Apache Kafka

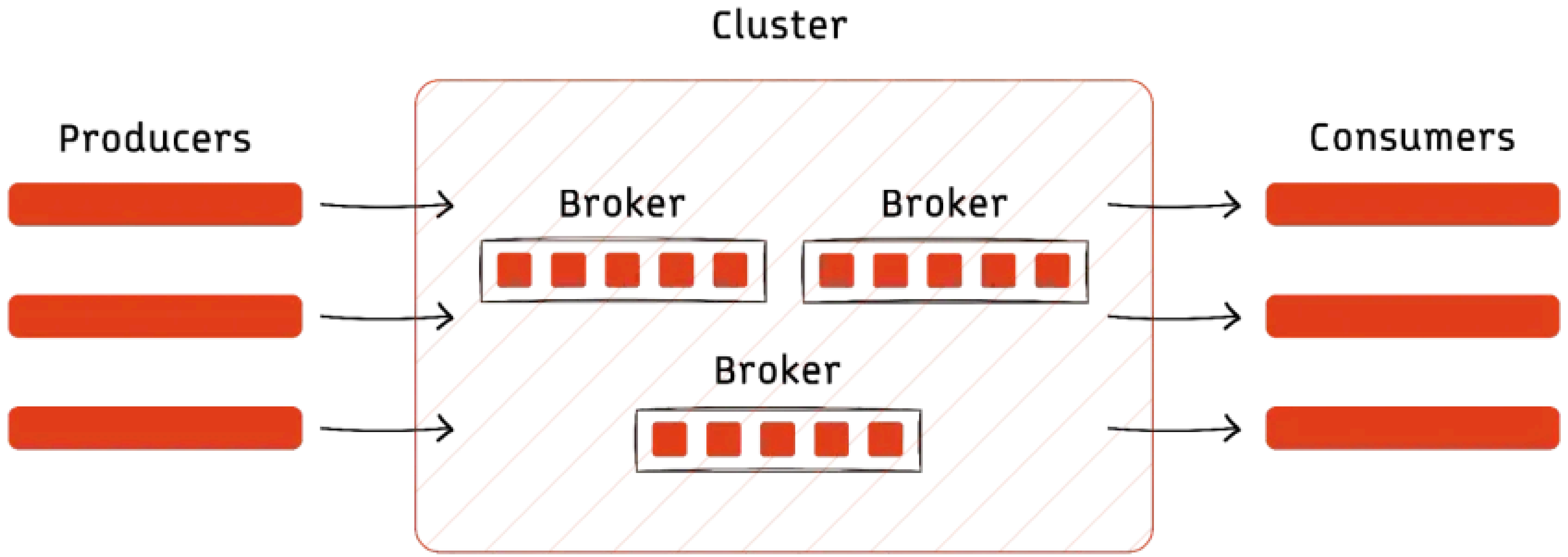
Rohan Joshua 22011102079
Sidheshwar S 22011102102

What is event streaming?

Event streaming is the process of continuously capturing, storing, processing, and reacting to real-time events as they happen.

Event Stream Processing (ESP) takes a continuous stream of events and processes them as single points of data rather than as a batch as soon as a change happens.

In software, any significant action can be recorded as an event. For example, it could be as simple as someone clicking a link or viewing a webpage, or something more involved like paying for an order, withdrawing money, or even communicating with numerous, distributed IoT devices at once.
These events can be organized into streams, essentially a series of events ordered by time. From there, events can be shared with other systems where they can be processed in real-time. Events are pushed and handled one at a time, as they happen. This allows the system to react in real-time, rather than waiting for batches to accumulate.


Cluster
Producers
Consumers
Broker
Broker
Broker

What is Apache Kafka

Apache Kafka® is an event streaming platform. What does that mean?
Kafka combines three key capabilities so you can implement your use cases for event streaming end-to-end with a single battle-tested solution:

1. **To publish (write) and subscribe to (read) streams of events, including continuous import/export of your data from other systems.**
2. **To store streams of events durably and reliably for as long as you want.**
3. **To process streams of events as they occur or retrospectively.**

And all this functionality is provided in a distributed, highly scalable, elastic, fault-tolerant, and secure manner. Kafka can be deployed on bare-metal hardware, virtual machines, and containers, and on-premises as well as in the cloud. You can choose between self-managing your Kafka environments and using fully managed services offered by a variety of vendors.

