[**https://spring.io/projects/spring-kafka**](https://spring.io/projects/spring-kafka)

**KAFKA**

# **What is Kafka?**

Kafka is like a **messaging system**

* Kafka is a distributed publish-subscribe messaging system.
* That allows producers to publish data to topics and consumers to subscribe and process that data in real-time.

# **How Kafka Works:**

* A **producer** sends a message to a **Kafka topic**.
* Kafka stores the message in a **partition** inside that topic.
* One or more **consumers** subscribe to the topic to read and process messages.
* Kafka ensures **high performance**, **reliability**, and **fault tolerance** throughout the process.

# **Why Use Apache Kafka?**

Apache Kafka is widely used because it solves the problem of **real-time data exchange and processing between systems** in a fast, reliable, and scalable way.

* Kafka can handle **millions of messages per second**.
* Suitable for **high-volume, real-time data pipelines** (e.g., logs, clicks, transactions).

**📘 What is a Topic in Kafka?**

* Topics are like **message queues** but allow **multiple consumers** to read the same data.
* Each topic can be **divided into partitions** for scalability and parallelism.
* Messages inside a topic are **ordered within each partition**.
* Producers write data **to a topic**, and consumers **subscribe to it**.

**📘 What is a Producer in Kafka?**

* In Apache Kafka, a **producer** is a **client application** that **sends (publishes) messages** to a **Kafka topic**. Producers are the **source of data** in the Kafka system.

📘 **What is a Consumer in Kafka?**

* A **Kafka consumer** is a component that **subscribes to topics**, **reads messages**, and processes them. It keeps track of which messages it has read using **offsets**.

**📘 What is Zookeeper in Kafka? (Point Format)**

* Zookeeper is used by Kafka to **manage brokers, coordinate the cluster, store metadata**, and ensure the **reliability and stability** of the Kafka system.

📘 **What is a Broker in Kafka?**

* A **Kafka broker** is a server that receives messages from producers, stores them on disk, and serves them to consumers when requested.

**📘 What is a Cluster in Kafka?**

* In Apache Kafka, a **cluster** is a **group of Kafka brokers (servers)** working together to manage and process data in a distributed, scalable, and fault-tolerant manner.

**📘 What is a Partition in Kafka?**

* A **partition** is an ordered, immutable sequence of messages in a Kafka topic that is continually appended to — and identified by a unique number (like partition-0, partition-1, etc.).

**📘 What is an Offset in Kafka?**

* In Apache Kafka, an **offset** is a **unique identifier** assigned to each message within a **partition**. It represents the **position** of a message in the partition.

# **With Kafka Without Kafka**

* API-1 calls API-2 directly via REST. Some issues API-2 is down then API-1 request fails.
* API-1 needs to wait for API-2 to process the request.
* Hard to add a new service (e.g., API-3) without changing API-1.

**With Kafka:**

* API-1 publishes a message to Kafka (like dropping a letter in a mailbox).
* API-2 reads messages from Kafka whenever it's ready (like picking up mail).
* If API-2 is down, Kafka stores the message until API-2 is up (no data loss).
* We can add API-3 or any number of consumers without changing API-1 (loose coupling).

# **What is Kafka cluster**

* **group of Kafka brokers (servers)** there inside a single Kafka cluster. There can be one or more brokers in the Kafka cluster.
* if your producer is producing huge volume of data then single Kafka broker unable to handle the load.so you might need to add additional Kafka brokers.

**NOTE:** To add a Kafka broker, **install Kafka** on a **new server**, set a **unique broker.id in server. Properties,** point it to the same **Zookeeper.**

# **What is Topic**

**Example:** One topic have we stored multiple messages Like, Order related message, booking related message, payment related messages. so, Consume wants to subscribe order related message but their multiple type of messages. There is some confusion to avoid this will use

* Payment related message it will store payment topic.
* booking related message it will store booking topic.
* Order related message it will store order topic.
* if whatever message consumer service wants then simply it will subscribe specific topic and get all the message want your need. like that it will work.

**NOTE:** Instead of storing different message types (order, booking, payment) in a single topic, use separate topics for each type. This allows consumers to subscribe only to the specific topic they need, avoiding confusion and ensuring clean message processing.

