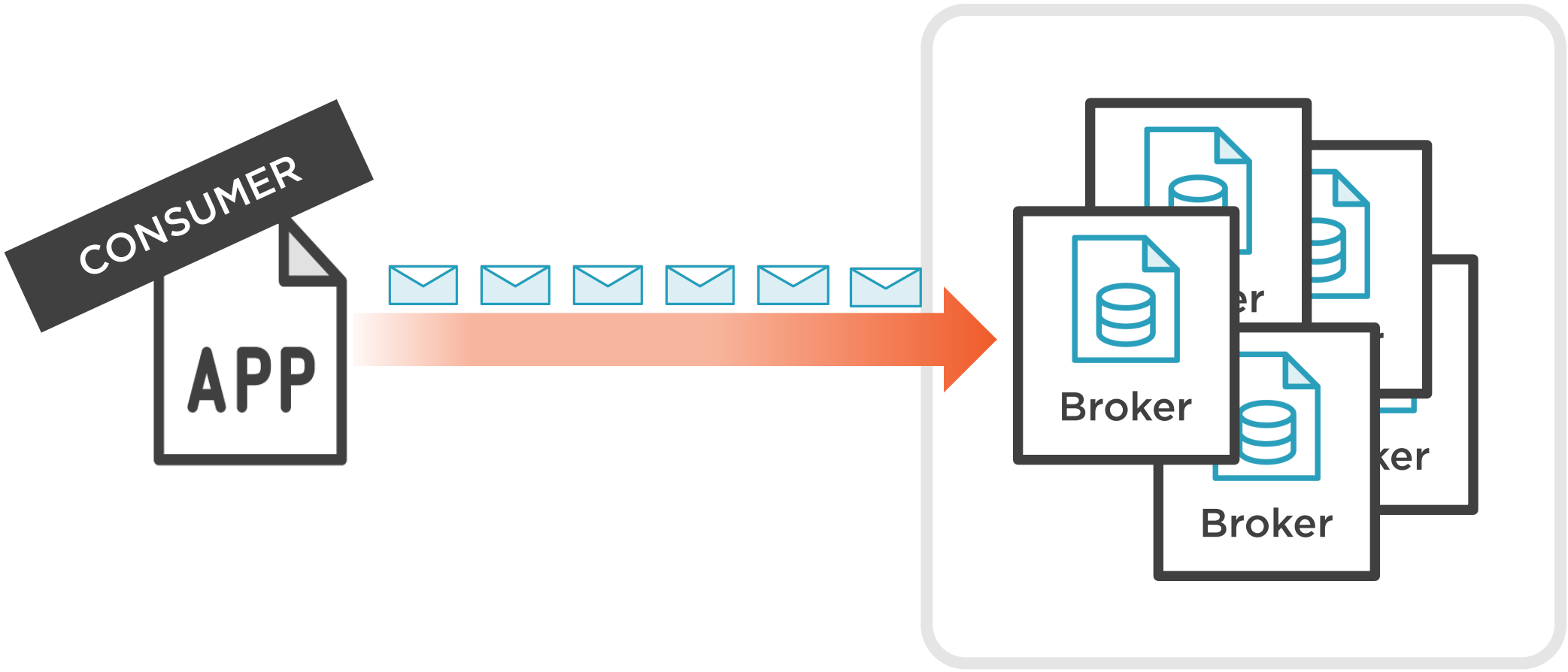
Consuming Messages with Kafka Consumers and Consumer Groups

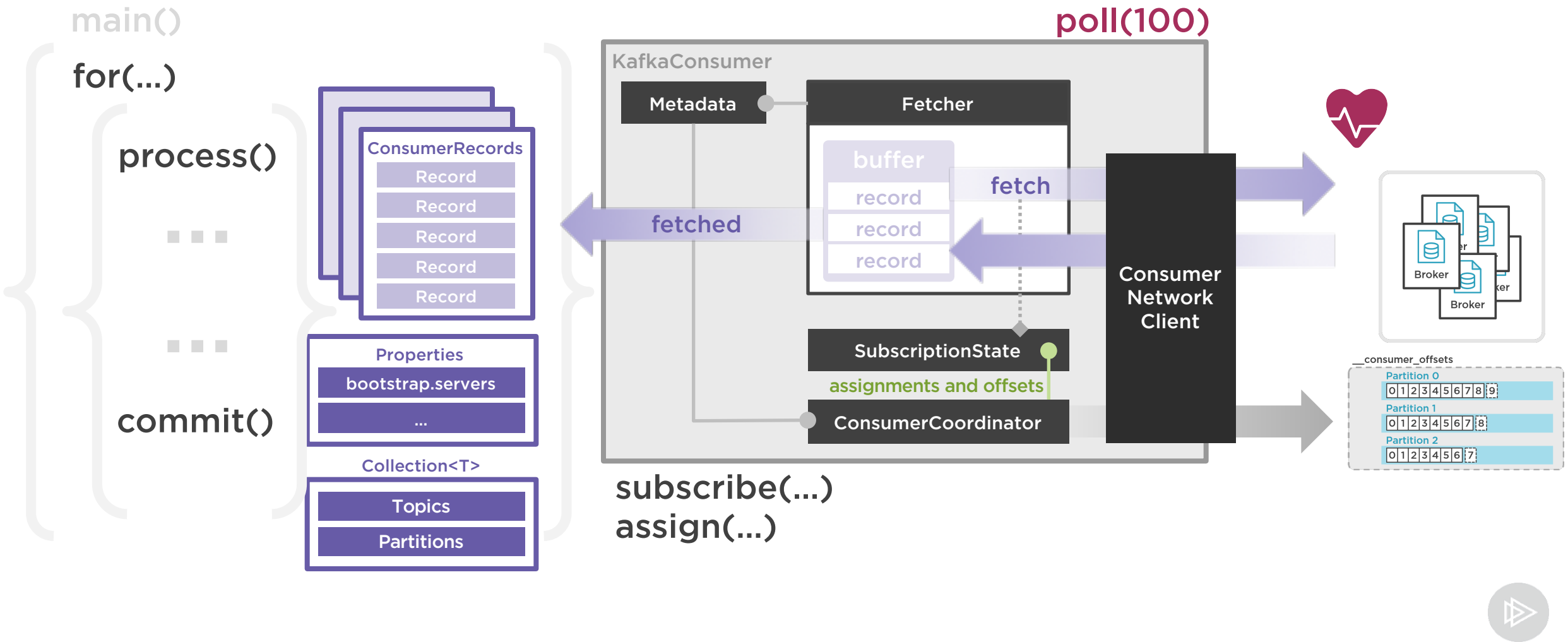
Ryan Plant
COURSE AUTHOR

@ryan_plant blog.ryanplant.com

Kafka Consumer Externals



Kafka Consumer Internals



http://kafka.apache.org/documentation.html#consumerconfigs

```
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");
```

Kafka Consumer: Required Properties

bootstrap.servers

- Cluster membership: partition leaders, etc.

key and value deserializers

- Classes used for message deserialization

Creating a 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);

      }

}
```

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-*");

     }

}
```

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);
```

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
```

subscribe()