Architecture

Apache Kafka follows a distributed, publish-subscribe architecture with multiple components working together.

Key Components of Kafka Architecture:

1. **Producers:**
   - Applications that publish (send) messages to Kafka topics.
   - Messages are written to specific partitions within a topic.
2. **Topics & Partitions:**
   - A topic is a logical channel where messages are stored.
   - Topics are divided into multiple partitions for parallelism and scalability.
3. **Brokers:**
   - Kafka runs on brokers (servers) that manage the storage and retrieval of messages.
   - A Kafka cluster consists of multiple brokers.
4. **Consumers & Consumer Groups:**
   - Consumers read messages from Kafka topics.
   - Consumers are organized into consumer groups, ensuring load balancing.
5. **Zookeeper**:
   - Manages Kafka cluster metadata, leader election, and configurations.
   - Keeps track of broker health and partitions.

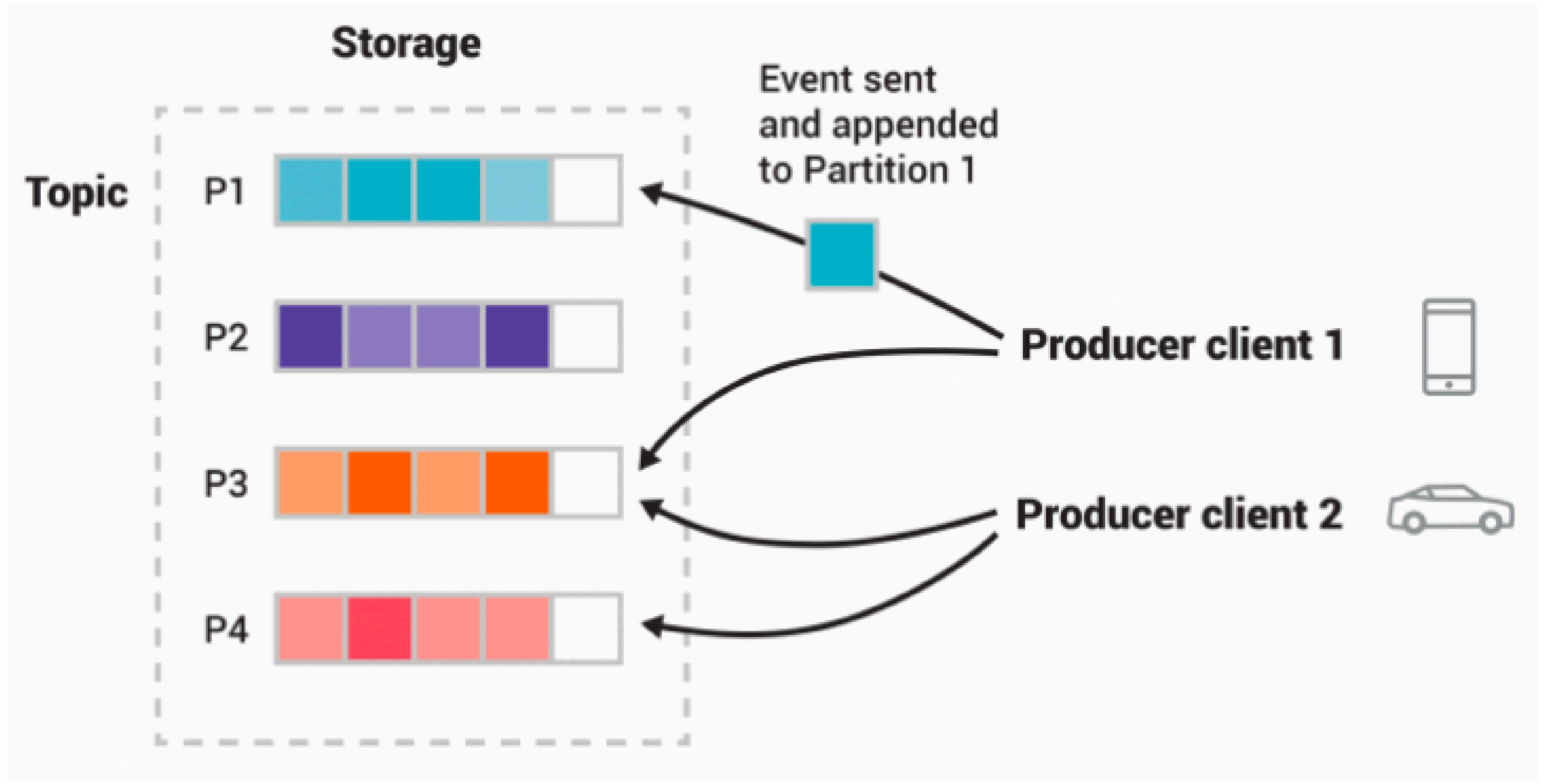
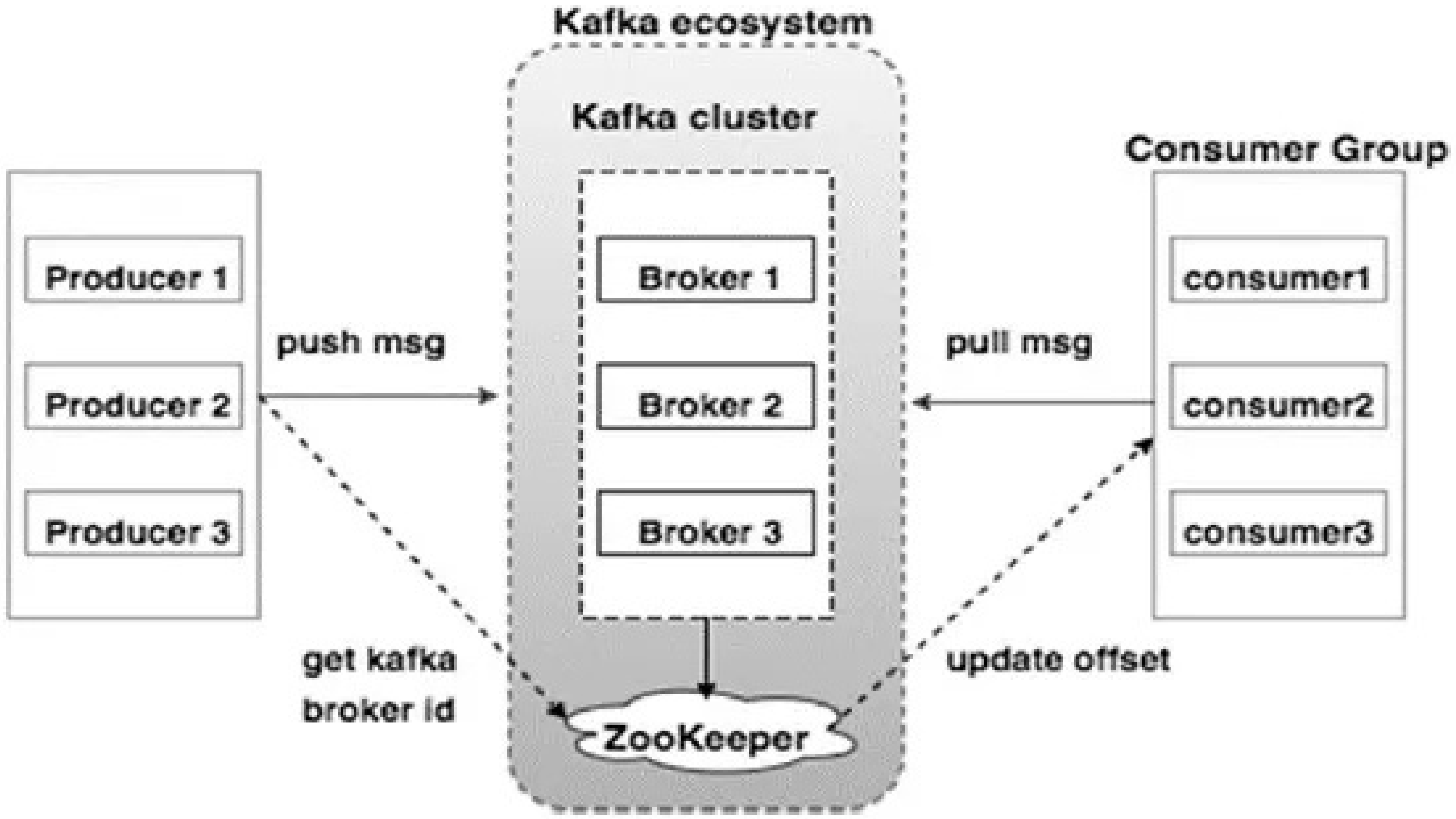
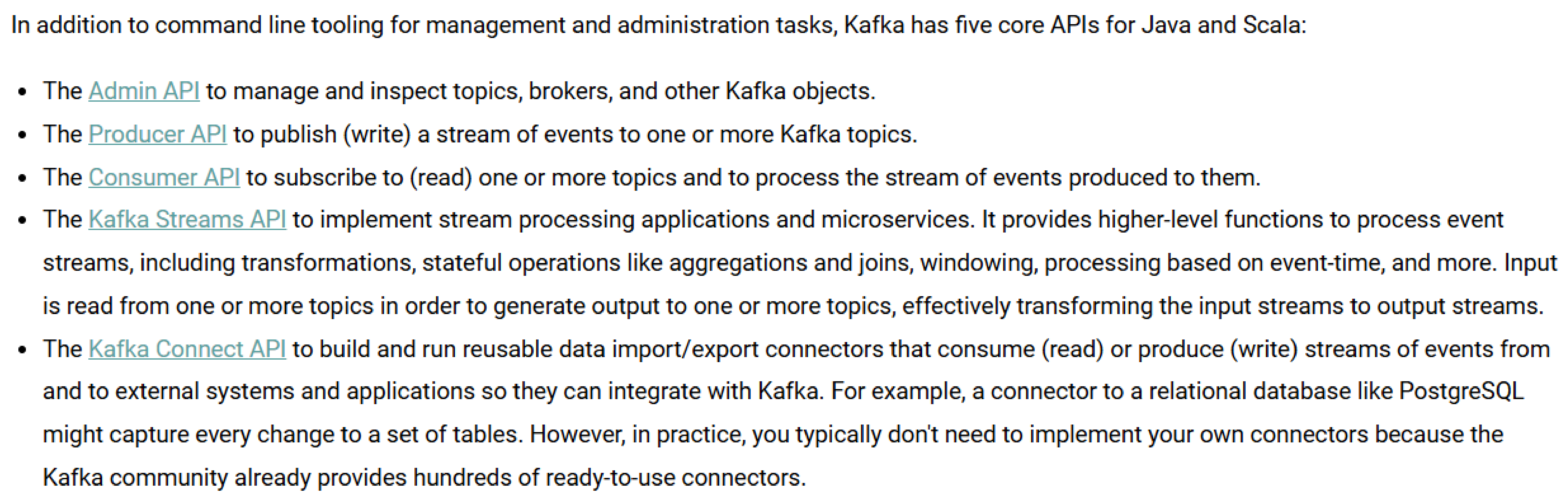


