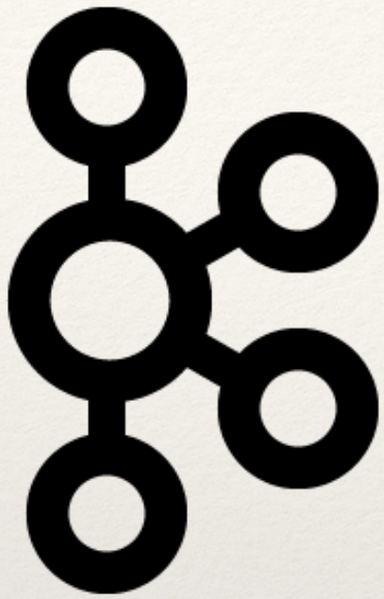


APACHE kafka®

A distributed streaming platform

Introduction to

Apache Kafka

— Nandakumar Purohit

Apache Kafka

What will you learn?

- What is Kafka?
- History
- Who's Who with Kafka
- Core Architecture & Architectural Components
- Message Topics
- Message Partitions
- Replication & Replicas
- Kafka Ecosystem

Apache Kafka

What is Kafka

- **LinkedIn built Kafka** as a **distributed, fault-tolerant** publish/subscribe system.
- It records messages organized into topics.
- All **messages are stored as logs** to persistent filesystems.
- Kafka is a **Write-Ahead Logging (WAL)** system that writes all published messages to log files before making it available for consumer applications.
- Kafka was built with the following goals in mind:
  - Loose coupling between Message Producers and Message Consumers
  - Persistence of message data to support a **variety of data consumption** scenarios and **failure handling**
  - **Maximum** end-to-end throughput with **low latency components**
  - Managing **diverse data formats** and types using binary data formats
  - Scaling servers linearly **without affecting** the **existing cluster setup**

Apache Kafka

History of Kafka

"Necessity is the mother of Invention!"

- **High Volume**
- Over **0.5 Trillion messages** per day
- **175 TB** of data per day
- **650 TB** of messages consumed per day
- Over **433 Million** Users

- **High Velocity**
- **Peak 13 Million messages** per second
- **2.75 GB** per second

- **High Variety**
- Multiple RDBMS (Oracle, MySQL, etc.,)
- Multiple NoSQL (Espresso, Voldemort, etc.,)

Apache Kafka

Who's Who with Kafka

	❖ For activity stream data and operational metrics
	❖ Real-time monitoring and event-processing pipeline
	❖ Real-time driver-rider matching, Uber-EATS, Share my ETA
	Used in our event pipeline, exception tracking

❖

Apache Kafka

Who's Who with Kafka

Apache Kafka Adoption

7X since 2015

Yahoo	Uber	Square	Airbnb
Etsy	Oracle	Coursera	Spotify
Microsoft	Goldman Sachs	IBM	Ancestry
Bing	Netflix	Pintrest	LinkedIn
Mailchimp	PayPal	Twitter	Hotels.com

Apache Kafka

The Meaning

Franz Kafka

kaf•ka•esque /’káf, kə, ɛsk/ | adjective

Basically it describes a nightmarish situation which most people can somehow relate to, although strongly surreal.

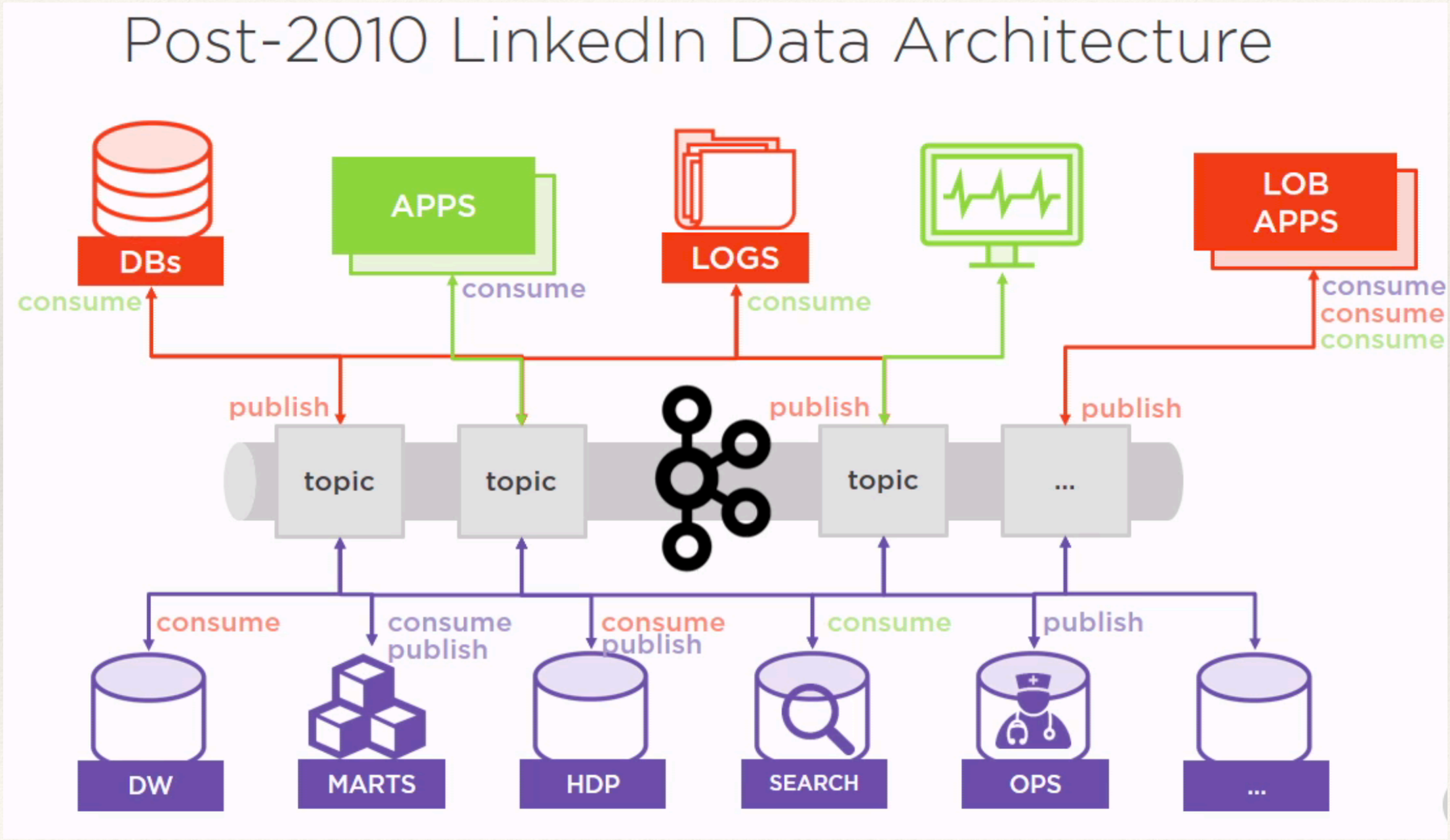
synonyms: surreal, lucid, spoilsbury toast boy

Usage: “Whoa! This flick is way kafkaesque...”