- For topics (dynamic/automatic)
- One topic, one-to-many partitions
- Many topics, many more partitions

assign()

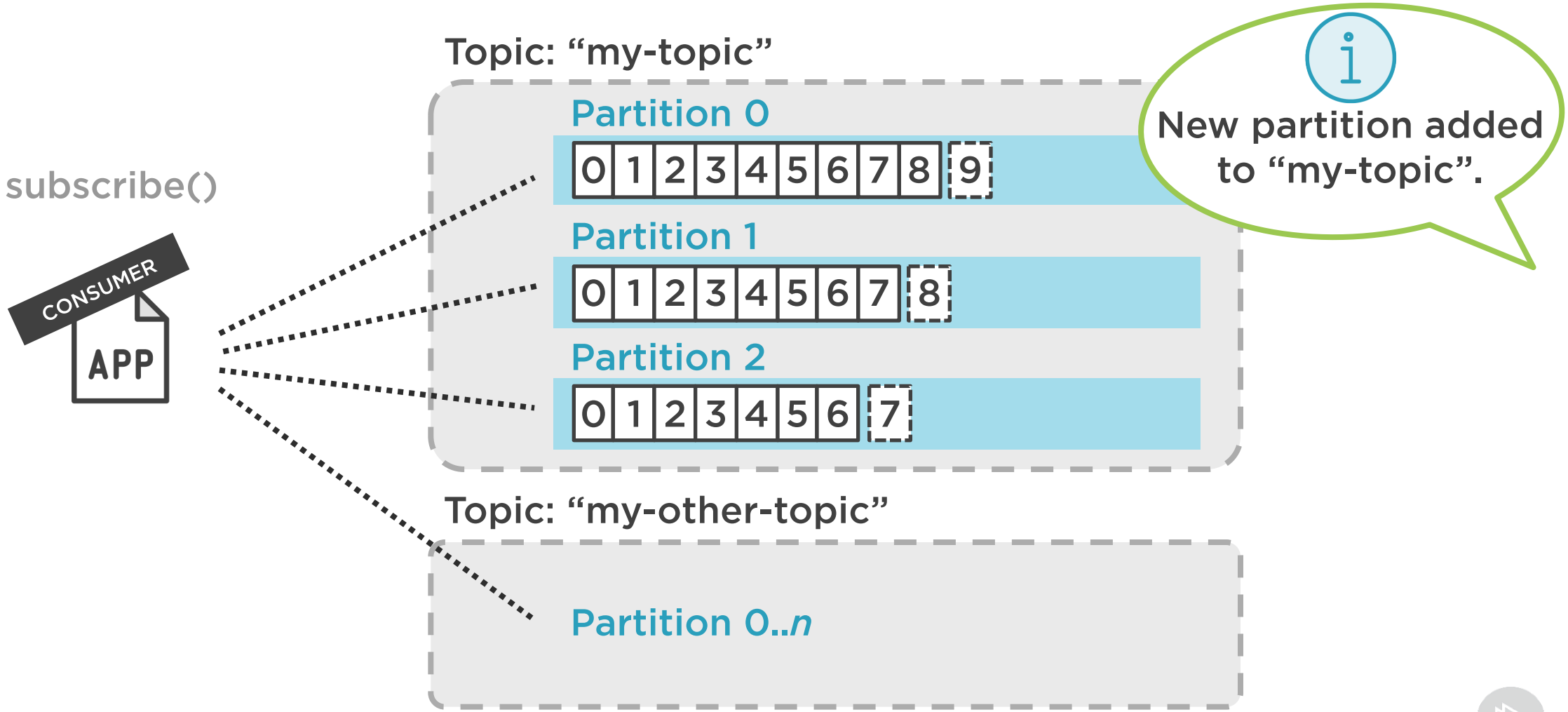
- For partitions
- One or more partitions, regardless of topic
- Manual, self-administering mode

Manual Partition Assignment

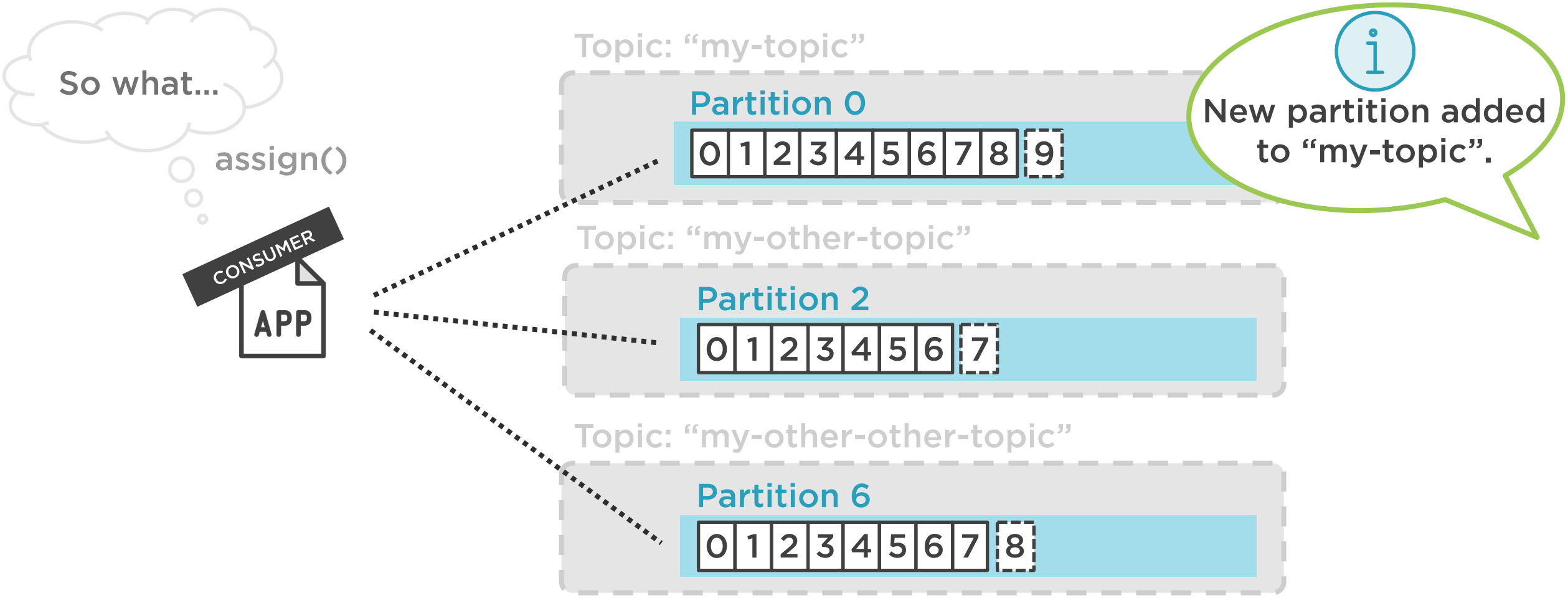
```
// Similar pattern as subscribe():
TopicPartition partition0 = new TopicPartition(“myTopic”, 0);
ArrayList<TopicPartition> partitions = new ArrayList<TopicPartition>();
partitions.add(partition0);

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

Single Consumer Topic Subscriptions



Single Consumer Partition Assignments



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

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();
      }
}
```

