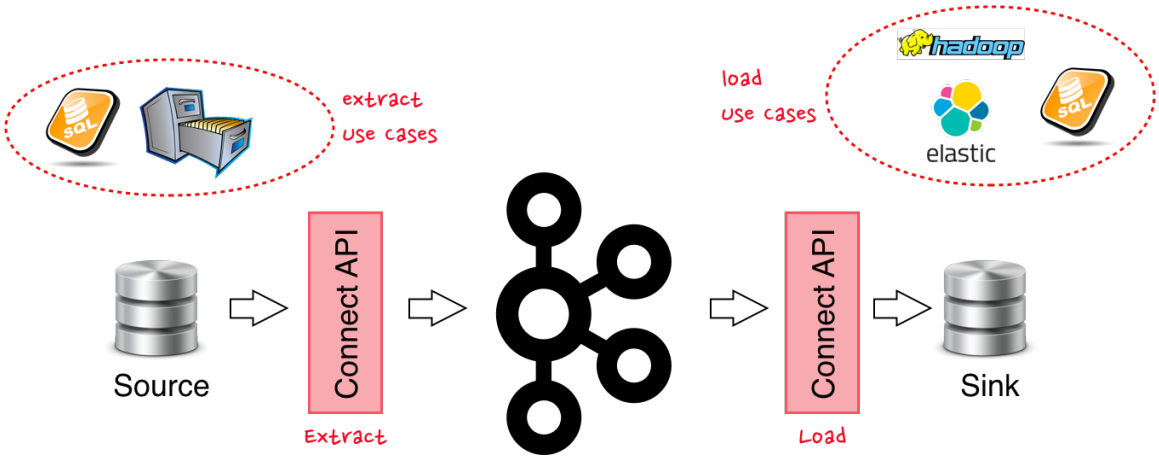


Kafka Connect, a part of the Apache Kafka project, is a standardized framework for handling import and export of data from Kafka. This framework can address a variety of use cases, makes adopting Kafka much simpler for users with existing data pipelines; encourages an ecosystem of tools for integration of other systems with Kafka using a unified interface; and provides a better user experience, guarantees, and scalability than other frameworks that are not Kafka-specific.

Kafka Connect is a pluggable framework for inbound and outbound connectors. It's fully distributed and integrates Kafka with a number of sources and sinks. This approach is operationally simpler, particularly if you have a diverse set of other systems you need to copy data to/from as it can be holistically and centrally managed. It also pushes a lot of the hard work of scalability into the framework, so if you want to work in the Kafka Connect framework you would only have to worry about getting your data from a source into the Kafka Connect format – scalability, fault tolerance, etc would be handled for intrinsically.

Most of the data systems illustrated here are supported with both Sources and Sinks (including JDBC, MySQL, Mongo, and Cassandra).

Apache Kafka Connect



- Framework for Streaming Data between Kafka and other Systems
- Open Source
- Simple, Scalable, and Reliable

- Kafka Connect is a framework for streaming data between Apache Kafka and other data systems
- Kafka Connect is open source, and is part of the Apache Kafka distribution
- It is simple, scalable, and reliable

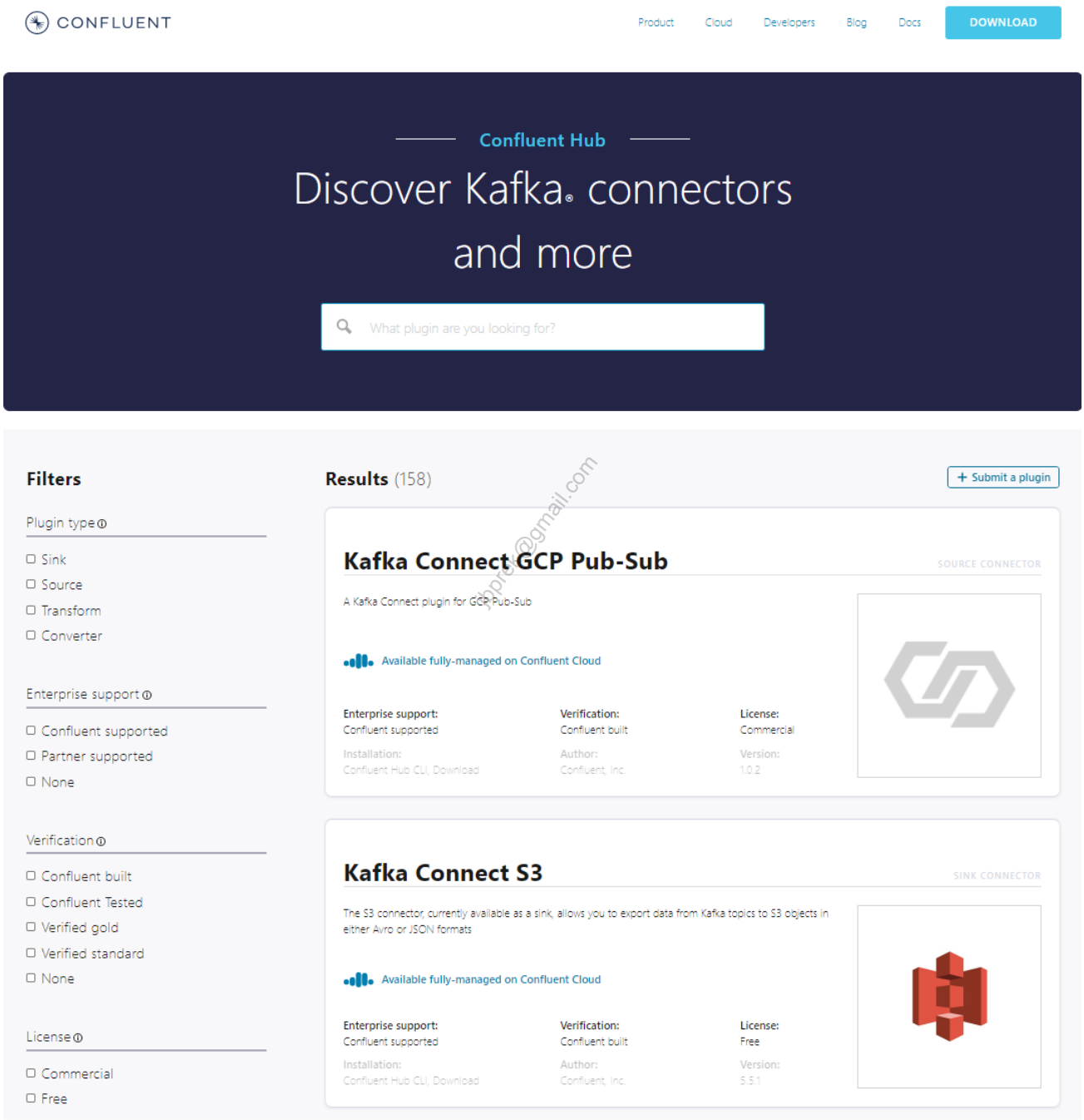
Kafka Connect is not an API like the Client API (which implements Producers and Consumers within the applications you write) or Kafka Streams. It is a reusable framework that uses plugins called Connectors to customize its behavior for the endpoints you choose to work with.

Example Use Cases:

- Example use cases for Kafka Connect include:
  - Stream an entire SQL database into Kafka
  - Stream Kafka Topics into HDFS for batch processing
  - Stream Kafka Topics into Elasticsearch for secondary indexing
  - ...

Apache Kafka Connect

Connectors: Connect Kafka Easily with Data Sources and Sinks



This slide shows Confluent Hub, your source for over 150 Kafka connectors. Using the available filters, you can search for Kafka connectors based upon type, support availability,

verification level, license requirement, and availability on Confluent Cloud.

You can find the Confluent Hub on the web at https://confluent.io/hub

Source Connector examples:

- Datagen
  - great for generating test data and doing load testing
  - feed it custom schemas to authentically simulate your use case
- JDBC
  - great for getting SQL database tables into Kafka
- Confluent Replicator
  - great for migrating data from one kafka cluster to another
- IoT Hub
  - get data from Azure IoT Hub into Kafka

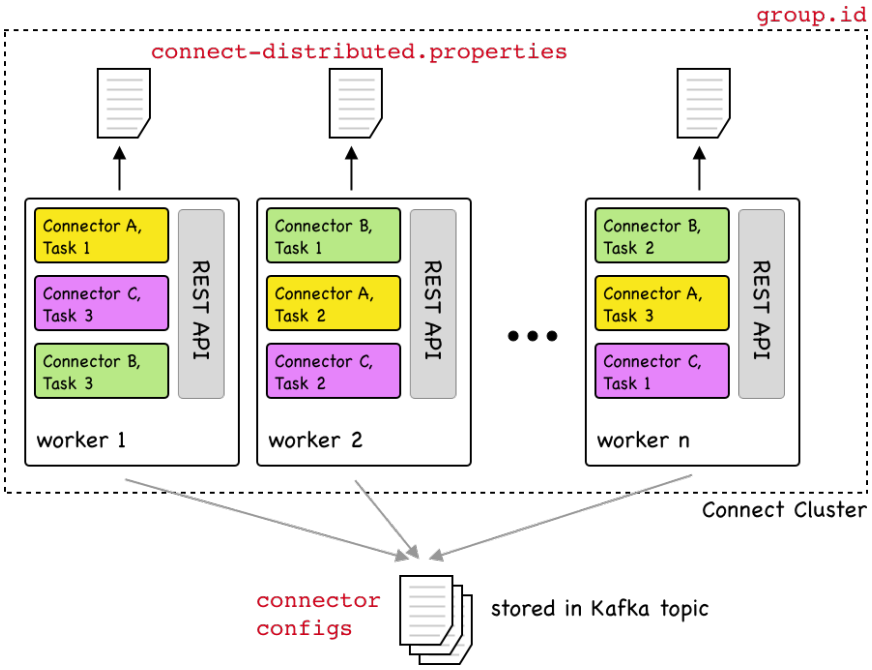
Sink Connector examples:

- Apache Druid
  - streaming analytics database that plays nicely with Kafka
- ElasticSearch
  - index everything
- AWS Simple Storage Service (S3)
  - archive data in S3 at high throughput
- Google Cloud Storage (GCS)
  - archive data in GCS at high throughput

For a detailed description of Confluent Hub:
https://www.confluent.io/blog/introducing-confluent-hub/

A goal of Confluent is to provide a comprehensive set of supported connectors.

Apache Kafka Connect



Providing Parallelism and Scalability

- Splitting the workload into smaller pieces provides the **parallelism** and **scalability**
- Connector jobs are broken down into **Tasks** that do the actual copying of the data
- **Workers** are processes running one or more tasks, each in a different thread
- Input stream can be partitioned for parallelism

A Connector is a Connector class and a configuration. Each connector defines and updates a set of tasks that actually copy the data. Connect distributes these tasks across workers.

In the case of Connect, the term partition can mean any subset of data in the source or the sink. How a partition is represented depends on the type of Connector (e.g., tables are partitions for the JDBC connector, files are the partitions for the FileStream connector).

Worker processes are not managed by Kafka. Other tools can do this: YARN, Kubernetes, Chef, Puppet, custom scripts, etc.

In addition to tasks, each connector has a **connector thread** that coordinates the job. For simplicity, we only show tasks in the diagram.

As with Kafka topics, the position within the partitions used by Connect must be tracked as well to prevent the unexpected replay of data. As with the partitions, the object used as the

offset varies from connector to connector, as does the method of tracking the offset. The most common way to track the offset is through the use of a Kafka topic, though the Connector developer can use whatever they want.

The variation is primarily with source connectors. With sink connectors, the offsets are Kafka topic partition offsets, and they're usually stored in the external system where the data is being written.

