



If you're learning Kafka, this article
is for you

Fourteen years ago, LinkedIn's built Kafka to handle its log processing demands.

The system combines the benefits of traditional log aggregators and publish/subscribe messaging systems. Kafka is designed to offer high throughput and scalability.

It provides an API similar to a messaging system and allows applications to consume real-time log events.

Now, you see Kafka everywhere.

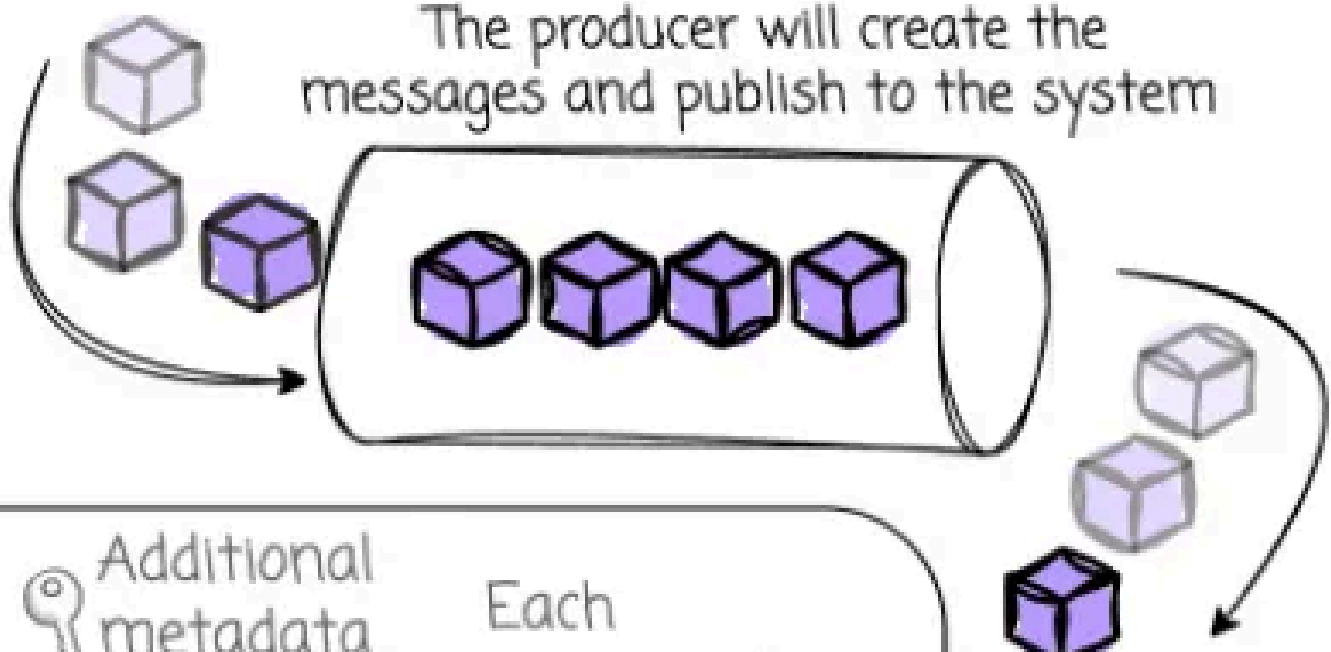
Overview

Messages

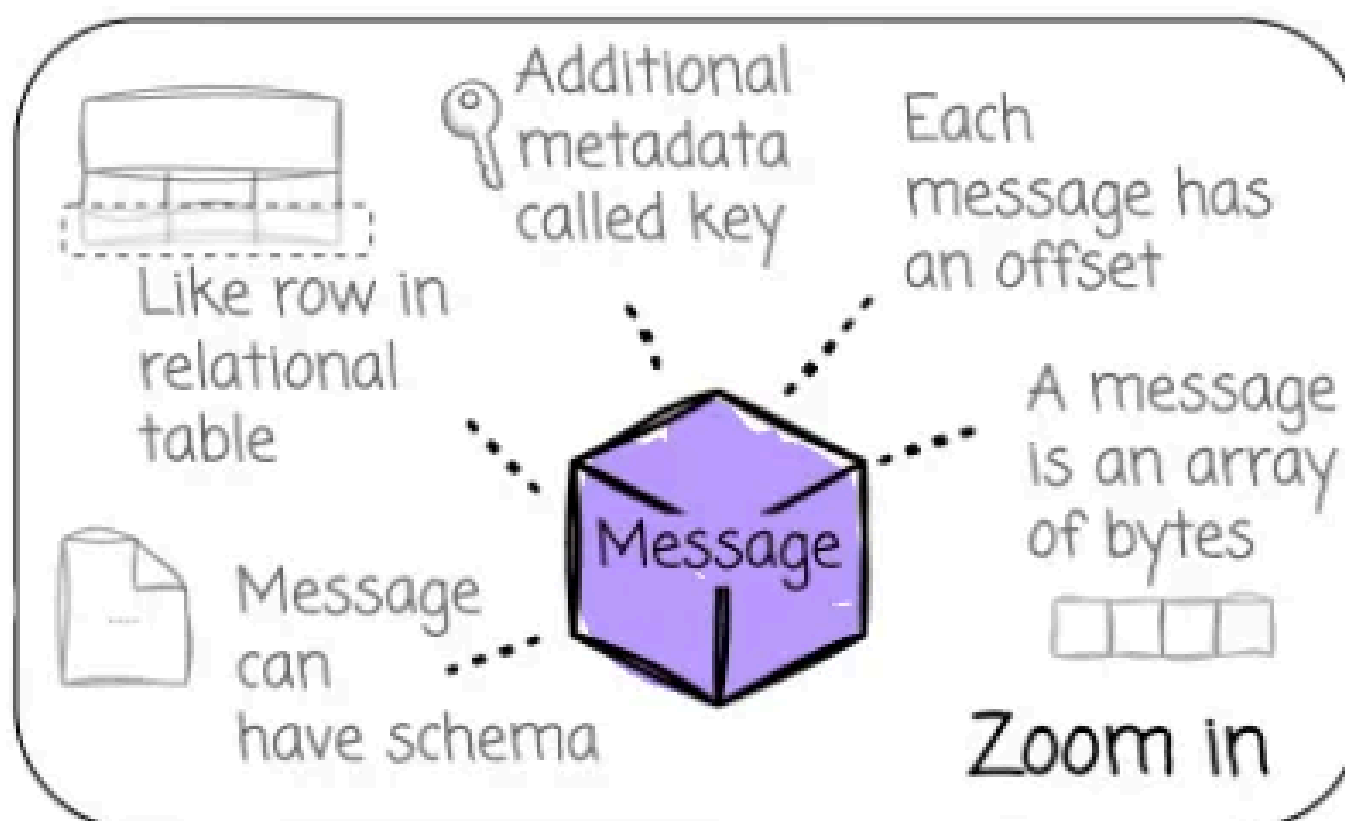
Kafka's unit of data is called a **message**. A message can have an optional piece of metadata called the key. The message and the key are just an **array of bytes**. The key can be used if users want more control in partitioning