Demo

Single Consumer in Java

- Same setup as before

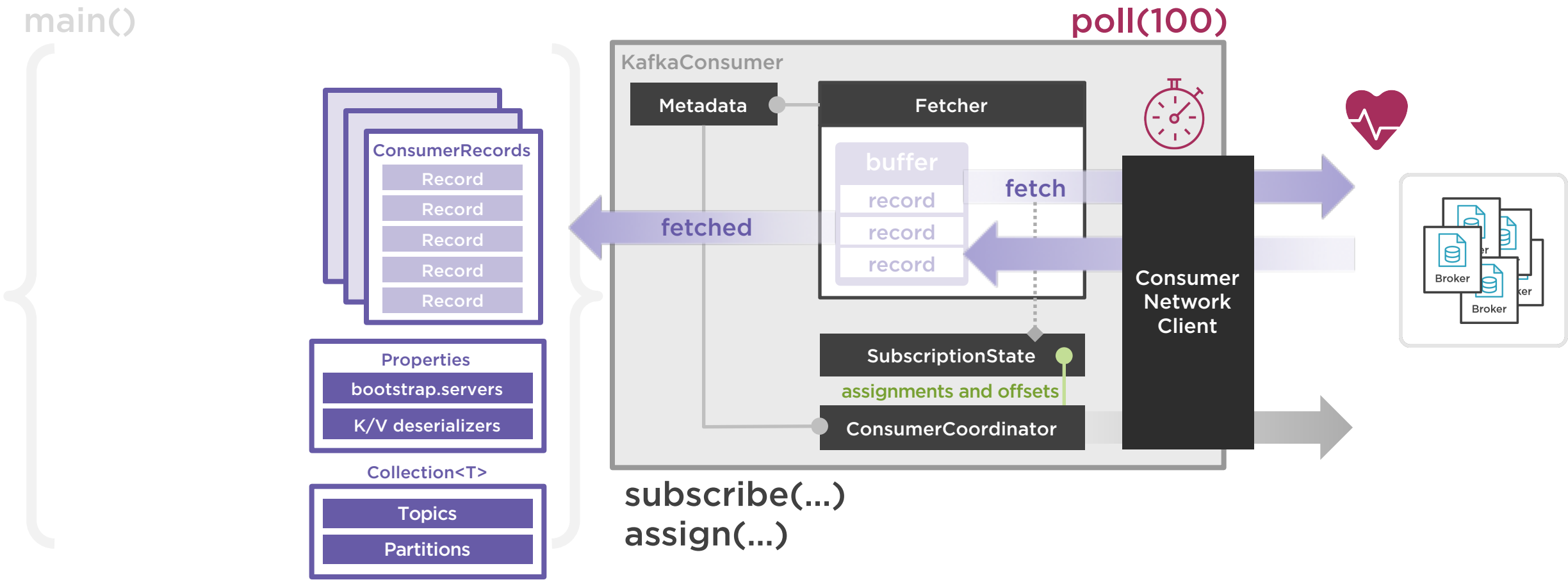
Cluster setup:

- Single broker
- Two topics
- Three partitions per topic
- Single replication factor

Look for:

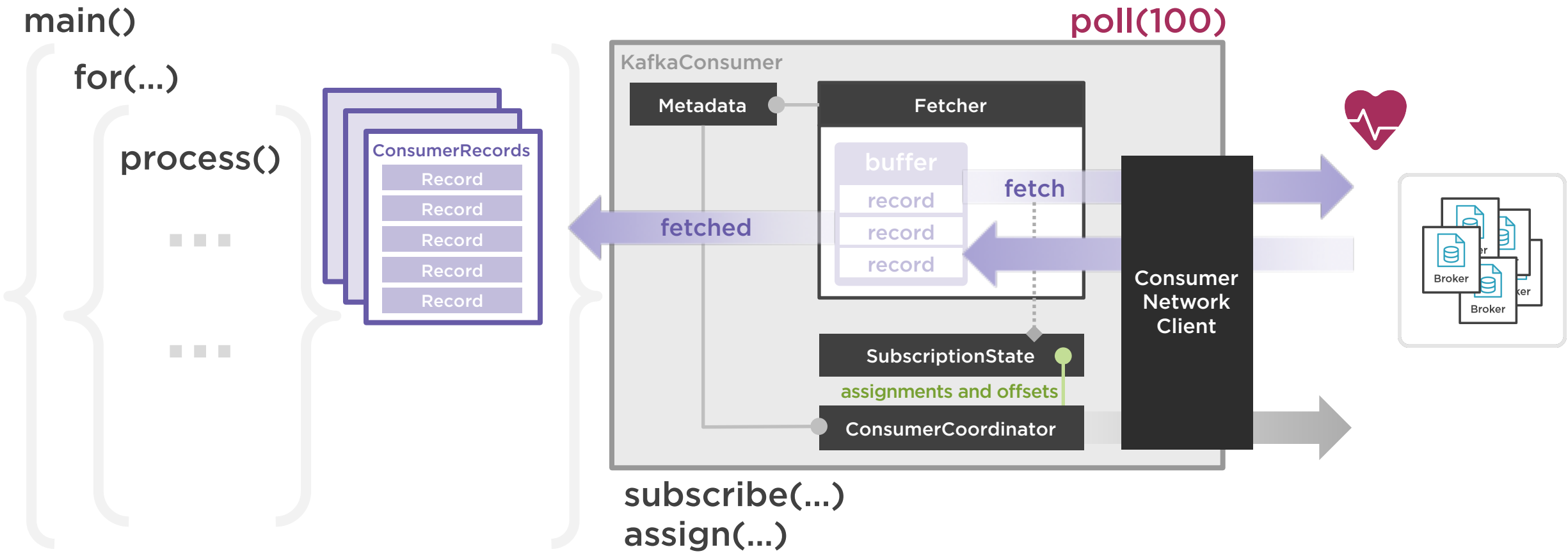
- kafka-producer-perf-test.sh
- subscribe() and assign()
- Add new partition
- Compare Consumer output

