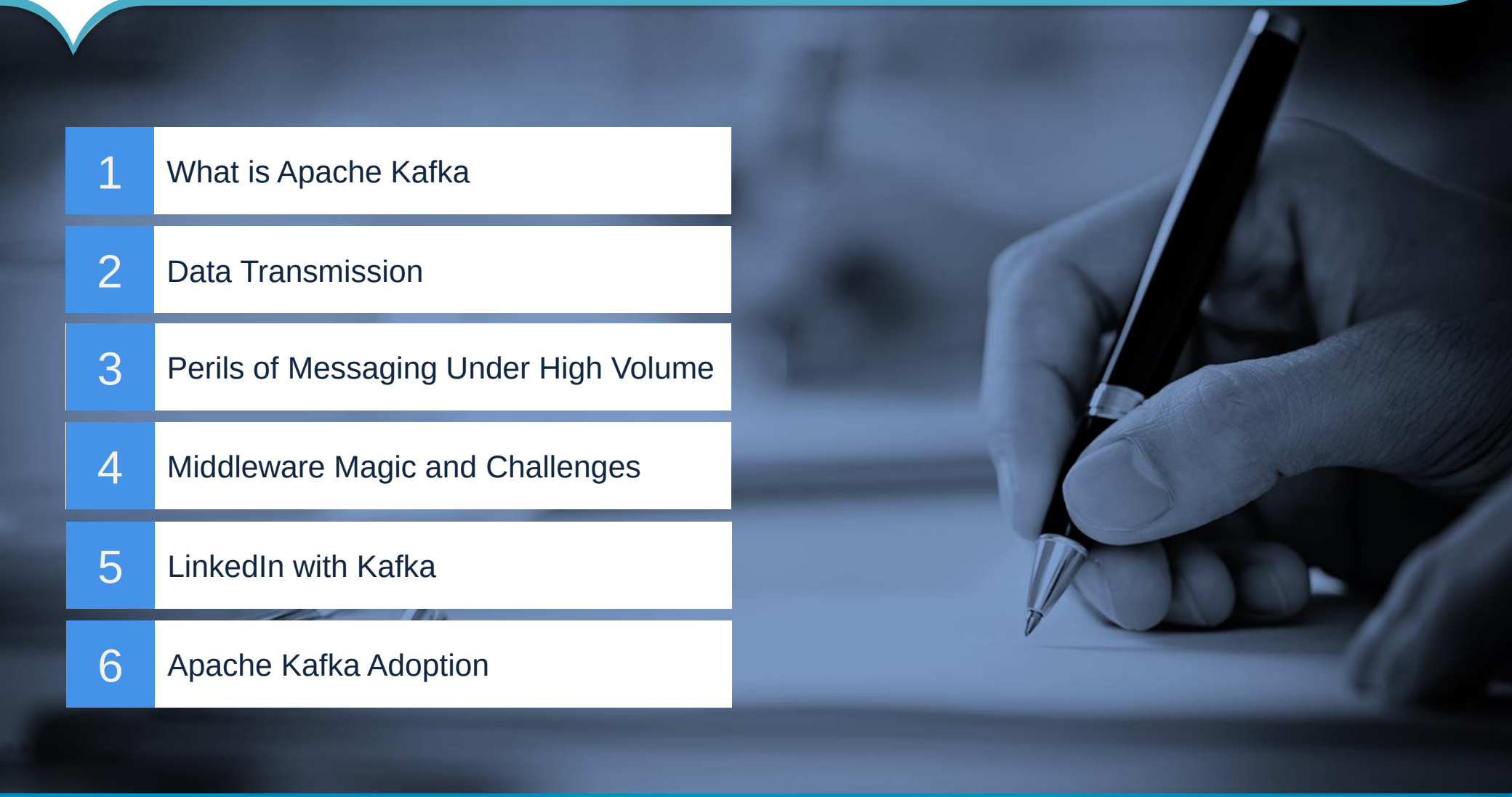
Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Apache Kafka

May 2017

People matter, results count.

Course Map



1. What is Apache Kafka
2. Data Transmission
3. Perils of Messaging Under High Volume
4. Middleware Magic and Challenges
5. LinkedIn with Kafka
6. Apache Kafka Adoption

What is Kafka?

Kafka-Messaging System

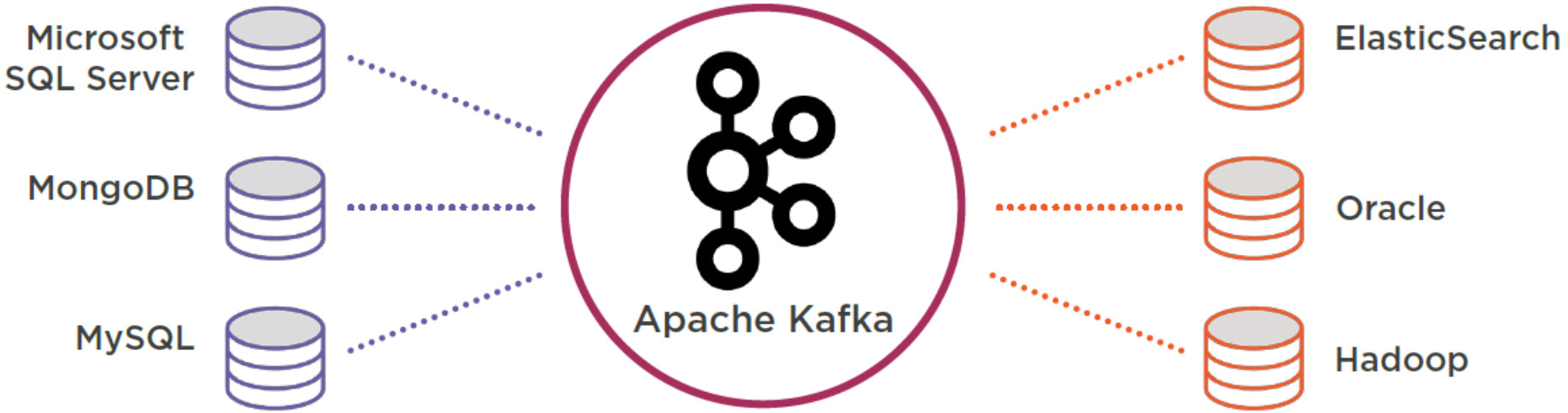
- Kafka is an open source distributed streaming platform that simplifies data integration between systems. A stream is a pipeline to which your applications receives data continuously. As a streaming platform Kafka has two primary uses:
- **Data Integration:** Kafka captures streams of events or data changes and feeds these to other data systems such as relational databases, key-value stores or data warehouses.
- **Stream processing:** Kafka accepts each stream of events and stores it in an append-only queue called a log. Information in the log is immutable hence enables continuous, real-time processing and transformation of these streams and makes the results available system-wide.
- Compared to other technologies, Kafka has a better throughput, built-in partitioning, replication, and fault-tolerance which makes it a good solution for large-scale message processing applications.

What is Kafka?

Kafka-Messaging System

- A Messaging System is responsible for transferring data from one application to another, so the applications can focus on data, but not worry about how to share it.
- Distributed messaging is based on the concept of reliable message queuing.
- Messages are queued asynchronously between client applications and messaging system.
- Two types of messaging patterns are available − one is point to point and the other is publish-subscribe (pub-sub) messaging system.
- Most of the messaging patterns follow pub-sub.

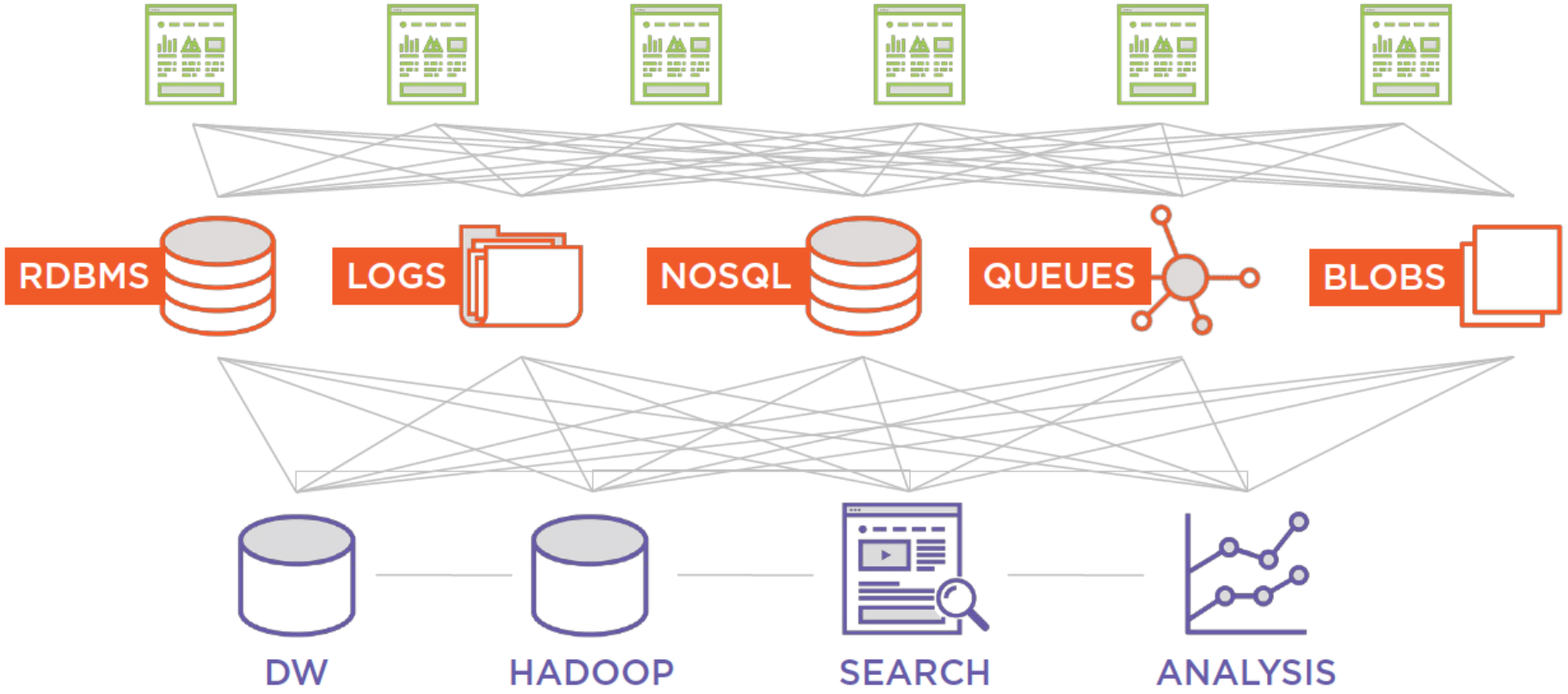
What is Kafka?



“A high - throughput distributed messaging system.”

What is Kafka?

What a Typical Enterprise Looks Like

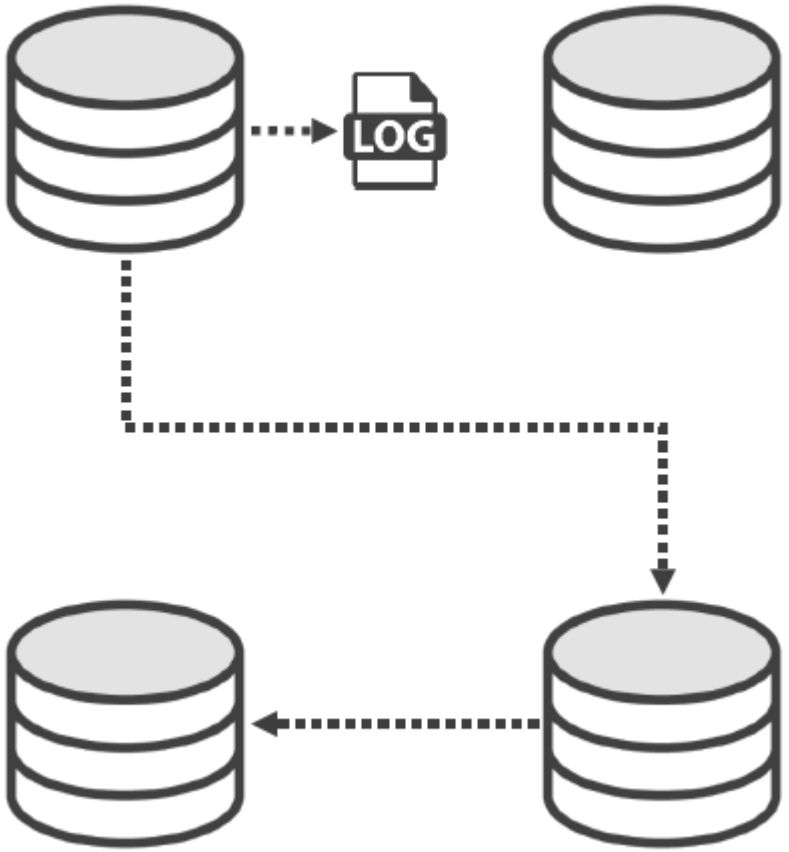


Data Transmission

- Database replication
- Log shipping
- Extract, Transform, and Load (ETL)
- Messaging
- Custom middleware magic

Data Transmission

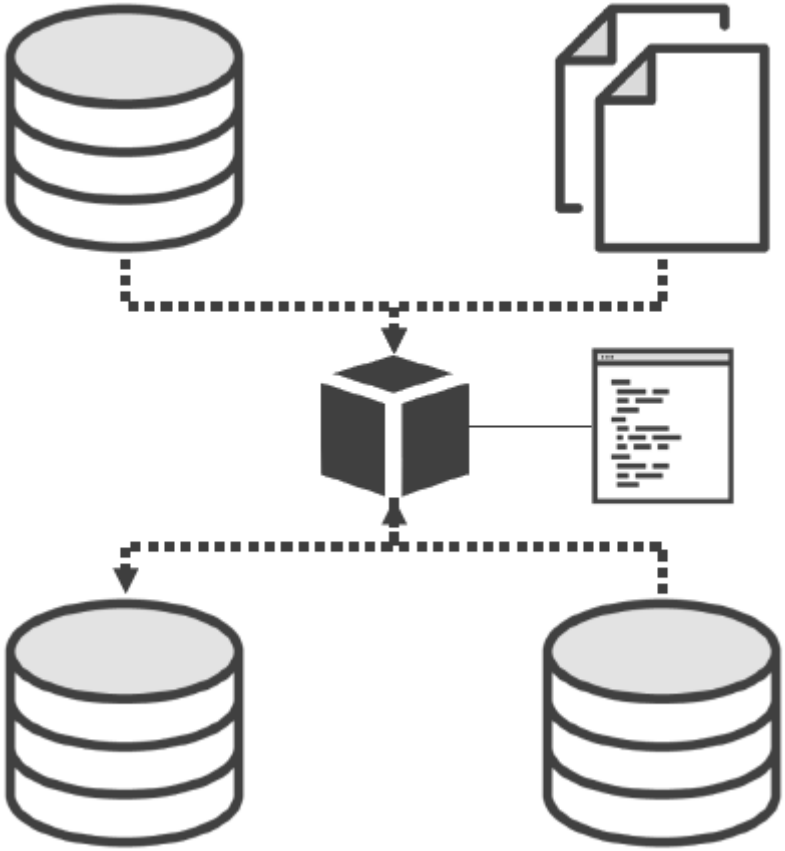
Database Replication and Log Shipping



- RDBMS to RDBMS only
- Database-specific
- Tight coupling (schema)
- Performance challenges (log shipping)
- Cumbersome (subscriptions)

Data Transmission

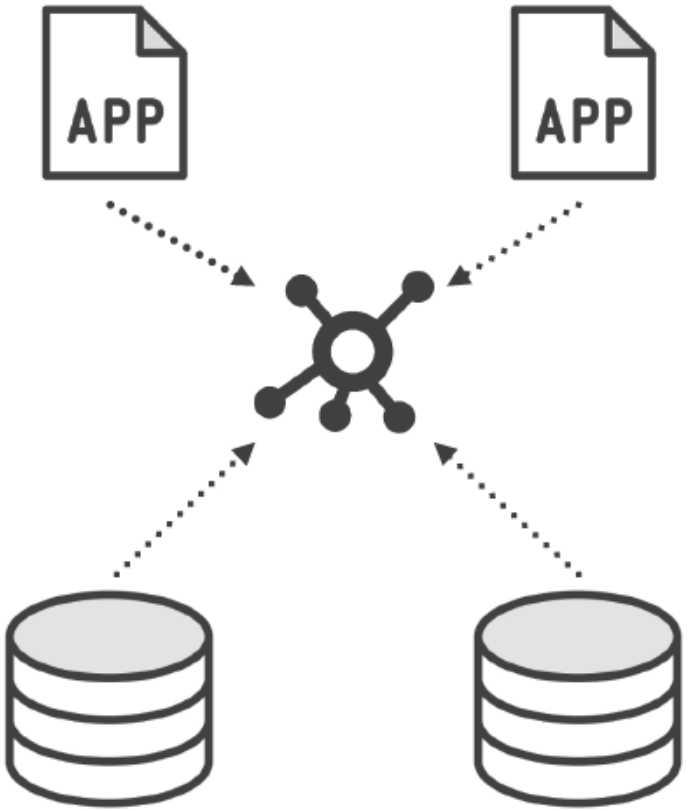
Extract, Transform, and Load (ETL)



- Typically proprietary and costly
- Lots of custom development
- Scalability challenged
- Performance challenged
- Often times requires multiple instances

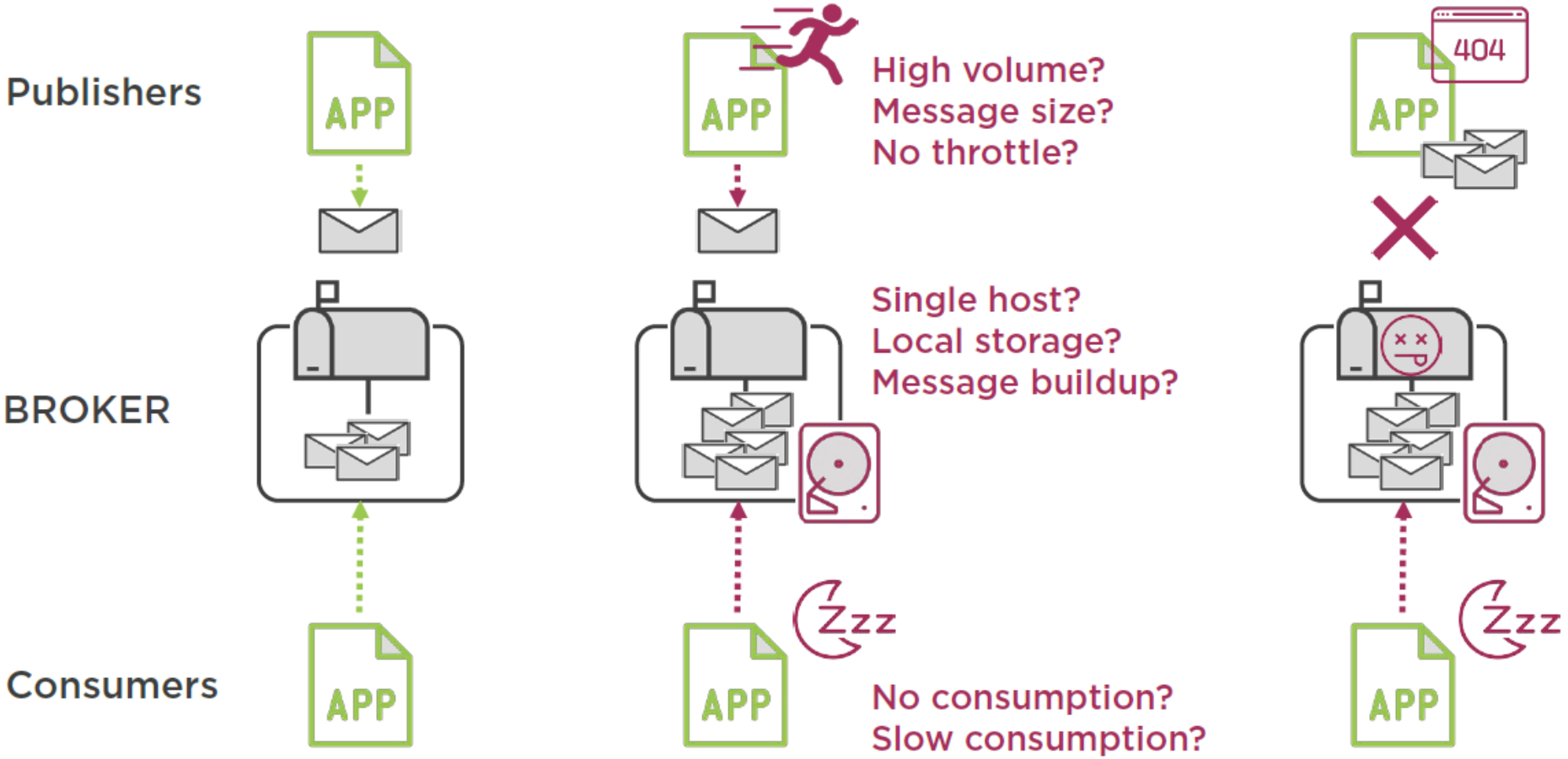
Data Transmission

Messaging

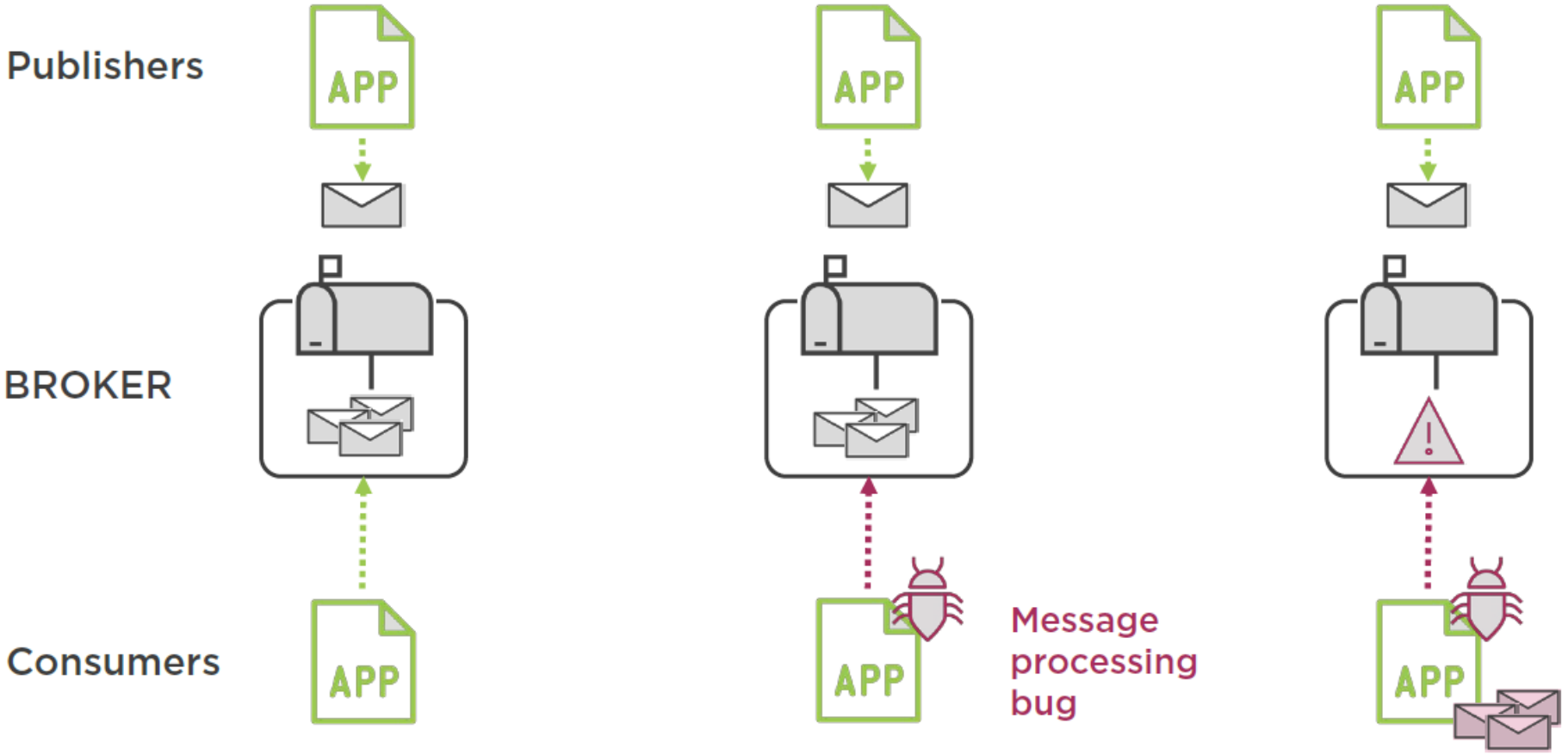


- Limited scalability
- Smaller messages
- Requires rapid consumption
- Not fault -tolerant (application)

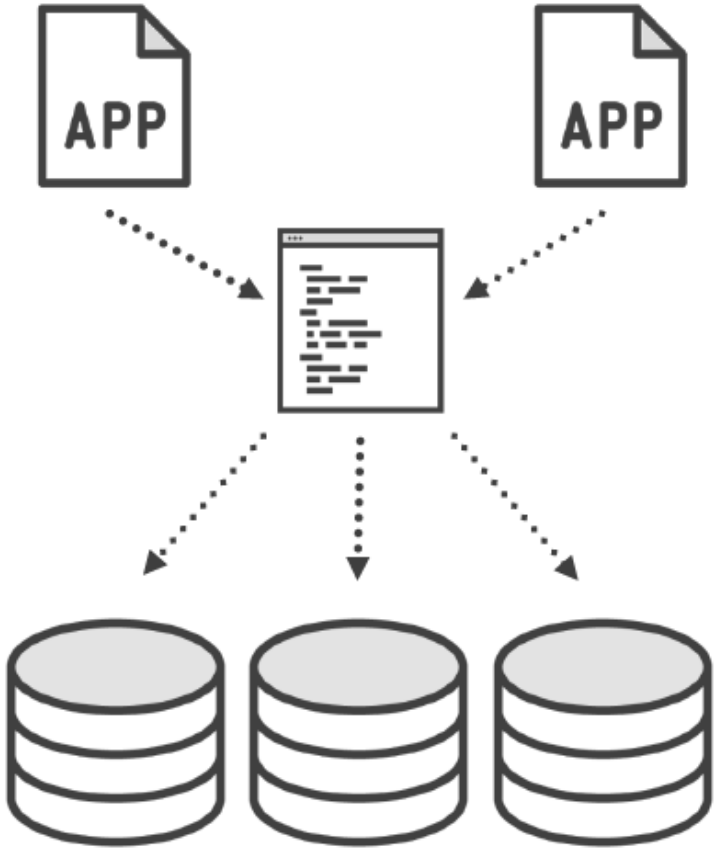
Perils of Messaging Under High Volume



Perils of Messaging Under High Volume



Middleware Magic

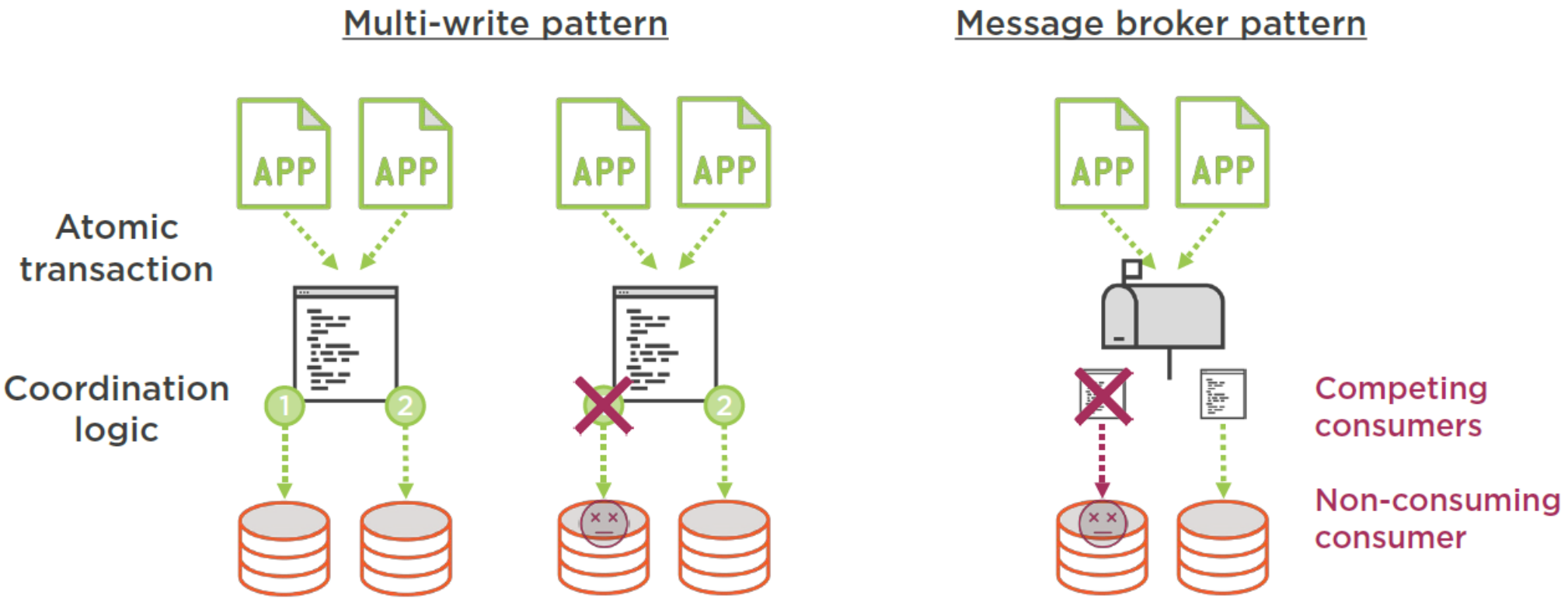


- Increasingly complex
- Deceiving
- Consistency concerns
- Potentially expensive

Middleware Challenges

Problems

Cleanly
Reliably
Quickly
Autonomously



LinkedIn with Kafka

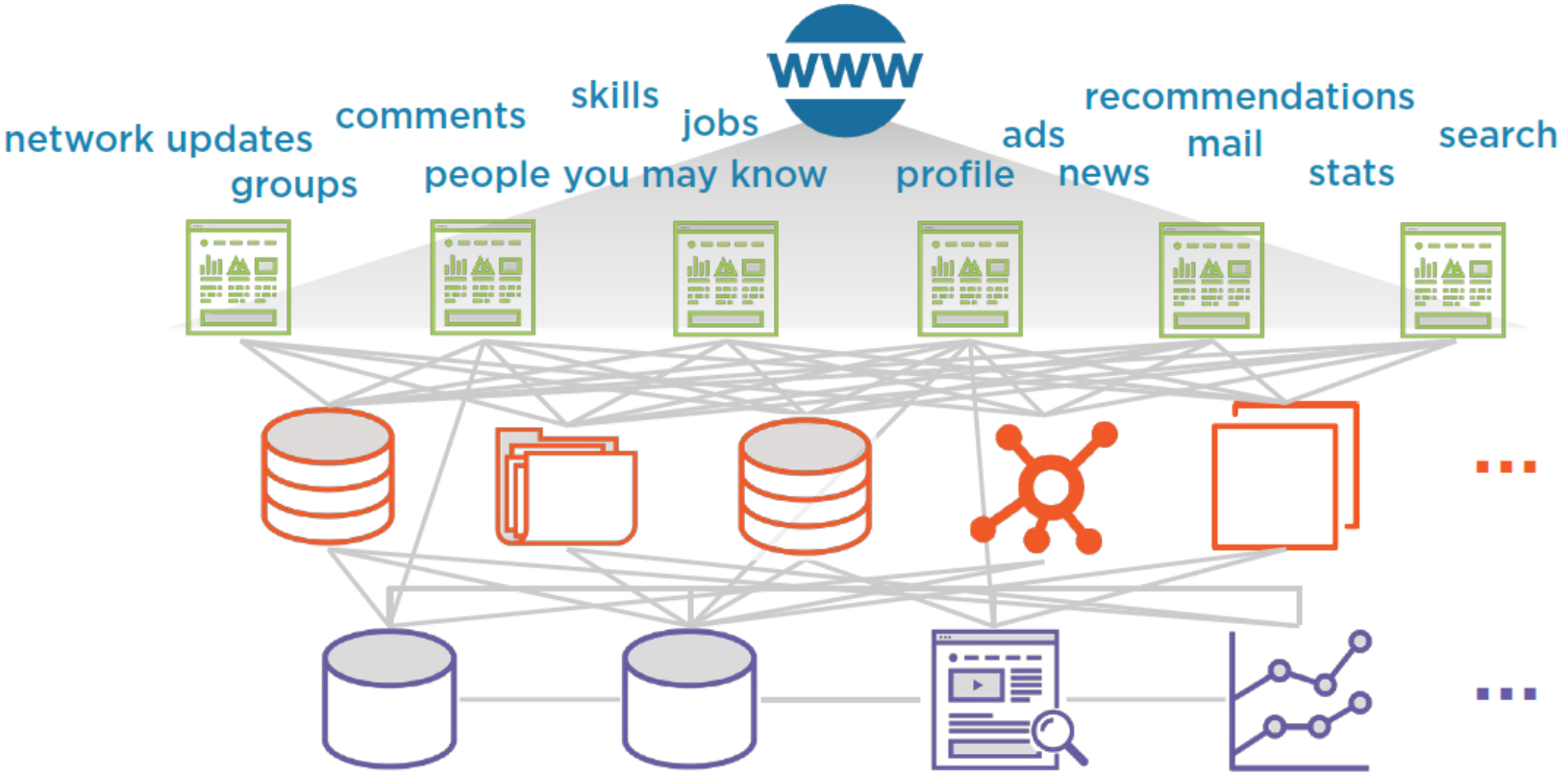
- Over 1.4 trillion messages per day
- 175 terabytes per day
- 650 terabytes of messages consumed per day
- Over 433 million users

- Peak 13 million messages per second
- 2.75 gigabytes per second

- Multiple RDBMS (Oracle, MySQL, etc.)
- Multiple NoSQL (Espresso, Voldemort)
- Hadoop, Spark, etc.