# **What are Partitions**

* If producer is publishing huge volume of messages to topic. then topic not handling there could be a storage challenges.
* To overcome this will use partitions, we need break the topic into multiple parts and distribute those parts into different machines. it is called topic partitions.
* if we are split our topic in multiple partitions, when producer send bulk messages then partitions it will accept those messages. that will improve the performance.
* If in case any partitions is goes down remain partitions are available and handle the loads.
* these partitions will give better performance, better availability.

**Note:** If a producer sends a large number of messages to a single topic, it can lead to storage and performance issues. To solve this, Kafka splits the topic into **partitions**, which are spread across different machines. If one partition fails, others still work — ensuring availability.

# **What is Offset**

* If Kafka cluster have one or more Kafka server, then Kafka server has multiple topics, each topics have multiple partitions.
* When producer send a message then it will go and store any partition inside a topic.
* Like it will work round Robbin principle, it will store sequence of numbers, assign each and every message’s that is called offset.

**Example:**

An **offset** is like a bookmark for a Kafka consumer.  
If a partition has 5 messages and the consumer reads 4 message, then consumer goes down, the offset helps consumer wants to **start from** 5 messages onwards t.

👉 If the offset was saved after reading message 4, it will read 5 messages onwards.  
👉 If not saved, it might re-read message 4.

# **What is Consumer Group**

* Consumer groups sharing the workload. In this case, we are defining the number of consumers. There are three partitions.
* We can divide workload in each consumer. All partitions will read each consumer and maintain any order.
* If one consumer down, then another consumer connected with the partitions read the messages.

# **Kafka Integration in Spring Boot**

## **1. Add Dependencies (pom.xml)**

|  |
| --- |
| <dependencies>  <!-- Spring Boot Starter for Kafka -->  <dependency>  <groupId>org.springframework.kafka</groupId>  <artifactId>spring-kafka</artifactId>  </dependency>  <!-- Optional: Spring Web for REST -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency> </dependencies> |

## **2. Configure Kafka (application.yml)**

|  |
| --- |
| spring:  kafka:  bootstrap-servers: localhost:9092  consumer:  group-id: my-group  auto-offset-reset: earliest  key-deserializer: org.apache.kafka.common.serialization.StringDeserializer  value-deserializer: org.apache.kafka.common.serialization.StringDeserializer  producer:  key-serializer: org.apache.kafka.common.serialization.StringSerializer  value-serializer: org.apache.kafka.common.serialization.StringSerializer |

## **3. Create Kafka Topic**

|  |
| --- |
| @Configuration public class KafkaTopicConfig {  @Bean  public NewTopic myTopic() {  return TopicBuilder.name("my-topic")  .partitions(1)  .replicas(1)  .build();  } } |

**4. Kafka Producer**

|  |
| --- |
| @Service public class KafkaProducer {   @Autowired  private KafkaTemplate<String, String> kafkaTemplate;   public void sendMessage(String message) {  kafkaTemplate.send("my-topic", message);  } } |

## **5. Kafka Consumer**

|  |
| --- |
| @Service public class KafkaConsumer {   @KafkaListener(topics = "my-topic", groupId = "my-group")  public void consume(String message) {  System.out.println("Consumed message: " + message);  } } |

## **6.REST Controller**

|  |
| --- |
| @RestController @RequestMapping("/kafka") public class KafkaController {   @Autowired  private KafkaProducer producer;   @PostMapping("/publish")  public ResponseEntity<String> publish(@RequestParam String message) {  producer.sendMessage(message);  return ResponseEntity.ok("Message sent to Kafka topic");  } } |

## **7. Docker Compose (Optional)**

|  |
| --- |
| version: '2' services:  zookeeper:  image: wurstmeister/zookeeper  ports:  - "2181:2181"   kafka:  image: wurstmeister/kafka  ports:  - "9092:9092"  environment:  KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181  KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://localhost:9092  KAFKA\_LISTENERS: PLAINTEXT://0.0.0.0:9092 |

# **Can we run Kafka without Zookeeper?**

Yes, it is possible after Kafka 2.8 version onwards.