Kafka Consumer Polling

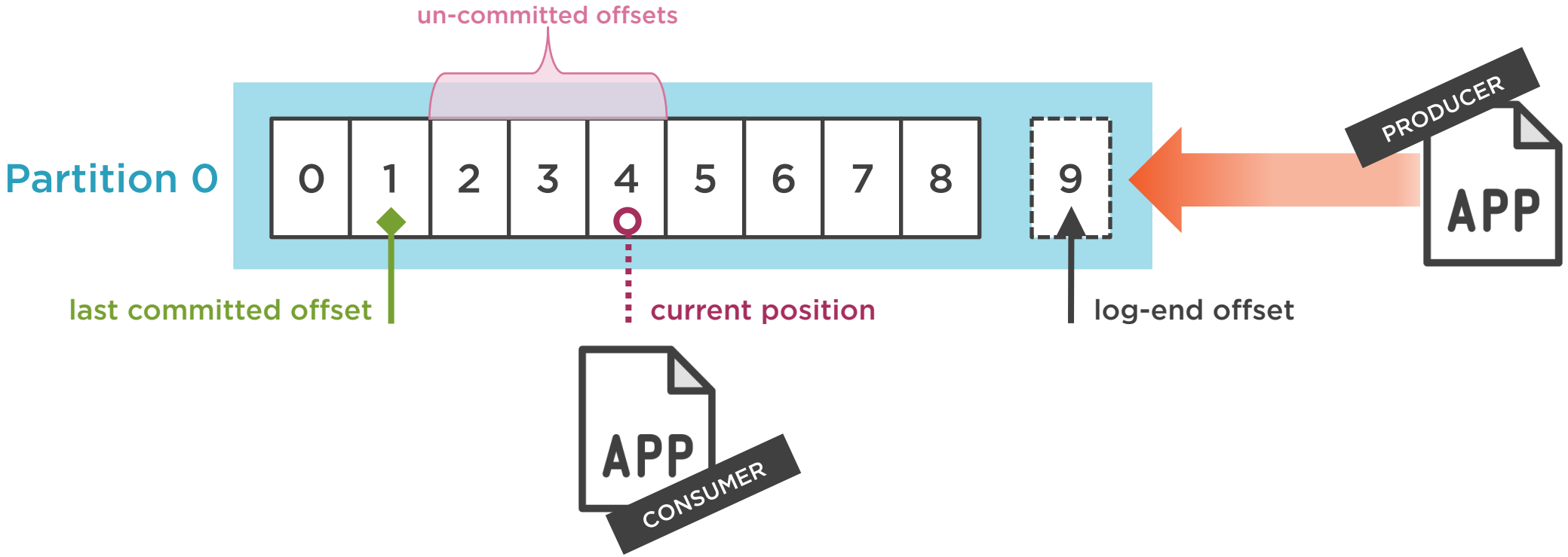


The poll() process is a single-threaded operation.

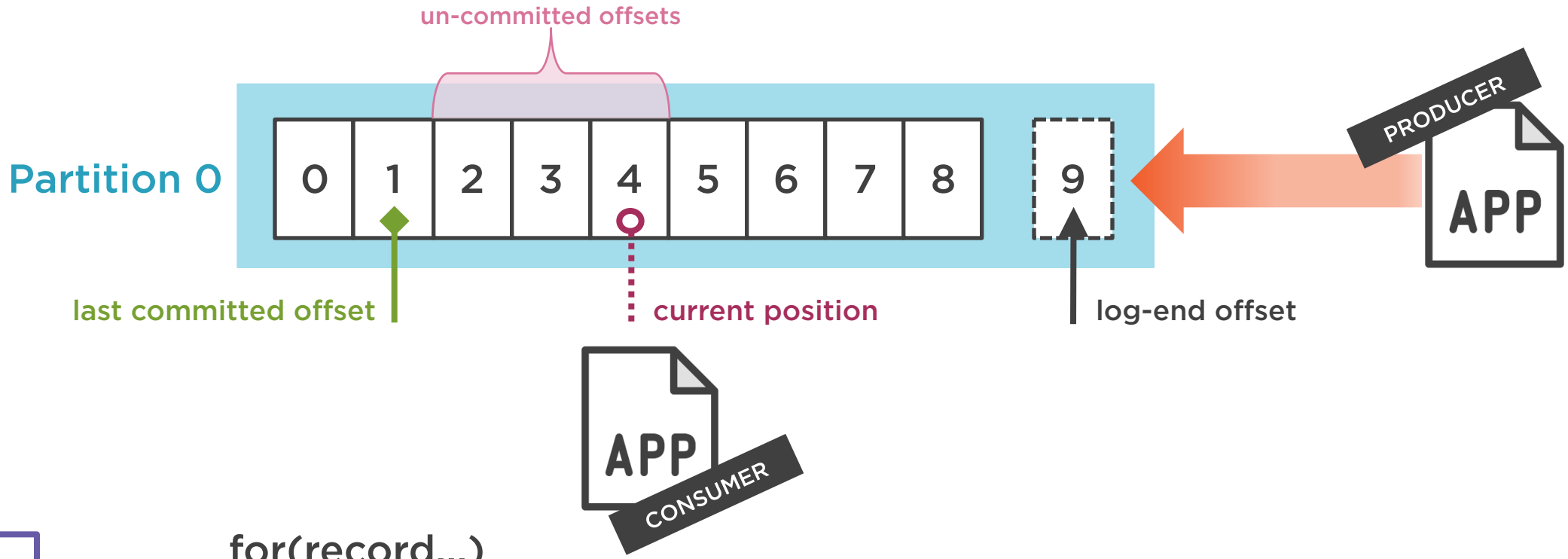
Processing Messages



More About the Offset



Mind the (Offset) Gap



Properties	
enable.auto.commit	true
auto.commit.interval	5000

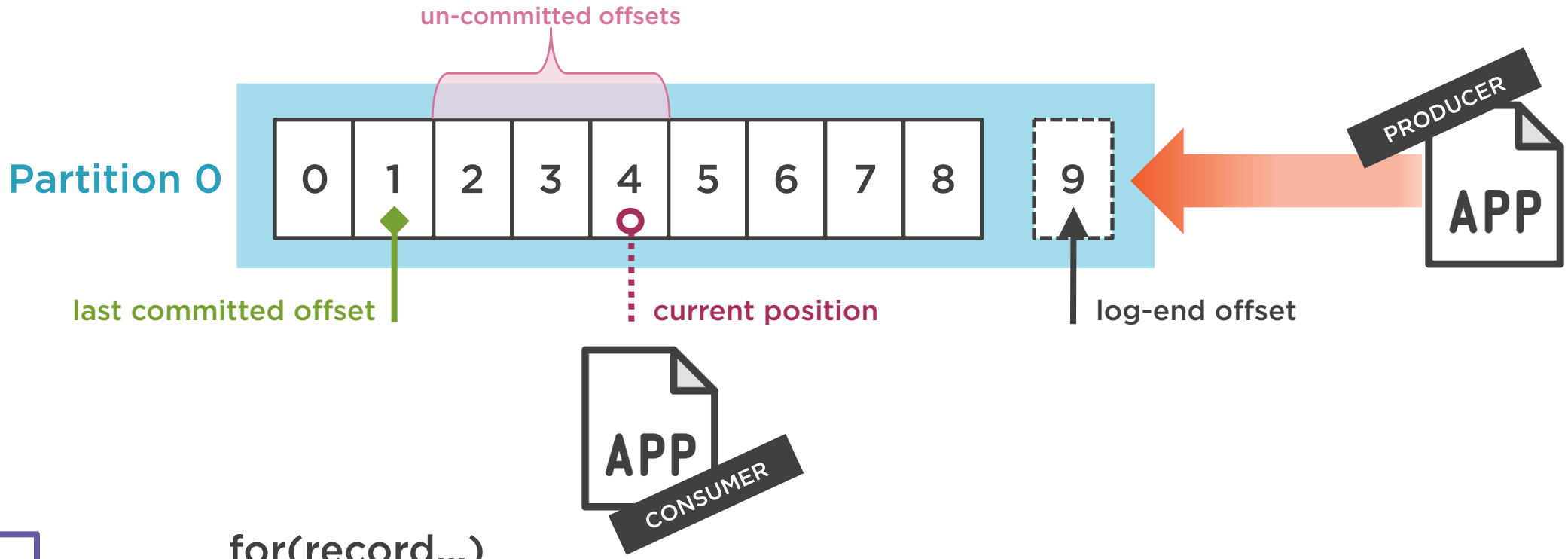
```
for(record...)
    process()
        // takes longer than 5000 ms
```

