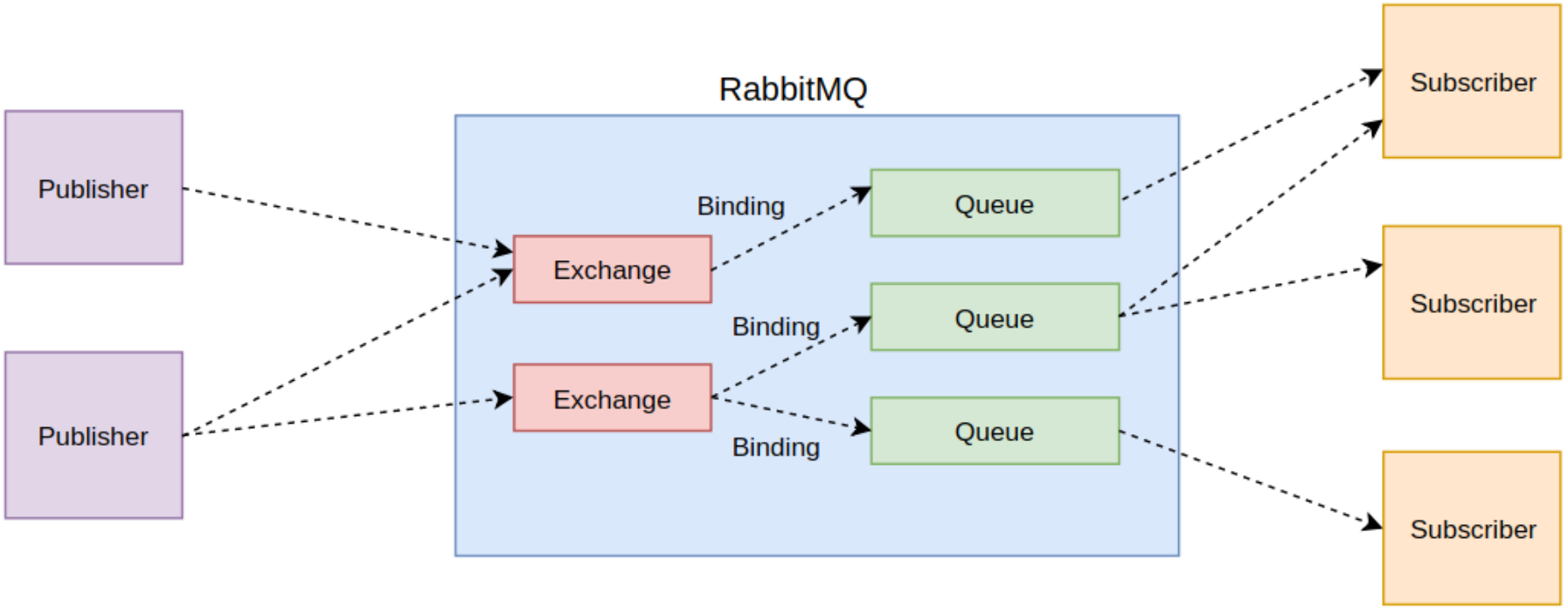
Application Delivery Fundamentals :

kafka

High performance. Delivered.

Rabbitmq Architectural Pattern

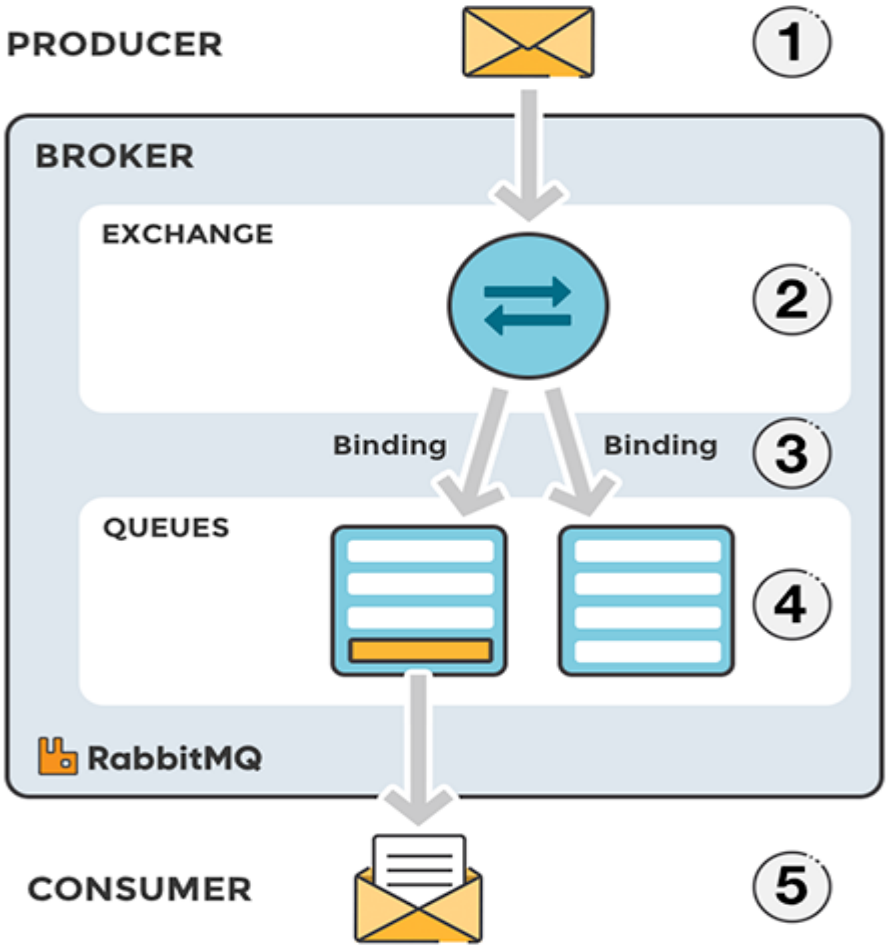


Rabbitmq Standard Exchange

kafka

- The producer publishes a message to the exchange.
- The exchange receives the message and is now responsible for the routing of the message.
- Binding must be set up between the queue and the exchange. In this case, we have bindings to two different queues from the exchange. The exchange routes the message into the queues.
- The messages stay in the queue until they are handled by a consumer.
- The consumer handles the message.

Rabbitmq Standard Exchange



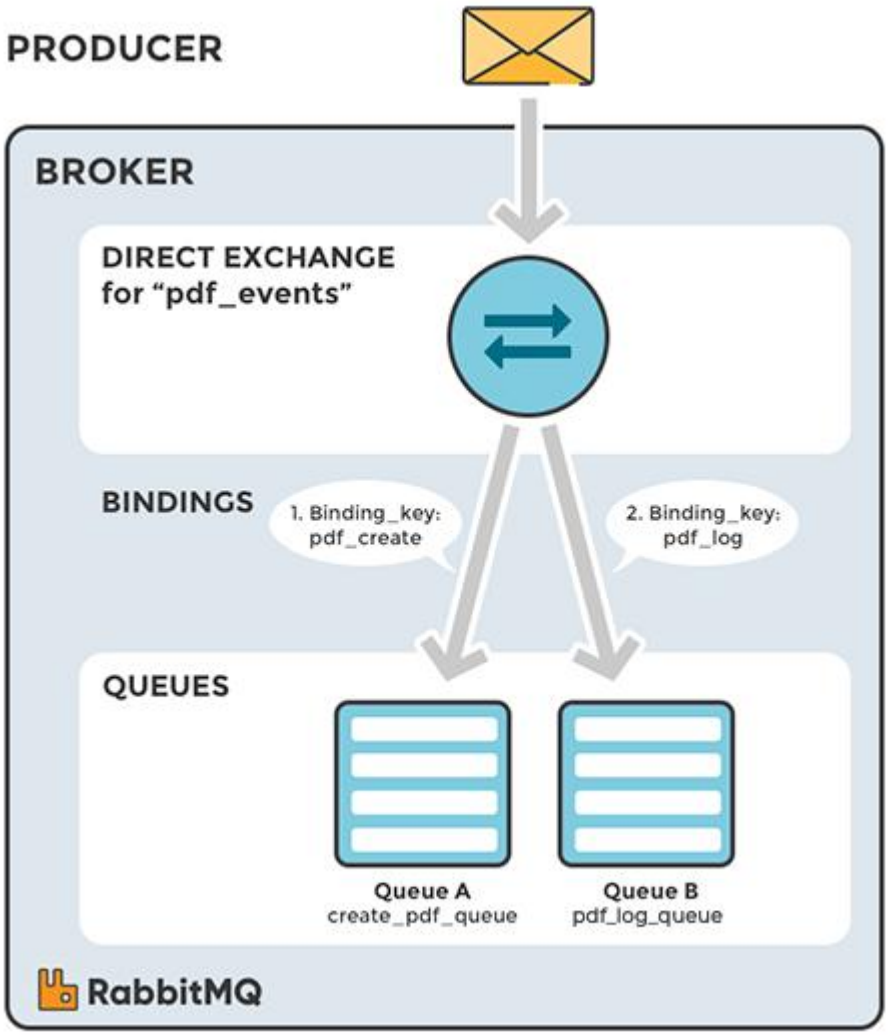
Exchange types - Direct Exchange

kafka

- A direct exchange delivers messages to queues based on a message routing key.
- The routing key is a message attribute added to the message header by the producer.
- Think of the routing key as an "address" that the exchange is using to decide how to route the message.
- A message goes to the queue(s) with the binding key that exactly matches the routing key of the message.

Exchange types - Direct Exchange

kafka



Exchange types - Default exchange

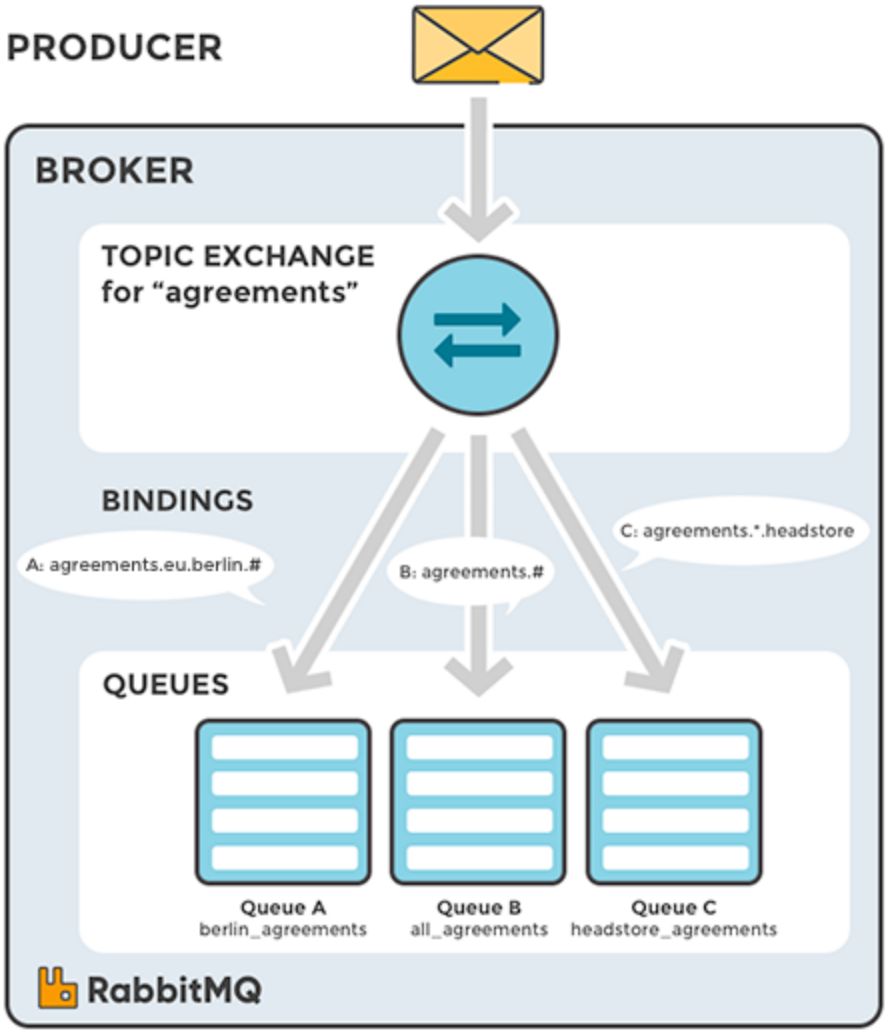
- The default exchange is a pre-declared direct exchange with no name, usually referred by an empty string.
- When you use default exchange, your message is delivered to the queue with a name equal to the routing key of the message.
- Every queue is automatically bound to the default exchange with a routing key which is the same as the queue name.

Exchange types - Topic Exchange

- Topic exchanges route messages to queues based on wildcard matches between the routing key and the routing pattern, which is specified by the queue binding.
- Messages are routed to one or many queues based on a matching between a message routing key and this pattern.

Exchange types - Topic Exchange

kafka



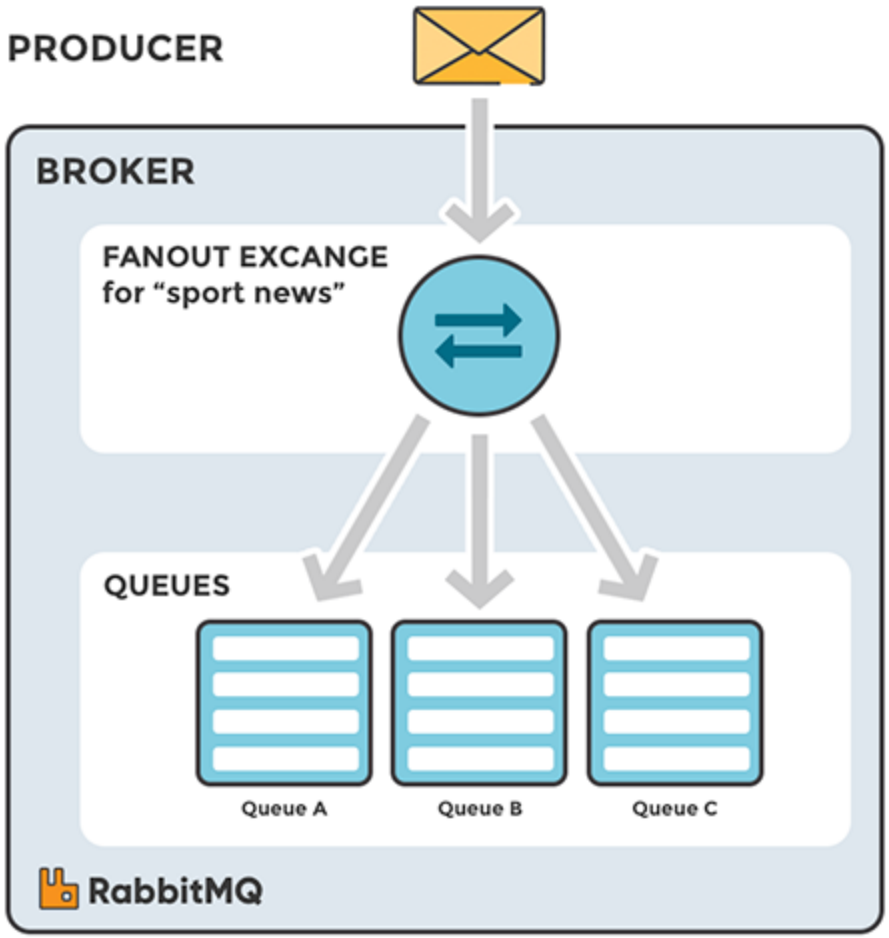
Exchange Types - Fanout Exchange

kafka

- A fanout exchange copies and routes a received message to all queues that are bound to it regardless of routing keys or pattern matching as with direct and topic exchanges. The keys provided will simply be ignored.
- Fanout exchanges can be useful when the same message needs to be sent to one or more queues with consumers who may process the same message in different ways.

Exchange Types - Fanout Exchange

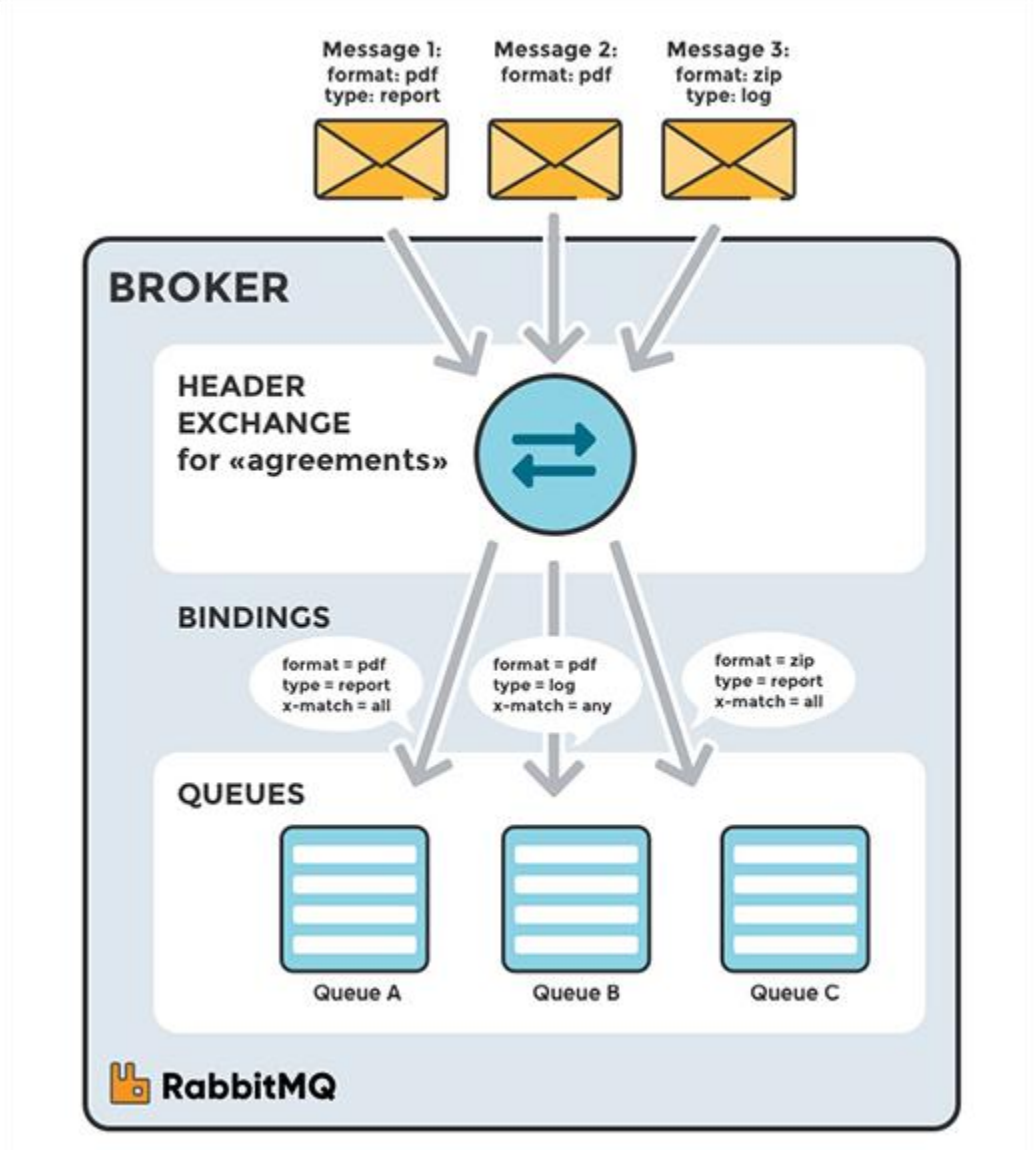
kafka



Exchange Types Headers Exchange

- A headers exchange routes messages based on arguments containing headers and optional values.
- Headers exchanges are very similar to topic exchanges, but route messages based on header values instead of routing keys.
- A message matches if the value of the header equals the value specified upon binding.

Rabbitmq – Headers Exchange



Exchange Type - DLQ

kafka

- If no matching queue can be found for the message, the message is silently dropped. RabbitMQ provides an AMQP extension known as the "Dead Letter Exchange", which provides the functionality to capture messages that are not deliverable.

RabbitMQ

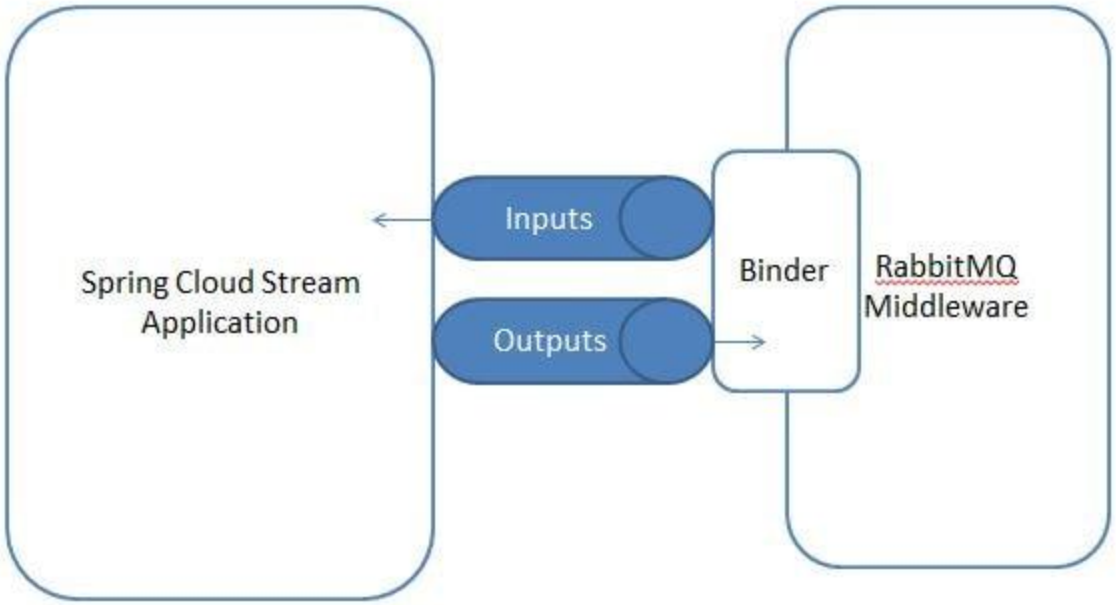
kafka

- The super simplified overview:
  - Publishers send messages to exchanges
  - Exchanges route messages to queues and other exchanges
  - RabbitMQ sends acknowledgements to publishers on message receipt
  - Consumers maintain persistent TCP connections with RabbitMQ and declare which queue(s) they consume
  - RabbitMQ pushes messages to consumers
  - Consumers send acknowledgements of success/failure
  - Messages are removed from queues once consumed successfully

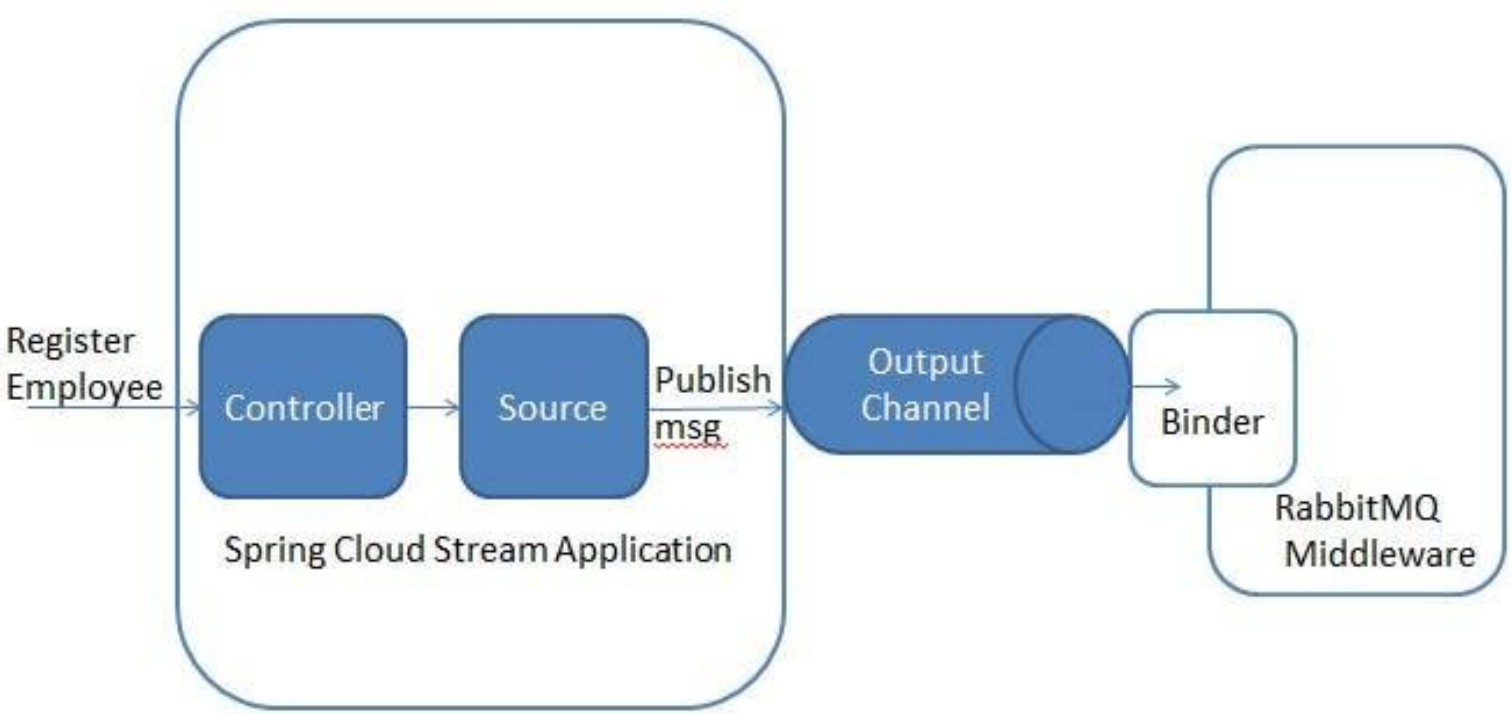
Core Components of Cloud Stream

- The core building blocks of Spring Cloud Stream are:
- Destination Binders: Components responsible to provide integration with the external messaging systems.
- Destination Bindings: Bridge between the external messaging systems and application provided Producers and Consumers of messages (created by the Destination Binders).
- Message: The canonical data structure used by producers and consumers to communicate with Destination Binders (and thus other applications via external messaging systems).

Core Components of Cloud Stream



Core Components of Cloud Stream



Asynchronous communication in Microservices

- Kafka
- RabbitMQ
- Google Pub/Sub
- Amazon Services
- ActiveMQ
- Azure Services

What is ZooKeeper

- ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.
- All of these kinds of services are used in some form or another by distributed applications.
- Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable.
- Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them, which make them brittle in the presence of change and difficult to manage.
- Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed.

What is ZooKeeper?

- Apache ZooKeeper plays the very important role in system architecture as it works in the shadow of more exposed Big Data tools, as Apache Spark or Apache Kafka.
- Originally, the ZooKeeper framework was built at “Yahoo!”. Because it helps to access their applications in an easy manner.


Why Apache ZooKeeper ?
IntelliPaat
1
Naming Service
2
Configuration Management
3
Synchronization Service
4
Group Services

Why Zookeeper

kafka

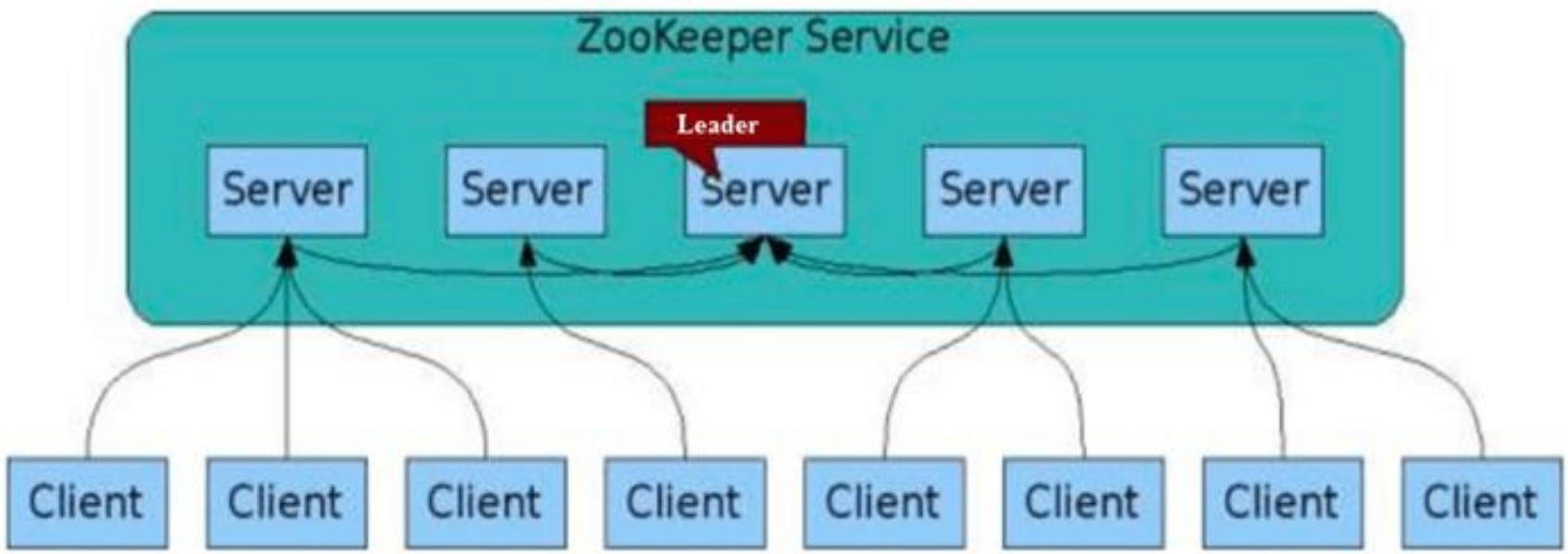
- It allows for mutual exclusion and cooperation between server processes.
- It ensures that your application runs consistently.
- The transaction process is never completed partially. It is either given the status of Success or failure. The distributed state can be held up, but it's never wrong.
- Irrespective of the server that it connects to, a client will be able to see the same view of the service
- Helps you to encode the data as per the specific set of rules

Why Zookeeper

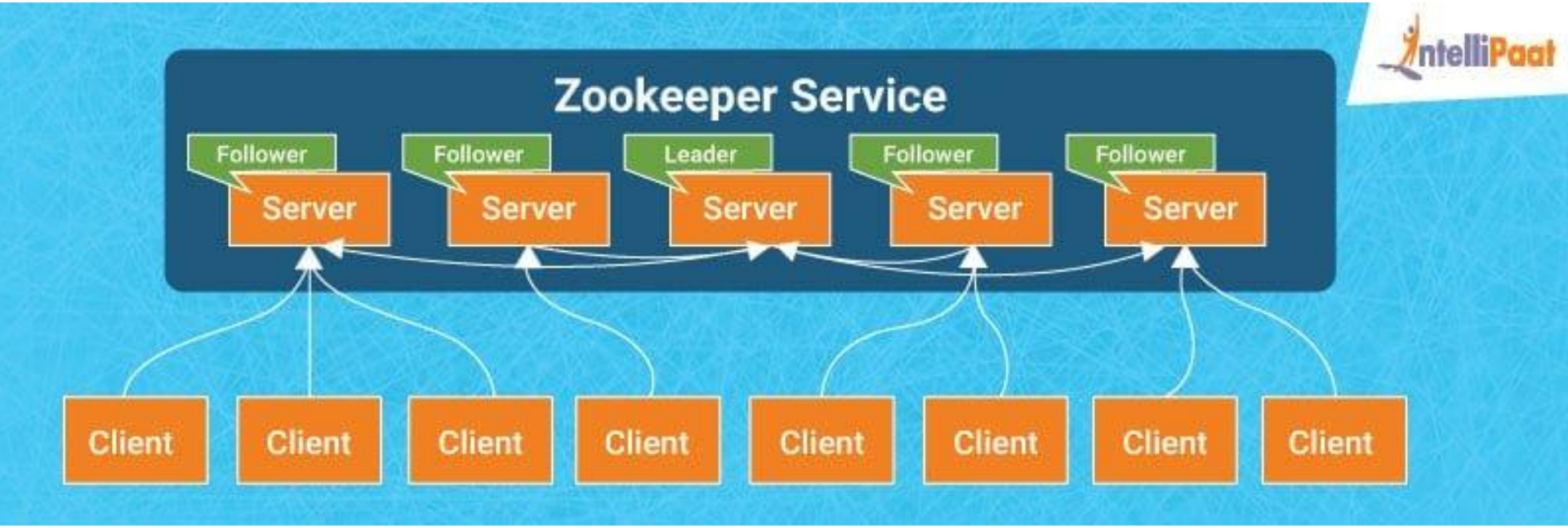
kafka

- It helps to maintain a standard hierarchical namespace similar to files and directories.
- Computers, which run as a single system which can be locally or geographically connected.
- It allows to Join/leave node in a cluster and node status at the real time
- You can increase performance by deploying more machines.
- It allows you to elect a node as a leader for better coordination.
- ZooKeeper works fast with workloads where reads to the data are more common than writes.