Message
is the
unit data
of Kafka

The producer will create the
messages and publish to the system



The
consumers
then
consume
those
message by
pulling from
the system



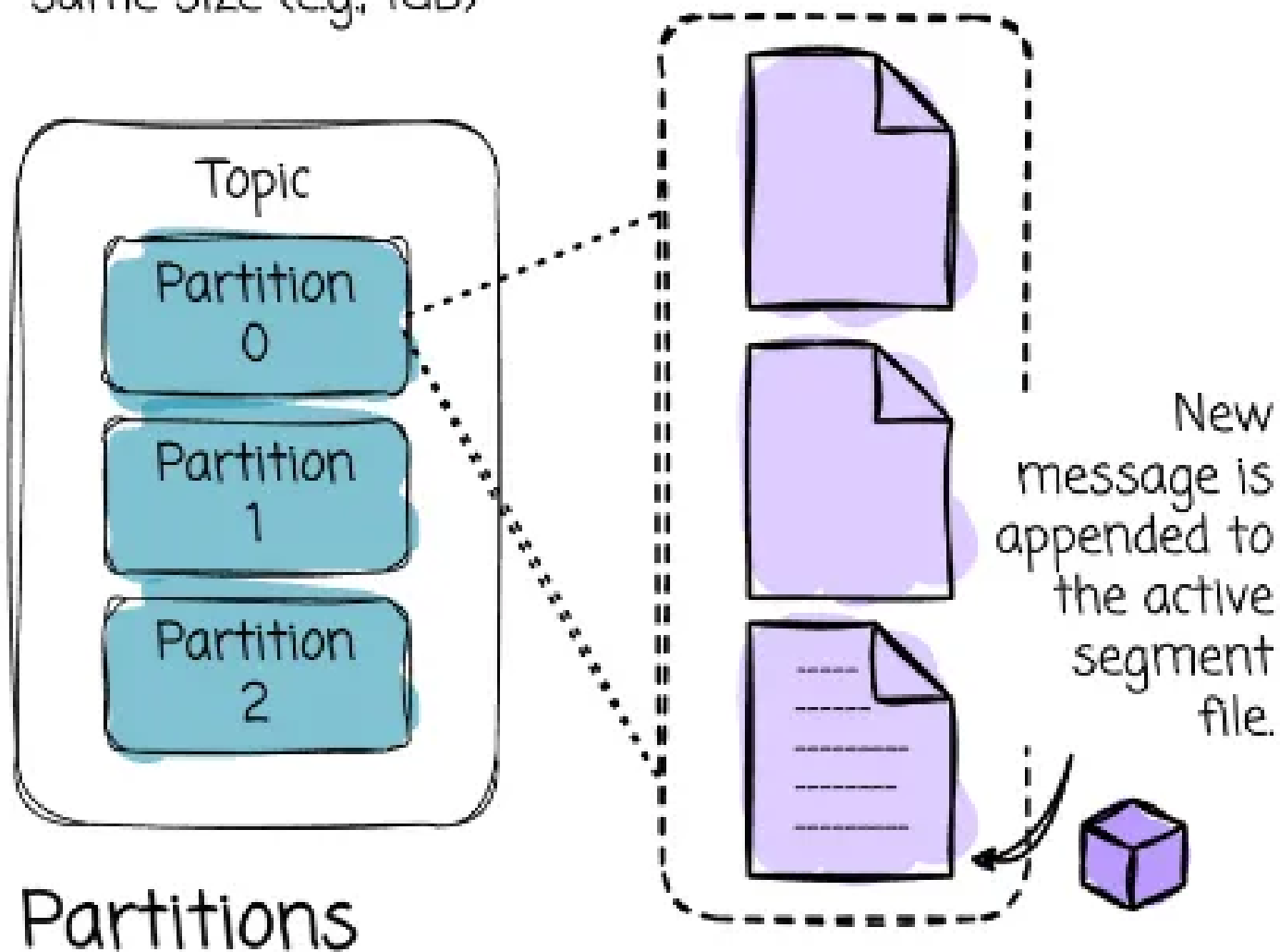
Topics and Partitions

Messages in Kafka are organized into topics.

A topic can be split into multiple partitions.

Each partition of a topic corresponds to a logical log. Physically, a log is implemented as a set of segment files of approximately the same size.

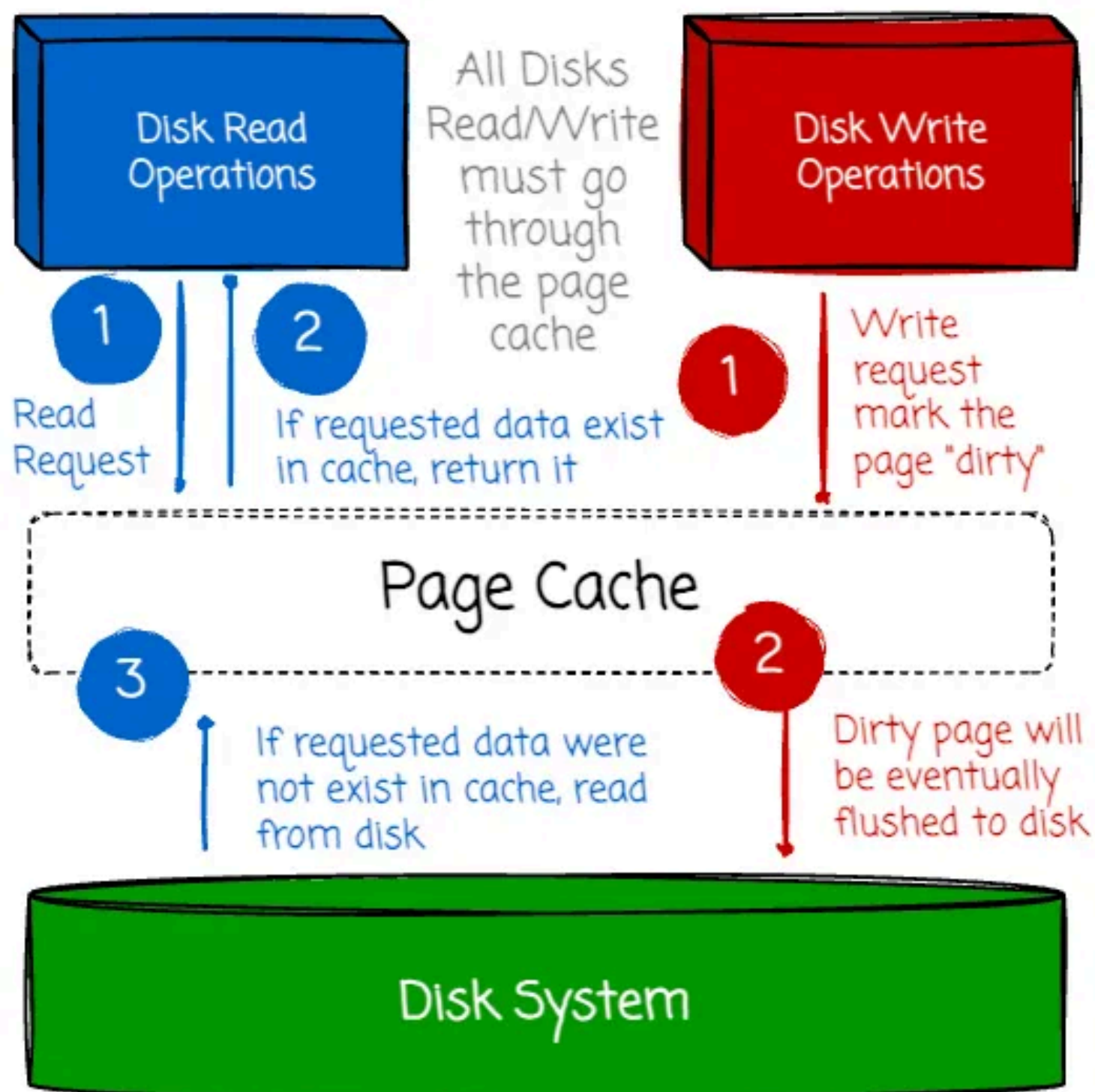
Each topic's partition corresponds to a logical log. A log is implemented as a set of segment files of approximately the same size (e.g., 1GB)