commit

last committed: 3
current position: 4
last committed: 4

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

Mind the (Offset) Gap



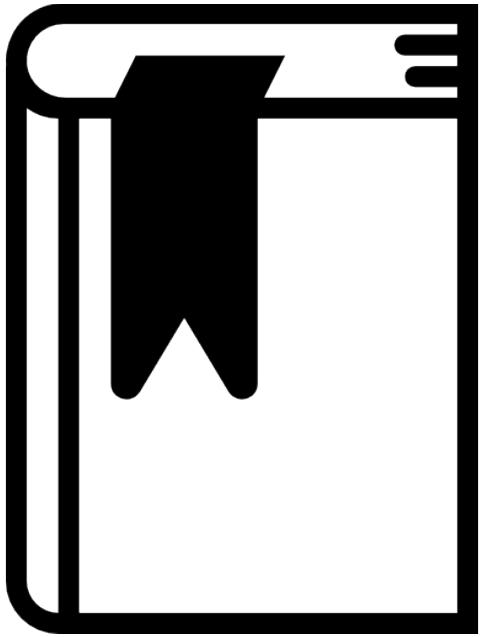
Properties	
enable.auto.commit	true
auto.commit.interval	5000

```
for(record...)
    process()
        // takes longer than 5000 ms
```

commit

last committed: 3
current position: 4
last committed: 4

Offset Behavior



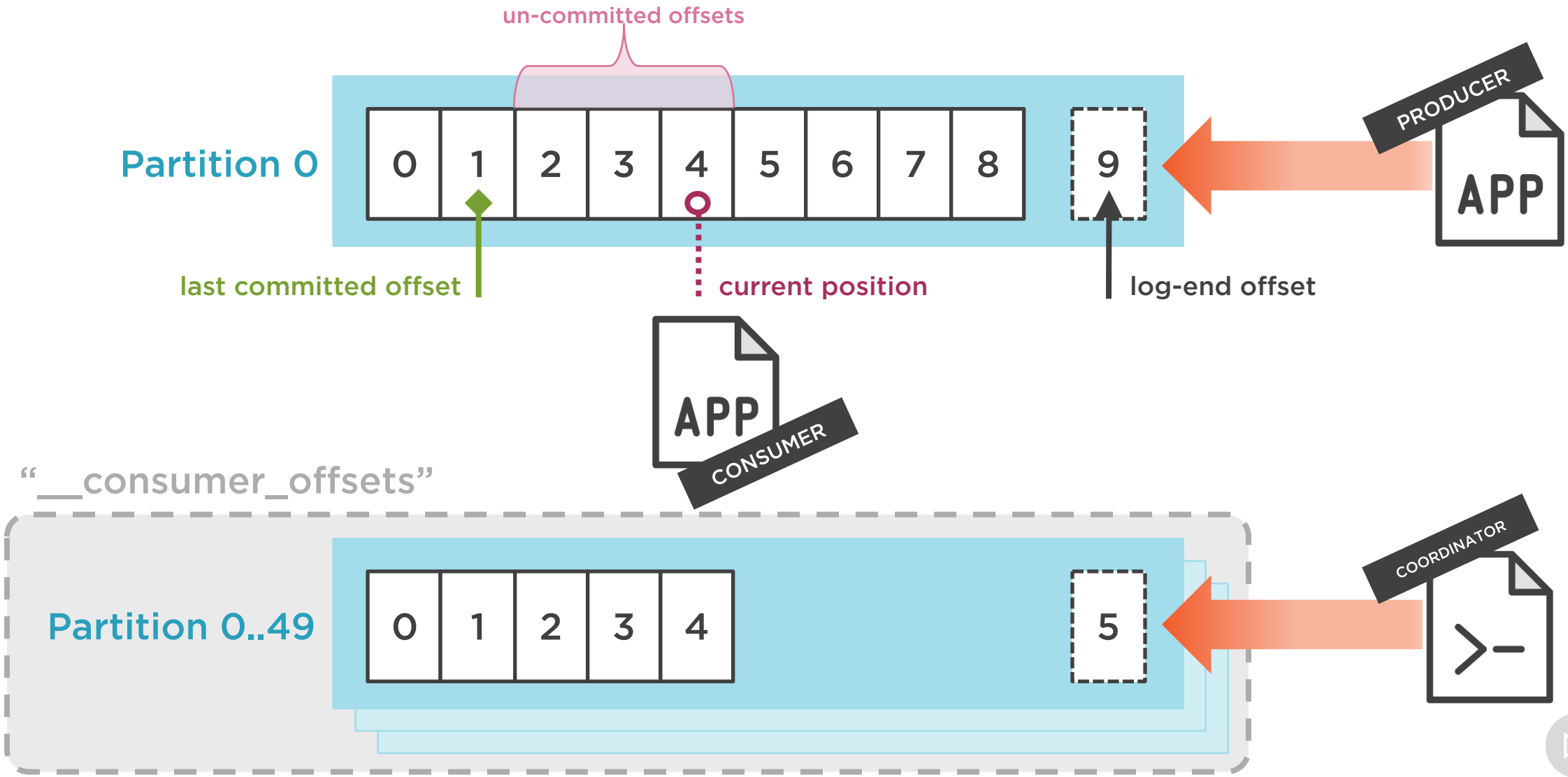
Read != Committed

Offset commit behavior is configurable

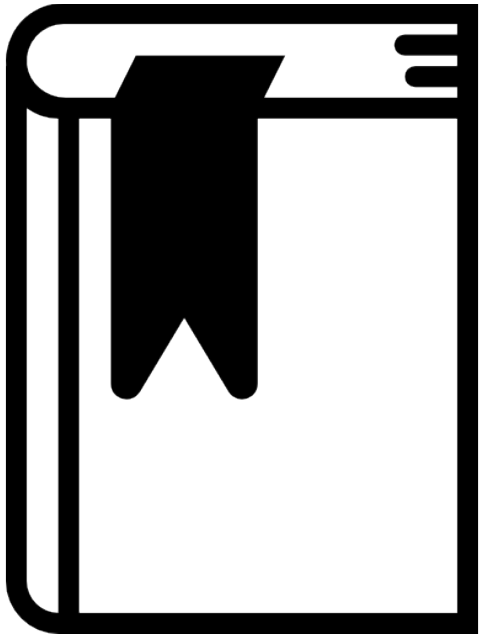
- enable.auto.commit = true (default)
- auto.commit.interval.ms = 5000 (default)
- auto.offset.reset = “latest” (default)
  - “earliest”
  - “none”