ZooKeeper Architecture: How it works?



ZooKeeper Architecture: How it works?



ZooKeeper Architecture

kafka

- Server: The server sends an acknowledge when any client connects. In the case when there is no response from the connected server, the client automatically redirects the message to another server.

- Client: Client is one of the nodes in the distributed application cluster. It helps you to accesses information from the server. Every client sends a message to the server at regular intervals that helps the server to know that the client is alive.

ZooKeeper Architecture

kafka

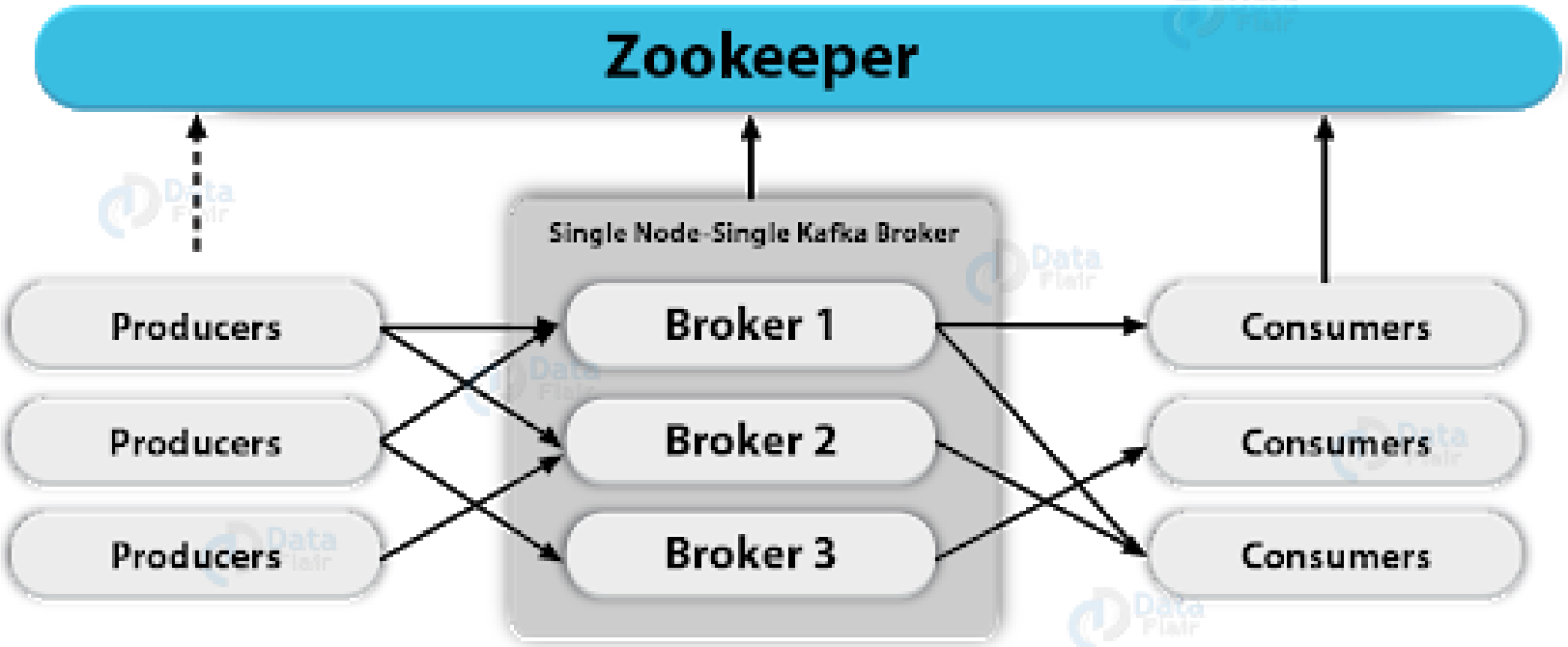
- Leader: One of the servers is designated a Leader. It gives all the information to the clients as well as an acknowledgment that the server is alive. It would performs automatic recovery if any of the connected nodes failed.
- Follower: Server node which follows leader instruction is called a follower.
- Client read requests are handled by the correspondingly connected Zookeeper server
- The client writes requests are handled by the Zookeeper leader.

ZooKeeper Architecture

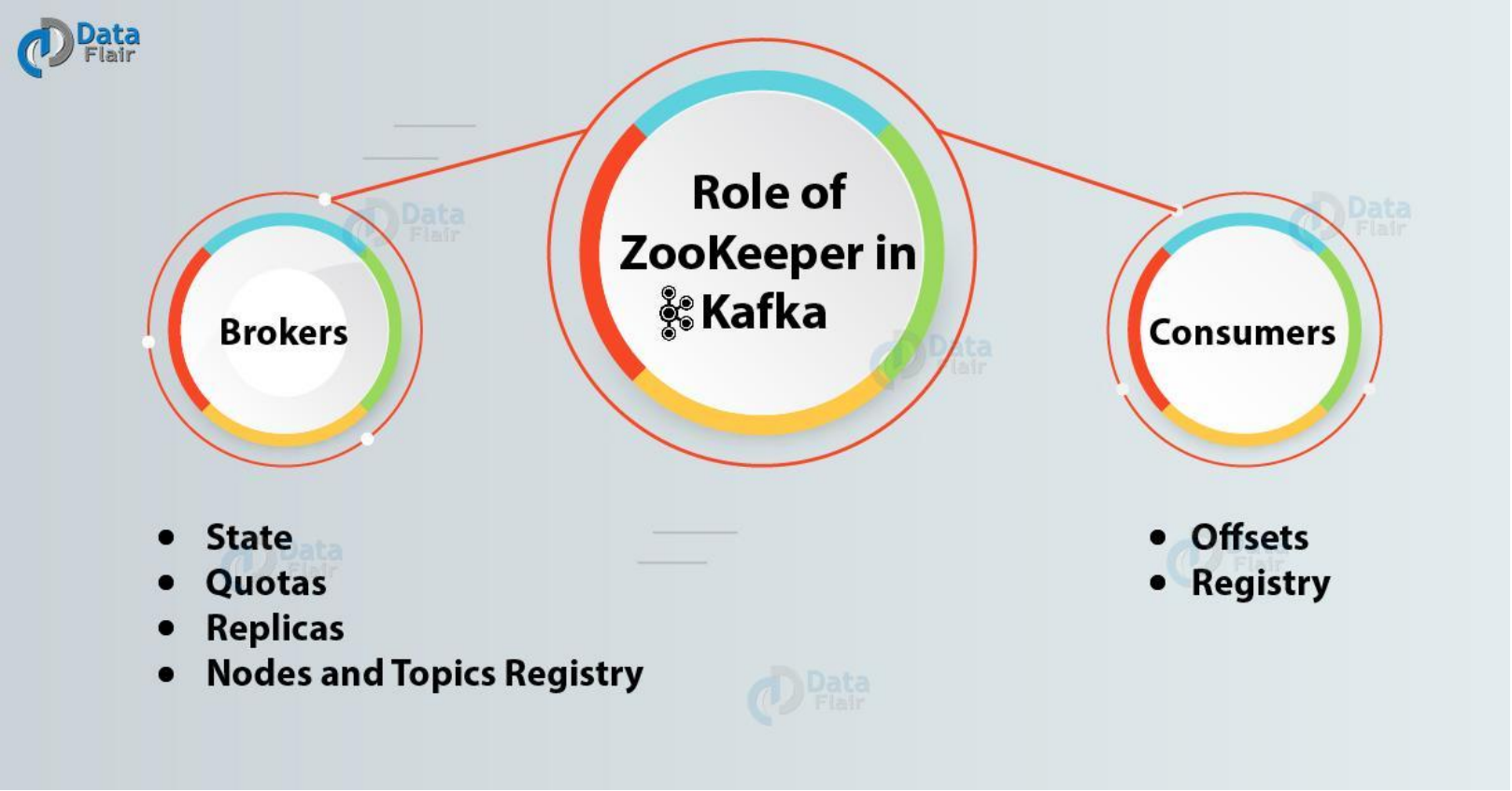
kafka

- Ensemble/Cluster: Group of Zookeeper servers which is called ensemble or a Cluster. You can use ZooKeeper infrastructure in the cluster mode to have the system at the optimal value when you are running the Apache.
- ZooKeeper WebUI: If you want to work with ZooKeeper resource management, then you need to use WebUI. It allows working with ZooKeeper using the web user interface, instead of using the command line. It offers fast and effective communication with the ZooKeeper application.

What is ZooKeeper?



ZooKeeper in Kafka



Companies using Zookeeper

- Yahoo
- Facebook
- eBay
- Twitter
- Netflix
- Zynga
- Nutanix

What is Kafka

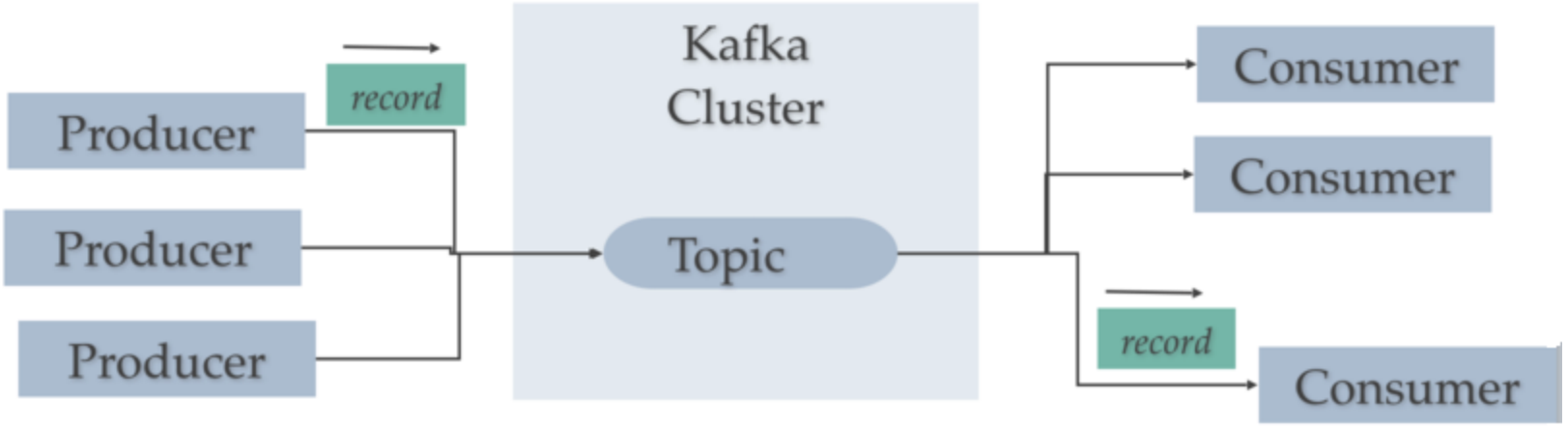
- Kafka consists of Records, Topics, Consumers, Producers, Brokers, Logs, Partitions, and Clusters. Records can have key (optional), value and timestamp.
- Kafka Records are immutable.
- A Kafka Topic is a stream of records ("/orders", "/user-signups").
- You can think of a Topic as a feed name.
- A topic has a Log which is the topic’s storage on disk.
- A Topic Log is broken up into partitions and segments.

What is Kafka

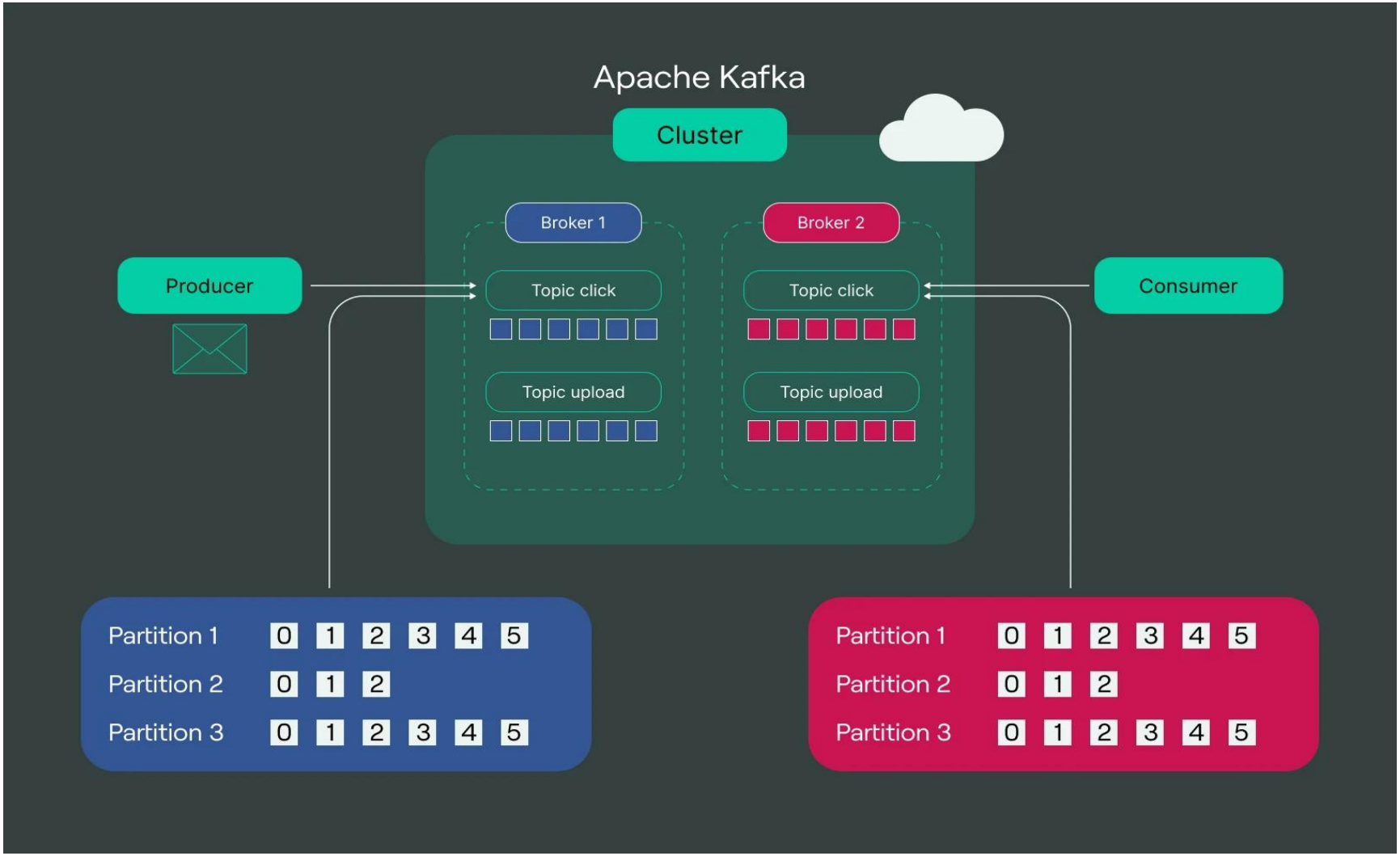
- The Kafka Producer API is used to produce streams of data records.
- The Kafka Consumer API is used to consume a stream of records from Kafka.
- A Broker is a Kafka server that runs in a Kafka Cluster.
- Kafka Brokers form a cluster.
- The Kafka Cluster consists of many Kafka Brokers on many servers.
- Broker sometimes refer to more of a logical system or as Kafka as a whole.

Kafka

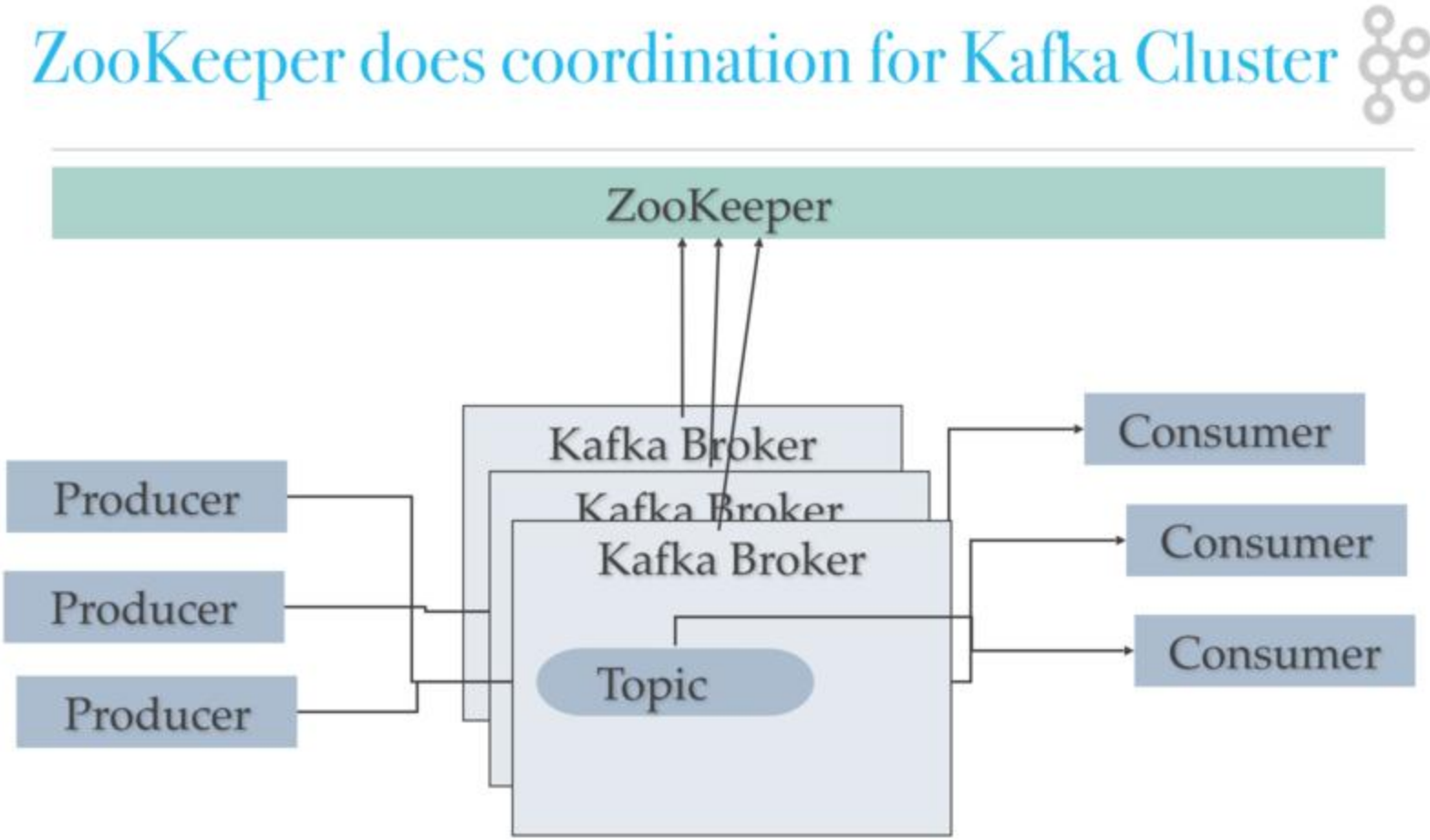
Kafka: Topics, Producers, and Consumers



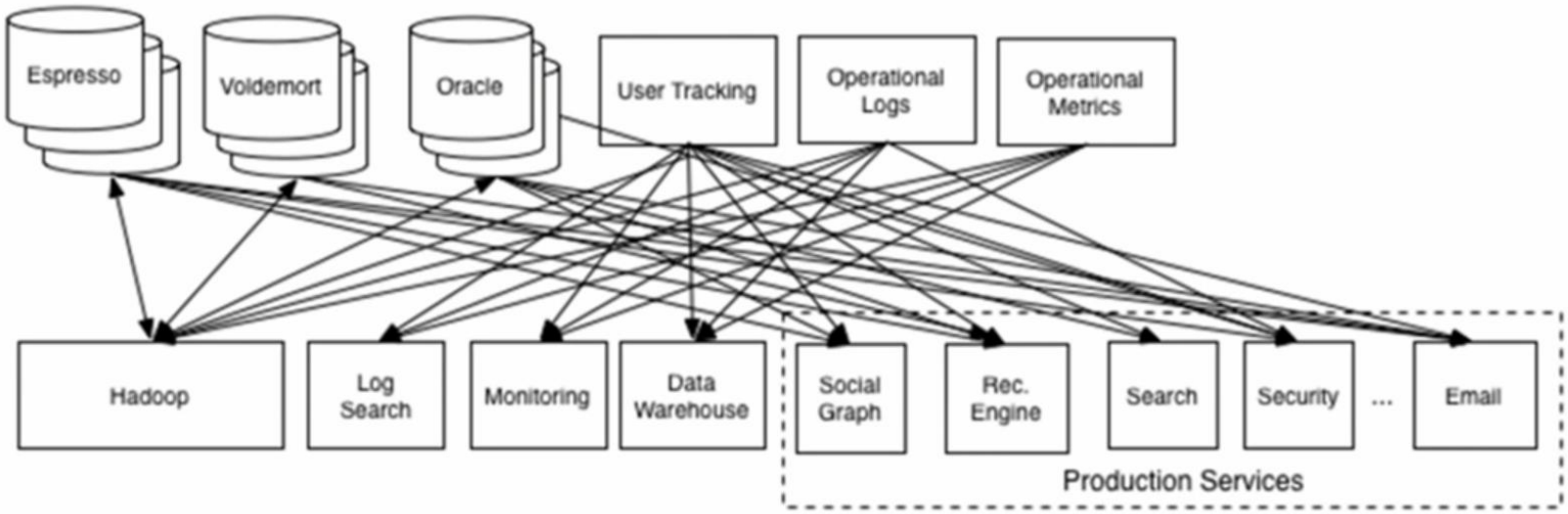
Apache Kafka



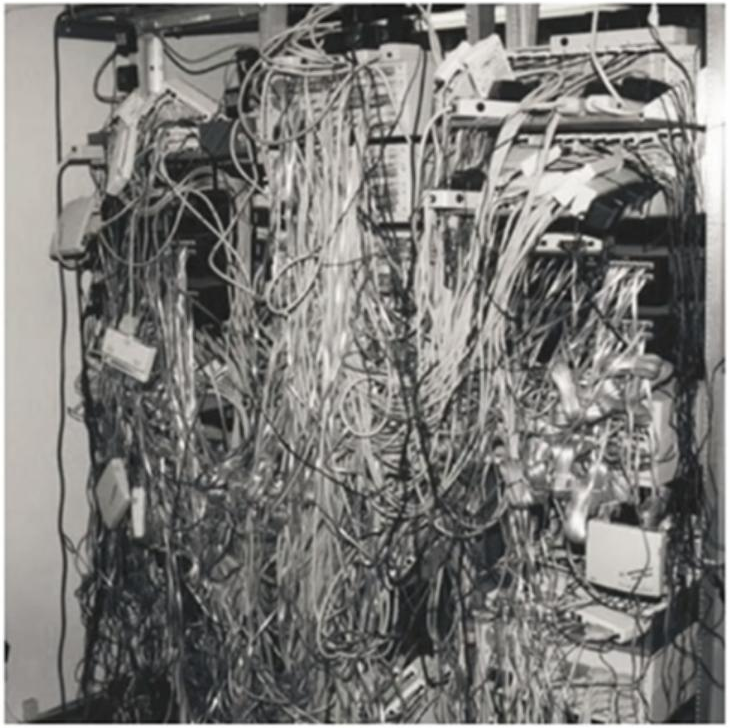
Kafka


ZooKeeper does coordination for Kafka Cluster
ZooKeeper
Kafka Broker
Kafka Broker
Kafka Broker
Topic
Producer
Producer
Producer
Consumer
Consumer
Consumer

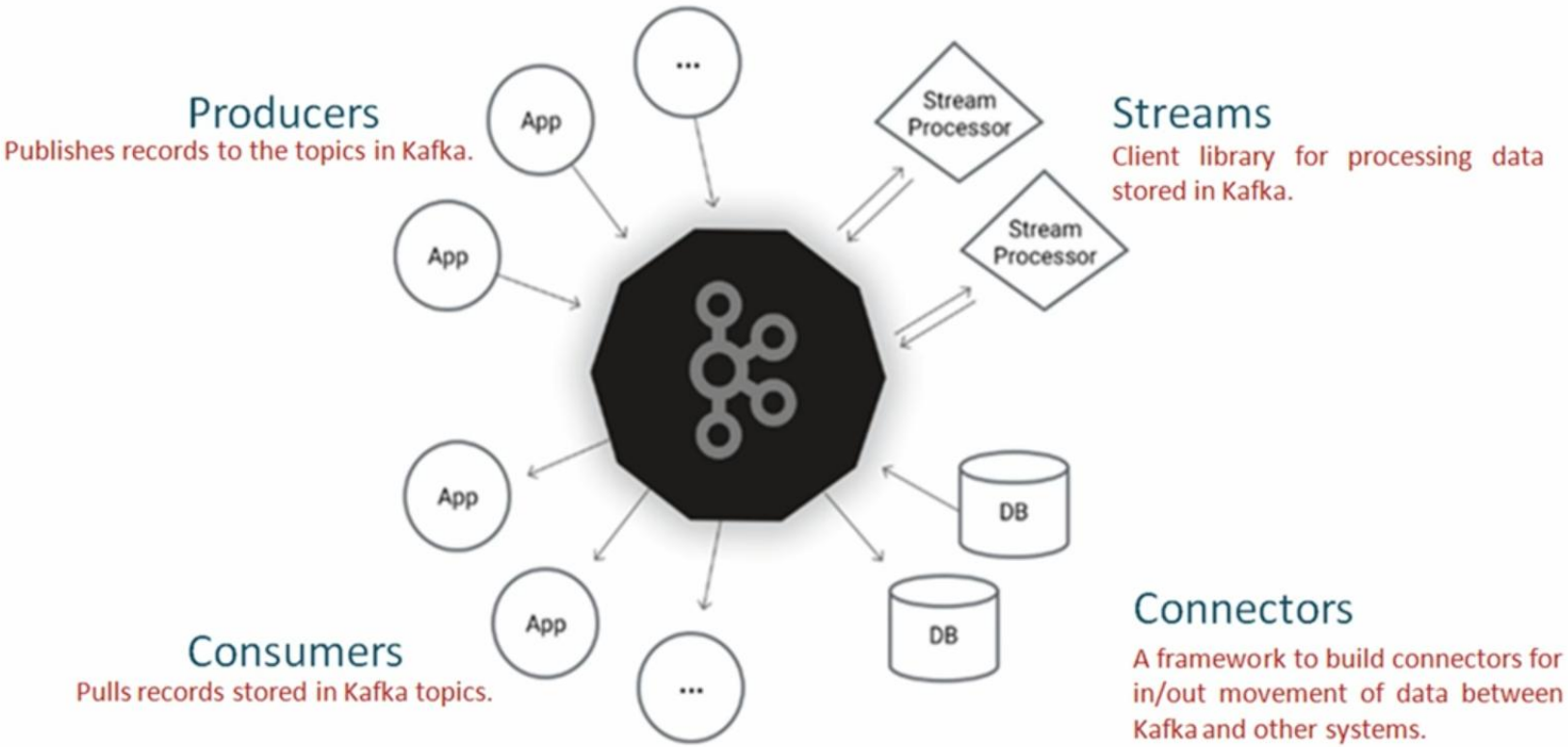
Why Kafka


Espresso
Voldemort
Oracle
User Tracking
Operational Logs
Operational Metrics
Hadoop
Log Search
Monitoring
Data Warehouse
Social Graph
Rec. Engine
Search
Security
...
Email
Production Services