Running in Distributed Mode

- To run in distributed mode, start Kafka Connect on each worker node

```
$ connect-distributed connect-distributed.properties
```

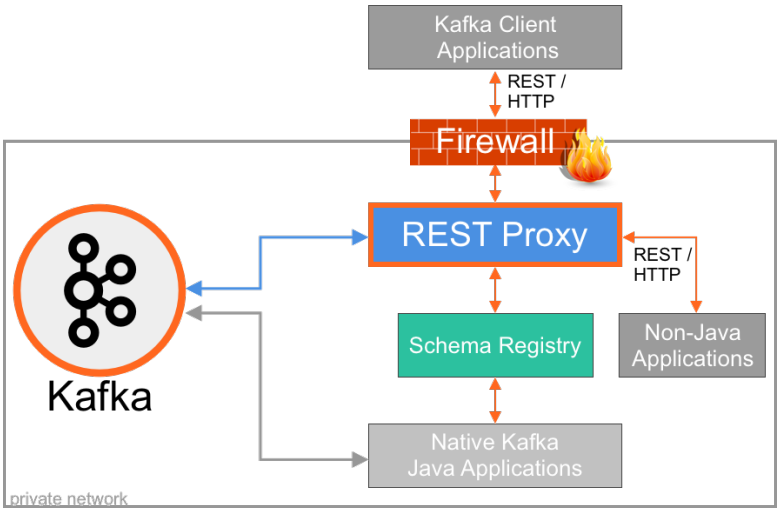
- Group coordination
  - Connect leverages Kafka's group membership protocol
    - Configure workers with the same `group.id`
  - Workers distribute load within this Kafka Connect "cluster"

In distributed mode, the connector configurations cannot be kept on the Worker systems; a failure of a worker should not make the configurations unavailable. Instead, distributed workers keep their connector configurations in a special Kafka topic which is specified in the worker configuration file.

Workers coordinate their tasks to distribute the workload using the same mechanisms as Consumer Groups.

Confluent REST Proxy

Talk to non-native Kafka Apps and outside the Firewall



Provides a **RESTful interface** to a Kafka cluster

Simplifies message creation and consumption

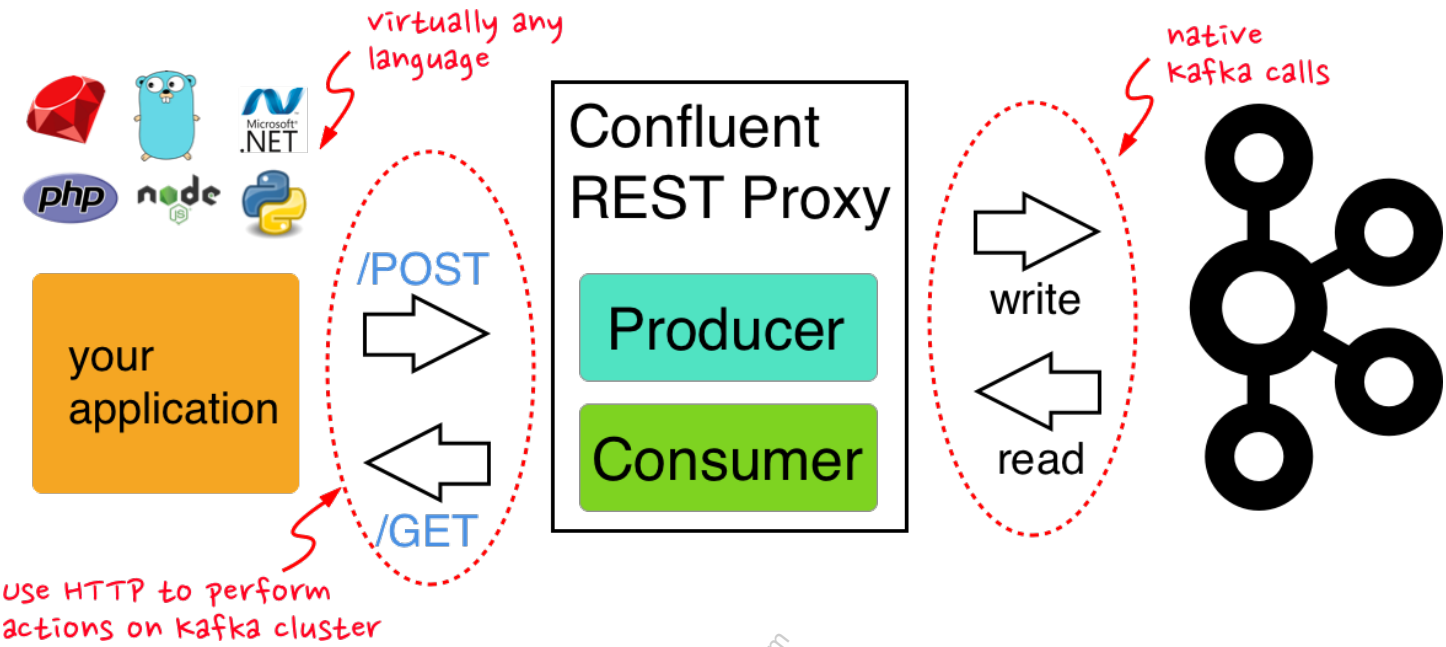
Simplifies administrative actions

Three uses for the REST Proxy

1. For remote clients (such as outside the datacenter including over the public internet)
2. For internal client apps that are written in a language not supported by native Kafka client libraries (i.e. other than Java, C/C++, Python, and Go)
3. For developers or existing apps for which REST is more productive and familiar than Kafka Producer/Consumer API.

Regarding "**Simplifies** administrative actions": REST Proxy also allows to perform administrative actions without using the native Kafka protocol or clients.

Confluent REST Proxy



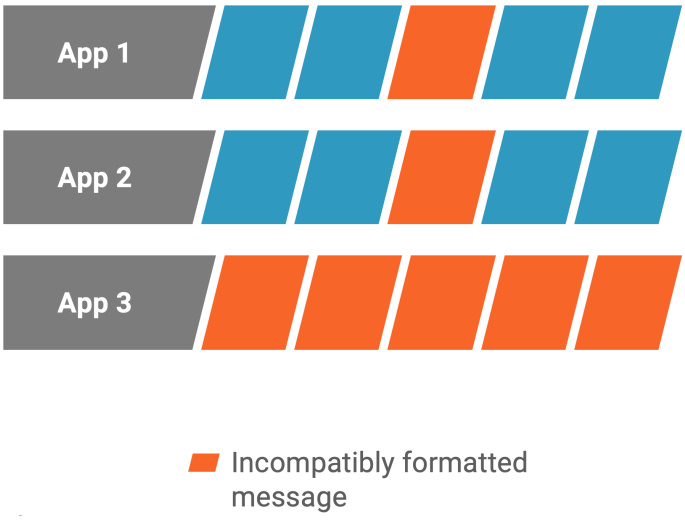
- The REST Proxy allows an administrator to use HTTP to perform actions on the Kafka cluster
- The REST calls are translated into native Kafka calls
- This allows virtually any language to access Kafka
- Uses POST to send data to Kafka
- Embedded formats: JSON, base64-encoded JSON, or Avro-encoded JSON
- Uses GET to retrieve data from Kafka

	Although there are native clients for the most important languages next to Java such as .NET/C#, Python, Go, C/C++, etc., there is still a need for REST Proxy even when using one of those languages to build Kafka clients. Imagine a situation where the Kafka Cluster sits behind a firewall which only opens port 80 and 443. In this case Kafka clients sitting outside this firewall can still access Kafka (via the REST Proxy).

Data Compatibility

How do I maintain my data formats and ensure compatibility?

The Challenge of Data Compatibility at Scale



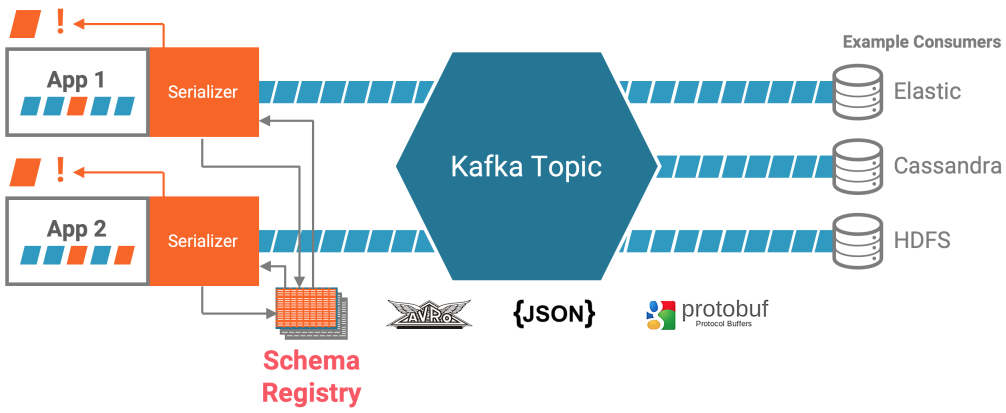
Many sources without a policy **causes mayhem** in a centralized data pipeline

Ensuring downstream systems can **use the data is key** to an operational stream pipeline

Even within a single application, **different formats can be presented**

Confluent Schema Registry

Make Data Backwards Compatible and Future-Proof



- **Define** the expected fields for each Kafka topic
- Automatically **handle** schema changes (e.g. new fields)
- **Prevent** backwards incompatible changes
- **Support** multi-data center environments

This diagram shows what schema registry is for. From left to right:

- Schema registry is specific to a Kafka cluster, so you can have one schema registry per cluster.
- Applications can be third party apps like Twitter or SFDC or a custom application. Schema registry is relevant to every producer who can feed messages to your cluster.
- The schema registry exists outside your applications as a separate process.
- Within an application, there is a function called serializer that serializes messages for delivery to the kafka data pipeline. Confluent's schema registry is integrated into the serializer. Some other SaaS provider companies have created their own schema registries outside the serializer; argument is that's more complexity for the same functionality.