Single Consumer vs. Consumer Group

Storing the Offsets



Offset Management



Automatic vs. Manual

- enable.auto.commit = **false**

Full control of offset commits

- commitSync()
- commitAsync()

```
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)

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

commitAsync

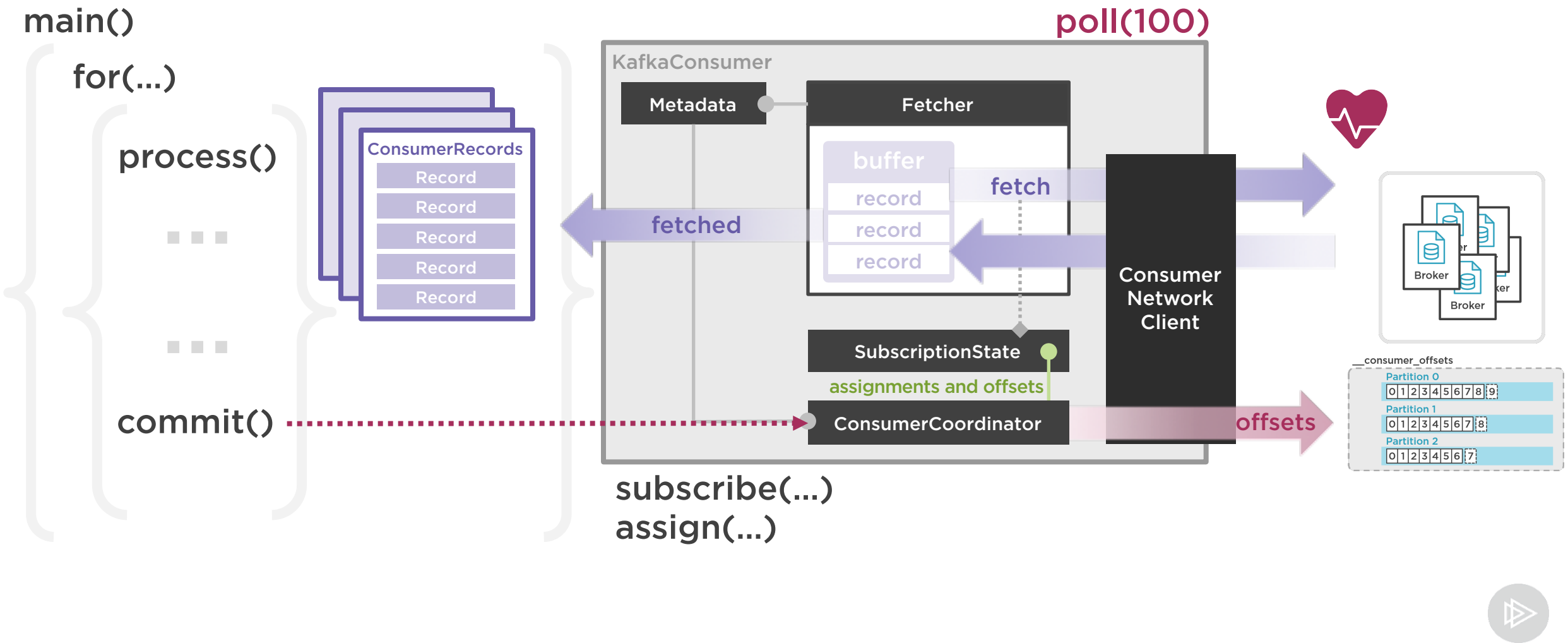
Asynchronous

- non-blocking but non-deterministic

No retries

Callback option

Committing Offsets



Going It Alone

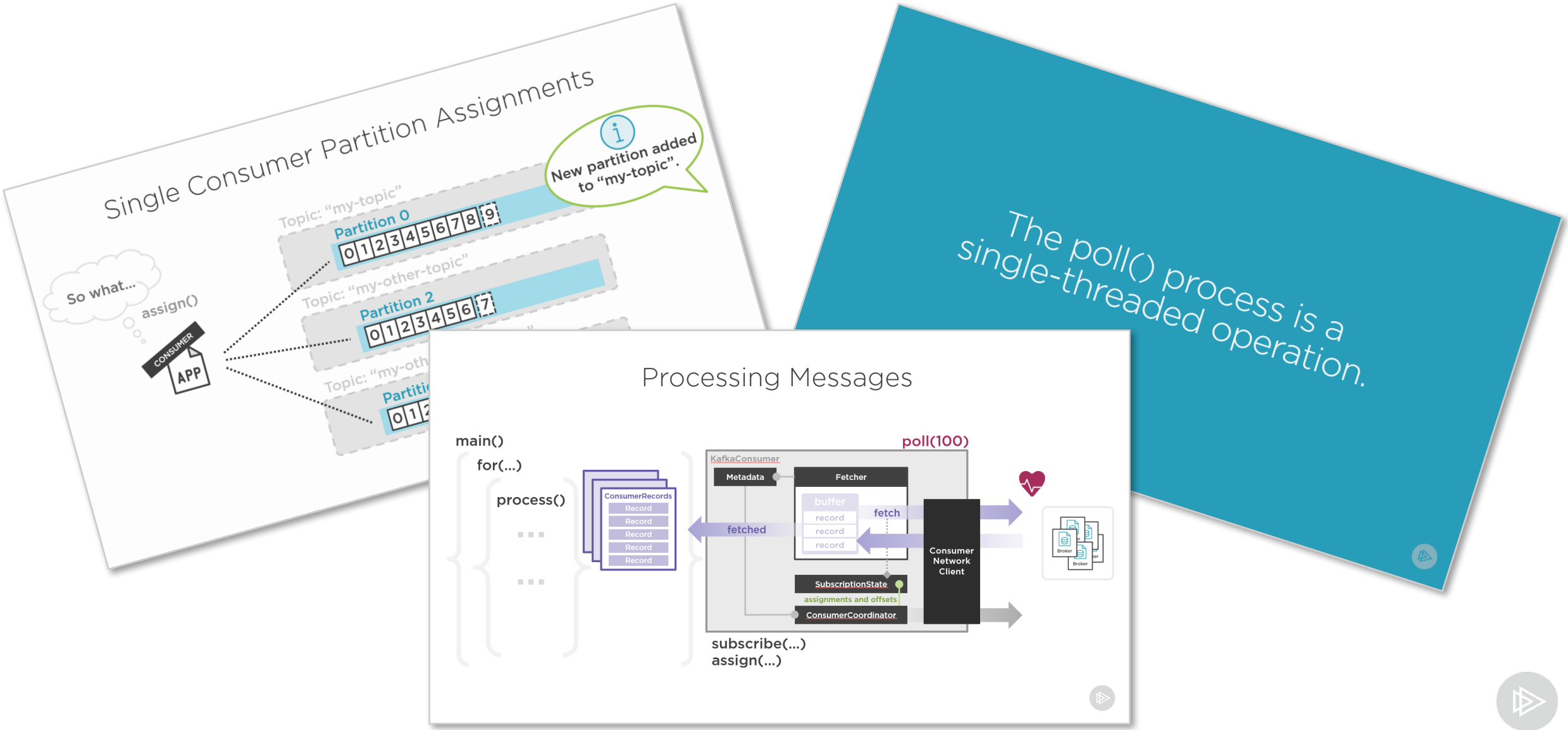
Consistency control

- When is “done”

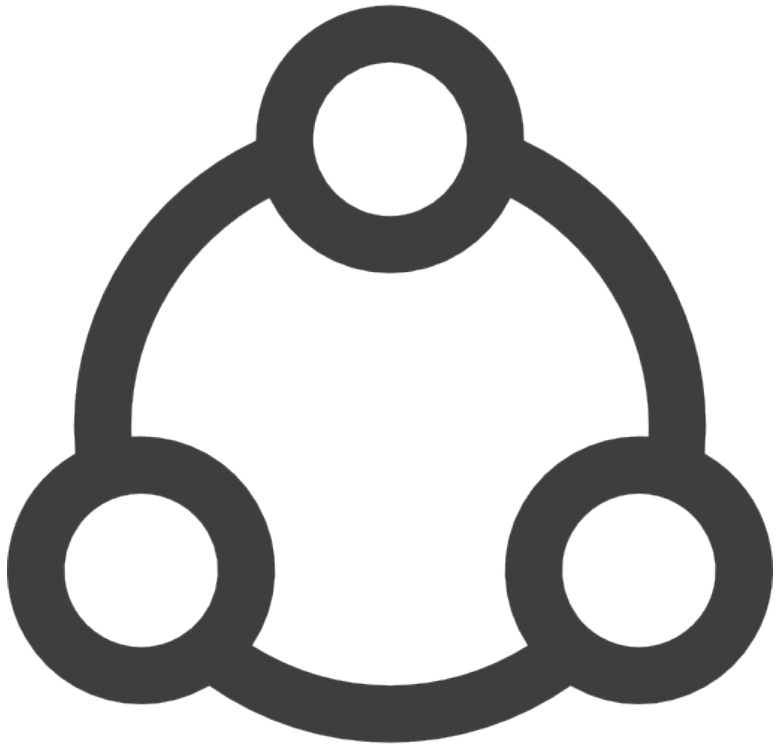
Atomicity

- Exactly once vs. At-least-once

Scaling-out Consumers



Consumer Groups



Kafka's solution to Consumer-side scale-out