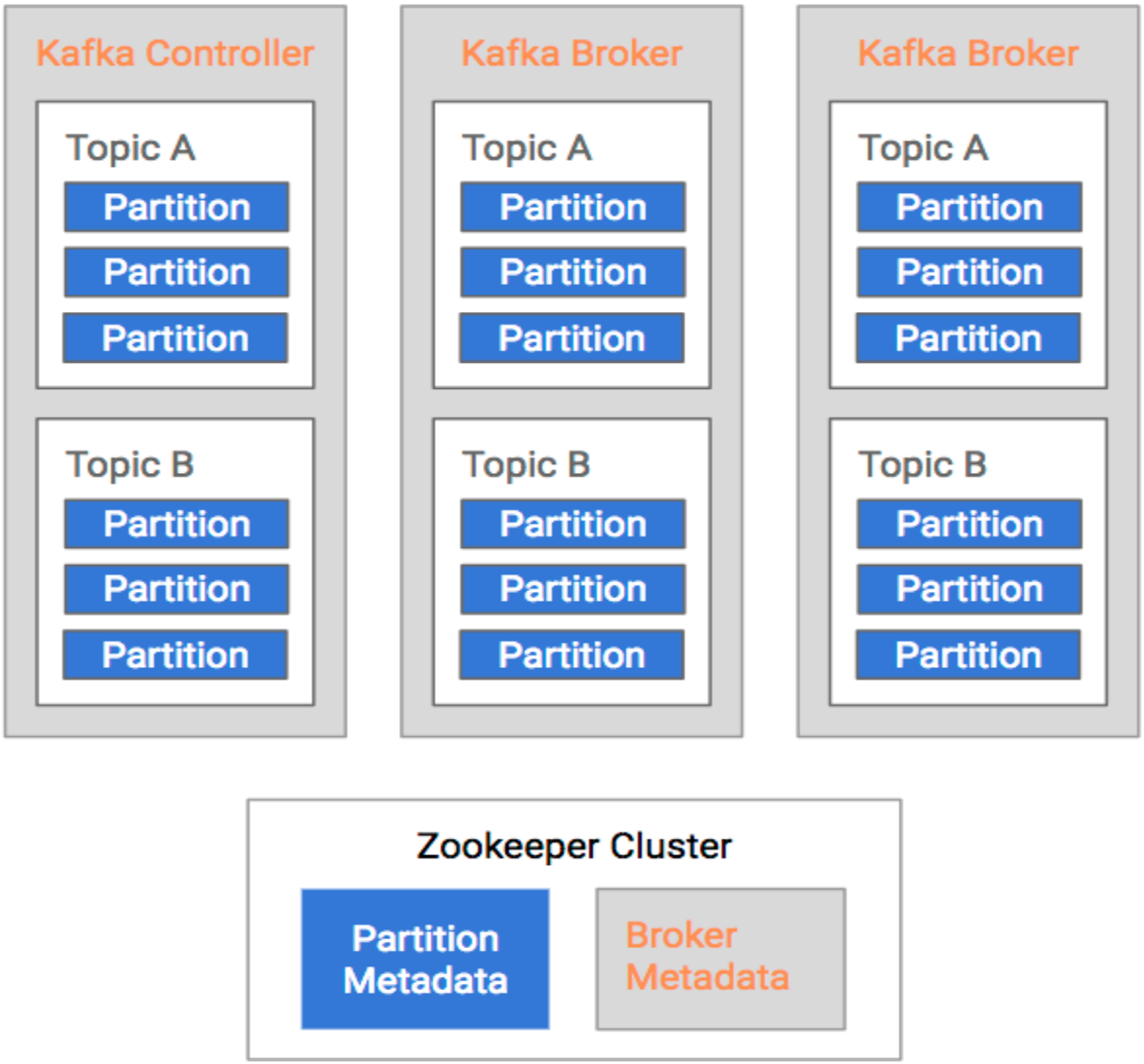
Why Kafka



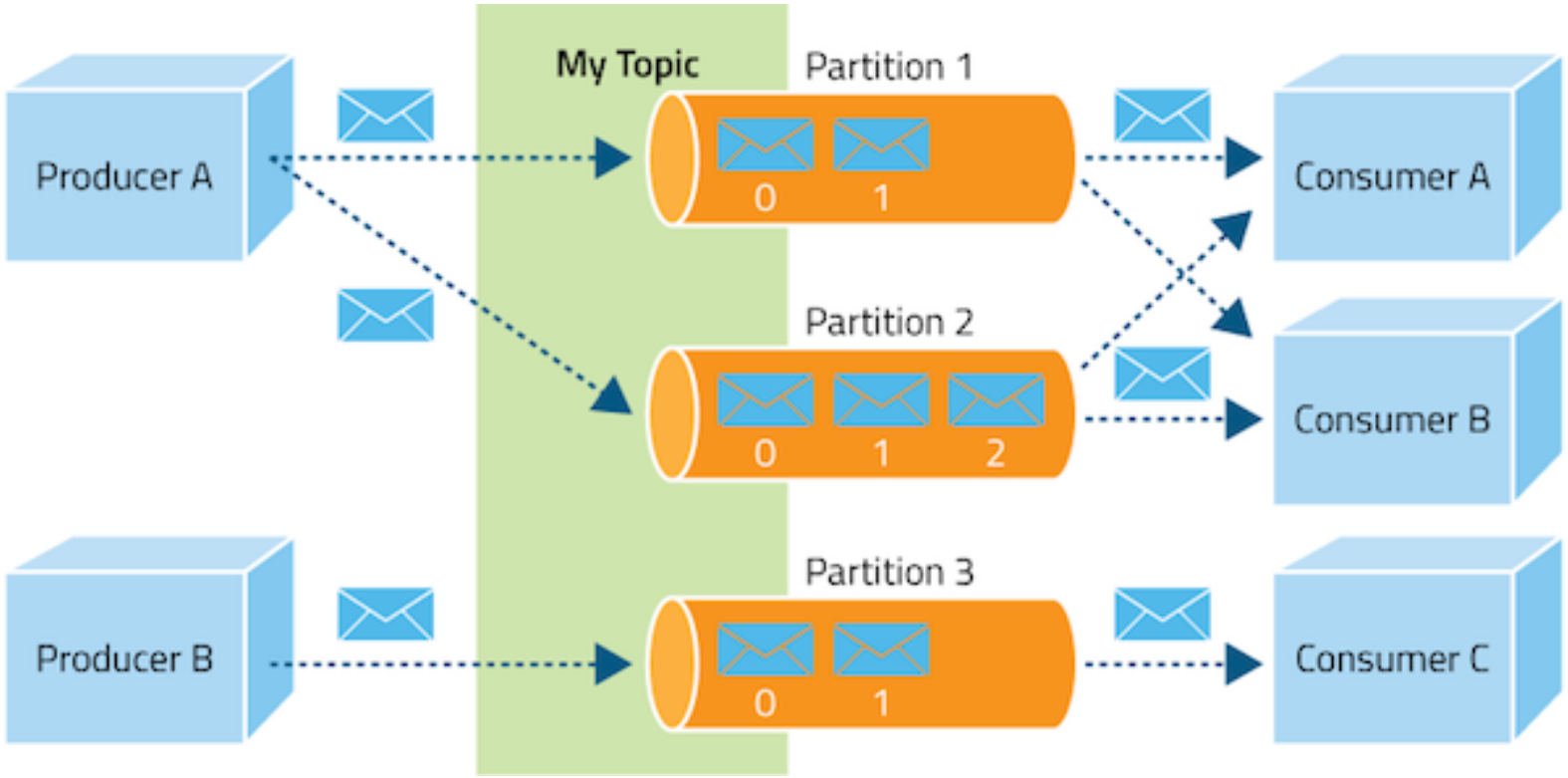
Why Kafka


Producers
Publishes records to the topics in Kafka.
App
App
App
App
...
...
Stream Processor
Stream Processor
Streams
Client library for processing data stored in Kafka.
DB
DB
Connectors
A framework to build connectors for in/out movement of data between Kafka and other systems.
Consumers
Pulls records stored in Kafka topics.

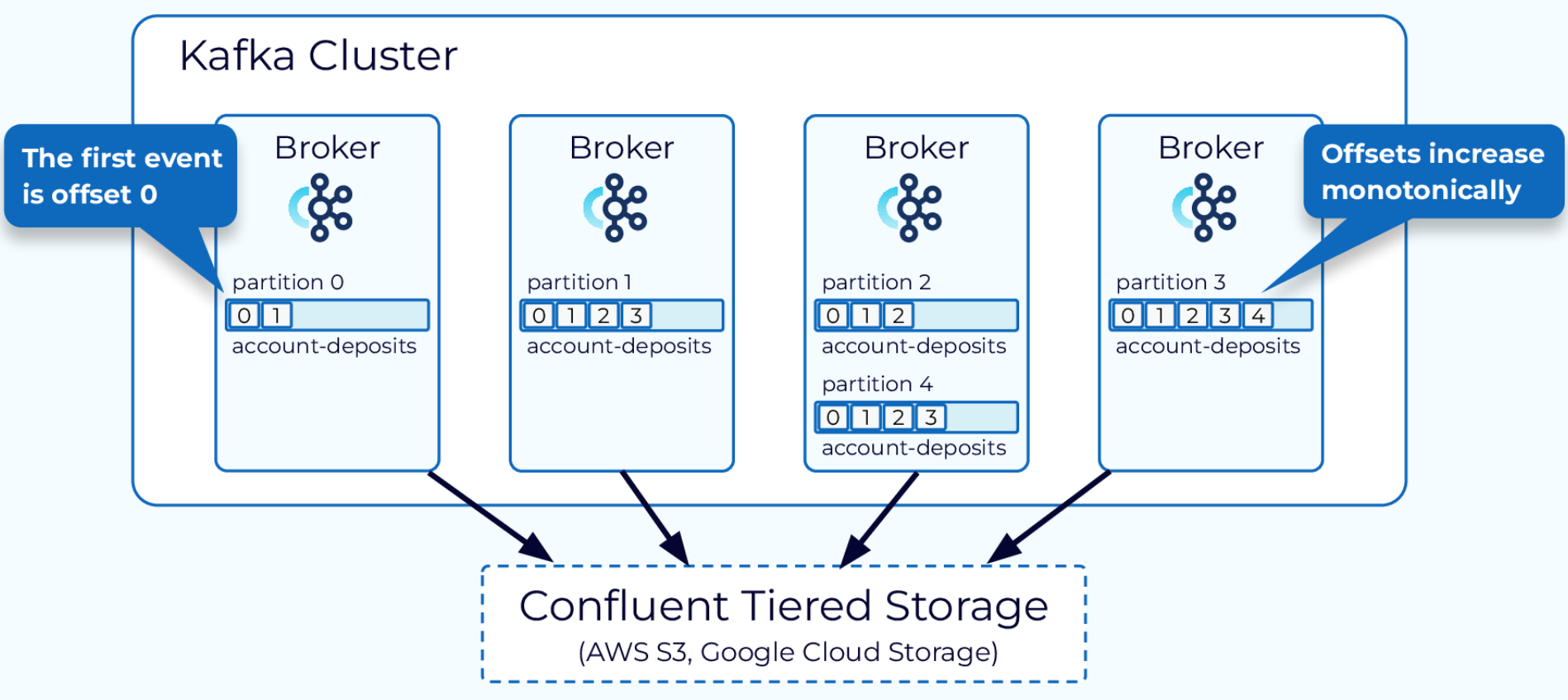
Kafka brokers and Zookeeper



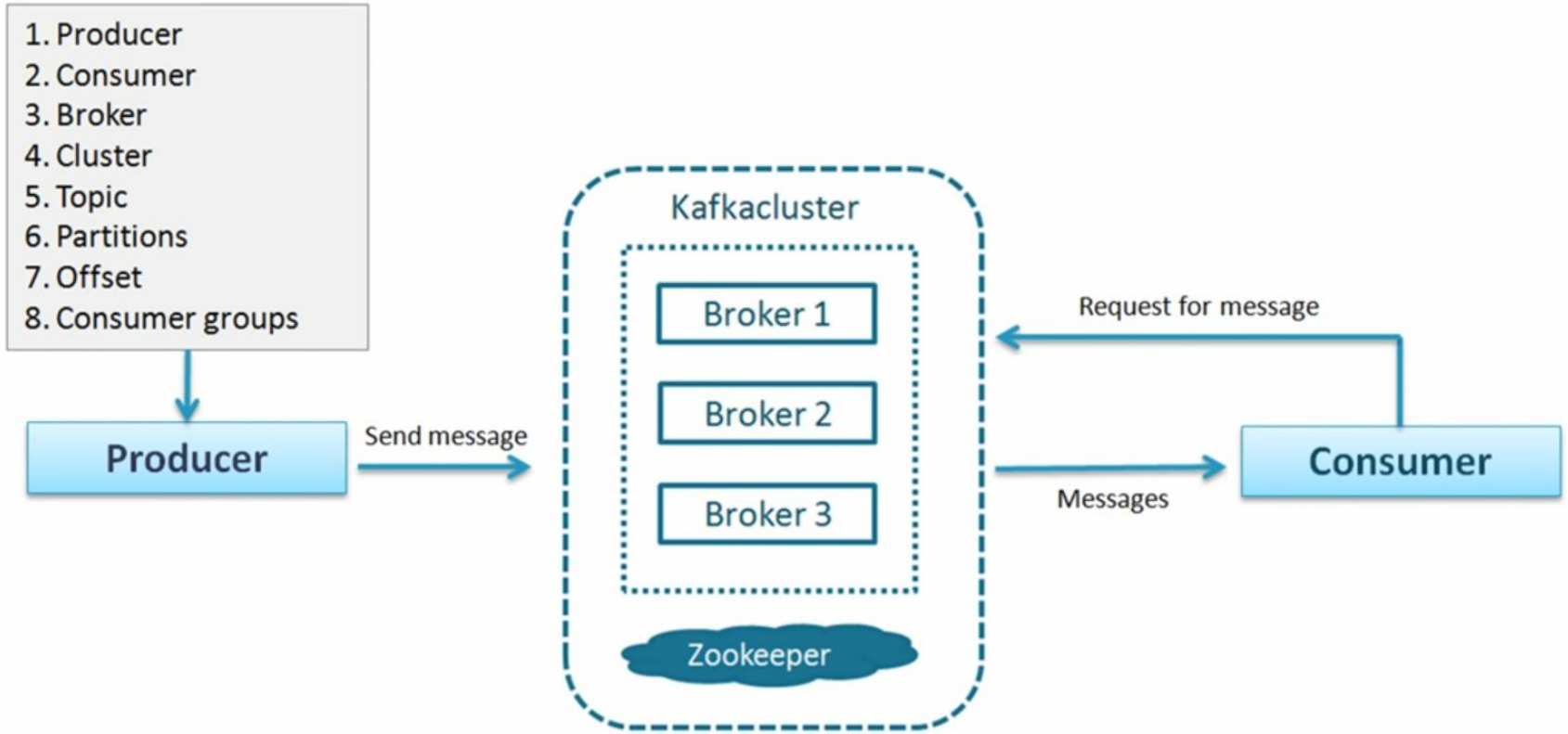
Kafka brokers and Zookeeper



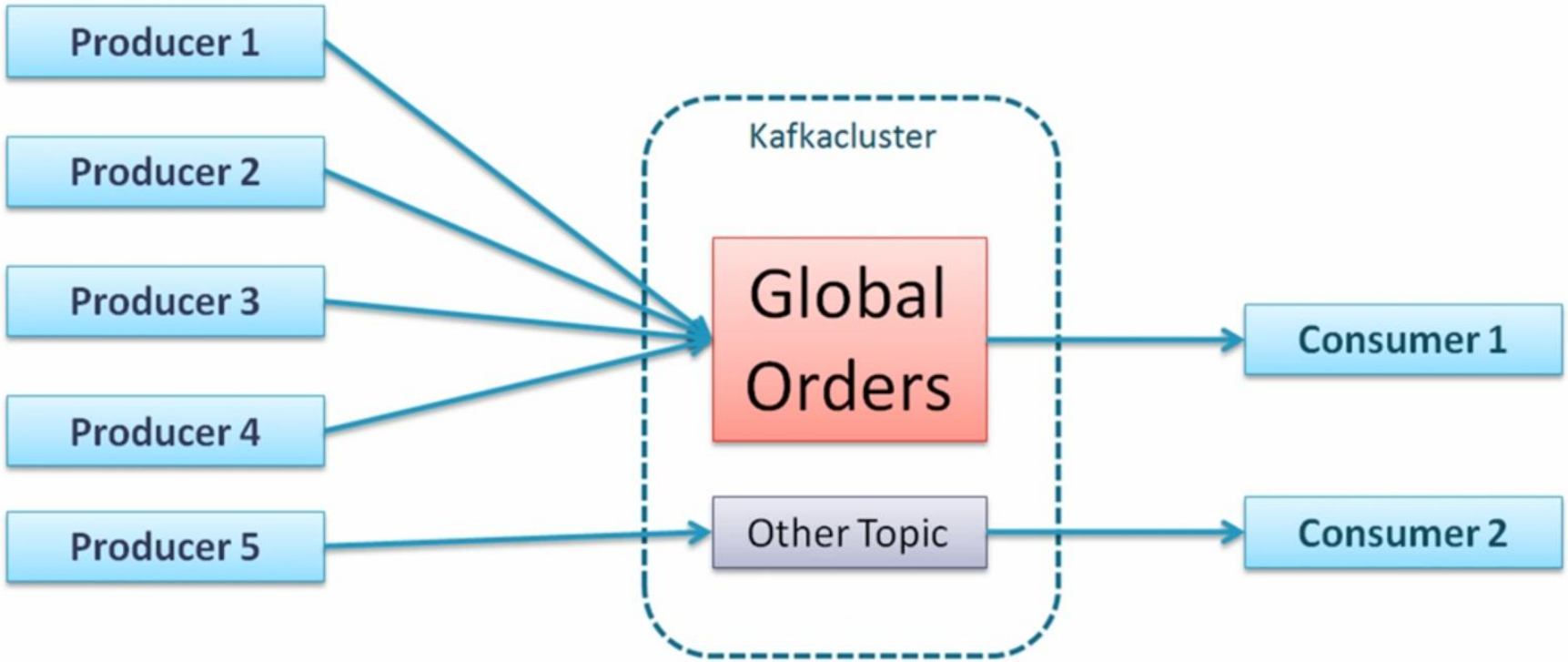
Kafka brokers, Topics and Partitions



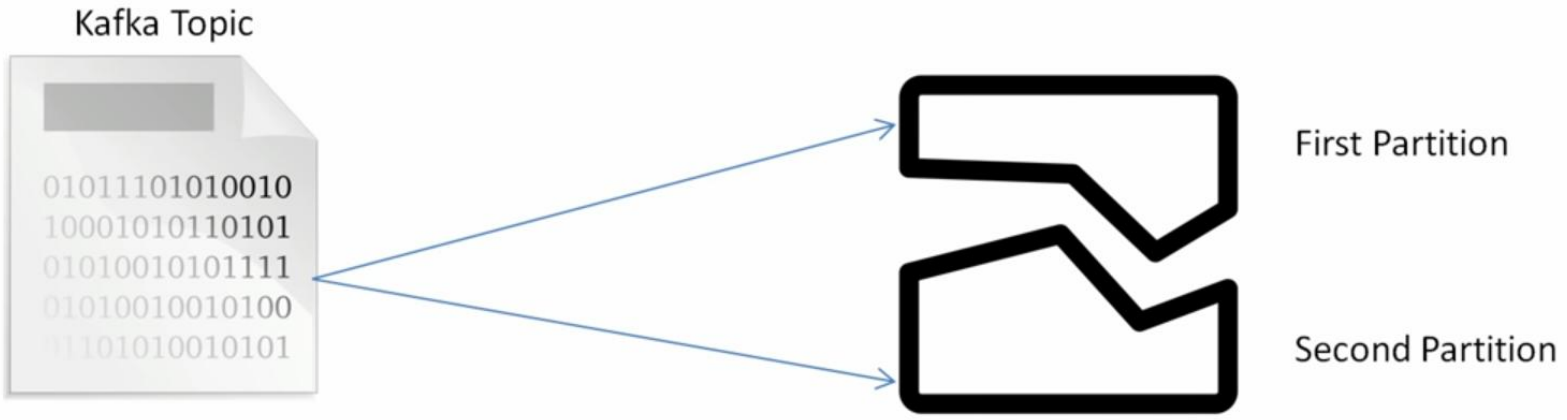
Kafka brokers and Zookeeper



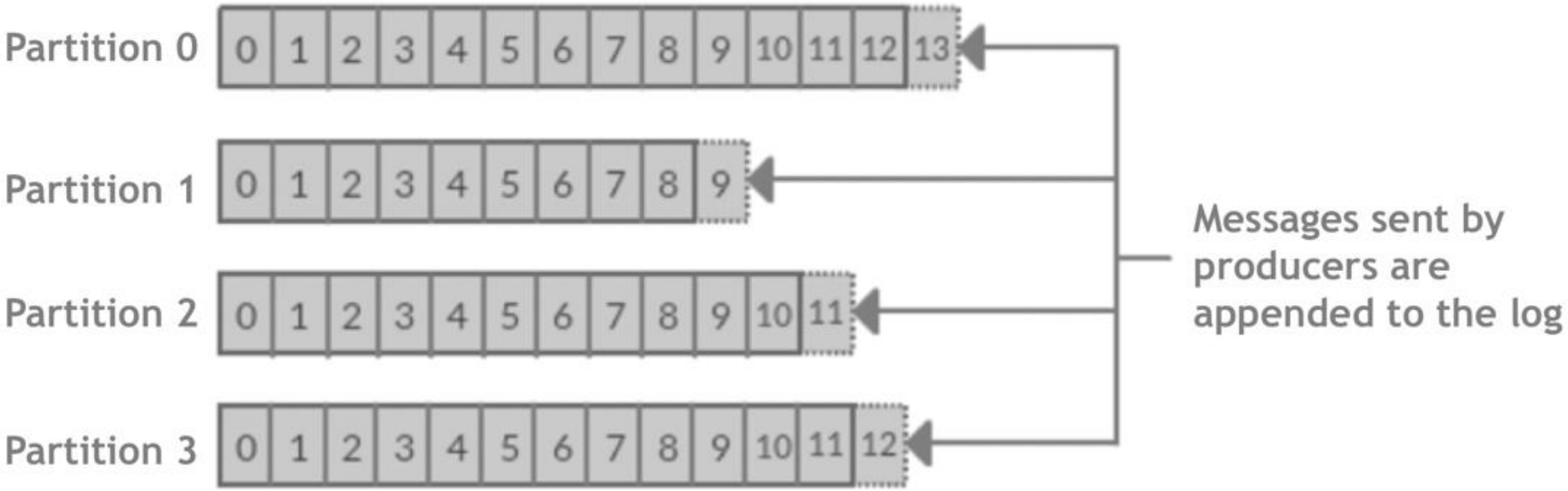
Kafka brokers and Zookeeper



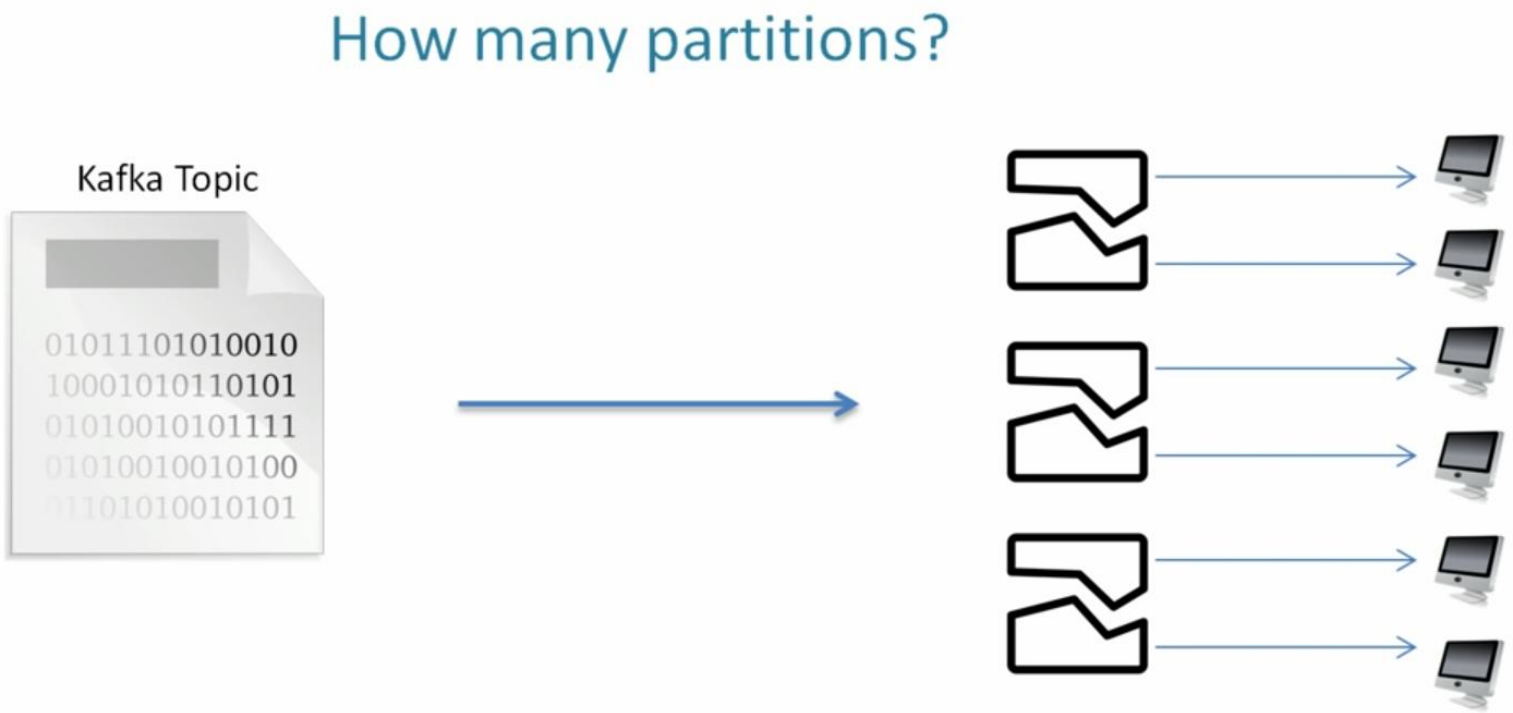
Kafka brokers and Zookeeper



Kafka brokers and Zookeeper



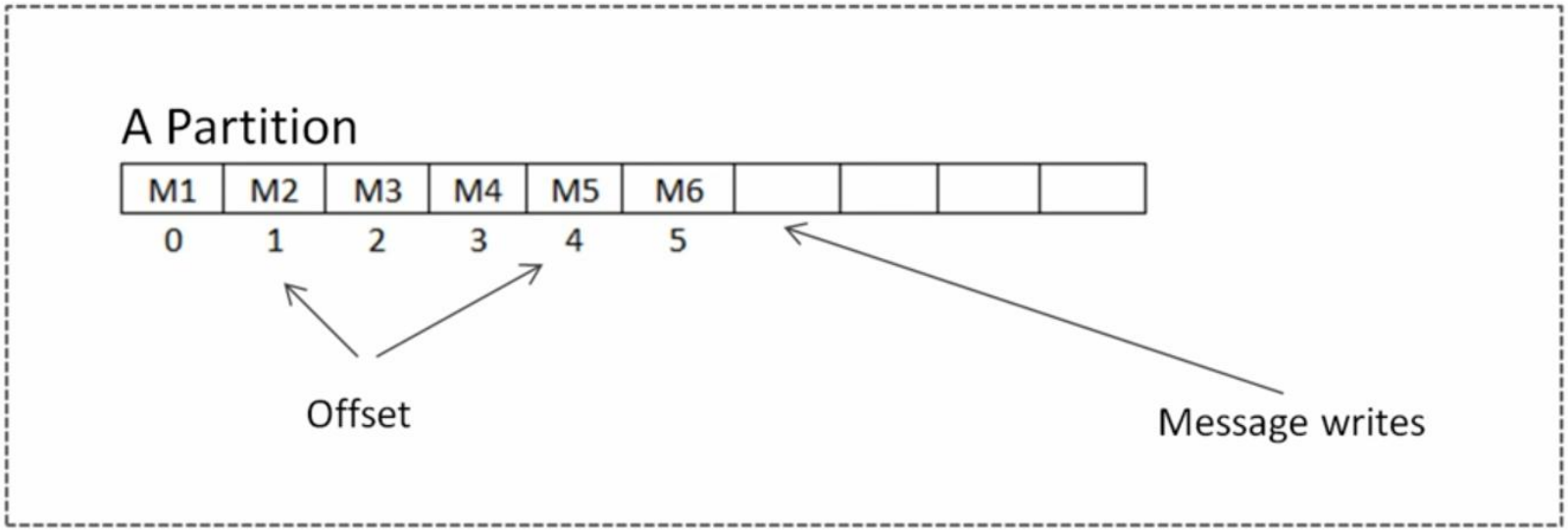
Kafka brokers and Zookeeper



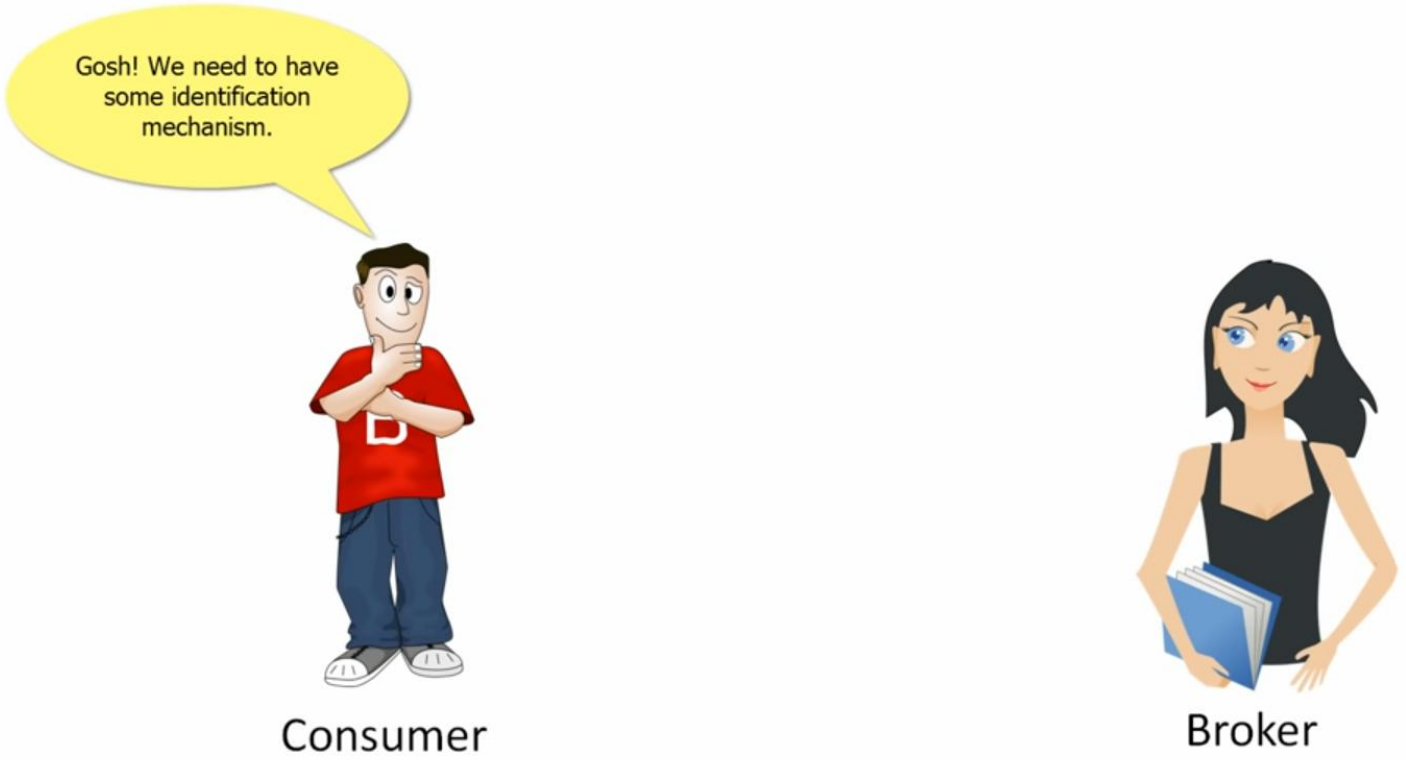
Kafka brokers and Zookeeper

What is offset?

