**What is Kafka? How it works? What are the main advantages/disadvantages/benefits of using Kafka?**

Apache Kafkais an event streaming platform. Kafka combines three key capabilities 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 data from other systems.
2. To **store** streams of events durably and reliably for as long as it’s needed.
3. To **process** streams of events as they occur or retrospectively.

This functionality is provided in a distributed, 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. It could provide self-managing Kafka environments and use fully managed services offered by a variety of vendors.

Hot it works?

Kafka is a distributed system consisting of **servers** and **clients** that communicate via a high-performance TCP network protocol. It can be deployed on bare-metal hardware, virtual machines, and containers in on-premise as well as cloud environments.

**Servers**: Kafka runs as a cluster of one or more servers that can span multiple data centers or cloud regions. Some of these servers are from the storage layer, called the brokers. Other servers run Kafka Connect to continuously import and export data as event streams to integrate Kafka with your existing systems such as relational databases as well as other Kafka clusters.  
**Clients**: They allow you to write distributed applications and microservices that read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner even in the case of network problems or machine failures.

An **event** records the fact that "something happened" in the world or in your business. An event has a key, value, timestamp, and optional metadata headers.

**Producers** are those client applications that publish (write) events to Kafka, and **consumers** are those that subscribe to (read and process) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. Events are organized and durably stored in **topics.**

Advantages/disadvantages/benefits?

Advantages:

1. **Low Latency:** Apache Kafka offers low latency value, i.e., upto 10 milliseconds. It is because it decouples the message which lets the consumer consume that message anytime.
2. **High Throughput:** Due to low latency, Kafka is able to handle more messages of high volume and high velocity. Kafka can support thousands of messages in a second. Many companies such as Uber use Kafka to load a high volume of data.
3. **Fault tolerance:** Kafka has an essential feature to provide resistance to node/machine failure within the cluster.
4. **Durability:** Kafka offers the replication feature, which makes data or messages to persist more on the cluster over a disk. This makes it durable.
5. **Reduces the need for multiple integrations:** All the data that a producer writes go through Kafka. Therefore, we just need to create one integration with Kafka, which automatically integrates us with each producing and consuming system.
6. **Easily accessible:** As all our data gets stored in Kafka, it becomes easily accessible to anyone.
7. **Distributed System:** Apache Kafka contains a distributed architecture which makes it scalable. Partitioning and replication are the two capabilities under the distributed system.
8. **Real-Time handling:** Apache Kafka is able to handle real-time data pipeline. Building a real-time data pipeline includes processors, analytics, storage, etc.
9. **Batch approach:** Kafka uses batch-like use cases. It can also work like an ETL tool because of its data persistence capability.
10. **Scalability:** The quality of Kafka to handle large amounts of messages simultaneously make it a scalable software product.

Disadvantages:

1. **Do not have a complete set of monitoring tools:** Apache Kafka does not contain a complete set of monitoring as well as managing tools. Thus, new startups or enterprises fear to work with Kafka.
2. **Message tweaking issues:** The Kafka broker uses system calls to deliver messages to the consumer. In case the message needs some tweaking, the performance of Kafka gets significantly reduced. So, it works well if the message does not need to change.
3. **Do not support wildcard topic selection:** Apache Kafka does not support wildcard topic selection. Instead, it matches only the exact topic name. It is because selecting wildcard topics makes it incapable to address certain use cases.
4. **Reduces Performance:** Brokers and consumers reduce the performance of Kafka by compressing and decompressing the data flow. This not only affects its performance but also affects its throughput.
5. **Clumsy Behaviour:** Apache Kafka most often behaves a bit clumsy when the number of queues increases in the Kafka Cluster.
6. **Lack some message paradigms:** Certain message paradigms such as point-to-point queues, request/reply, etc. are missing in Kafka for some use cases.

Benefits:  
**Scalability:** By dividing a topic into multiple partitions, Apache Kafka provides load balancing over a pool of servers. This allows you to scale production clusters up or down to fit your needs and to spread clusters across geographic regions or availability zones.

**Speed:** By decoupling data streams, Apache Kafka is able to deliver messages at network limited throughput using a cluster of servers with extremely low latency (as low as 2ms).

**Durability:** Apache Kafka makes the data highly fault-tolerant and durable in two main ways. First, it protects against server failure by distributing storage of data streams in a fault-tolerant cluster. Second, it provides intra-cluster replication because it persists the messages to disk.

I haven't used kafka yet but in my last company we used RabbitMQ for a message broker.