Source: Urban Dictionary

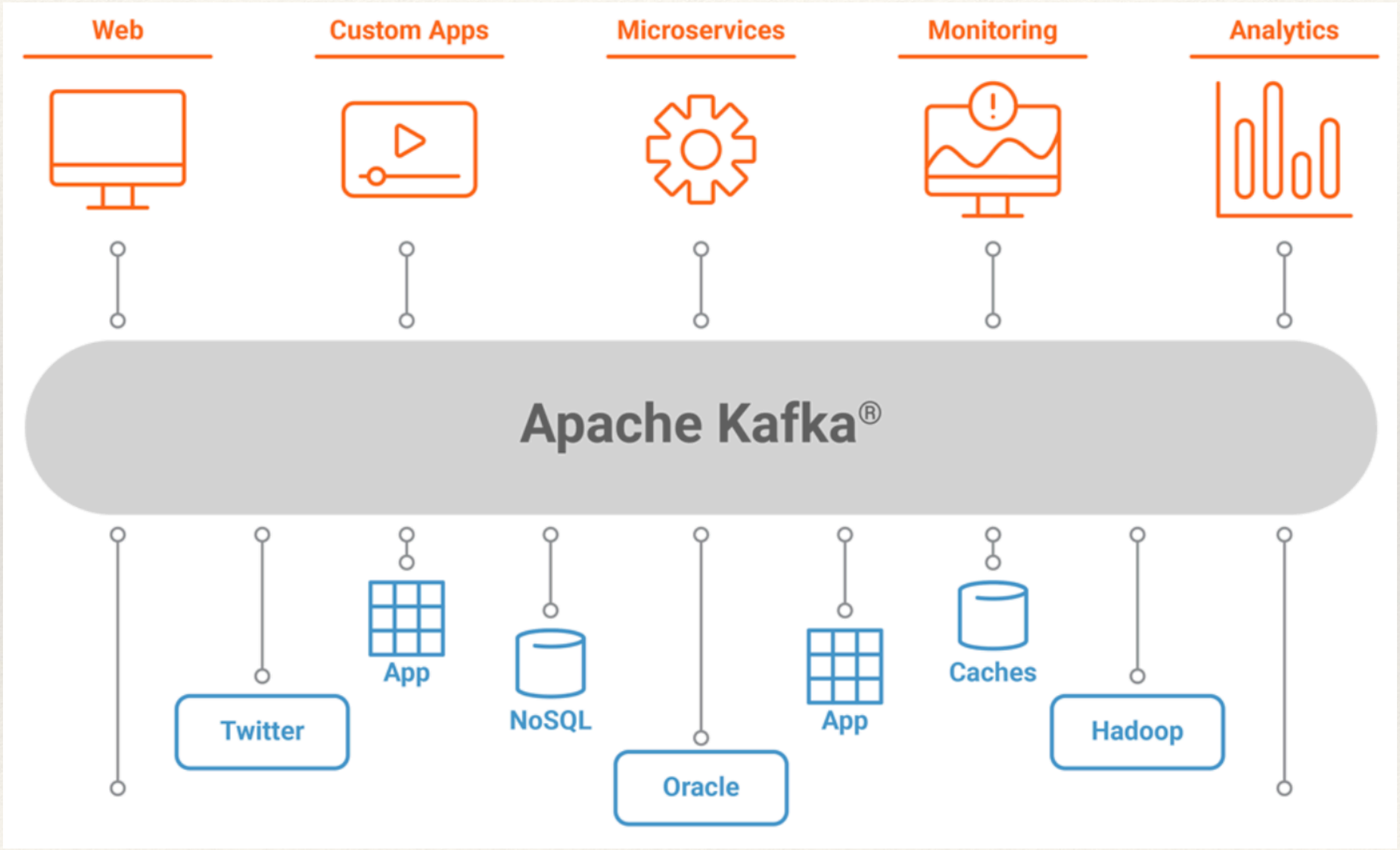
Apache Kafka

LinkedIn Apps with Kafka



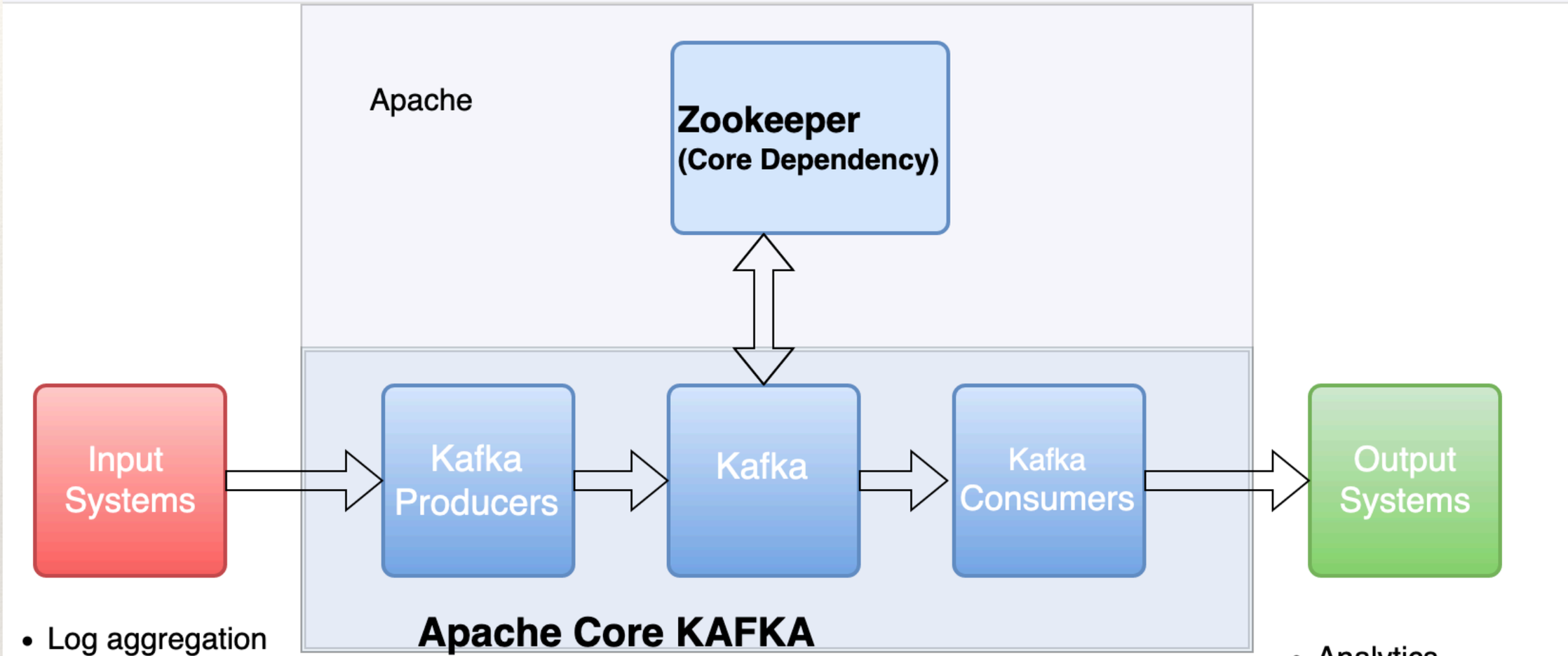
Apache Kafka

Sample Architectures with Kafka



Apache Kafka

Core Architecture



- Log aggregation
- Metrics
- KPIs
- Batch imports
- Audit trail
- User activity logs
- Web logs

Not part of core

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

Apache Kafka Core

- Server/Broker
- Scripts to start libs
- Script to start up Zookeeper
- Utils to create topics
- Utils to monitor stats

- Analytics
- Databases
- Machine Learning
- Dashboards
- Indexed for Search
- Business Intelligene