LinkedIn with Kafka

Pre-2010 LinkedIn Data Architecture



LinkedIn with Kafka

Franz Kafka

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

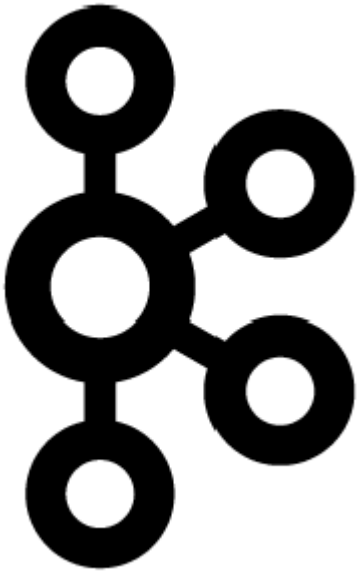
Basically it describes a nightmarish situation which most people can somehow relate to, although strongly surreal.

synonyms: surreal, lucid, spoilsburytoast boy

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

LinkedIn with Kafka

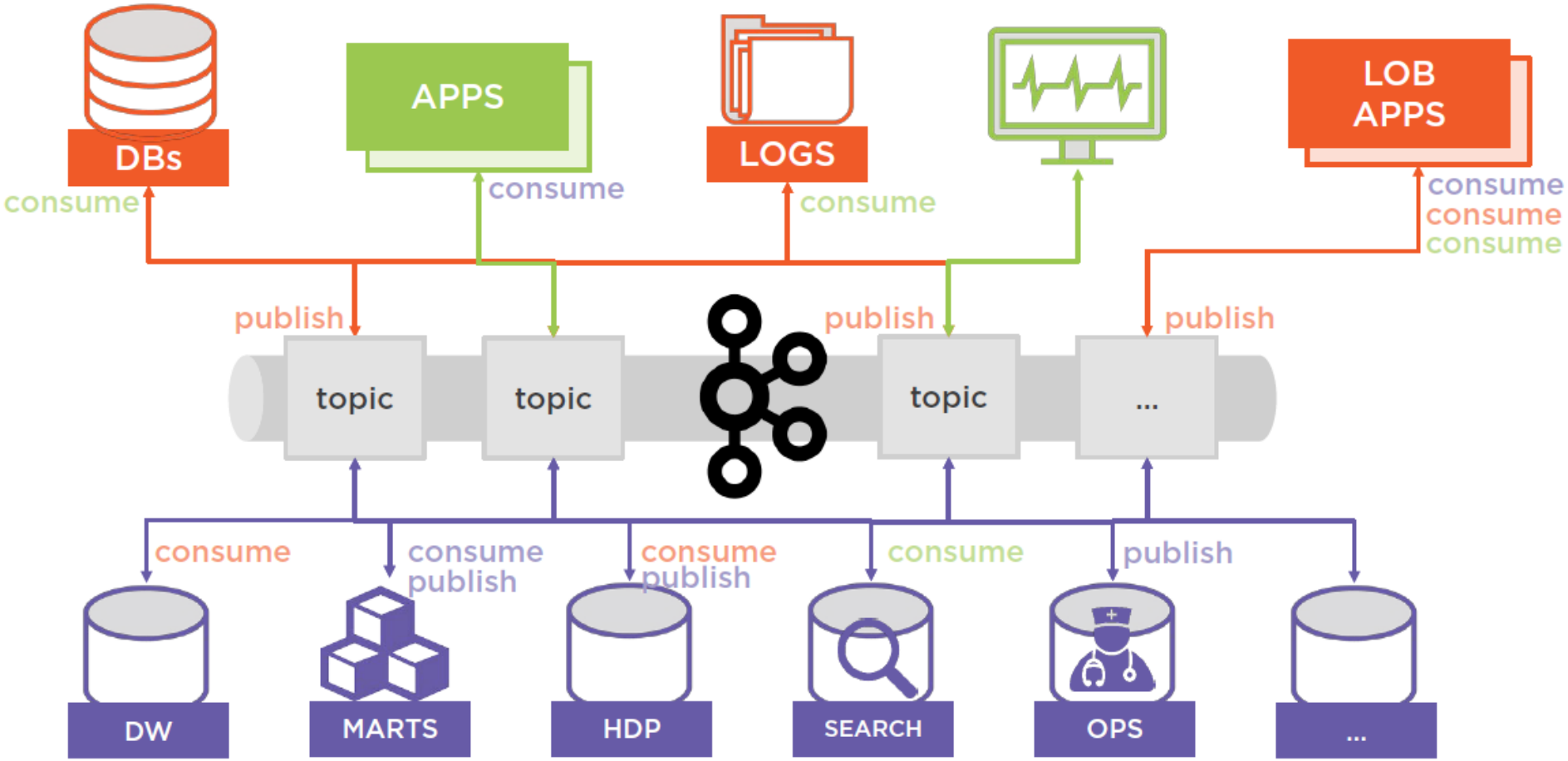
Next-generation Messaging Goals



- High throughput
- Horizontally scalable
- Reliable and durable
- Loosely coupled Producers and Consumers
- Flexible publish -subscribe semantics

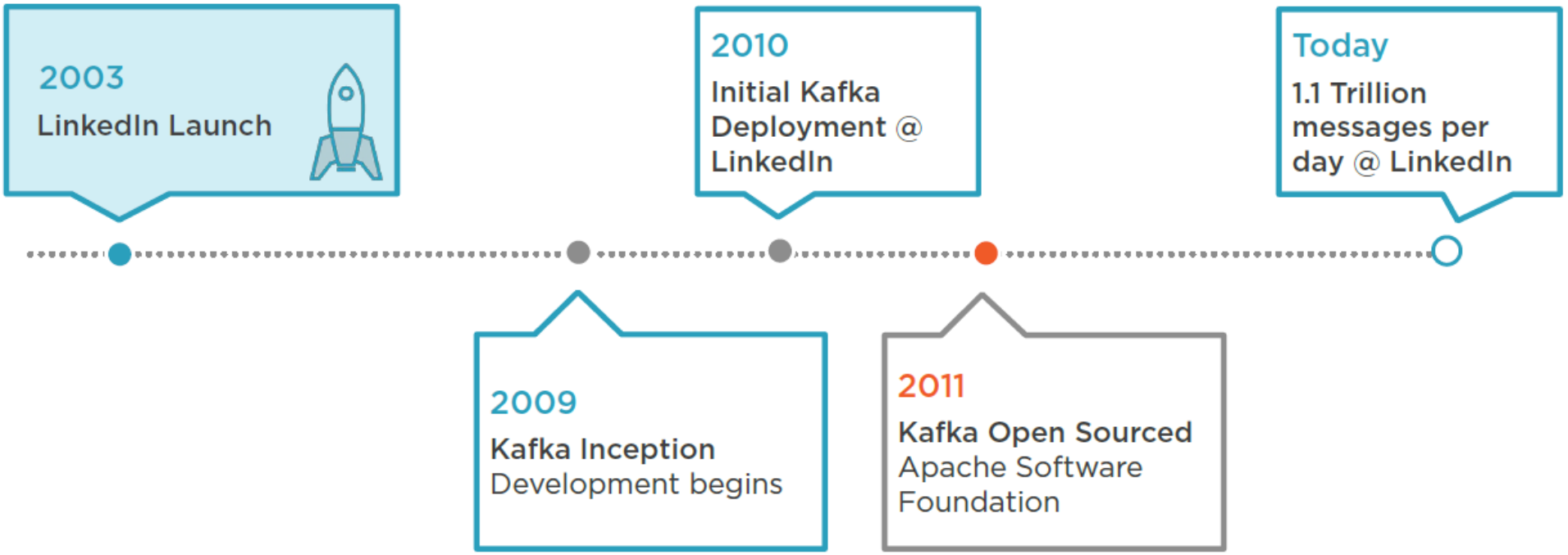
LinkedIn with Kafka

Post -2010 LinkedIn Data Architecture



LinkedIn with Kafka

TimeLine Events



Apache Kafka Adoption

7X since 2015

Yahoo
Etsy
Microsoft
Bing
Mailchimp
Uber

Oracle
Goldman Sachs
Netflix
PayPal
Square
Coursera
IBM
Pintrest

Twitter
Airbnb
Spotify
Ancestry
LinkedIn
Hotels.com

Summary

- Kafka is a distributed messaging system
- Designed to move data at high volumes
- Addresses shortcomings of traditional
- data movement tools and approaches
- Invented by LinkedIn to address data
- growth issues common to many enterprises
- Open - sourced under Apache Software Foundation in 2012
- First -choice adoption for data movement for hundreds of enterprise and internet-scale companies

Course Map – Kafka Architecture

1	Apache Kafka Architecture
2	Workflow of Pub-Sub Messaging
3	Apache Kafka Cluster
4	Distributed Systems
5	Apache Zookeeper
6	Apache Kafka Distributed Architeture
7	Kafka Use cases, Competitors

Workflow of Pub-Sub Messaging

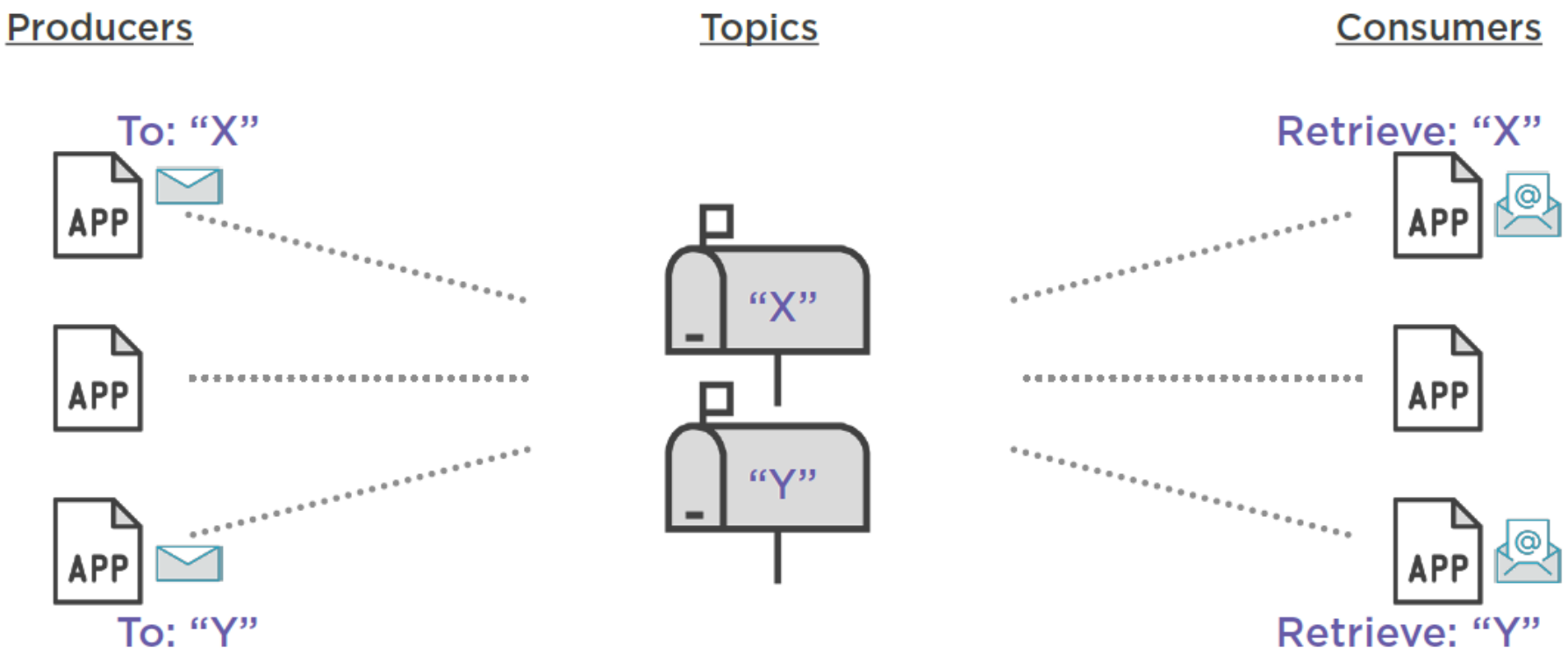
Following is the step wise workflow of the Pub-Sub Messaging –

- Producers send message to a topic at regular intervals.
- Kafka broker stores all messages in the partitions configured for that particular topic. It ensures the messages are equally shared between partitions.
- If the producer sends two messages and there are two partitions, Kafka will store one message in the first partition and the second message in the second partition.
- Consumer subscribes to a specific topic.
- Once the consumer subscribes to a topic, Kafka will provide the current offset of the topic to the consumer and also saves the offset in the Zookeeper ensemble.
- Consumer will request the Kafka in a regular interval (like 100 Ms) for new messages.

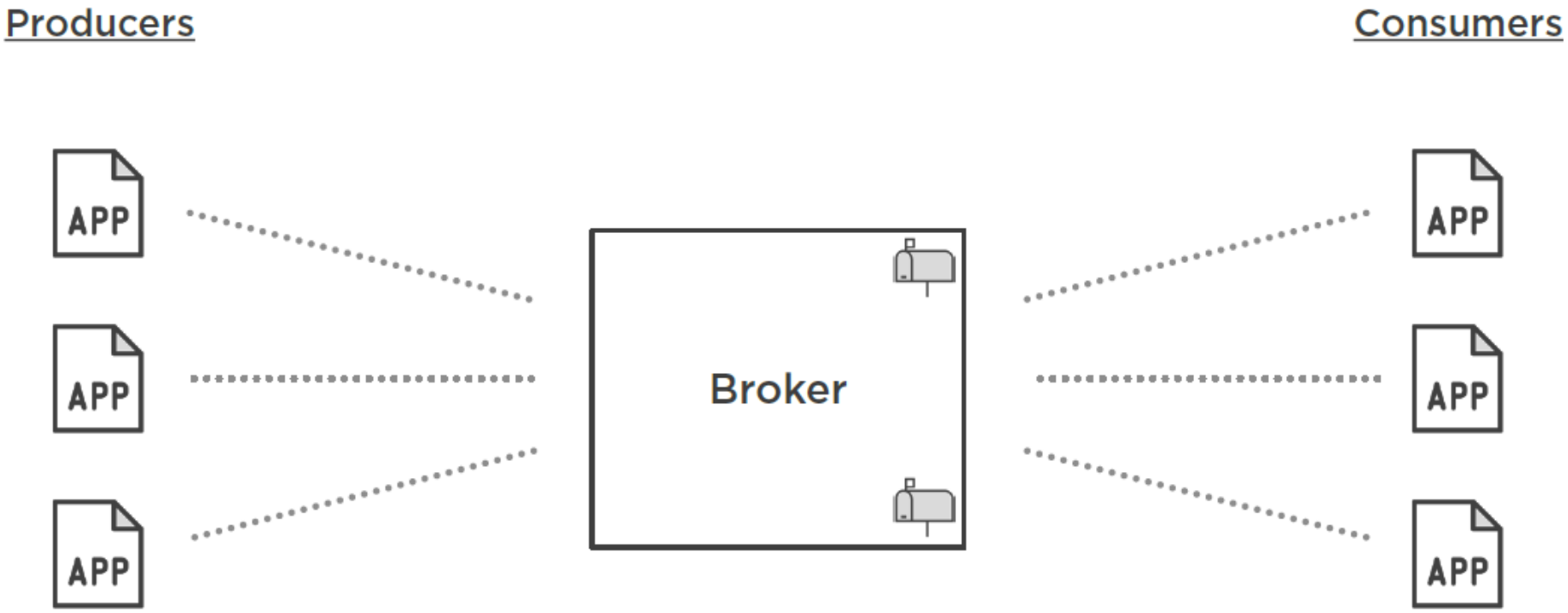
Workflow of Pub-Sub Messaging(Contd…)

- Once Kafka receives the messages from producers, it forwards these messages to the consumers.
- Consumer will receive the message and process it.
- Once the messages are processed, consumer will send an acknowledgement to the Kafka broker.
- Once Kafka receives an acknowledgement, it changes the offset to the new value and updates it in the Zookeeper.
- Since offsets are maintained in the Zookeeper, the consumer can read next message correctly even during server outrages.
- This above flow will repeat until the consumer stops the request.
- Consumer has the option to rewind/skip to the desired offset of a topic at any time and read all the subsequent messages.

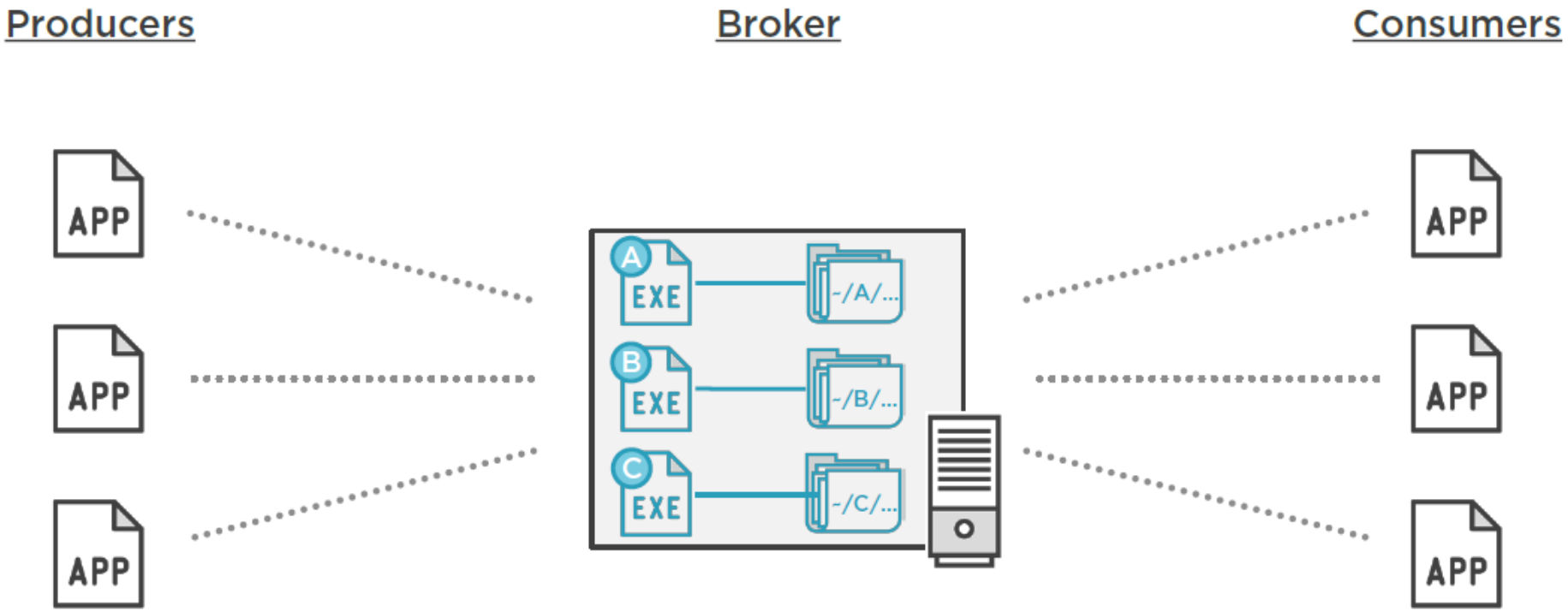
Apache Kafka Architecture



Apache Kafka Architecture

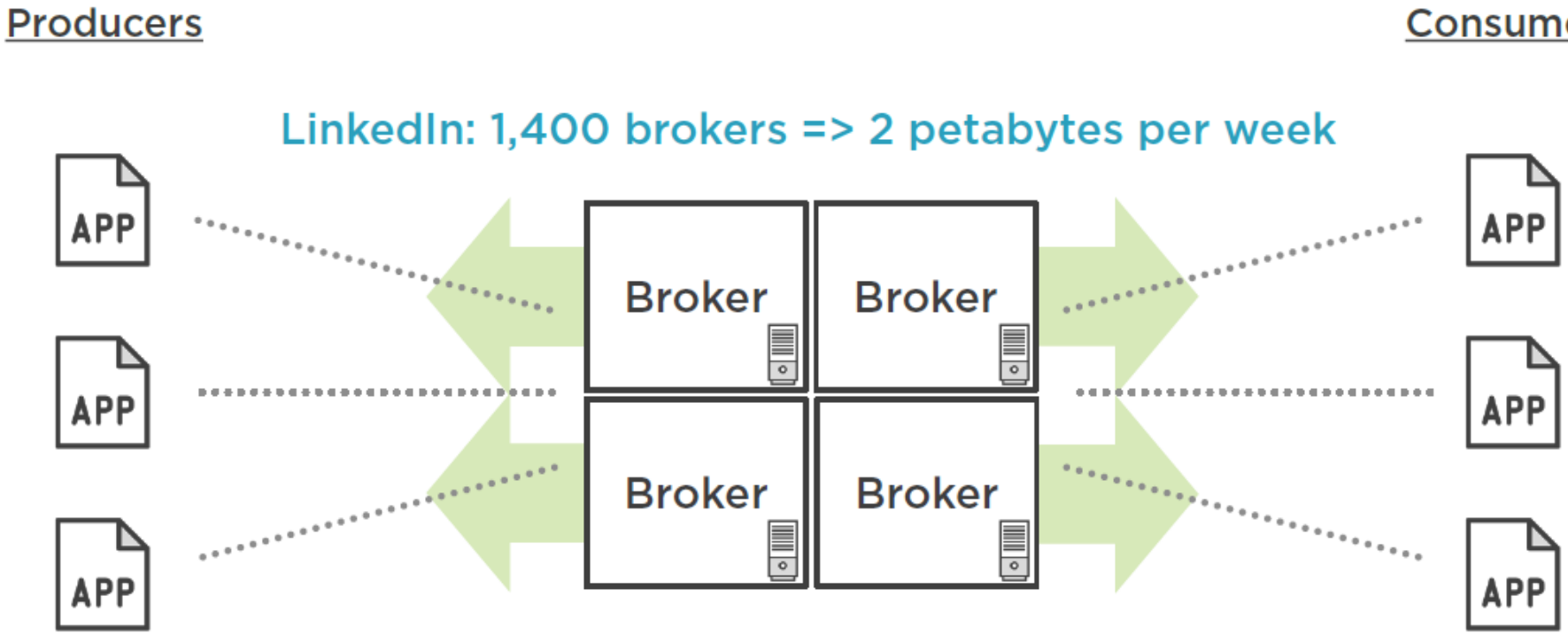


Apache Kafka Architecture



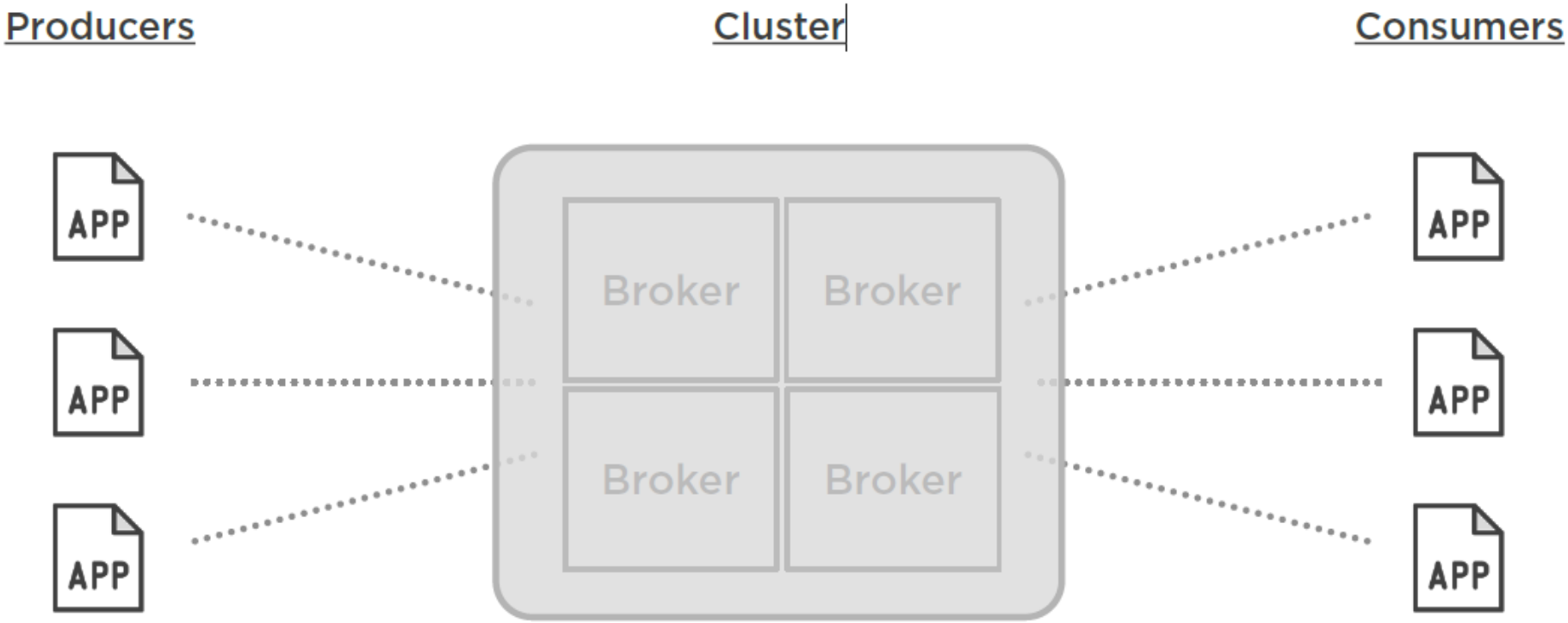
Apache Kafka Architecture

How Apache Kafka Starts to Differentiate

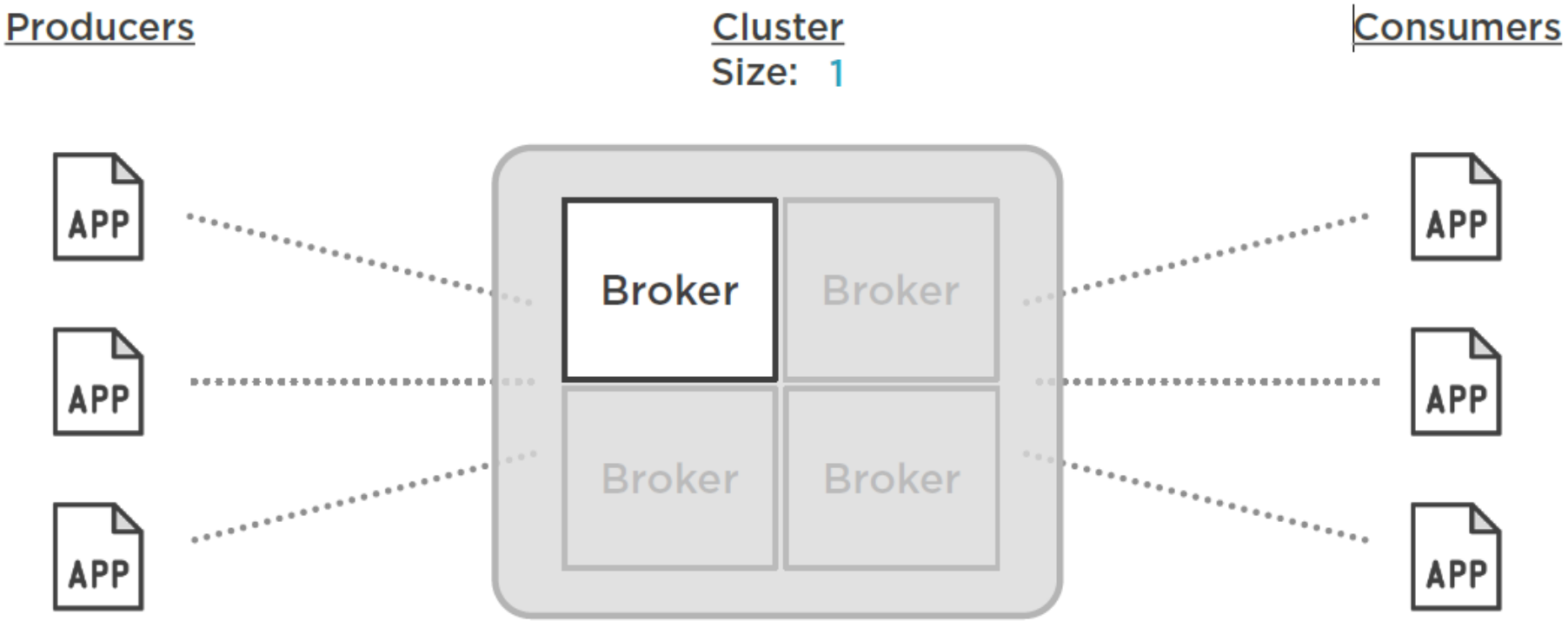


High - throughput distributed messaging system."

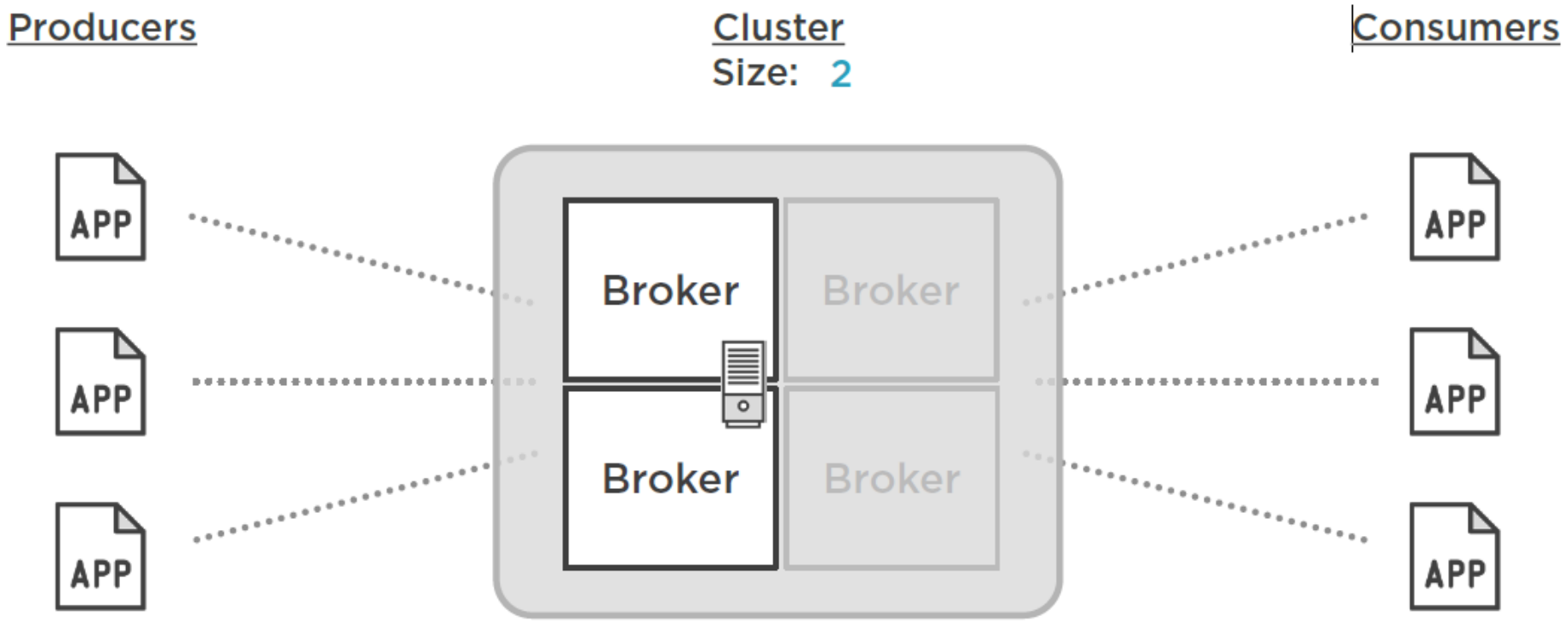
Apache Kafka Cluster



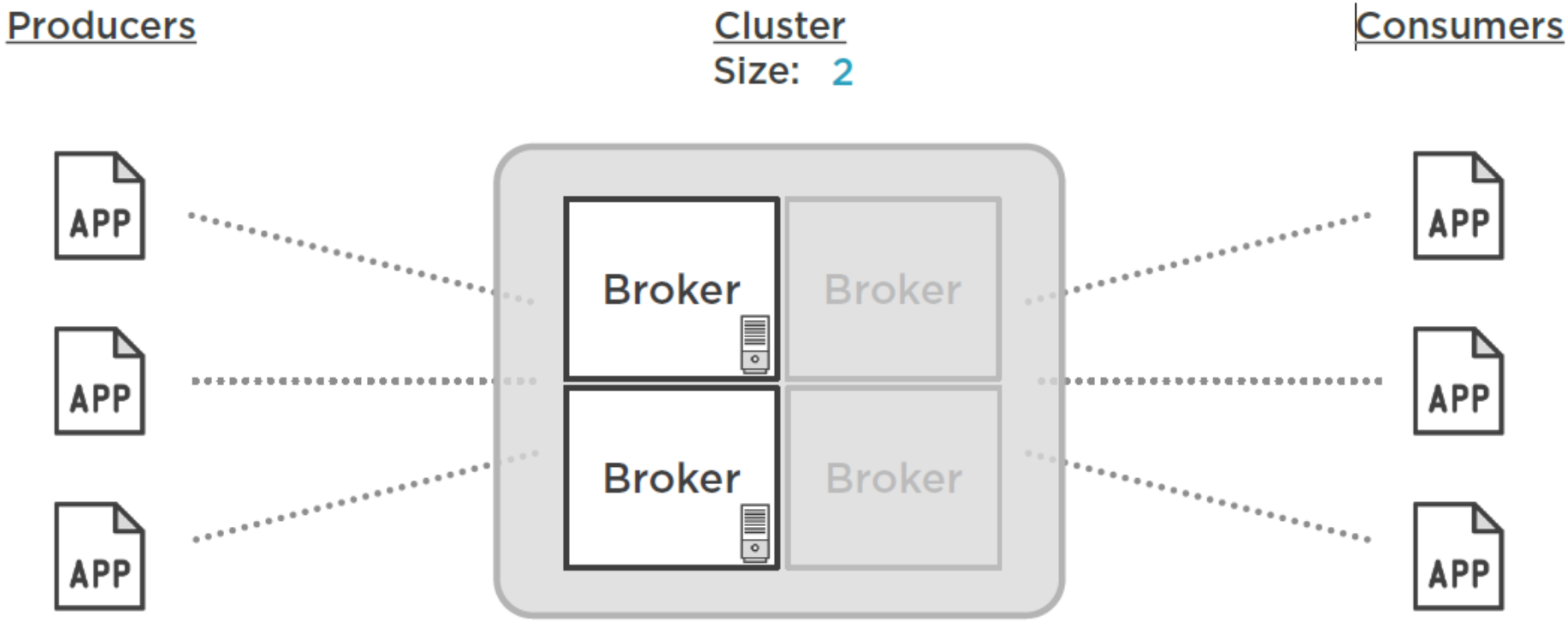
Apache Kafka Cluster



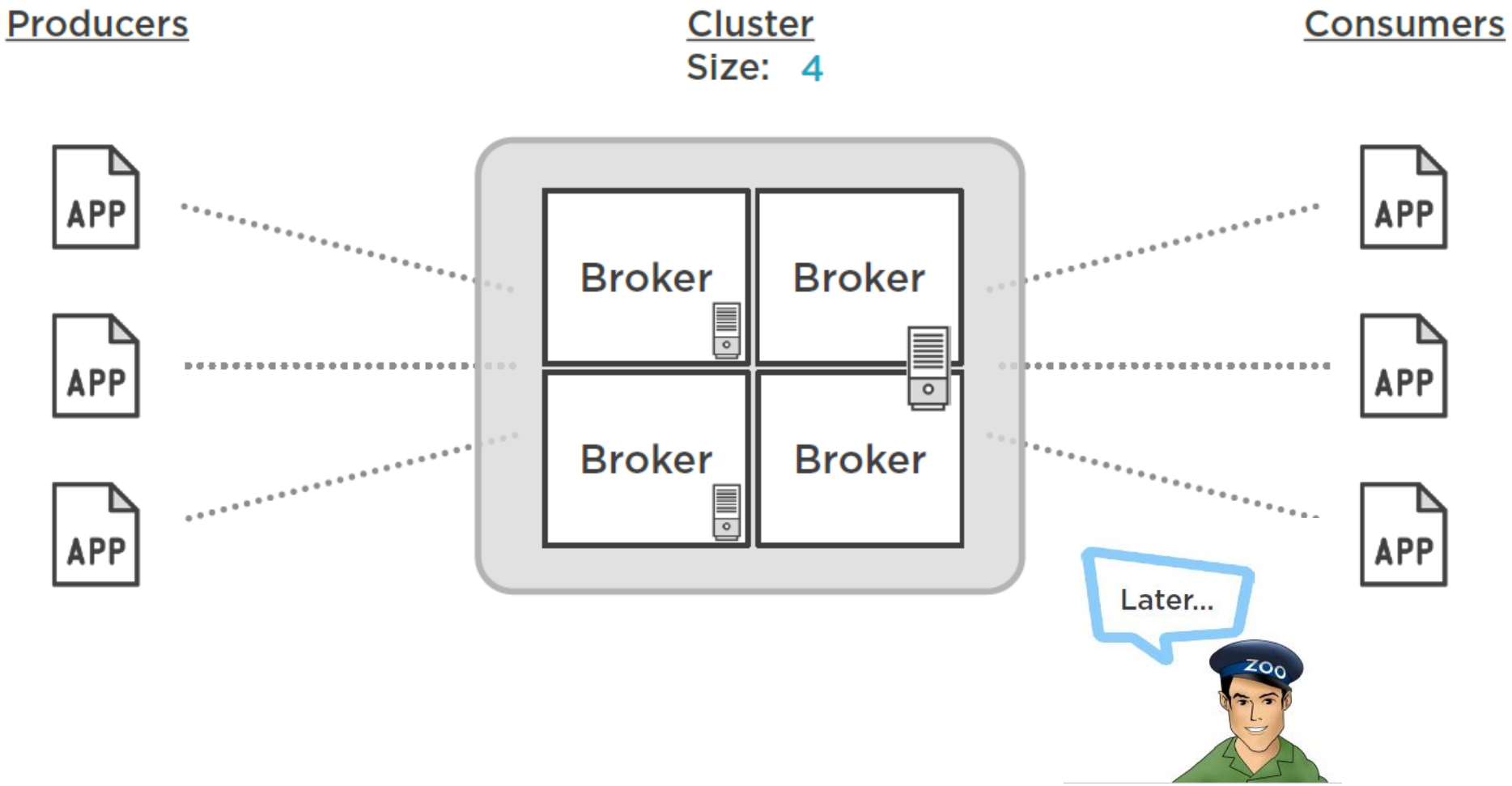
Apache Kafka Cluster



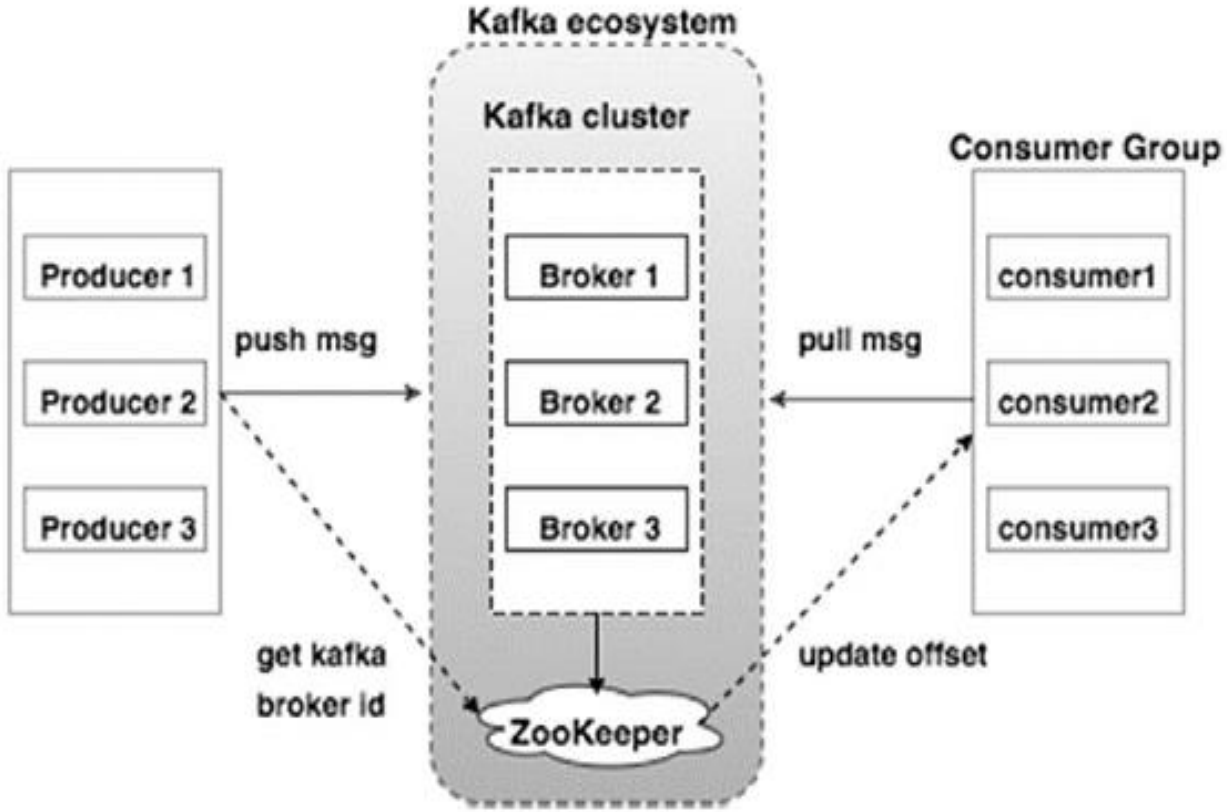
Apache Kafka Cluster



Apache Kafka Cluster



Kafka Architecture



Distributed Systems

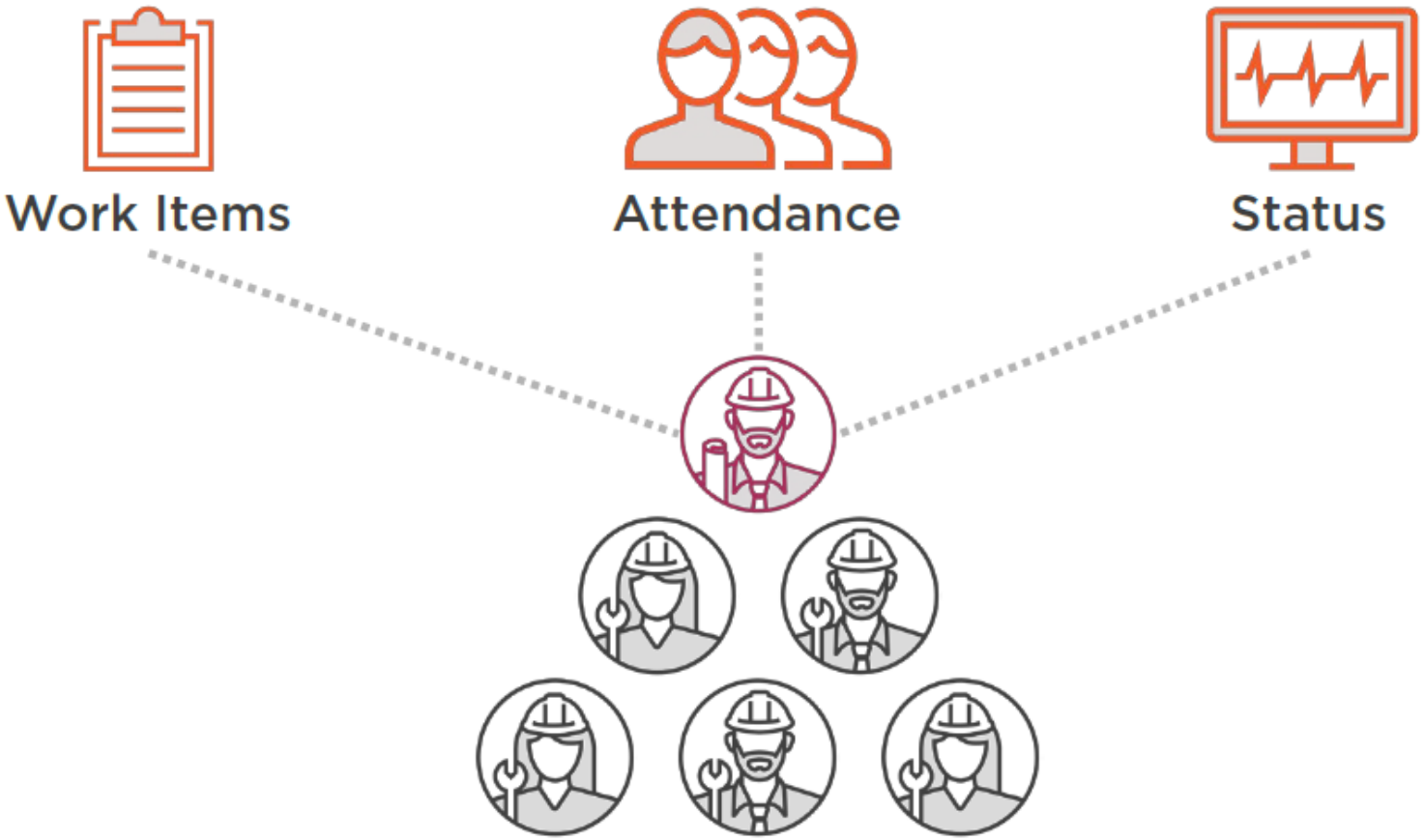
- Collection of resources that are instructed
- to achieve a specific goal or function
- Consist of multiple workers or nodes
- The system of nodes require coordination
- to ensure consistency and progress towards a common goal
- Each node communicates with each other through messages