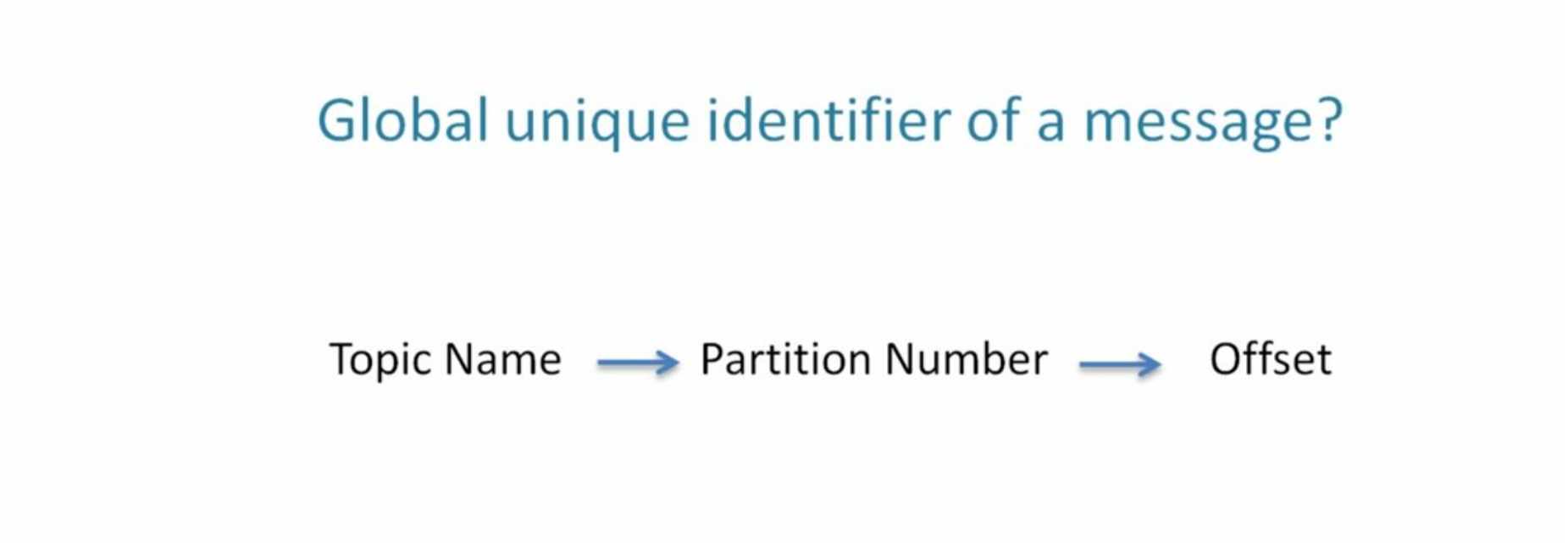
A sequence id given to messages as they arrive in a partition.



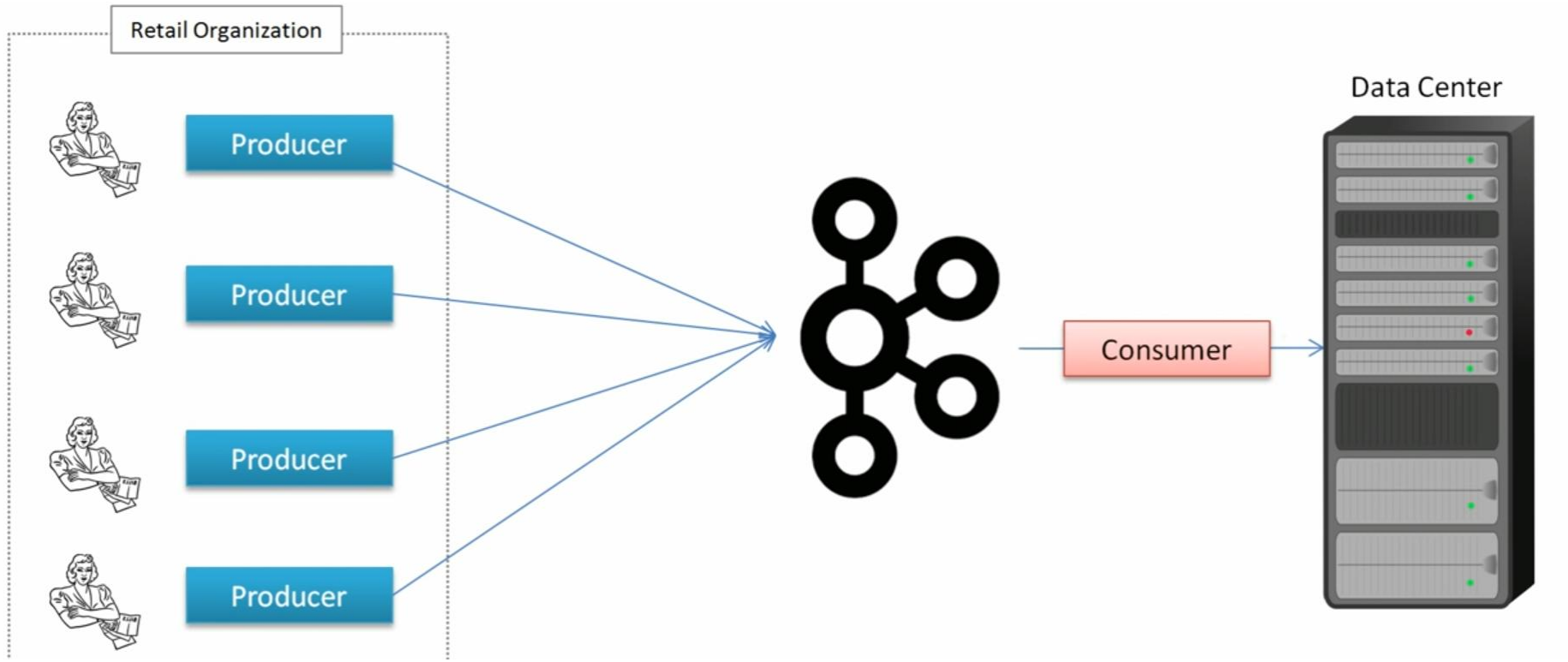
Kafka brokers and Zookeeper



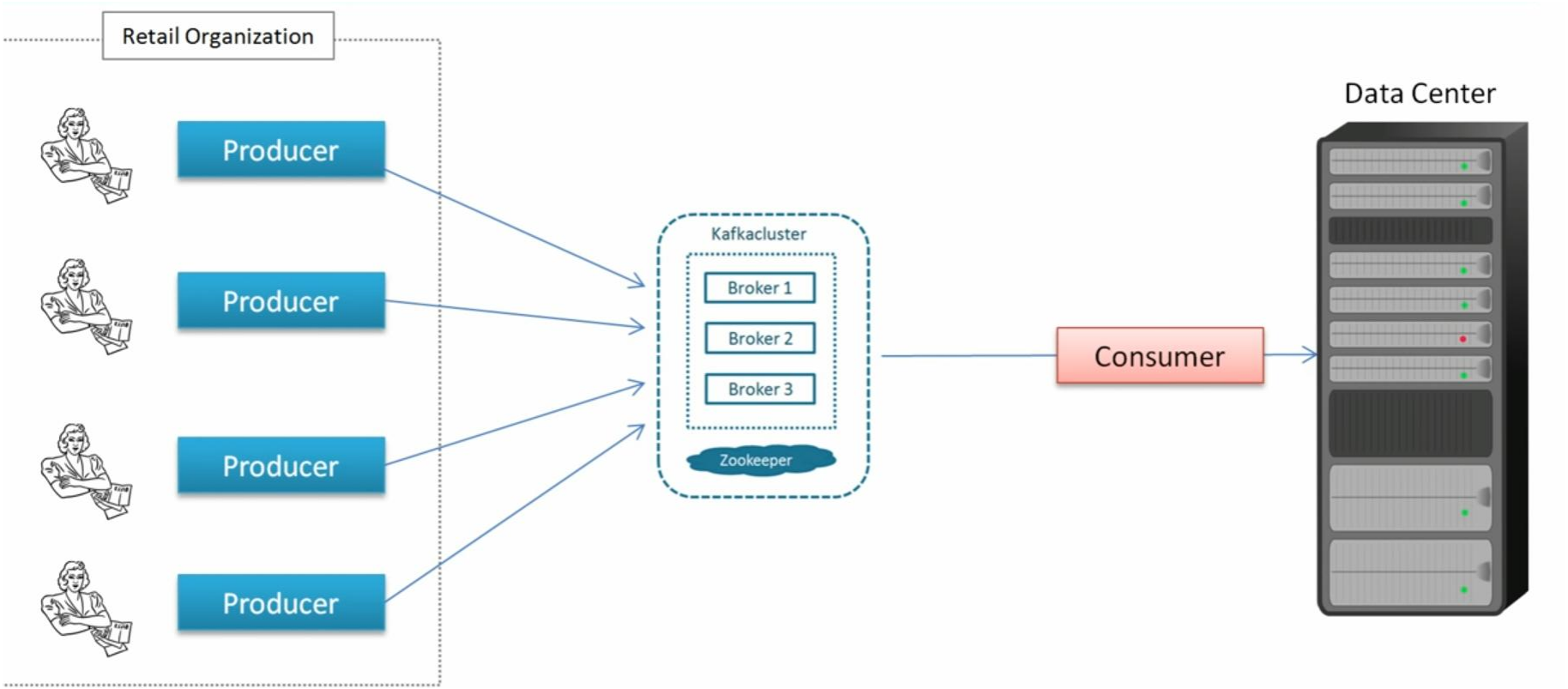
Kafka brokers and Zookeeper



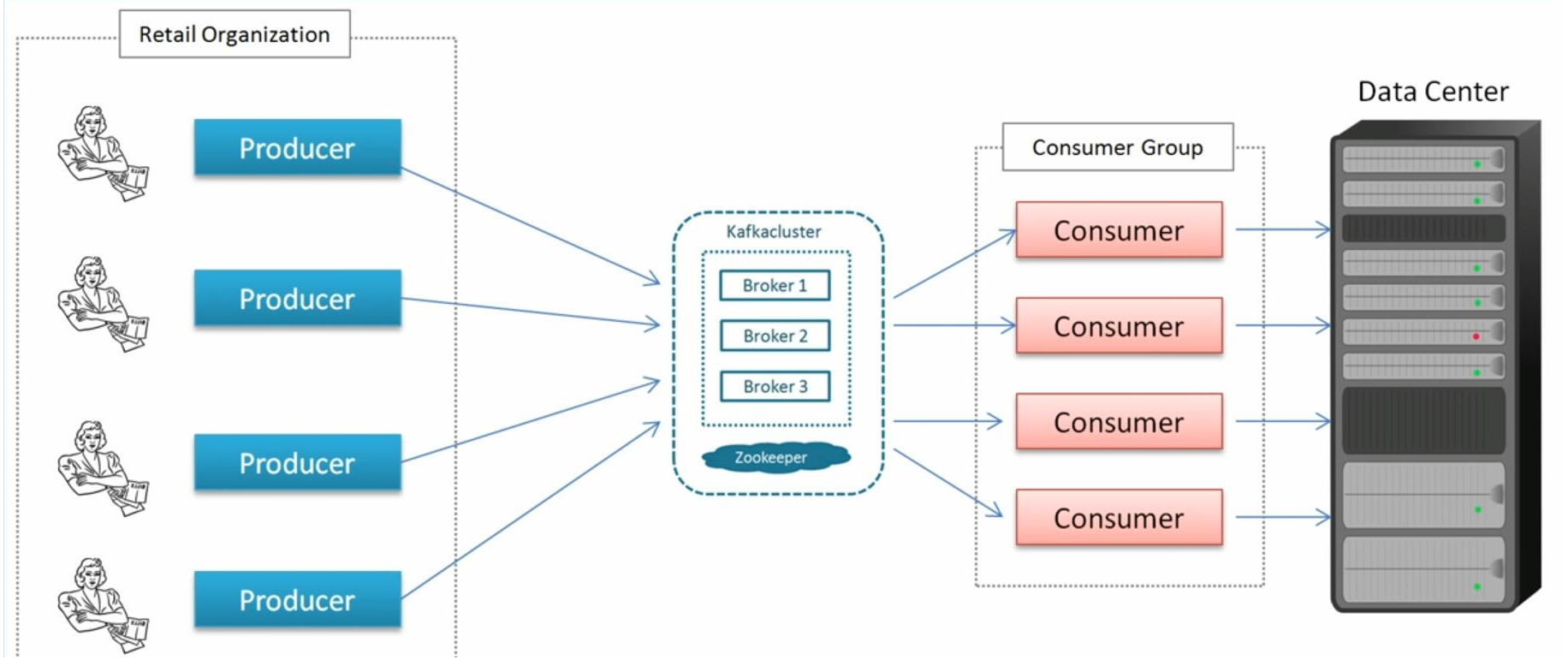
Kafka brokers and Zookeeper



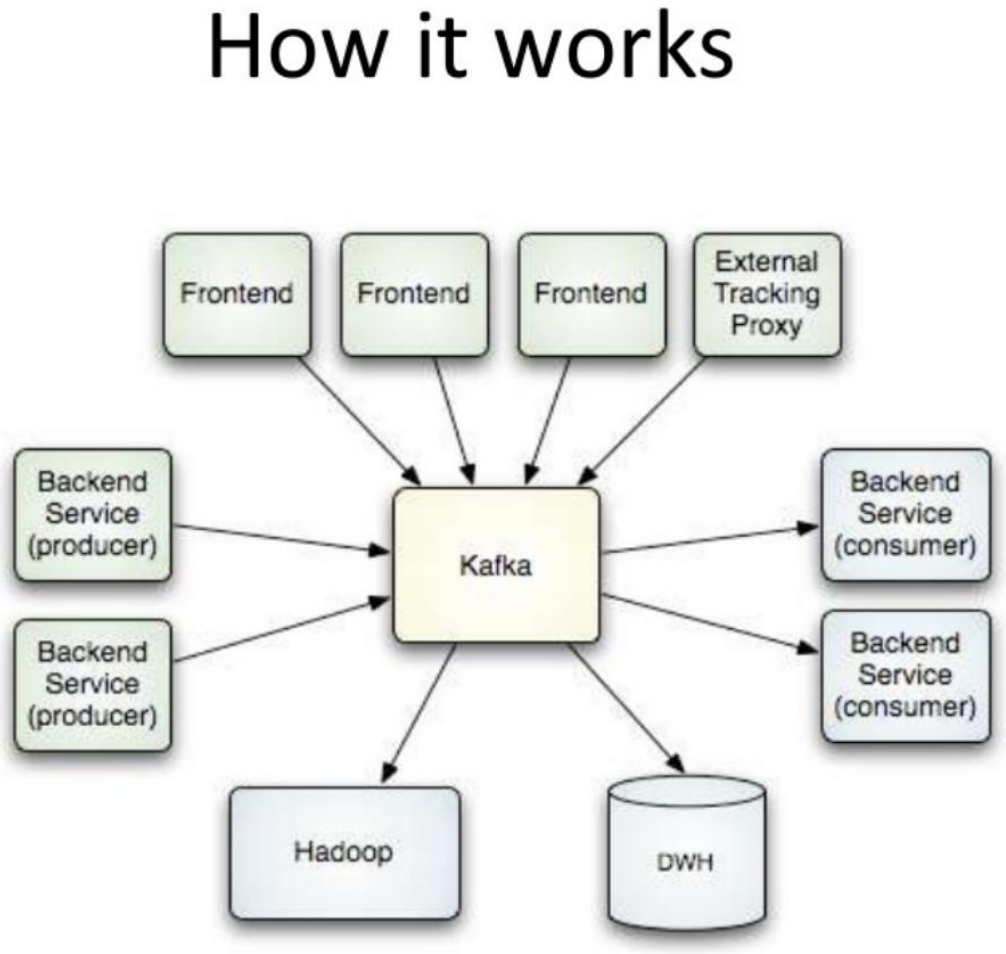
Kafka brokers and Zookeeper



Kafka brokers and Zookeeper

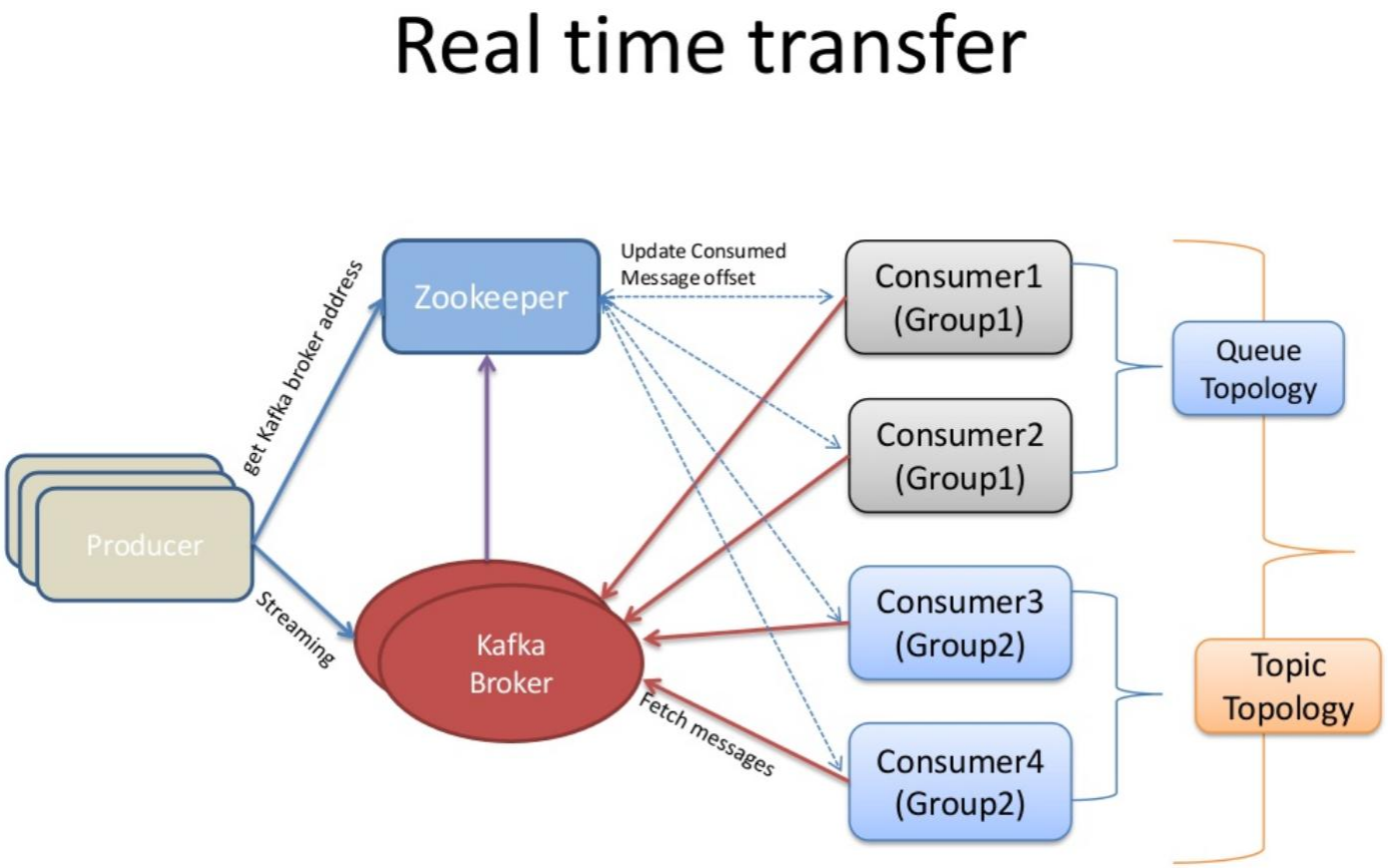


Kafka Work Flow



Credit : http://kafka.apache.org/design.html

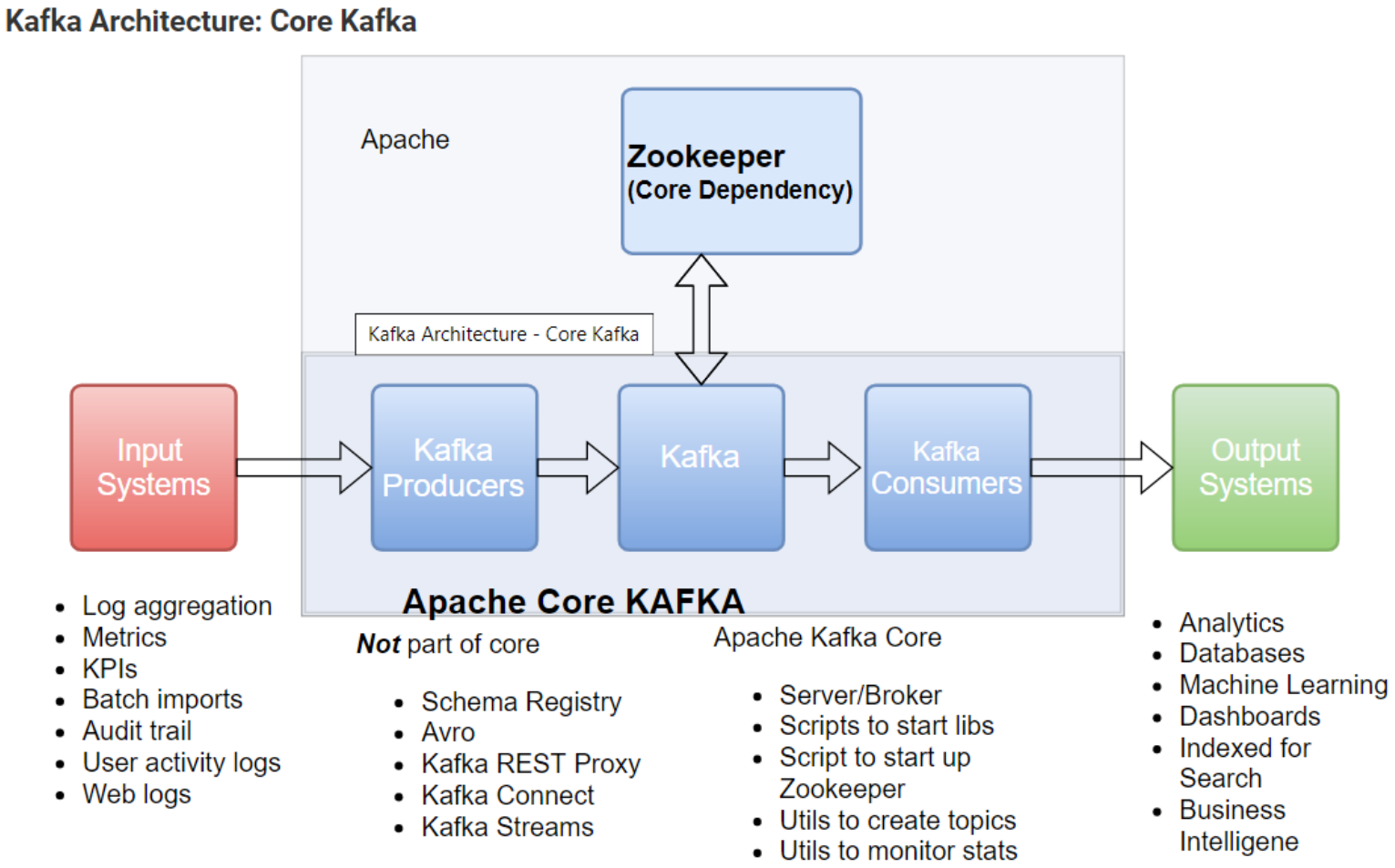
Real Time Transfer



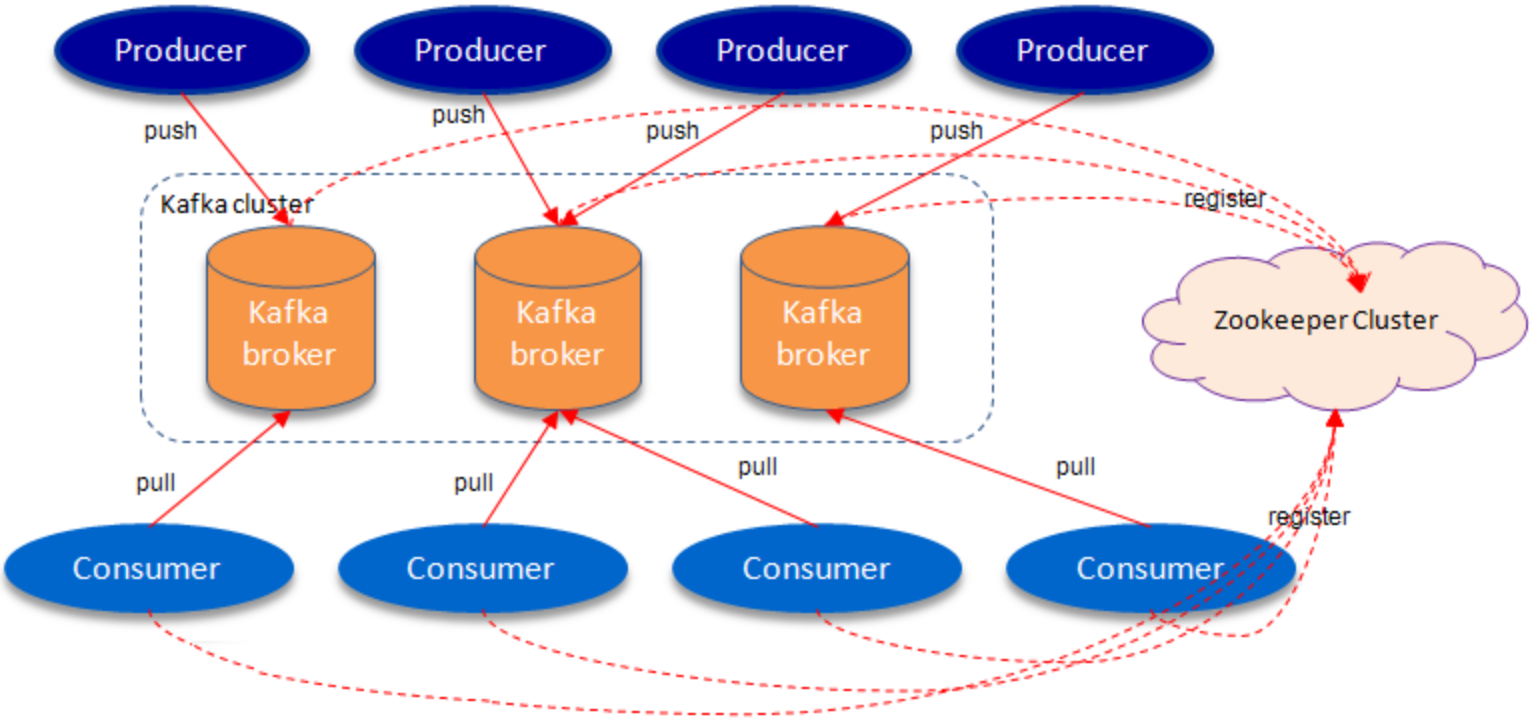
ZooKeeper in Kafka

- Kafka needs ZooKeeper
- Kafka uses Zookeeper to do leadership election of Kafka Broker and Topic Partition pairs.
- Kafka uses Zookeeper to manage service discovery for Kafka Brokers that form the cluster.
- Zookeeper sends changes of the topology to Kafka, so each node in the cluster knows when a new broker joined, a Broker died, a topic was removed or a topic was added, etc.
- Zookeeper provides an in-sync view of Kafka Cluster configuration.