The process goes like this:

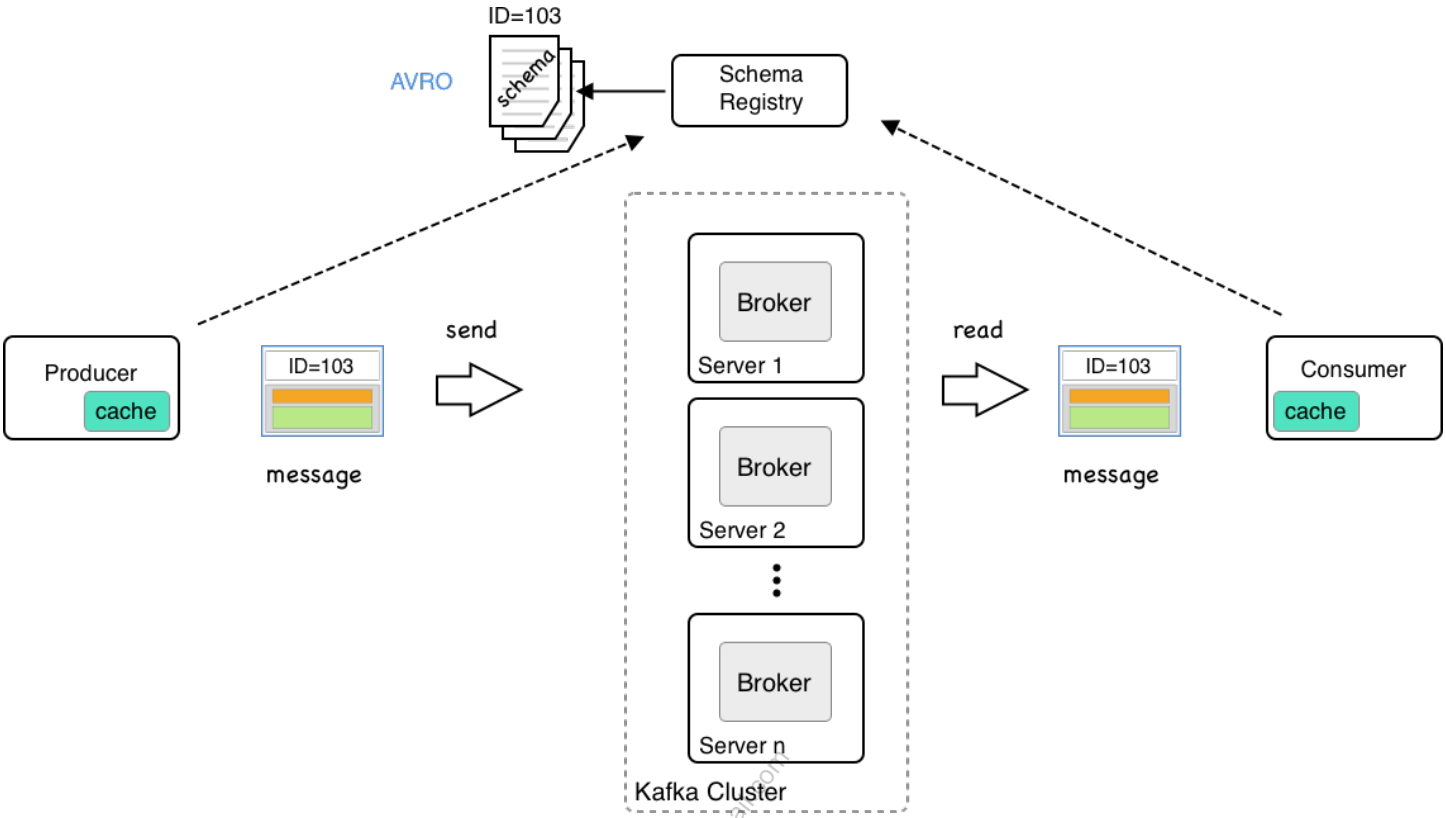
- The serializer places a call to the schema registry, to see if it has a format for the data the application wants to publish.
- If it does, then schema registry passes that format to the application's serializer, which uses it to filter out incorrectly formatted messages. This keeps your data pipeline clean.

The formats Confluent uses for its schema registry are **Avro, Protobuf, and JSON**.

When you use the Confluent schema registry, it's automatically integrated into the serializer and there's no effort you need to put into it. Your data pipeline will just be clean. You simply need to have all applications call the schema registry when publishing.

The ultimate benefit of schema registry is that the consumer applications of your kafka topic can successfully read the messages they receive.

Confluent Schema Registry

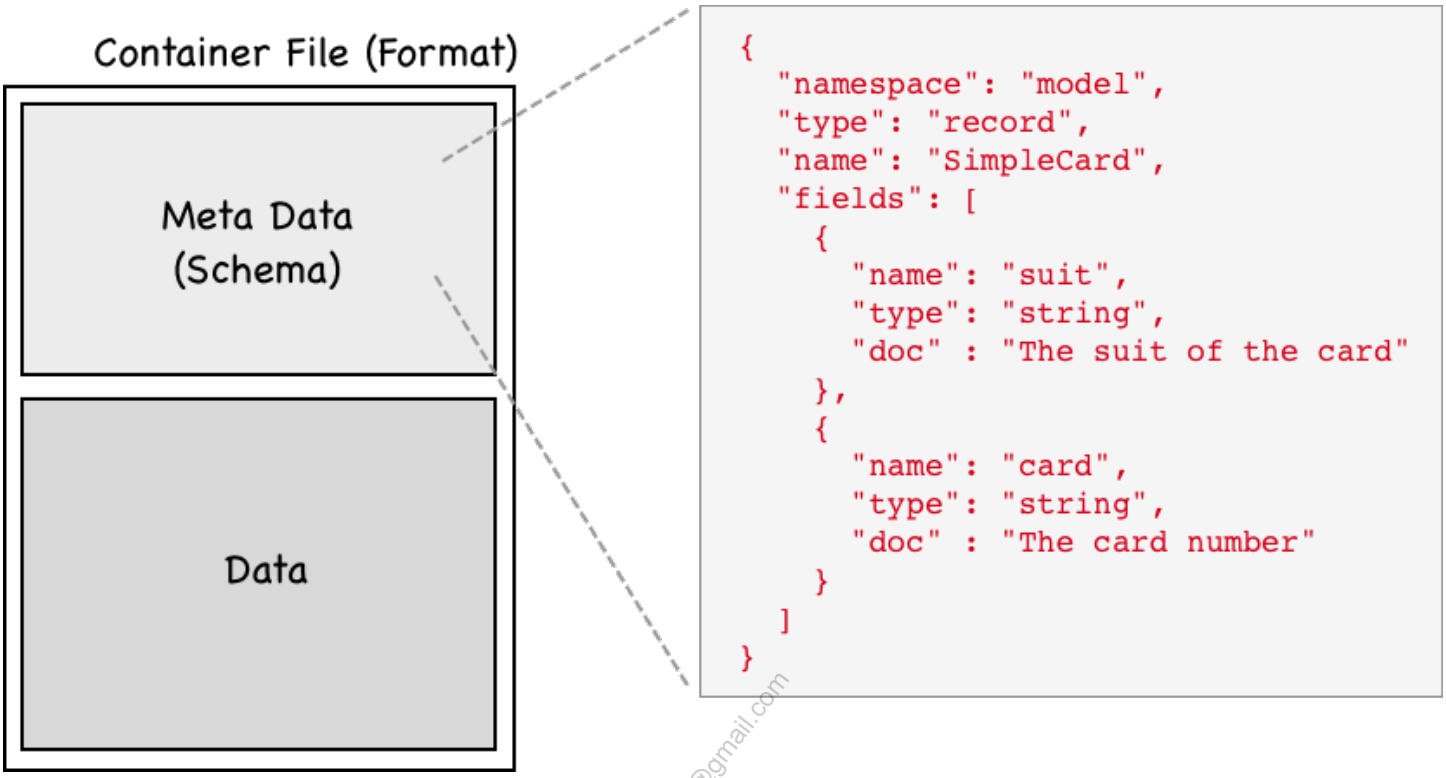


- Schema registry provides centralized management of schemas
  - Stores a versioned history of all schemas
  - Provides a RESTful interface for storing and retrieving Avro schemas
  - Checks schemas and throws an exception if data does not conform to the schema
  - Allows evolution of schemas according to the configured compatibility setting
- Sending the Avro schema with each message would be inefficient
  - Instead, a globally unique ID representing the Avro schema is sent with each message
- The Schema Registry stores schema information in a special Kafka topic
- The Schema Registry is accessible both via a REST API and a Java API
- Command-line tools to work with Avro: `kafka-avro-console-producer` & `kafka-avro-console-consumer`
- In addition to Command-line tools tools kafka-avro-console-producer & kafka-avro-console-consumer, we now have protobuf and JSON Schema equivalents

Schema Registry and Data Flow:

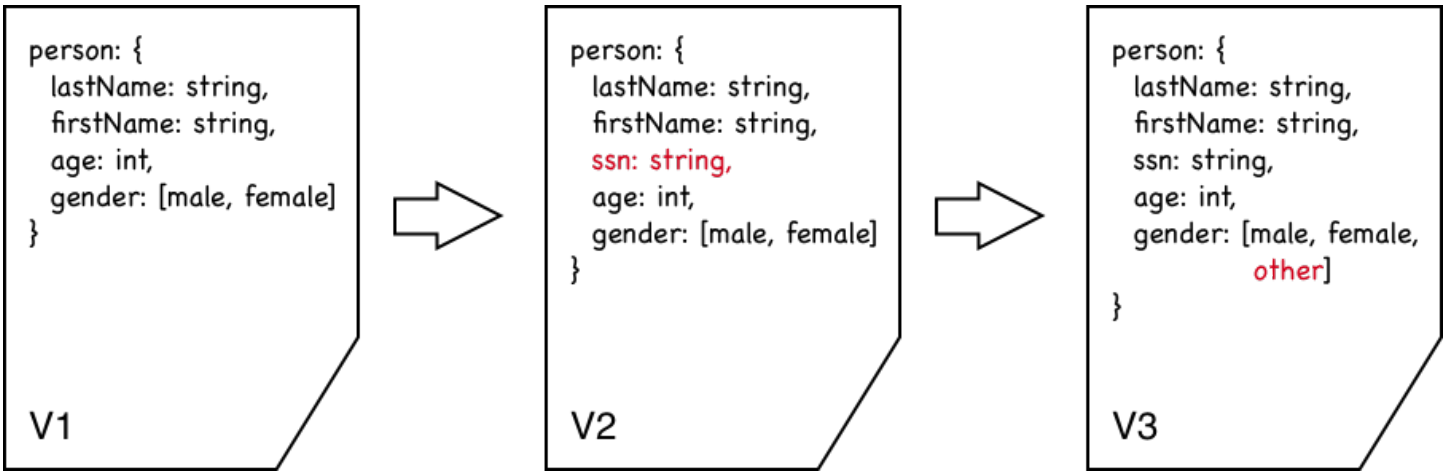
1. The message key and value can be independently serialized
2. Producers serialize data and prepend the schema ID
3. Consumers use the schema ID to deserialize the data
4. Schema Registry communication is only on the first message of a new schema
5. Producers and Consumers cache the schema/ID mapping for future messages

AVRO



- Avro is an Apache open source project
- Provides data serialization
- Data is defined with a self-describing schema
- Supported by many programming languages, including Java
- Provides a data structure format
- Supports code generation of data types
- Provides a container file format
- Avro data is binary, so it stores data efficiently
- Type checking is performed at write time

Schema Evolution



- Avro schemas may evolve as updates to code happen
- The **Confluent Schema Registry** allows schema evolution
- We often want compatibility between schemas. The following can be distinguished:
  - `Backward` compatibility
    - Code with a new version of the schema can read data written in the old schema
    - Code that reads data written with the schema will assume default values if fields are not provided
  - `Forward` compatibility
    - Code with previous versions of the schema can read data written in a new schema
    - Code that reads data written with the schema ignores new fields
  - `Full` compatibility
    - If both **Forward** and **Backward** compatibility are fulfilled

Stream Processing

How do I build real-time applications?