Distributed Systems

- Collection of resources that are instructed
- to achieve a specific goal or function
- Consist of multiple workers or nodes
- The system of nodes require coordination
- to ensure consistency and progress towards a common goal
- Each node communicates with each other through messages

Distributed Systems

Distributed Systems: Controller Election

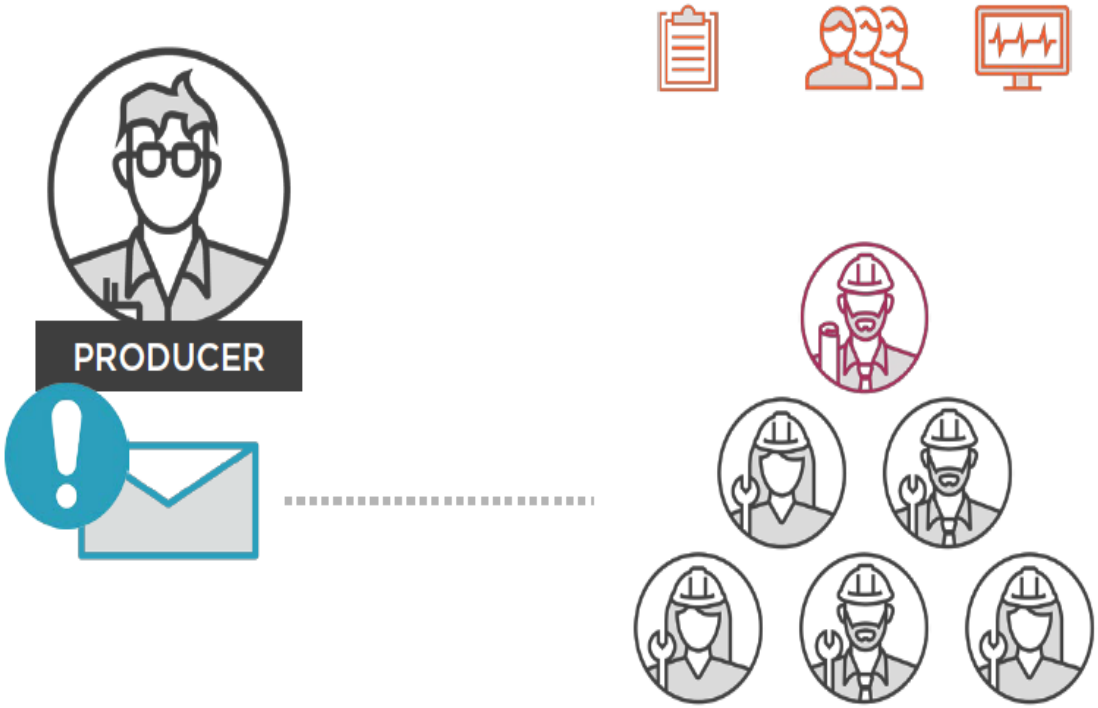


Distributed Systems

Distributed Systems: The Clusters

Distributed Systems

Distributed Systems: Getting Work Done



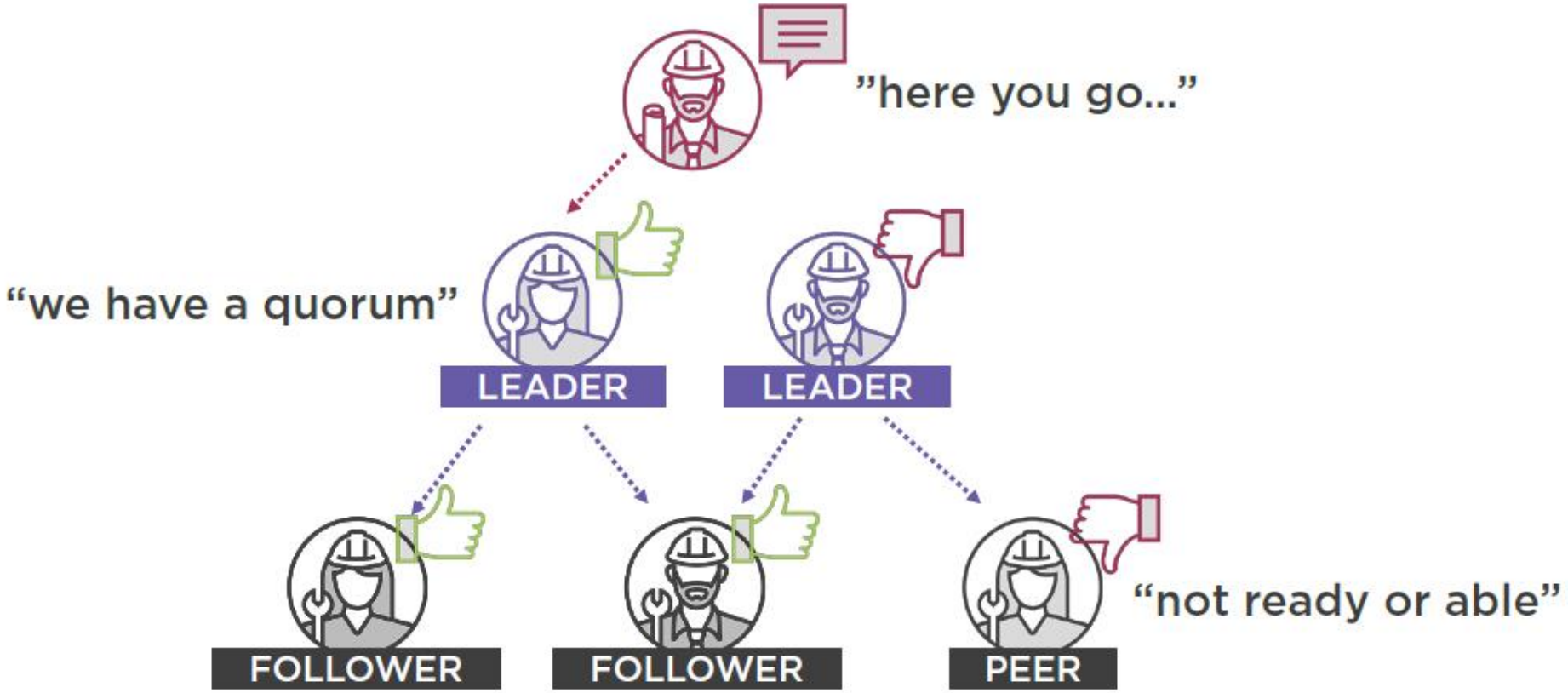
- Worker availability and health
- Task redundancy

Distributed Systems

Distributed Systems: Getting Work Done (Reliably)

Distributed Systems

Distributed Systems: Getting Work Done (Reliably)

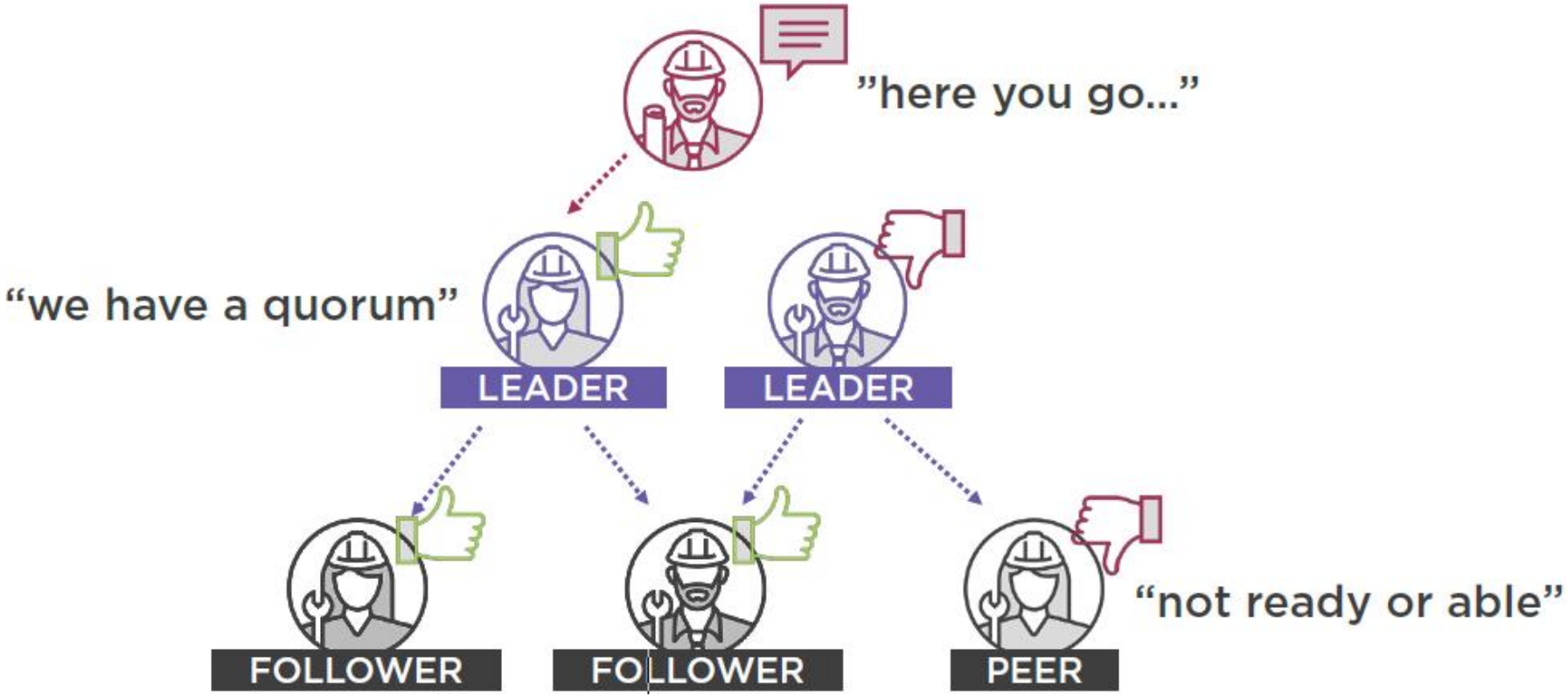


Distributed Systems

Distributed Systems: Getting Work Done (Reliably)

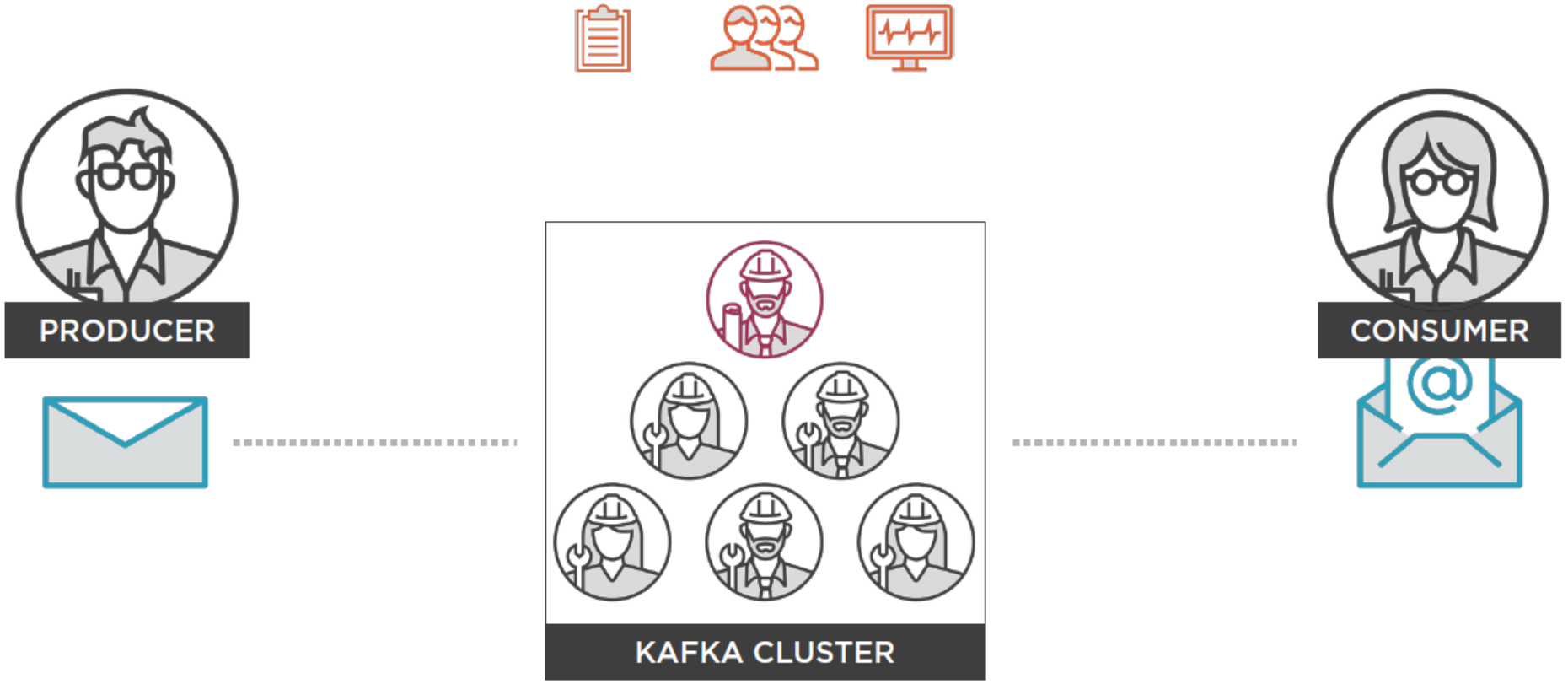
Distributed Systems

Distributed Systems: Getting Work Done (Reliably)



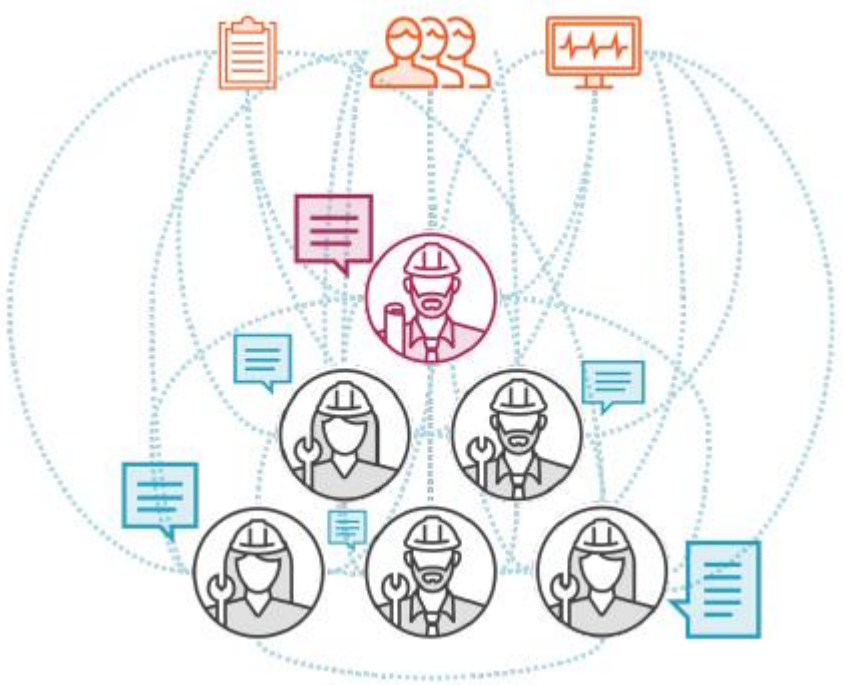
Distributed Systems

Sources of Work in Apache Kafka



Distributed Systems

Distributed Systems: Communication and Consensus

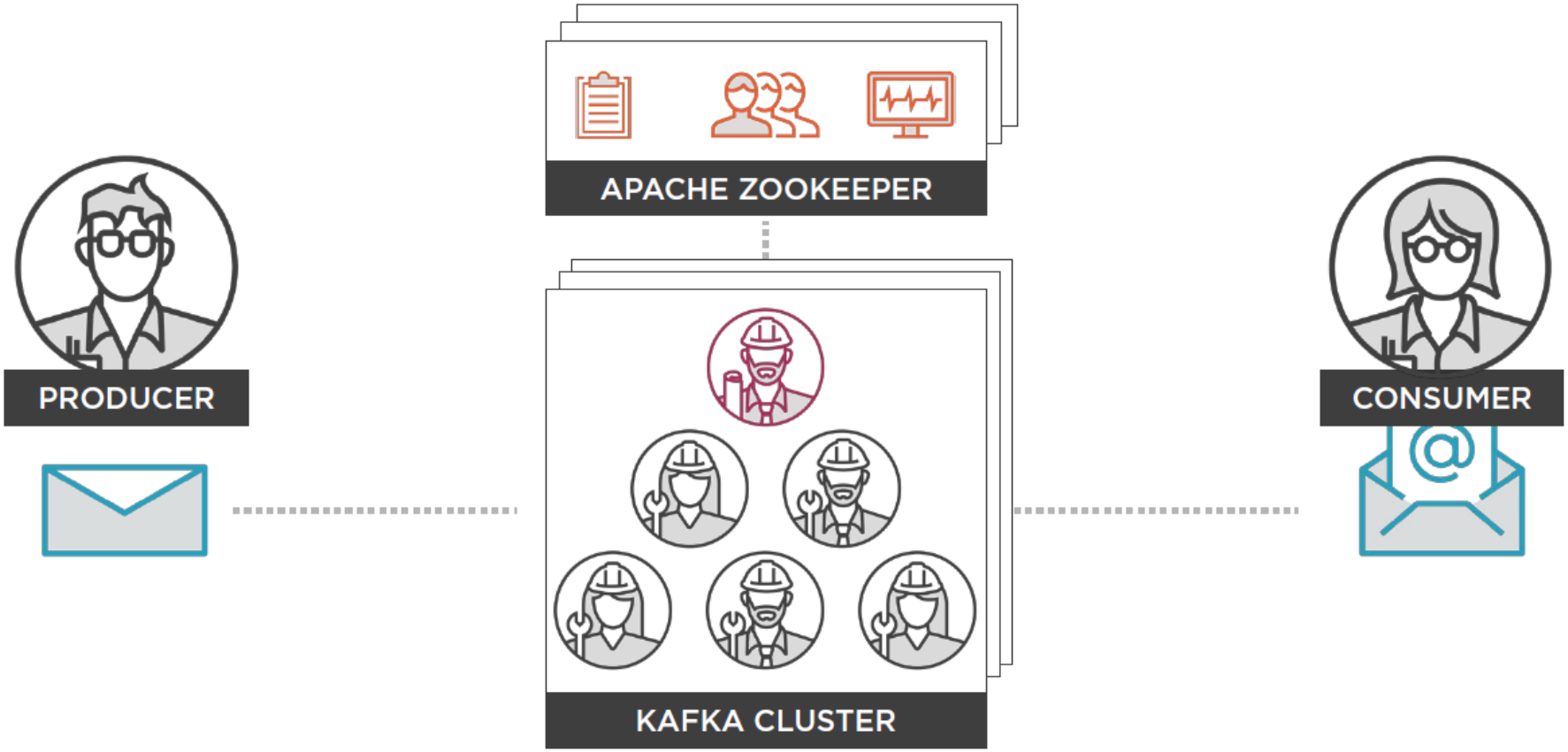


- Worker node membership and naming
- Configuration management
- Leader election
- Health status

Apache Zookeeper

- Centralized service for maintaining metadata about a cluster of distributed nodes
  - Configuration information
  - Health status
  - Group membership
- Hadoop, HBase, Mesos, Solr, Redis, and Neo4j
- Distributed system consisting of multiple nodes in an “ensemble”

Apache Kafka – Distributed Architecture



Kafka Competitors

	ActiveMQ / Apollo	RabbitMQ	ZeroMQ	Kafka	IronMQ	Apcahe Qpid
Brokerless/ Decentralized	No	No	Yes	Distributed	Distributed & Cloud Based	No
Clients	C,C++, Java, Others	C,C++, Java, Others	C,C++, Java, Others	C,C++, Java, Others	C,C++, Java, Others	C,C++, Java, Others
Transaction	Yes	Yes	No	No. But can be implemented with plugin	No	Yes
Persistence/ Reliability	Yes (configurable)	Yes (built-in)	No persistence – requiring higher layers to manage persistence	Yes (built-in - File system)	Yes (built-in)	Yes (additional Plugin required)
Routing	Yes (easier to implment)	Yes (easier to implment)	Yes (complex to implment)	Can be implemented	No	No
Failover/ HA	Yes	Yes	No	Yes	Yes	Yes

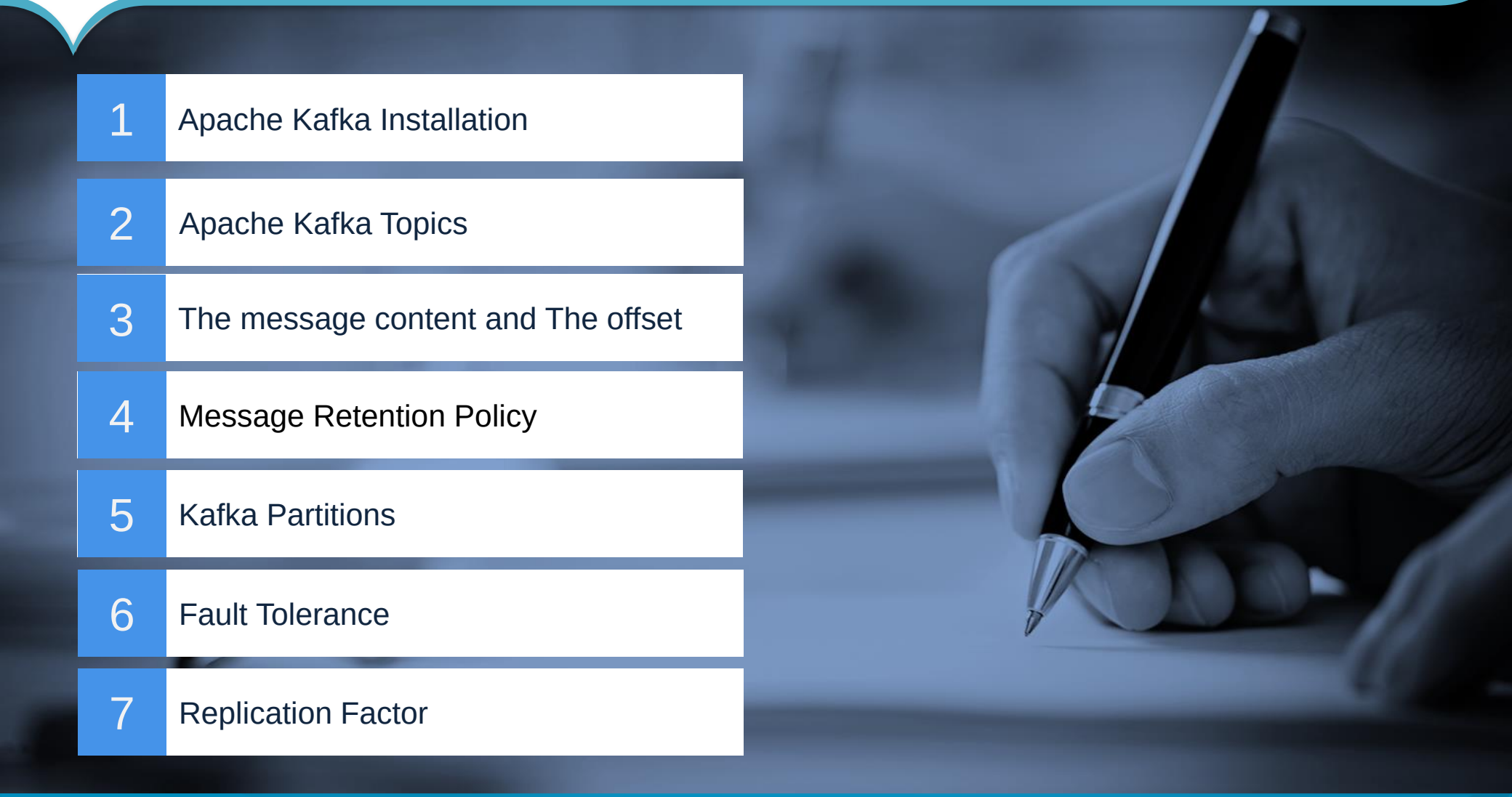
Kafka Use Cases

- Metrics / KPIs gathering
  - Aggregate statistics from many sources
- Even Sourcing
  - Used with microservices (in-memory) and actor systems
- Commit Log
  - External commit log for distributed systems. Replicated data between nodes, re-sync for nodes to restore state
- Real-time data analytics, Stream Processing, Log Aggregation, Messaging, Click-stream tracking, Audit trail, etc.

Summary

- Apache Kafka is a Pub - Sub messaging system, consisting of:
  - Producers and Consumers
  - Brokers within a Cluster
- Characteristics of distributed systems
  - Worker node roles: Controllers, Leaders, and Followers
  - Reliability through replication
  - Consensus-based communication
- Role of Apache Zookeeper

Course Map – Topics, Partitions, and Brokers



1. Apache Kafka Installation
2. Apache Kafka Topics
3. The message content and The offset
4. Message Retention Policy
5. Kafka Partitions
6. Fault Tolerance
7. Replication Factor

Apache Kafka Installation

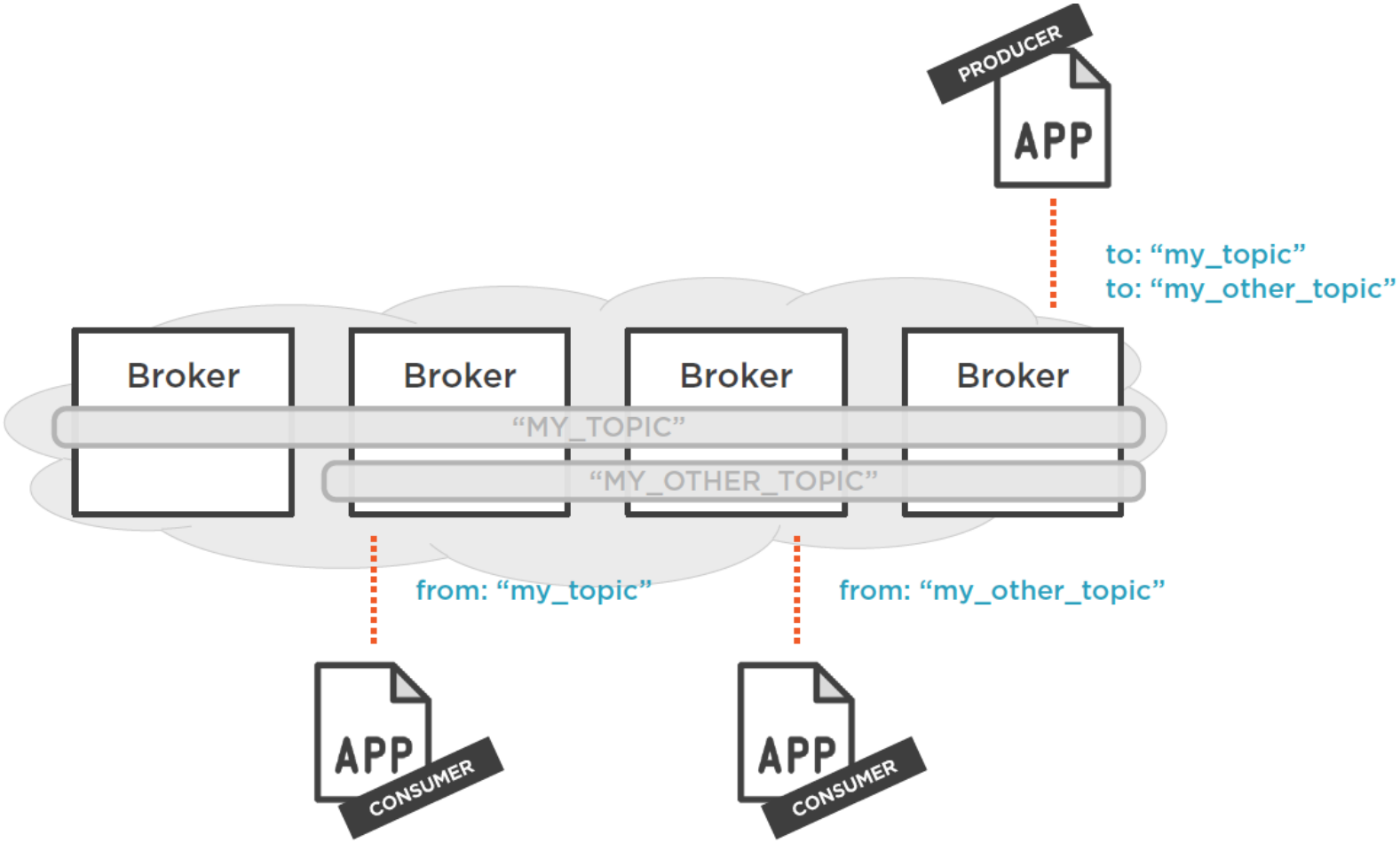
- Basic Apache Kafka installation:
  - Download the binary package
  - Extract the archive
  - Explore the installation directory contents

- Prerequisites:
  - Windows/Linux operating system
  - Java 8 JDK installed
  - Scala 2.11.x installed (optional)

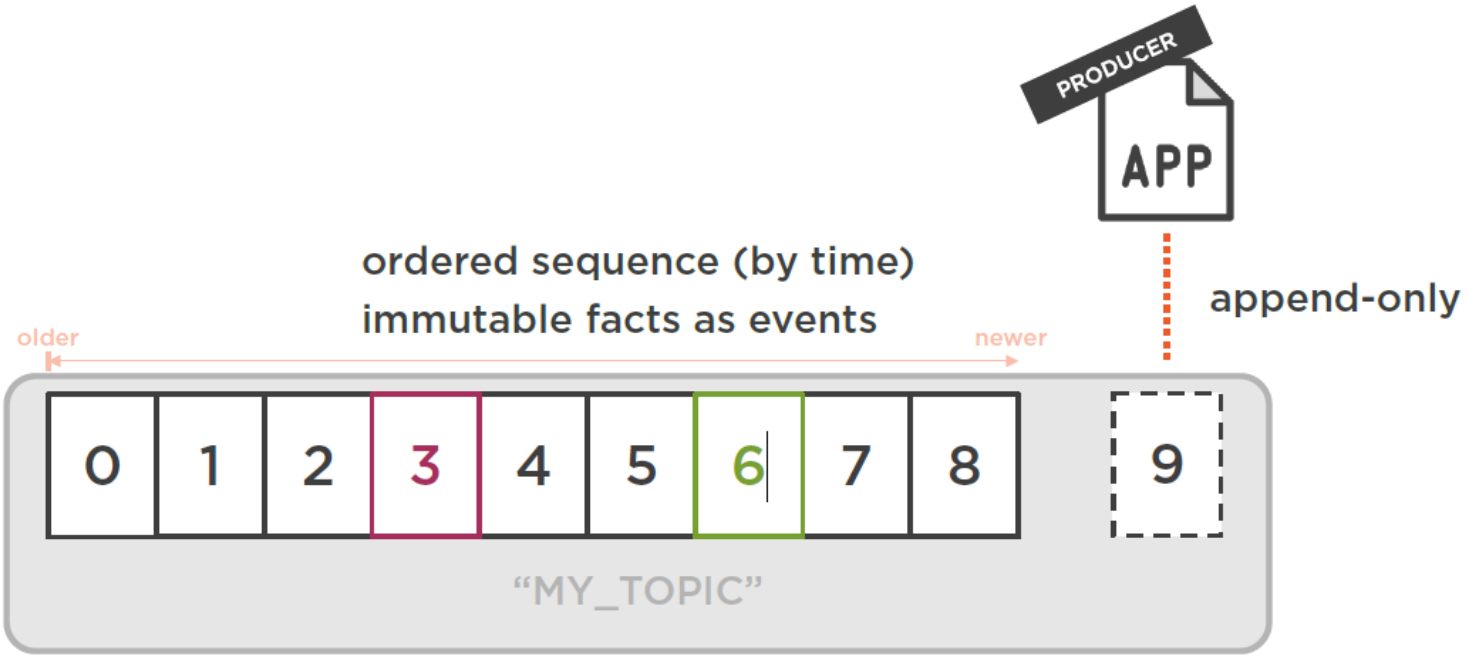
Apache Kafka Topics

- Central Kafka abstraction
- Named feed or category of messages
  - Producers produce to a topic
  - Consumers consume from a topic
- Logical entity
- Physically represented as a log

Apache Kafka Topics



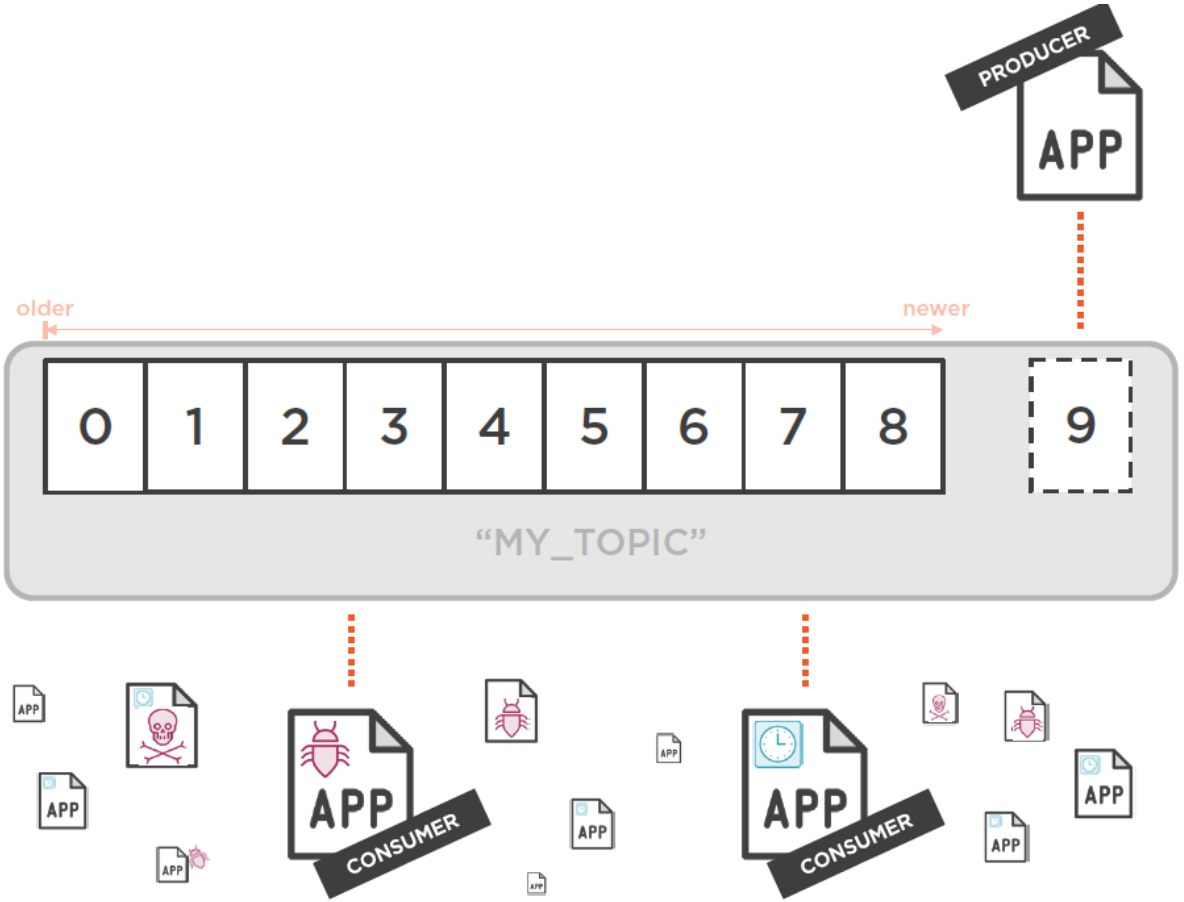
Apache Kafka Topics



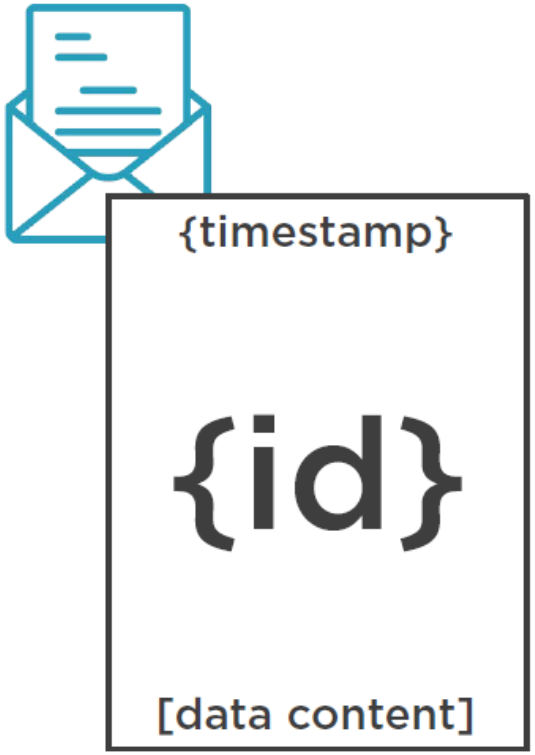
Event Sourcing

An architectural style or approach to maintaining an application's state by capturing all changes as a sequence of time-ordered, immutable events.