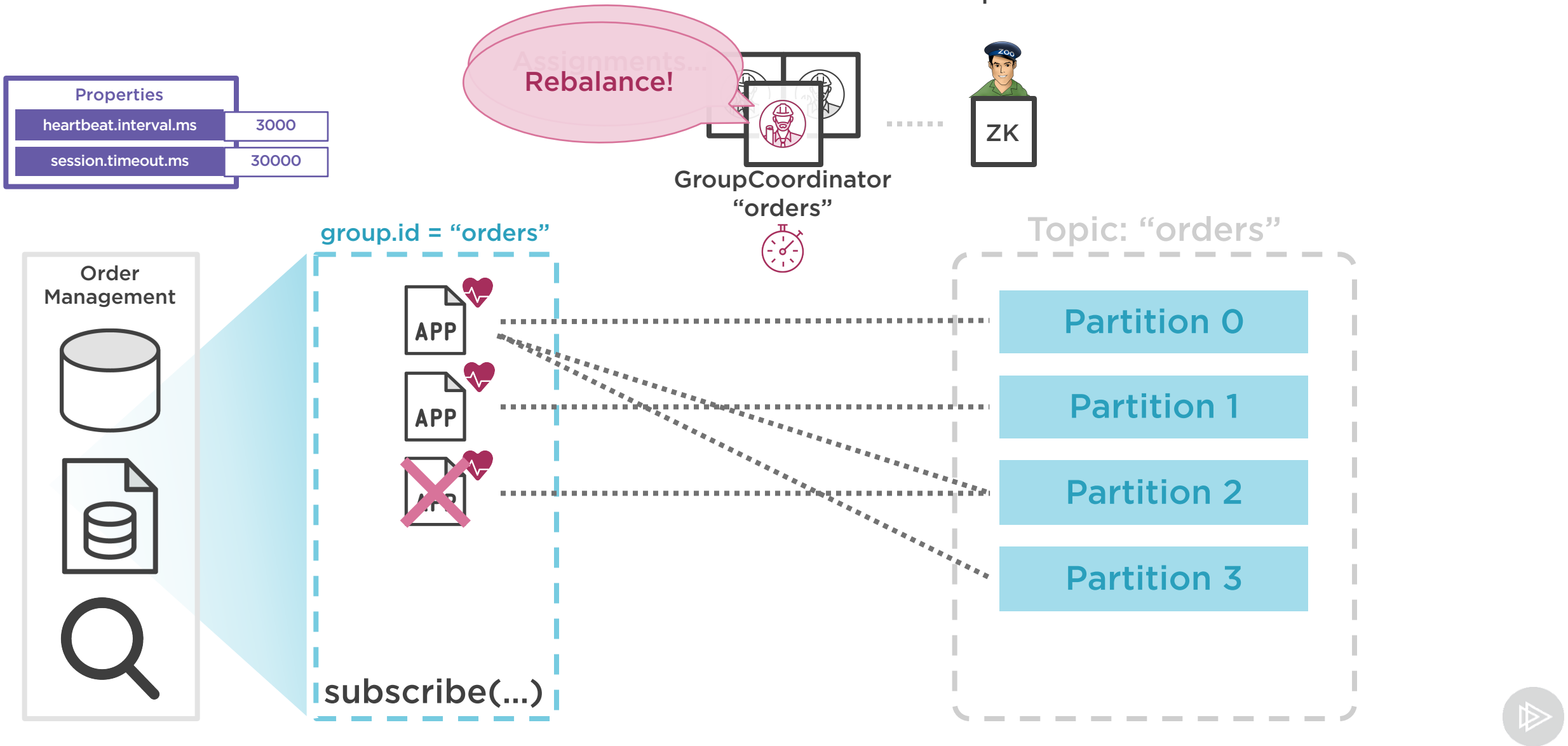
Independent Consumers working as a team

- "group.id" setting

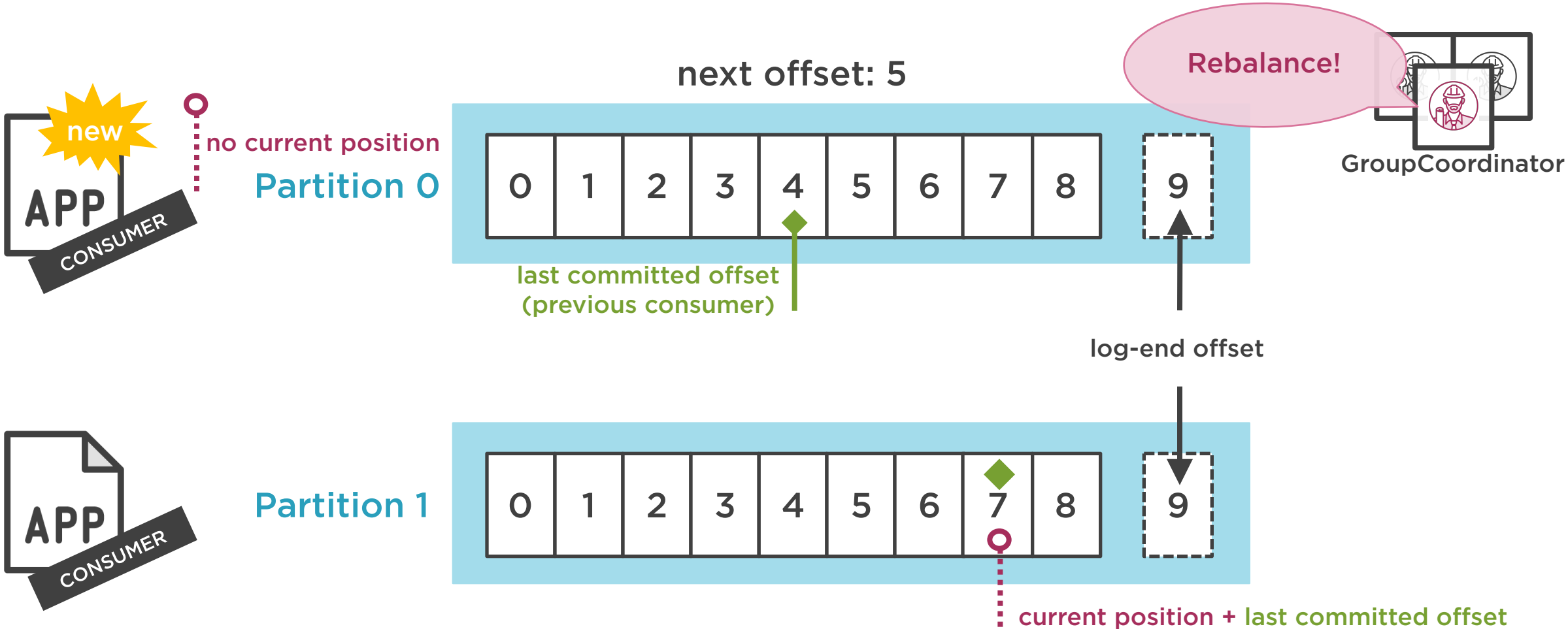
Sharing the message consumption and processing load

- Parallelism and throughput
- Redundancy
- Performance

Consumer Groups



Consumer Group Rebalancing



Group Coordinator

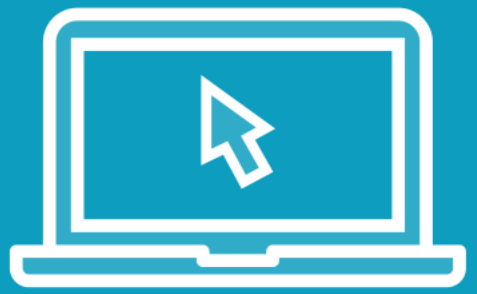
Evenly balances available Consumers to partitions

- 1:1 Consumer-to-partition ratio
- Can't avoid over-provisioning

Initiates the rebalancing protocol

- Topic changes (partition added)
- Consumer failure

Demo



Consumer Group comprising of Java-based Consumer applications

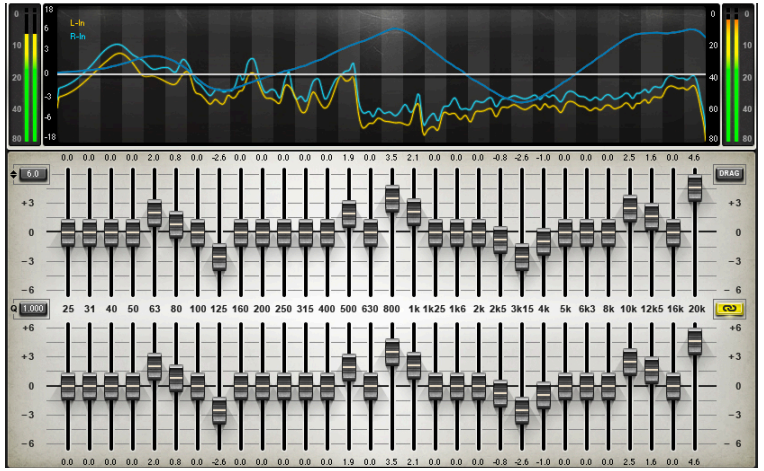
Setup:

- Three Consumers with same group id