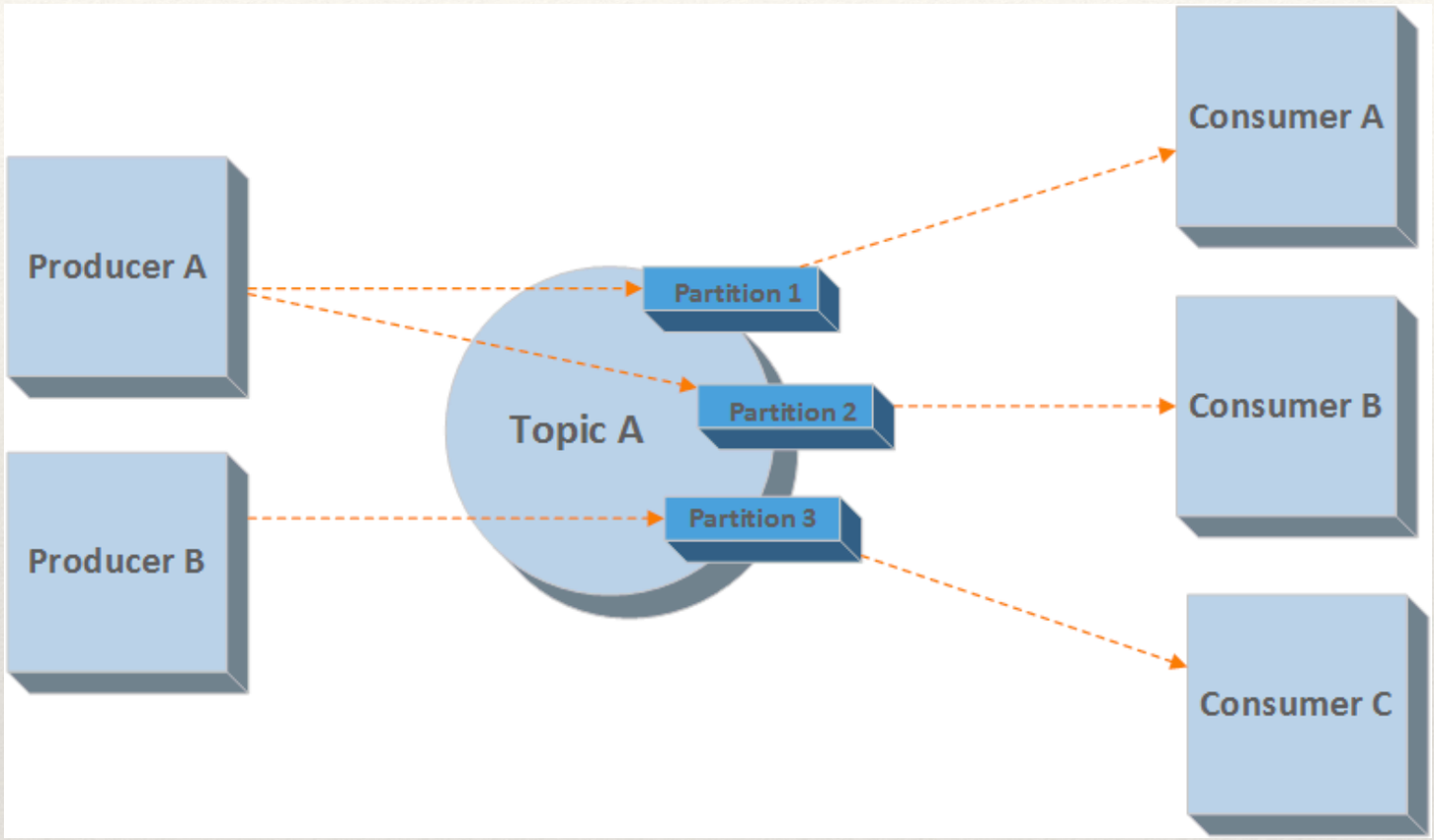
Apache Kafka

Core Architecture

- Every message in **Kafka topics** is a **collection of bytes**. This collection is represented as an **array**.
- **Producers** are the applications that store information in **Kafka queues**.
- Every topic is further differentiated into **partitions**.
- Each **partition** stores messages in the **sequence** in which **they arrive**.
- There are two major operations that producers/consumers can perform in Kafka:
  - **Producers** append to the end of the **write-ahead log** files.
  - **Consumers** fetch messages from these log files belonging to a given **topic partition**.
- Each **topic is spread** over different **Kafka brokers**, which host **one or two partitions of each topic**.
- Ideally, **Kafka pipelines** should have a **uniform number of partitions per broker** and all topics on each machine.

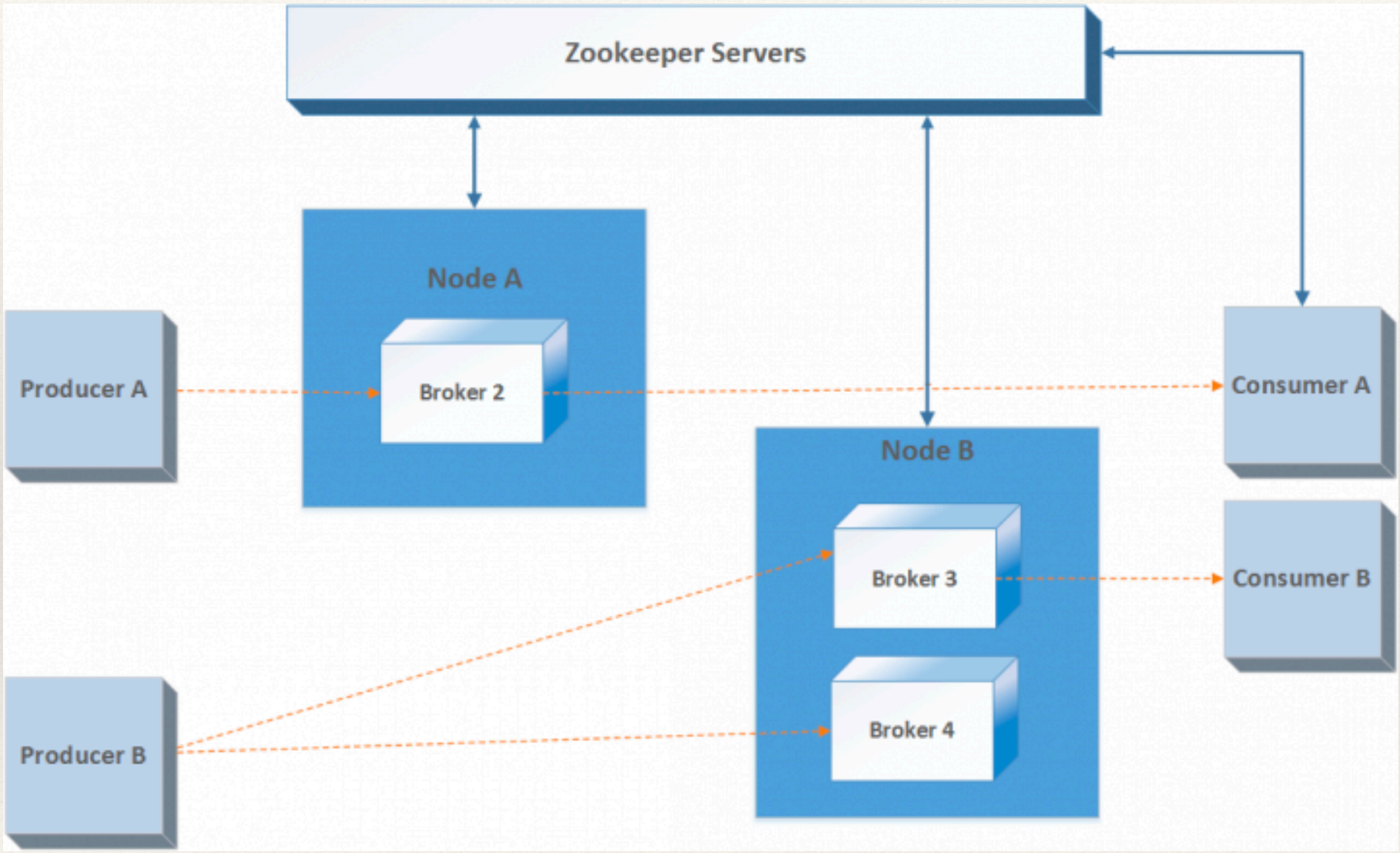
Apache Kafka

Logical Architecture



Apache Kafka

Physical Architecture



Apache Kafka

Physical Architecture

- A **Kafka cluster** is basically composed of **one or more servers (nodes)**
- A typical **Kafka cluster** consists of **multiple brokers**.
- It helps in **load-balancing message reads and writes** to the cluster.
- Each of these **brokers is stateless**.
- However, they use **Zookeeper** to **maintain** their **states**.
- Each **topic partition** has **one of the brokers as a leader** and **zero or more** brokers as **followers**.
- The **leaders manage** any read or write requests for their respective **partitions**.
- **Followers replicate the leader** in the background without actively interfering with the leader's working.
- You should think of **followers as a backup for the leader** and **one** of those **followers** will be chosen as the leader **in the case of leader failure**.

Apache Kafka

Physical Architecture - Zookeeper

- **Zookeeper** is an important component of a **Kafka cluster.**
- It manages and coordinates **Kafka brokers and consumers**.
- Zookeeper keeps **track** of any **new broker** additions or any existing **broker failures** in the **Kafka cluster**.
- Accordingly, it will **notify** the **producer or consumers** of **Kafka queues** about the **cluster state**.
- This helps both producers and consumers in coordinating work with active brokers.
- **Keeps track of Broker Leader information** about **Topic Partitions** and shares with Producers/Consumers for Read/Write

Apache Kafka

Physical Architecture - Zookeeper

- **Producers push data to brokers.**
- At the time of publishing data, **producers search for the elected leader** (broker) of the respective topic partition and automatically send a message to that leader broker server.
- Similarly, the **consumer** reads messages from **brokers**.
- The **consumer records** its **state** with the help of **Zookeeper** as Kafka brokers are stateless. It uses **Partition Offset.**
- This design helps in **scaling Kafka** well.
- The **consumer offset value** is maintained by **Zookeeper**.
- Once message offset is sent by Consumer to Zookeeper, it means that the consumer has consumed all prior messages.