Design

Kafka use the Filesystem

Kafka lets the OS filesystem handle the storage layer. It relies on the OS transferring all data to the page cache before flushing it to the disk.



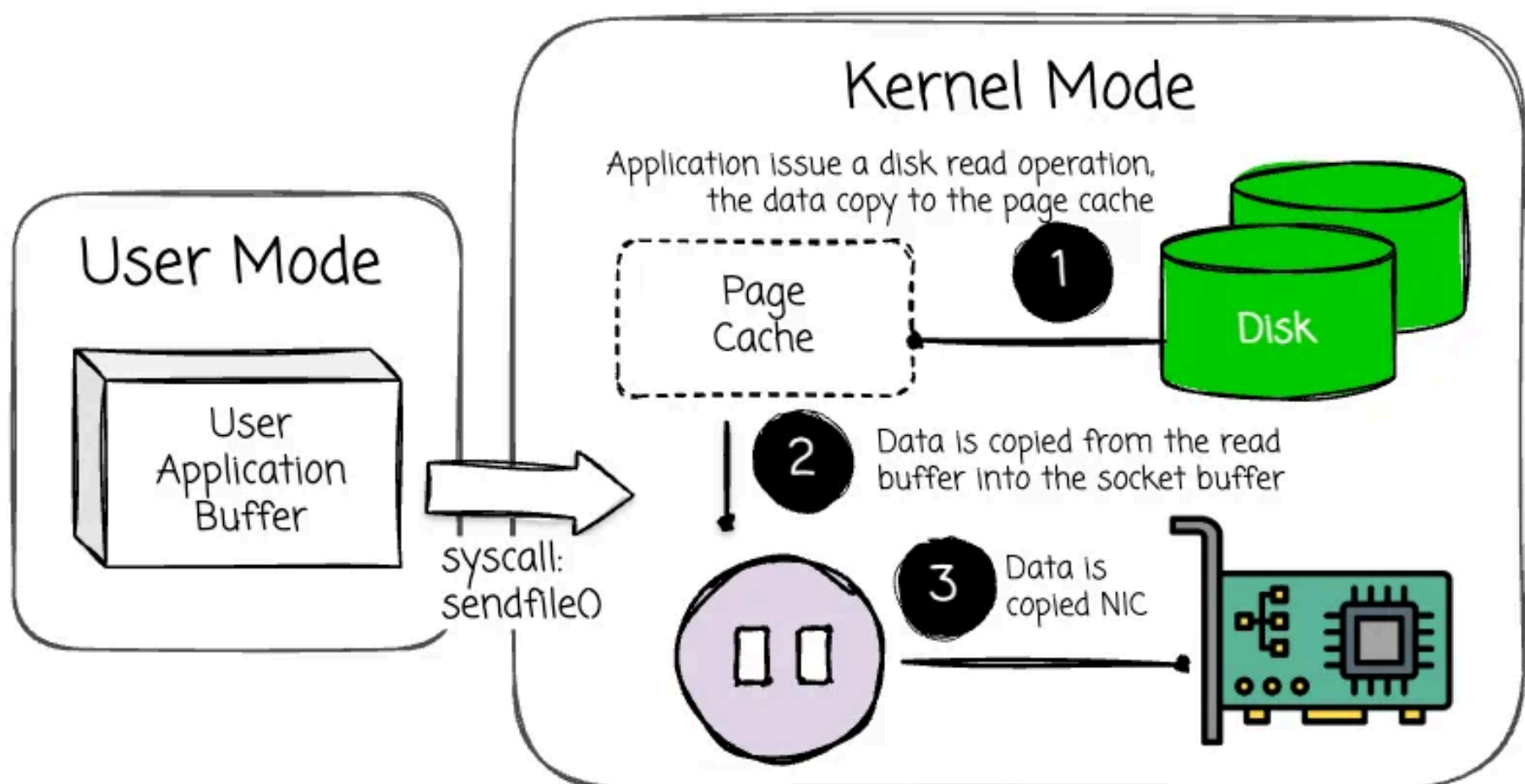
Sequential access pattern

"Because the disk is always slower than RAM, is that going to affect the Kafka performance?", you might wonder.

The key here is **the access pattern**. There is no doubt that with random access, the disk will be slower than RAM, but it can outperform memory slightly when it comes to **sequential access**.

Zero-copy

With the zero-copy optimization, the data is **copied directly** from the page cache to the socket buffer. Thus, this optimization can help Kafka avoid redundant data copies



Batching

To make the client-broker request more efficient, the Kafka protocol has a message set abstraction that helps group messages together.

This helps mitigate the network round-trip overhead when sending too many single message requests. Batching also helps the broker write the message more efficiently