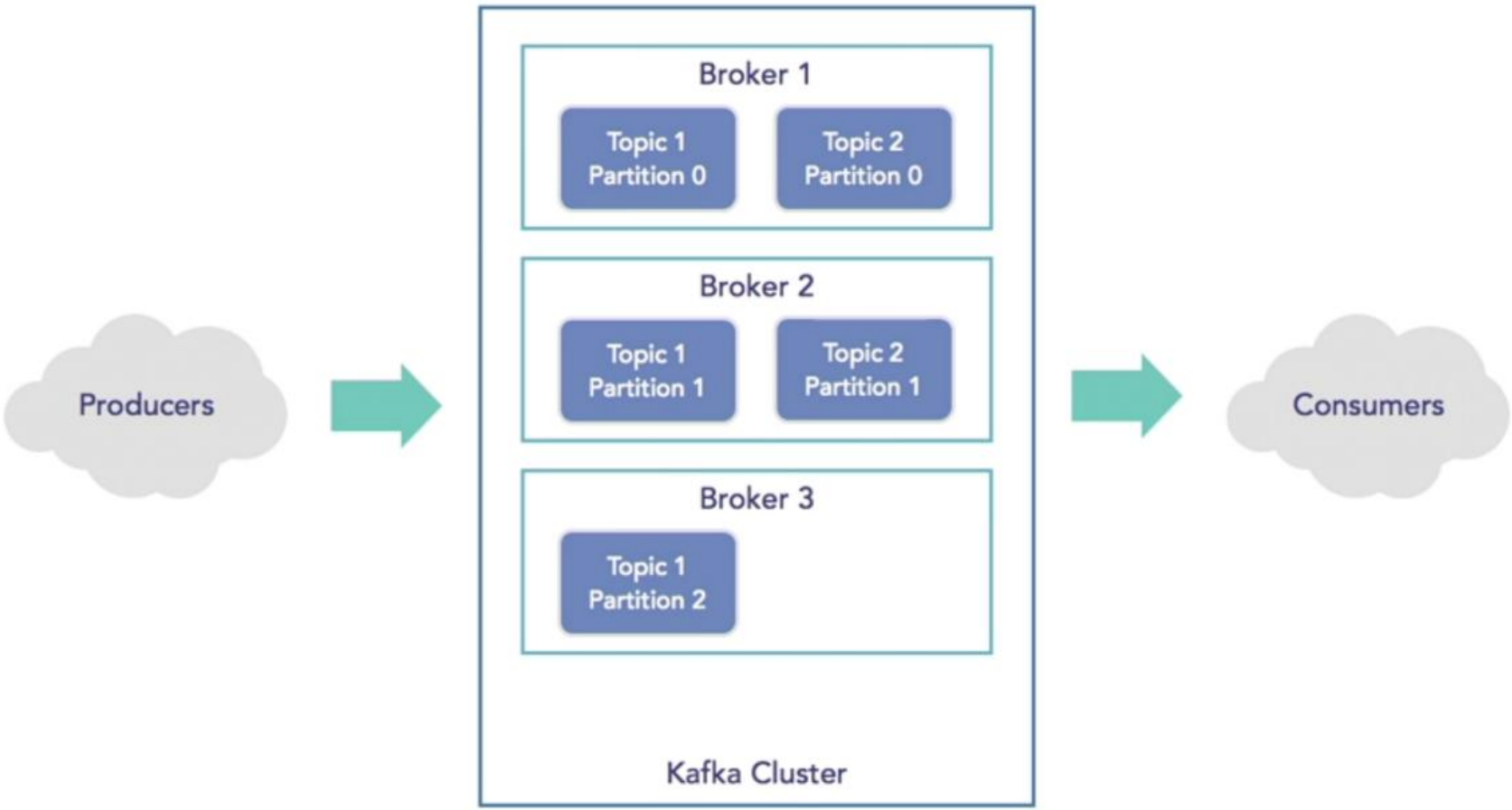
Kafka Architecture



Kafka architecture



Kafka architecture



Go to E:\software\A08\file\apache-zookeeper-3.5.6-bin.tar\apache-zookeeper-3.5.6-bin\apache-zookeeper-3.5.6-bin\conf

Zkserver from command prompt

```
2019-11-03 11:48:12,127 [myid:] - WARN  [NIOWorkerThread-1:NIOServerCnxn@370] - Exception
 causing close of session 0x0: null
2019-11-03 11:48:12,992 [myid:] - WARN  [NIOWorkerThread-2:NIOServerCnxn@370] - Exception
 causing close of session 0x0: null
2019-11-03 11:48:13,906 [myid:] - WARN  [NIOWorkerThread-4:NIOServerCnxn@370] - Exception
 causing close of session 0x0: null
2019-11-03 11:48:15,023 [myid:] - WARN  [NIOWorkerThread-6:NIOServerCnxn@370] - Exception
 causing close of session 0x0: null
2019-11-03 11:48:16,051 [myid:] - WARN  [NIOWorkerThread-9:NIOServerCnxn@370] - Exception
 causing close of session 0x0: null
2019-11-03 11:48:17,642 [myid:] - WARN  [NIOWorkerThread-13:NIOServerCnxn@370] - Exceptio
n causing close of session 0x0: null
2019-11-03 11:48:18,605 [myid:] - WARN  [NIOWorkerThread-12:NIOServerCnxn@370] - Exceptio
n causing close of session 0x0: null
2019-11-03 11:48:19,724 [myid:] - WARN  [NIOWorkerThread-16:NIOServerCnxn@370] - Exceptio
n causing close of session 0x0: null
2019-11-03 11:48:20,792 [myid:] - WARN  [NIOWorkerThread-1:NIOServerCnxn@370] - Exception
 causing close of session 0x0: null
2019-11-03 11:48:21,807 [myid:] - WARN  [NIOWorkerThread-5:NIOServerCnxn@370] - Exception
 causing close of session 0x0: null
```

C:\Windows\System32\cmd.exe - .\bin\windows\kafka-server-start.bat .\config\server.properties

```
tention.ms -> 86400000, cleanup.policy -> [delete], flush.ms -> 9223372036854775807, segm
ent.ms -> 604800000, segment.bytes -> 1073741824, retention.ms -> 604800000, message.time
stamp.difference.max.ms -> 9223372036854775807, segment.index.bytes -> 10485760, flush.me
ssages -> 9223372036854775807}. (kafka.log.LogManager)
[2019-11-03 11:43:35,072] INFO [Partition test11-0 broker=0] No checkpointed highwatermar
k is found for partition test11-0 (kafka.cluster.Partition)
[2019-11-03 11:43:35,073] INFO Replica loaded for partition test11-0 with initial high wa
termark 0 (kafka.cluster.Replica)
[2019-11-03 11:43:35,074] INFO [Partition test11-0 broker=0] test11-0 starts at Leader Ep
och 0 from offset 0. Previous Leader Epoch was: -1 (kafka.cluster.Partition)
[2019-11-03 11:48:44,857] INFO [GroupCoordinator 0]: Preparing to rebalance group console
-consumer-8213 in state PreparingRebalance with old generation 0 (__consumer_offsets-33)
(reason: Adding new member consumer-1-217673c0-2a77-44ea-b5dc-5b486eefded2) (kafka.coordi
nator.group.GroupCoordinator)
[2019-11-03 11:48:44,868] INFO [GroupCoordinator 0]: Stabilized group console-consumer-82
13 generation 1 (__consumer_offsets-33) (kafka.coordinator.group.GroupCoordinator)
[2019-11-03 11:48:44,882] INFO [GroupCoordinator 0]: Assignment received from leader for
group console-consumer-8213 for generation 1 (kafka.coordinator.group.GroupCoordinator)
[2019-11-03 11:51:33,455] INFO [GroupMetadataManager brokerId=0] Removed 0 expired offset
s in 3 milliseconds. (kafka.coordinator.group.GroupMetadataManager)
```

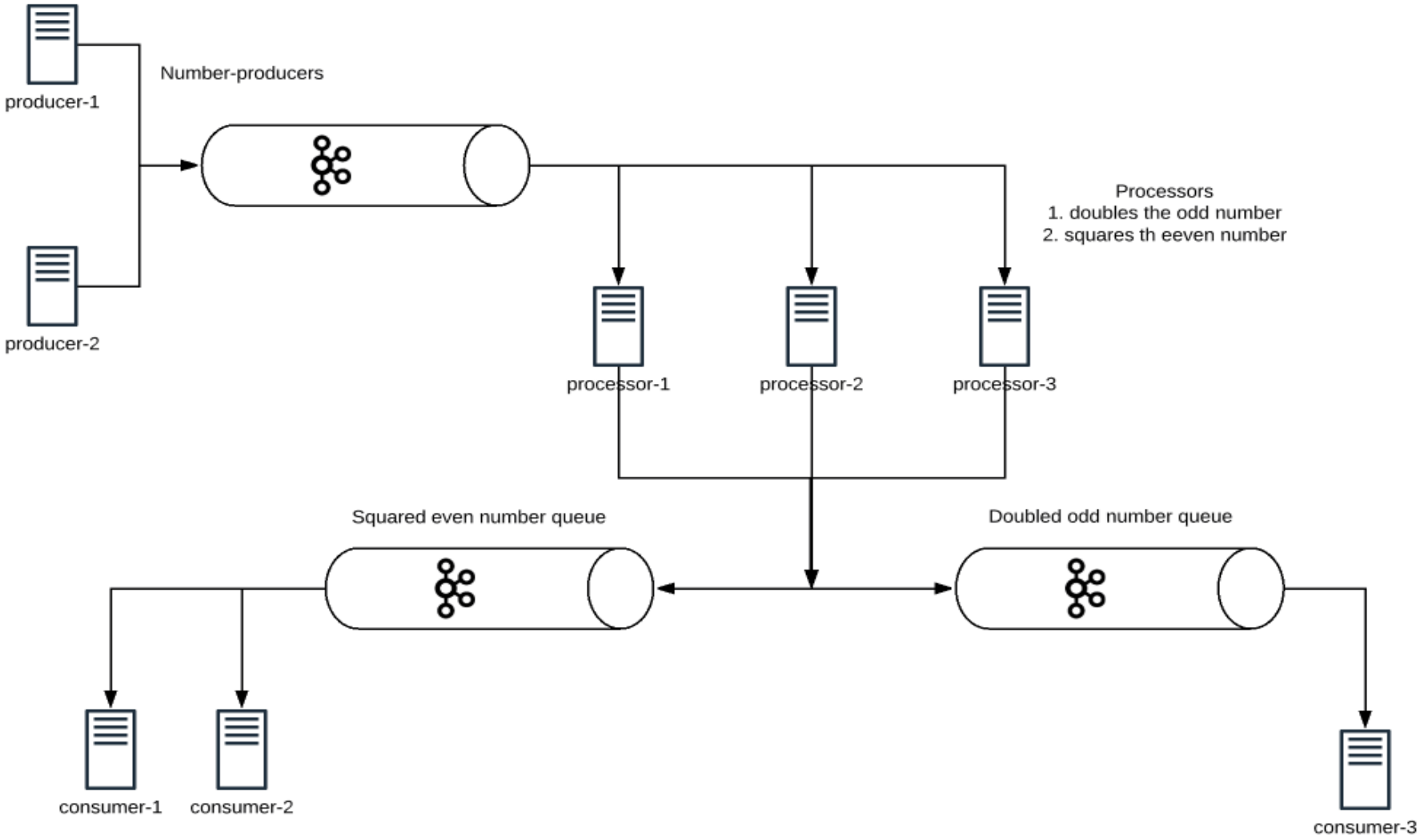
Administrator: RabbitMQ Command Prompt (sbin dir) - kafka-console-producer.bat --broker-list localhost:9092 --topic test

```
[2019-11-03 11:43:19,657] ERROR org.apache.kafka.common.errors.TopicExistsException: Topic 'test' already exists.
 (kafka.admin.TopicCommand$)

E:\software\A08\file\kafka\bin\windows>kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --p
artitions 1 --topic test11
Created topic test11.

E:\software\A08\file\kafka\bin\windows>kafka-console-producer.bat --broker-list localhost:9092 --topic test
>hi
>how are yoy
>enjoy
>
```

Administrator: RabbitMQ Command Prompt (sbin dir) - kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic test --from-beginning

```
                                        consumption.

E:\software\A08\file\kafka\bin\windows>kafka-console-consumer.bat --bootstrap-server localhost:2181 --topic test
Processed a total of 0 messages
Terminate batch job (Y/N)? y

E:\software\A08\file\kafka\bin\windows>kafka-console-consumer.bat --bootstrap-server localhost:2181 --topic test
Processed a total of 0 messages
Terminate batch job (Y/N)? y

E:\software\A08\file\kafka\bin\windows>kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic test --fro
m-beginning
hi
how are yoy
enjoy
```

Process the incoming messages, filter and split into multiple topics based on certain conditions.



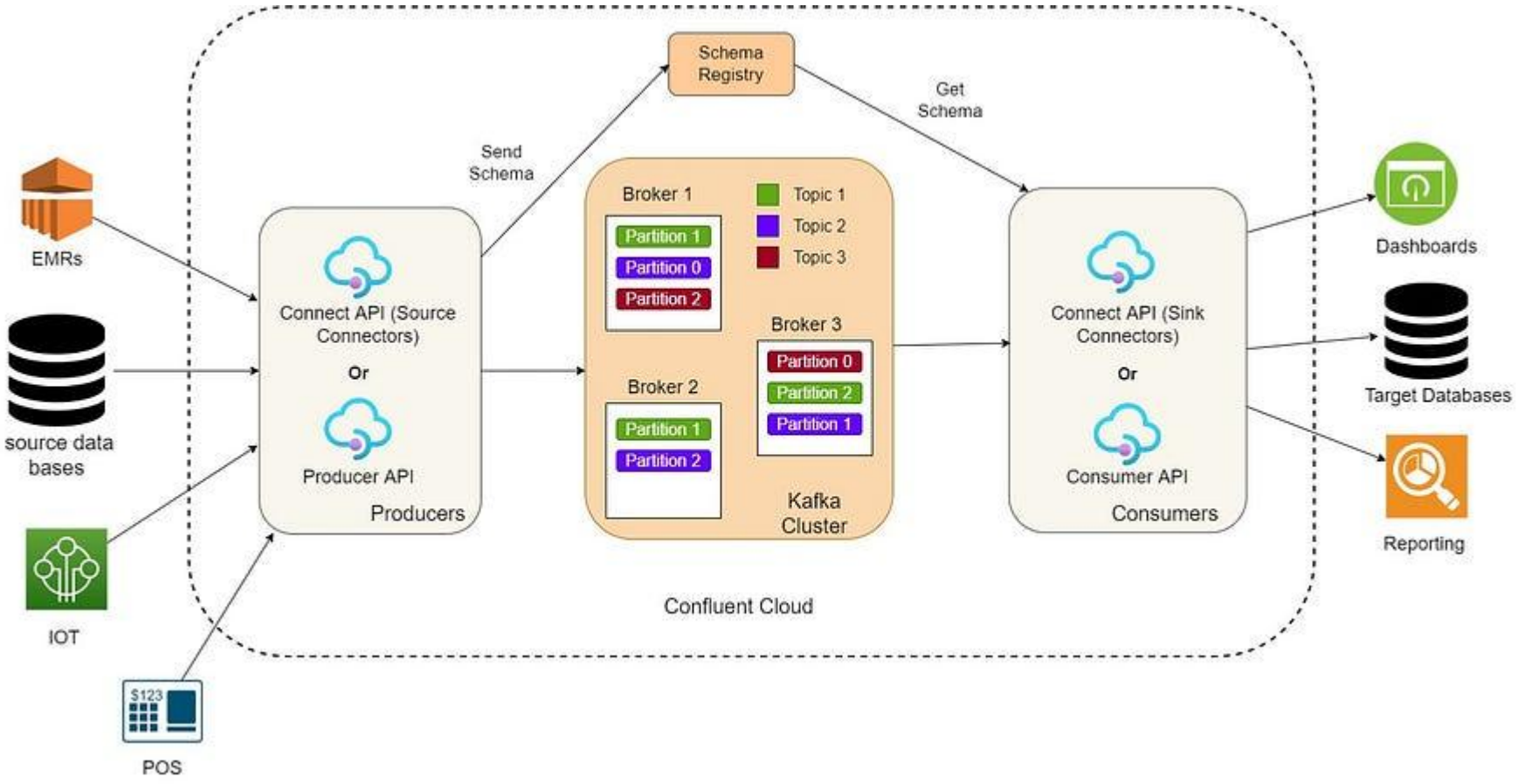
Kafka vs RabbitMQ and Others

Category	Kafka	RabbitMQ	Amazon SQS	Google Pub/Sub
Message Rate (msgs/sec)	100k+	20k+	Varies (can be scaled)	10k+
Message Acknowledgements	No	Yes	Yes	Yes
Built in UI	No	Yes	Yes	Yes
Protocol	Kafka	AMQP	HTTP REST	HTTP REST
Additional features	Apache Storm, etc	Several plugins available to add more features	In-flight messages	-
Use cases	High ingestion platforms where speed,scale and efficiency is the prime concern	Robust systems where features like acknowled-gements, deeper management are important	Useful when deploying applications on AWS EC2, Elastic Beanstalk to facilitate low-latency communication	Works well with applications deployed on GCE

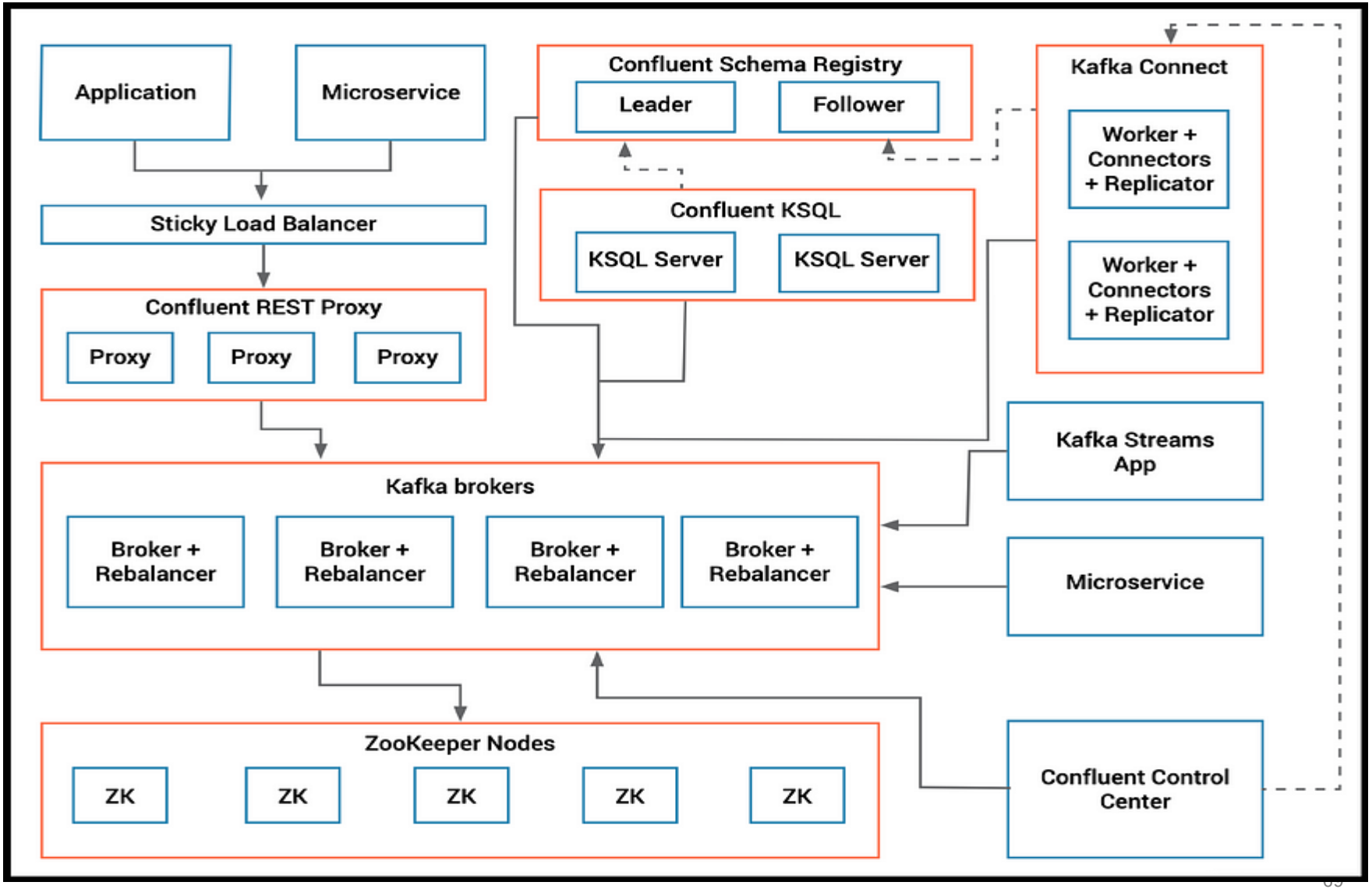
Confluent Kafka

- A commercial distribution of Apache Kafka developed by Confluent Inc. (the company founded by the original creators of Kafka).
- Confluent Kafka extends the core Kafka functionality with additional enterprise-grade features and managed services.

Confluent Kafka Architecture



Confluent Kafka Architecture



1. ZooKeeper:

- Manages distributed processes, cluster memberships, and elects cluster controllers.
- High availability requires 5 Zookeeper nodes (2 for failure node)
- Number of ZooKeeper nodes must be odd.

2. Kafka Brokers:

- Main storage, messaging components, and maintains streams of ordered logs of messages in Topics.
- Contains clusters that maintain streams of messages called Topics.
- Topics are shared into partitions of ordered immutable logs of messages.
- Partitions are replicated and distributed for high availability.
- Need 4 Kafka Brokers in a cluster each replicating partition on a separate server/node.
- Highly loaded Brokers should have their own nodes otherwise they can share Zookeepers but separate disks/controllers for Zookeepers.

3. Kafka Connect Workers:

- Stateless Workers integrate external systems to pull/push data on source/target.
- Pluggable JDBC Connectors in Avro or JSON format to Azure Data Lake Storage Gen2 Sink.
- The Kafka Connect Snowflake Sink connector maps and persists events from Topics directly exposing the data to services for querying, enrichment, and analytics.

3. Kafka Connect Workers:

- ·***Standalone Mode*** — File or Spooling Directory Connector on the Producer machine to read the file and send events to Kafka.
- ***Cluster Mode*** –Deployment on multiple machines discover each other and Brokers serving as synchronization layer automatically load-balance and failover work between them.
- Manage Connectors on cluster using Connect Node (aka. “Worker”) uses RESP API to configure, start, stop, pause and resume. Once connector starts regardless of node, it will load balance parallel tasks to the least loaded available worker node.
- Because they are stateless, they can be deployed in containers.

4. Kafka Clients

- Java client JARs interfaces applications to Kafka.

- · Are installed along Kafka Brokers but deployed with application client libraries.

- · APIs and semantics in variety programming languages (C, C++, Python, and Go).

5. Kafka Streams APIs

- Application embedded library for building distributed stream processing applications for late-arriving data.
- Benefits from higher cpu/core count.
- Deploying multiple instances of application on multiple servers Kafka Sreams will auto load-balance and fail-over.

6. ksqlDB Server

- SQL engine that queries continuously against Apache Kafka in real-time.
- Command Line Interface allows interactive querries to ksqlDB server from any machine.
- ksqlDB server includes data processing to the target Kafka cluster.
- ksqlDB is deployed on a set of servers/nodes that form a cluster determined by processing capacity required (concurrent queries/complexity) handling auto load-balancing/fail-over.
- Benefits from higher CPU counts, networking throughput and SSD storage.

7. Confluent REST Proxy

- HTTP server that provides RESTful interface to Kafka cluster.
- Mandatory component when using RESTful HTTP protocol on separate machine/node.
- If application uses Native Kafka Client then no need to deploy REST Proxy.
- Deploy multiple REST Proxies behind a sticky load-balancer for the same Consumer.

8. Confluent Schema Registry

- Metadata layer using RESTful interface for storing and retrieving Avro schemas.
- Stores versioned history of all schemas and allows compatibility settings.
- Includes Message Serializers that plug into Kafka clients
- Handles Schema Storage and retrieval of Kafka messages from Avro format.
- Installed on its own server/node. But small installations can be alongside REST Proxy and Connect Workers.

8. Confluent Schema Registry

- High availability requires multiple Schema Registry servers/nodes.
- Multi Schema Registry uses a leader-followers architecture with at most 1 node being Leader any given time.
- Only Leader can publish writes to Kafka Log where all nodes can process read requests.
- Follower nodes forward write requests to Leader.
- Schema Registry stores its schemas in Kafka and do not require storage and can be deployed in containers.

9. Confluent Replicator

- Manages multi-cluster deployments of Kafka.
- Provides centralized configuration of cross-cluster replication.
- Replicates Topic configuration in addition to Topic Messages.
- Integrated with Kafka Connect framework and be installed in Connect Nodes in destination cluster. For Multiple Connect Workers the Replicator should be installed in all of the Connect Nodes.
- Large number of Replicator Nodes will have high availability with built-in fail-over.

10. Confluent Auto Data Balancing

- Optimizes resource utilization to scale Kafka Clusters.
- Installed on any machine in the Confluent Platform cluster to communicate to Brokers and ZooKeeper but recommended install on the Brokers or on Control Center Node.
- Collects load metrics and sends instructions to move partitions.

11. Confluent Control Center

- Web based tool for managing and monitoring data pipelines and streaming applications on Apache Kafka.
- Data Stream Monitoring and Alerting using drill-down from Producer to Consumer.
- Multi-cluster monitoring and management data replication between clusters.
- Kafka Connect Configuration to add new sources to load external data systems.
- Runs on a single dedicated machine.

Confluent Kafka

Feature	Description
Confluent Control Center	GUI for **monitoring, managing, and troubleshooting** Kafka clusters.
Schema Registry	Helps enforce **schema evolution** and compatibility for message formats (e.g., Avro, JSON, Protobuf).
Kafka Connect	A library for **connecting Kafka to external systems** using pre-built connectors (e.g., databases, S3).
ksqlDB	Allows real-time **stream processing using SQL-like syntax** without writing Java code.
Enhanced Security	Includes **RBAC (Role-Based Access Control)**, **OAuth**, encryption, and **audit logging**.
Multi-Region Clustering	Enables Kafka replication and failover across geographic locations.
Managed Services (Cloud Offering)	Fully managed Kafka clusters on **AWS, Azure, or GCP** via Confluent Cloud.
Tiered Storage	Moves cold data to **cost-effective cloud storage**, reducing storage costs.

Key Differences

Aspect	Apache Kafka	Confluent Kafka
Open-source License	Apache 2.0 License	Open-core: Apache Kafka + proprietary features under Confluent Community License
Ease of Use	Requires manual installation, scaling, and config	Confluent simplifies deployment, management, and scaling (especially for enterprise).
Schema Management	No native schema enforcement	Built-in **Schema Registry** for enforcing data format consistency.
Connectors	Limited (need custom or open-source plugins)	Large set of **pre-built connectors** for popular data sources and sinks.
Monitoring & Management	Limited (needs custom tools like Prometheus)	Confluent provides **Control Center** for full visibility.
Security	Basic (SASL, SSL, ACL)	Enterprise-grade security features such as **RBAC, audit logs**, and OAuth integration.
Cloud-Native/Managed Option	No official support (but cloud vendors offer it)	Fully managed **Confluent Cloud** on AWS, GCP, and Azure.
Cost	Free (self-managed)	Paid plans for enterprise features (on-prem and cloud).

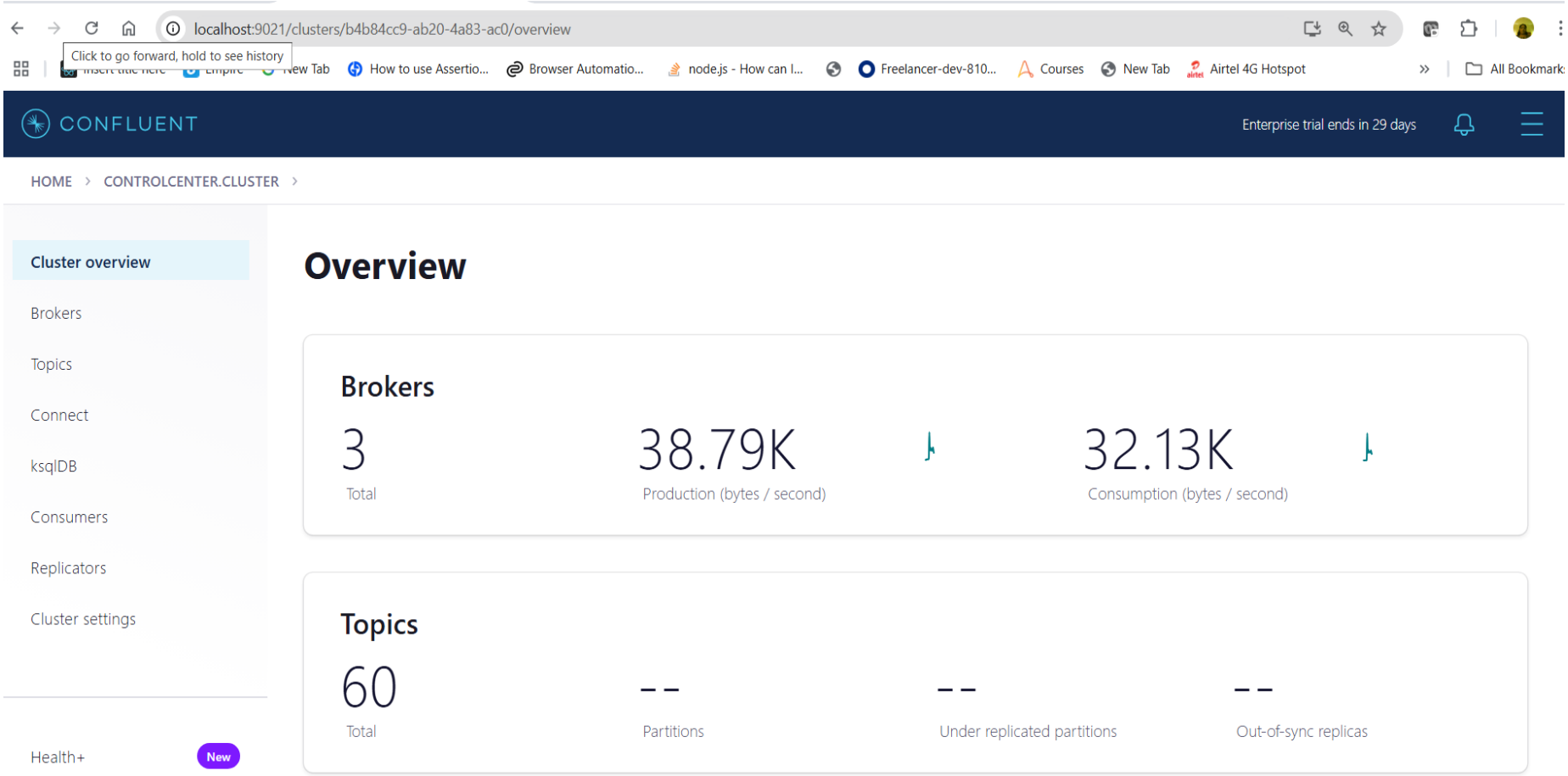
Key Differences

Use Case	Recommended Option
Small-scale, on-prem, or non-critical workloads	**Apache Kafka**
Enterprise use cases needing built-in security, monitoring, and connectors	**Confluent Kafka**
Want to minimize infrastructure management	**Confluent Cloud**
Simple development/testing environments	**Apache Kafka**
Real-time stream processing without coding	**Confluent Kafka (ksqlDB)**

Confluent Kafka

- Refer F:\Local disk\Docker\kafkahelm\helm

Confluent Kafka



Quick Summary of Options

Option	When to Use
New consumer group	If you want a fresh start without affecting previous offsets.
Seek to the beginning	If you want to force consumption of all messages without changing the group ID.
Delete committed offsets	If you want the current consumer group to reprocess all messages.
Manual partition assignment	If you want complete control over message consumption and offset management.