Confluent ksqlDB

Streaming SQL Engine for Apache Kafka

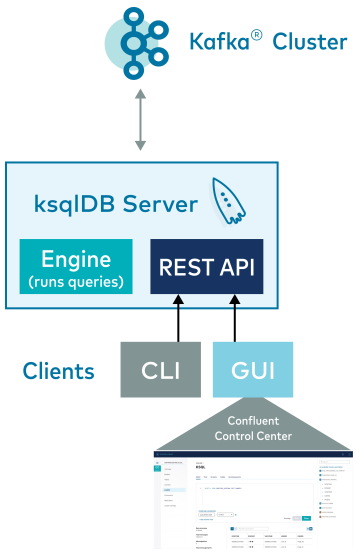
Develop real-time stream processing apps writing only SQL! No Java, Python, or other boilerplate to wrap around it

```
ksql> CREATE STREAM vip_actions AS
      SELECT userid, page, action
      FROM clickstream c
      LEFT JOIN users u ON c.userid = u.user_id
      WHERE u.level = 'Platinum';
```

Confluent ksqlDB

Enable Stream Processing using SQL-like Semantics

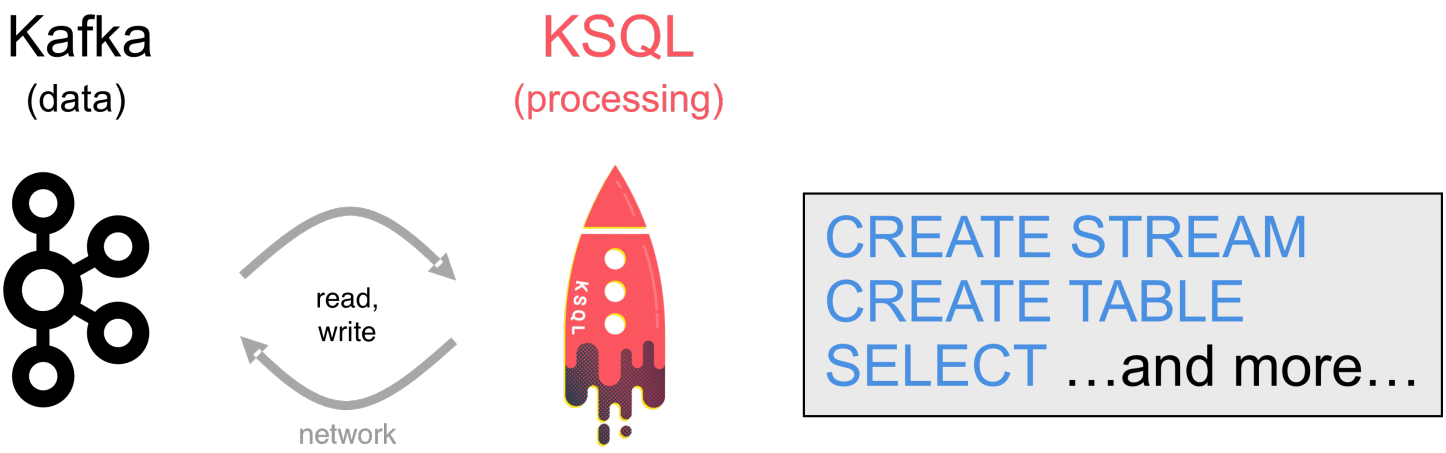
- Leverage Kafka Streams API without any coding required
- Use any programming language
- Connect via Control Center UI, CLI, REST or headless
- Example Use Cases
  - Streaming ETL
  - Anomaly detection
  - Event monitoring



ksqlDB is the open source streaming SQL engine for Apache Kafka. It provides an easy-to-use yet powerful interactive SQL interface for stream processing on Kafka, without the need to write code in a programming language such as Java or Python. ksqlDB is scalable, elastic, fault-tolerant, and real-time. It supports a wide range of streaming operations, including data filtering, transformations, aggregations, joins, windowing, and sessionization.

	ksqlDB can be extended by writing User Defined Functions (UDFs) or User Defined Aggregate Functions (UDAFs) in Java. This allows for new use cases such as machine learning and more.

Confluent ksqlDB & Apache Kafka = easy



All you need is Kafka – no complex deployments of bespoke systems for stream processing!

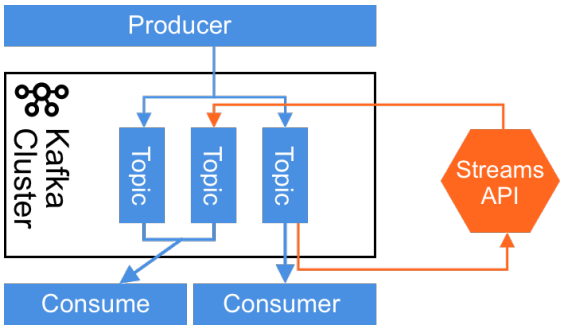
ksqlDB reads and writes data from and to your Kafka cluster over the network. Yes, that's right, ksqlDB runs on its own dedicated servers and not as part of the Kafka cluster

Apache Kafka Streams

Transform Data with Real-Time Applications

Overview

- Write standard Java applications
- No separate processing cluster required
- Exactly-once processing semantics
- Elastic, highly scalable, fault-tolerant
- Fully integrated with Kafka security

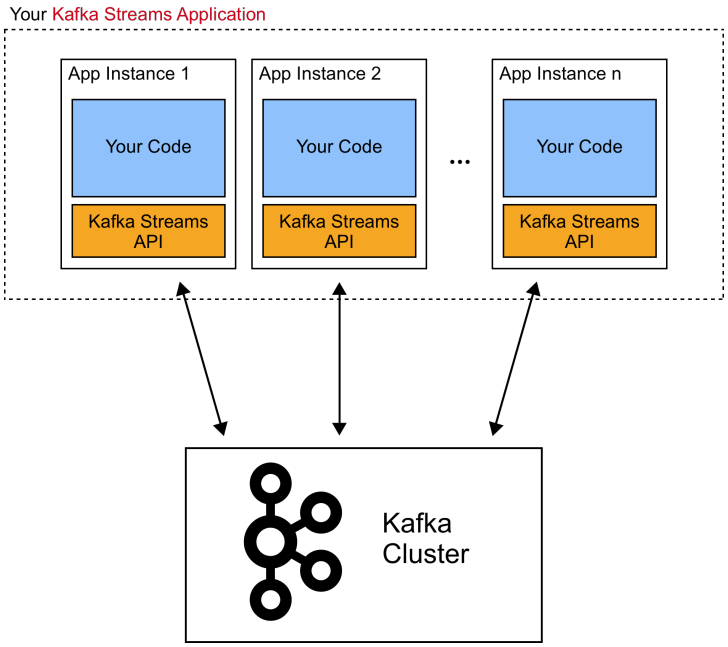


Example Use Cases

- Microservices
- Continuous queries
- Continuous transformations

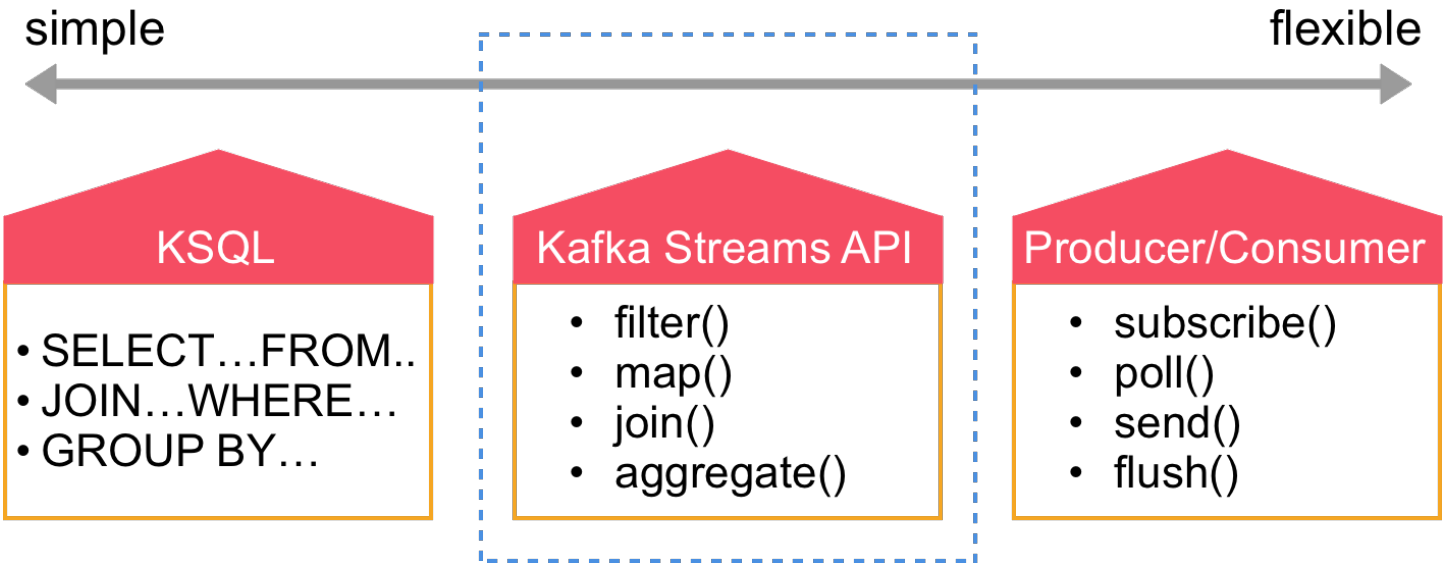
Kafka Streams is a client library for building applications and microservices, where the input and output data are stored in a Kafka cluster. It combines the simplicity of writing and deploying standard Java and Scala applications on the client side with the benefits of Kafka's server-side cluster technology.

Your Kafka Streams App



- The Streams API of Apache Kafka, available through a Java library, can be used to build highly scalable, elastic, fault-tolerant, distributed applications and microservices.
- A unique feature of the Kafka Streams API is that the applications we build with the Kafka Streams library are normal Java applications. These applications can be packaged, deployed, and monitored like any other Java application – there is no need to install separate processing clusters or similar special-purpose and expensive infrastructure!
- We can run more than one instance of the application. The instances run independently but will automatically discover each other and collaborate.
- We can elastically add or remove instances during live operation.
- If one instance **unexpectedly dies** the others will automatically take over its work and continue where it left off.

Apache Kafka Streams



Let's first elaborate about the motivation of **why** one would care about Kafka Streams.

- For the most flexibility, but also high complexity, we can rely purely on the **Kafka Client** library with its Producer and Consumer objects. We have learned about those in the previous module.
- If we want less **complexity** but are willing to trade in a bit of **flexibility** then that's where **Kafka Streams** comes into play. We can build an application that uses the Kafka Streams library. It offers us an easy to learn and manage DSL with functions such as **filter**, **map**, **groupBy** or **aggregate**.
- Later we will discuss KSQL, and seen why it appeals to so many people. One main point is its simplicity. But looking at the graph on the slide we have this divergence between **simplicity** and **flexibility**. You cannot have both at the same time. The more flexible you want to be the more complex a system usually becomes. **KSQL** is on the **flexibility** side. It is super easy to author a ksqlDB query that processes streams. That comes at a cost. ksqlDB has its clear boundaries.

In summary: Kafka Streams sits somewhere at the middle grounds and gives us a healthy mix of flexibility at a manageable complexity.

Q&A

Question:
In your company and/or team, which of the three methods to write a Kafka client application do you think will be the most appropriate:

a) Producer/Consumer
b) Kafka Streams application, or
c) KSQL

Justify your selection with 2 to 3 business relevant reasons.

The idea is that the learner understand that there are compromises in each method. Producer/consumer clients are the most flexible yet the most difficult to implement for complex problems. ksqlDB on the other hand is easy but also limited in its ability. The latter benefits from the possibility of extending it via UDFs and UDAFs, yet those have to be written in Java.

Somewhere in the middle sits the Kafka Streams API. It offers a fair amount of flexibility with a high level abstraction, that makes coding streaming applications fairly easy for a Java developer.

Hands-On Lab

Refer to the lab **04 - Integrating Kafka into your Environment** in the Exercise Book.