Apache Kafka

Message Topics

Important Terminologies in Kafka	
Retention Period	• The messages in the topic need to be stored **for a defined period** of time to save space irrespective of throughput. • We can configure the retention period, which is **by default seven days**, to whatever number of days we choose. • Kafka keeps messages up to the defined period of time and then ultimately deletes them.
Space Retention Policy	• We can also configure Kafka topics to clear messages when the size reaches the threshold mentioned in the configuration. • This is only when there is improper capacity planning
Offset	• Each message in Kafka is assigned with a number called as an offset. • Consumers acknowledge messages with an offset.
Partitions	• Each Kafka topic consists of a fixed number of partitions. • During topic creation in Kafka, you need to configure the number of partitions. • Partitions are distributed and help in achieving high throughput.
Compaction	• Topic compaction was introduced in **Kafka 0.8** • Consumers may receive messages with **same key but with changes** • Consumer want to **process ONLY the latest data** • Compacting **all messages** with the **same key** and creating a **map offset for key: offset**. • It helps in removing duplicates from a large number of messages.

Apache Kafka

Message Topics

Important Terminologies in Kafka	
Leader	**Partitions are replicated across the Kafka cluster** based on the replication factor specified. Each partition has a leader broker and followers and all the read write requests to the partition will go through the leader only. If the leader fails, another leader will get elected and the process will resume.
Buffering	Kafka **buffers messages** both at the **producer and consumer side** to increase throughput and reduce Input/Output (IO)

Apache Kafka

Message Partitions

Think of a Data Table with Purchase History of Items

The concept of **units of parallelism** is used when **topics** are broken into **partitions**

"The greater the number of partitions, the more throughput"

While creating topics you can always mention the number of partitions that you require for a topic.

Each of the **messages** will be appended to **partitions** and each message is then assigned with a number called an **offset**.

Kafka makes sure that messages with **similar keys** always go to the **same partition**; it calculates the **hash of the message key** and appends the message to the partition.

Time ordering of messages is not guaranteed in topics but within a partition, its always

Apache Kafka

Message Partitions

Pros & Cons of Large Number of Partitions	
High Throughput	• Partitions are a way to achieve parallelism in Kafka. • Write operations on different partitions happen in parallel. • All the time-consuming operations will happen in parallel as well; this operation will utilize hardware resources at the maximum.
Increases Producer Memory	• The producer buffers incoming messages per partition. • Once the upper bound or the time set is reached, the producer sends his messages to the broker and removes it from the buffer. • If we increase the number of partitions, the memory allocated for the buffering may exceed in a very short interval of time and the producer will block producing messages until it sends buffered data to the broker. • This may result in lower throughput.

Apache Kafka

Message Partitions

Pros & Cons of Large Number of Partitions

High Availability Issue

- Brokers in Kafka store thousands of partitions of different topics.
- Reading and writing to partitions happens through the leader of that partition.
- Generally, if the leader fails, electing a new leader takes only a few milliseconds. Observation of failure is done through controllers. Controllers are just one of the brokers.
- Before serving the request, Leader reads metadata of the partition from Zookeeper.
- For normal and expected failure, the window is very small and takes only a few milliseconds.
- In the case of unexpected failure, such as killing a broker unintentionally, it may result in a delay of a few seconds based on the number of partitions.

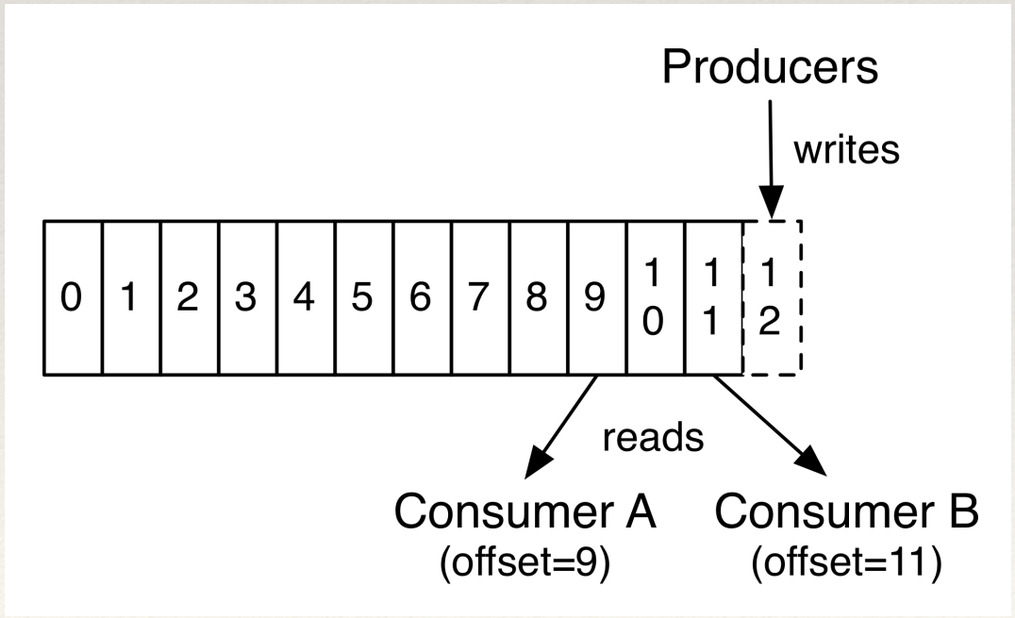
Apache Kafka

Message Partitions

Pros & Cons of Large Number of Partitions

- The other possibility could be that the failed broker is a controller, the controller replacement time depends on the number of partitions.
- The new controller reads the metadata of each partition, and the time to start the controller will increase with an increase in the number of partitions.

Delay Time = (Number of Partition/replication * Time to read metadata for single partition)



Apache Kafka

Replication & Replicated Logs

- **Replication** is one of the most important factors in achieving reliability for Kafka systems.
- **Replicas of message logs for each topic partition** are maintained across different servers in a Kafka cluster.
- You can have **varied replications** for each topic.
- Generally, **followers keep a copy of the leader's log**, which means that the **leader does not commit** the message until it receives **acknowledgment from all the followers**.
- There are different ways that the log replication algorithm has been implemented; it should ensure that, if leader tells the producer that the message is committed, it must be available for the consumer to read it.