Error Handling Scenarios

Scenario	Solution
Transient network errors	Use Kafka's built-in retry mechanism with `Retries` and `RetryBackoffMs`.
Message processing errors	Implement manual retry logic as shown in the consumer example.
Brokers unavailable	Set `MessageTimeoutMs` and handle retries in the producer.
Duplicate messages	Use `EnableIdempotence` on the producer to avoid sending duplicates during retries.
Poison messages	Log unprocessable messages to a dead-letter queue (or external system) after retries fail.

Recommended Producer and Consumer Configurations

Parameter	Recommended Value	Description
`Acks`	`All`	Wait for acknowledgment from all brokers.
`Retries`	5 or higher	Number of retries on transient errors.
`EnableIdempotence`	`true`	Ensure exactly-once delivery by preventing duplicate messages.
`RetryBackoffMs`	1000 or higher	Backoff time between retries.

Consumer Configuration

Parameter	Recommended Value	Description
`EnableAutoCommit`	`false`	Manually commit offsets to handle retries properly.
`AutoOffsetReset`	`Earliest`	Start reading from the beginning when no offset is committed.
`MaxPollIntervalMs`	Adjust based on workload	Maximum time between polls to prevent consumer rebalancing during processing.

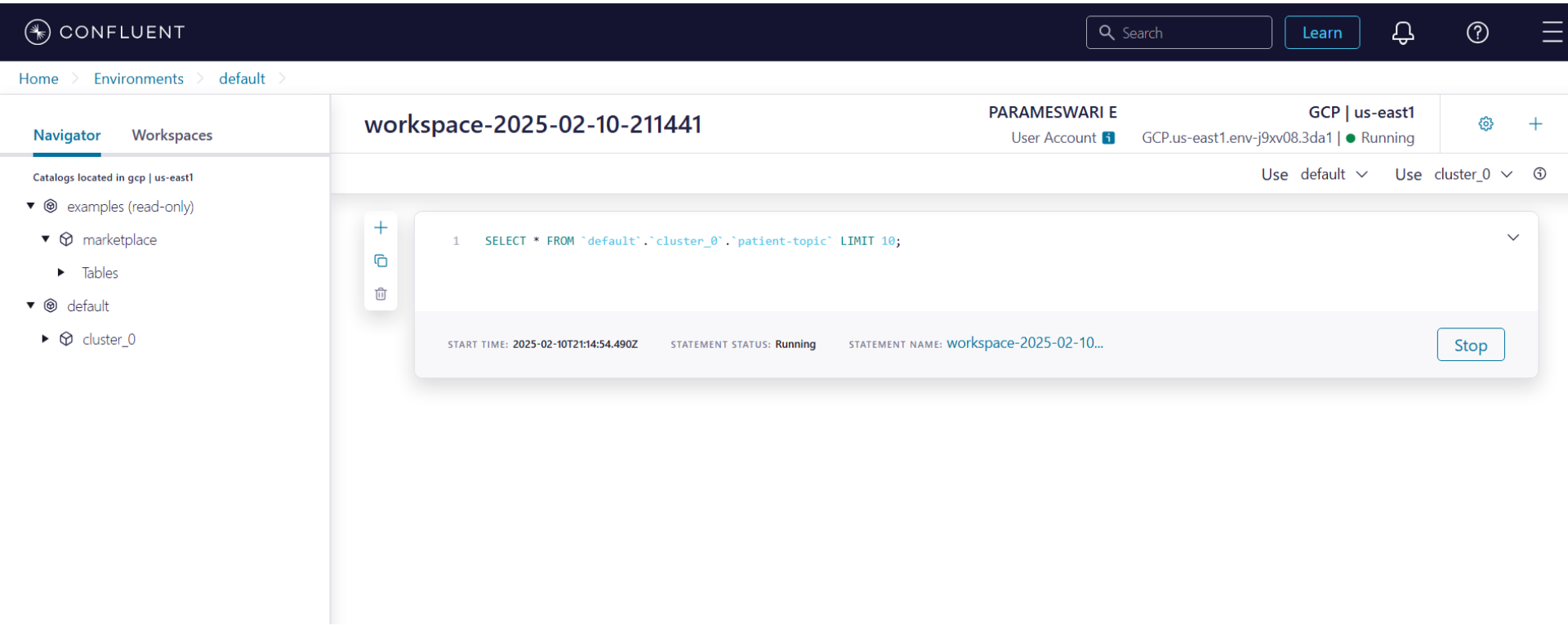
Transactional Messages

- Transactional messaging in Kafka ensures that messages are atomically produced to multiple topics or partitions and consumed exactly once.
- This is crucial when you have scenarios involving:
  - Chained processing where data needs to be produced/consumed consistently across multiple topics.
  - Distributed transactions requiring atomicity.
  - Exactly-once processing semantics where messages must not be duplicated or lost.

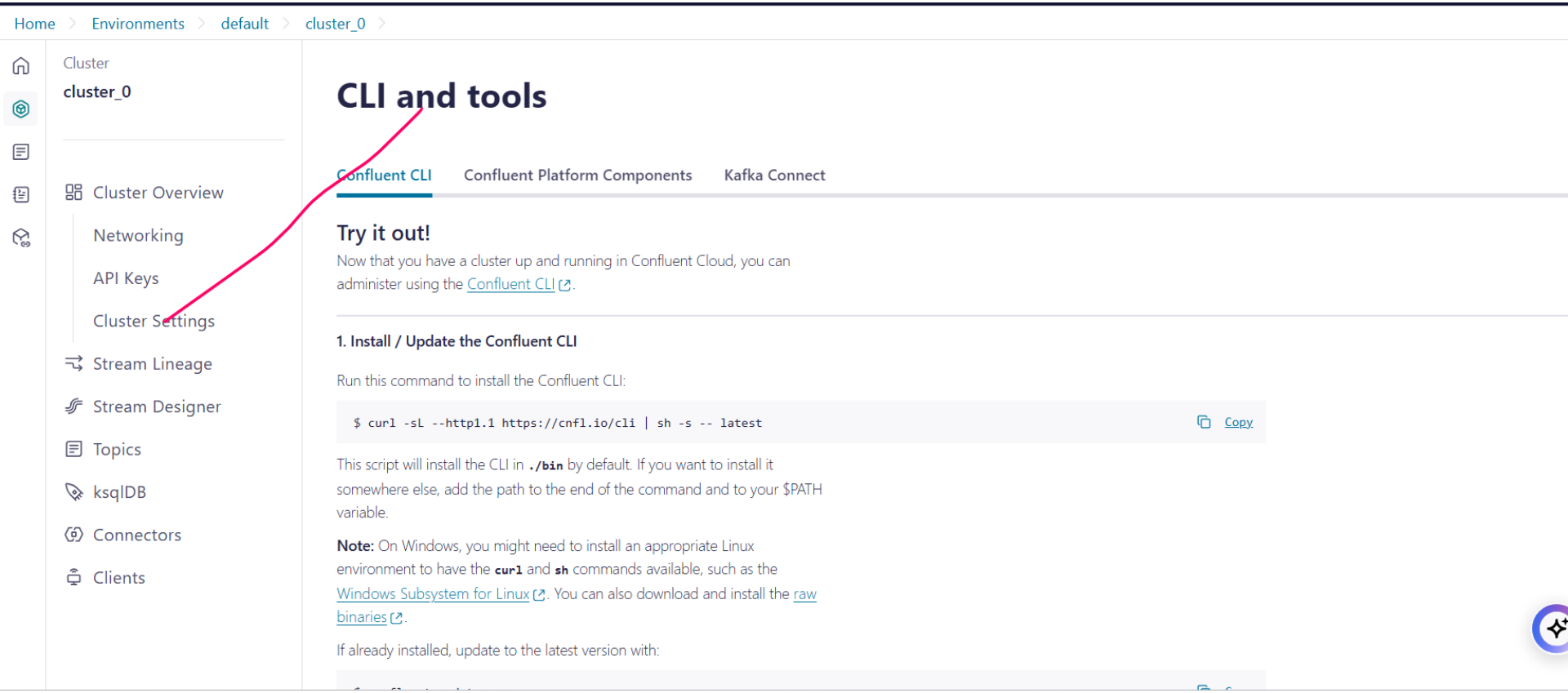
Key Concepts

- Transactional Producer: Guarantees atomic writes of a group of messages to multiple Kafka partitions.
- Transaction ID: The producer uses this ID to identify its ongoing transaction and recover after failures.
- Exactly-once Semantics (EOS): The combination of transactional producers and idempotent writes guarantees that each message is processed exactly once.

Confluent Kafka Flink Query

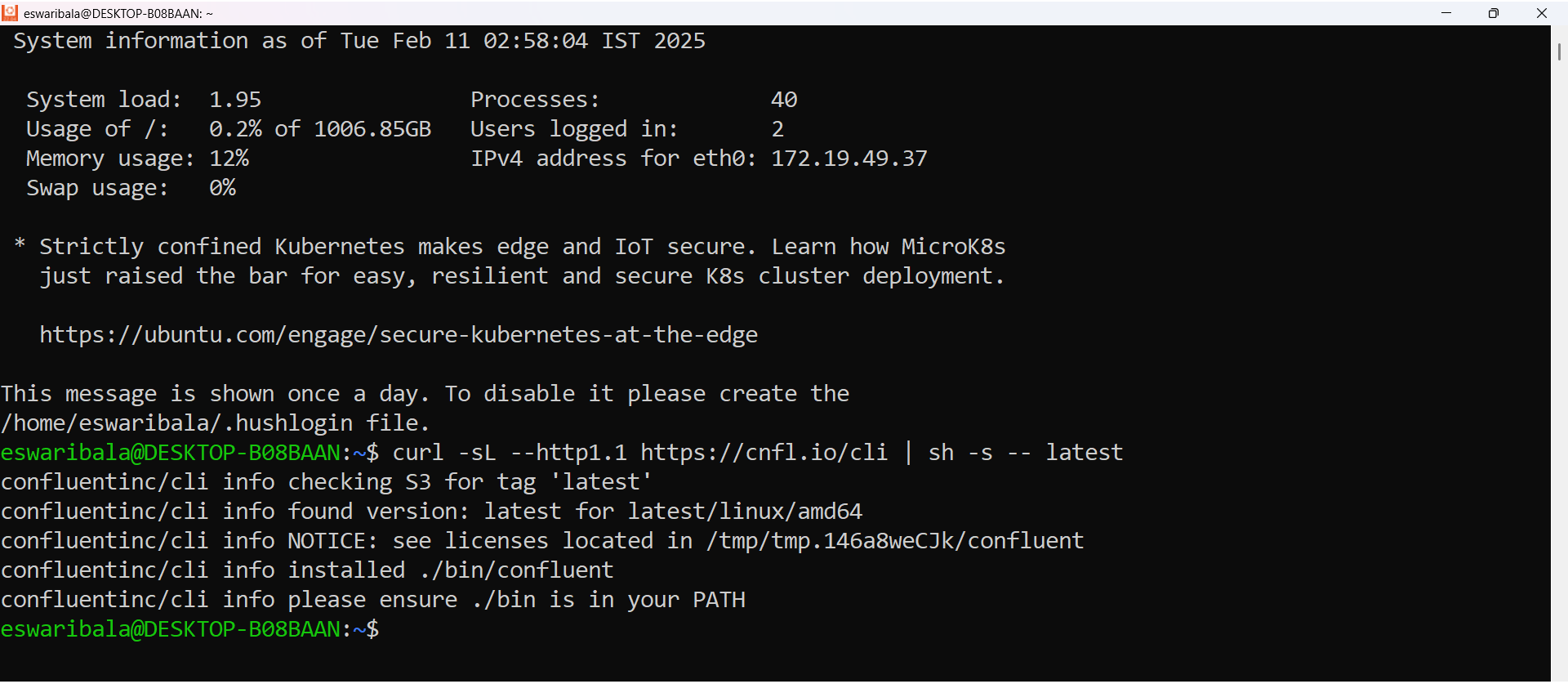


Confluent CLI under Cluster Settings (scroll down)

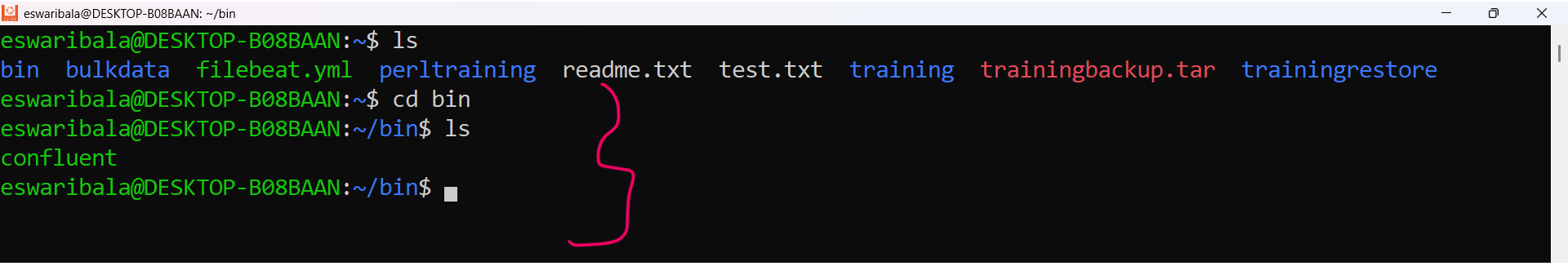


Confluent CLI using Ubuntu

- curl -sL --http1.1 https://cnfl.io/cli | sh -s -- latest



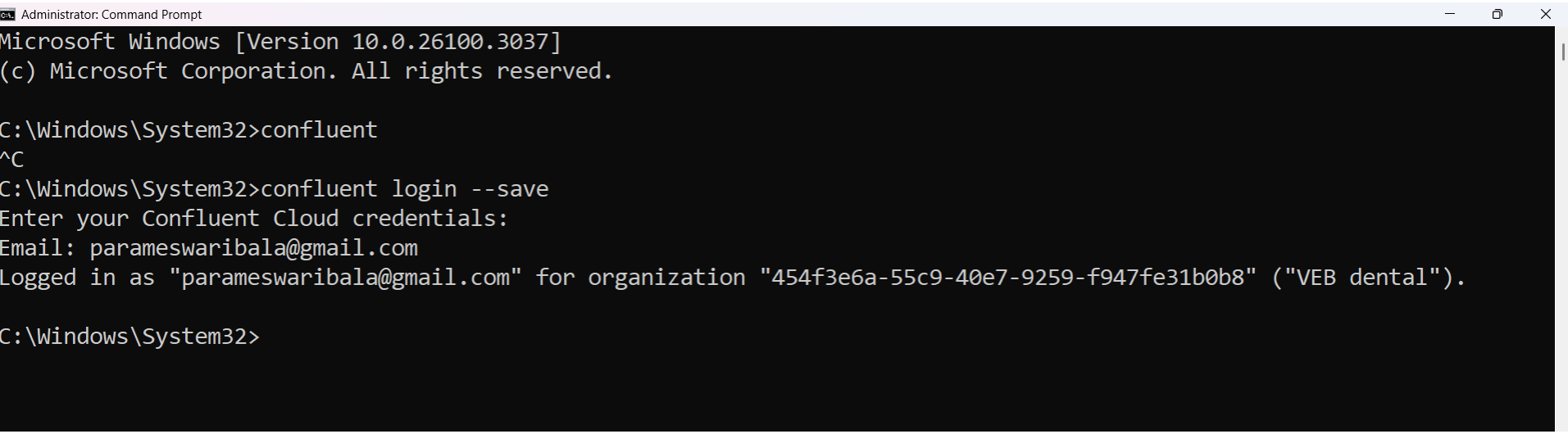
Confluent CLI using Ubuntu



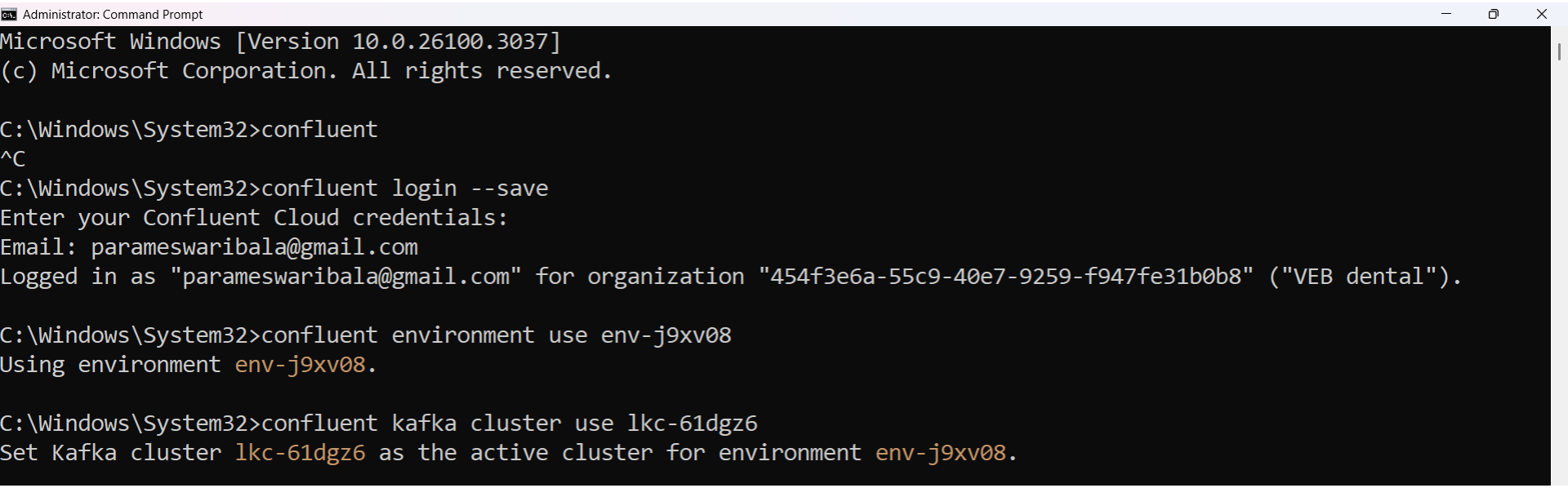
Confluent CLI using Windows

- https://github.com/confluentinc/cli/releases/tag/v4.17.0

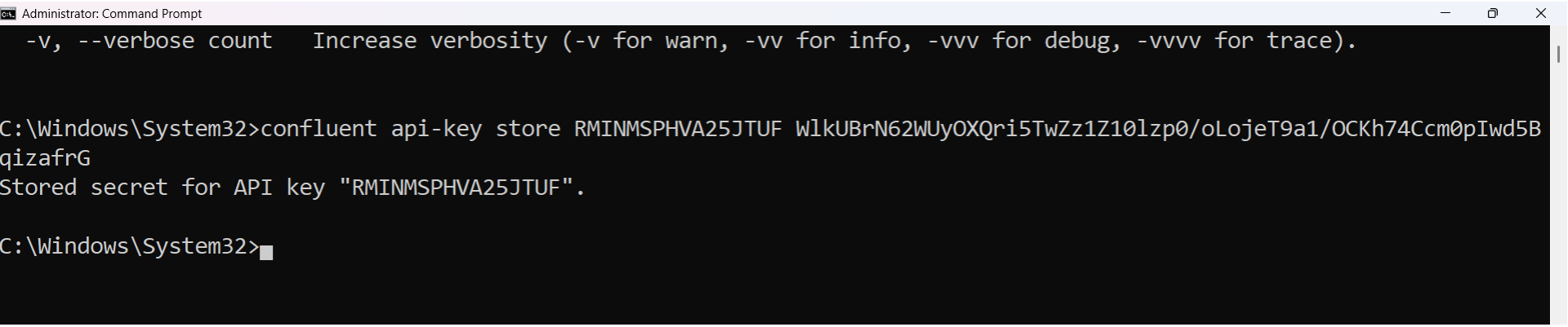
- Download zip file and env variable



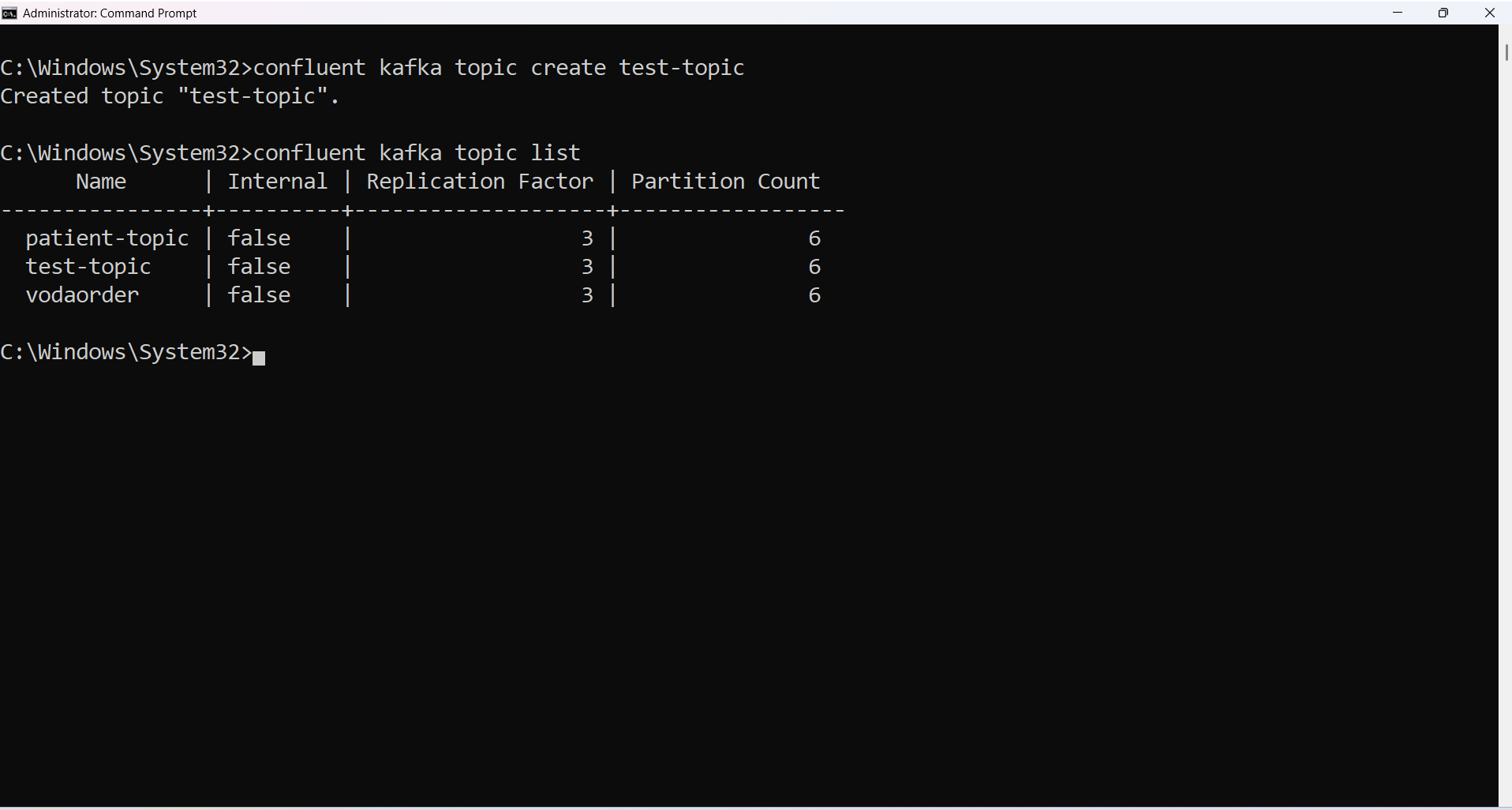
Confluent CLI using Windows



Confluent CLI using Windows



Confluent CLI using Windows



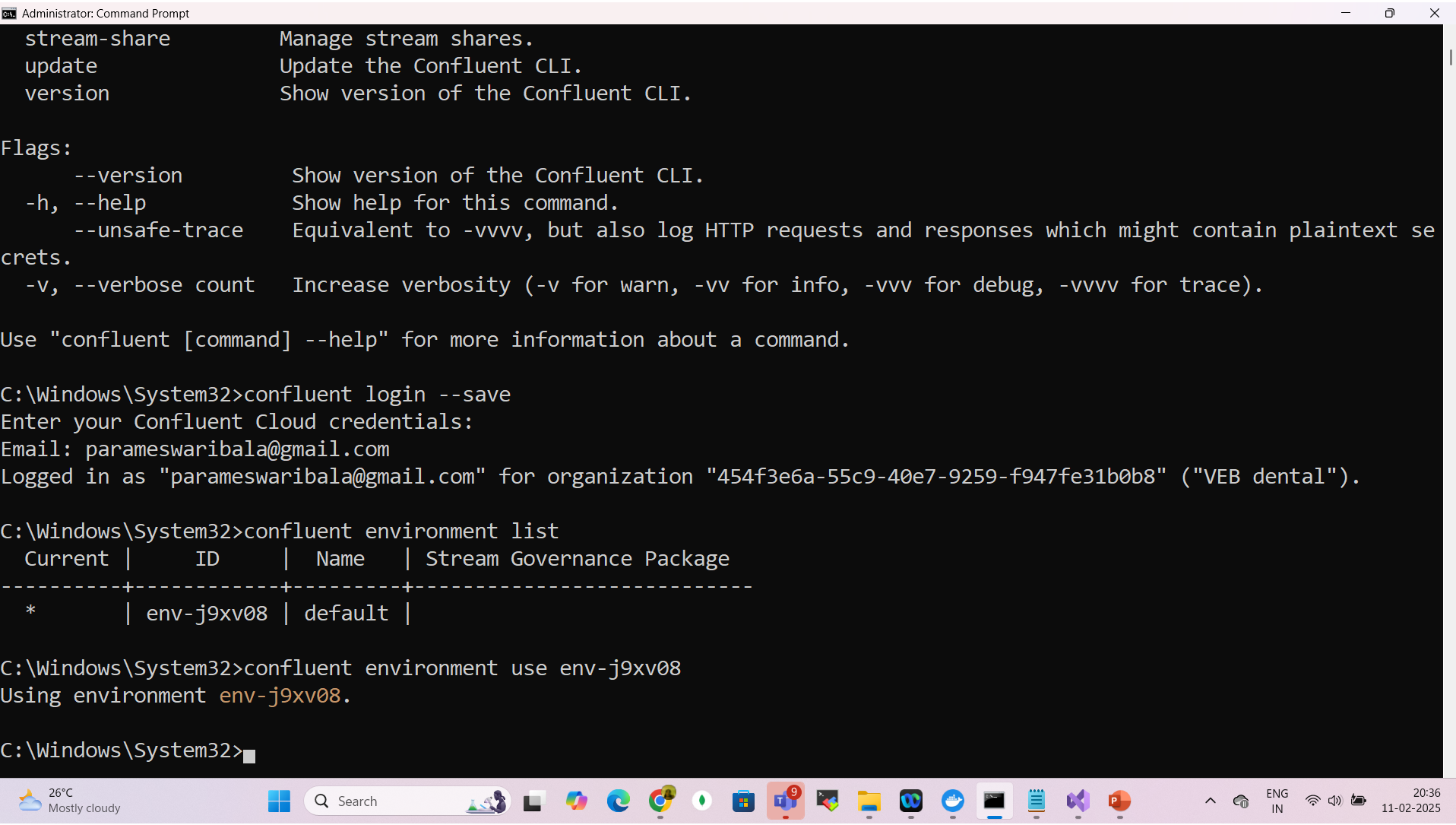
Confluent CLI using Windows

- 3. Select your environment
- Run this command to set your environment. To view the available environments, use the confluent environment list command.
- confluent environment use env-j9xv08