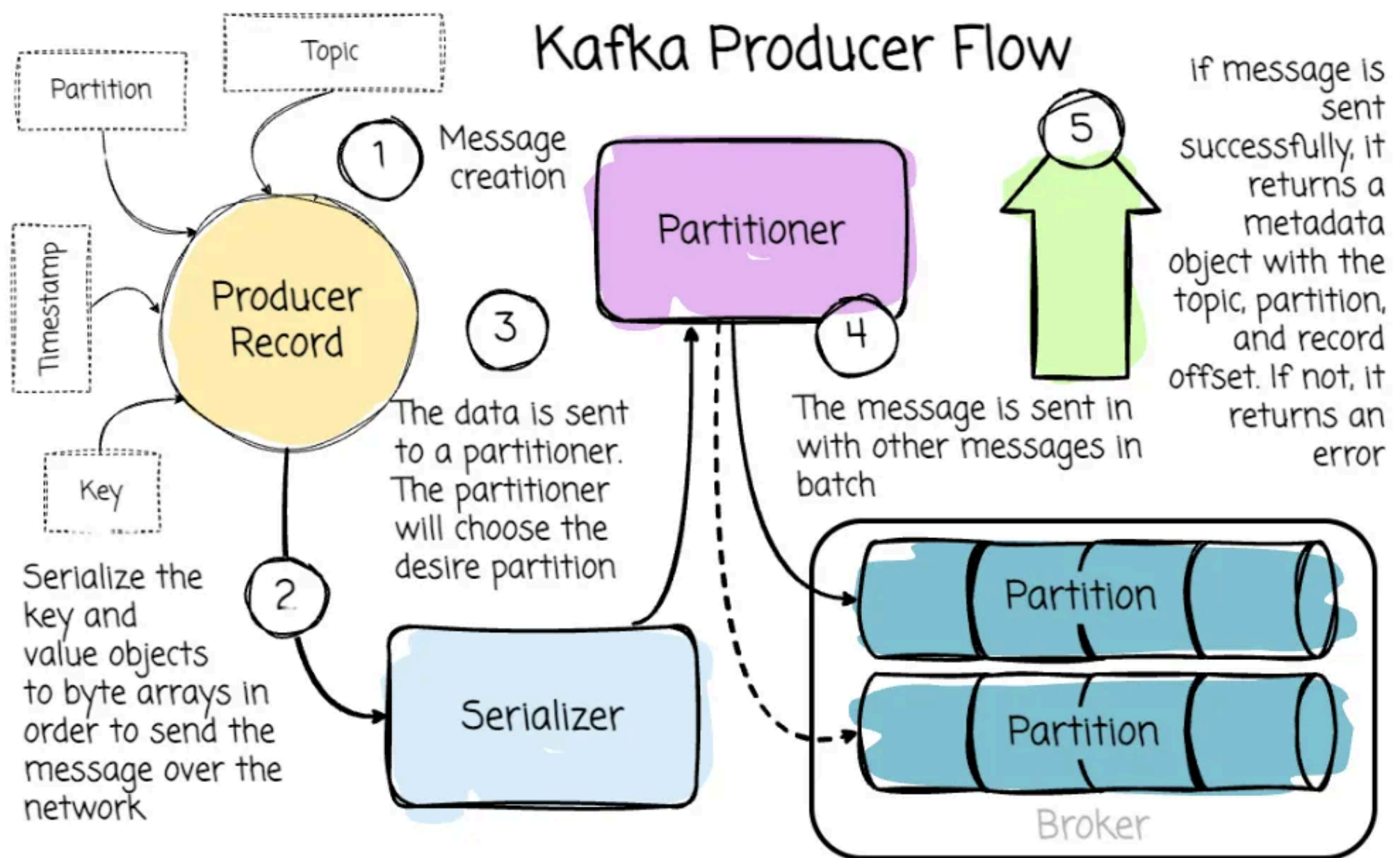
Producer

The flow

When you use the Kafka producer API, a few things happen:

- The process creates a `ProducerRecord`
- The producer serializes the `ProducerRecord`
- If no partition is specified, the data is routed to the partitioner
-
- After knowing the destination, the producer adds the record to the batch of messages.
- A different thread will send these batches
- When the broker receives messages, if successful, it returns a metadata object. If not, it returns an error.