- Consuming a single topic with three partitions

Look for:

- Shared topic consumption
- Adding an additional Consumer
- Adding an additional topic
- Forcing a rebalance

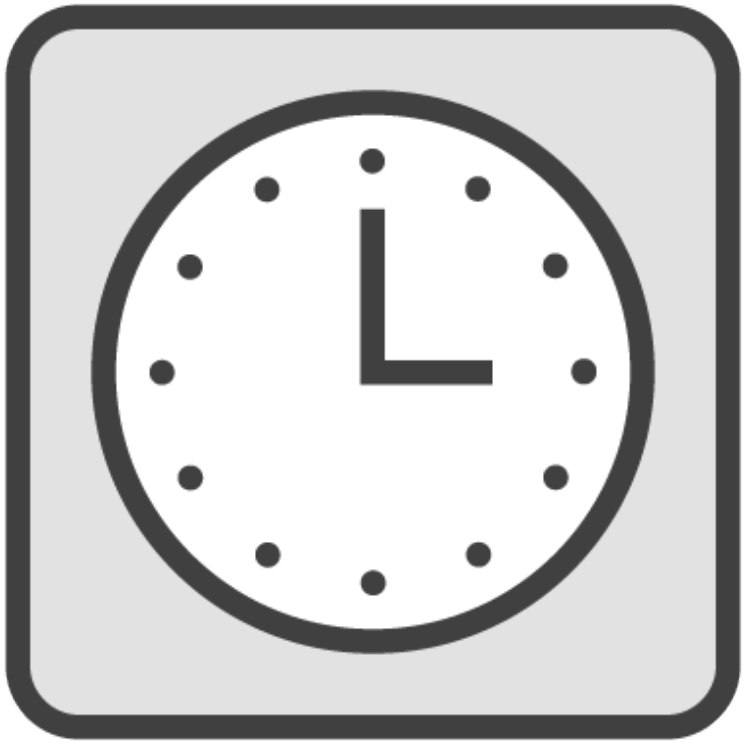
Consumer Configuration



Consumer performance and efficiency

- fetch.min.bytes
- max.fetch.wait.ms
- max.partition.fetch.bytes
- max.poll.records

Advanced Topics Not Covered



Consumer position control

- seek()
- seekToBeginning()
- seekToEnd()

Flow control

- pause()
- resume()

Rebalance Listeners

Summary

Kafka Consumer Internals

- Properties -> ConsumerConfig
- Message -> ConsumerRecord
- Subscriptions and assigments
- Message polling and consumption
- Offset management

Consumer Groups

Consumer Configuration

Java-based Consumer