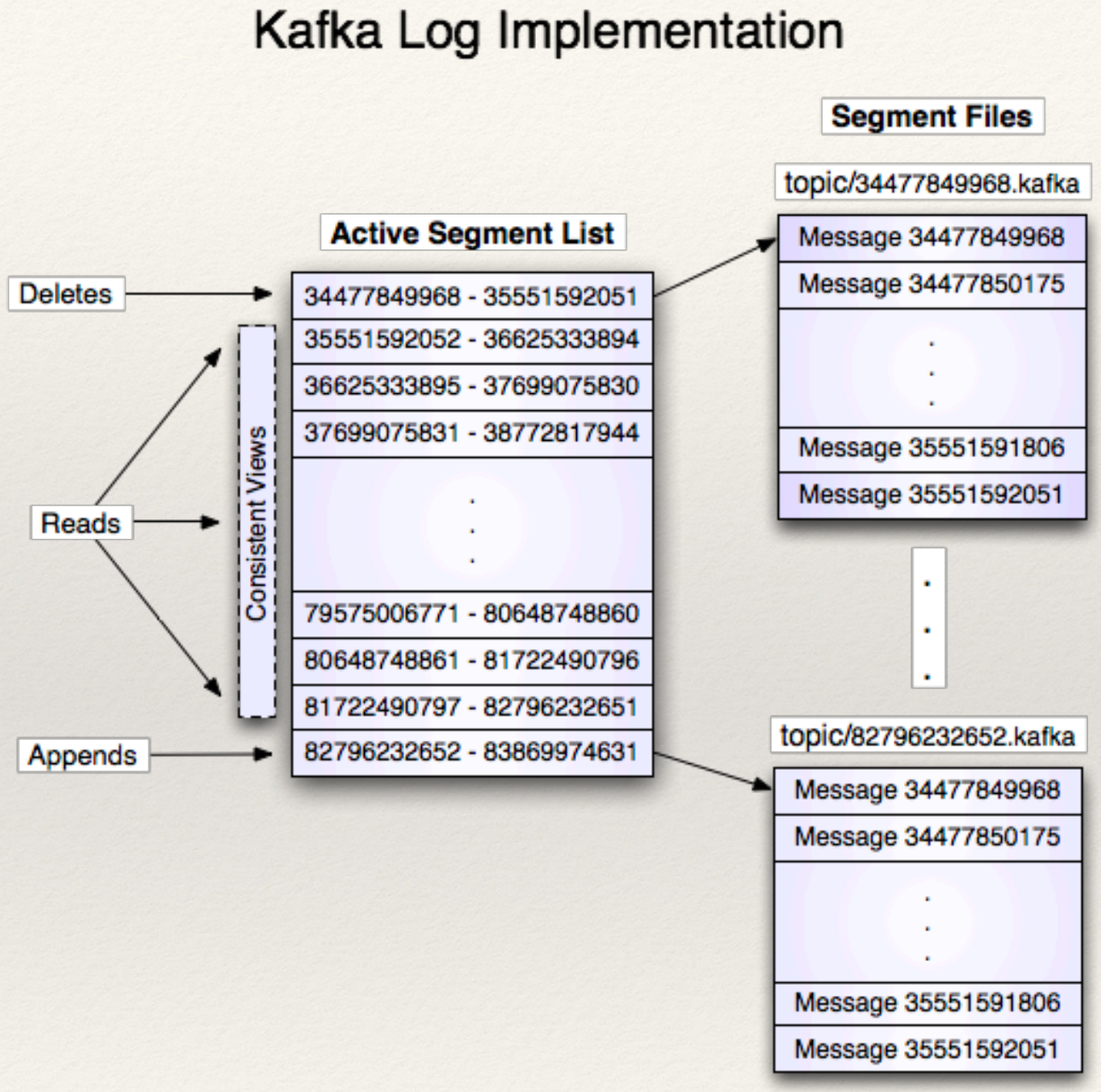
Apache Kafka

Replication & Replicated Logs

Approaches for Replicas	
Quorum-based approach	• In this approach, the leader **will mark messages committed** only when the **majority of replicas have acknowledged** receiving the message. • If the leader fails, the election of the new a leader will only happen with the coordination between followers. • **Zookeeper** follows a **quorum-based approach** for leader election.
Primary Backup Approach	• **Kafka** follows a different approach to **maintaining replicas** • The **leader** in Kafka **waits** for an **acknowledgement from all the followers** before marking the message as committed. • If the leader fails, any of the followers can take over as leader.

Apache Kafka

Log Implementation



Apache Kafka

Message Producers

- In Kafka, the producer is responsible for sending data to the partition of the topic for which it is producing data.
- The producer generally does not write data to partitions, it creates write requests for messages and sends them to the leader broker.
- Partitioner calculates the hash value of the message, which helps the producer to choose which partition should be selected.
- The message with a null key will be distributed in a round-robin fashion across partitions to ensure even distribution of messages.
- In Kafka, each partition has a leader and each read write request goes through the leader only. So a request to write messages to a partition of a topic will go through the leader broker.
- The producer waits for an acknowledgement of messages depending on the setting. Generally, it waits until the replication for a particular message is successfully acknowledged.
- Remember that until and unless all replicas have been acknowledged to commit the message, it will not be available to read. This setting is the default and ensures that a message cannot be lost if a leader broker fails.

Apache Kafka

Kafka Producer Internals

Resopnsibilties of Kafka Producers	
Bootstrapping Kafka Broker URLs	• The **Producer connects to at least one broker** to fetch metadata about the **Kafka cluster.** • It may happen that the **first broker** to which the producer wants to connect **may be down.** • To ensure a **failover**, the producer implementation takes a **list** of more than one **broker URL** to bootstrap from. • Producer **iterates through a list of Kafka** broker addresses until it finds the one to connect to **fetch cluster metadata**.
Data Serialization	• Kafka uses a **binary protocol** to send and receive data over TCP. • This means that while writing data to Kafka, **producers** need to send the **ordered byte sequence** to the defined **Kafka broker's** network port. • Subsequently, it will **read** the **response byte sequence** from the **Kafka broker** in the same ordered fashion. • Kafka producer **serializes** every message data object into **ByteArrays** before sending any record to the **respective broker** over the wire. • Similarly, it converts any byte sequence received from the broker as a response to the message object.

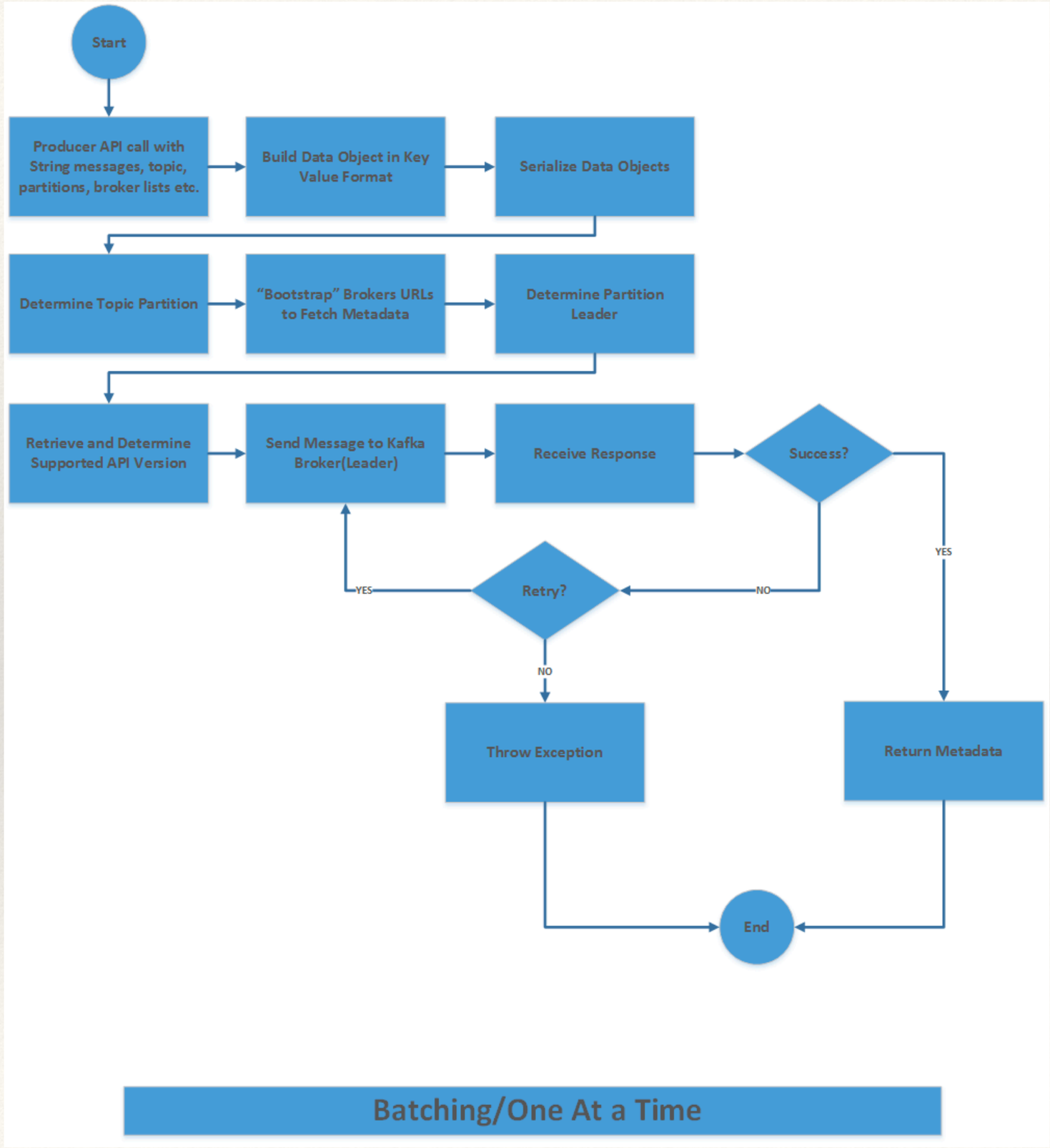
Apache Kafka

Kafka Producer Internals

Resopnsibilties of Kafka Producers	
Bootstrapping Kafka Broker URLs	• It is the responsibility of the Kafka **producer** to determine **which topic partition** the data needs to be sent. • If the partition is specified by the caller program, then Producer APIs do not determine topic partition and send data directly to it. • Selecting partition is based on the **key of the message** data object.
Data Serialization	• Producers send data to the leader of the partition directly. • Producers ask for metadata from any of the Kafka brokers. • Brokers answer the request for metadata about active servers and leaders of the topic's partitions at that point of time.
Failure handling / Retry Ability	Producer application can configure the number of retries through Producer API configuration. Exception handling should be done through the producer application component.
Batching	• Batching ensures reduced I/O and optimum utilization of producer memory. • While deciding on the number of messages in a batch, keep in mind the end-to-end latency. • End-to-end latency increases with the number of messages in a batch.