* Kafka uses Zookeeper to store and manage all the metadata information of the Kafka cluster.

And also manages all the Kafka brokers.

* In newer Kafka versions, **topic partition metadata** can be stored directly in the Kafka server instead of ZooKeeper.
* No need to keep all Kafka cluster metadata in ZooKeeper.
* Kafka can run **without ZooKeeper** using **KRaft mode** (Kafka Raft metadata mode).
* In KRaft mode, metadata is stored in Kafka itself, enabling a **Zookeeper-free setup**.

# **what is use of Apache Kafka JSON Serialization & Deserialization Example.**

Apache Kafka JSON serialization and deserialization enables sending and receiving structured data (objects), readable data between producers and consumers.

**1️⃣ Producer Side – JSON Serialization**

* Java object → JSON string → Sent to Kafka topic.Uses JsonSerializer (from spring-kafka or jackson-databind).

|  |  |
| --- | --- |
| * @Bean * public ProducerFactory<String, Employee> producerFactory() { * Map<String, Object> configProps = new HashMap<>(); * configProps.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092"); * configProps.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class); * configProps.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, JsonSerializer.class); * return new DefaultKafkaProducerFactory<>(configProps); * }  |  | | --- | |  | |

**2️⃣ Consumer Side – JSON Deserialization**

* Kafka JSON string → Java object. Uses JsonDeserializer and the target class type.

|  |
| --- |
| * @Bean * public ConsumerFactory<String, Employee> consumerFactory() { * Map<String, Object> configProps = new HashMap<>(); * configProps.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092"); * configProps.put(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class); * configProps.put(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, JsonDeserializer.class); * return new DefaultKafkaConsumerFactory<>( configProps,new StringDeserializer(), * new JsonDeserializer<>(Employee.class) * ); * } |

# **Consider a scenario where you are processing a financial transaction. If a transaction fails to be processed due to some temporary issue, then how will you handle it?**

**Retry Mechanism:**

* If a consumer fails to process a message, Kafka can **retry** it.
* Retry count is configurable (e.g., 4 means **3 retries** — N-1 rule: first is the original attempt, rest are retrying).
* Retries happen **sequentially** until:
  + ✅ Message is successfully processed, or
  + ❌ Retry limit is reached.

**Dead Letter Topic (DLT):**

* If a message fails after all retries, it’s sent to a **Dead Letter Topic** (special Kafka topic for failed events).
* Purpose:
  + Monitor failed events.
  + Debug root causes.
  + Reprocess messages after fixing issues.

**Benefits:**

* **No data loss** — messages are retained in DLT.
* **Traceability** — failed messages and reasons are known.
* **Recovery** — DLT messages can be reprocessed anytime.

**Note:**

* If a consumer fails to process a message, Kafka can **retry** it. reattempt failed messages.
* **N-1 rule**: If retry count = 4, Kafka retries 3 more times.
* After retries fail, message is sent to a **Dead Letter Topic (DLT)**.
* DLT stores failed events for later **investigation** and **reprocessing**.
* Ensures **no data loss** and provides **visibility** into failures.
* **@RetryableTopic:** This annotation will tell to the Kafka how many times attempts you can Perform. Added in Spring Kafka 2.7+.

**@RetryableTopic** (attempts = "4", // total attempts = 1 original + 3 retries

backoff = @Backoff (delay = 2000, multiplier = 2.0), // exponential delay

dltTopicSuffix = "-dlt" // name suffix for Dead Letter Topic

**NOTE:** If attempts is 4, then it will be attempting 3 times, and also it will be following n-1 order after 4 times attempts. Retry issue is not resolved. If retry is exceeded, it will be move to DLT

* **@DLThandler:** We can able to enable DLT logic for failure message based on the Kafka topic.