The flow

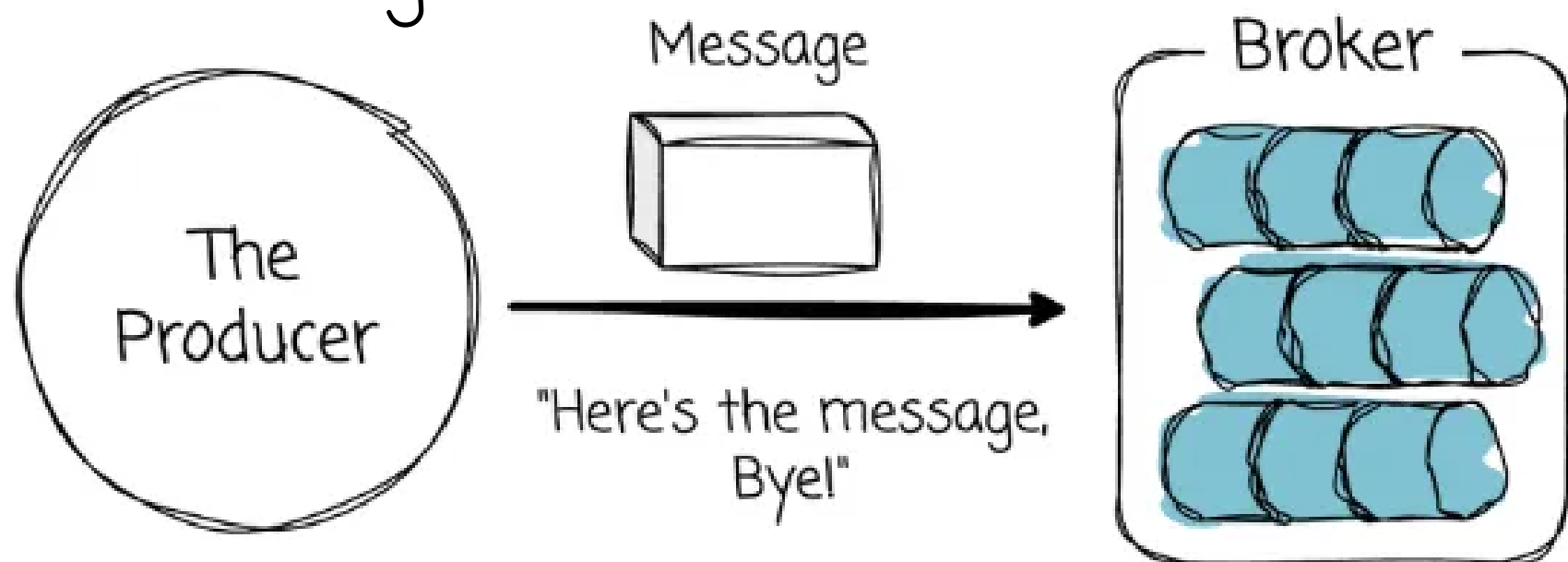


Sending method

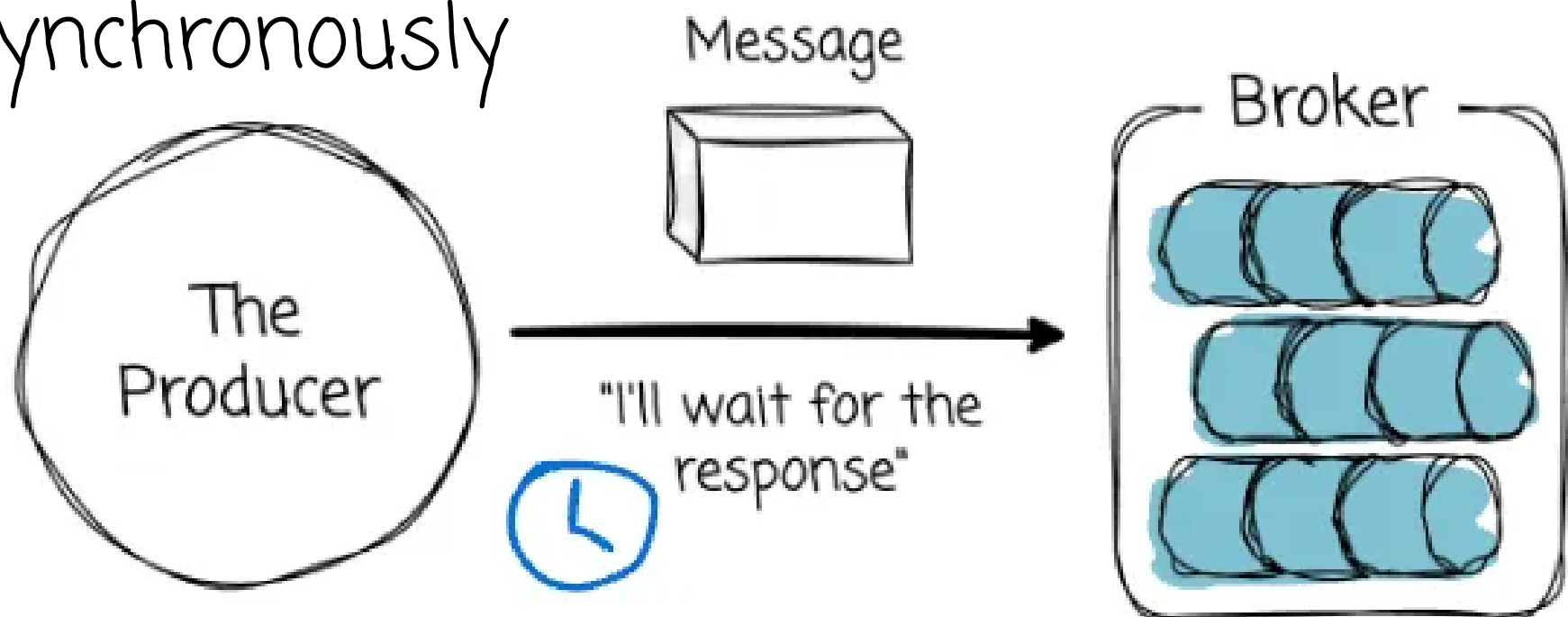
We control the way we want to send the message

- Fire-and-forget: The producer sends a message to the server and **doesn't check** if it arrives.
- Synchronously: the producer sends the message and **waits for the response**. This method is rare in production because it can impact the performance.
- Asynchronously: The producers send all messages **without waiting for replies**. They support adding a callback to handle errors while executing an asynchronous send.