Further Reading

- Confluent REST Proxy: https://docs.confluent.io/current/kafka-rest/docs/index.html
- Confluent Schema Registry:
  - https://www.confluent.io/confluent-schema-registry/
  - https://docs.confluent.io/current/schema-registry/docs/index.html
- KAFKA, AVRO Serialization and the Schema Registry: https://bit.ly/2OLUkbb
- Introducing Kafka Streams: Stream Processing Made Simple: https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/
- ksqlDB Tutorials and Examples: https://docs.confluent.io/current/ksql/docs/tutorials/index.html
- Manage, Monitor and Understand the Apache Kafka Cluster: https://www.confluent.io/confluent-control-center/

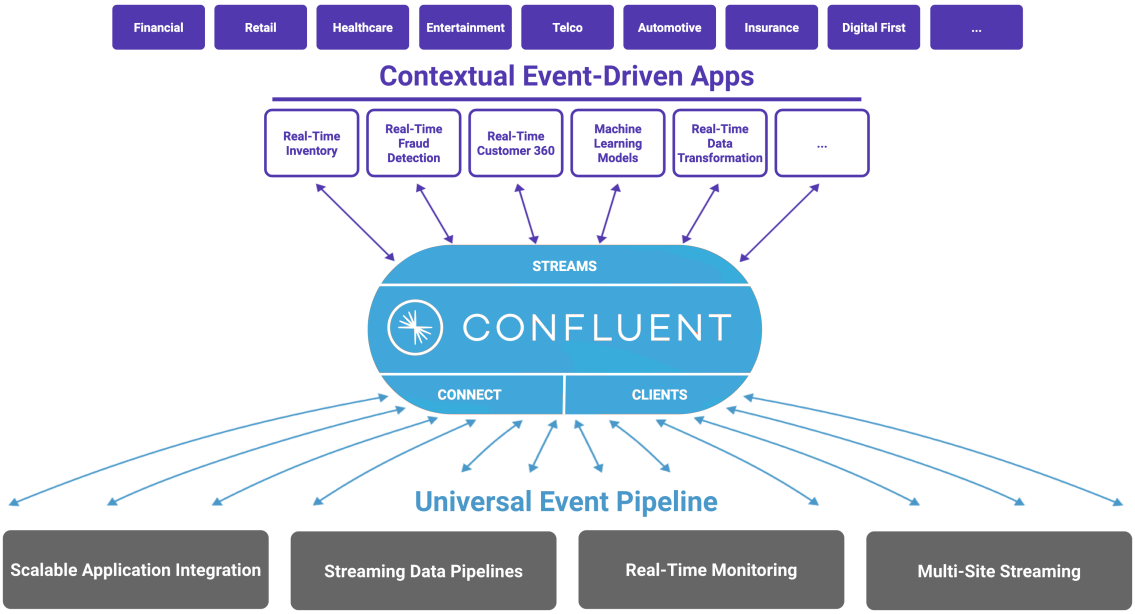
These are are few links to material diving into the topics of this module in more depth.

06 The Confluent Platform

Agenda

1. Introduction
2. Motivation & Customer Use Cases
3. Apache Kafka Fundamentals
4. How Kafka Works
5. Integrating Kafka into your Environment
6. The Confluent Platform …
7. Conclusion

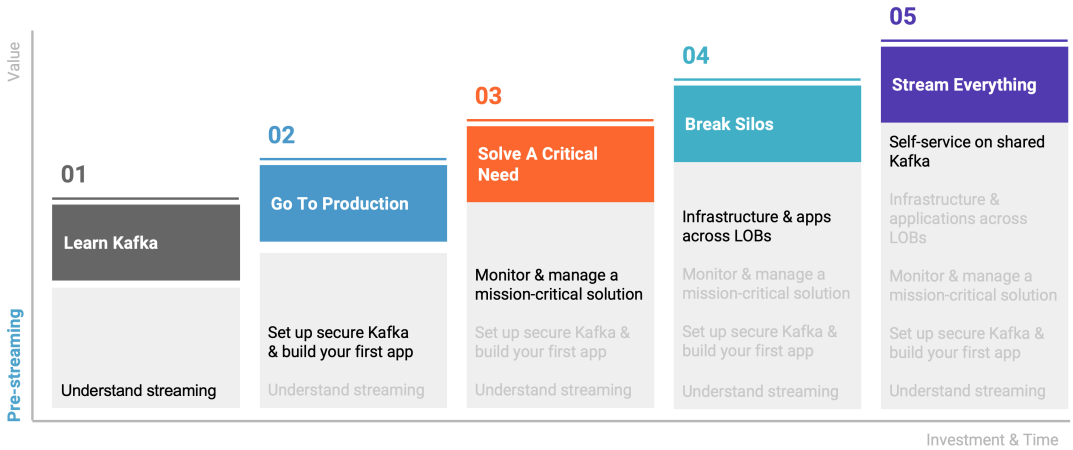
Central Nervous System



You'll see how an event streaming platform sits at the center of your applications and data infrastructure, connecting to other systems such as Data Warehouse, Hadoop, or custom apps, 3rd party apps, as well as to your event-driven applications. For this reason, customers have referred to it as the Central Nervous System of their enterprise.

The Maturity Model

Kafka is a Good Starting Point, Confluent Completes the Journey



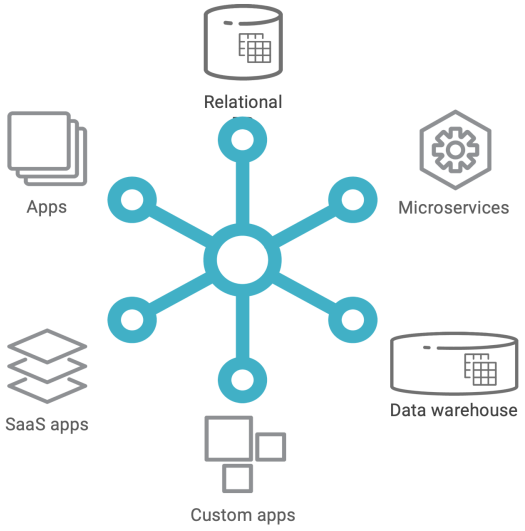
Confluent uses the **Maturity Model** shown on the slide.

The journey starts with awareness and moves up a curve, ending with a streaming platform acting as the central nervous system of the enterprise.

1. you are starting to realize the potential that streaming offers to your company
2. you go to production with a simple application
3. now you put a mission critical application into production and setup management and monitoring in parallel
4. more and more departments realize the benefit of streaming and like to participate. Silos are broken up
5. Everything is integrated with your company's streaming platform. The platform acts as the backbone or even better, as the central nervous system. It allows for self-servicing

Using Apache Kafka® on this journey is a good starting point. Yet along your journey, as your company matures you may encounter some challenges that the Confluent Platform can help you to solve in an elegant, cost and resource effective way. Confluent accompanies you with products, processes and support on all stages of the journey, until you stream everything.

Confluent Platform



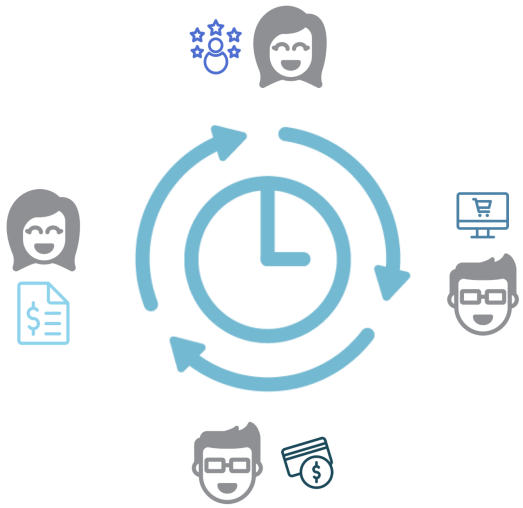
Build a Central Nervous System for your Modern Event-driven Enterprise

---

Confluent Platform complements Apache Kafka with a complete set of capabilities around connectivity, stream processing and operations that maximize developer productivity and ensure mission-critical reliability.

Developers focus primarily on building streaming applications on top of Kafka, caring mainly about the data and topics needed to solve important business problems. Others, like BI analysts, mainly run queries against Kafka and need to know what data is available and how to access it to feed their analytics toolset. Adding to this diversity, users also vary in their preferred modes of interacting with software. Some prefer to solve their problems by writing code, while others prefer to click buttons.

Confluent Platform



The Confluent Platform enables companies to respond accurately and in real time to business events

Confluent Platform

Confluent is Enabling Event-driven Transformation across Industries

Banking & Capital Markets

- Risk management
- Trade data capture
- Customer 360
- Fraud detection
- Mainframe offload

Retail

- Inventory management
- Product catalog
- A/B testing
- Customized experiences

Healthcare & Pharma

- Patient monitoring
- Prescription control
- Lab alerts
- Medication tracking

Automotive & Transportation

- Connected car
- Fleet management
- Manufacturing data processing

Telecommunications

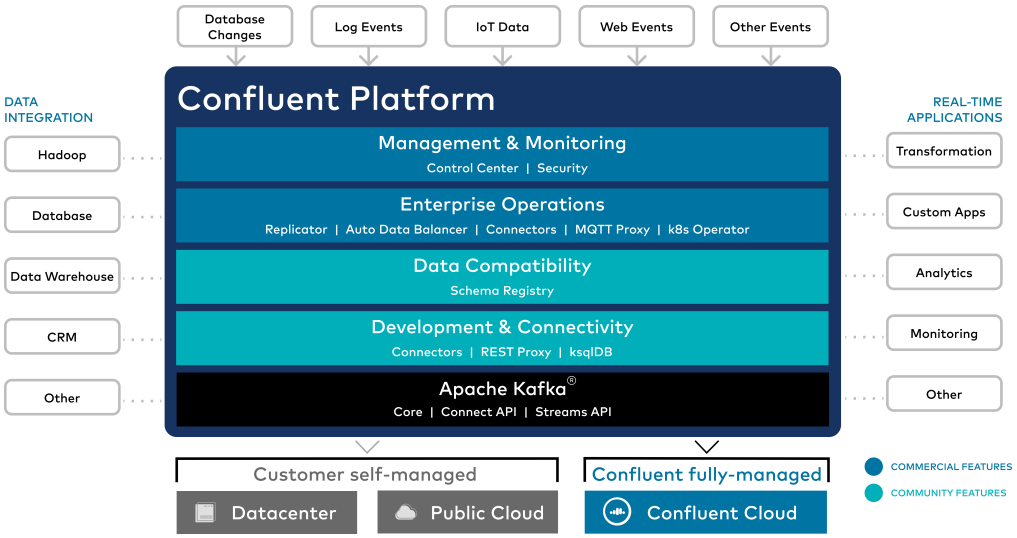
- Personalized ads
- Customer 360
- Network integrity

Travel & Leisure

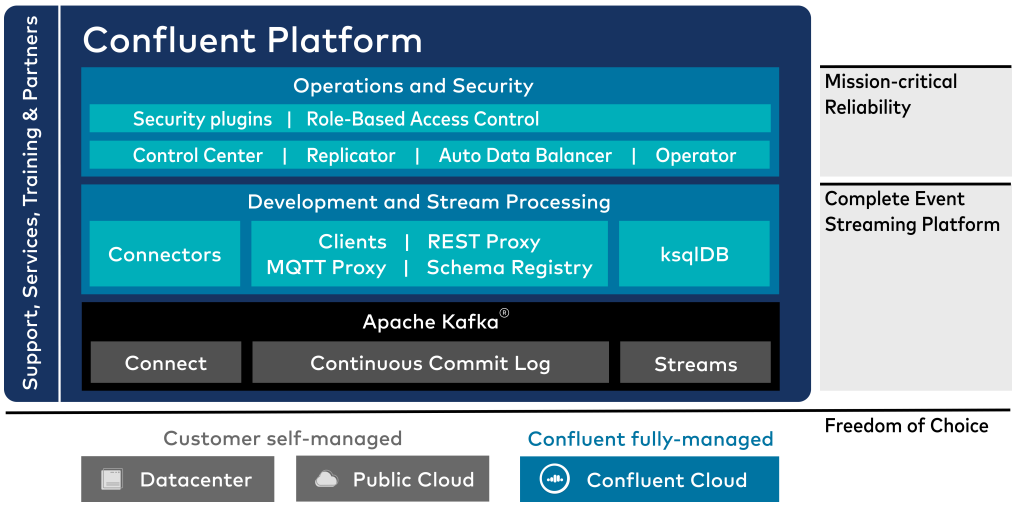
- Visitor segmentation
- Booking systems
- Pricing services
- Fraud detection

Confluent Platform

Complete Set of Development, Operations and Management Capabilities to run Kafka at Scale



Confluent Platform



Let's walk through an overview of Confluent Platform, the enterprise-ready event streaming platform built by the original creators of Apache Kafka.