Apache Kafka Topics

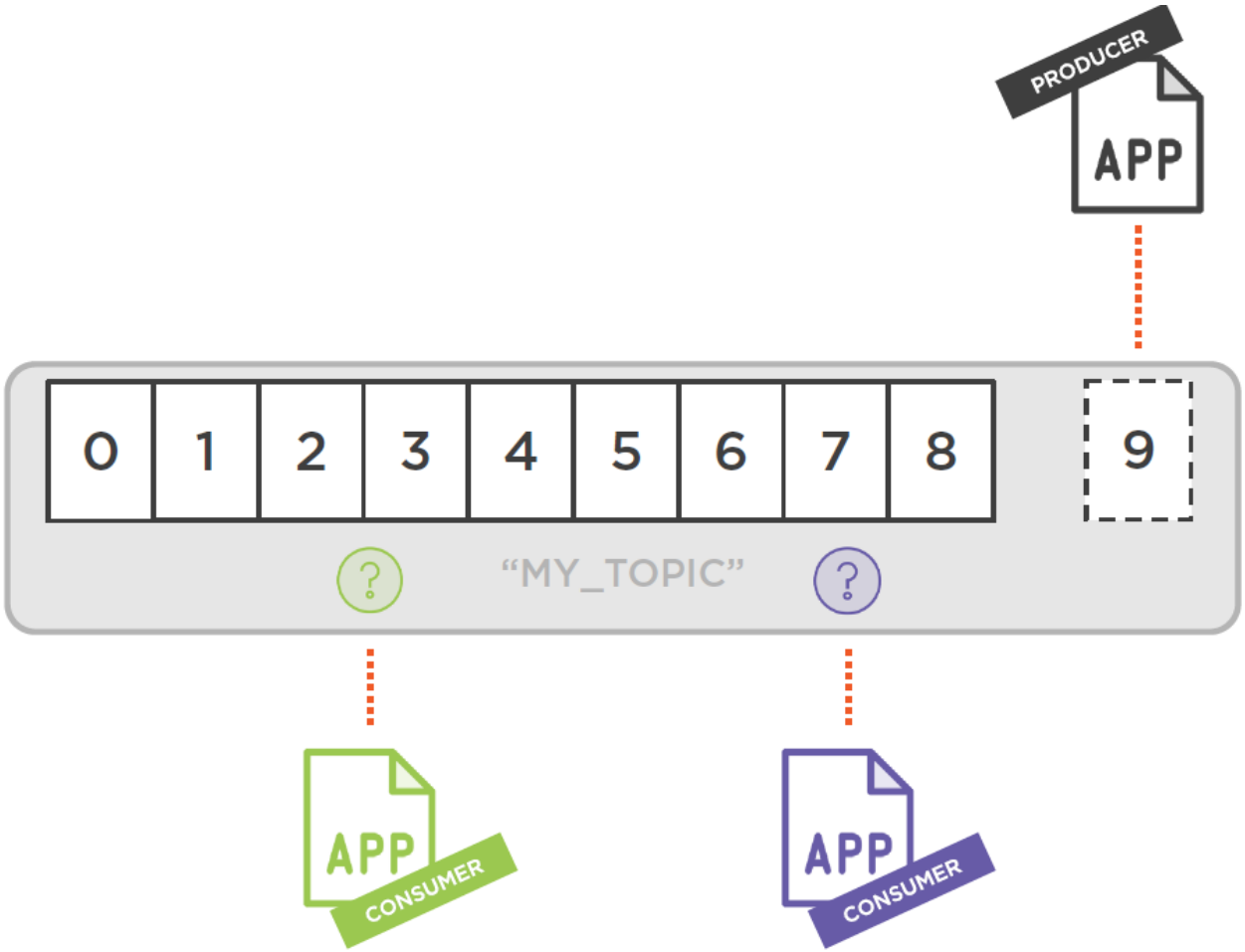


Message Content

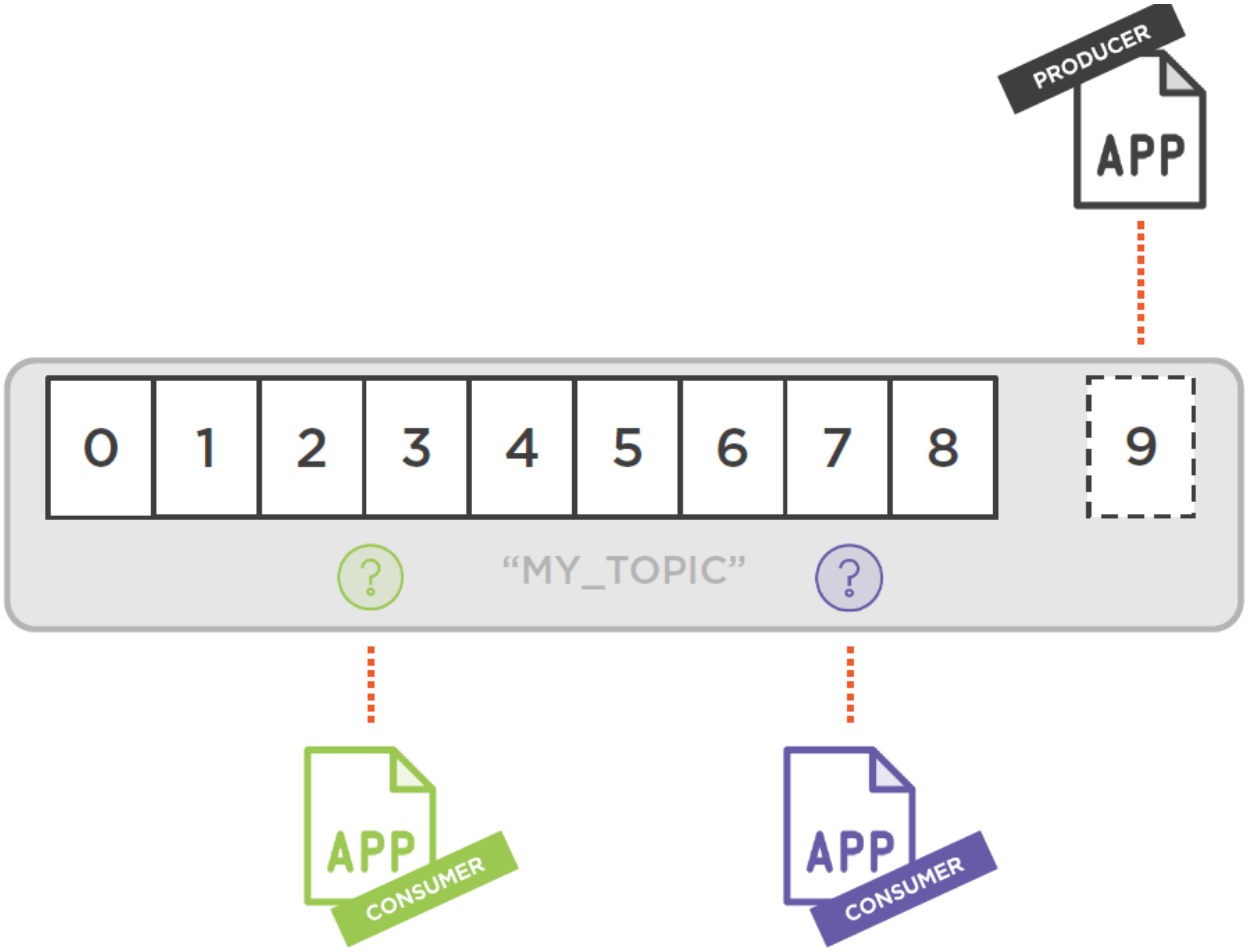


- Timestamp
- Referenceable identifier
- Payload (binary)

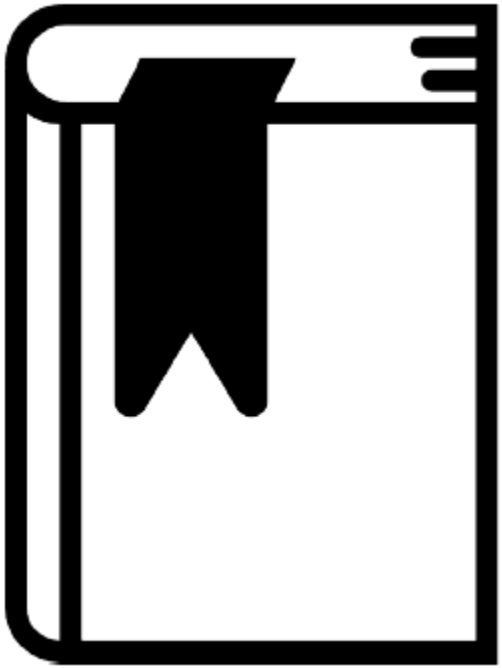
Message Content



Message Content

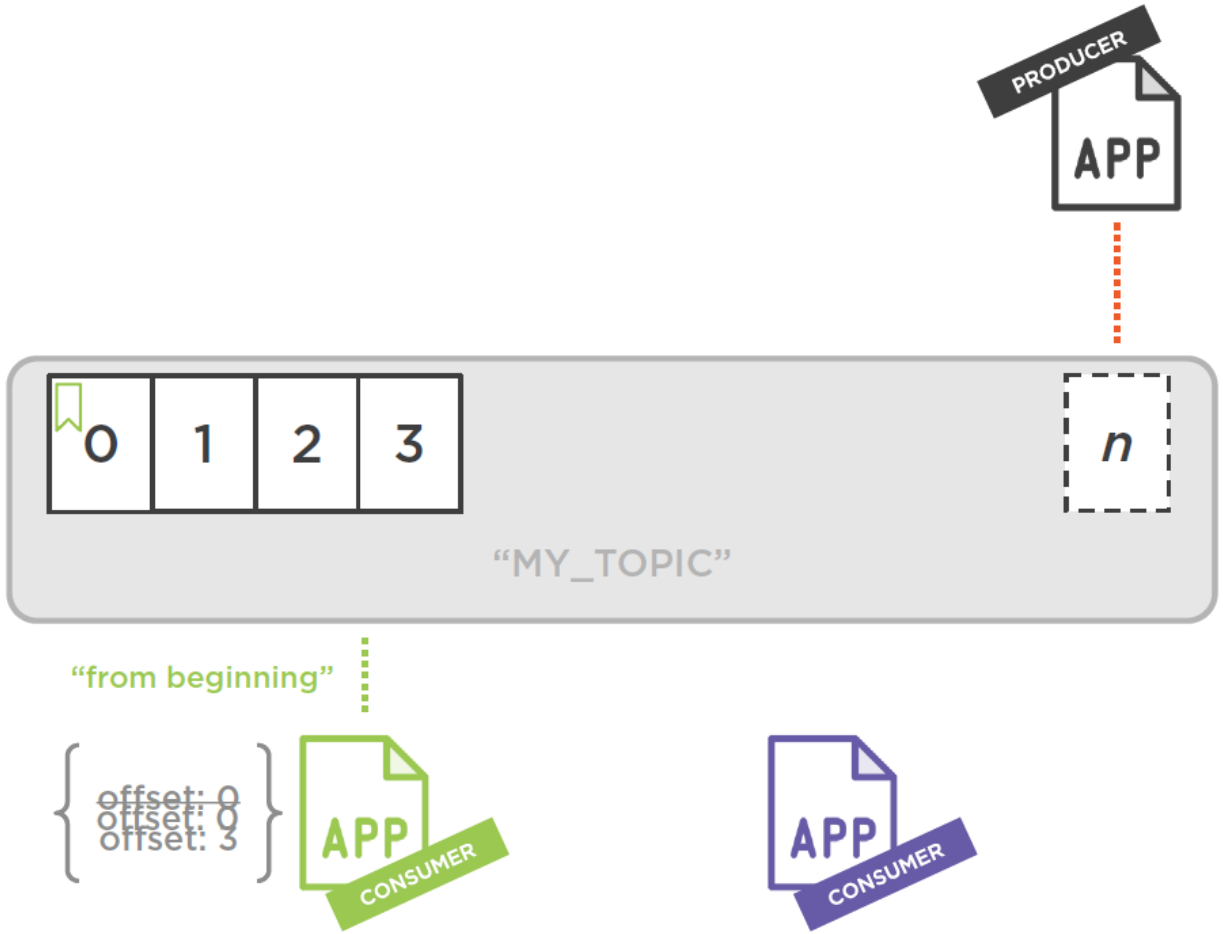


The offset

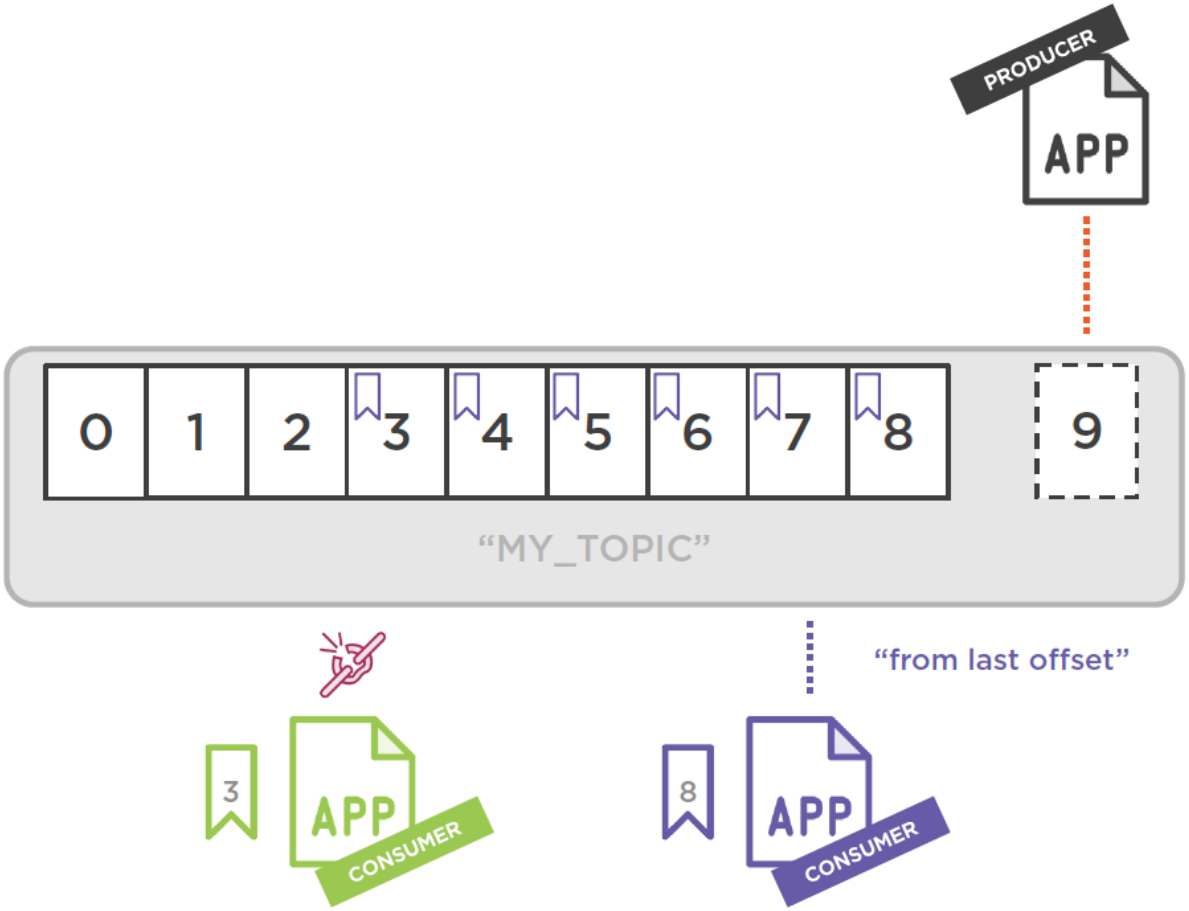


- A placeholder:
  - Last read message position
  - Maintained by the Kafka Consumer
  - Corresponds to the message identifier

The offset



The offset



Message Retention Policy

- Apache Kafka retains all published messages regardless of consumption
- Retention period is configurable
  - Default is 168 hours or seven days
- Retention period is defined on a per-topic basis
- Physical storage resources can constrain message retention

Message Retention Policy

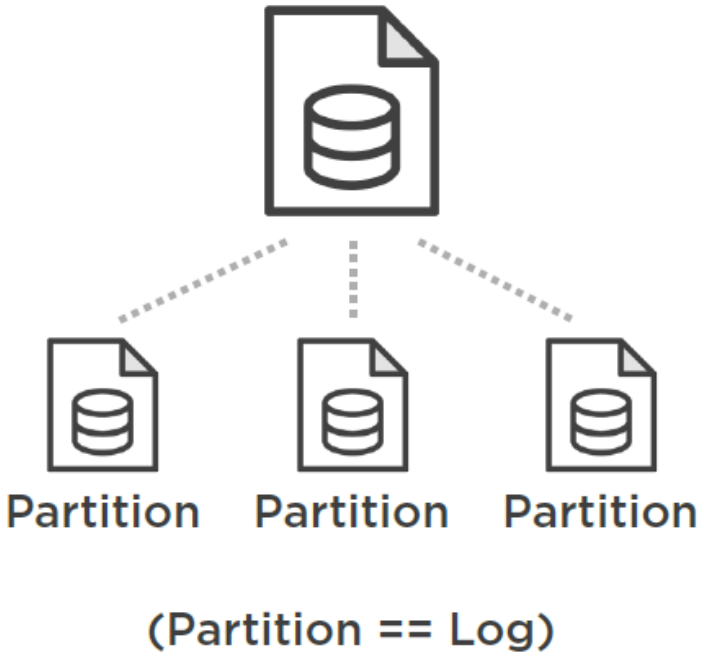
- Does Look This Look Familiar?
  - Append only
  - Ordered sequence (by time)
  - Immutable facts as events

Transaction or Commit Logs

- Source of truth
- Physically stored and maintained
- Higher-order data structures derive from the log
  - Tables, indexes, views, etc.
- Point of recovery
- Basis for replication and distribution

Apache Kafka is publish-subscribe messaging rethought as a distributed commit log.

Kafka Partitions



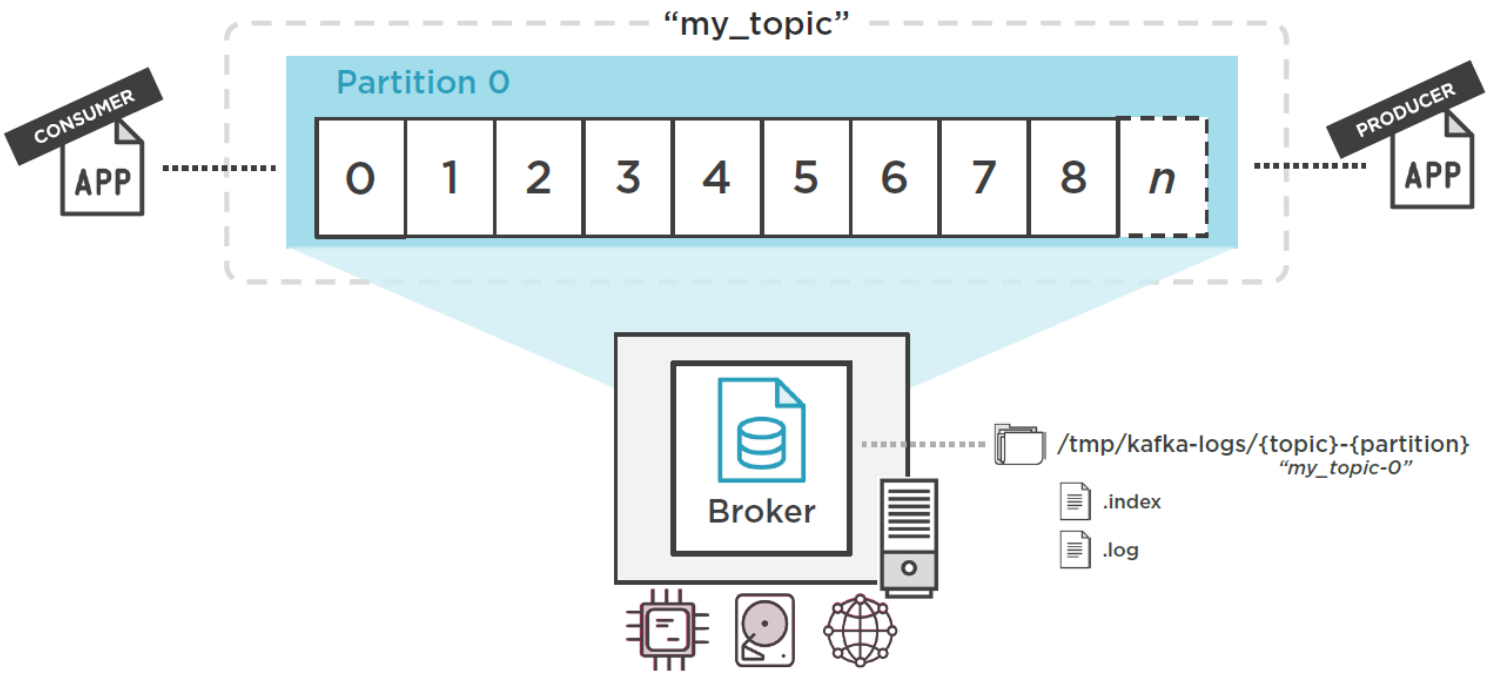
- Each topic has one or more partitions
- A partition is the basis for which Kafka can:
  - Scale
  - Become fault-tolerant
  - Achieve higher levels of throughput
- Each partition is maintained on at least one or more Brokers

Kafka Partitions

- Creating a Topic: Single Partition

```
~$ bin/kafka-topics.sh --create --topic my_topic\
> --zookeeper localhost:2181 \
> --partitions 1 \
> --replication-factor 1
```

Kafka Partitions



Each partition must fit entirely on one machine.

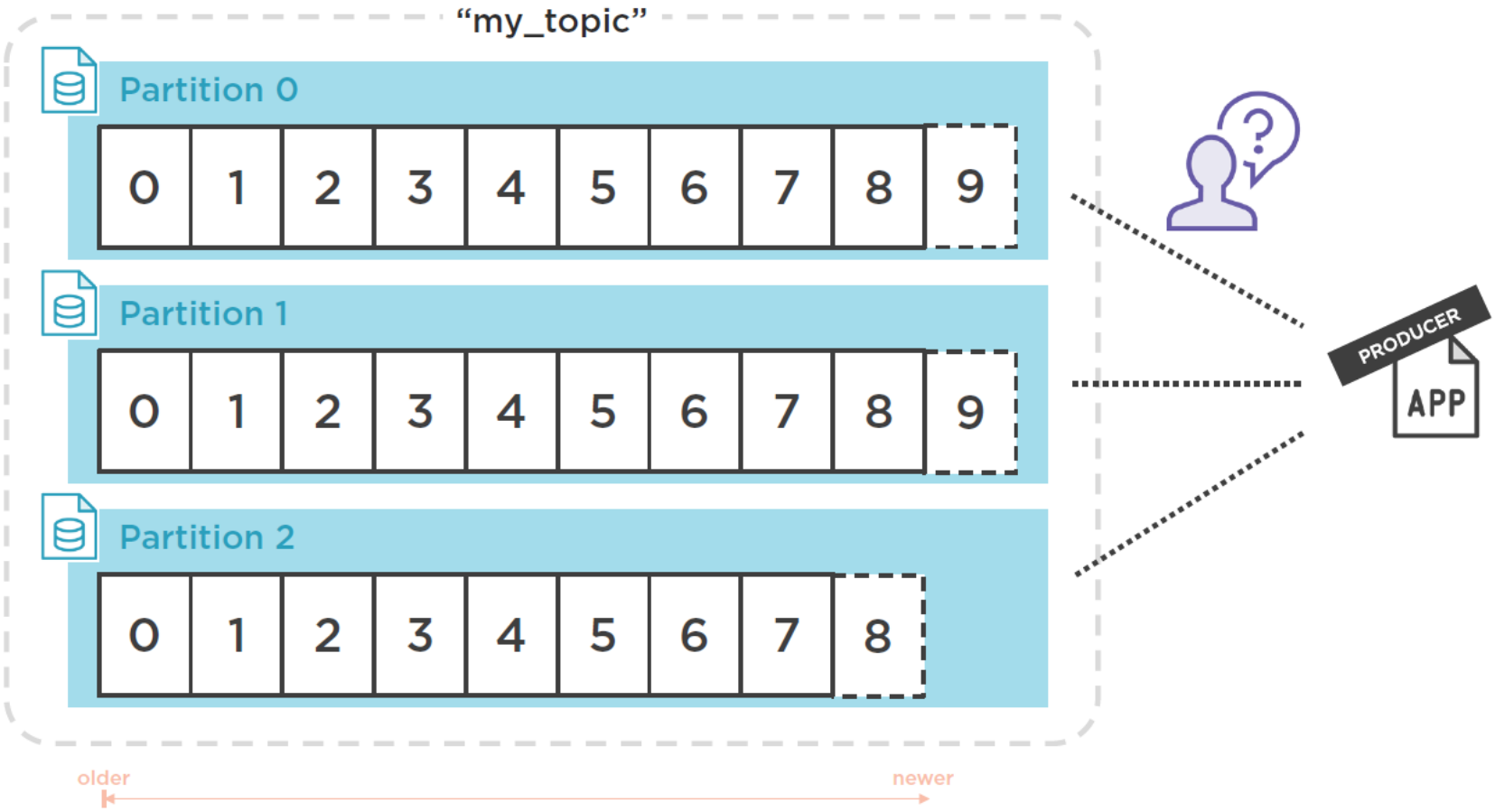
In general, the scalability of Apache Kafka is determined by the number of partitions being managed by multiple broker nodes.

Kafka Partitions

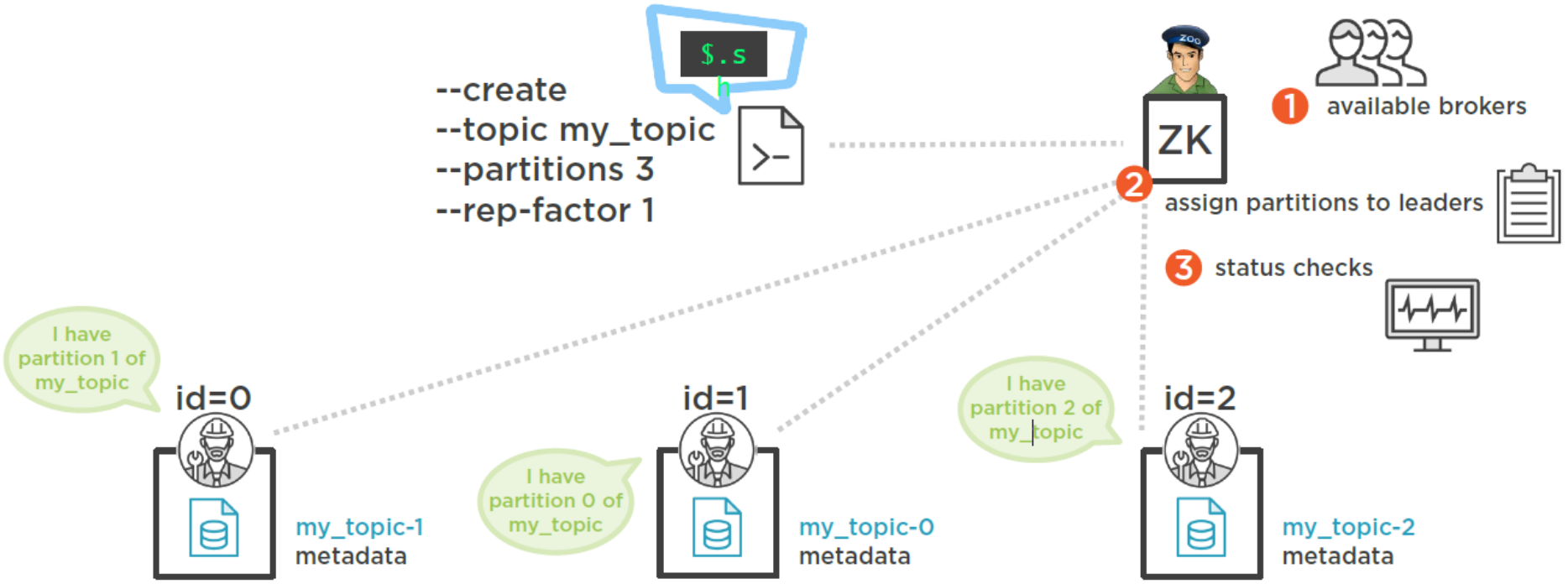
- Creating a Topic: Multiple Partitions

```
~$ bin/kafka-topics.sh --create --topic my_topic\
> --zookeeper localhost:2181 \
> --partitions 3 \
> --replication-factor 1
```