Figure: This example topic has four partitions P1–P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate.


Kafka ecosystem
Kafka cluster
Consumer Group
Producer 1
Producer 2
Producer 3
Broker 1
Broker 2
Broker 3
consumer1
consumer2
consumer3
push msg
pull msg
get kafka
broker id
update offset
ZooKeeper

Kafka APIs

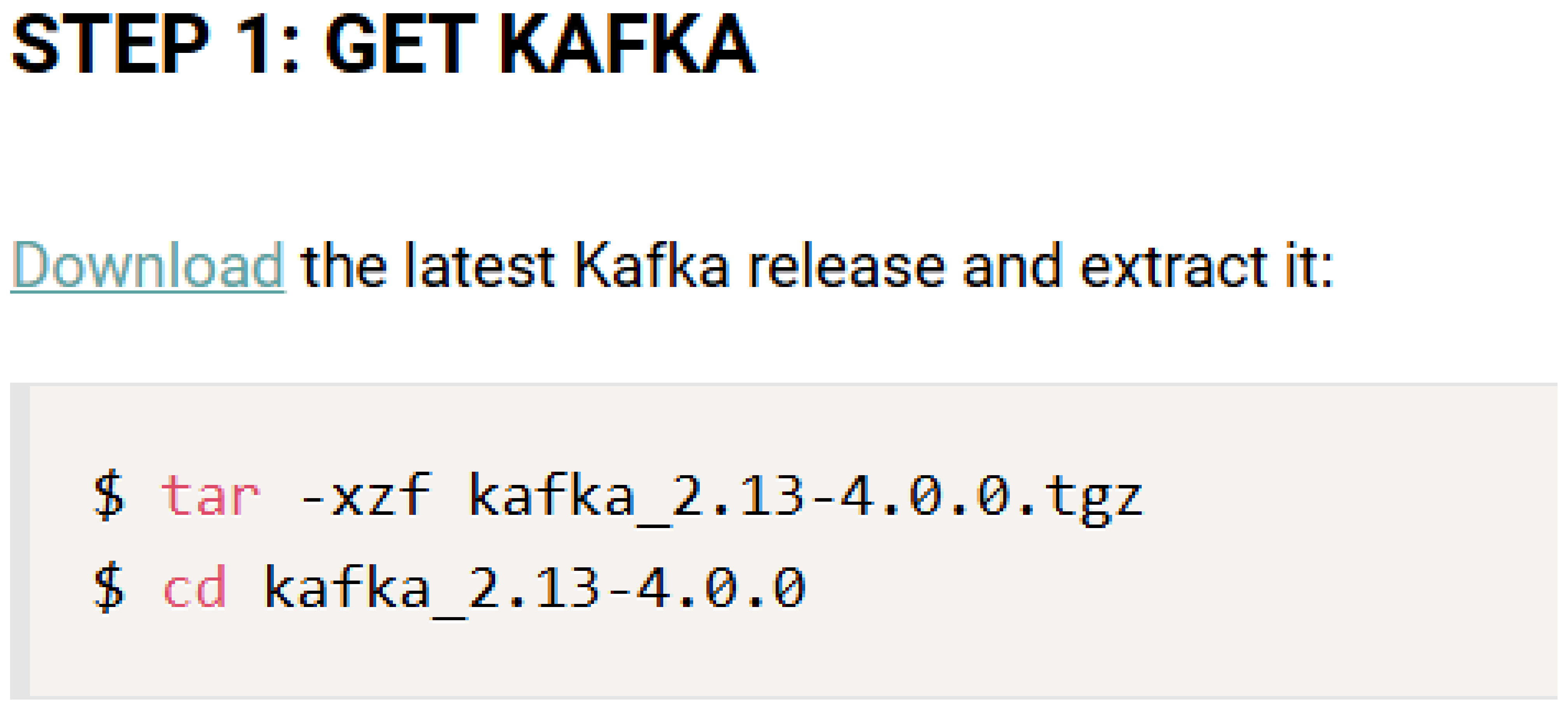


In addition to command line tooling for management and administration tasks, Kafka has five core APIs for Java and Scala:

- The Admin API to manage and inspect topics, brokers, and other Kafka objects.
- The Producer API to publish (write) a stream of events to one or more Kafka topics.
- The Consumer API to subscribe to (read) one or more topics and to process the stream of events produced to them.
- The Kafka Streams API to implement stream processing applications and microservices. It provides higher-level functions to process event streams, including transformations, stateful operations like aggregations and joins, windowing, processing based on event-time, and more. Input is read from one or more topics in order to generate output to one or more topics, effectively transforming the input streams to output streams.
- The Kafka Connect API to build and run reusable data import/export connectors that consume (read) or produce (write) streams of events from and to external systems and applications so they can integrate with Kafka. For example, a connector to a relational database like PostgreSQL might capture every change to a set of tables. However, in practice, you typically don't need to implement your own connectors because the Kafka community already provides hundreds of ready-to-use connectors.