Apache Kafka

- Let's start with Kafka. A lot of people think of Kafka as pub/sub messaging, but it's much more than that. At its core, Kafka delivers what we call a continuous commit log, which allows users to treat data neither as stored records nor transient messages, but as a continually updating stream of events. These streams are readily accessible, fast enough to handle events as they occur, and able to store events that previously occurred.
- Over the years, Kafka has evolved to add capabilities that complement the continuous commit log. In Kafka 0.9, we added in Kafka Connect, which is a framework to build connectors into external systems, such as databases, applications, other messaging systems, and so on.
- Then, in Kafka 0.10 we added in Kafka Streams, which is another framework, this time for building stateful and stateless applications that can process and enrich the event streams flowing through the platform in real time.
- The combination of the continuous commit log, along with the Connect and Streams frameworks constitutes what we call an Event Streaming Platform. However, running Kafka at enterprise-scale can come with its own set of challenges, so our goal at Confluent is to deliver an enterprise-ready platform that can become the central nervous system for any organization.

Complete Event Streaming Platform

- The first thing the Confluent Platform (CP) does to deliver a complete event streaming platform is to complement Kafka with advanced development and stream processing capabilities.
- To enhance the continuous commit log, CP offers tools to facilitate the flow of event streams through the platform:
  - Native client libraries to develop custom Kafka producers and consumers using programming languages different from Java or Scala which are the one packaged with open source Kafka. Confluent Platform adds clients for C/C++, Python, Go and .NET
  - REST Proxy provides a simple RESTful interface to produce and consume messages into Kafka, view the state of the cluster, and perform administrative actions without using the native Kafka protocol or clients.
  - For customers working on IoT projects, CP's MQTT Proxy offers an interface for MQTT clients to produce messages into Kafka without the need for third party brokers.
  - As event streams continue to flow into the event streaming platform, customers need a way to ensure data compatibility between all the producers and consumers. Schema Registry stores a versioned history of all schemas and allows the evolution of schemas while preserving backwards compatibility for existing consumers.
- Confluent Platform includes connectors that run on Kafka Connect. These are pre-built connectors for popular data sources and sinks, such as S3, HDFS, Elasticsearch, IBM MQ, Splunk, Cassandra and many others. We have many connector developed and supported by Confluent, and there are many more developed by partners and the broader Kafka community. To allow easy access to them, Confluent has an online marketplace called Confluent Hub.
- To complement Kafka Streams, Confluent delivers **ksqlDB**, which provides an easy-to-use, powerful SQL interface to perform stream processing operations against Kafka. ksqlDB fundamentally democratizes stream processing, because it eliminates the need to write code in a programming language such as Java or Python. ksqlDB is scalable, elastic, fault-tolerant, and it supports a wide range of streaming operations, including data filtering, transformations, aggregations, joins, windowing, and sessionization. ksqlDB is licenced under the Confluent Community open source license.

Mission-Critical Reliability

- Now that we have a complete event streaming platform, CP adds mission-critical reliability through a set of operations and security features.
- Let's start with the operations layer:
  - **Confluent Control Center** provides a simple way to view and understand key aspects and metrics of Kafka using a graphical user interface. It allows customers to track key metrics about event streams with expertly curated dashboards, check the health of Kafka brokers, and it provides useful integration points into other components of

Confluent Platform, such as Kafka Connect, ksqlDB and Schema Registry. Overall, Control Center is a critical component to meet even streaming SLAs in distributed Kafka environments.

  - For customers with geographically distributed operations that absolutely need to protect their mission-critical apps and data, CP offers **Confluent Replicator**. Replicator copies topics from one Kafka cluster to another using asynchronous replication, but in addition to copying the messages, Replicator will create topics as needed preserving the topic configuration in the source cluster. This capability allows customers to implement a variety of use cases, such as disaster recovery, data aggregation, data center consolidation and bridge to cloud.
  - Scaling Kafka up and down can be challenging at larger scale. **Auto Data Balancer** monitors the cluster and shifts data automatically to balance the load across brokers. This helps customers expand and contract the size of the cluster more dynamically.
  - Organizations tapping into cloud native operating models have started to standardize on Kubernetes as their platform runtime. **Operator** provides an implementation of the Kubernetes Operator API to automate the management of Confluent Platform in a cloud-native environment, and performs key operations such as rolling upgrades.

- On top of all of this, we have a security layer:
  - Confluent Platform provides a series of security plugins and features that allow customers to enhance the security of the complete event streaming platform.
  - Starting from version CP `5.3` Role-Based Access Control (RBAC) is introduced which will allow users to be added to predefined roles, which ensure that each persona in the organization has the right level of access to Kafka. This is a key feature that customers have been asking for.
- And mission critical reliability would not be complete without comprehensive enterprise support. With Confluent Platform, you will have access to 24/7 support and experts with the deepest knowledge base on Kafka and Confluent Platform, with guaranteed sub 1-hour response times and full application lifecycle support from development to operations.

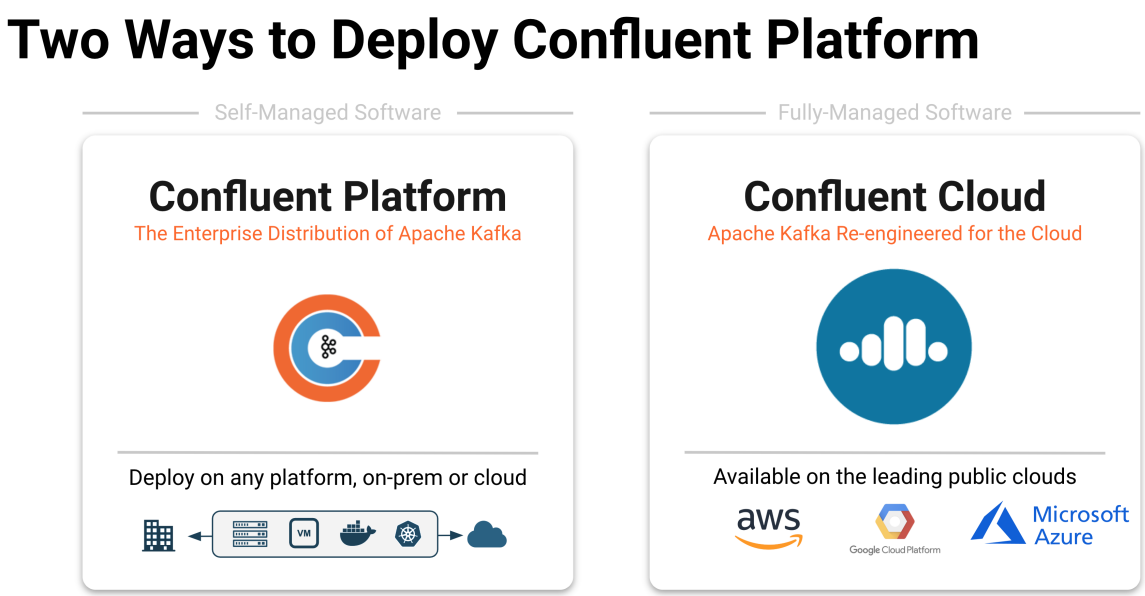
Freedom of Choice

- The last piece to delivering an enterprise-ready event streaming platform is to enable freedom of choice. With Confluent, customers have the ability to deploy anywhere and stream everywhere.
  - Confluent Platform can run on any kind of infrastructure, from bare metal to VMs to containers, and can be deployed on any cloud, private or public, with the consistency of single platform.
  - Moreover, Confluent delivers choice in the operational model. Customers can run Confluent Platform as self-managed software, but they can also offload operations to

Confluent's experts through **Confluent Cloud**, the industry's only fully-managed, complete streaming service.

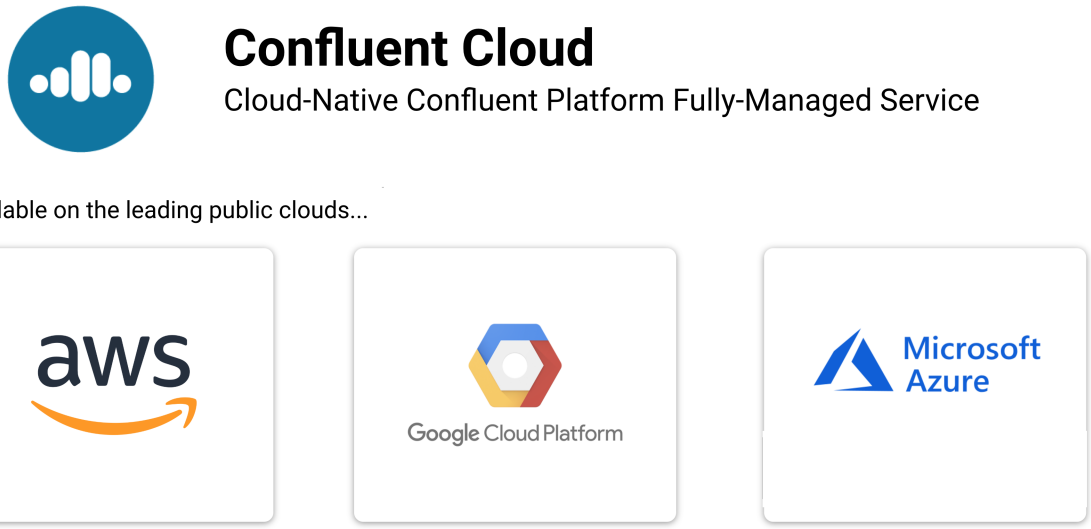
- Customers can build a bridge to cloud by streaming data between private and public clouds, enabling the use of best-of-breed cloud services regardless of where the data is produced.

Confluent Platform Deployment Models



There are two primary ways you can use Confluent Platform. On on premise distribution is called Confluent Platform and Confluent Cloud is Confluent's fully managed solution on AWS, GCP and Azure.