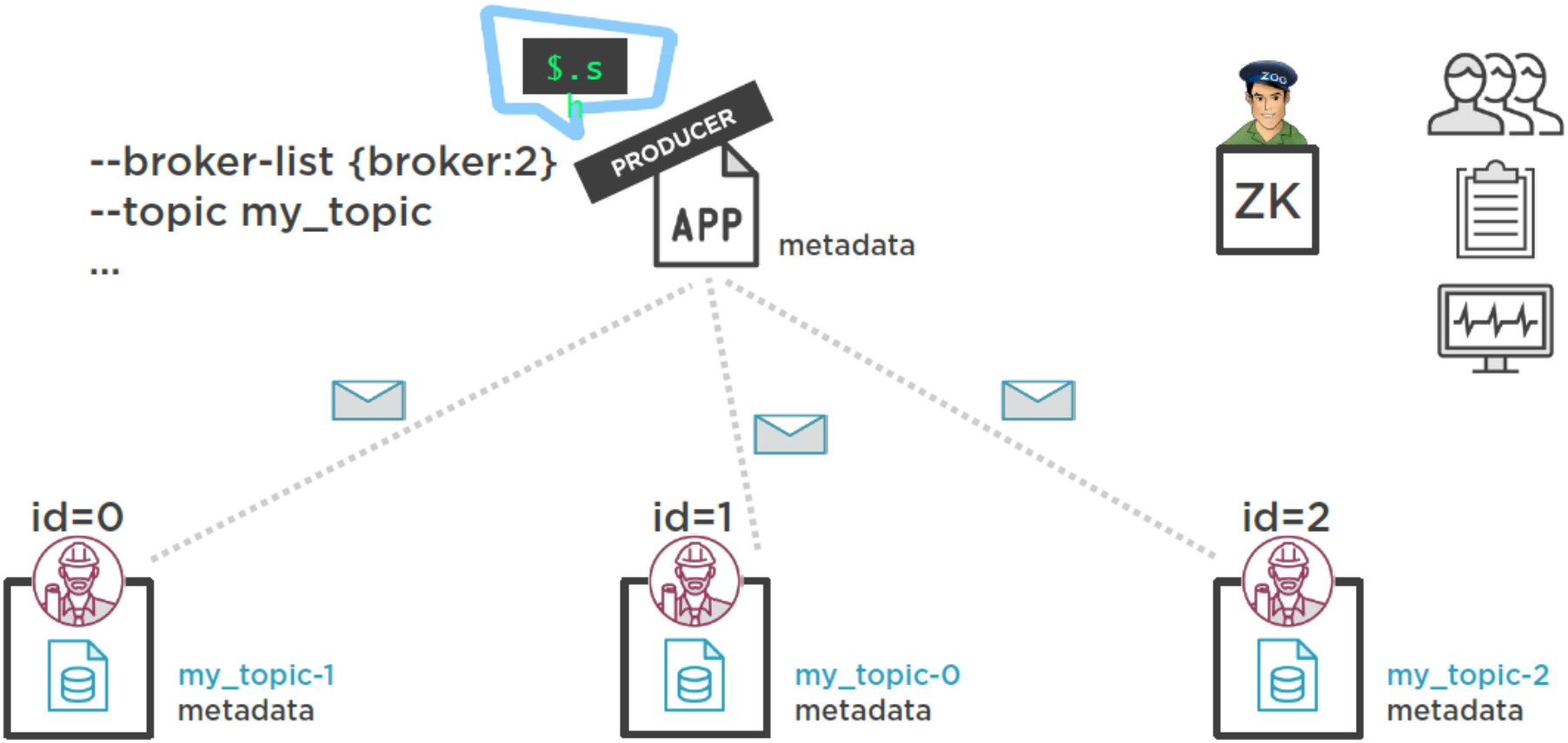
Kafka Partitions



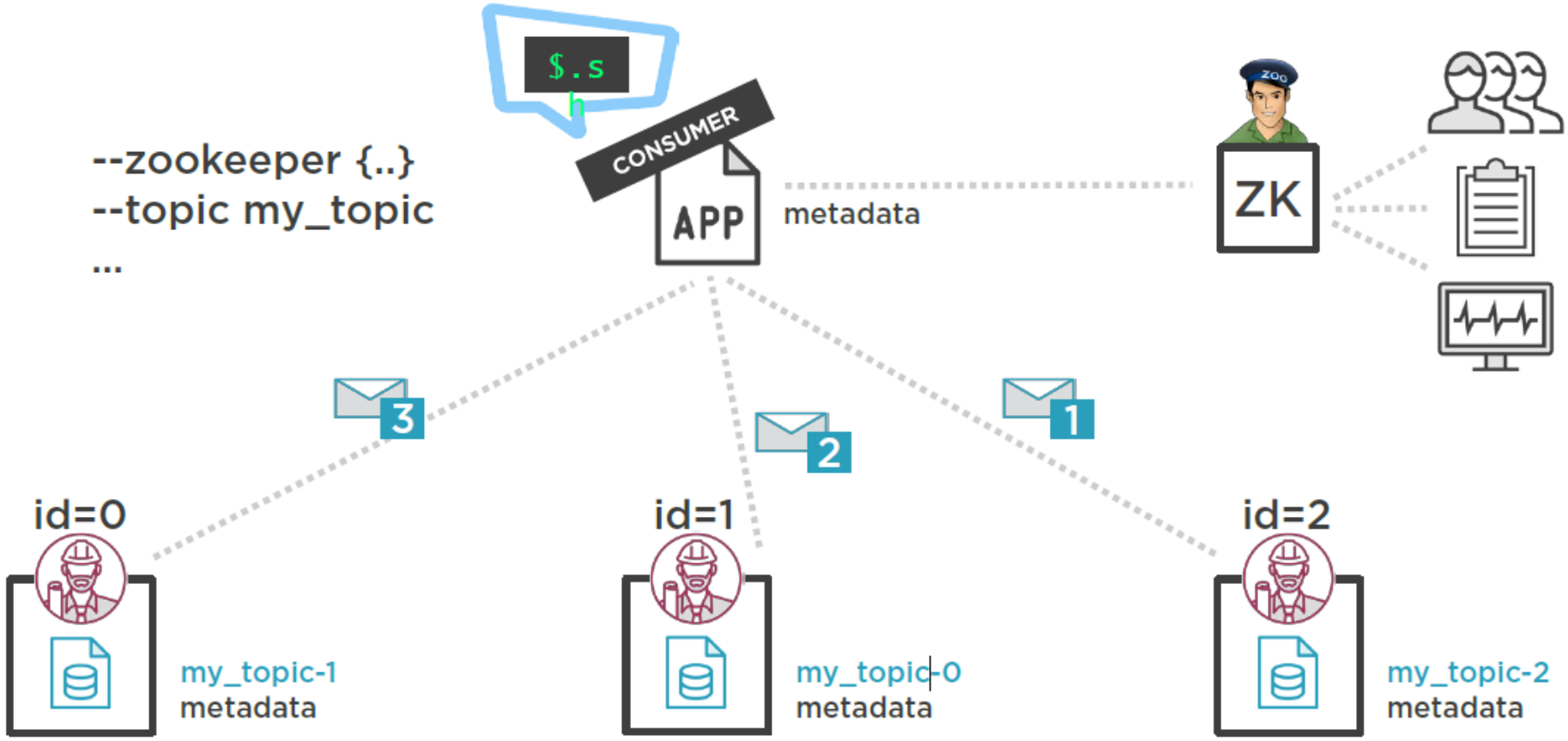
Kafka Partitions



Kafka Partitions



Kafka Partitions

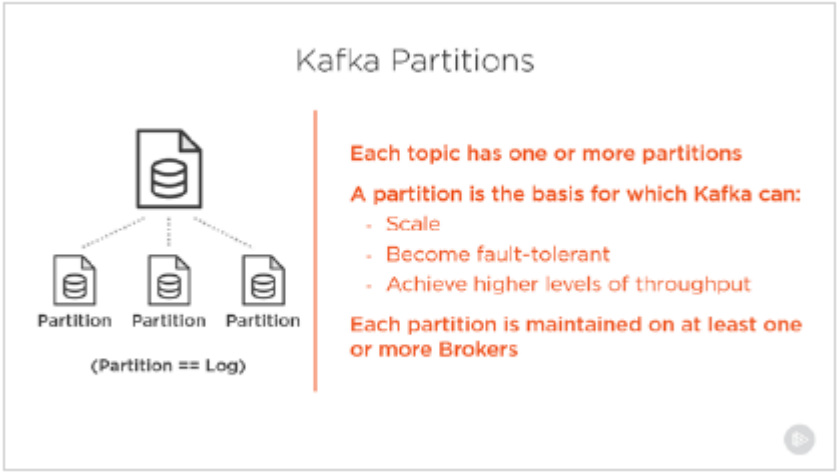


Kafka Partitions

Partitioning Trade-offs

- The more partitions the greater the Zookeeper overhead
  - With large partition numbers ensure proper ZK capacity
- Message ordering can become complex
  - Single partition for global ordering
  - Consumer-handling for ordering
- The more partitions the longer the leader fail-over time

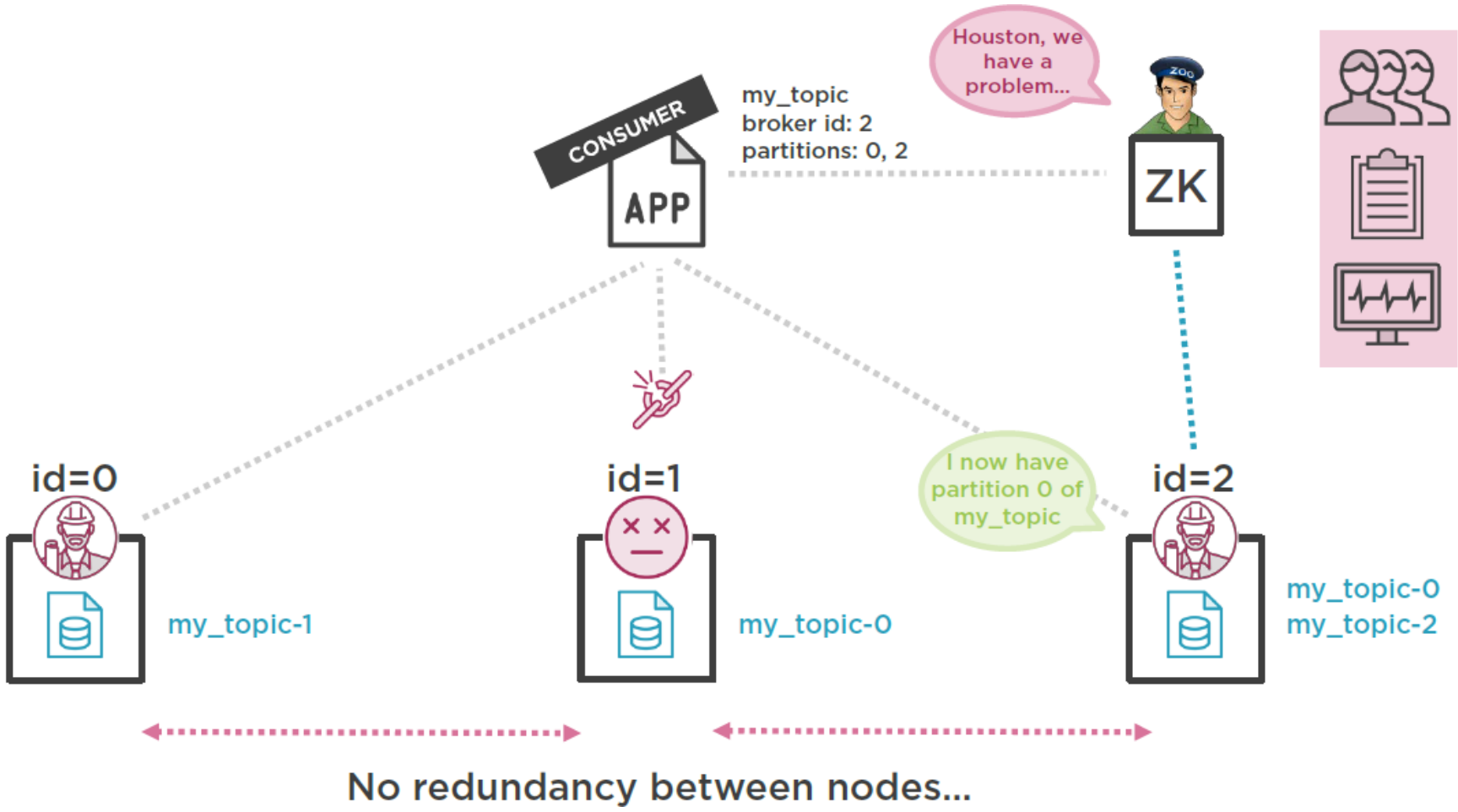
Fault-Tolerance



Something Is Missing

- What about fault-tolerance?
  - Broker failure
  - Network issue
  - Disk failure

Fault-Tolerance

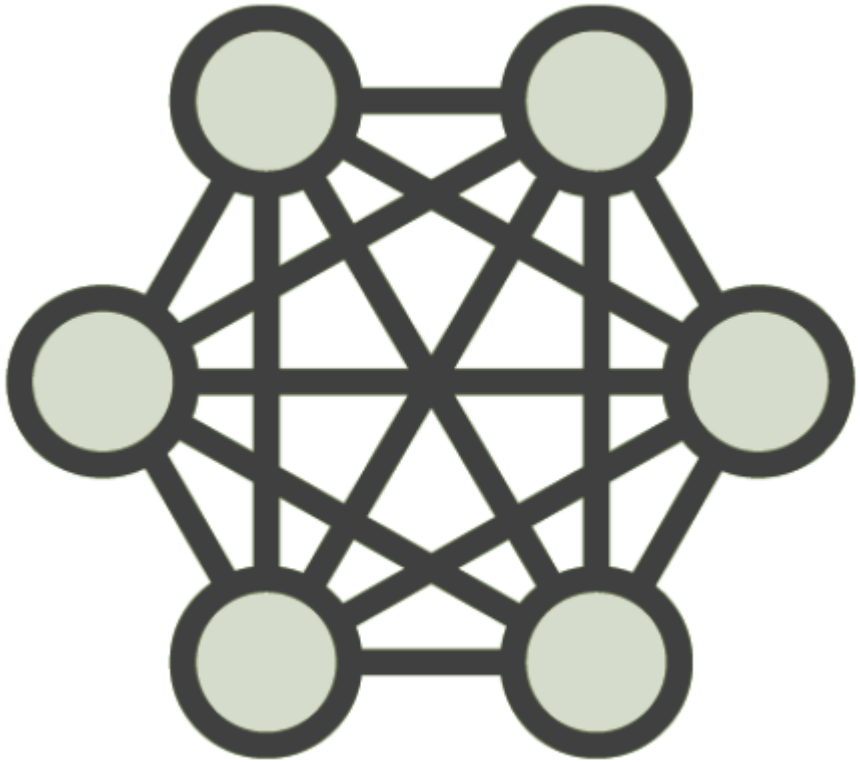


Fault-Tolerance

- Don’t Forget the Replication Factor

```
~$ bin/kafka-topics.sh --create --topic my_topic\
> --zookeeper localhost:2181 \
> --partitions 3 \
> --replication-factor 1
```

Replication Factor



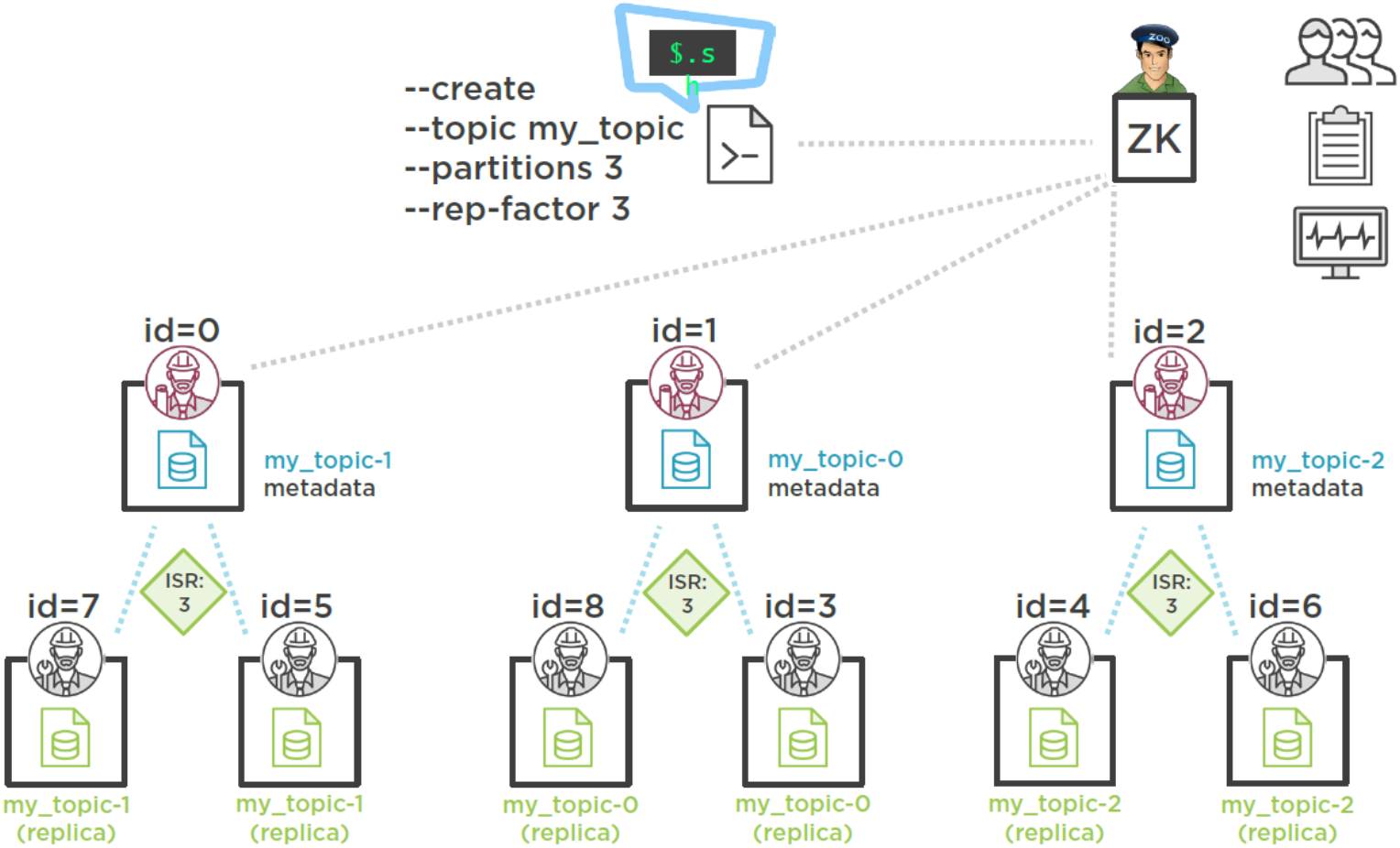
- **Reliable work distribution**
  - **Redundancy of messages**
  - **Cluster resiliency**
  - **Fault-tolerance**
- **Guarantees**
  - **N-1 broker failure tolerance**
  - **2 or 3 minimum**
- **Configured on a per-topic basis**

Replication Factor

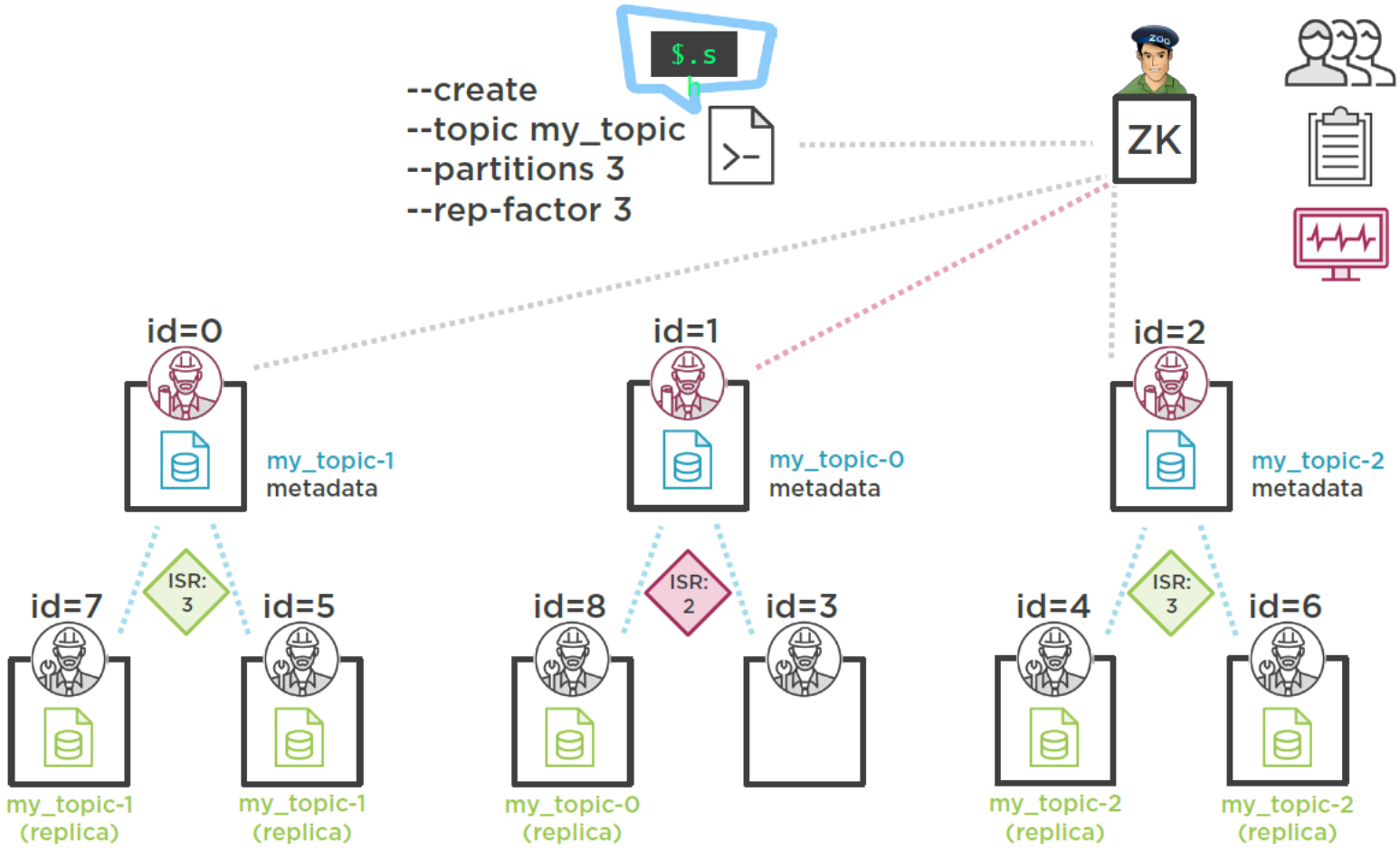
- Multiple Replica Sets

```
~$ bin/kafka-topics.sh --create --topic my_topic\
> --zookeeper localhost:2181 \
> --partitions 3 \
> --replication-factor 3
```

Replication Factor



Replication Factor



Replication Factor

- Viewing Topic State

```
~$ bin/kafka-topics.sh --describe --topic my_topic\
> --zookeeper localhost:2181
```

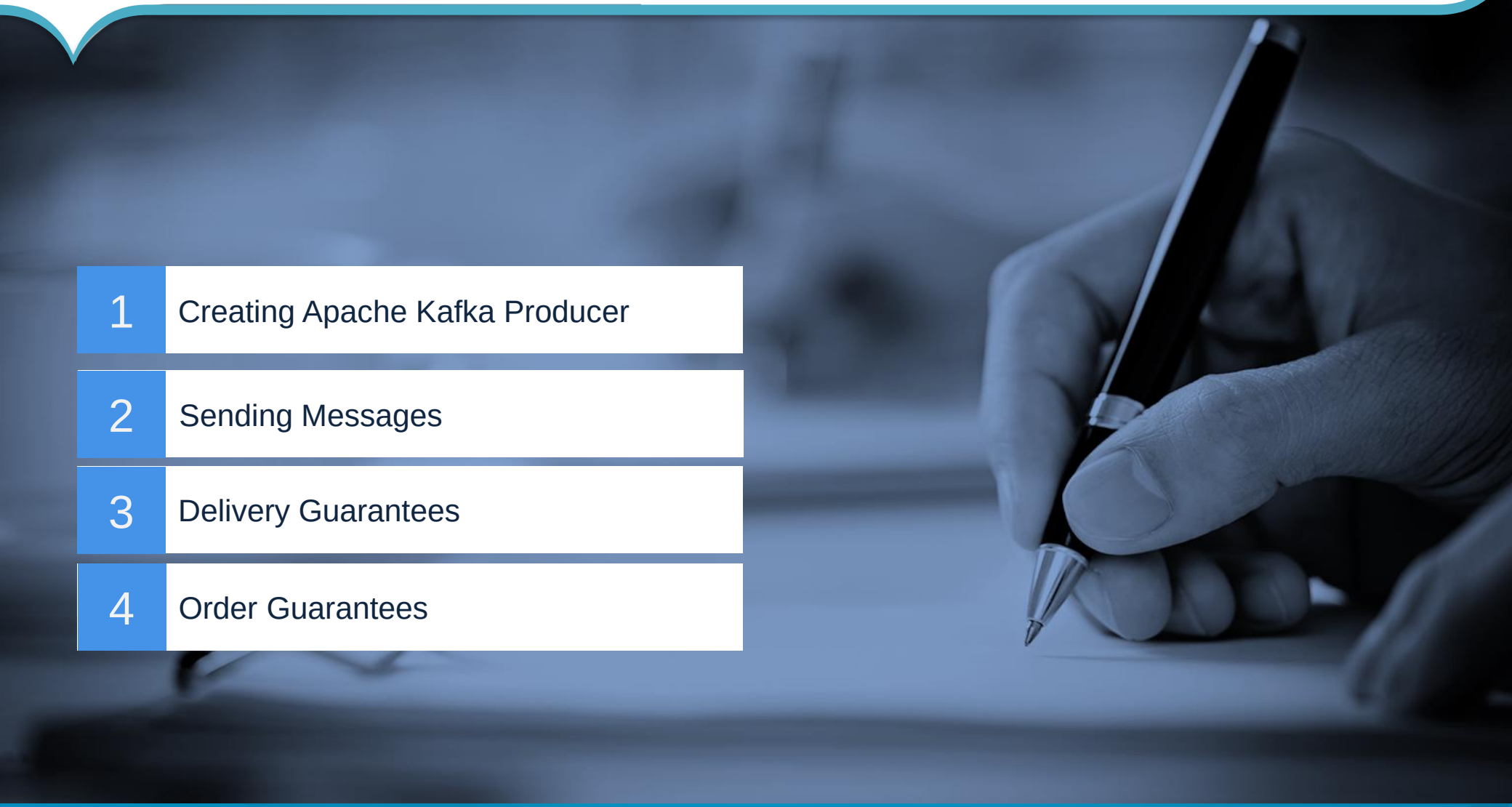
Summary

- Detailed explanation and view:
  - Topics and Partitions
  - Broker partition management and behavior
- Aligned with distributed systems principles
  - Leader election of partitions
  - Work distribution and failover
- Kafka in action
  - Demos
  - Configuration
- Foundation upon which to dive deeper into Producers and Consumers

Apache Kafka Tools

- Consumer Offset Checker
- Dump Log Segment
- Export Zookeeper Offsets
- Get Offset Shell
- Import Zookeeper Offsets
- JMX Tool
- Kafka Migration Tool
- Mirror Maker
- Replay Log Producer
- Simple Consumer Shell
- State Change Log Merger
- Update Offsets In Zookeeper
- Verify Consumer Rebalance

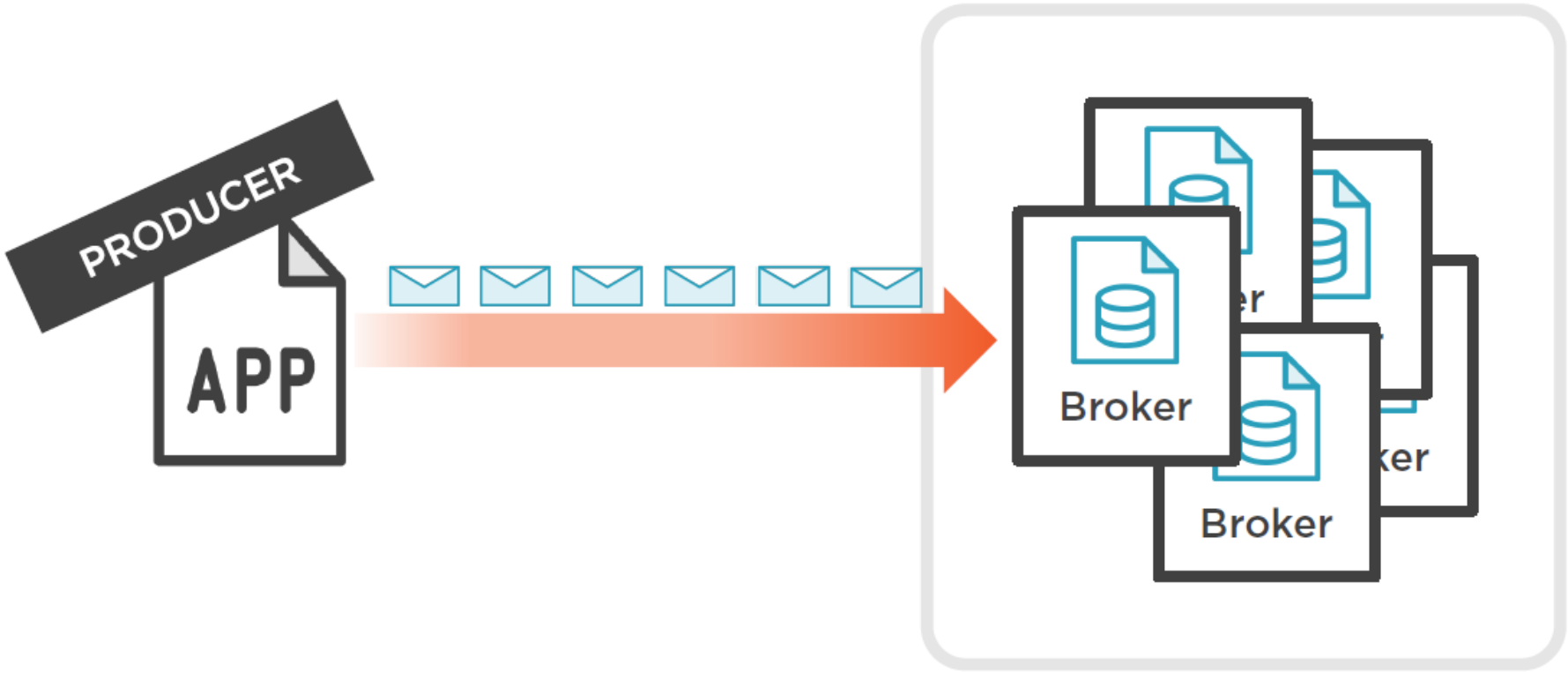
Course Map – Producing Message with Kafka Producers



1. Creating Apache Kafka Producer
2. Sending Messages
3. Delivery Guarantees
4. Order Guarantees

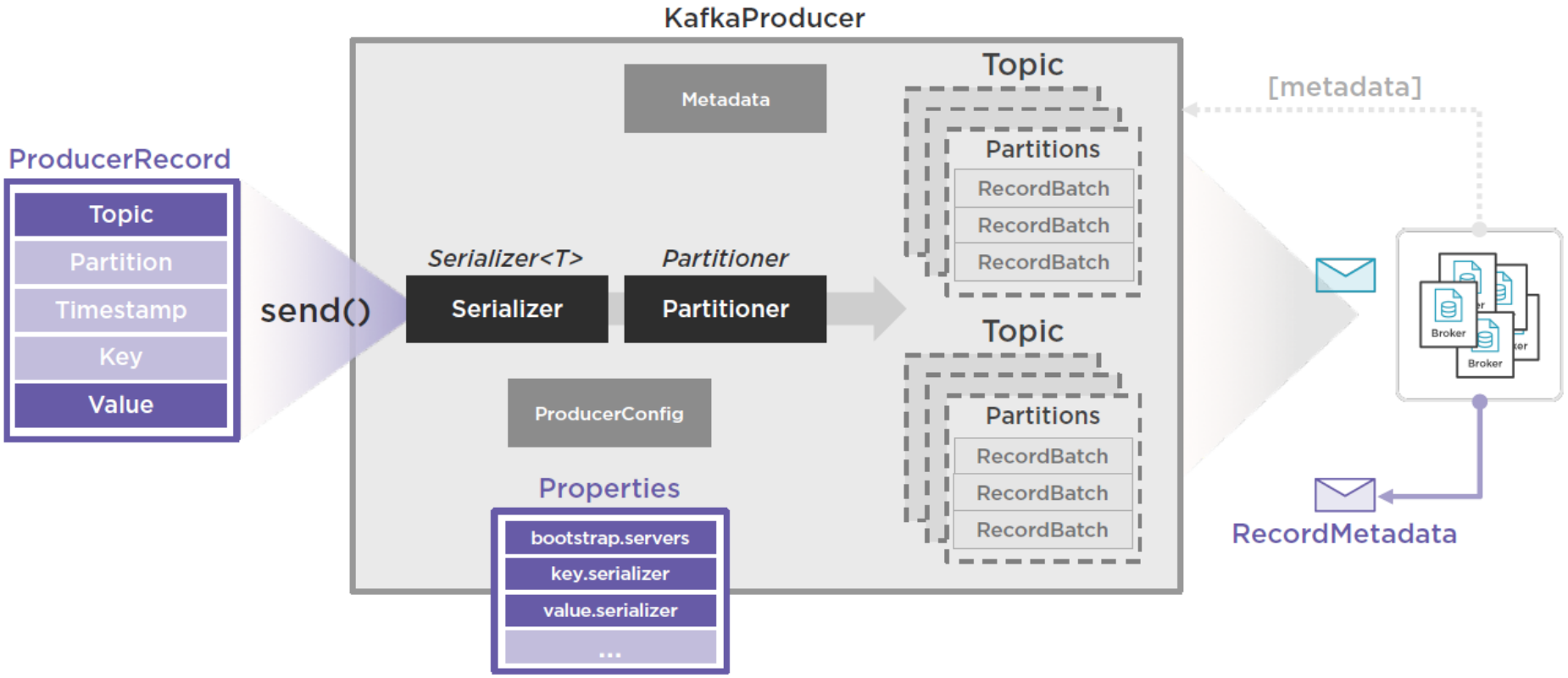
Creating Apache Kafka Producer

Kafka Producer - External



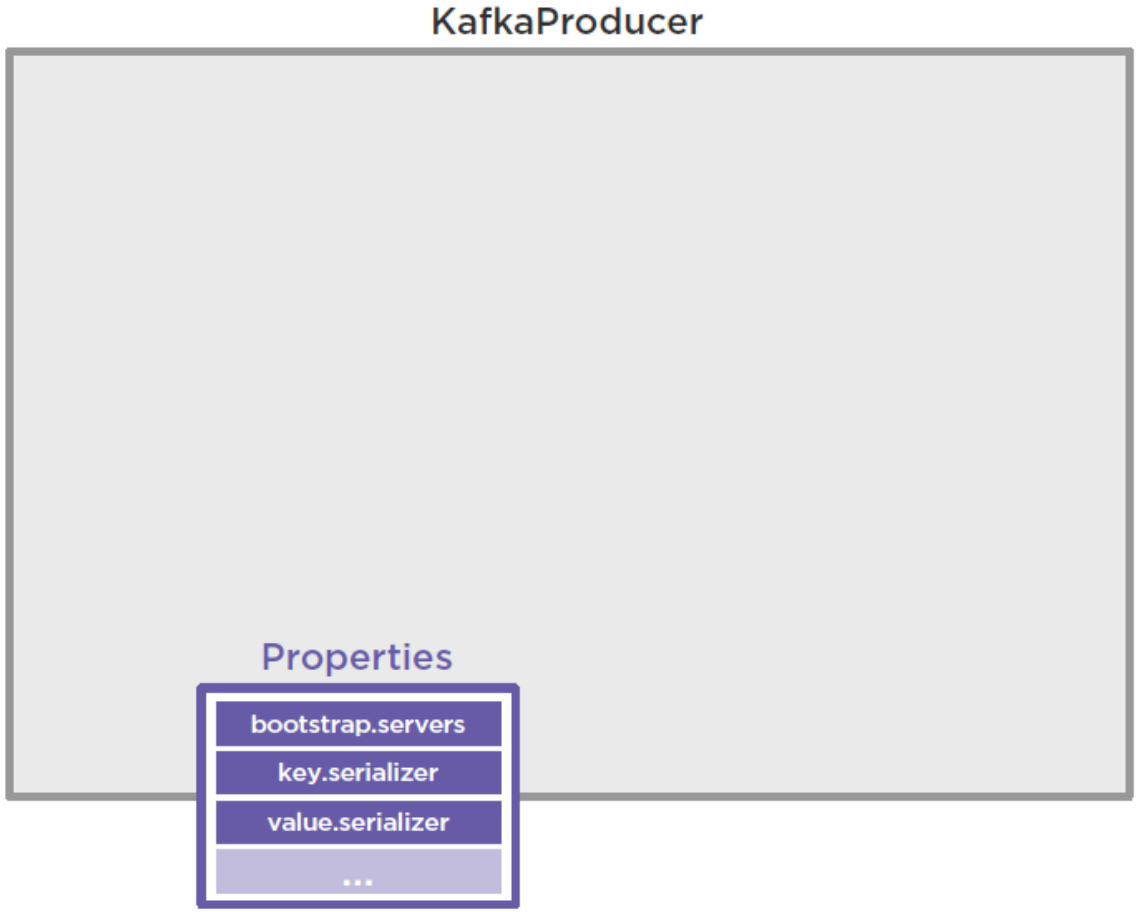
Creating Apache Kafka Producer

Kafka Producer - Internal



Creating Apache Kafka Producer

Creating a Kafka Producer



Creating Apache Kafka Producer

- Kafka Producer: Required Properties
- bootstrap.servers
- - Cluster membership: partition leaders, etc.
- key and value serializers
- - Classes used for message serialization and deserialization

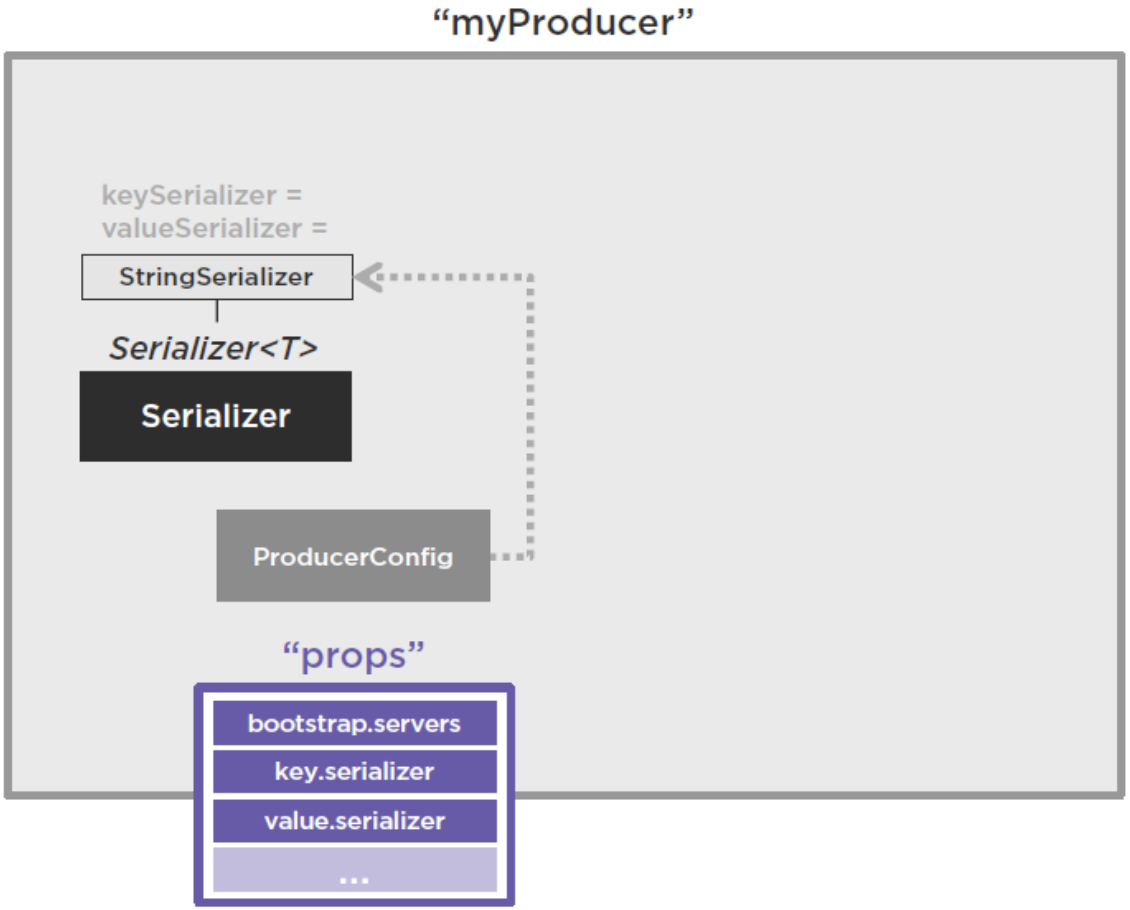
```
Properties props = new Properties();
props.put(“bootstrap.servers”, “BROKER-1:9092, BROKER-2:9093”);
props.put(“key.serializer”, “org.apache.kafka.common.serialization.StringSerializer”);
props.put(“value.serializer”, “org.apache.kafka.common.serialization.StringSerializer”);
```