# **Simple Kafka Flow (Step-by-Step)**

1. **User sends a request**  
   👉 Example: Submitting a form or placing an order.
2. **Producer app sends the message to Kafka**  
   👉 The app creates a message and sends it to a Kafka **topic**.
3. **Kafka stores the message in a topic**  
   👉 The message is saved in **partitions** inside the topic.
4. **Consumer listens to the topic**  
   👉 Another app or service is waiting for new messages.
5. **Consumer picks up the message**  
   👉 It reads the message from Kafka automatically.
6. **Consumer processes the message**  
   👉 It might save it to a database or trigger another action.
7. **(Optional) Response is sent back to another topic**  
   👉 The consumer can publish a result to a **response topic**.
8. **Main app reads the response and replies to the user**  
   👉 Final response is sent back to the user (e.g., “Success”).

# **Summary of Flow**

1. User sends a request.
2. Application (producer) publishes a message to Kafka.
3. Kafka stores the message in a topic (split into partitions).
4. Consumer listens to the topic and processes the message.
5. (Optional) Consumer sends a response back via Kafka.
6. Main app returns final output to user.

Note: “In Kafka, each topic is divided into partitions, which store messages in order. Offsets are used to identify each message within a partition. If a key is used, Kafka guarantees ordering within that key by placing all related messages in the same partition. Without a key, messages are distributed randomly.”

# **Kafka Implementation.**

1.Add org.springframework.kafka dependency.

2.Configure Kafka in (application.yml).

**bootstrap-servers:** localhost:9092

**consumer:**

**group-id:** my-group

**auto-offset-reset:** earliest

**key-deserializer:** org.apache.kafka.common.serialization.StringDeserializer

**value-deserializer:** org.apache.kafka.common.serialization.StringDeserializer

**producer:**

**key-serializer:** org.apache.kafka.common.serialization.StringSerializer

**value-serializer:** org.apache.kafka.common.serialization.StringSerializer

# **Kafka Implementation in Spring Boot**

1. Add the **spring-Kafka** dependency to the pom.xml file.
2. **Configure Kafka Properties** like:

* bootstrap-servers
* key/value serializers
* consumer group ID

1. Create a **topic** and along with Partitions and replicas.

Return TopicBuilder.name("my-topic").partitions(1).replicas(1).build();

1. Create Produceras **Kafka Template** to send messages to a **Kafka topic**. And Create Consumer as **@KafkaListener** to listen and **consume messages** from a topic.
2. Create Rest controller class publish the message and run the application.
3. Spring Boot simplifies Kafka integration through auto-configuration, enabling quick setup for event-driven communication.

# **Kafka commands:**

**Create Zookeeper:**

* .\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties

**Create Kafka server:**

* .\bin\windows\kafka-server-start.bat .\config\server.properties

**Create topic:**

* .\bin\windows\kafka-topics.bat --create --topic my\_topic --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1

**Describe the topics**

* .\bin\windows\kafka-topics.bat --list --bootstrap-server localhost:9092

**Run Producer:**

* .\bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic my\_topic

**Run Consumer:**

* .\bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic my\_topic --from-beginning

# **How to run Kafka in spring boot?**

* **Start Kafka & Zookeeper (or KRaft mode in latest Kafka)**
* **Add Spring Kafka dependency** to your project (spring-kafka).
* **Configure Kafka properties** in application.yml (bootstrap server, consumer, producer settings).
* **Create a Producer service** using KafkaTemplate to send messages to a topic.
* **Create a Consumer service** using @KafkaListener to consume messages from the topic.
* **Expose a REST controller (optional)** to trigger sending messages from an API.
* **Run the Spring Boot application** and test by producing and consuming messages.

# How many Ports Required for Spring Boot + Kafka

**1. Spring Boot Application**

* Runs on **one port** (default **8080**) → only if you expose REST APIs or web endpoints.
* If you disable the web server (non-web app), Spring Boot may run with **no port** at all.

**2. Kafka Broker**

* **Client port** → default **9092** (used by Spring Boot app to connect as producer/consumer).
* If running a cluster, each broker needs its own port (e.g., 9092, 9093, 9094…).

**3. Zookeeper (only if using older Kafka < 3.3)**

* Default port **2181** (used internally by Kafka, not by Spring Boot).