These APIs allow developers to build scalable, fault-tolerant, and event-driven architectures without relying on traditional messaging systems.

How to download and use kafka



STEP 1: GET KAFKA

Download the latest Kafka release and extract it:

```
$ tar -xzf kafka_2.13-4.0.0.tgz
$ cd kafka_2.13-4.0.0
```

STEP 2: START THE KAFKA ENVIRONMENT

NOTE: Your local environment must have Java 17+ installed.

Kafka can be run using local scripts and downloaded files or the docker image.

Using downloaded files

Generate a Cluster UUID

```
$ KAFKA_CLUSTER_ID="$(bin/kafka-storage.sh random-uuid)"
```

Format Log Directories

```
$ bin/kafka-storage.sh format --standalone -t $KAFKA_CLUSTER_ID -c config/server.properties
```

Start the Kafka Server

```
$ bin/kafka-server-start.sh config/server.properties
```

Once the Kafka server has successfully launched, you will have a basic Kafka environment running and ready to use.

Using JVM Based Apache Kafka Docker Image

Get the Docker image:

```
$ docker pull apache/kafka:4.0.0
```

Start the Kafka Docker container:

```
$ docker run -p 9092:9092 apache/kafka:4.0.0
```

STEP 3: CREATE A TOPIC TO STORE YOUR EVENTS

Kafka is a distributed *event streaming platform* that lets you read, write, store, and process *events* (also called *records* or *messages* in the documentation) across many machines.

Example events are payment transactions, geolocation updates from mobile phones, shipping orders, sensor measurements from IoT devices or medical equipment, and much more. These events are organized and stored in *topics*. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder.

So before you can write your first events, you must create a topic. Open another terminal session and run:

```
$ bin/kafka-topics.sh --create --topic quickstart-events --bootstrap-server localhost:9092
```

STEP 4: WRITE SOME EVENTS INTO THE TOPIC

A Kafka client communicates with the Kafka brokers via the network for writing (or reading) events. Once received, the brokers will store the events in a durable and fault-tolerant manner for as long as you need—even forever.

Run the console producer client to write a few events into your topic. By default, each line you enter will result in a separate event being written to the topic.

```
$ bin/kafka-console-producer.sh --topic quickstart-events --bootstrap-server localhost:9092
>This is my first event
>This is my second event
```

STEP 5: READ THE EVENTS

Open another terminal session and run the console consumer client to read the events you just created:

```
$ bin/kafka-console-consumer.sh --topic quickstart-events --from-beginning --bootstrap-serv
This is my first event
This is my second event
```

You can stop the consumer client with `Ctrl-C` at any time.

Once your data is stored in Kafka as events, you can process the data with the Kafka Streams client library for Java/Scala. It allows you to implement mission-critical real-time applications and microservices, where the input and/or output data is stored in Kafka topics. Kafka Streams combines the simplicity of writing and deploying standard Java and Scala applications on the client side with the benefits of Kafka's server-side cluster technology to make these applications highly scalable, elastic, fault-tolerant, and distributed. The library supports exactly-once processing, stateful operations and aggregations, windowing, joins, processing based on event-time, and much more.

To give you a first taste, here's how one would implement the popular `WordCount` algorithm:

```java
KStream<String, String> textLines = builder.stream("quickstart-events");

KTable<String, Long> wordCounts = textLines
            .flatMapValues(line -> Arrays.asList(line.toLowerCase().split(" ")))
            .groupBy((keyIgnored, word) -> word)
            .count();

wordCounts.toStream().to("output-topic", Produced.with(Serdes.String(), Serdes.Long())
```

STEP 8: TERMINATE THE KAFKA ENVIRONMENT

Now that you reached the end of the quickstart, feel free to tear down the Kafka environment—or continue playing around.

1. Stop the producer and consumer clients with `Ctrl-C`, if you haven't done so already.
2. Stop the Kafka broker with `Ctrl-C`.

If you also want to delete any data of your local Kafka environment including any events you have created along the way, run the command:

```
$ rm -rf /tmp/kafka-logs /tmp/kraft-combined-logs
```