Creating Apache Kafka Producer

```
public class KafkaProducerApp {
    public static void main(String[] args){
        Properties props = new Properties();
        props.put("bootstrap.servers", "BROKER-1:9092, BROKER-2:9093");
        props.put("key.serializer",
        "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
        "org.apache.kafka.common.serialization.StringSerializer");
        KafkaProducer myProducer = new KafkaProducer(props);
    }
}
```

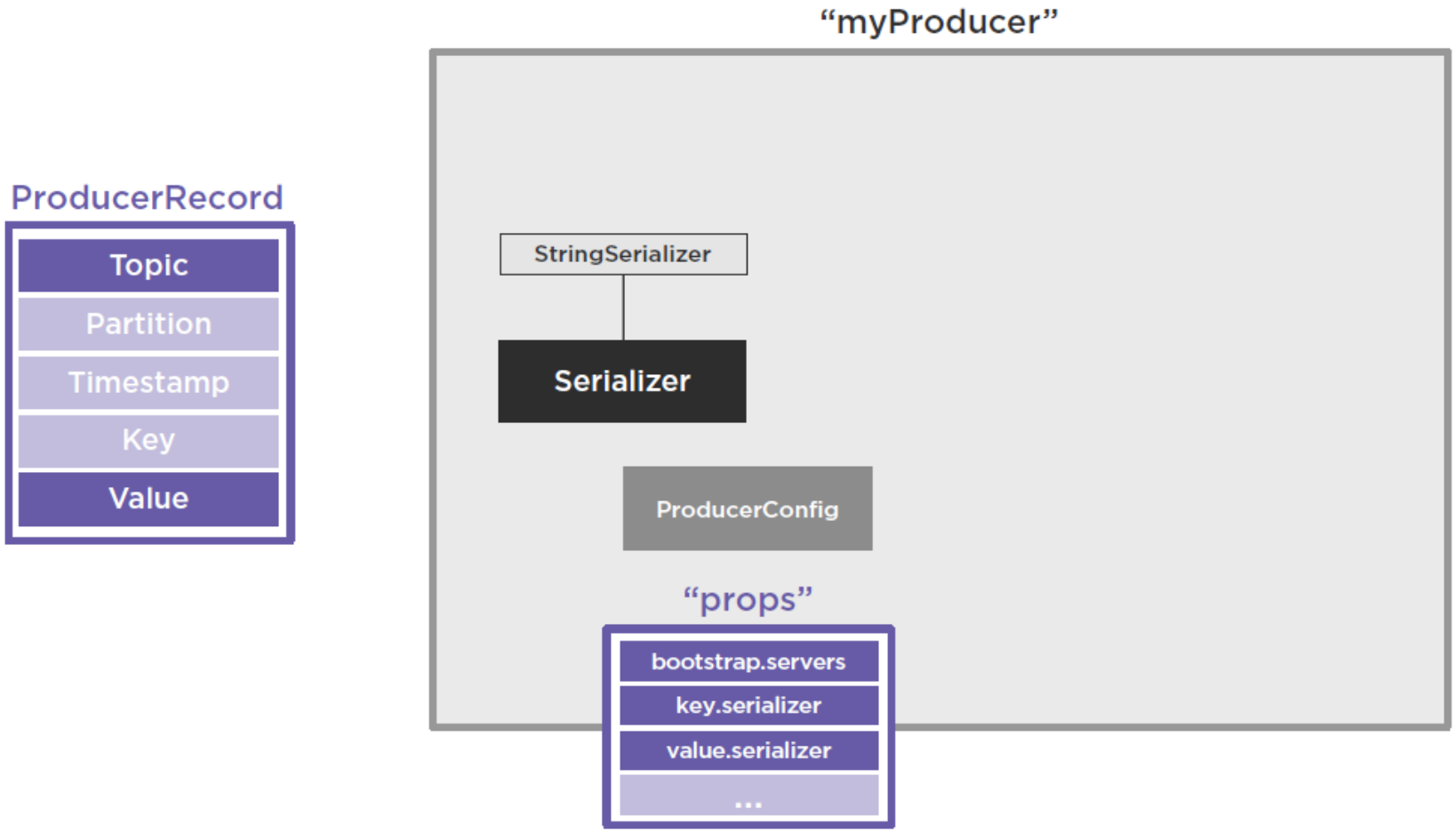
Creating Apache Kafka Producer

Basic Kafka Producer



Creating Apache Kafka Producer

Kafka Producer ~~Messages~~ Records



Creating Apache Kafka Producer

- **ProducerRecord: Required Properties**
- **topic**
- **-** Topic to which the ProducerRecord will be sent
- **value**
- **-** The message content (matching the serializer type for value)

```
props.put(“value.serializer”, “org.apache.kafka.common.serialization.StringSerializer”)
ProducerRecord myMessage = new ProducerRecord(“my_topic”,“My Message 1”);
```

Creating Apache Kafka Producer

Kafka Producer Shell Program

```
~$ bin/kafka-console-producer.sh \
> --broker-list localhost:9092, localhost:9093 \
➢  --topic my_topic
```

MyMessage 1
MyMessage 2

Creating Apache Kafka Producer

- **ProducerRecord: Required Properties**
- **topic**
- **-** Topic to which the ProducerRecord will be sent
- **value**
- **-** The message content (matching the serializer type for value)

```
props.put(“value.serializer”, “org.apache.kafka.common.serialization.StringSerializer”)
ProducerRecord myMessage = new ProducerRecord(“my_topic”,“My Message 1”);

ProducerRecord myMessage = new ProducerRecord(“my_topic”,3.14159);
SerializationException: Can’t convert value of class ...
```

Notes: KafkaProducer instances can only send ProducerRecords that match the key and value serializers types it is configured with.

Creating Apache Kafka Producer

- **ProducerRecord: Optional Properties**
- **partition**
- - specific partition within the topic to send ProducerRecord
- **timestamp**
- - the Unix timestamp applied to the record

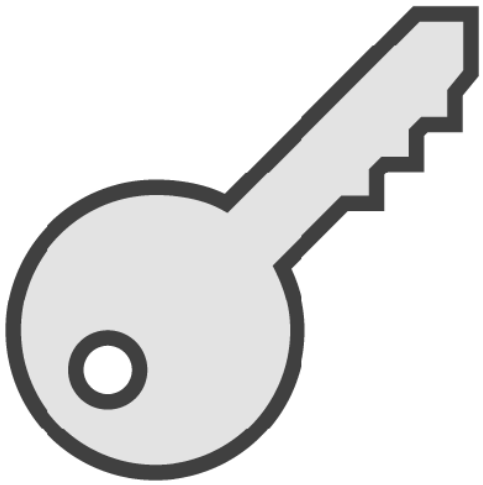
```
ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value);
// Example:
ProducerRecord("my_topic", 1, 124535353325, "Course-001","My Message 1");

// Defined in server.properties:
log.message.timestamp.type = [CreateTime, LogAppendTime]

// CreateTime: producer-set timestamp used.
// LogAppendTime: broker-set timestamp used when message is appended to commit log.
```

Creating Apache Kafka Producer

- **ProducerRecord: Optional Properties**
- **key**
- - a value to be used as the basis of determining the partitioning strategy to be employed by the Kafka Producer

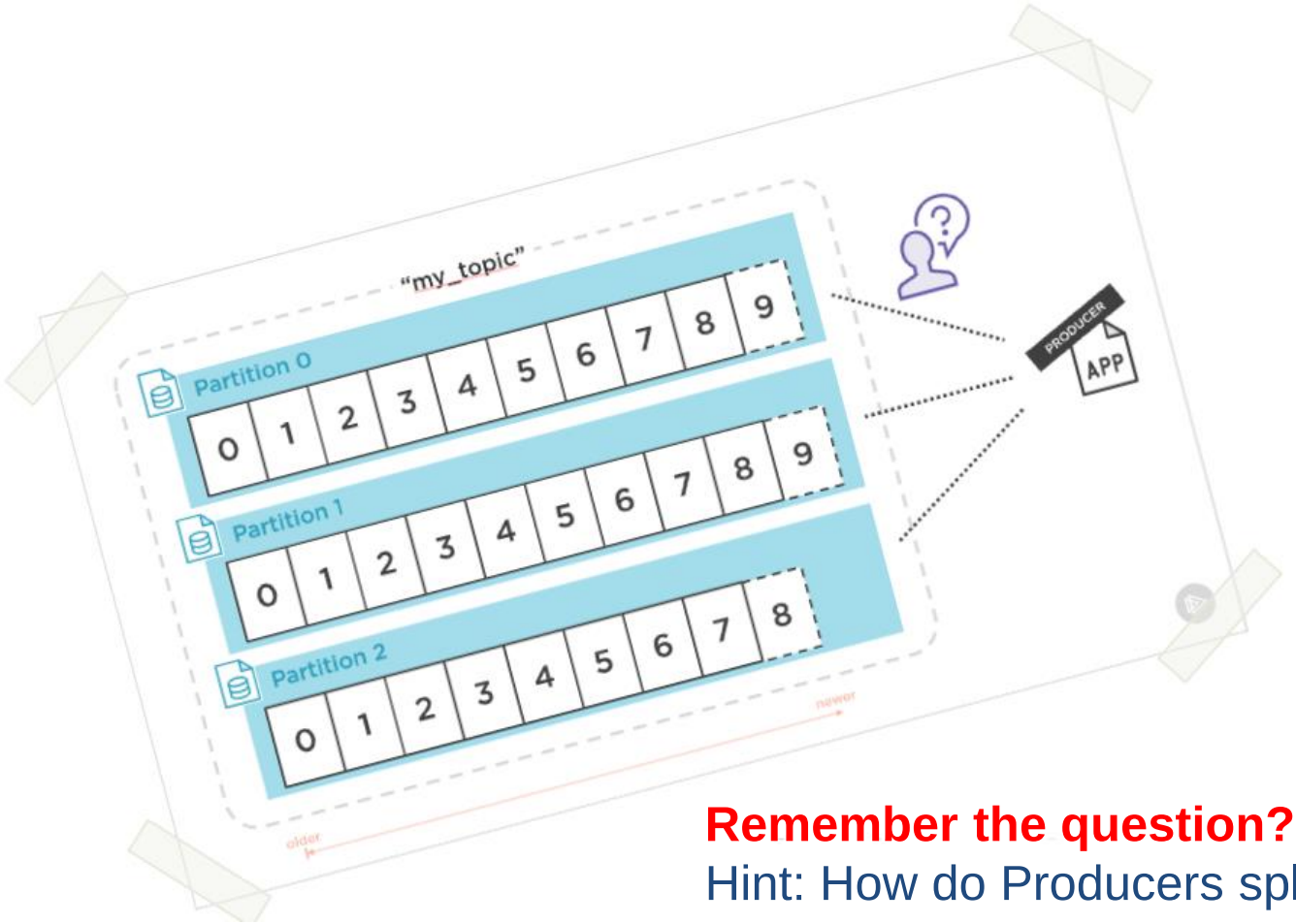
```
ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value);


// Example:
ProducerRecord(“my_topic”, 1, 124535353325, “Course-001”,“My Message 1”);
```

Creating Apache Kafka Producer



- **Two useful purposes:**
- - Additional information in the message
- - Can determine what partitions the message will be written to
- **Downside:**
- - Additional overhead
- - Depends on the serializer type used

Creating Apache Kafka Producer



Remember the question?

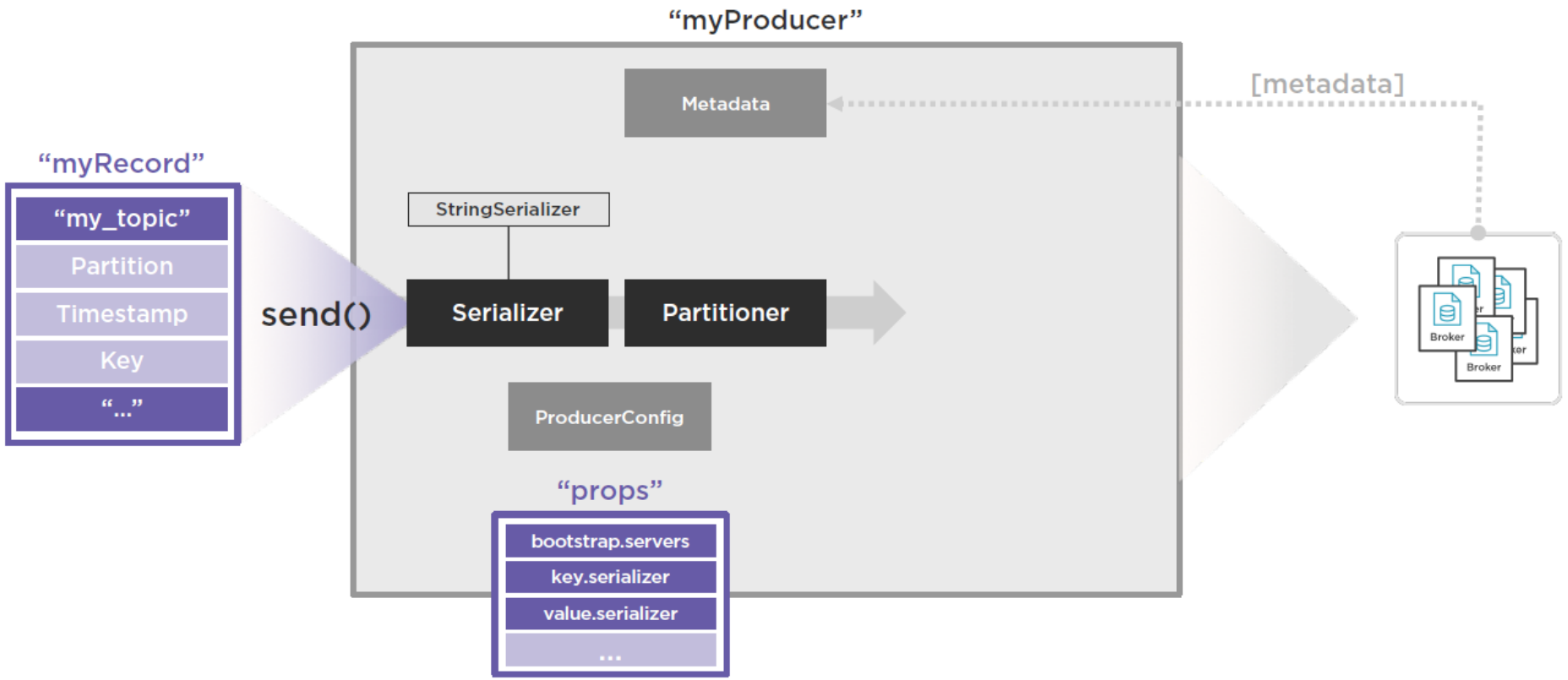
Hint: How do Producers split messages to partitions?

Creating Apache Kafka Producer

```
public class KafkaProducerApp {

public static void main(String[] args){

        Properties props = new Properties();

        props.put(“bootstrap.servers”, “BROKER-1:9092, BROKER-2:9093”);

        props.put(“key.serializer”, “org.apache.kafka.common.serialization.StringSerializer”);

        props.put(“value.serializer”, “org.apache.kafka.common.serialization.StringSerializer”);

        KafkaProducer myProducer = new KafkaProducer(props);

        ProducerRecord myRecord = new ProducerRecord(“my_topic”, “Course-001”, “My Message 1”);

        myProducer.send(myRecord); // Best practice: try..catch

}}
```

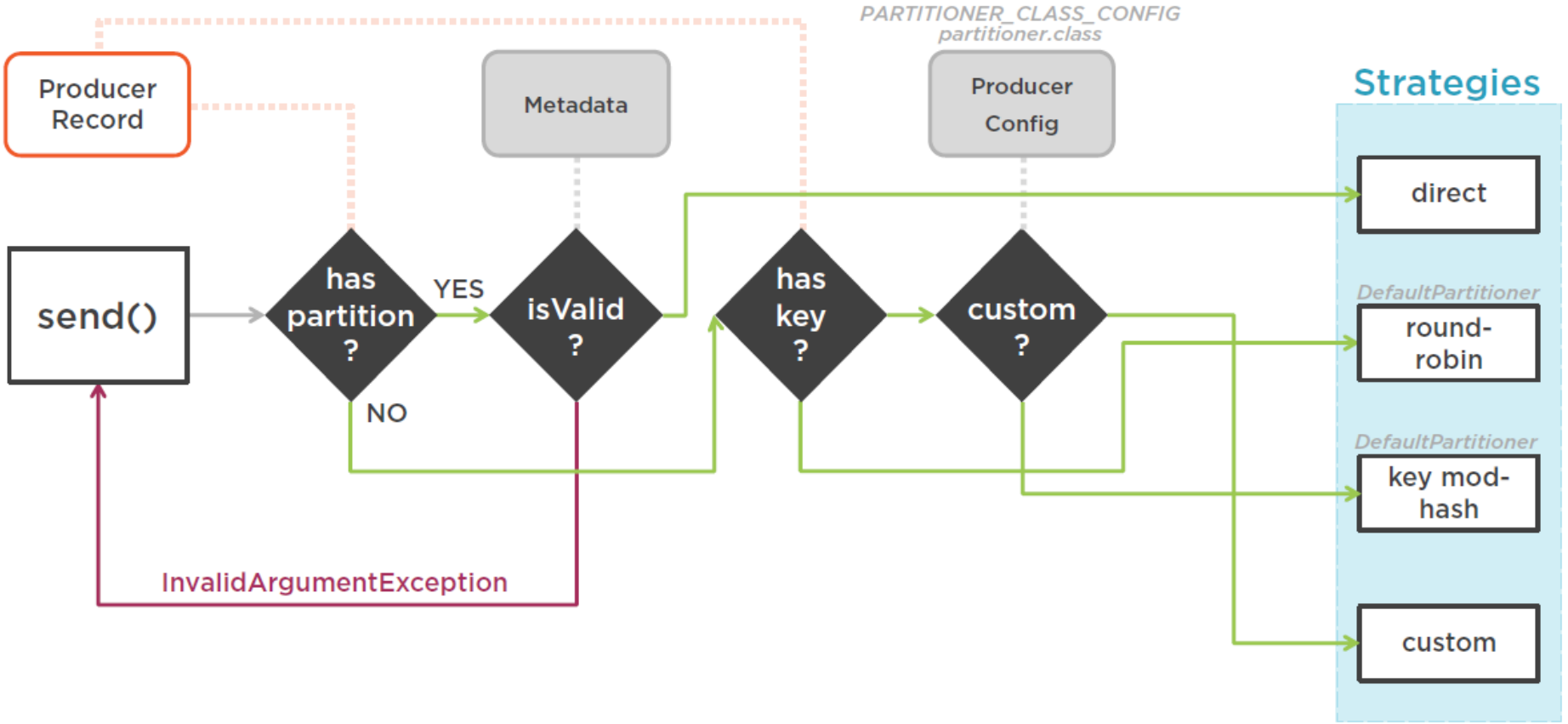
Sending Messages

Sending the messages, Part - 1



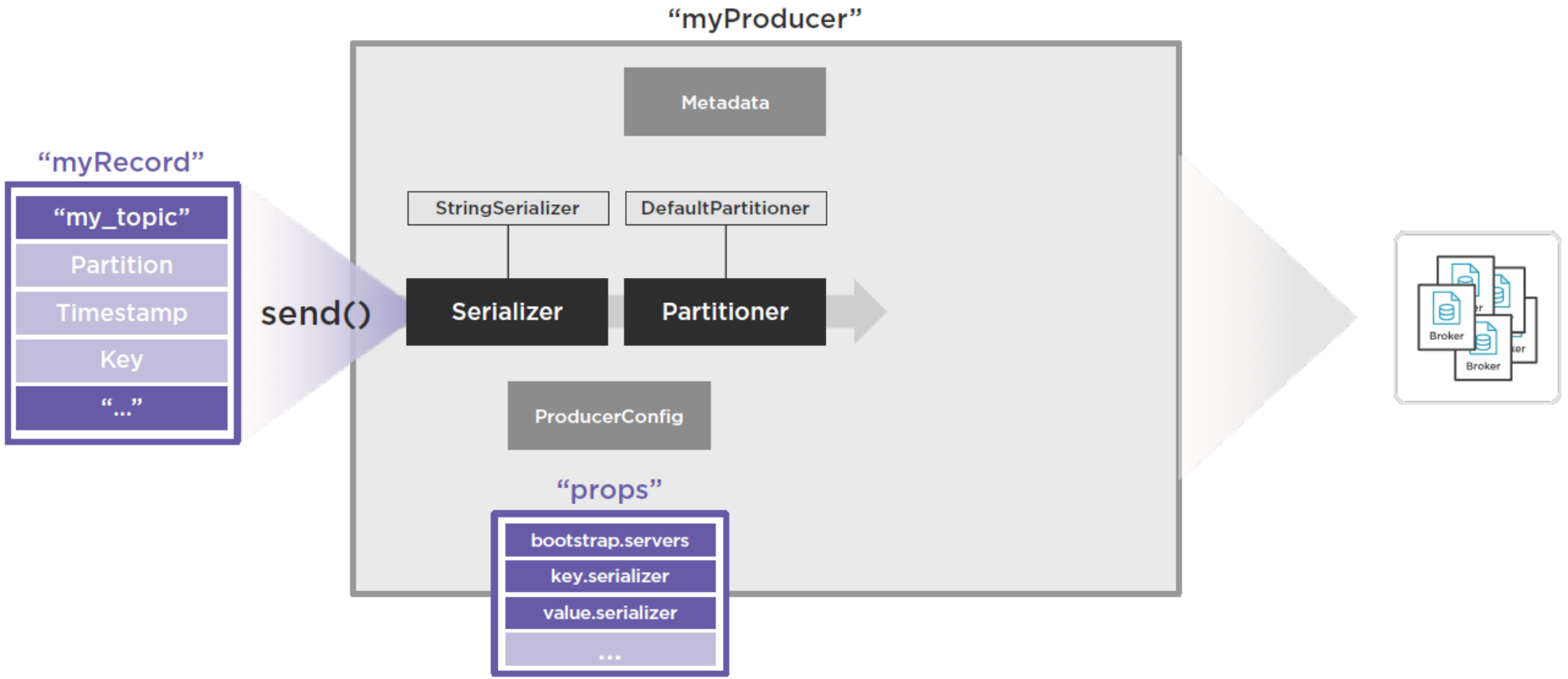
Sending Messages

Sending the messages, Part - 1



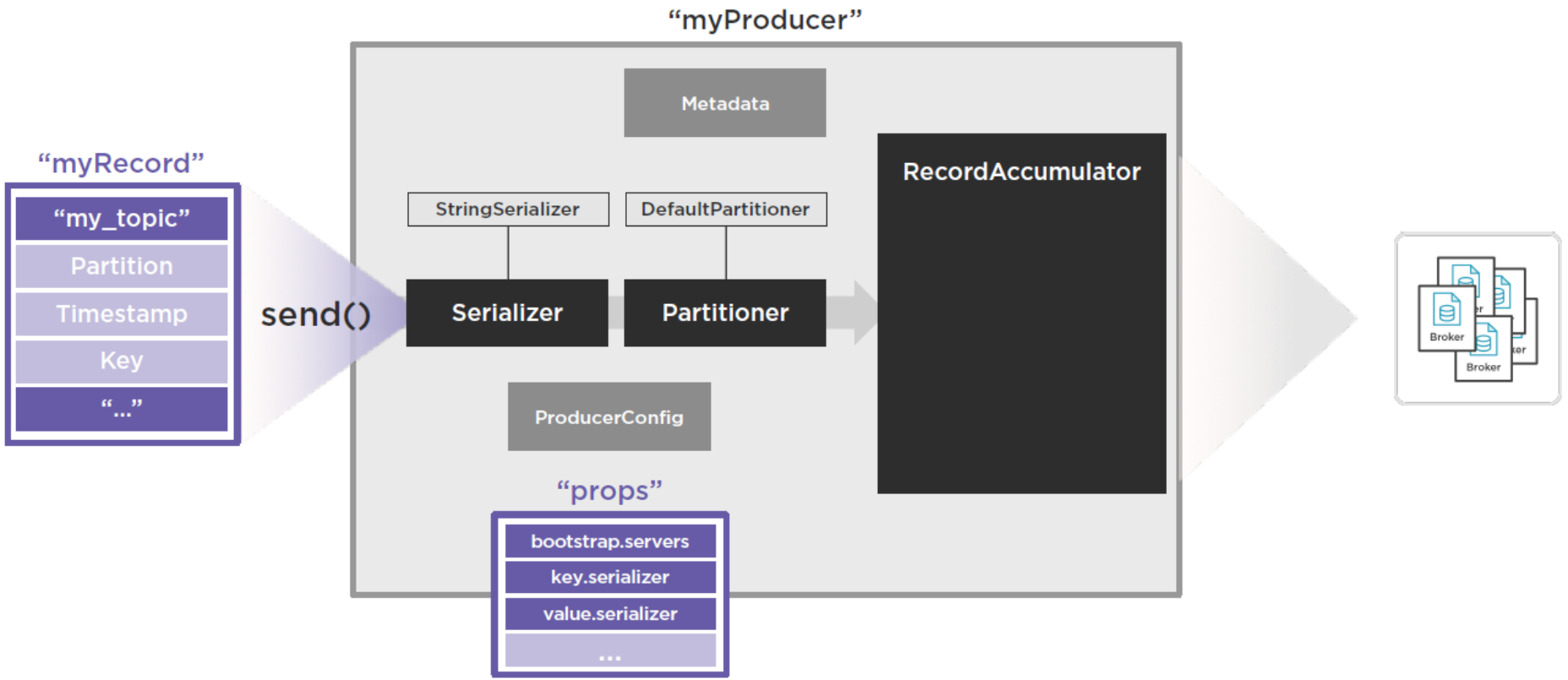
Sending Messages

Sending the messages, Part - 1



Sending Messages

Sending the messages, Part - 2



Sending Messages

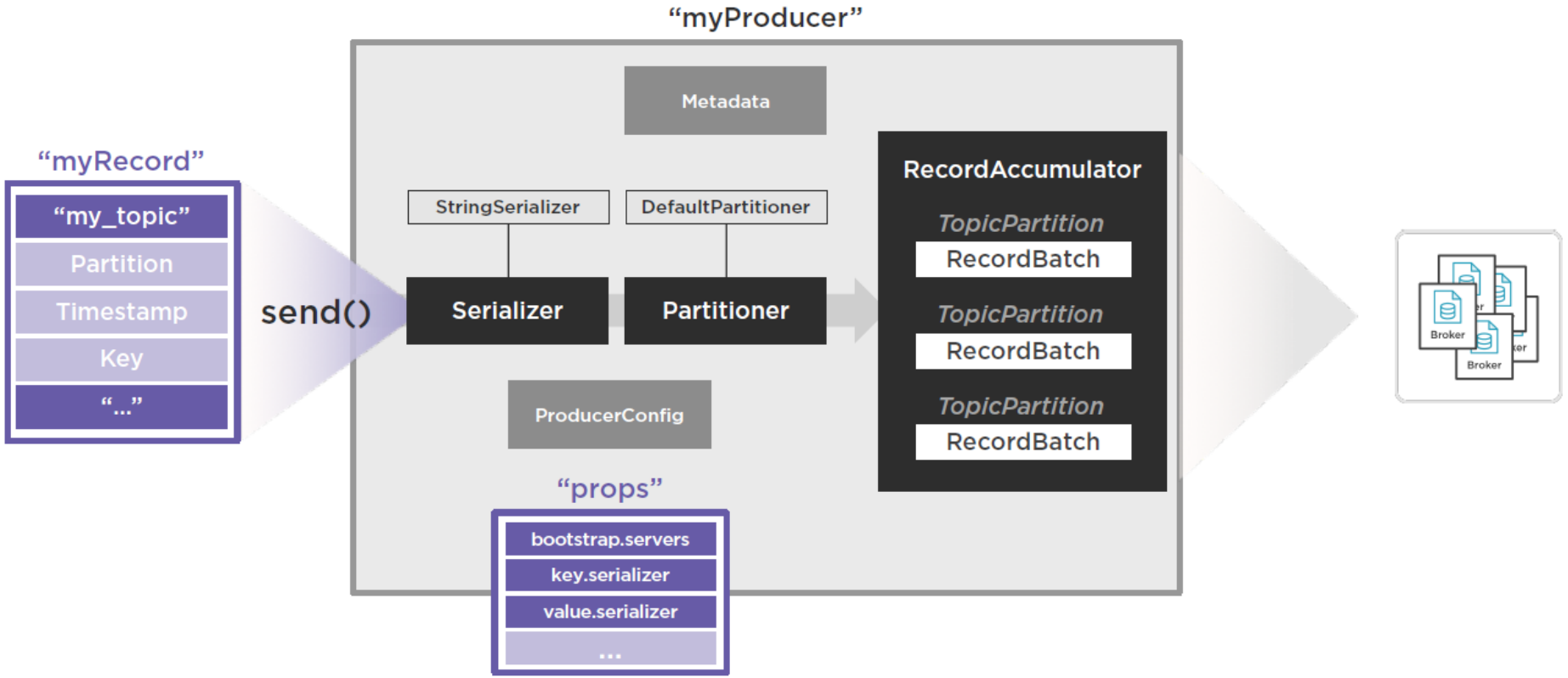
Sending the messages, Part - 2

Micro-batching in Apache Kafka

- At scale, efficiency is everything.
- Small, fast batches of messages:
- - Sending (Producer)
- - Writing (Broker)
- - Reading (Consumer)
- Modern operating system functions:
- - Pagecache
- - Linux sendfile() system call (kernel)
- Amortization of the constant cost

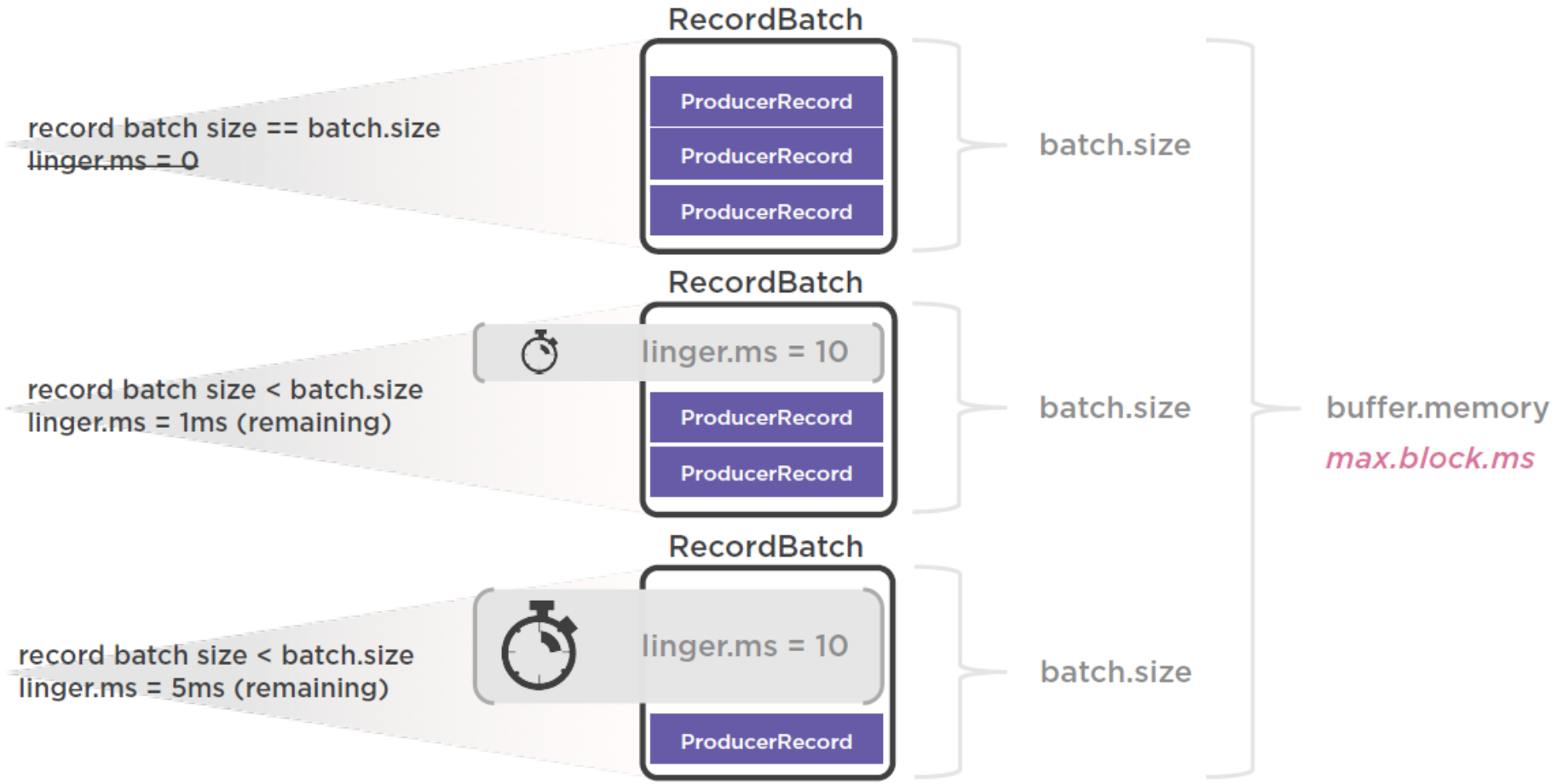
Sending Messages

Sending the messages, Part - 2



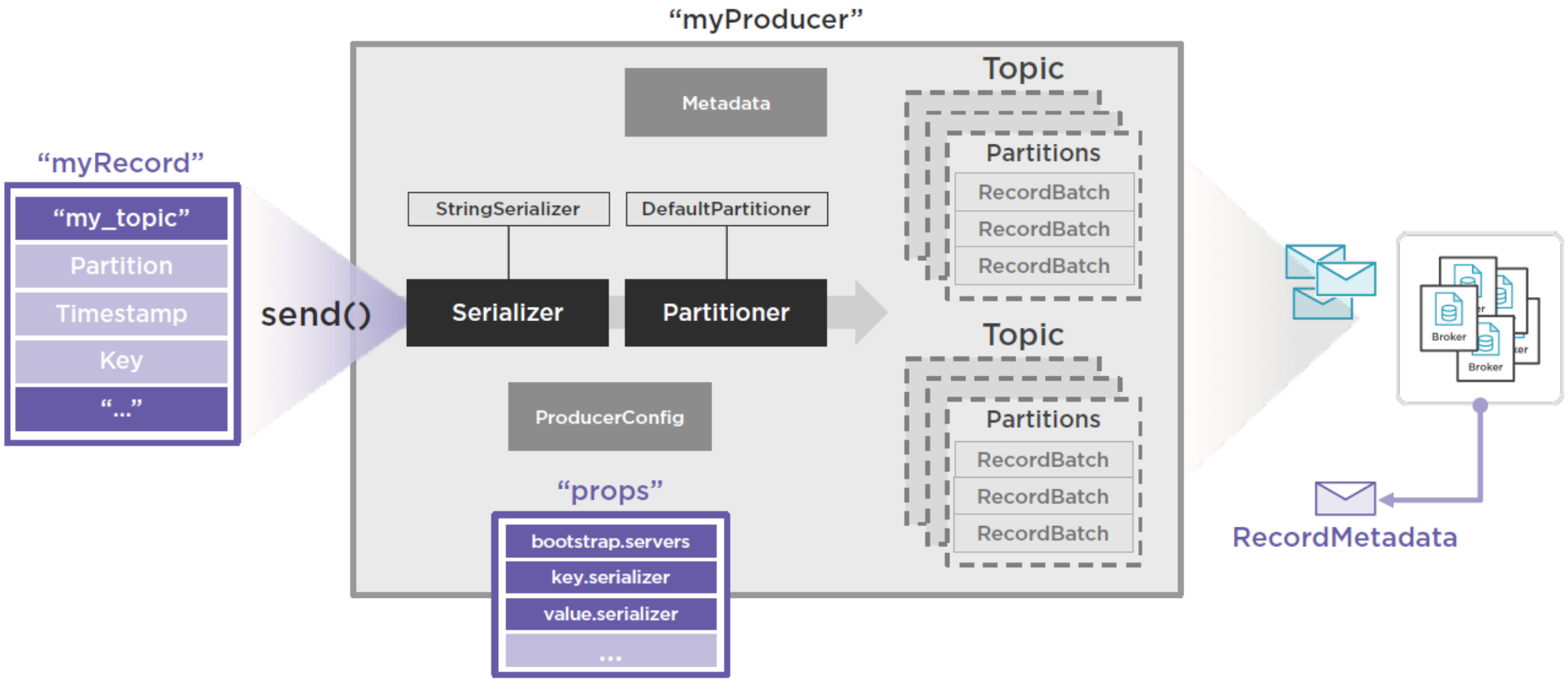
Sending Messages

Message buffering

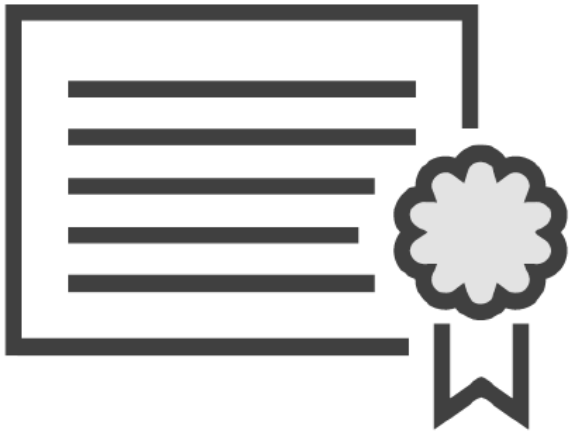


Sending Messages

Sending the messages, Part - 2

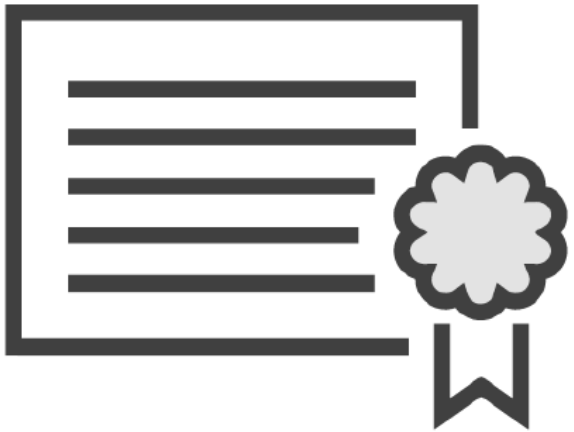


Delivery Guarantees



- Broker acknowledgement (“acks”)
- - 0: fire and forget
- - 1: leader acknowledged
- - 2: replication quorum acknowledged
- Broker responds with error
- - “retries”
- - “retry.backoff.ms”