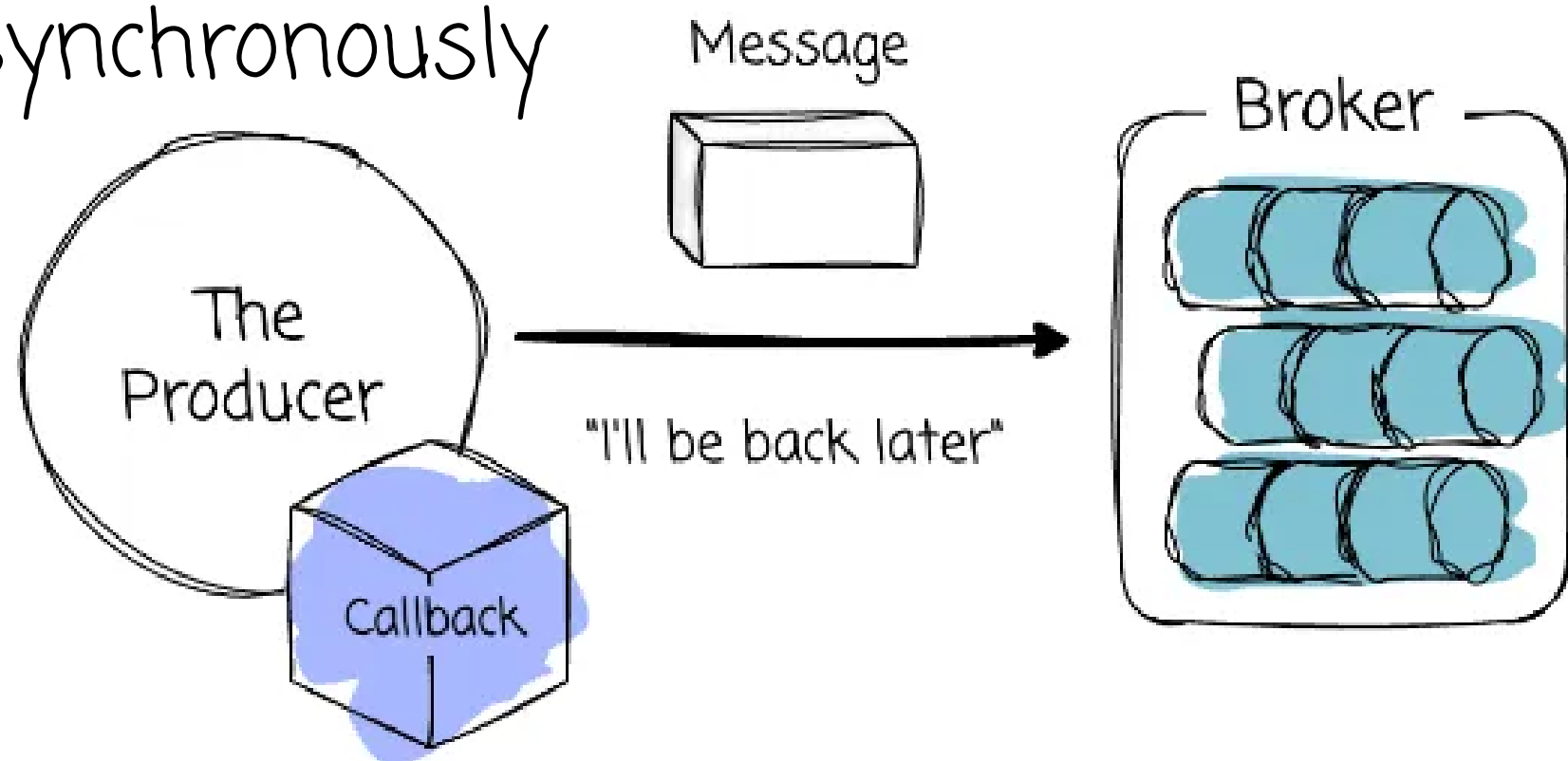
Fire-and-forget



Synchronously

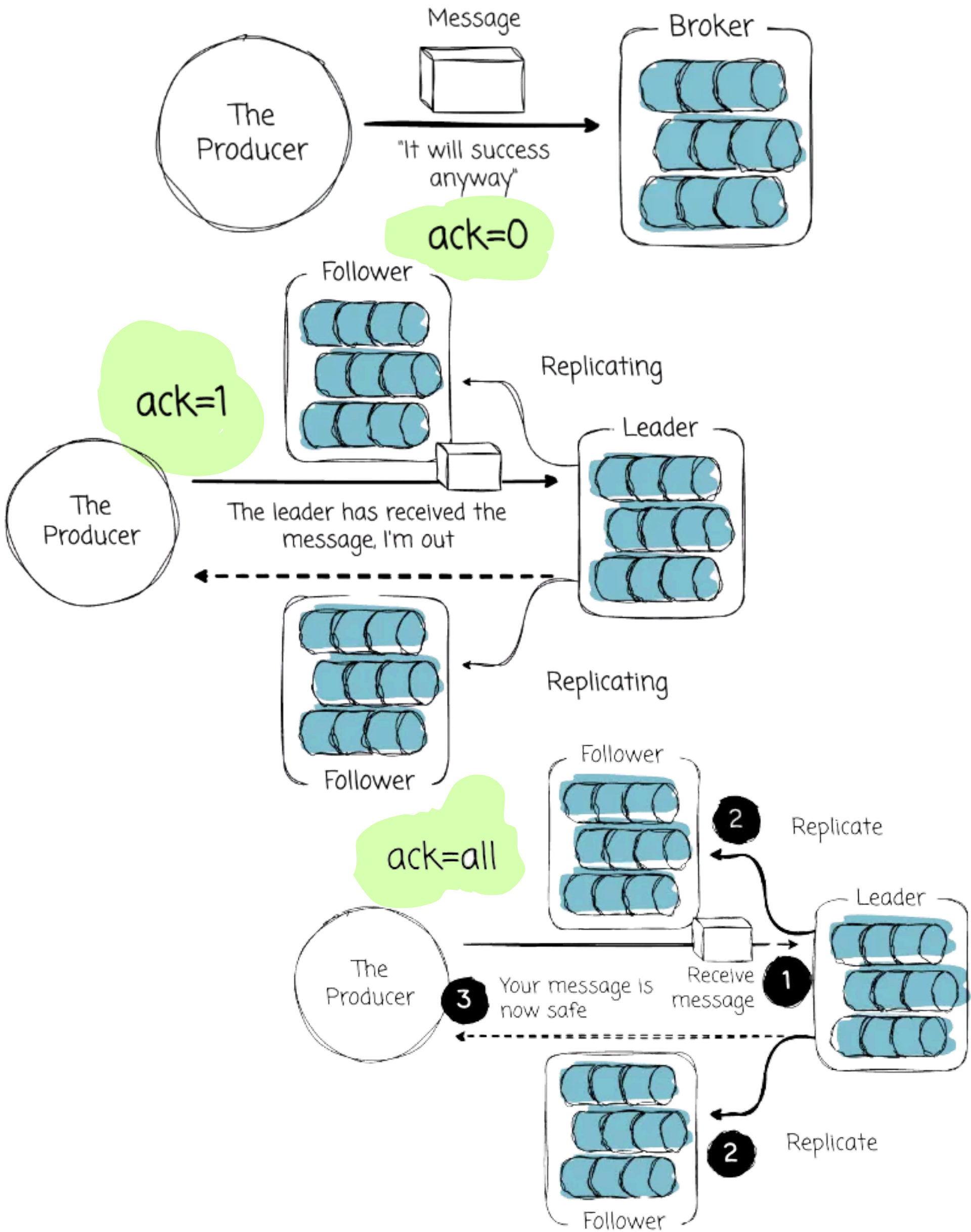


Asynchronously



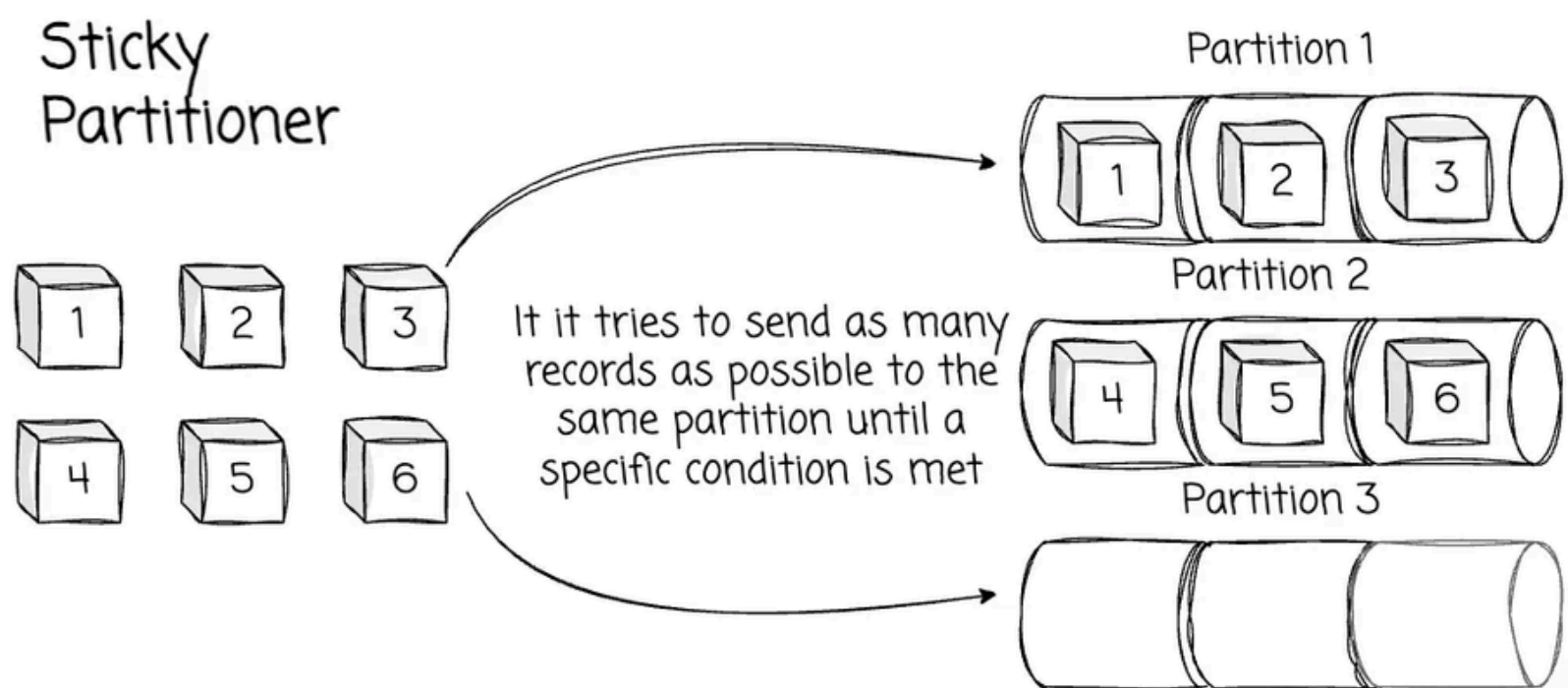
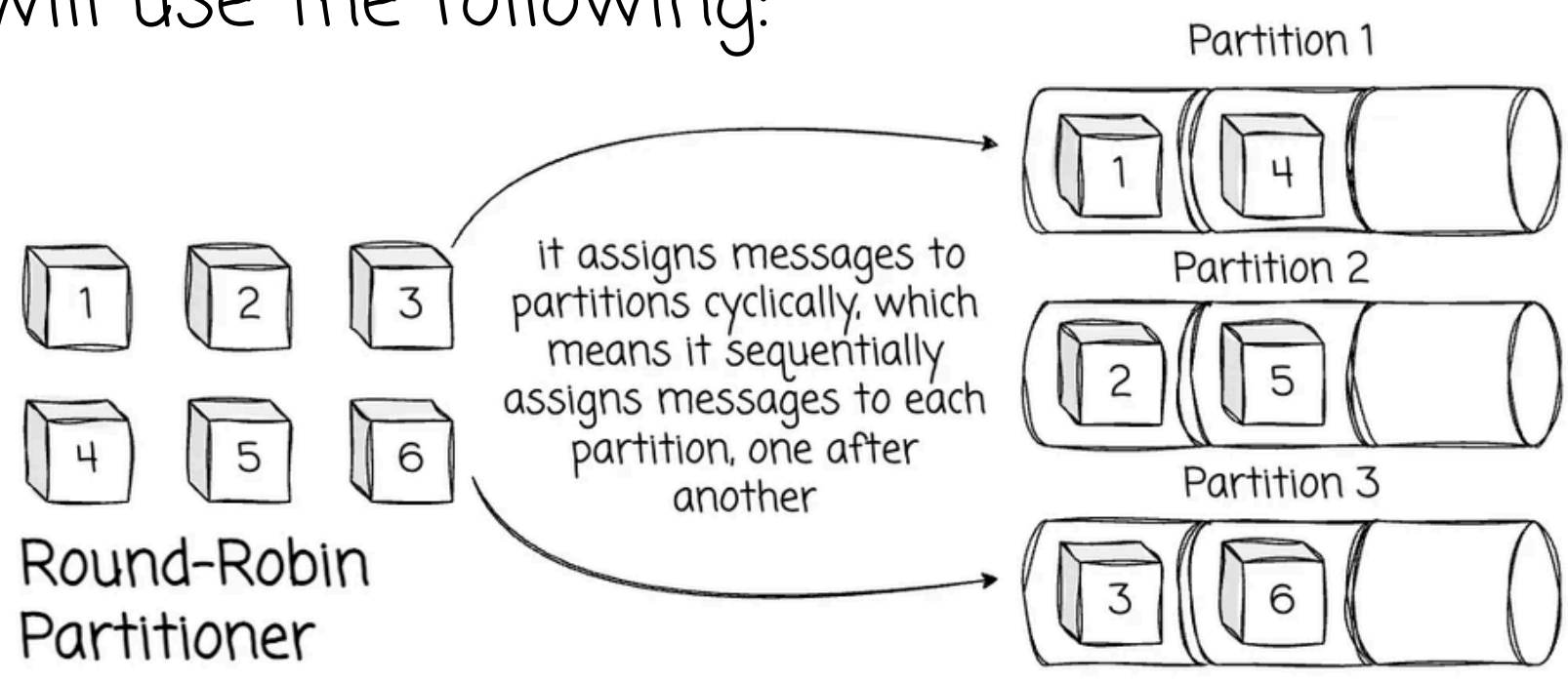
Was the message delivered successfully?

- `acks=0`: The producer **doesn't wait** for a reply from the broker and assumes the message was sent successfully.