Confluent Cloud

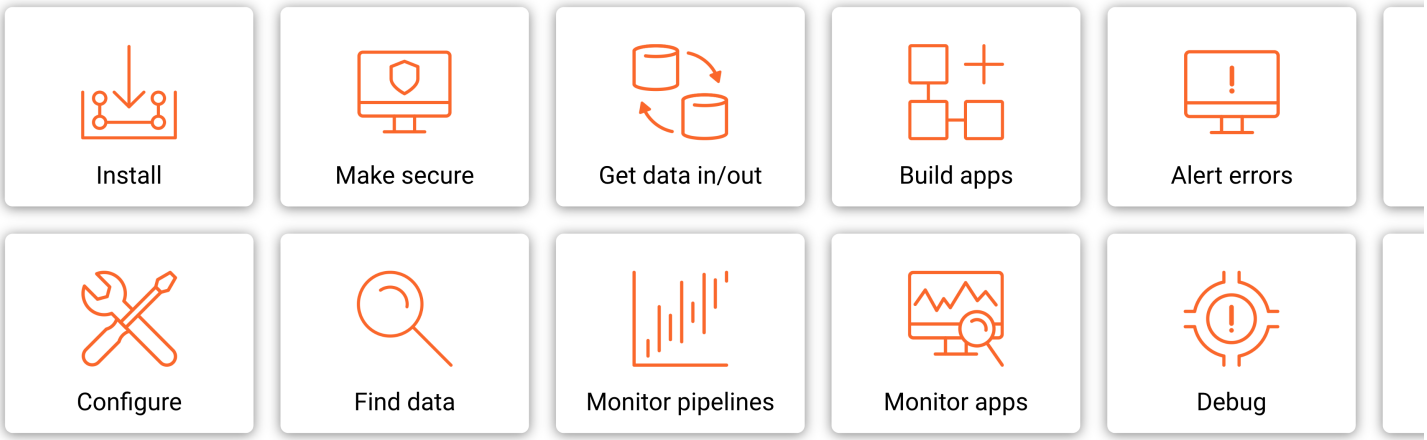


Confluent Cloud is a fully managed offering of the whole Confluent Platform. It frees you from managing your own deployment and allows you to concentrate on the things that bring value to your business, such as your context driven streaming applications.

Confluent Cloud is available on AWS, GCP and Azure.

Confluent Control Center

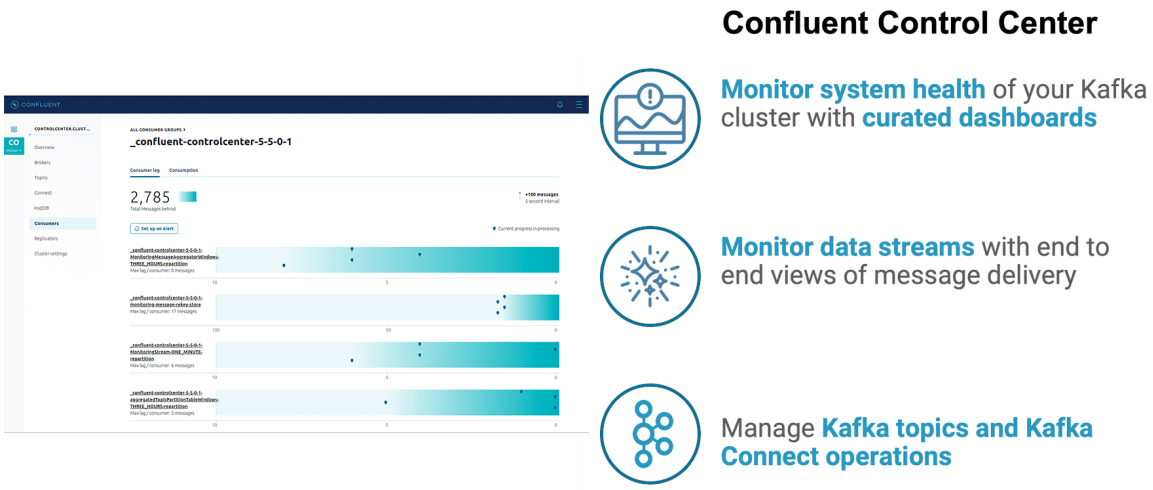
Kafka is powerful … but has many parts



We all know that Kafka is very powerful and very popular. But, there are also many things you need to both understand and do in order to use it. There are brokers, topics, partitions, producers, consumers, schemas, connectors, configuration, security, pipelines, and many other things that operators and application developers need to become familiar with in order to use Kafka. It's both a lot to understand and a lot to manage as part of standing up and using a streaming infrastructure.

Confluent Control Center

Management and Monitoring for the Enterprise



Confluent Platform offers intuitive GUIs for managing and monitoring Apache Kafka® (Control Center and Health+). These tools allow developers and operators to centrally manage and control key components of the platform, maintain and optimize cluster health, and use intelligent alerts to reduce downtime by identifying potential issues before they occur.

- Control Center is a self-hosted GUI with expert-designed dashboards to enable centralized management and monitoring of key components of the platform, including clusters, brokers, schemas, topics, messages, connectors, ksqlDB queries, security, replication and more.
- Health+ is a Confluent-hosted, web-based GUI that offers intelligent alerting and monitoring tools to reduce the risk of downtime, streamline troubleshooting, surface key metrics, and accelerate issue resolution. It helps offload monitoring costs related to storing historical data and is built on Confluent's deep understanding of data in motion infrastructure.

Confluent CLI

Overview

- **Platforms:** Linux, Unix-based
- **License:** Proprietary
- **Packaging:** Independent of CP

Grammar `<resource: noun> <optional subresource: noun> <operation: verb>`

Key features

- RBAC management
- Password protection
- Subsumed confluent-cli commands for `local`

```
Manage your Confluent Platform.
Usage:
  confluent [command]
Available Commands:
  completion  Print shell completion
code.
  help        Help about any command
  iam         Manage RBAC and IAM
permissions.
  local       Manage local Confluent
Platform
              development environment.
  login       Login to Confluent
Platform.
  logout      Logout of Confluent
Platform.
  secret      Manage secrets for
Confluent
              Platform.
  update      Update the confluent CLI.
  version     Print the confluent CLI
version.
  ...
```

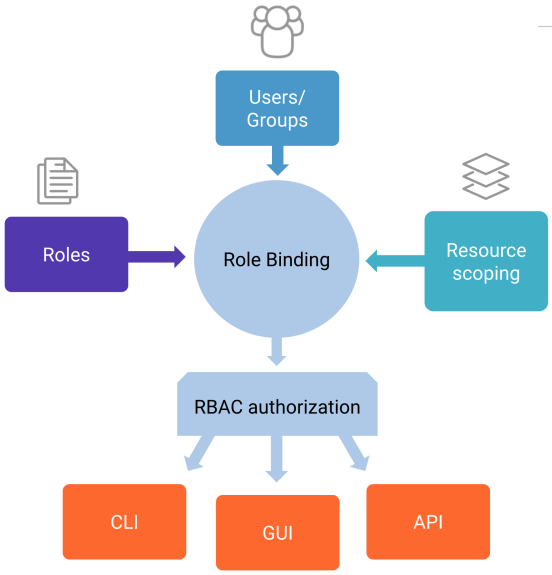
Confluent CLI is ready for production environments and fully supported by Confluent. The new CLI offers the following characteristics:

- **Consistent** – CLI grammar that's intuitive and easy-to-understand so new concepts fit into your existing mental model
- **Unambiguous** – Commands will be self-explanatory for those core concepts with which you are familiar, e.g. topics, schemas, etc.
- **Concise** – Command (with subcommands and arguments) will be easy to type
- **Scriptable** – Allows you to automate repeatable operations

Role Based Access Control

RBAC

- Utilizes a predefined set of Roles
- User who is assigned a role receives all privileges of that role
- Each user could belong to multiple roles
- Privileged user within a given scope can create/update other user's roles
- Privileged user can update roles for existing users
- Across the platform (KSQL, Connect, Schema Registry, Connectors, etc.)



Modern enterprises are extremely careful about ensuring their data is secure. Security breaches are very costly and put a major burden on organizations around system downtime and damage to their reputation in front of customers and partners. In many cases, organizations have dedicated security staff to ensure that any new platform that is utilized in production follows strict security best practices.

One of the pillars of a comprehensive security strategy is access controls. As the number of enterprise applications increases, so does the number of users (developers and operators) involved across multiple groups. Managing users and groups at scale is complex without a framework to provide controlled access to applications according to the roles of the users.

This is especially true with Apache Kafka, because of it's distributed nature. We have heard from Kafka users that it can take them several days to set ACLs for all their teams, because they do not know about the content of topics and who should have access to them. Overall, the process of setting access controls takes too long when it involves authorization for 100s or 1000s of resources, it is difficult to maintain and manage and it is error prone.

These are some of the key characteristics of RBAC:

- It is configured via the new CLI
- Authorization is enforced via all user interfaces:
  - Control Center (GUI)

    - the new CLI
    - APIs
- Authorization is enforced on all Confluent Platform components: KSQL, Connect, Schema Registry, REST Proxy and MQTT Proxy
- On Kafka Connect clusters, it provides Connector-level granularity
- It uses a set of 11 predefined roles to perform role-binding of resources to users/groups
- A user who is assigned a role receives all the privileges of that role
- Each user can belong to multiple roles

While the overall benefit of increased enterprise-level security across the entire platform is obvious, RBAC also allows you to:

- Delegate responsibility of managing permissions and access to true resource owners, such as departments/business units. This greatly simplifies the management of authorization at scale in shared platforms
- Set up Connector-level authorizations, so you can run multi-tenant Kafka Connect clusters shared across departments/business units. These increases resource utilization for your Connect clusters and provides a heightened level of security
- Use a centralized location to configure connections to AD/LDAP across all Confluent Platform components, instead of having to set this up independently for each one

These are the 11 predefined roles:

- **super.user** - The purpose of super.user is to have a bootstrap user who can initially grant another user the SystemAdmin role
- **SystemAdmin** - Has full access to all resources in the cluster
- **UserAdmin** - Responsible for managing user/groups and mapping them to roles (setting role bindings)
- **ClusterAdmin** - Responsible for managing Kafka clusters and brokers, including ksqlDB and Connect clusters. Can create topics but cannot read/write from/to topics. Cannot create/change applications. Can change the setup of topics, etc. Can make a user/group as the owner of that resource. Can make a user/group as the owner of an application. Manage quotas
- **Operator** - Responsible for monitoring the health of the application(s). Can start/stop applications but cannot change them
- **SecurityAdmin** - Responsible for managing security initiatives like managing audit logs, etc.
- **AuditAdmin** - Responsible for managing the audit log configuration using the Confluent

Metadata API

- **ResourceOwner** - Owns a set of resources and can grant permission to others to get access to those resources
- **DeveloperRead, DeveloperWrite, DeveloperManage** - With proper permissions can create/delete resources/applications. Can read/write from/to resources depending on scoping

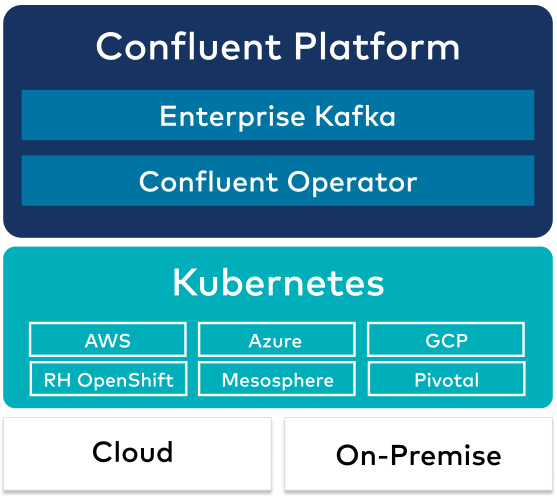
Confluent for Kubernetes (formerly Confluent Operator)

Confluent for Kubernetes (CFK) runs on any platform at any scale

Confluent for Kubernetes is the next generation of Confluent Operator

What:
Deployment and management automation for Confluent Platform across environments.

For Whom:
Organizations that have standardized on Kubernetes as the platform runtime.