Ordering Guarantees



- Message order by partition
- - No global order across partitions

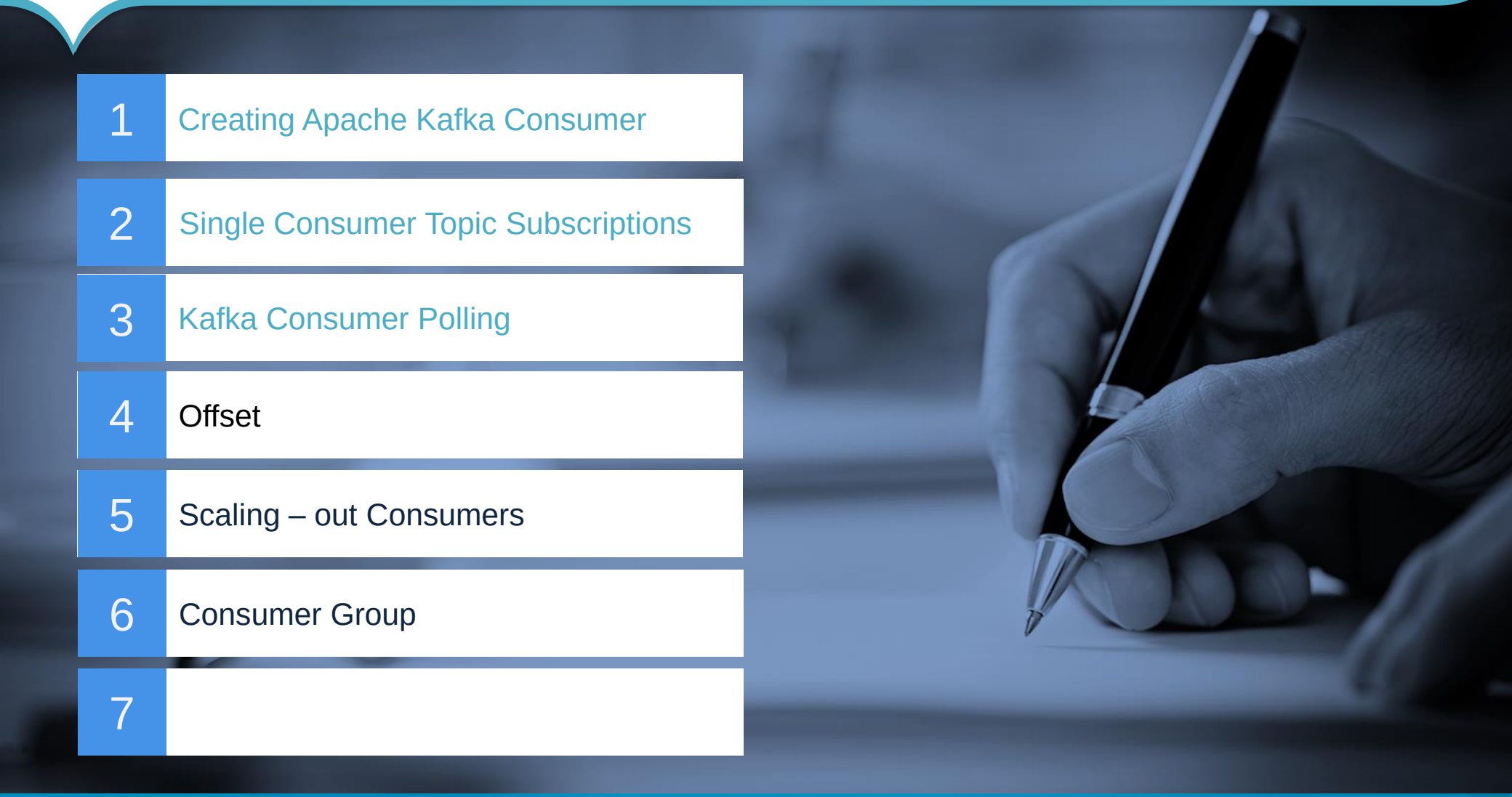
- Can get complicated with errors
- - retries, retry.backoff.ms
- - max.in.flight.request.per.connection

- Delivery semantics
- - At-most-once, at-least-once, only-once

Summary

- Kafka Producer Internals
  - Properties -> ProducerConfig
  - Message -> ProducerRecord
  - Processing Pipeline: Serializers and Partitioners
  - Micro-batching -> Record Accumulator and RecordBuffer
- Delivery and Ordering Guarantees
- Java-based Producer

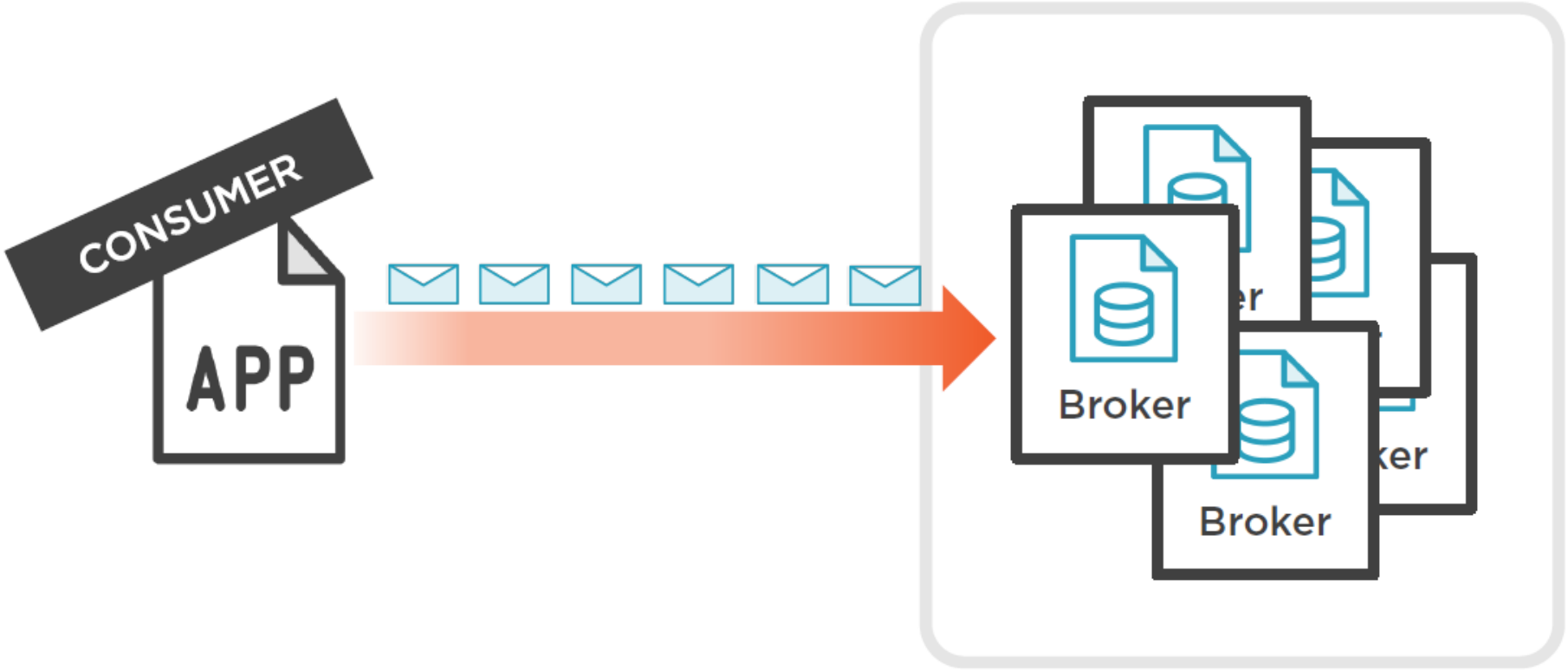
Course Map – Consuming Message with Kafka Consumer and Consumer Groups



1	Creating Apache Kafka Consumer
2	Single Consumer Topic Subscriptions
3	Kafka Consumer Polling
4	Offset
5	Scaling – out Consumers
6	Consumer Group
7	

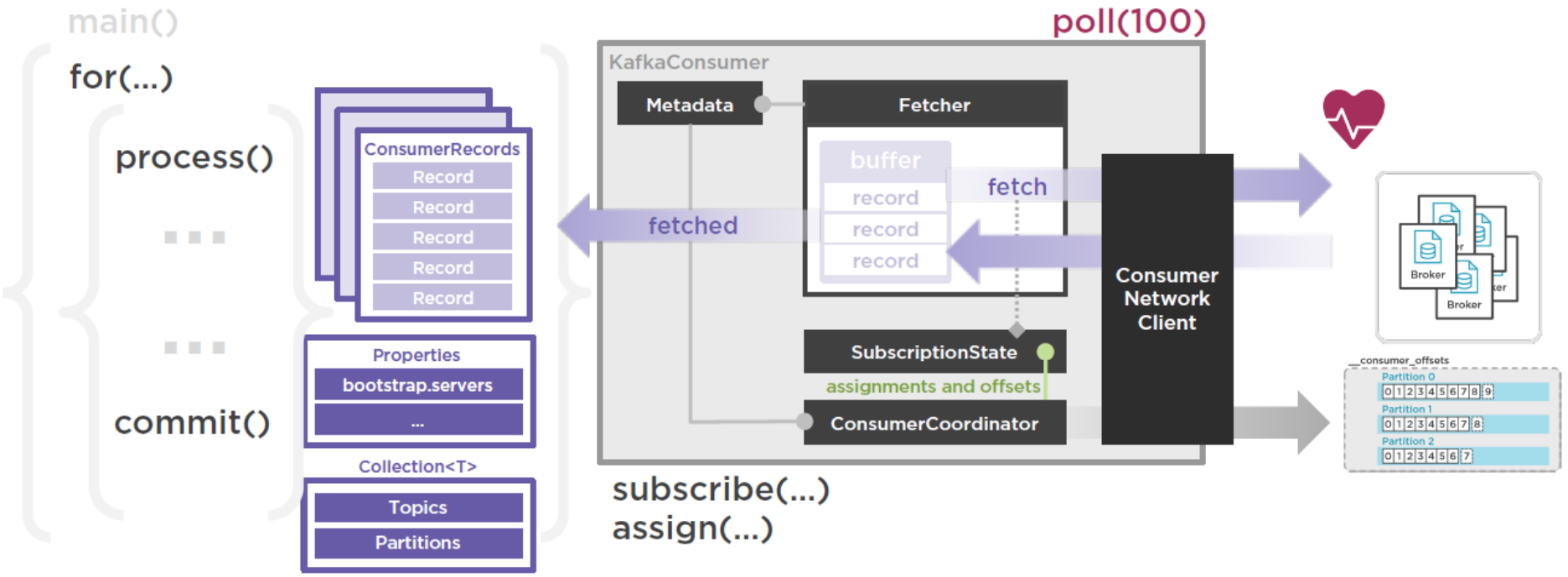
Creating Apache Kafka Consumer

Kafka Consumer - External



Creating Apache Kafka Consumer

Kafka Consumer - Internals



Creating Apache Kafka Consumer

- **Kafka Consumer: Required Properties**
- **bootstrap.servers**
  - Cluster membership: partition leaders, etc.
- **key and value deserializers**
  - Classes used for message deserialization

```
Properties props = new Properties();
props.put(“bootstrap.servers”, “BROKER-1:9092, BROKER-2:9093”);
props.put(“key.deserializer”, “org.apache.kafka.common.serialization.StringDeserializer”);
props.put(“value.deserializer”,
“org.apache.kafka.common.serialization.StringDeserializer”);
```

Creating Apache Kafka Consumer

```
public class KafkaConsumerApp {

        public static void main(String[] args){

                Properties props = new Properties();

                props.put(“bootstrap.servers”, “BROKER-1:9092, BROKER-2:9093”);

                props.put(“key.deserializer”, “org.apache.kafka.common.serialization.StringDeserializer”);

                props.put(“value.deserializer”, “org.apache.kafka.common.serialization.StringDeserializer”);

                KafkaConsumer myConsumer = new KafkaConsumer(props);

        }

}
```

Creating Apache Kafka Consumer

Subscribing to Topics

```
public class KafkaConsumerApp {

    public static void main(String[] args){

        // Properties code ommitted...

        KafkaConsumer myConsumer = new KafkaConsumer(props);

        myConsumer.subscribe(Arrays.asList("my-topic"));

        // Alternatively, use regular expressions:

        myConsumer.subscribe("my-*");

    }}
```

Creating Apache Kafka Consumer

Subscribing to Topics

```
// Initial subscription:

myConsumer.subscribe(Arrays.asList(“my-topic”));

// Later, add another topic to the subscription (intentional):

myConsumer.subscribe(Arrays.asList(“my-other-topic”));

// Better for incremental topic subscription management:

ArrayList<String> topics = new ArrayList<String>();

topics.add(“myTopic”);

topics.add(“myOtherTopic”);

myConsumer.subscribe(topics);
```

Creating Apache Kafka Consumer

Un - Subscribing to Topics

```
ArrayList<String> topics = new ArrayList<String>();

topics.add(“myTopic”);

topics.add(“myOtherTopic”);

myConsumer.subscribe(topics);

myConsumer.unsubscribe();

// Less-than-intuitive unsubscribe alternative:

topics.clear(); // Emptying out the list

myConsumer.subscribe(topics); // passing the subscribe() method a list of empty strings
```

Creating Apache Kafka Consumer

- **subscribe()**
  - For topics (dynamic/automatic)
  - One topic, one-to-many partitions
  - Many topics, many more partitions
- **assign()**
  - For partitions
  - One or more partitions, regardless of
- topic
  - Manual, self-administering mode

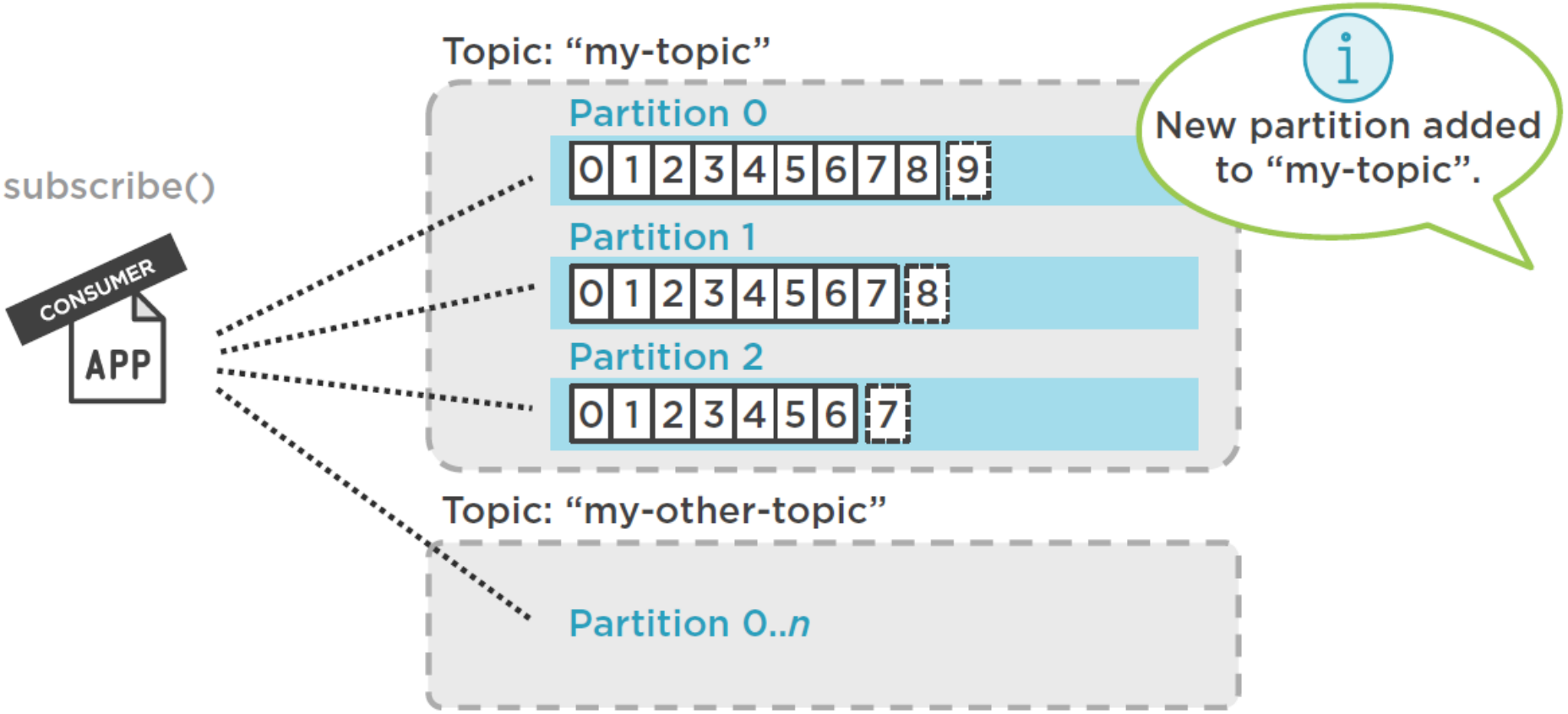
Creating Apache Kafka Consumer

Manual Partition Assignments

```
// Similar pattern as subscribe():

TopicPartition partition0 = new TopicPartition("myTopic", 0);

ArrayList<TopicPartition> partitions = new ArrayList<TopicPartition>();

partitions.add(partition0);

myConsumer.assign(partitions); // Remember this is NOT incremental!
```

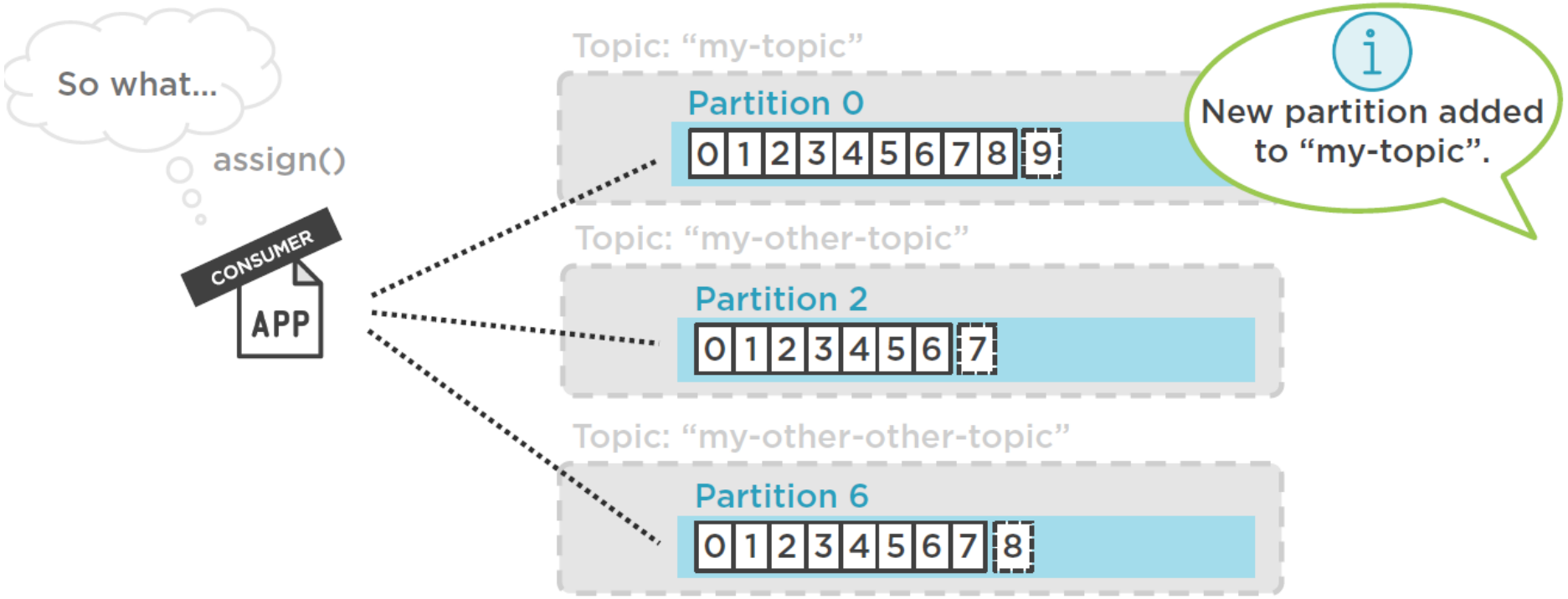
Single Consumer Topic Subscriptions

Single Consumer Topic Subscriptions



Single Consumer Topic Subscriptions

Single Consumer Topic Subscriptions



Single Consumer Topic Subscriptions

The poll - loop

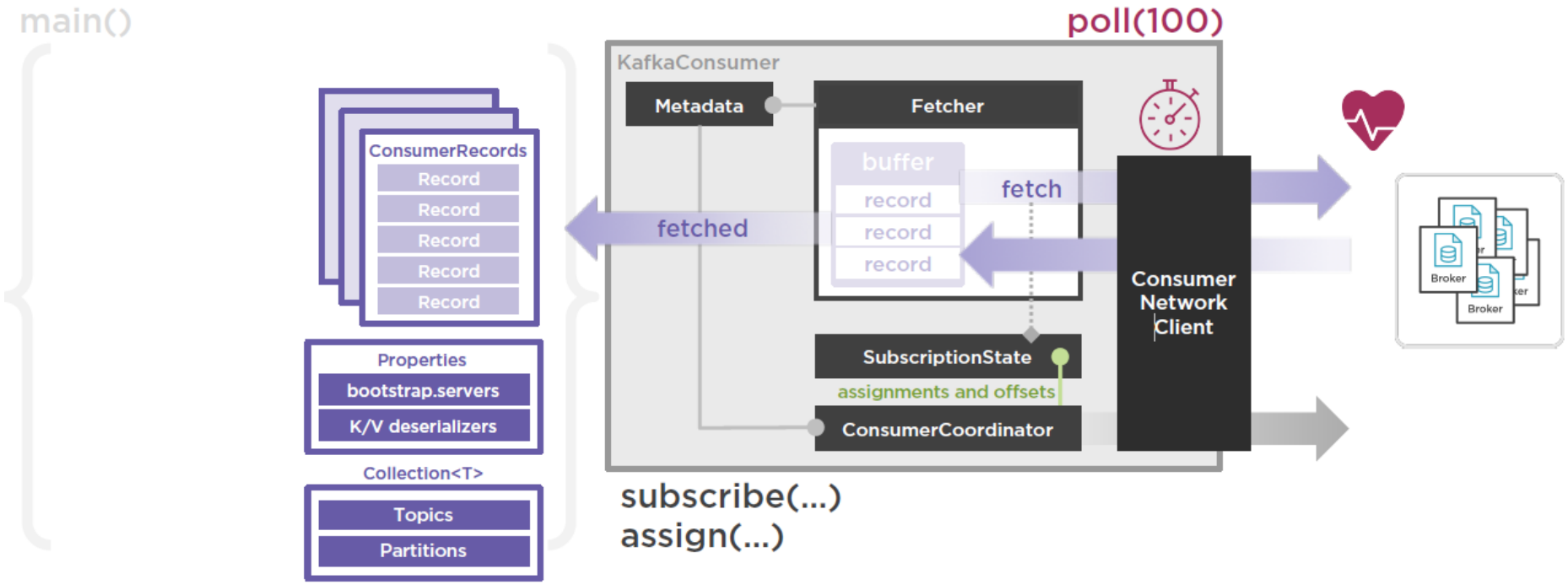
- Primary function of the Kafka Consumer
  - poll()
- Continuously polling the brokers for data
- Single API for handling all Consumer-Broker interactions
  - A lot of interactions beyond message retrieval

Single Consumer Topic Subscriptions

Starting the Poll Loop

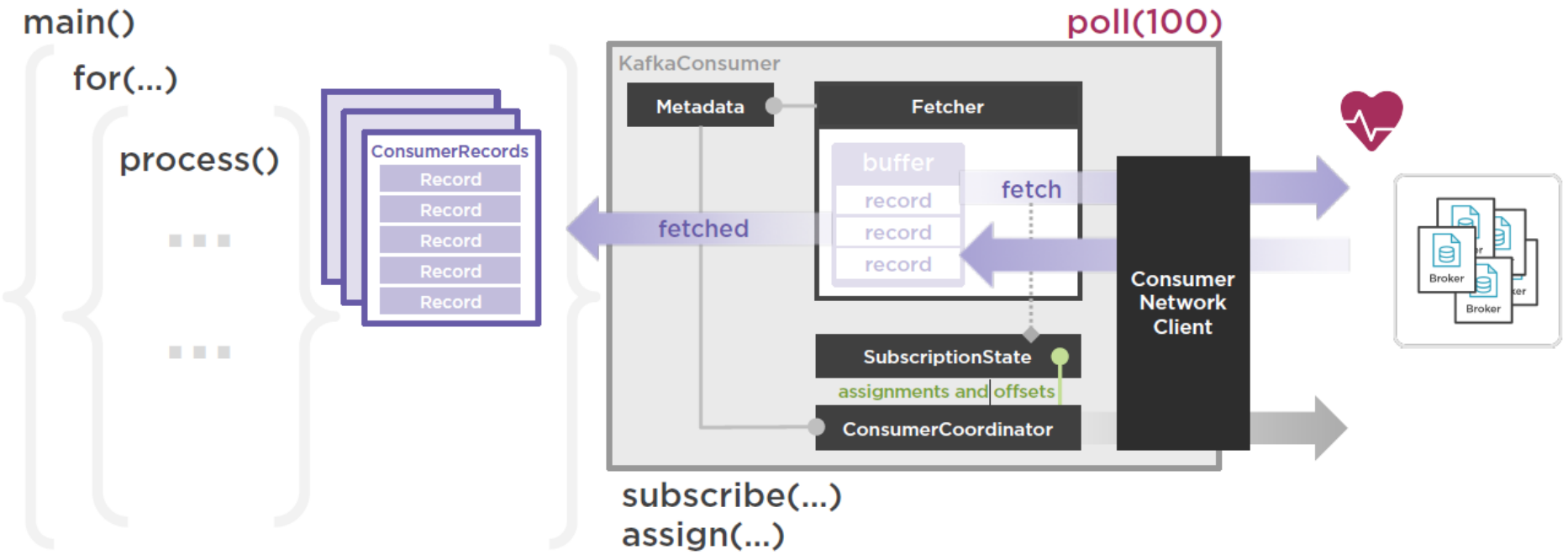
```
// Set the topic subscription or partition assignments:
myConsumer.subscribe(topics);
myConsumer.assign(partitions);
try {
        while (true) {
                ConsumerRecords<String, String> records = myConsumer.poll(100);
                // Your processing logic goes here...
                }
            finally {
            myConsumer.close();
        }
}
```

Kafka Consumer Polling

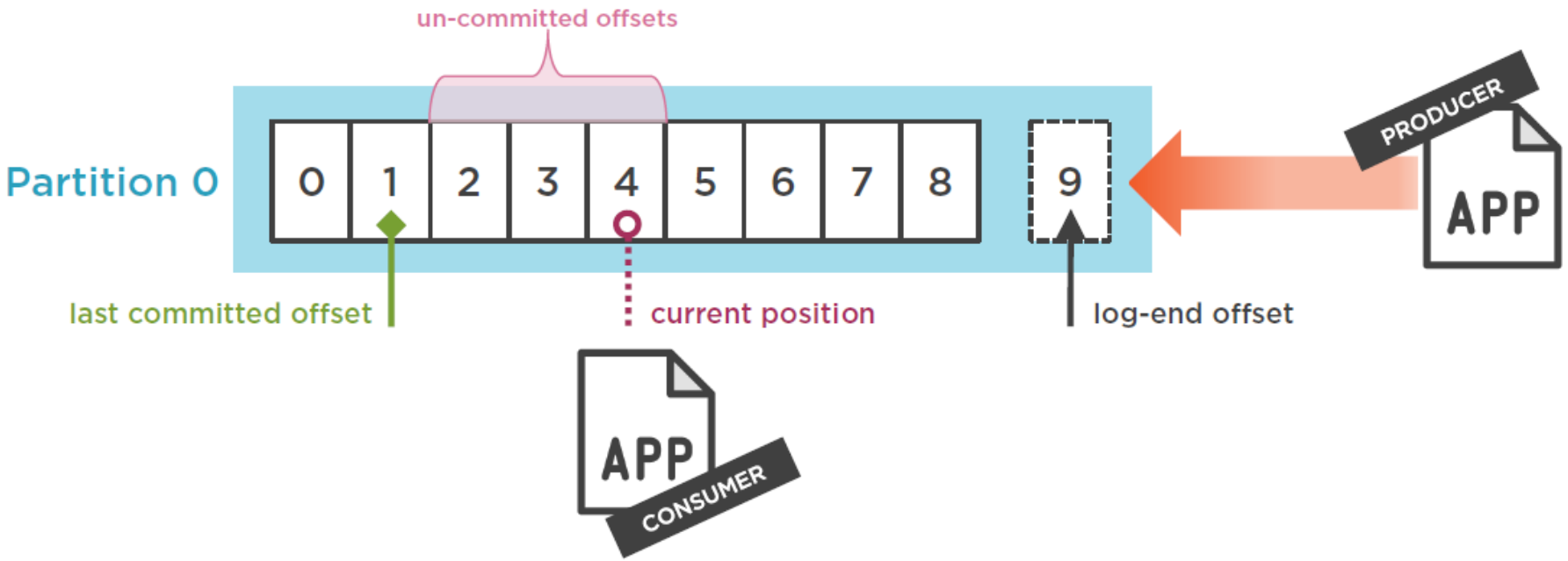


The poll() process is a single-threaded operation.

Processing Messages

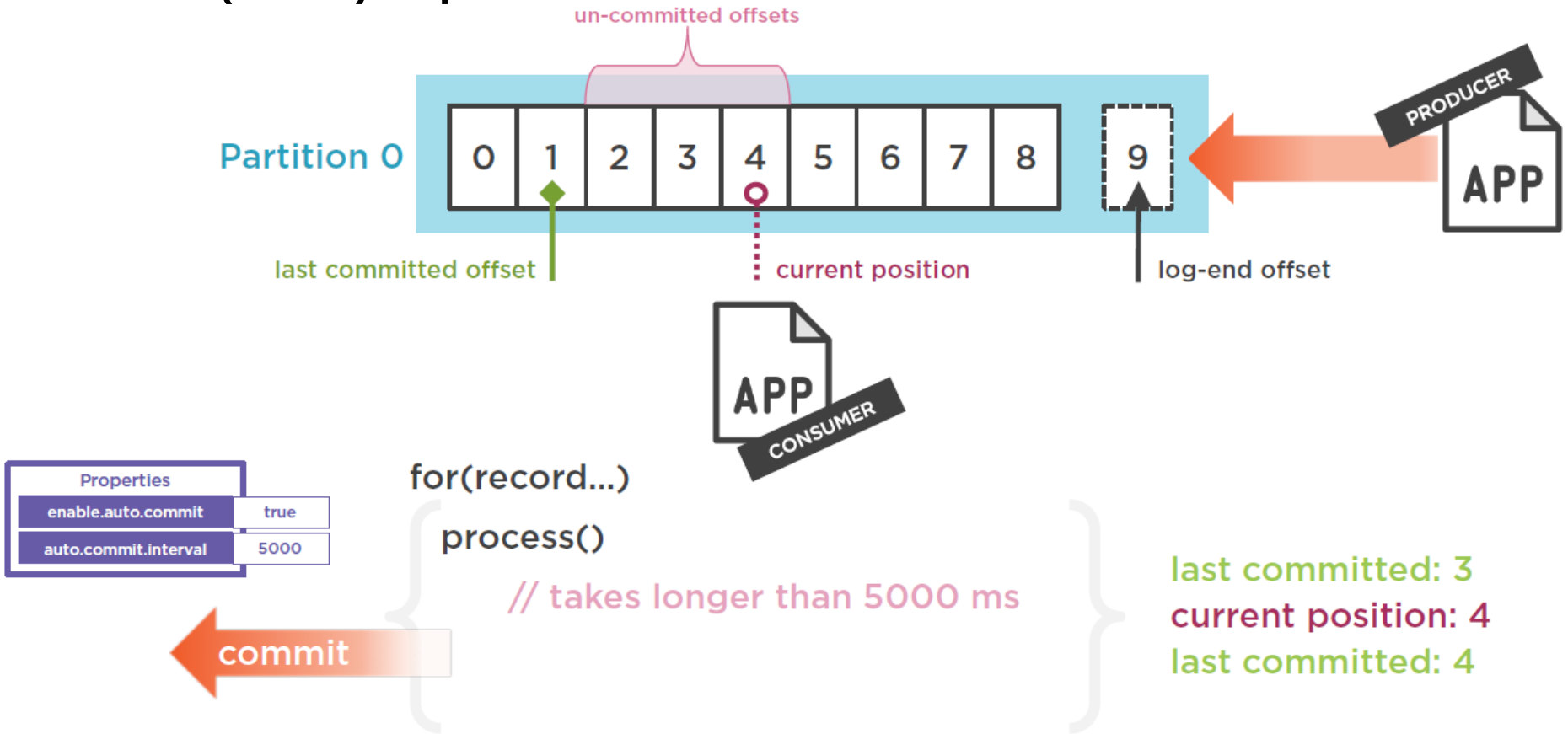


Offset



Offset

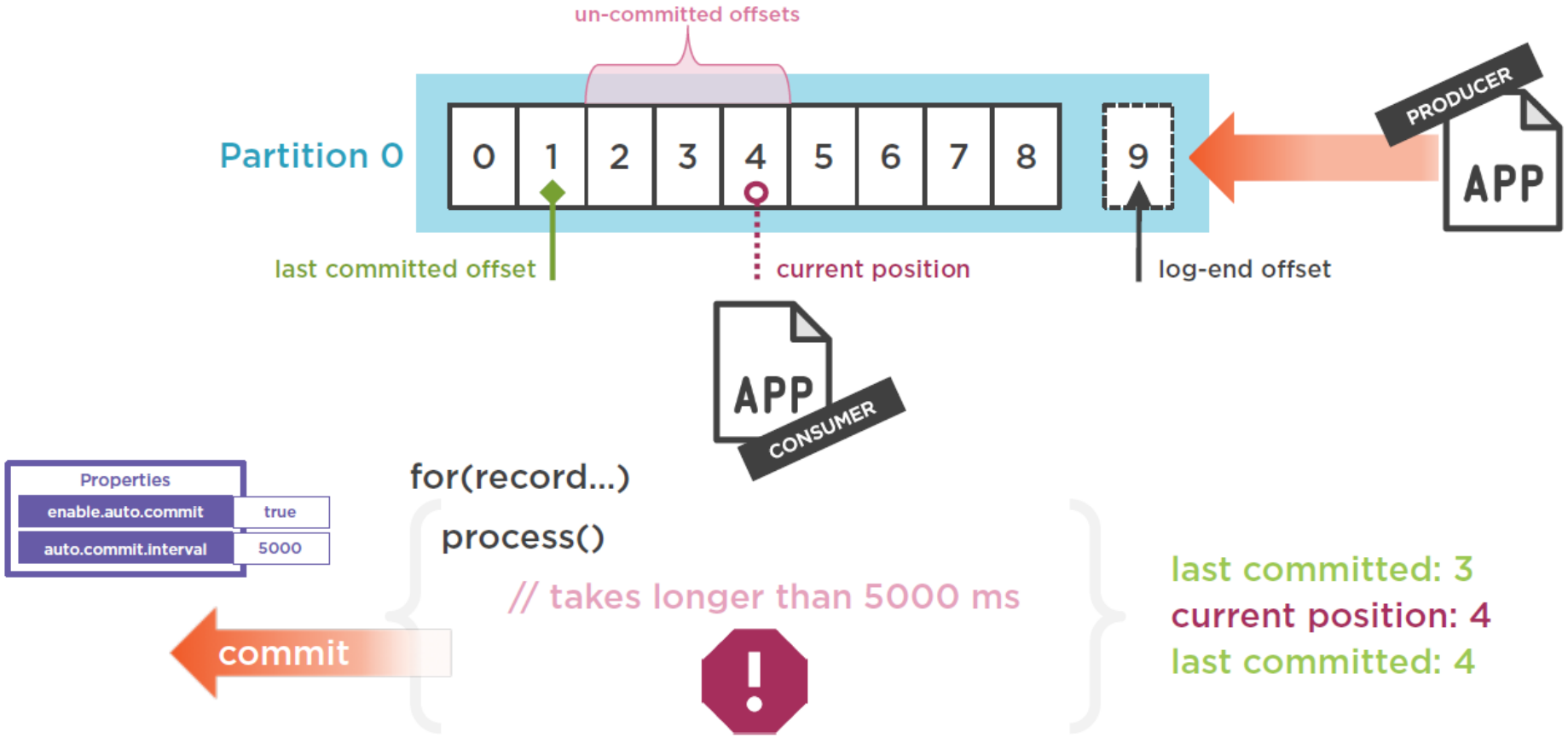
Mind the (Offset) Gap



The extent in which your system can be tolerant of eventually consistency is determined by its reliability.