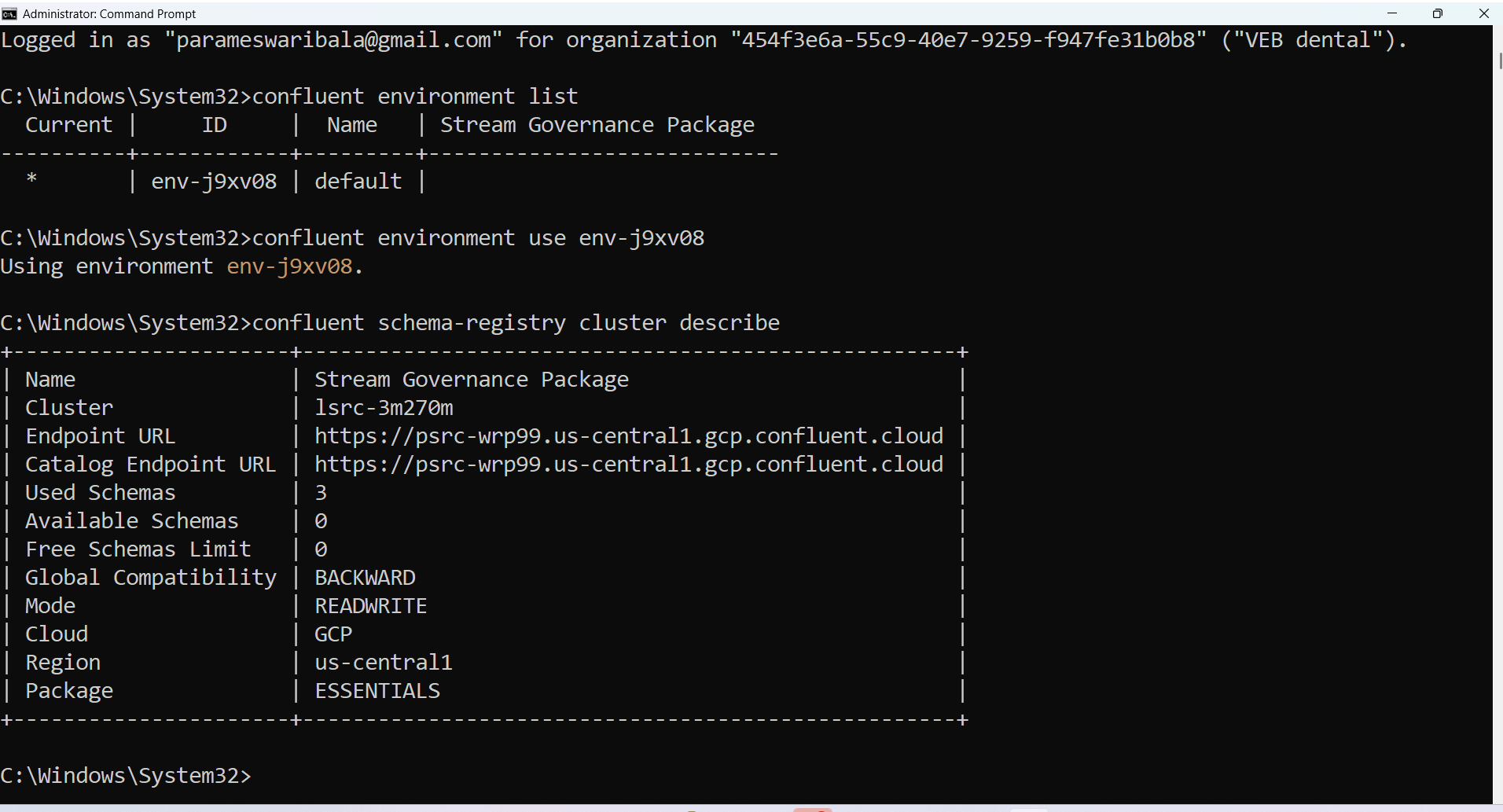
Confluent CLI using Windows

- 4. Select your cluster
- Run this command to set your cluster. To view the available clusters, use the confluent kafka cluster list command.
- confluent kafka cluster use lkc-61dgz6

Confluent CLI using Windows



Confluent CLI using Windows



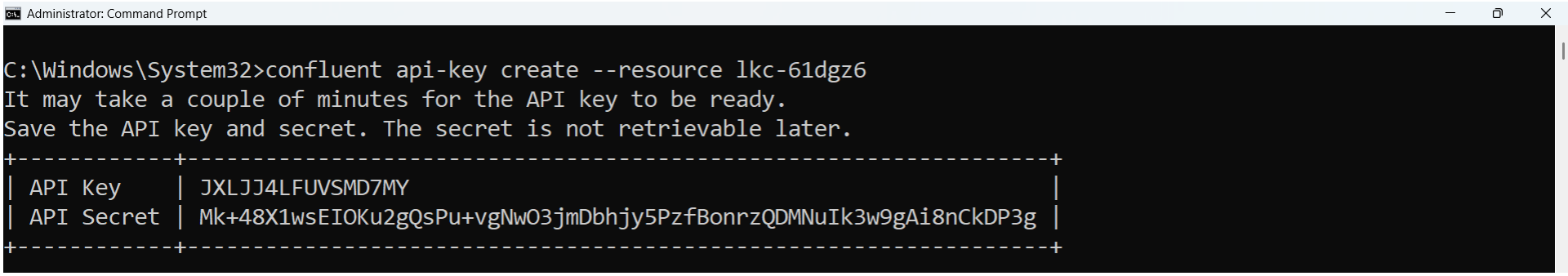
Confluent CLI using Windows

confluent api-key create --resource lsrc-3m270m

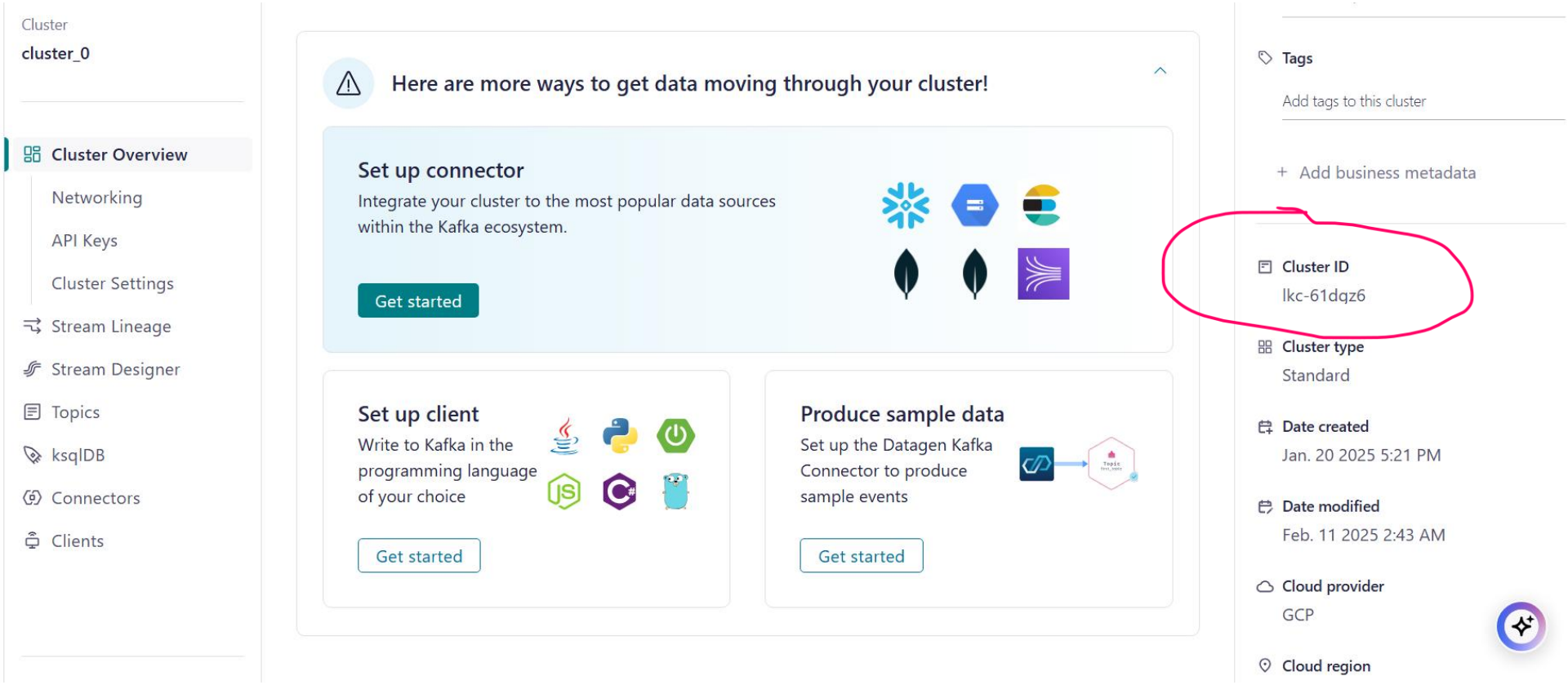
```
C:\Windows\System32>confluent api-key create --resource lsrc-3m270m
It may take a couple of minutes for the API key to be ready.
Save the API key and secret. The secret is not retrievable later.
+------------+------------------------------------------------------------------+
| API Key    | 3BFMWQFZ5OXHUDLQ                                                 |
| API Secret | 6D3IyPRg4kA2SmFJRR/CFqVkF3fd6CMVAVvTS8w7EK33nvYLJ/Uc9MJIptWhuBDC |
+------------+------------------------------------------------------------------+

C:\Windows\System32>
```

Confluent CLI using Windows

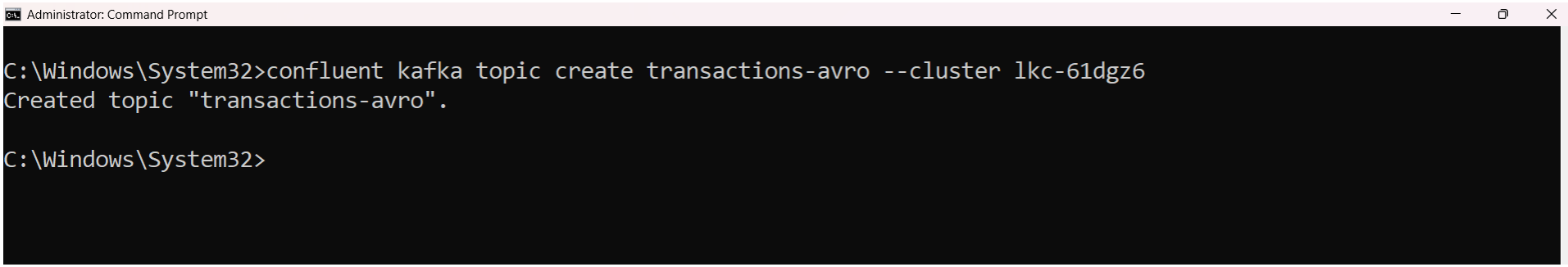
confluent api-key create --resource lkc-61dgz6



Confluent CLI using Windows



Confluent CLI using Windows



AVRO Message

- **Apache Avro** is a data serialization framework that helps serialize structured data efficiently. When producing messages to Kafka, **Avro messages** provide:

1. **Compact and efficient serialization:** Avro uses binary encoding, making messages small and efficient for transport.
2. **Schema-based serialization:** Avro relies on schemas defined in **Avro IDL** or **JSON format** to ensure that data can be deserialized correctly by consumers.
3. **Forward and backward compatibility:** Because schemas are versioned, producers and consumers can evolve independently, making schema evolution easy to manage.

Schema Evolution and Compatibility

- One of Avro's strengths is its support for **schema evolution**. You can add or modify fields in the schema while maintaining compatibility between producers and consumers.
- **Types of Compatibility**
- **Backward compatibility:** Consumers using the old schema can still read messages produced with the new schema.
- **Forward compatibility:** Consumers using the new schema can read messages produced with the old schema.
- **Full compatibility:** Ensures both forward and backward compatibility.

Schema Evolution and Compatibility

Initial schema:

```json
{
  "type": "record",
  "name": "Transaction",
  "fields": [
    { "name": "id", "type": "string" },
    { "name": "amount", "type": "double" }
  ]
}
```

Schema Evolution and Compatibility

New schema (adding a new optional field):

json

```
{
  "type": "record",
  "name": "Transaction",
  "fields": [
    { "name": "id", "type": "string" },
    { "name": "amount", "type": "double" },
    { "name": "description", "type": ["null", "string"], "default": null }
  ]
}
```

Why Use Avro in Kafka?

kafka

- Compact binary format: Smaller messages reduce network overhead and storage costs.
- Schema evolution: Producers and consumers can have different schema versions without breaking compatibility.
- Schema registry integration: Confluent Kafka integrates Avro with its Schema Registry, where schemas are stored, validated, and version-controlled.

Avro Message Workflow in Kafka

1. **Define an Avro schema:** Define the structure of the message using the Avro schema language.
2. **Serialize the message using the schema:** The producer serializes the message using the Avro schema before sending it to Kafka.
3. **Store schema in Schema Registry:** The producer sends the schema to the **Confluent Schema Registry**, where it is assigned a schema ID.
4. **Send the message to Kafka:** The producer sends the message (along with the schema ID) to the Kafka topic.
5. **Consumer deserialization:** The consumer fetches the schema from the **Schema Registry** using the schema ID and deserializes the Avro message.

Avro Message Structure

- An Avro message typically consists of:

1. **Schema ID:** Stored in the header of the Kafka message, this identifies the schema version used to serialize the message.
2. **Serialized message data:** Binary-encoded data representing the Avro message fields.

Example: Avro Schema Definition

Schema for a `Transaction` message:

```json
{
  "type": "record",
  "name": "Transaction",
  "namespace": "com.example",
  "fields": [
    { "name": "id", "type": "string" },
    { "name": "amount", "type": "double" }
  ]
}
```

Avro Serialization Flow

- **Avro Serialization Flow**

1. **Producer Side:**
    1. When a producer sends an Avro message, the **AvroSerializer**:
        1. Serializes the message according to the schema.
        2. Stores the schema in the **Schema Registry** (if it's not already there).
        3. Adds the **schema ID** to the Kafka message headers.
2. **Kafka Broker:**
    1. The Kafka broker simply stores the Avro message as binary data without being aware of its structure.
3. **Consumer Side:**
    1. When a consumer reads an Avro message:
        1. It fetches the **schema ID** from the message header.
        2. Queries the **Schema Registry** to get the corresponding schema.
        3. Deserializes the binary Avro data back into the original object using the schema.

Confluent CLI using Windows (consumer)

- confluent kafka topic produce transactions-avro --cluster lkc-61dgz6 --schema "F:\Local disk\Docker\kafkahelm\helm\schema.txt" --schema-registry-endpoint "https://psrc-wrp99.us-central1.gcp.confluent.cloud" --schema-registry-api-key "3BFMWQFZ5OXHUDLQ" --schema-registry-api-secret "6D3IyPRg4kA2SmFJRR/CFqVkF3fd6CMVAVvTS8w7EK33nvYLJ/Uc9MJIptWhuBDC" --api-key "JXLJJ4LFUVSMD7MY" --api-secret "Mk+48X1wsEIOKu2gQsPu+vgNwO3jmDbhjy5PzfBonrzQDMNuIk3w9gAi8nCkDP3g" --value-format "avro"

Confluent CLI using Windows (producer)

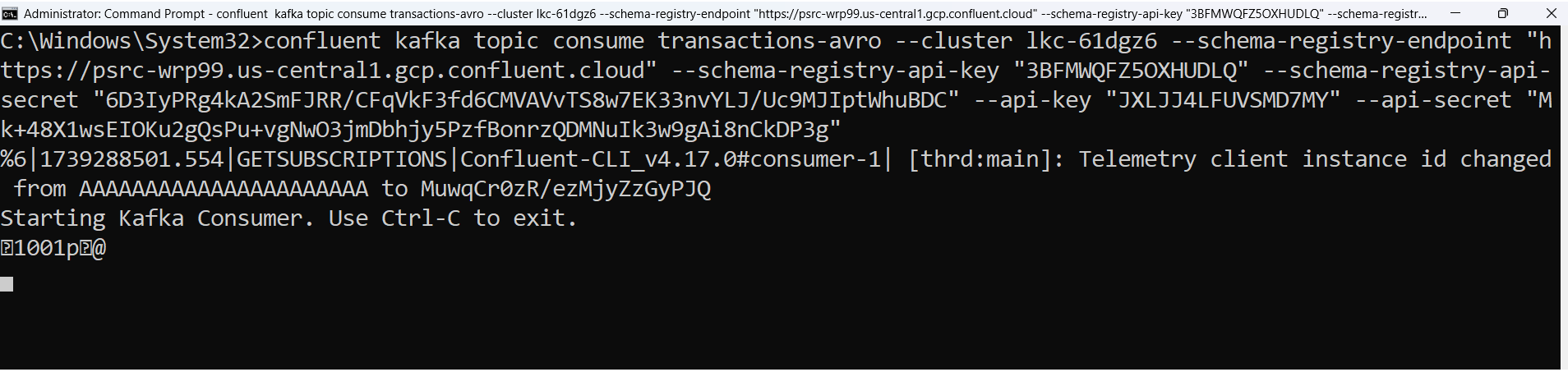
Administrator: Command Prompt - confluent kafka topic produce transactions-avro --cluster lkc-61dgz6 --schema "F:\Local disk\Docker\kafkahelm\helm\schema.txt" --schema-registry-endpoint "https://psrc-wrp99.us-central1.gcp.confluent.cloud" --schema...

```
C:\Windows\System32>confluent api-key create --resource lkc-61dgz6
It may take a couple of minutes for the API key to be ready.
Save the API key and secret. The secret is not retrievable later.
+------------+------------------------------------------------------------------+
| API Key    | JXLJJ4LFUVSMD7MY                                                 |
| API Secret | Mk+48X1wsEIOKu2gQsPu+vgNwO3jmDbhjy5PzfBonrzQDMNuIk3w9gAi8nCkDP3g |
+------------+------------------------------------------------------------------+

C:\Windows\System32>confluent kafka topic produce transactions-avro --cluster lkc-61dgz6 --schema "F:\Local disk\Docke
r\kafkahelm\helm\schema.txt" --schema-registry-endpoint "https://psrc-wrp99.us-central1.gcp.confluent.cloud" --schema-
registry-api-key "3BFMWQFZ5OXHUDLQ" --schema-registry-api-secret "6D3IyPRg4kA2SmFJRR/CFqVkF3fd6CMVAVvTS8w7EK33nvYLJ/Uc
9MJIptWhuBDC" --api-key "JXLJJ4LFUVSMD7MY" --api-secret "Mk+48X1wsEIOKu2gQsPu+vgNwO3jmDbhjy5PzfBonrzQDMNuIk3w9gAi8nCkD
P3g" --value-format "avro"
Successfully registered schema with ID "100004".
%Starting Kafka Producer. Use Ctrl-C to exit.
6|1739287716.680|GETSUBSCRIPTIONS|Confluent-CLI_v4.17.0#producer-1| [thrd:main]: Telemetry client instance id changed
from AAAAAAAAAAAAAAAAAAAAAA to YogWMqPATaygACKYWZtC2g
{ "id":"1000", "amount":500 }
```

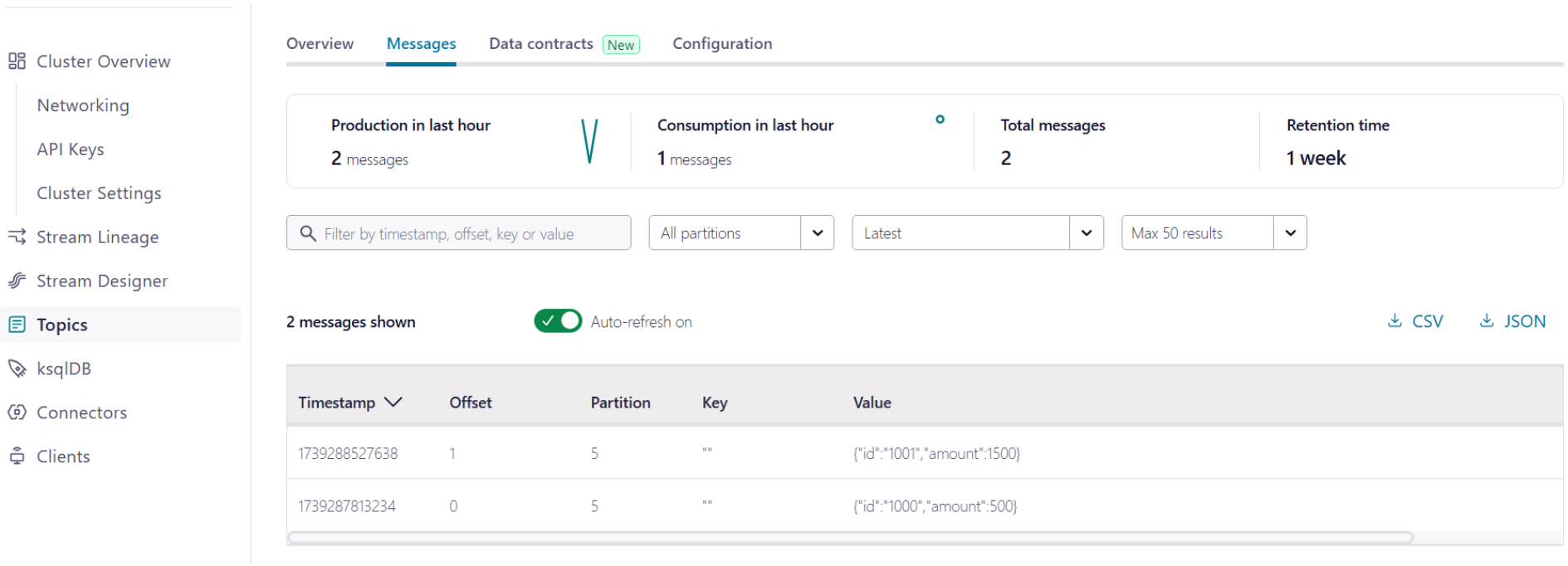
Confluent CLI using Windows (consumer)

- kafka topic consume transactions-avro --cluster lkc-61dgz6 --schema-registry-endpoint "https://psrc-wrp99.us-central1.gcp.confluent.cloud" --schema-registry-api-key "3BFMWQFZ5OXHUDLQ" --schema-registry-api-secret "6D3IyPRg4kA2SmFJRR/CFqVkF3fd6CMVAVvTS8w7EK33nvYLJ/Uc9MJIptWhuBDC" --api-key "JXLJJ4LFUVSMD7MY" --api-secret "Mk+48X1wsEIOKu2gQsPu+vgNwO3jmDbhjy5PzfBonrzQDMNuIk3w9gAi8nCkDP3g"

Confluent CLI using Windows (consumer)



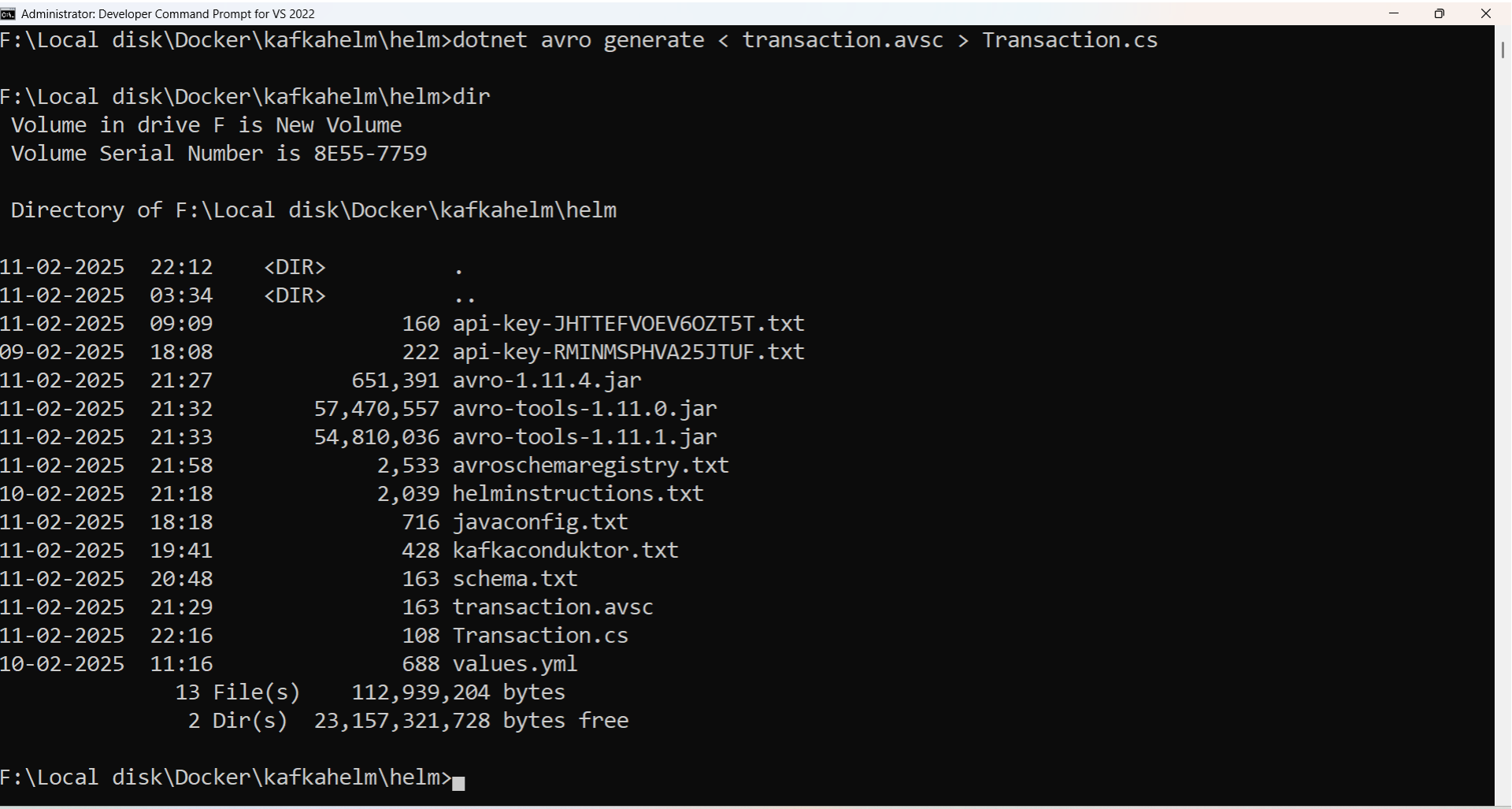
Confluent CLI using Windows (consumer)



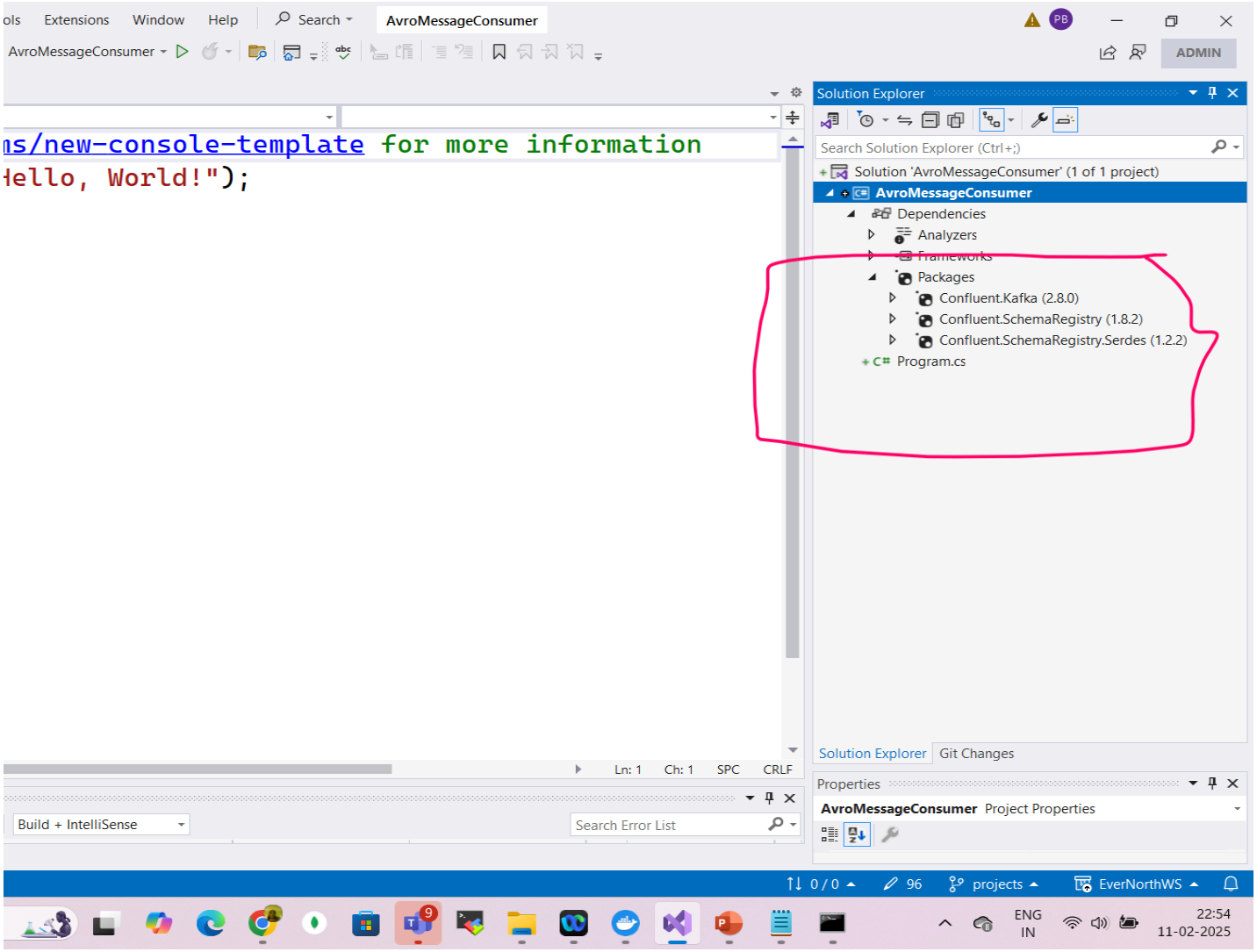
AVRO c#

- dotnet tool install Chr.Avro.Cli –global
- dotnet avro generate < transaction.avsc > Transaction.cs

AVRO c#


Administrator: Developer Command Prompt for VS 2022
F:\Local disk\Docker\kafkahelm\helm>dotnet avro generate < transaction.avsc > Transaction.cs
F:\Local disk\Docker\kafkahelm\helm>dir
Volume in drive F is New Volume
Volume Serial Number is 8E55-7759
Directory of F:\Local disk\Docker\kafkahelm\helm
11-02-2025 22:12 <DIR> .
11-02-2025 03:34 <DIR> ..
11-02-2025 09:09 160 api-key-JHTTEFVOEV6OZT5T.txt
09-02-2025 18:08 222 api-key-RMINMSPHVA25JTUF.txt
11-02-2025 21:27 651,391 avro-1.11.4.jar
11-02-2025 21:32 57,470,557 avro-tools-1.11.0.jar
11-02-2025 21:33 54,810,036 avro-tools-1.11.1.jar
11-02-2025 21:58 2,533 avroschemaregistry.txt
10-02-2025 21:18 2,039 helminstructions.txt
11-02-2025 18:18 716 javaconfig.txt
11-02-2025 19:41 428 kafkaconduktor.txt
11-02-2025 20:48 163 schema.txt
11-02-2025 21:29 163 transaction.avsc
11-02-2025 22:16 108 Transaction.cs
10-02-2025 11:16 688 values.yml
13 File(s) 112,939,204 bytes
2 Dir(s) 23,157,321,728 bytes free
F:\Local disk\Docker\kafkahelm\helm>
26°C
Mostly cloudy
Search
ENG
IN
22:16
11-02-2025

AVRO c#


Extensions
Window
Help
Search
AvroMessageConsumer
ADMIN
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'AvroMessageConsumer' (1 of 1 project)
AvroMessageConsumer
Dependencies
Analyzers
Frameworks
Packages
Confluent.Kafka (2.8.0)
Confluent.SchemaRegistry (1.8.2)
Confluent.SchemaRegistry.Serdes (1.2.2)
Program.cs
Solution Explorer
Git Changes
Properties
AvroMessageConsumer Project Properties
Build + IntelliSense
Search Error List
Ln: 1
Ch: 1
SPC
CRLF
projects
EverNorthWS
ENG
IN
22:54
11-02-2025

ProtoBuf

kafka

- **Protocol Buffers (ProtoBuf)** is a **language-neutral, platform-neutral** mechanism for serializing structured data.
- Developed by **Google**, it's widely used for **data serialization**, offering compact and efficient encoding of data compared to formats like **JSON** or **XML**.
- **Compact:** Smaller message sizes due to binary encoding.
- **Fast:** Faster serialization/deserialization.
- **Schema evolution-friendly:** Designed to handle changes to message schemas over time.