- That is why we created Confluent for Kubernetes. It runs Kafka and soon the entire CP on Kubernetes, automates secure Kafka provisioning, automatically balances Kafka clusters, and helps scale it up and down as needed.
- Our fully managed service, Confluent Cloud, runs Confluent for Kubernetes underneath the covers. It is well tested, includes several operational improvements reflecting our experience of running hosted Kafka

Confluent for Kubernetes Capabilities

Deploy

- Single-Command deployment of Confluent Platform on Kubernetes
- Storage uses persistent Kubernetes Volumes (Storage Class based)
- Includes automated authentication and security configuration
  - SASL Plain, TLS certificate based 2-way authentication
  - TLS in transit encryption for all deployed components

Update / Upgrade

- Automated rolling upgrades (no impact to availability)
- Automated rolling updates to make config changes

Scale Up and Down

- Elastic Scaling of new Kafka / CP components (run data balancer manually)

Monitor

- JMX Kafka metrics aggregation - exportable to Prometheus

Confluent - Feature Comparison

Feature	Confluent Platform (On Premise)	Confluent Cloud (Fully Managed)
Scale	Unlimited throughput, unlimited retention	Unlimited throughput, unlimited retention (CCE)
Availability	Self Managed	99.99% uptime SLA
Durability	Self Managed	Multi-AZ with 3 availability zones (option)
Updates	3-4 Annual Releases	Weekly Fully Managed Updates
Support	24x7 Gold & Platinum (option)	24x7 Gold SVPC peering (option)
Pricing	Node based pricing	Usage Based Pricing
Packaging/ Environment	Tarball, deb, rpm, zip, Docker images	AWS, GCP, Azure

Usage Based Billing:

- Pay only for what you use based on ingress, egress and storage
- Launched May 2019 in Confluent Cloud, and in H2 2019 in Cloud Enterprise

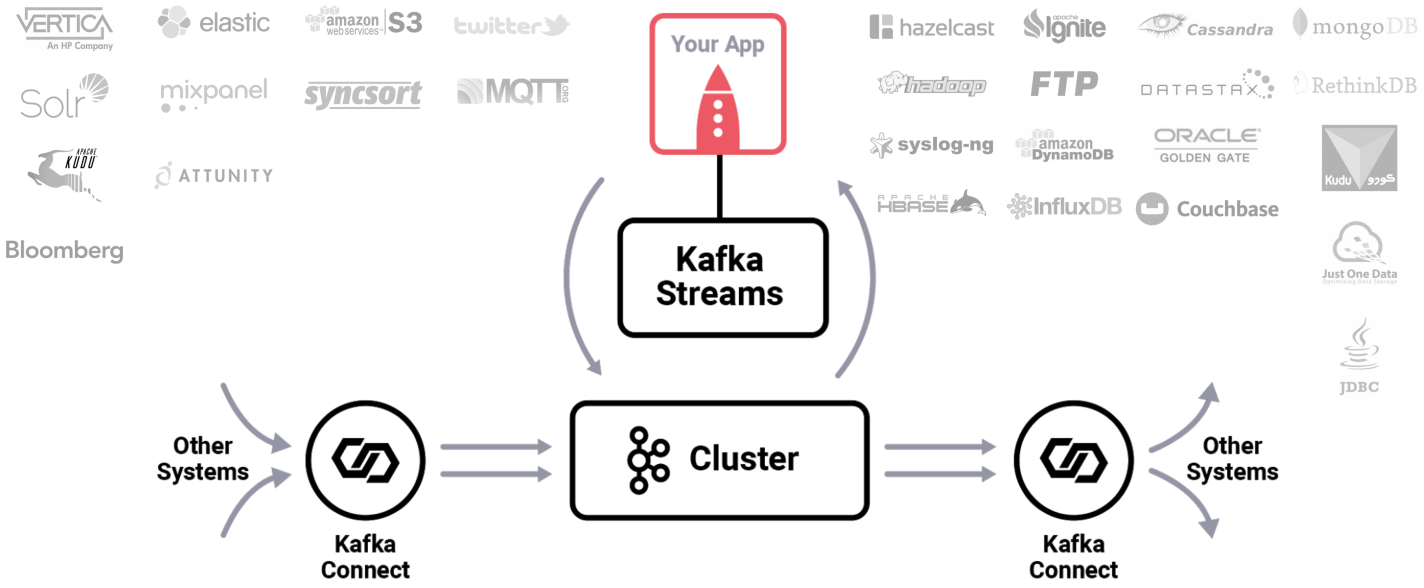
Better Monitoring Options:

- User level usage metrics available to Cloud Enterprise customers via Kafka API

Confluent Cloud Compliance:

- Now on AWS/GCP: SOC I, II, II, PCI phase 1(requires message level encryption), and HIPAA
- Confluent Cloud is also GDPR ready

Broad Connector Eco-System



We have introduced Kafka Connect in a previous module. Kafka Connect is a framework that is part of Apache Kafka, and thus fully open source. But Confluent doesn't stop there. Let's first recap, before we dive into the extras that our company provides.

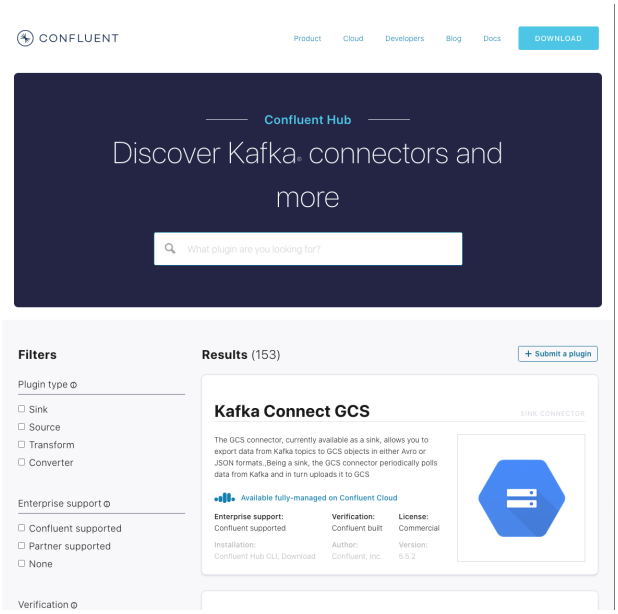
Kafka Connect, a part of the Apache Kafka project, is a standardized framework for handling import and export of data from/to Kafka. This framework can address a variety of use cases, makes adopting Kafka much simpler for users with various data stores; encourages an ecosystem of tools for integration of other systems with Kafka using a unified interface; and provides a better user experience, guarantees, and scalability than other frameworks that are not Kafka-specific.

Kafka Connect is a pluggable framework for inbound and outbound connectors. It's fully distributed and integrates Kafka with a number of sources and sinks. This approach is operationally simpler, particularly if you have a diverse set of other systems you need to copy data to/from as it can be holistically and centrally managed. It also pushes a lot of the hard work of scalability into the framework, so if you want to work in the Kafka Connect framework you would only have to worry about getting your data from a source into the Kafka Connect format – scalability, fault tolerance, etc would be handled intrinsically.

Most of the data systems illustrated here are supported with both Sources and Sinks (including JMS, JDBC, and Cassandra).

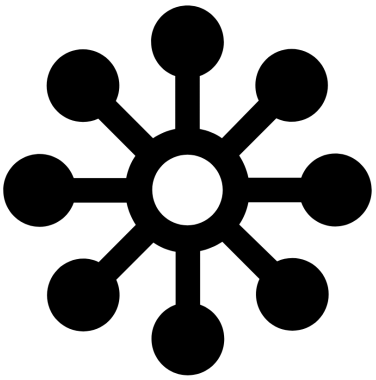
Confluent Hub

Discover and use trusted sources and sinks from Confluent, our partners, and the community



Confluent Hub is an online library of pre-packaged and ready-to-install extensions or add-ons for Confluent Platform and Apache Kafka. You can browse the large ecosystem of connectors, transforms, and converters to find the components that suit your needs and easily install them into your local Confluent Platform environment.

Confluent Hub



GOAL:

Connect Everything!

Confluent has the ambitious goal of connecting everything with Kafka! All connectors will be available through the Confluent Hub and be fully supported by Confluent or certified partners. Some connectors are and will be contributed and supported by the community.

Module Review

Question:

Assuming you are going to use Confluent Control Center in your company or team, what are 3 to 4 main benefits of the investment?

Hands-On Lab

Refer to the lab **05 - The Confluent Platform** in the Exercise Book.

07 Conclusion

Course Review

You are now able to:

- Explain the concepts of **Topics** and **Partitions**
- List the main responsibilities of a **Broker**
- Describe how a **Producer** works
- Setup a minimal **Kafka Cluster**
- Explain the main differences between **Stream** and **Batch** Processing

---

Verify for yourself that you are now really able to successfully perform all tasks listed above. If not, please feel free to contact us on education-support@confluent.io or seek for help on the numerous Confluent Community Slack channels.

Other Confluent Training Courses

- Apache Kafka® Administration by Confluent
- Confluent Developer Skills for Building Apache Kafka®
- Stream Processing using ksqlDB & Apache Kafka® Streams
- Confluent Advanced Skills for Optimizing Apache Kafka®

For more details, see http://confluent.io/training

- **Apache Kafka® Administration by Confluent** covers:
  - Data Durability in Kafka
  - Replication and log management
  - How to optimize Kafka performance
  - How to secure the Kafka cluster
  - Basic cluster management
  - Design principles for high availability
  - Inter-cluster design
- **Confluent Developer Skills for Building Apache Kafka** covers:
  - How to write Producers to send data to Kafka
  - How to write Consumers to read data from Kafka
  - How the REST Proxy supports development in languages other than Java
  - Common patterns for application development
  - How to use the Schema Registry to store Avro, Protobuff, and JSON data in Kafka
  - Basic Kafka cluster administration
  - How to write streaming applications with Kafka Streams API and KSQL

Other Confluent Training Courses

- **Stream Processing using ksqlDB & Apache Kafka Streams** covers:

- Installing ksqlDB containerized and natively
- Transforming and aggregating data streams with KSQL
- Writing Kafka Streams App using DSL and Processor API
- Testing a Kafka Streams App
- Monitoring Kafka Streams and ksqlDB Applications
- Securing Kafka Streams and ksqlDB Applications
- Scaling Kafka Streams and ksqlDB Applications

- **Confluent Advanced Skills for Optimizing Apache Kafka** covers:
  - Monitoring all components of the Confluent Event Streaming Platform (CP)
  - Troubleshooting of all components of CP
  - Tuning the components of CP

Confluent Certified Administrator for Apache Kafka

Duration: 90 minutes

Qualifications: Solid understanding of Apache Kafka and Confluent products, and 6-to-9 months hands-on experience

Availability: Live, online, 24-hours per day!

Cost: $150

Register online: `www.confluent.io/certification`

Benefits:

- Recognition for your Confluent skills with an official credential
- Digital certificate and use of the official Confluent Certified Operator Associate logo

Exam Details:

- The exam is linked to the current Confluent Platform version
- 55 multiple choice questions in 90 minutes
- Designed to validate professionals with a minimum of 6 - 12 months of Confluent experience
- Remotely proctored on your computer
- Available globally in English

Confluent Certified Developer for Apache Kafka

Duration: 90 minutes

Qualifications: Solid understanding of Apache Kafka and Confluent products, and 6-to-9 months hands-on experience

Availability: Live, online, 24-hours a day!

Cost: $150

Register online: `www.confluent.io/certification`

Benefits:

- Recognition for your Confluent skills with an official credential
- Digital certificate and use of the official Confluent Certified Developer Associate logo
Exam Details:
- The exam is linked to the current Confluent Platform version
- 55 multiple choice questions in 90 minutes
- Designed to validate professionals with a minimum of 6 - 12 months of Confluent experience
- Remotely proctored on your computer
- Available globally in English

Thank You

- Thank you for attending the course!
- Please complete the course survey.
- Feedback to: training-admin@confluent.io

- your instructor will give you details on how to access the survey
- If you have any further feedback, please email training-admin@confluent.io