**Write down a testing strategy for an application that uses kafka, describe how the testing will be organized and provide 1 test case as an example?**First of all, we need to know where the **topic** resides and what types of **messages** (records) are **produced** to the topic, then what happens when the messages are **consumed** by the listeners.

### **Know the record format**

A record can be of various formats, e.g. RAW, JSON, CSV, AVRO, etc., and many others.

Let’s say our record is in JSON format who has Key - Value pair:

{

"key": "4321",

"value": "Star Wars movie"

}

Then we can start with:

### **Producer testing**

Once we know our Record format, we can write one or more records (our key-value pair) to a topic. Assert that we can successfully produce our record.

{

"key": "4321",

"value": "Star Wars movie"

}

### **Consumer testing**

Since our message has been produced, it is ready to be fetched (consumed) from our consumers (listeners).

When we are connected to the respective topic, we can assert that the same response is present in the fetched record. The result should looks like:

{

"key": "4321",

"value": "Star Wars movie"

}

## **Other things we can test in applications using Kafka:**

A must is to assert that the record could be successfully produced to a topic.  
A must is to assert that the consumed record matches the produced record.

Also, we can dive in further and assert at granular levels, for instance :

1. Validate the type of record we are able to produce or consume.
2. Validate whether we have produced the record to a particular partition.
3. Check the number of records written to a topic or fetched from a topic.
4. Validate record’s offset.
5. Assert the outcome of sending and receiving AVRO or JSON records.
6. Schema Registry for AVRO and validate records.
7. Check whether DLQs (Dead Letter Queue) exist. Test for message sent to missing queue; message with exceeded length limit; message rejected by other queue; message which is not processed successfully.

| **Test case** | | OpenTag\_ID\_01 | | | **Description:** | Verify that a record can be produced successfully | | **Positive** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Required successfully executed test cases / Prerequisites:** | | | Kafka service is up and running | | | | | |
| **Priority:** | | | High | | | | | |
| **#** | **Step** | | | **Input** | | | **Expected result** | |
| 1 | Navigate Kafka dashboard | | |  | | | Kafka’s dashboard is present. | |
| 2 | Choose a topic | | | **Select topic “X”** | | | The page is reloaded and all prices are in GBP currency. | |
| 3 | Write a new record | | | **Body:** {  "key": "4321",  "value": "Star Wars movie"  } | | | The data is populated in the message field. | |
| 4 | Produce the newly added record | | | **Submit the message** | | | Success message is returned. No errors are shown. | |

**What is microservice architecture? Please write down your approach for testing microservices and what will be the scope of your testing.**

Microservices support a variety of different clients including desktop browsers, mobile browsers and native mobile applications.

* Microservices are small, independent, and loosely coupled. A single small team of developers can write and maintain a service.
* Each service is a separate codebase, which can be managed by a small development team.
* Services can be deployed independently. A team can update an existing service without rebuilding and redeploying the entire application.
* Services are responsible for persisting their own data or external state. This differs from the traditional model, where a separate data layer handles data persistence.
* Services communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services.
* Supports polyglot programming. For example, services don't need to share the same technology stack, libraries, or frameworks.

The best approach to test microservices is to follow the bottom-up levels of the Test pyramid and cover as many levels as possible.

**Unit testing**   
Cover the smallest testable code in the application to determine whether it works as expected.

**Integration test**  
Verify the communication paths and interactions between the modules and between their dependencies.

**Component testing**  
Test the end-to-end functionality of the given microservice in isolation by replacing its dependencies by test doubles and/or mock services.

**Contract testing**  
Test at the boundary of an external service verifying that it meets the contract expected by consuming service.

**E2e testing/UI tests**

Writing and maintaining end-to-end tests can be very difficult, write as few end-to-end tests as possible.

**Exploratory testing**

Discover risks and issues during the software development process.

**What will be your approach to ensure that tests can be re-run in future?**

First of all we should have the test cases in written format.  
Once we have the big picture of what we need to cover with tests, we need to come up with a Plan of what makes sense to be automated (UI and API), what we can cover with tools (such as Postman, SOAP UI, etc.), and what will be covered with Unit tests.

When we identify and automate our tests, we need to make them run repeatedly using CI tools like GitLab, Jenkins, CircleCI, GitHub Actions, etc. There we can config the tests to run upon application build, predefined schedule or upon manual run.   
Following the Continuous Integration (CI) development practice the tests could be attached to the build pipeline of the application. In this way each piece of code runs the automated tests.

In the Agile world, Continuous Delivery (CD) allows commitment of new pieces of functionality when they are ready, when the change passes the tests.