Apache Kafka

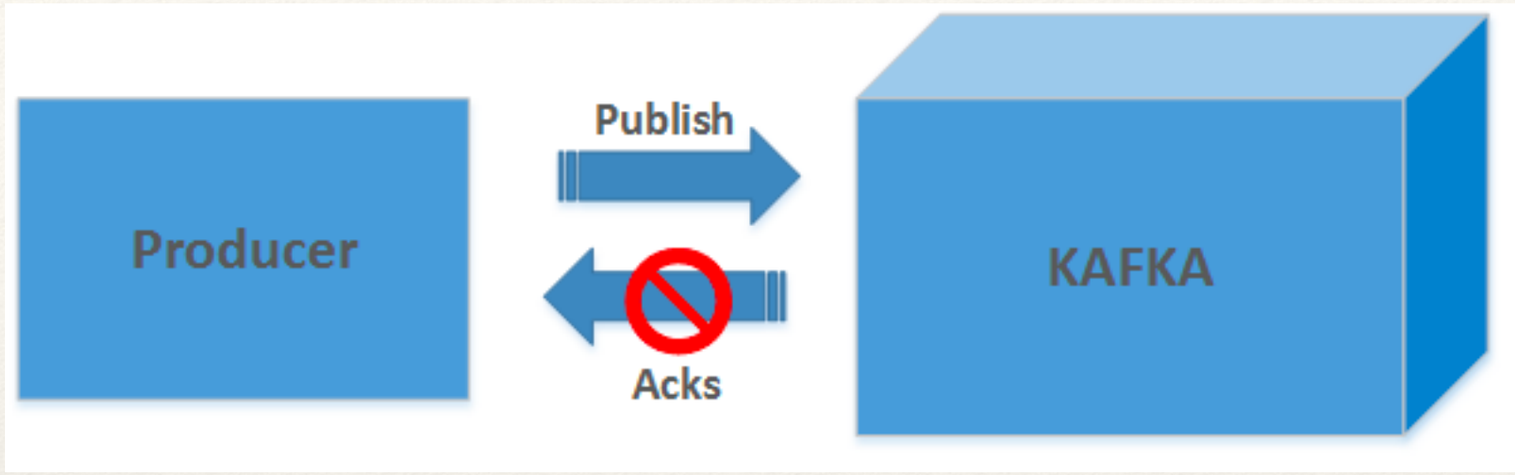
Kafka Producer Internals



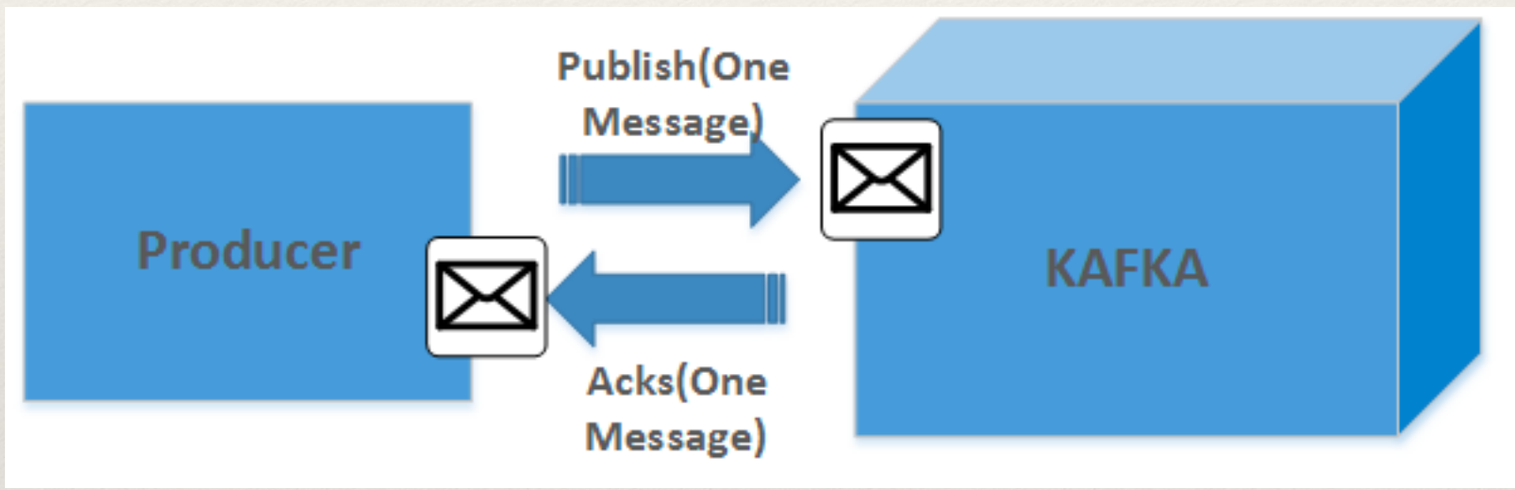
Apache Kafka

Common Message Publishing Patterns

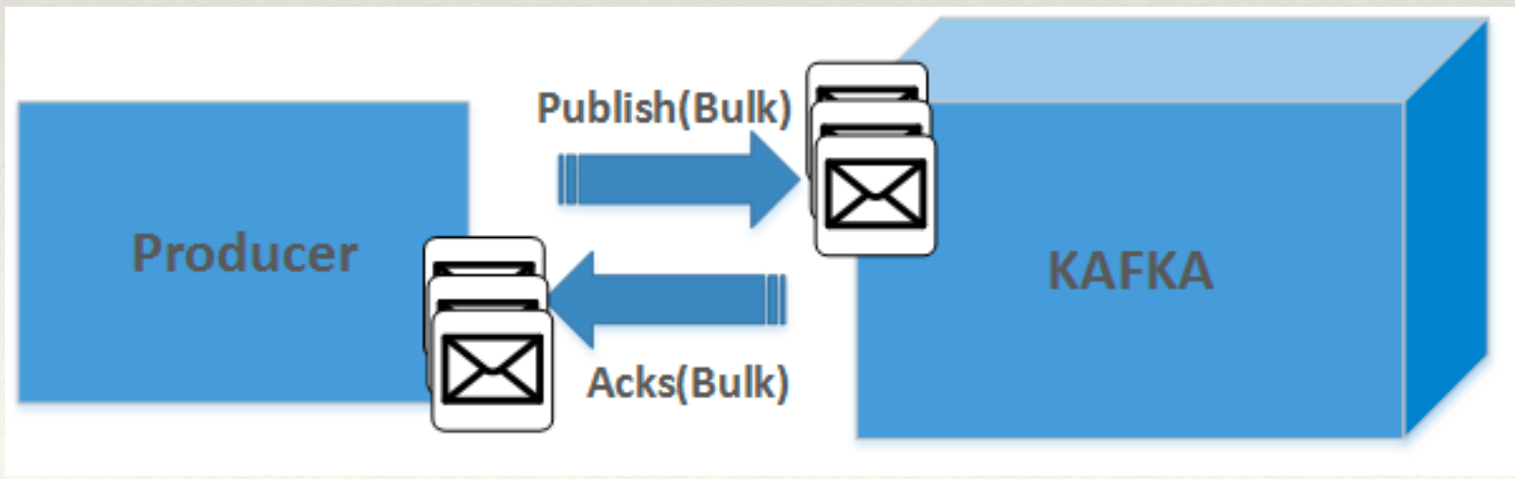
Fire-and-Forget



One-message-transfers



Batching



Apache Kafka

Kafka Consumer Internals

Resopnsibilties of Kafka Consumers	
Subscribing to a Topic	• Consumer operations start with subscribing to a topic. • If consumer is part of a **consumer group**, it will be assigned a **subset of partitions** from that topic. • Consumer process would eventually read data from those **assigned partitions**. • You can think of topic subscription as a registration process to read data from topic partitions.
Consumer Offset Position	• Kafka, unlike any other queues, does not maintain message offsets. • Every consumer is responsible for maintaining its own consumer offset. • Consumer offsets are maintained by consumer APIs and you do not have to do any additional coding for this.
Replay / Rewind / Skip Messages	• Kafka **consumer** has **full control over starting offsets** to read messages from a topic partition. • **Using consumer APIs,** any consumer application can pass the **starting offsets** to read messages from topic partitions. • They can choose to read messages from the beginning or from some specific integer offset value irrespective of what the current offset value of a partition is. • In this way, consumers have the capability of replaying or skipping messages as per specific business scenarios.