**✅ Summary**

* **Spring Boot (web app)** → 8080
* **Kafka broker** → 9092
* **Zookeeper (if used)** → 2181

👉 So in total, for a simple setup:

* **With Zookeeper** → 3 ports (8080 + 9092 + 2181)
* **Without Zookeeper (KRaft mode)** → 2 ports (8080 + 9092)

# **✅ Which services are producers and consumers?**

**1. User Service – Producer:** Publishes UserCreated events when a new user registers.

* **Why:** Allows other services (Policy, Notification) to act without direct calls, reducing coupling.

**2.Policy Service – Consumer & Producer**

* **Consumer:** Listens to UserCreated events and creates default policies.
* **Producer:** Publishes PolicyCreated events for Claim, Payment, and Notification services.

**Why:**

* Enables asynchronous processing so services don’t block each other.
* Supports scalability by handling multiple policies concurrently.

**3. Claim Service & Notification Service – Consumers**

* **Claim Service:** Consumes PolicyCreated events to process claims for active policies.
* **Notification Service:** Consumes PolicyCreated and ClaimProcessed events to notify users about policies, claims, or payments.

**Why:**

* Decouples notification logic from other services.
* Sends real-time updates without blocking business processes.

**4.Payment Service – Consumer & Producer:**

* **Consumer:** Listens to PolicyCreated events to schedule payments.
* **Producer:** Publishes PaymentConfirmed7/ events after successful payments.
* **Why:** Keeps payment processing independent and informs other services like Notification or Accounting.

**🧠 1. What is Apache Kafka?**

**Apache Kafka** is a **distributed event streaming platform** used for:

* **High-throughput data pipelines**
* **Real-time analytics**
* **Event-driven microservices**

It allows systems to **publish**, **subscribe**, **store**, and **process** streams of records in real-time.

**Core idea:**  
Kafka decouples data producers from consumers — instead of directly sending messages, producers write to **topics**, and consumers read from them asynchronously.

**⚙️ 2. Kafka Architecture Overview**

Kafka consists of the following components:

| **Component** | **Description** |
| --- | --- |
| **Producer** | Application that **sends messages** (events) to Kafka topics. |
| **Consumer** | Application that **reads messages** from Kafka topics. |
| **Broker** | Kafka server that **stores** and **serves** messages. |
| **Cluster** | A group of Kafka brokers working together. |
| **Topic** | A **category or feed name** to which messages are published. |
| **Partition** | Topics are divided into **partitions** for scalability and parallelism. |
| **Offset** | A unique ID assigned to each message within a partition. |
| **Consumer Group** | A set of consumers working together to consume a topic. |
| **Zookeeper / KRaft** | Manages cluster metadata (Zookeeper is being replaced by **KRaft** mode). |

**📦 3. Topics and Partitions**

* A **Topic** is like a table or stream name.
* Each topic is divided into **partitions**.
* Each **partition** is an **ordered, immutable log** of messages.
* Each message within a partition has a **unique offset**.

**Example:**

Topic: user-activity  
Partitions:

* Partition 0 → messages 0–100
* Partition 1 → messages 0–100
* Partition 2 → messages 0–100

Partitioning enables **scalability** and **parallel processing**.

**🏗️ 4. Producers**

* Send data to Kafka topics.
* Can specify **partition key** — helps decide which partition to send to.
* Use **acks** (acknowledgment levels):
  + acks=0: Fire and forget.
  + acks=1: Leader acknowledges write.
  + acks=all: Leader + replicas acknowledge (strongest guarantee).

**📥 5. Consumers**

* Subscribe to topics.
* Each consumer belongs to a **consumer group**.
* Kafka ensures:
  + Each partition is read by **only one consumer** in a group.
  + Different groups can consume the same topic **independently**.

**🧩 6. Consumer Groups**

* **Group ID** uniquely identifies a consumer group.
* Consumers in the same group share work.
* Kafka maintains **offsets per group**, enabling **fault tolerance**.

If one consumer fails, another consumer in the group takes over that partition.

**🔁 7. Offsets**

Each record in a partition has a **unique offset number**:

* Used to keep track of read position.
* Consumers can commit offsets manually or automatically.

**Manual commit** = More control (committing only after successful processing).  
**Auto-commit** = Simpler, but less reliable.