- `acks=1`: The producer receives a "yes" response **once the leader** gets the message.
- `acks=all`: The producer gets a "yes" response only after **all replicas receive the message**. This mode is the safest, ensuring the message survives even if a broker crashes. However, it increases latency.



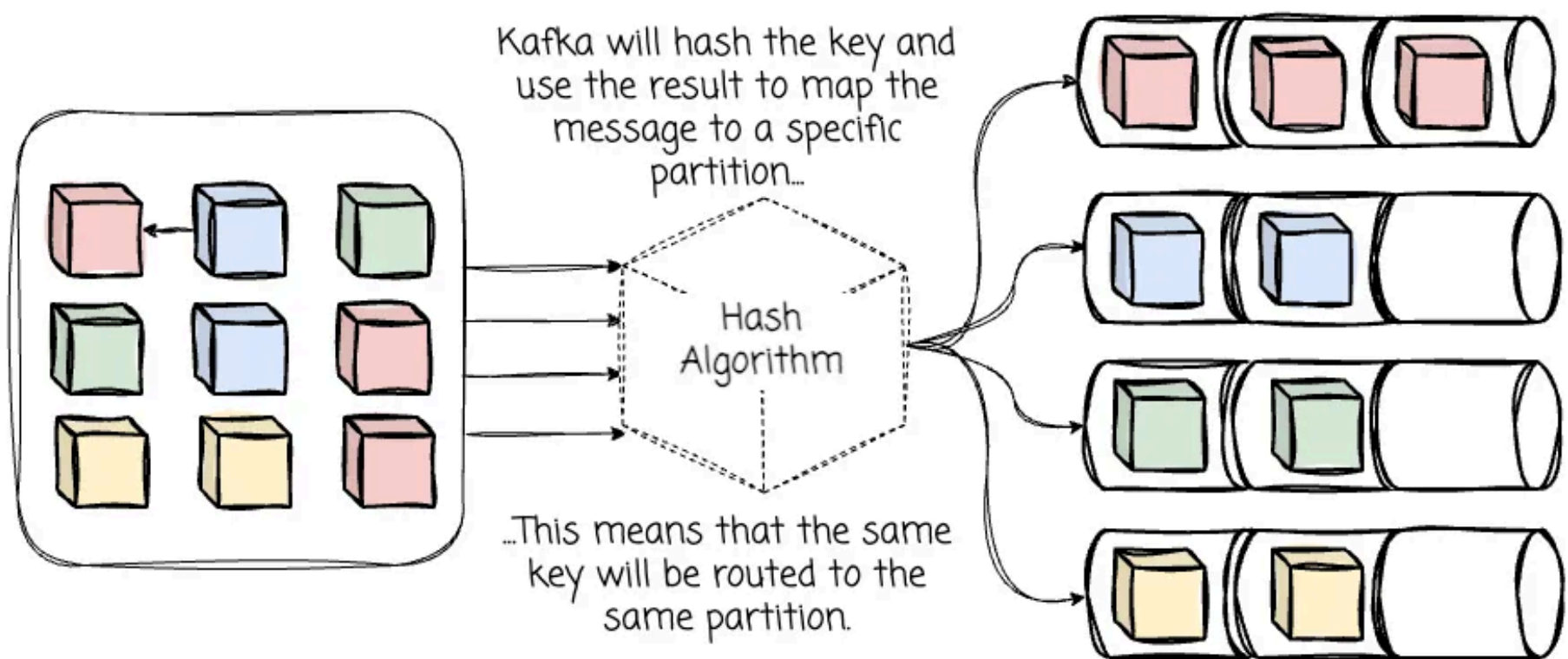
How do we distribute the message?