Apache Kafka

Kafka Consumer Internals

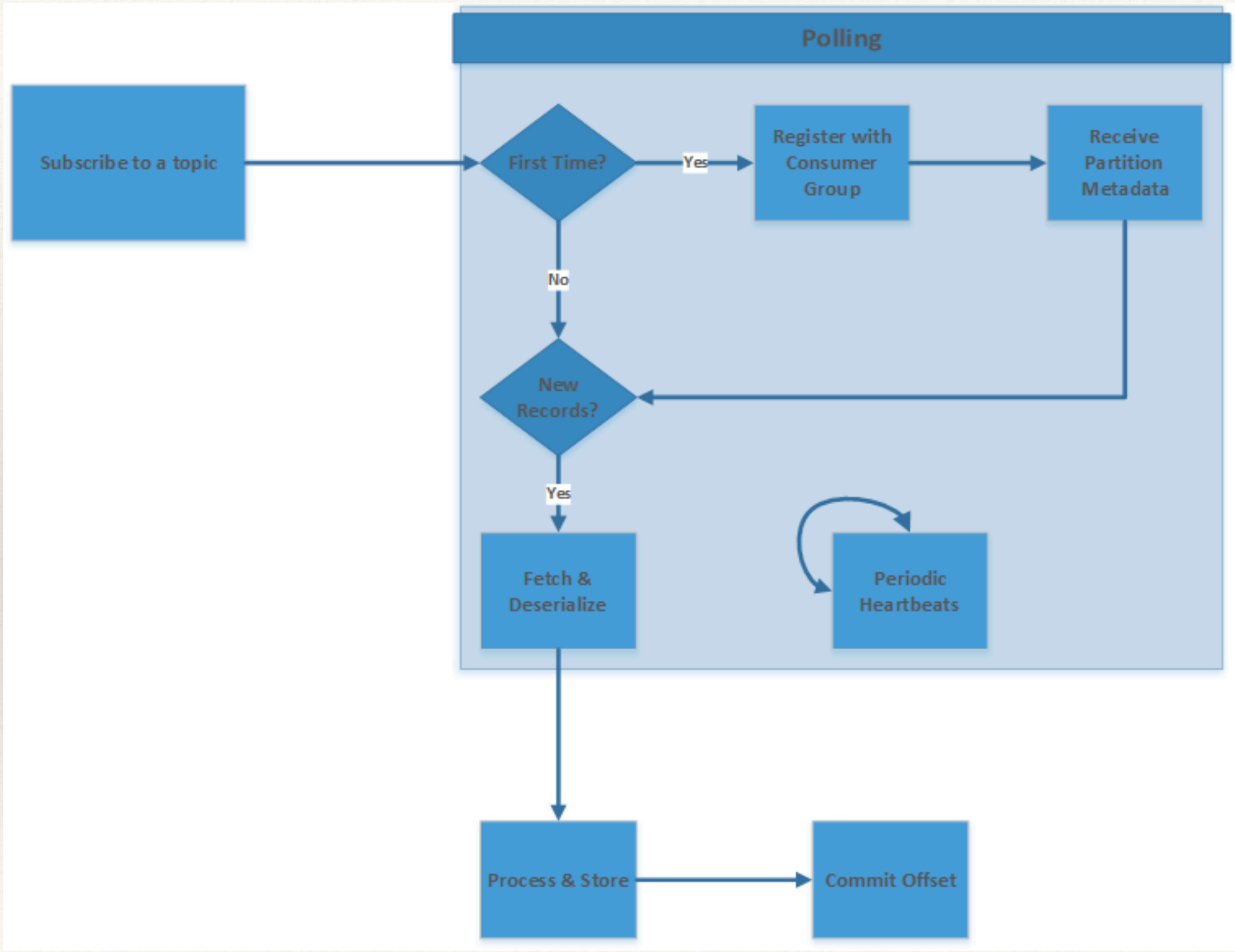
Resopnsibilties of Kafka Consumers	
Heartbeat	• It is the consumer's responsibility to ensure that it **sends regular heartbeat signals to the Kafka broker** (consumer group leader) to confirm their membership and ownership of designated partitions. • If heartbeats are not received by the group leader in a certain time interval, then the **partition's ownership** would be **reassigned** to some other consumer in the **consumer group.**

Consumer Offset Position	• **Kafka does not track positions or offsets** of the messages that are read from consumer applications. It is the responsibility of the **consumer application to track their partition offset** and commit it.
	• **This has two advantages:**
	  • this improves **broker performance** as they do not have to track each consumer offset
	  • this gives **flexibility to consumer applications** in managing their offsets as per their specific scenarios.
	• They can commit offsets after they finish processing a batch or they can commit offsets in the middle of very large batch processing to reduce side-effects of rebalancing.

Deserialization	Consumers deserialize Byte arrays into Java objects

Apache Kafka

Kafka Consumer Internals

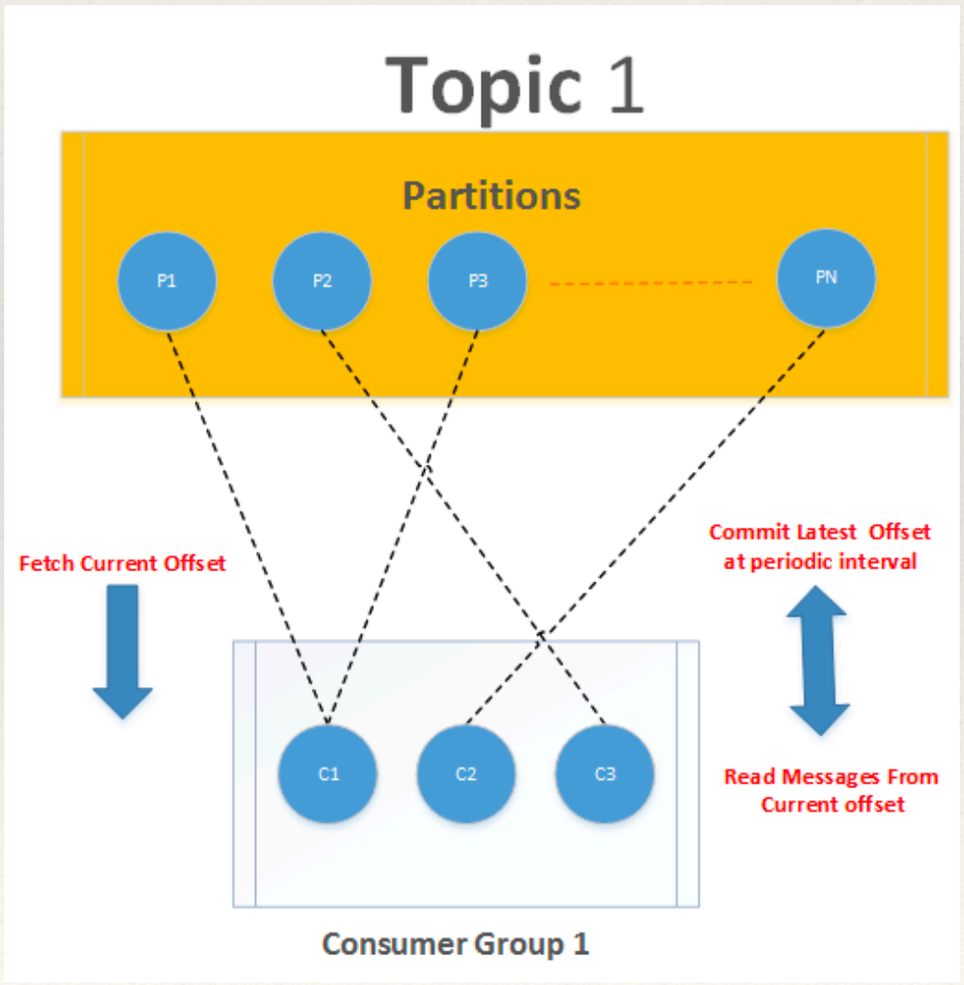


Apache Kafka

Common Message Consuming Patterns

Consumer Group - Continuous Data Processing

- In this pattern, once consumer is created and subscribes to a topic, it **starts receiving messages from the current offset.**
- The consumer **commits the latest offsets based on the count of messages received** in a batch at a regular, configured interval.
- The **consumer checks whether it's time to commit**, and if it is, it will commit the offsets.
- Offset commit can happen **synchronously or asynchronously**. It uses the **auto-commit** feature of the consumer API.