**🔄 8. Replication & Fault Tolerance**

Each partition has:

* **1 leader** → handles all reads/writes.
* **N followers** → replicate data from leader.

If leader fails → one follower becomes the new leader.

**⚡ 9. Message Delivery Semantics**

Kafka supports 3 delivery guarantees:

| **Type** | **Description** |
| --- | --- |
| **At most once** | Messages may be lost but never redelivered. |
| **At least once** | Messages are never lost, but can be redelivered. |
| **Exactly once** | Messages are neither lost nor duplicated. (Requires Kafka 0.11+ with idempotent producers and transactions.) |

**🧮 10. Log Retention and Compaction**

* Kafka **stores** messages for a configurable period (e.g., 7 days).
* Two policies:
  + **Retention policy**: Delete messages older than X days or larger than Y GB.
  + **Log compaction**: Keep only **latest message per key** (useful for changelog topics).

**🧰 11. Kafka Tools**

* **kafka-topics.sh** → Create, list, describe topics
* **kafka-console-producer.sh** → Send test messages
* **kafka-console-consumer.sh** → Read messages
* **kafka-consumer-groups.sh** → Manage consumer offsets/groups
* **kafka-configs.sh** → Alter configurations

**🚀 12. Kafka Streams API**

A **Java library** for real-time data processing using Kafka topics as both input and output.

Features:

* Filter, aggregate, join, and transform streams.
* Stateless and stateful processing.
* Windowed computations.

Example:

KStream<String, String> source = builder.stream("input-topic");

KStream<String, String> uppercased = source.mapValues(value -> value.toUpperCase());

uppercased.to("output-topic");

**🧮 13. Kafka Connect**

A **framework** to connect Kafka with external systems (databases, file systems, etc.)

* **Source connector**: Pulls data into Kafka.
* **Sink connector**: Pushes data out of Kafka.

Example:

* Source: JDBC → Kafka topic
* Sink: Kafka → Elasticsearch

**🧱 14. Kafka Schema Registry (Confluent)**

Manages **Avro/JSON/Protobuf schemas** for Kafka topics.  
Ensures producer and consumer use **compatible schemas** — prevents breaking changes.

**🧭 15. Kafka Security**

Main mechanisms:

* **SSL/TLS** → Encryption
* **SASL** → Authentication
* **ACLs** → Authorization (who can access what)
* **Encryption at rest** (optional)

**📊 16. Kafka Monitoring**

Tools for observing Kafka:

* **JMX Metrics**
* **Prometheus + Grafana**
* **Confluent Control Center**
* **Kafka Manager / Burrow**

Monitor:

* Lag per consumer group
* Broker health
* Under-replicated partitions
* Disk usage

**🧩 17. Kafka in Microservices**

Common pattern: **Event-Driven Architecture**

* Each microservice publishes domain events.
* Other services consume relevant events asynchronously.

Example:

Order Service → publishes → OrderCreatedEvent

Inventory Service → consumes → OrderCreatedEvent

**💡 18. Kafka Advanced Features**

| **Feature** | **Description** |
| --- | --- |
| **Idempotent Producer** | Ensures no duplicates even on retries. |
| **Transactions** | Atomic writes across multiple partitions. |
| **Exactly-Once Semantics** | Combines idempotence + transactions. |
| **Rebalance Protocol** | Handles consumer joins/leaves gracefully. |
| **KRaft Mode** | Removes Zookeeper; Kafka manages metadata internally. |

**🧱 19. Kafka Deployment Considerations**

Key configuration choices:

* Replication factor (≥ 3 recommended)
* Partitions per topic
* Retention period
* Acknowledgment level (acks)
* Compression (snappy, lz4, zstd)
* Proper hardware (SSD, network bandwidth)

**🧩 20. Kafka Real-World Use Cases**

| **Use Case** | **Example** |
| --- | --- |
| **Log aggregation** | Collect logs from multiple services. |
| **Event sourcing** | Keep full history of events in applications. |
| **Data pipelines** | Stream data between systems (DB → Analytics). |
| **Metrics tracking** | Monitor app behavior in real-time. |
| **Microservice communication** | Event-driven architecture. |