Kafka messages can have a key, which is null by default. The message's key is mainly **used to decide the message destination partition**. When the key is null, and no defined custom partitioner, Kafka will use the following:



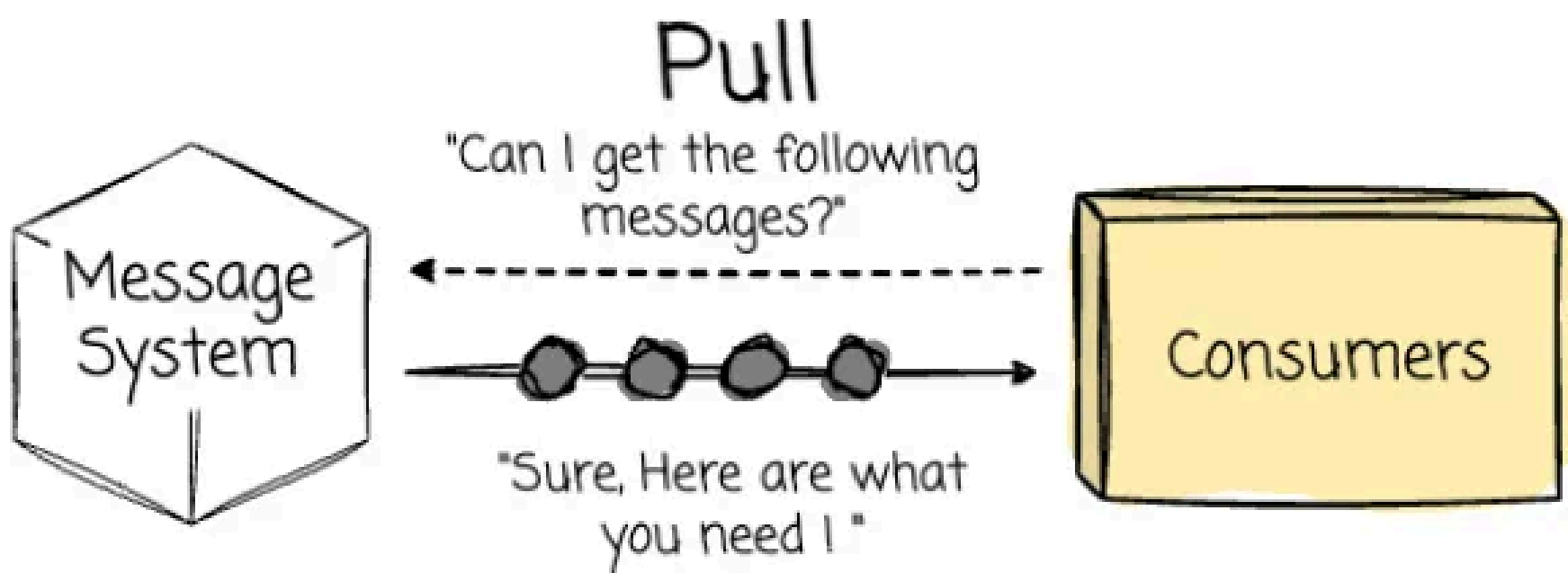
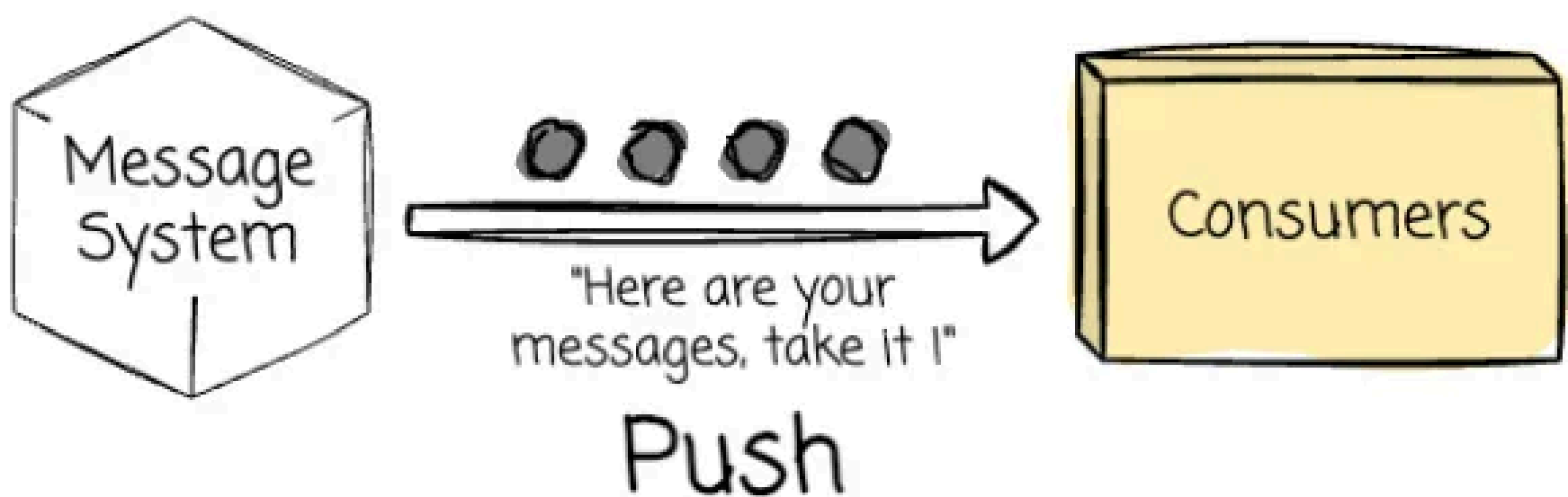
How do we distribute the message?

If the message's key is not null, Kafka will **hash** it with a hash algorithm and **use the result to map** the message to a particular partition.



Consumer

LinkedIn engineers found the "pull" model more suitable for their applications because consumers can read the messages at a rate ideal for their capacity, allowing them to manage their workload effectively. The consumer can also avoid being flooded by messages pushed faster than they can manage.



The request

A consumer always consumes messages from a particular partition sequentially. If the consumer acknowledges a message offset, the broker implies that the consumer has received all the previous partition's messages from this offset.

The Consumer API is an infinite loop for polling the broker for more data. It will issue asynchronous pull requests to the broker to retrieve the data. Each request contains the offset of the message from which the consumption begins.

Consumer Group

Kafka has a concept of consumer groups.

Each group has one or more consumers who will consume a set of subscribed topics.

LinkedIn made a topic's partition the smallest unit of parallelism; all messages from one partition are consumed only by a single consumer within a group.

Consumer Group