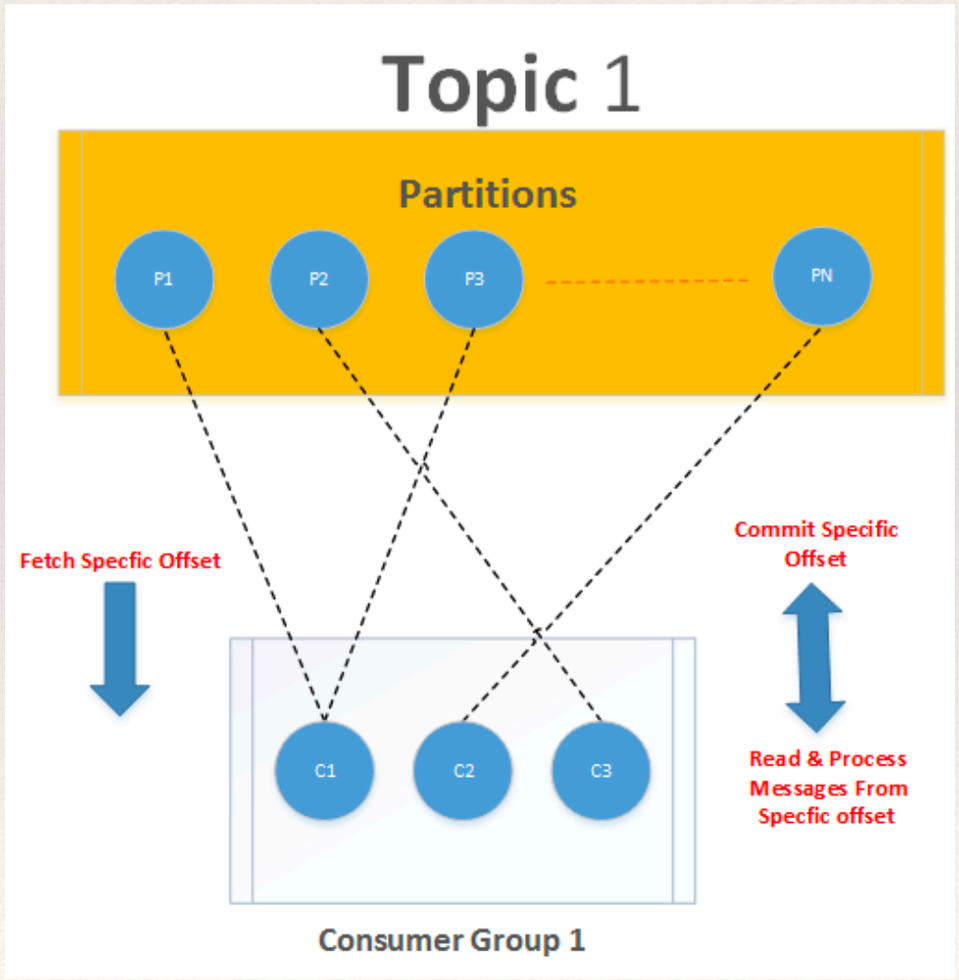


Apache Kafka

Common Message Consuming Patterns

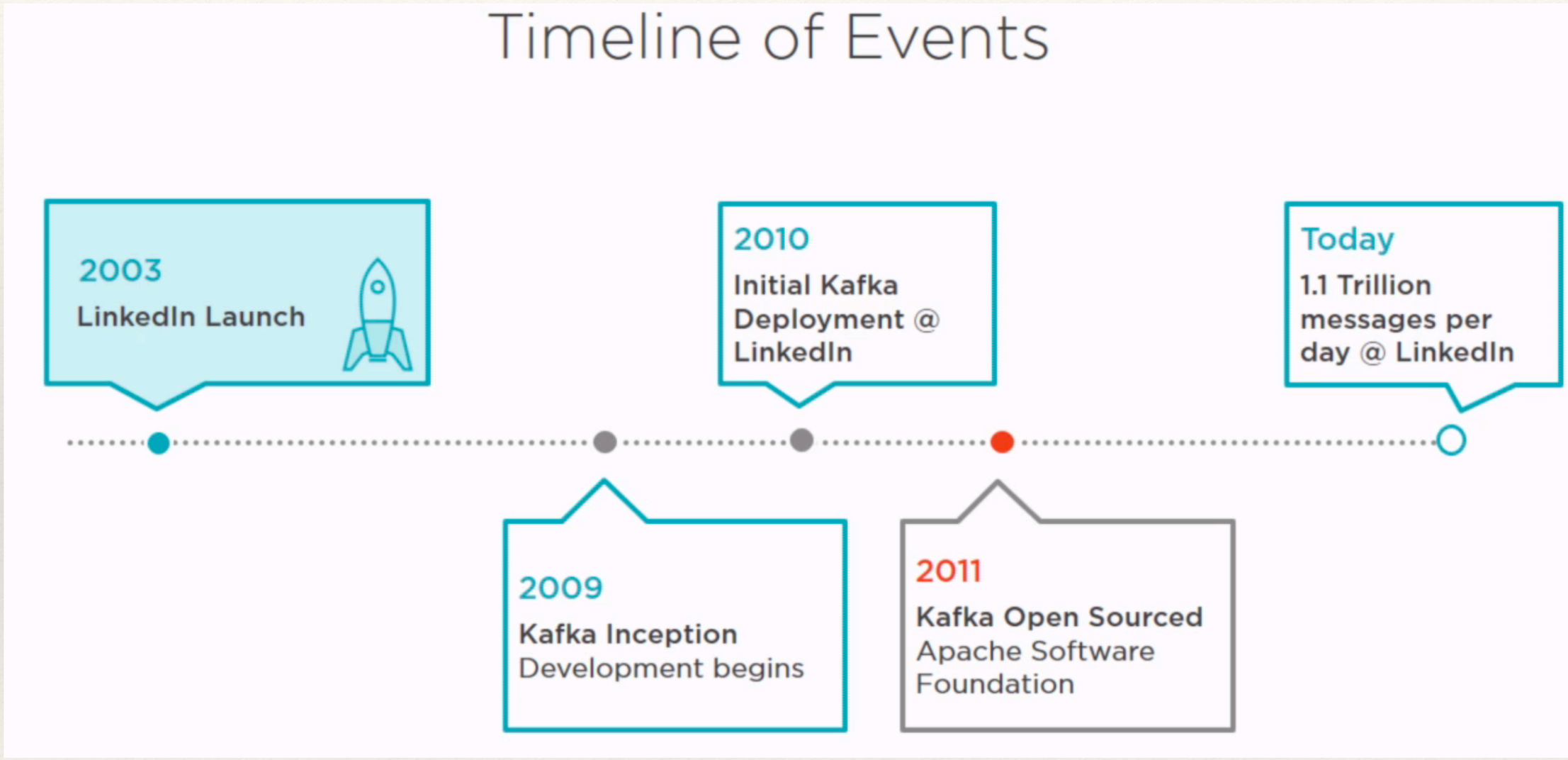
Consumer Group - Discrete Data Processing

- Sometimes you want **more control over consuming messages** from Kafka. You want to **read specific offsets** of messages that may or **may not be the latest current offset** of the particular partition.
- Subsequently, you may want to **commit specific offsets** and not the regular latest offsets.
- In this, consumers fetch data based on the offset provided by them and they commit specific offsets that are as per their specific application requirements.



Apache Kafka

Release History



Apache Kafka

Ecosystem

- *https://cwiki.apache.org/confluence/display/KAFKA/Ecosystem*