Offset

Mind the (Offset) Gap



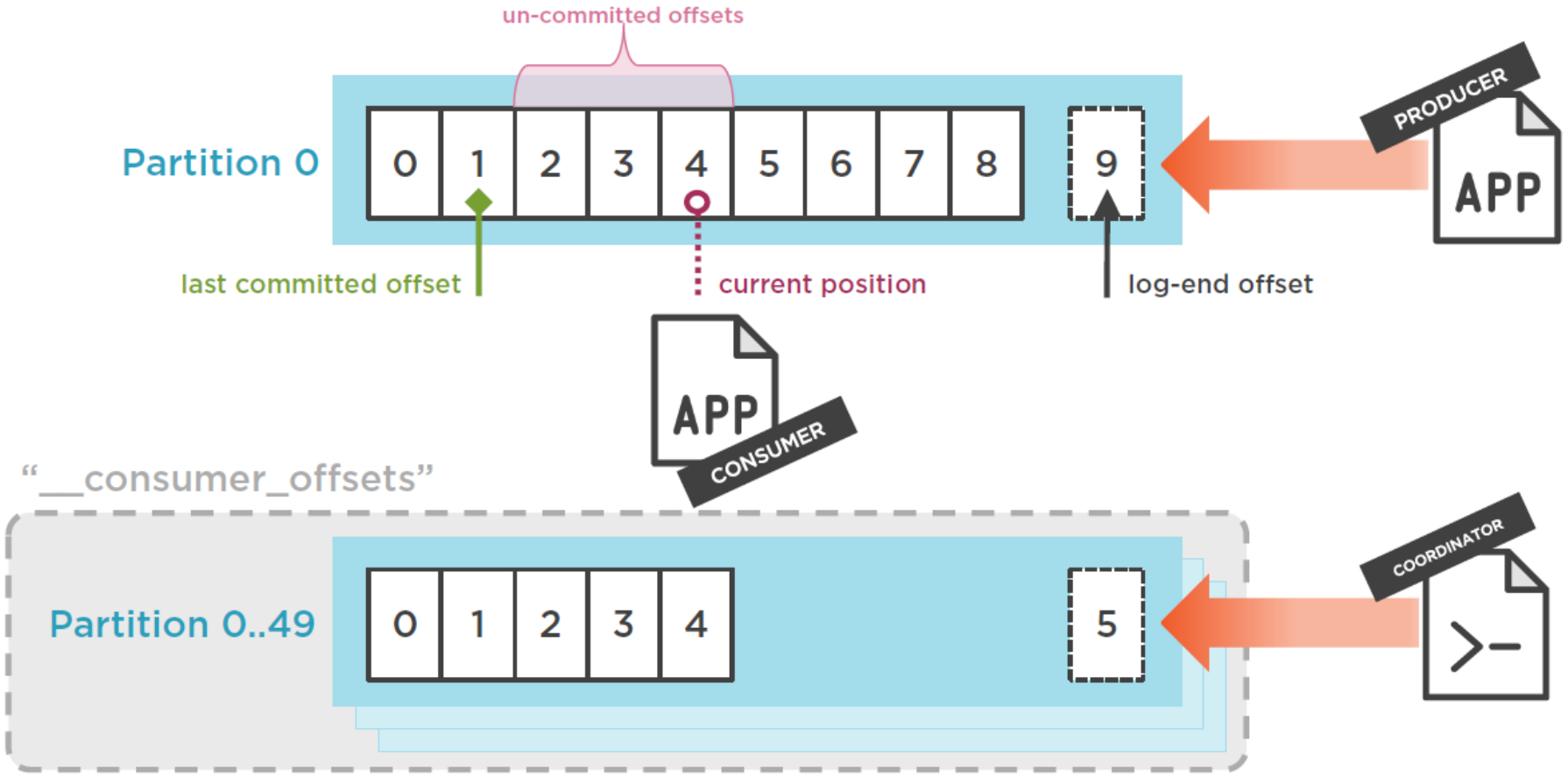
Offset

Offset Behavior

- Read != Committed
- Offset commit behavior is configurable
  - enable.auto.commit = true (default)
  - auto.commit.interval.ms = 5000 (default)
  - auto.offset.reset = “latest” (default)
    - “earliest”
    - “none”
- Single Consumer vs. Consumer Group

Offset

Storing the Offsets



Offset

Offset Management

- Automatic vs. Manual
  - enable.auto.commit = false
- Full control of offset commits
  - commitSync()
  - commitAsync()

Offset

```
try {
        for (...) { // Processing batches of records... }
                // Commit when you know you're done, after the batch is processed:
                myConsumer.commitSync();
        } catch (CommitFailedException) {
                log.error("there's not much else we can do at this point...");
}
```

commitSync

- **Synchronous**
  - blocks until receives response from cluster
- **Retries until succeeds or unrecoverable error**
  - retry.backoff.ms (default: 100)

Offset

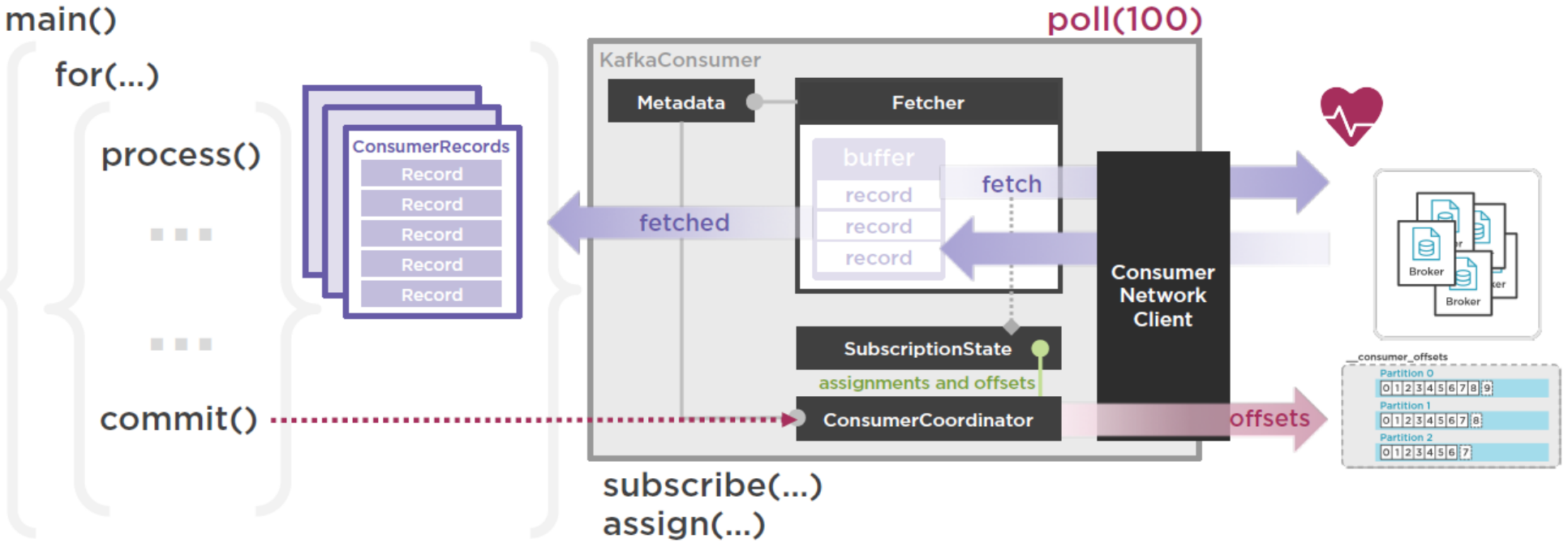
```
try {
        for (...) { // Processing batches of records... }
                // Not recommended:
                myConsumer.commitAsync();
                // Recommended:
                myConsumer.commitAsync(new OffsetCommitCallback() {
                                public void onComplete(..., ..., ...) { // do something...}
});
```

commitAsync

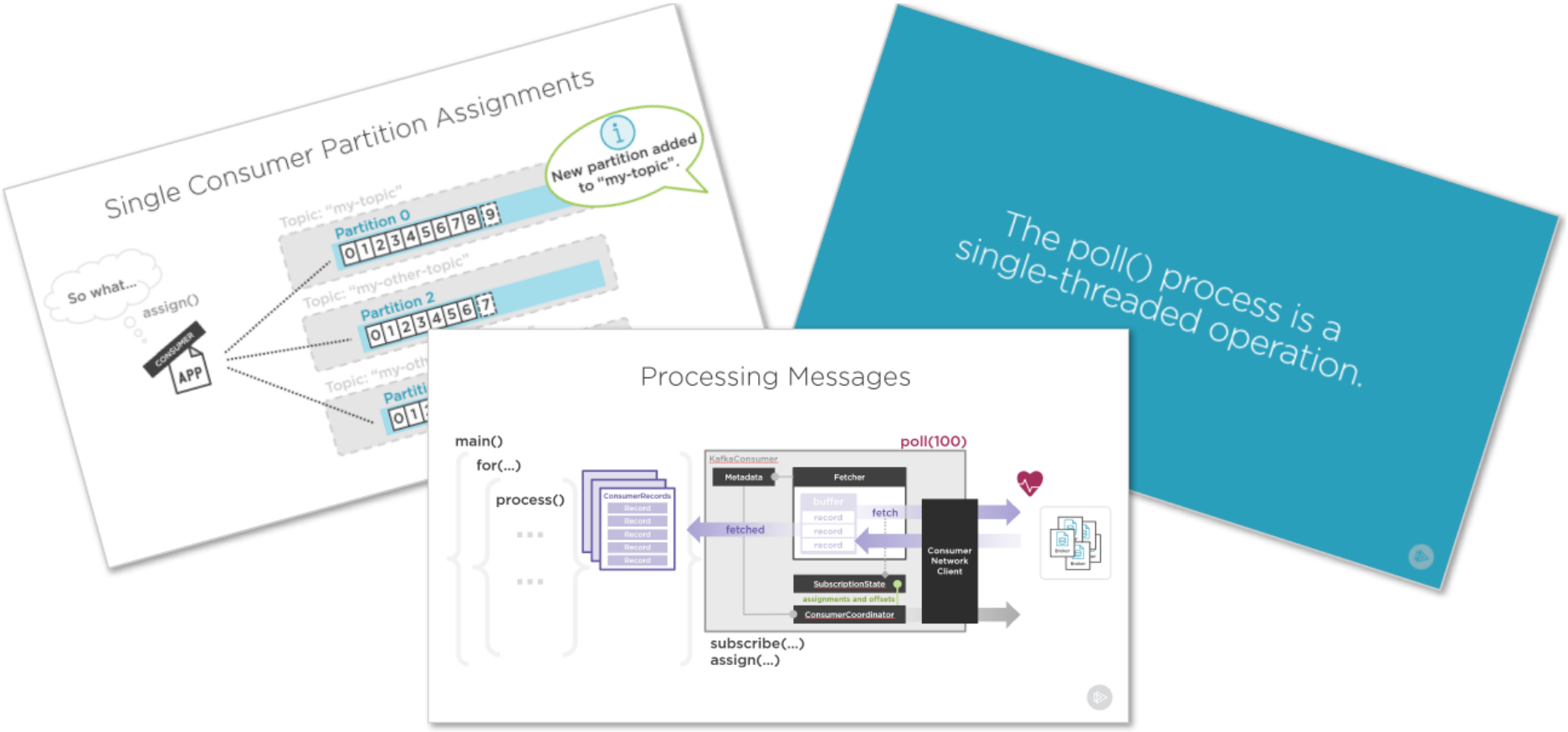
- Asynchronous
  - non-blocking but non-deterministic
- No retries
- Callback option

Offset

Committing Offsets



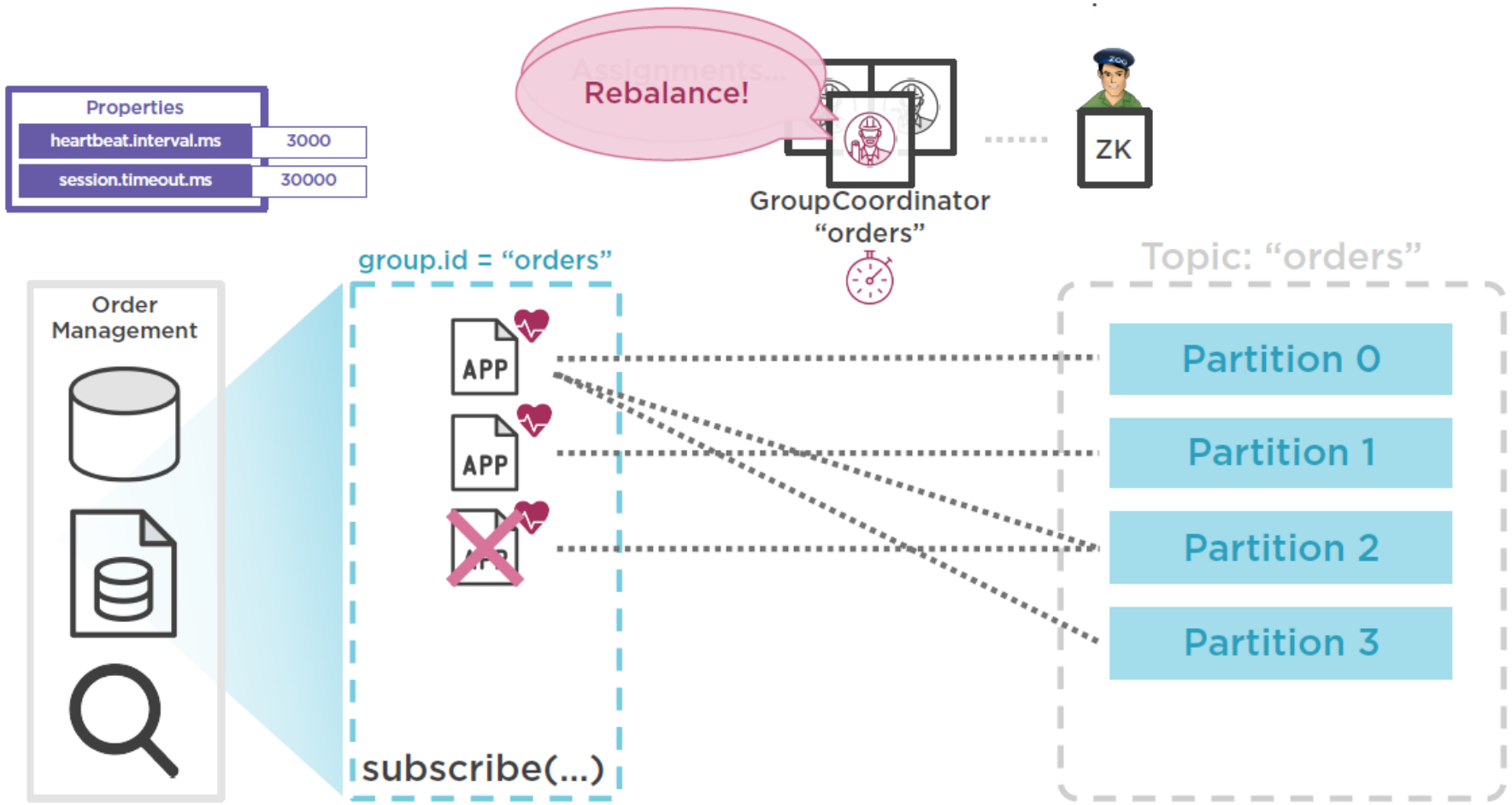
Scaling – out Consumers



Consumer Groups

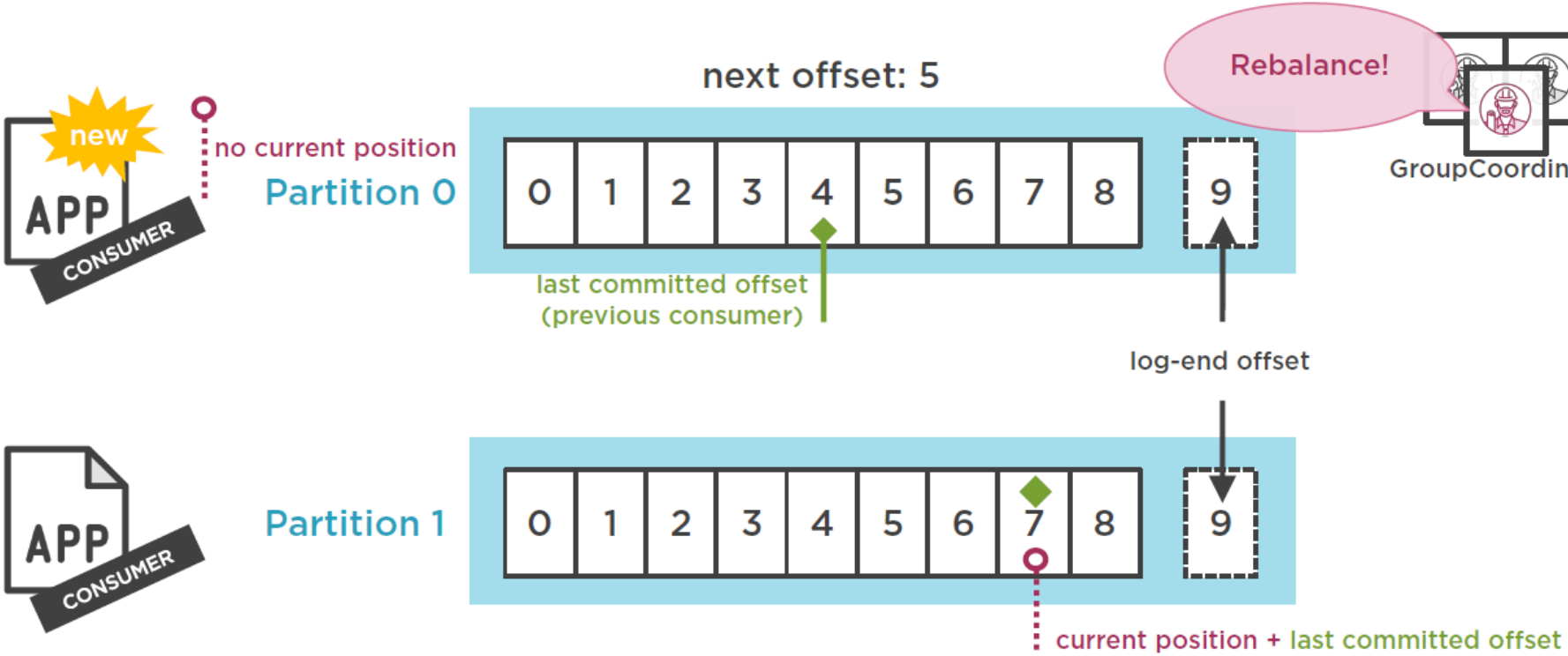
- Kafka's solution to Consumer-side scaleout
- Independent Consumers working as a team
  - "group.id" setting
- Sharing the message consumption and processing load
  - Parallelism and throughput
  - Redundancy
  - Performance

Consumer Groups



Consumer Groups

Consumer Group Rebalancing



Consumer Groups

Group Coordinator

- Evenly balances available Consumers to partitions
  - 1:1 Consumer-to-partition ratio
  - Can’t avoid over-provisioning
- Initiates the rebalancing protocol
  - Topic changes (partition added)
  - Consumer failure

Consumer Groups

Consumer Configuration

- Consumer performance and efficiency
  - fetch.min.bytes
  - max.fetch.wait.ms
  - max.partition.fetch.bytes
  - max.poll.records

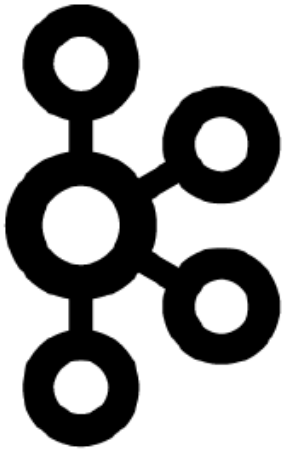
Summary

- Kafka Consumer Internals
  - Properties -> ConsumerConfig
  - Message -> ConsumerRecord
  - Subscriptions and assigments
  - Message polling and consumption
  - Offset management
- Consumer Groups
- Consumer Configuration
- Java-based Consumer

Course Map – Challenges

1	Primary Use Cases for Apache Kafka
2	Challenges Remain
3	Challenges in Governance and Evolution
4	Kafka Schema Registry
5	Challenges with consistency and Productivity
6	Apache Kafka Connect
7	Challenges with Fast Data
8	Kafka Streams

Primary Use Cases for Apache Kafka

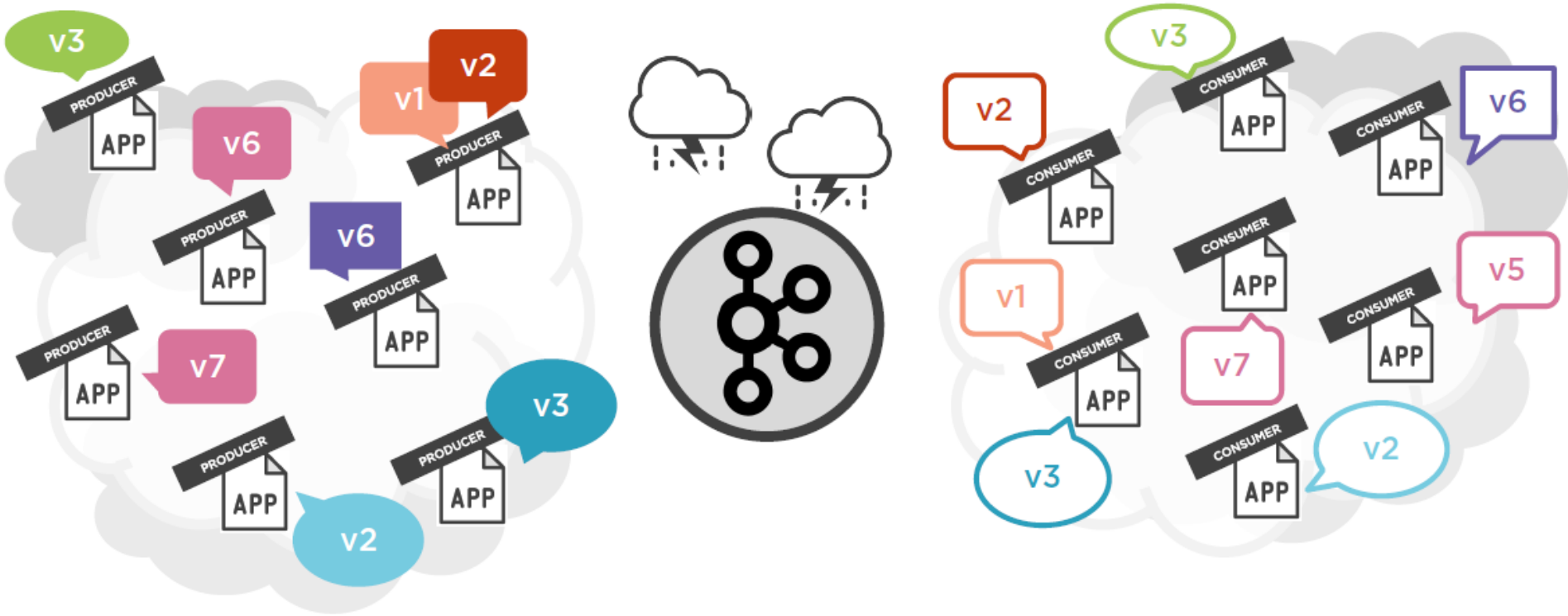


- Connecting disparate sources of data
- Large-scale data movement pipelines
- “Big Data” integration

Challenges

- Governance and Data Evolution
- Consistency and Productivity
- Big and Fast Data

Challenges in Governance and Evolution



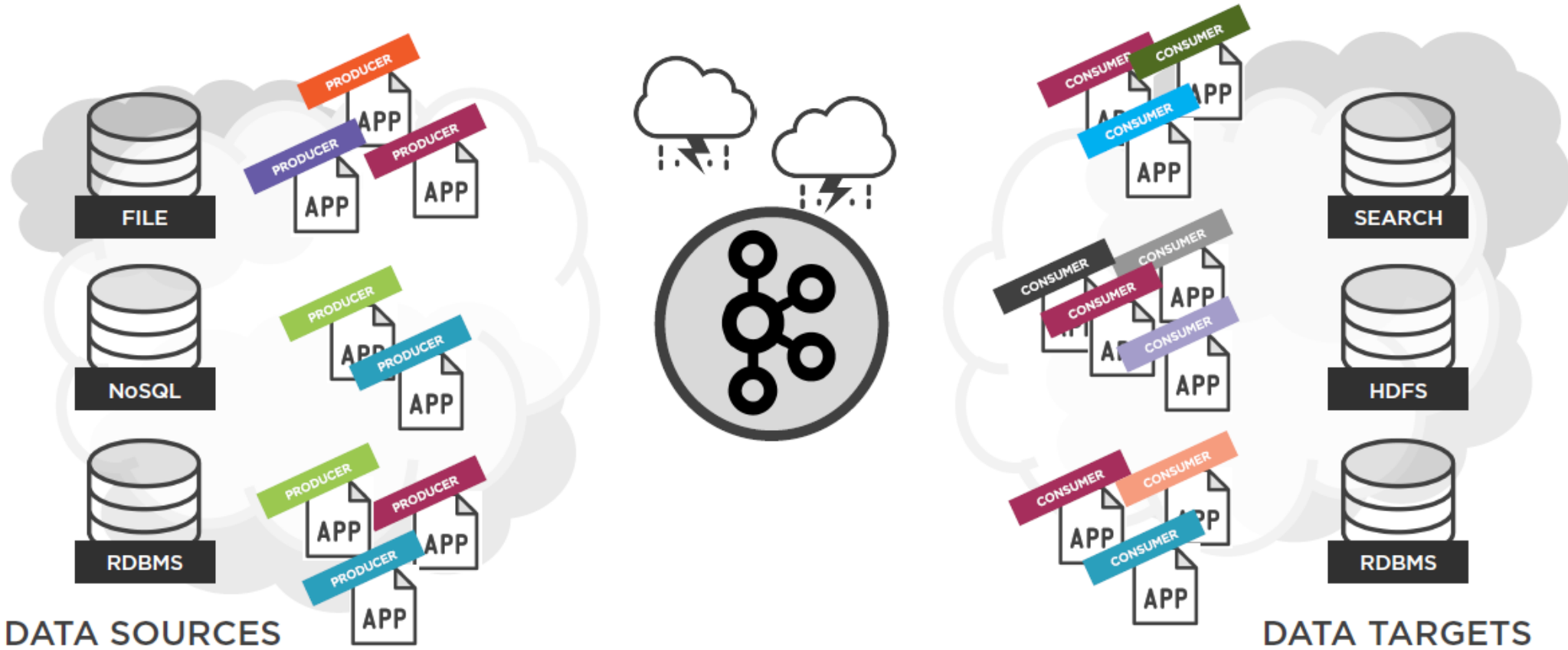
KEY.SERIALIZER ..... KEY.DESERIALIZER

VALUE.SERIALIZER ..... VALUE.DESERIALIZER

Kafka Schema Registry

- Apache Avro serialization format
- First-class Avro serializers and deserializers
- Schema registry and version management
- RESTful service discovery
- Compatibility broker
- Open source with Apache license

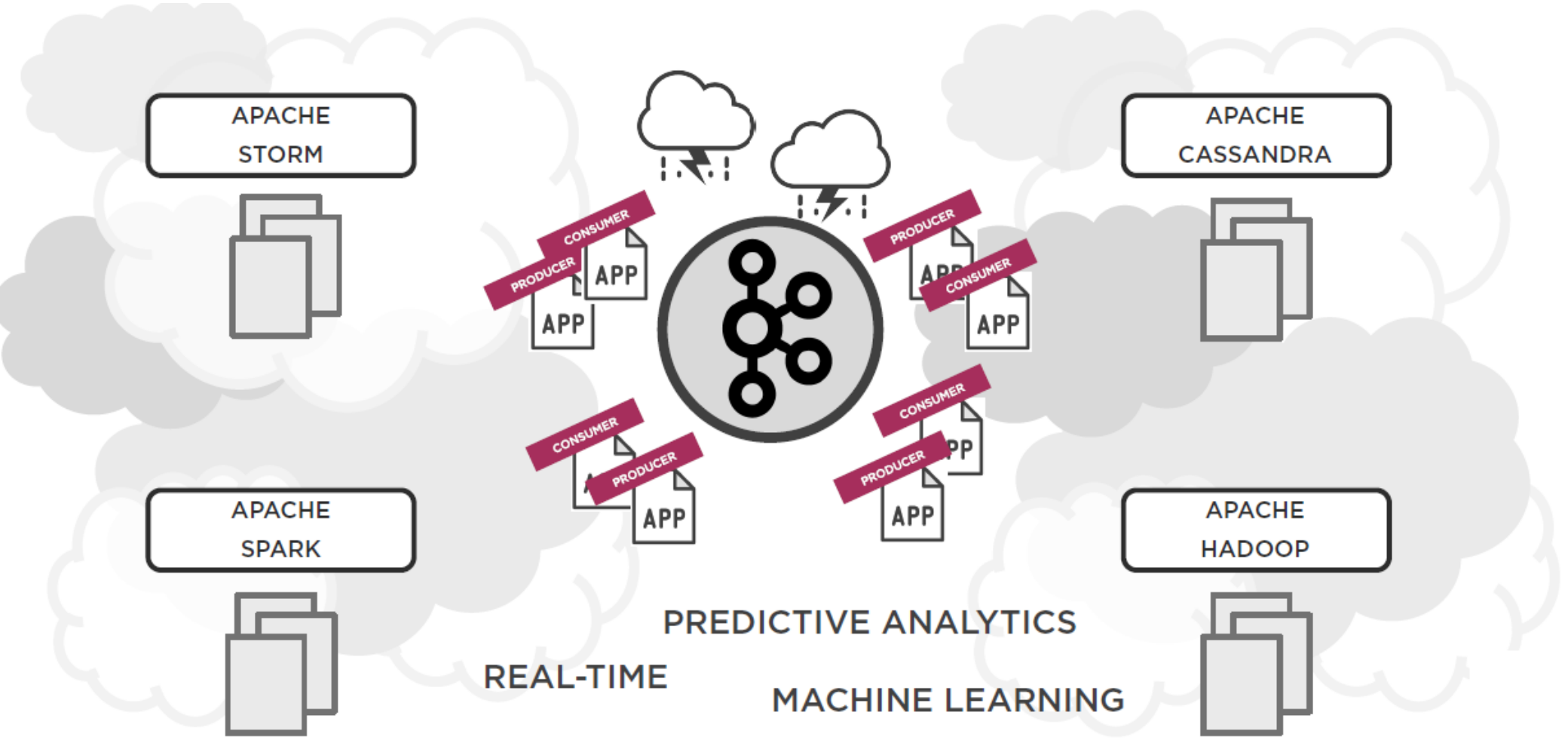
Challenges with Consistency and Productivity



Apache Kafka Connect

- Common framework for integration
  - Standardization of common approaches
  - Producers and Consumers
- Platform Connectors
  - Oracle, HP, etc.
  - 50+ and growing
- Connector Hub

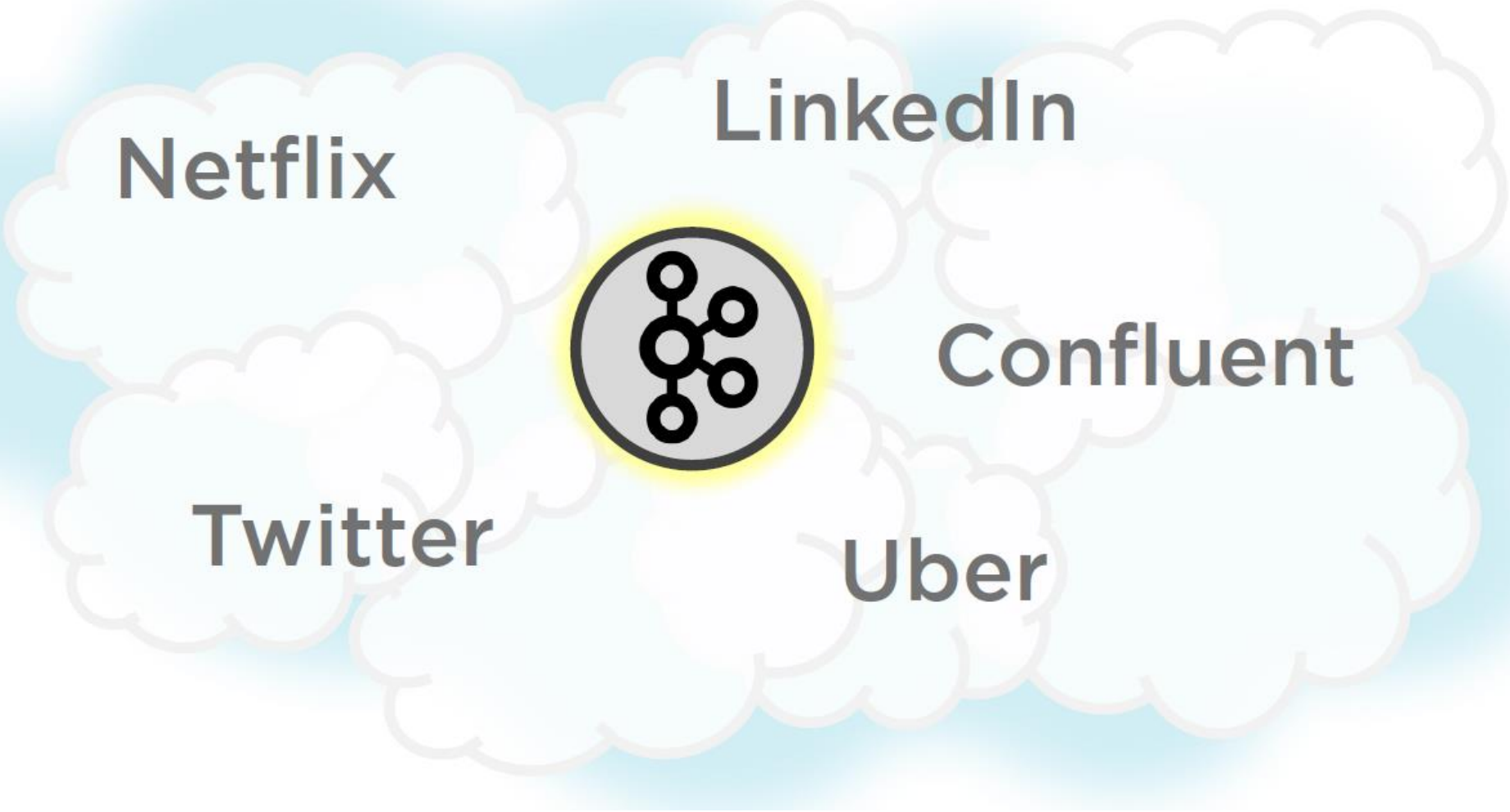
Challenges with Fast Data



Kafka Streams

- Leverages existing Kafka machinery
- Single infrastructure solution
  - At least for streaming-based processing
- Embeddable within existing applications

Growing and Healthy Ecosystem



Capgemini

CONSULTING.TECHNOLOGY.OUTSOURCING

People matter, results count.

About Capgemini

With almost 180,000 people in over 40 countries, Capgemini is one of the world's foremost providers of consulting, technology and outsourcing services. The Group reported 2014 global revenues of EUR 10.573 billion.

Together with its clients, Capgemini creates and delivers business and technology solutions that fit their needs and drive the results they want. A deeply multicultural organization, Capgemini has developed its own way of working, the Collaborative Business Experience™, and draws on Rightshore®, its worldwide delivery model.

www.capgemini.com