Why Use ProtoBuf Instead of JSON or XML?

kafka

Feature	ProtoBuf	JSON	XML
Efficiency	Compact binary format (smaller size)	Human-readable text format	Human-readable text format
Speed	Faster serialization/deserialization	Slower compared to ProtoBuf	Slowest among the three
Schema	Strongly typed using `.proto` schema	No formal schema enforcement	Optional schema (DTD/XSD)
Schema Evolution	Handles forward and backward compatibility	Schema changes require extra logic	Schema changes require extra logic

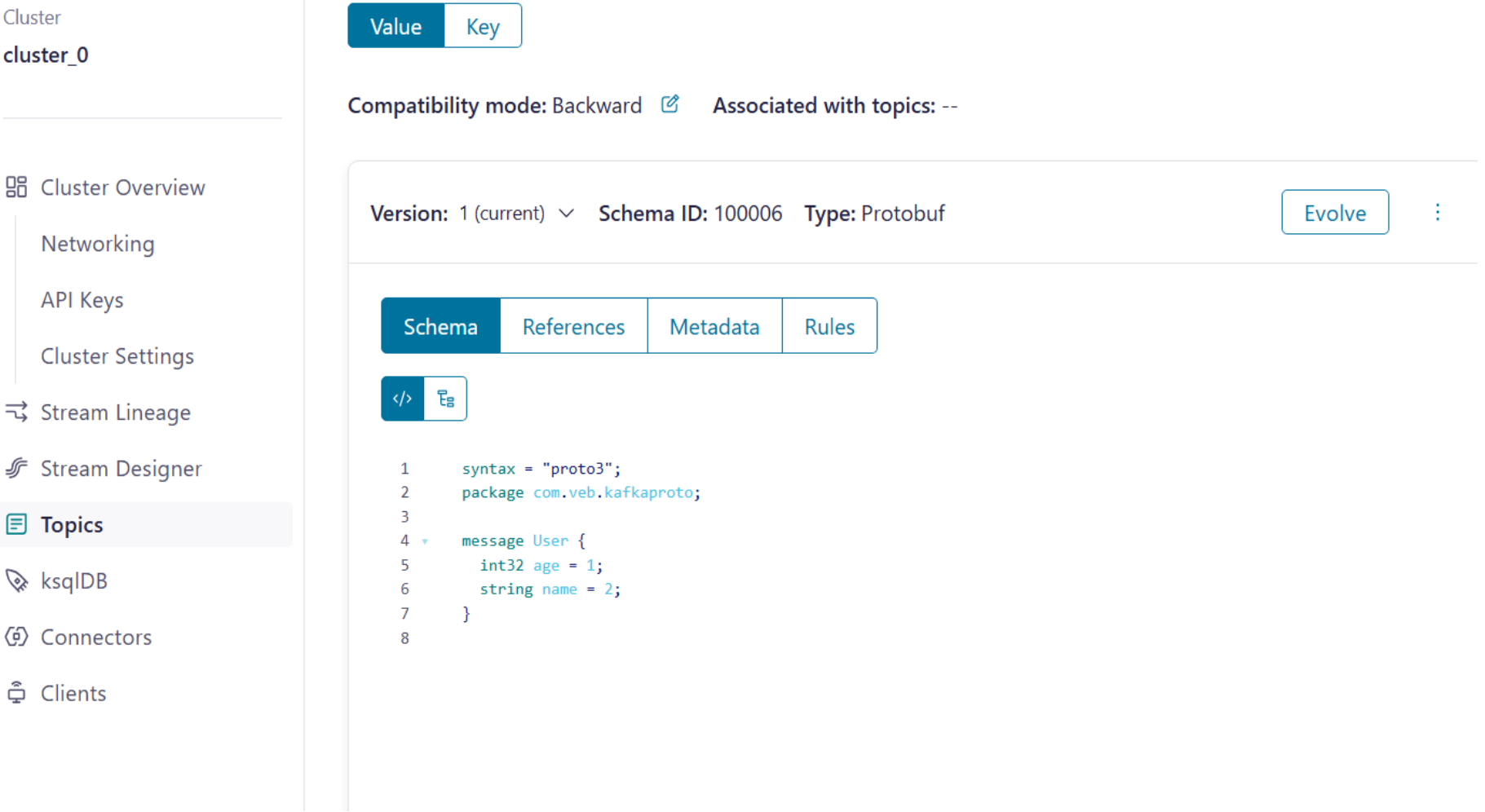
How ProtoBuf Works

- Define a schema using the .proto file format.
- Compile the schema using the ProtoBuf compiler (protoc) to generate code for your programming language.
- Use the generated code to serialize and deserialize messages.

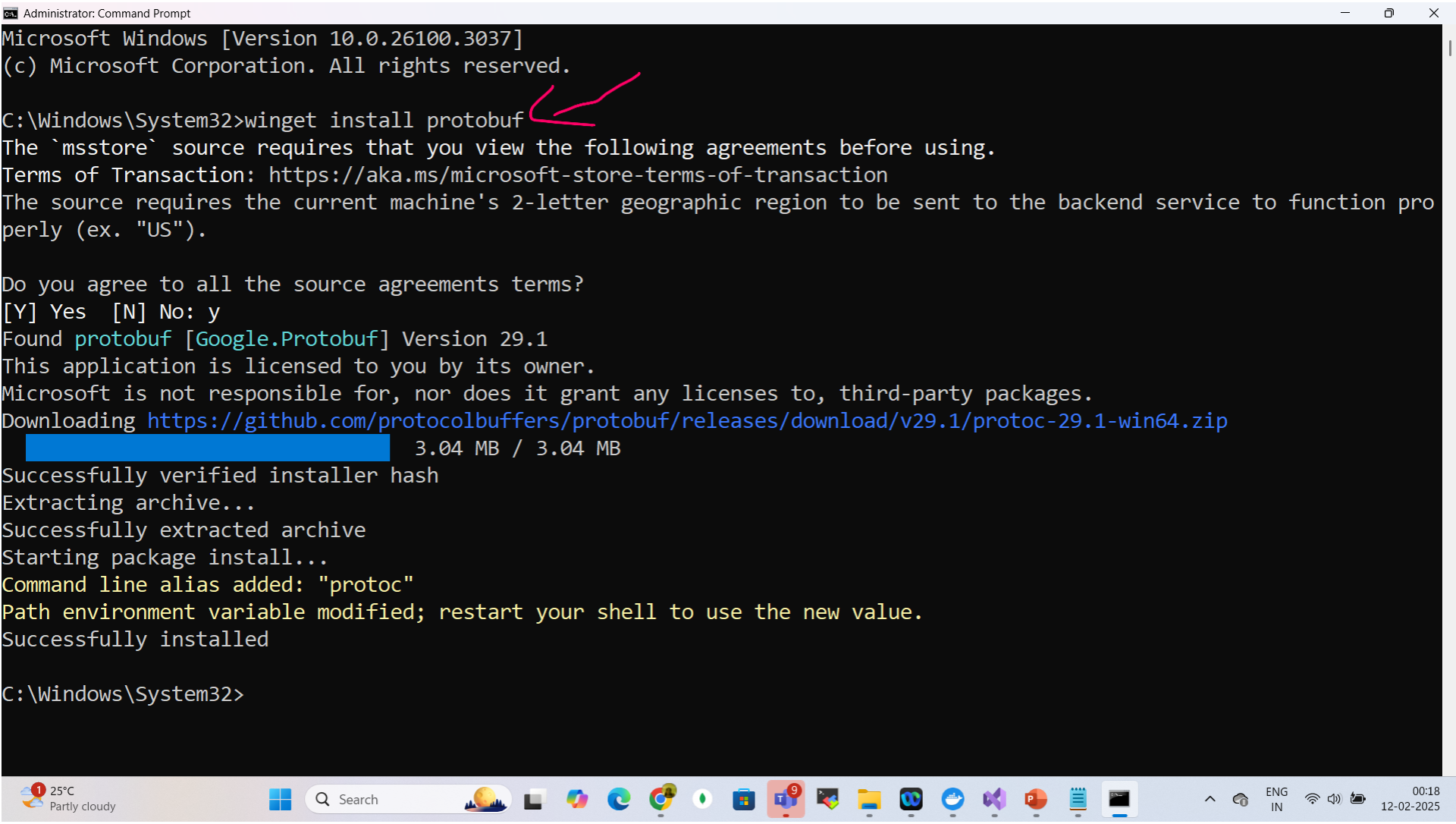
Use Cases of ProtoBuf

- Inter-service communication (gRPC): ProtoBuf is the underlying serialization mechanism for gRPC.
- Efficient storage: Storing large amounts of structured data efficiently.
- Configuration files: Machine-readable configuration files for high-performance applications.

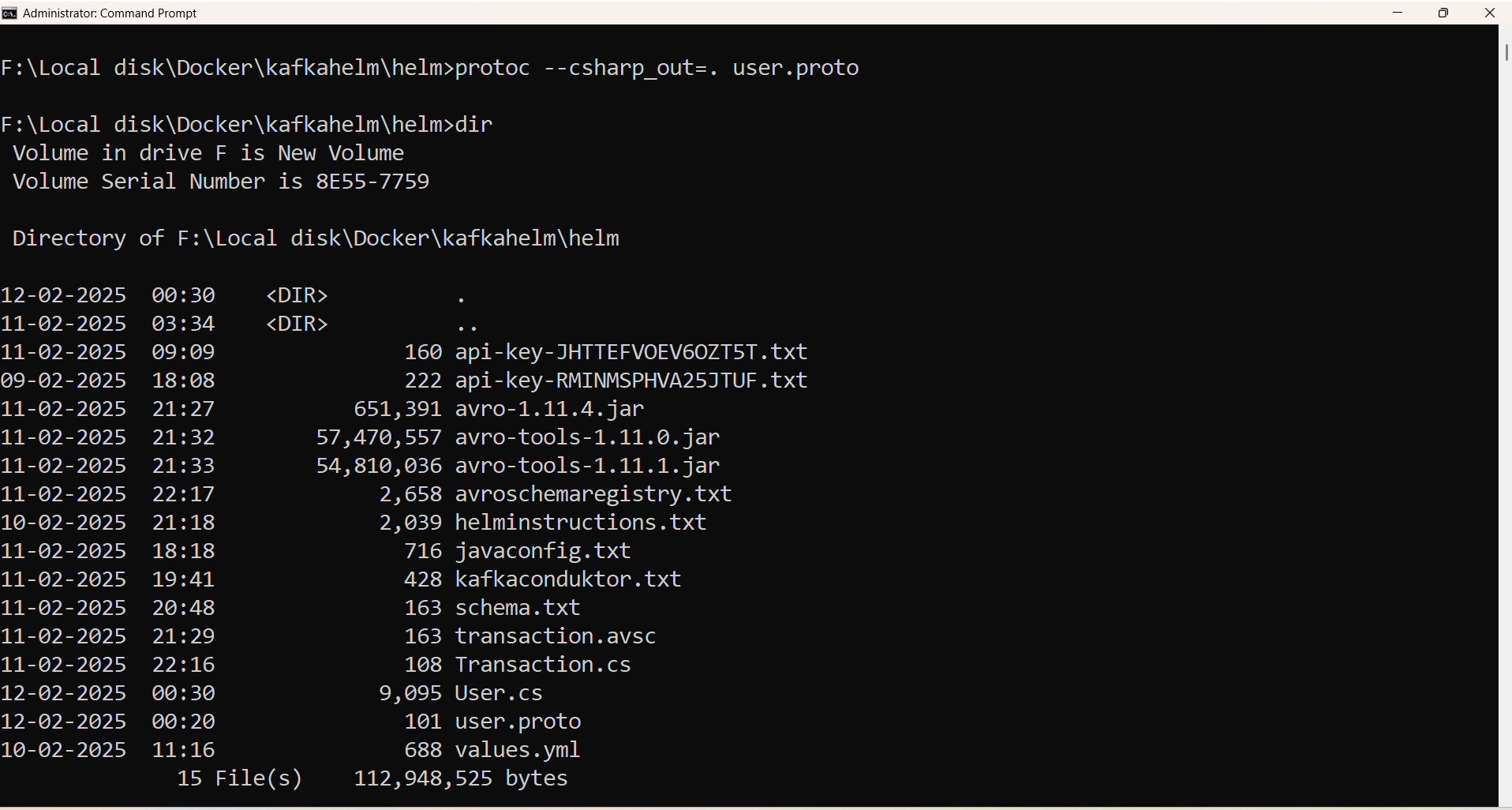
Create ProtoBuf Schema



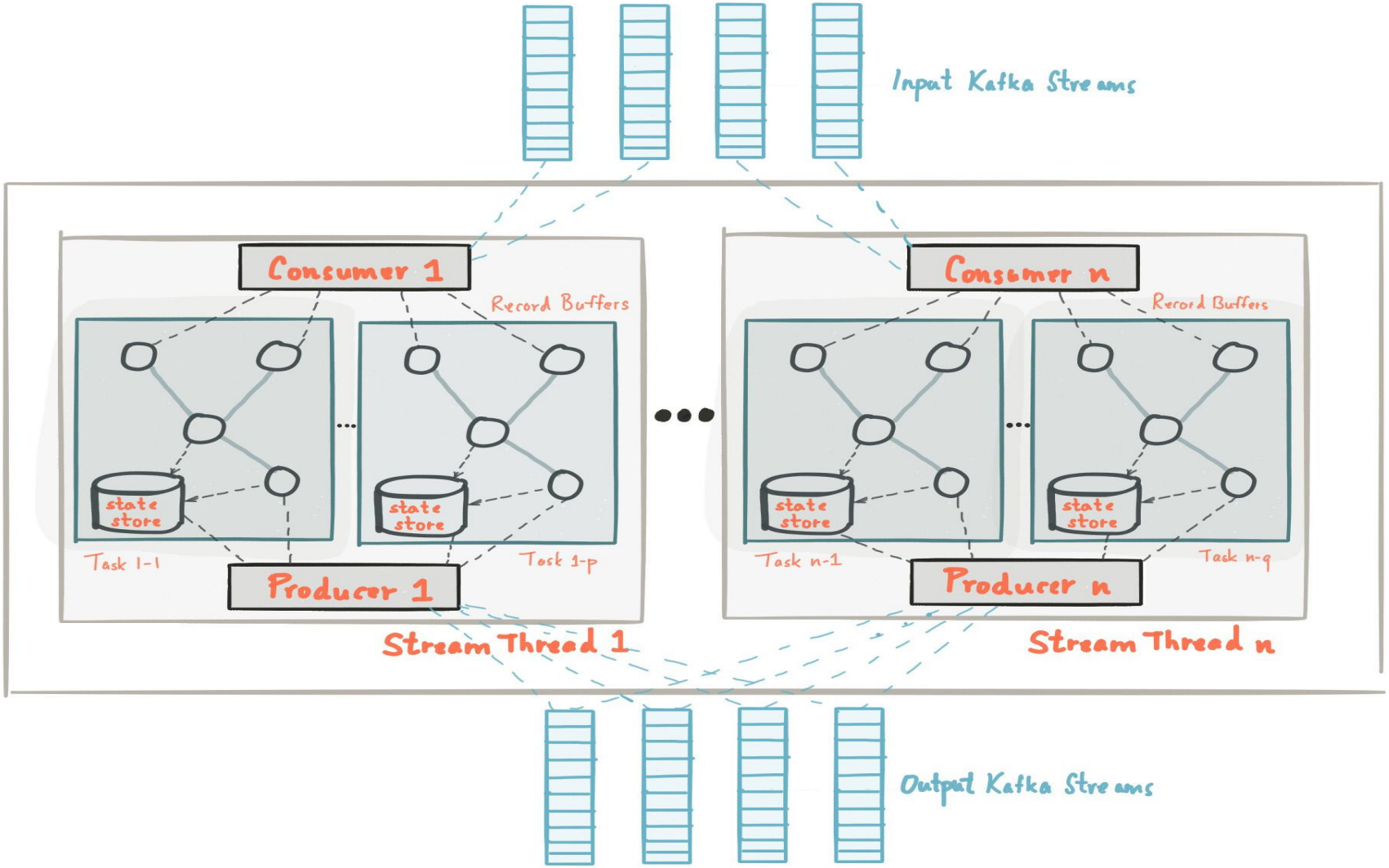
Install Protoc compiler



Generate c# class from proto file



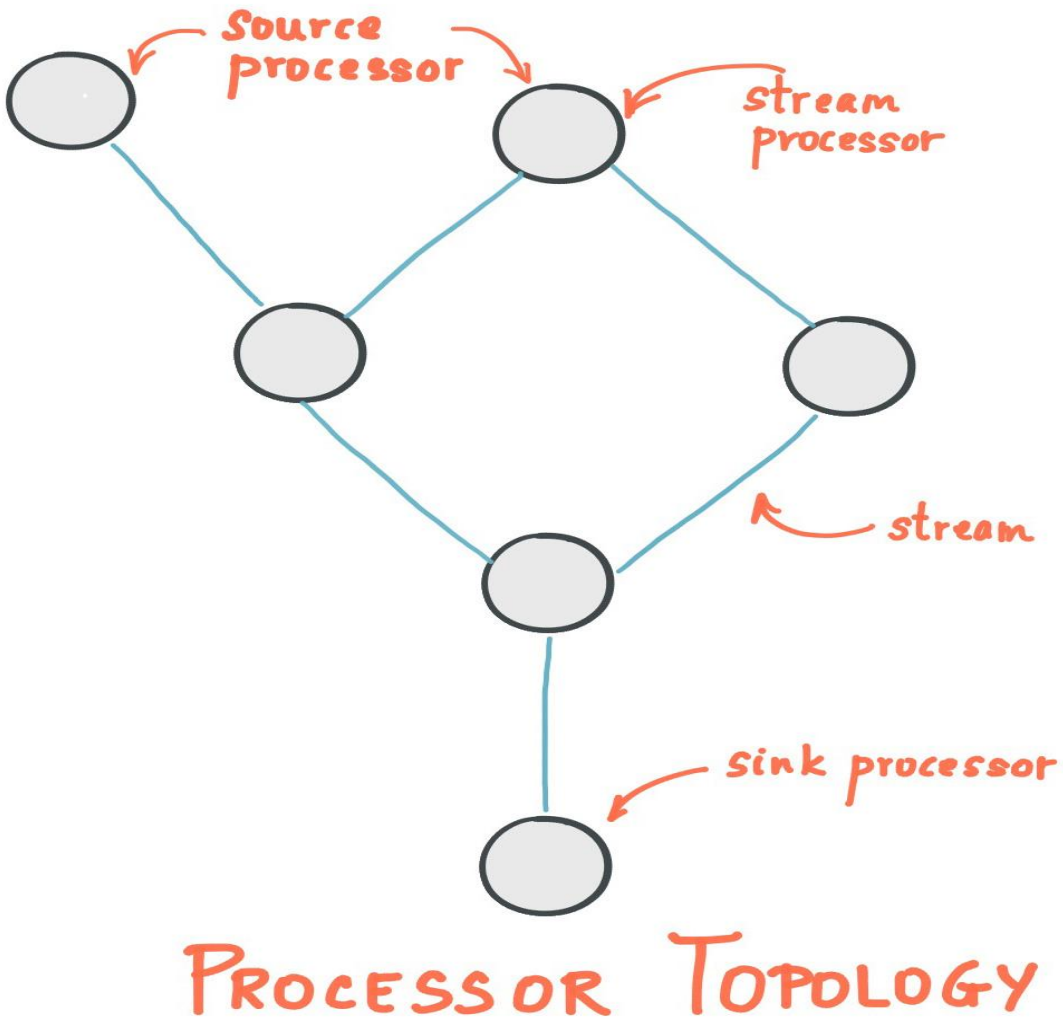
Key Concepts of Kafka Streams



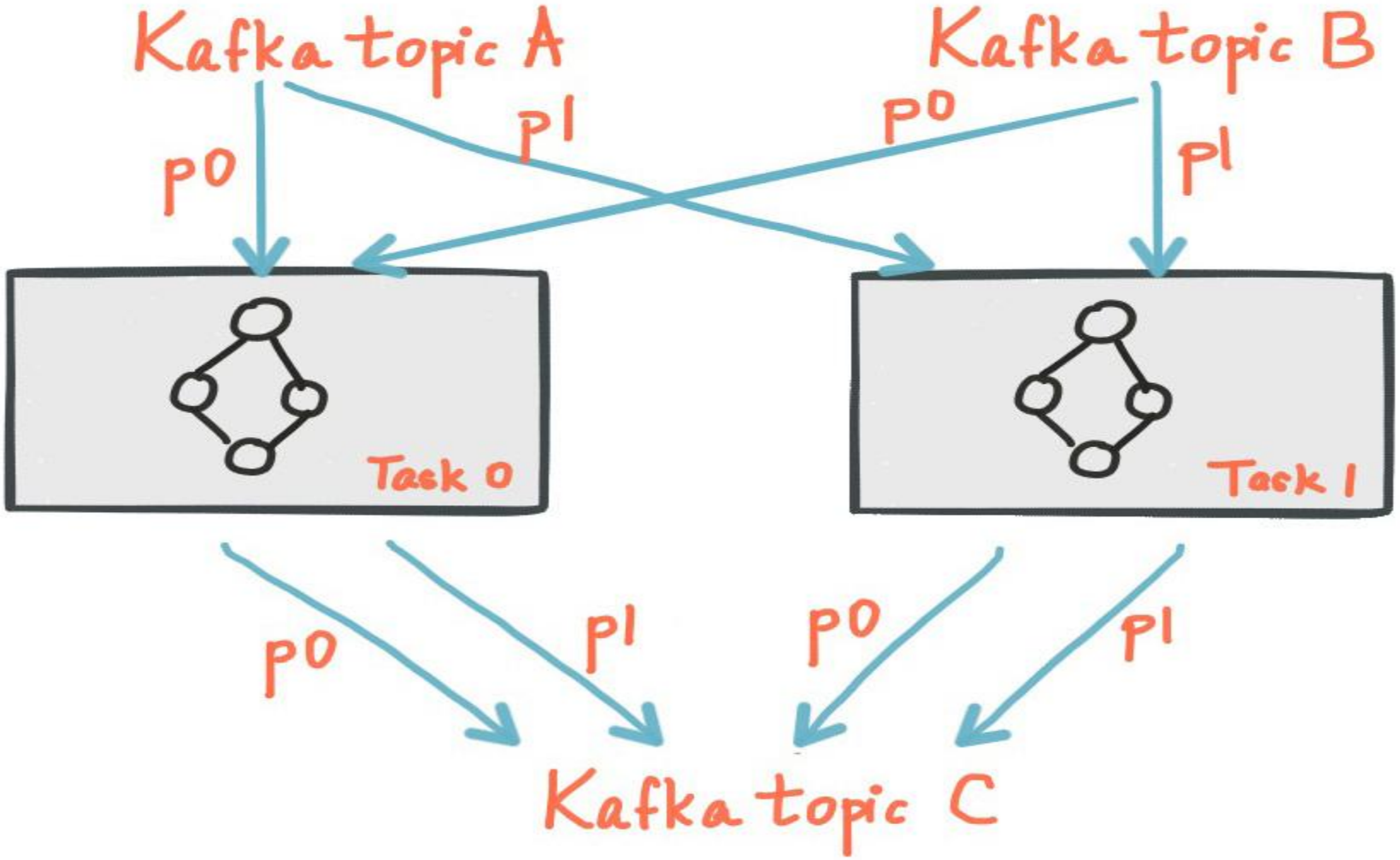
Key Concepts of Kafka Streams

1. **Stream**: A continuous flow of data records.
2. **Topic**: Kafka stores streams in topics.
3. **KStream & KTable**: Core abstractions for processing records.
    1. **KStream**: Represents a stream of records (immutable, append-only).
    2. **KTable**: Represents a changelog stream (stateful, latest value per key).
4. **Stateful Processing**: Supports operations like windowing, aggregations, and joins.
5. **Topology**: Defines how streams are processed in a DAG (Directed Acyclic Graph).
6. **Processor API**: Allows custom processing logic beyond the DSL.

Processor topology



Processor topology



Stream Processing Model

- Kafka Streams operates on Kafka topics and continuously processes data as it arrives.
- The processing model follows the **event-driven** and **distributed computing** paradigms.
- **Record (Event):** The fundamental unit of data in Kafka Streams.
- **Streams:** A continuous flow of records.
- **Tables (Stateful Data):** A key-value abstraction for tracking state over time.
- **Topology:** A directed graph that defines transformations.

Key Abstractions

Abstraction	Description
KStream	Represents a stream of records (immutable, append-only)
KTable	Represents stateful data (latest value per key)
GlobalKTable	Similar to KTable but globally available on all instances
State Stores	Persist intermediate results for aggregations, joins, and queries
Processor API	Low-level API for custom transformations

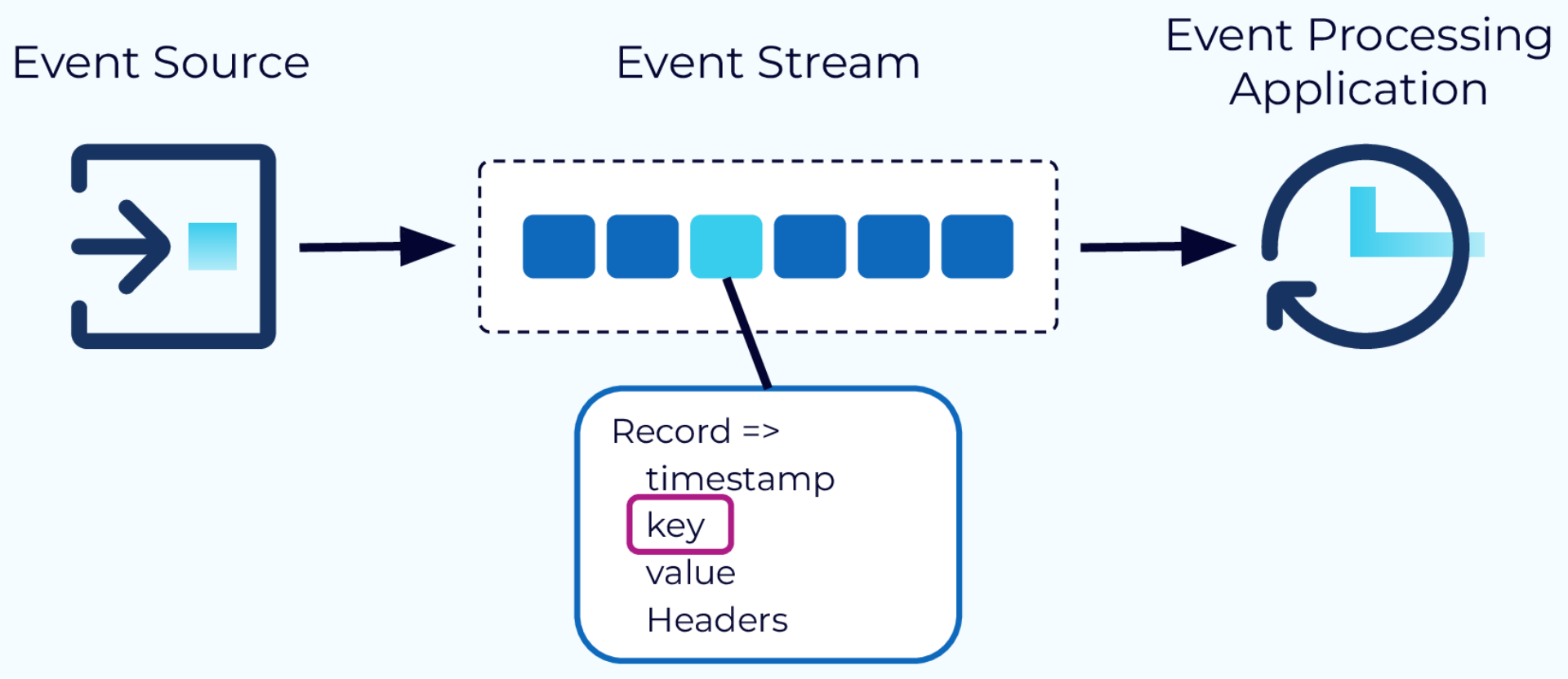
Stream Processing Operations

- Kafka Streams provides various transformation operations:
- **2.1 Stateless Transformations**
- **map()** – Transforms key/value pairs.
- **filter()** – Filters records based on conditions.
- **flatMap()** – Produces multiple output records from one input record.
- **peek()** – Performs a side effect (e.g., logging) without modifying the stream.

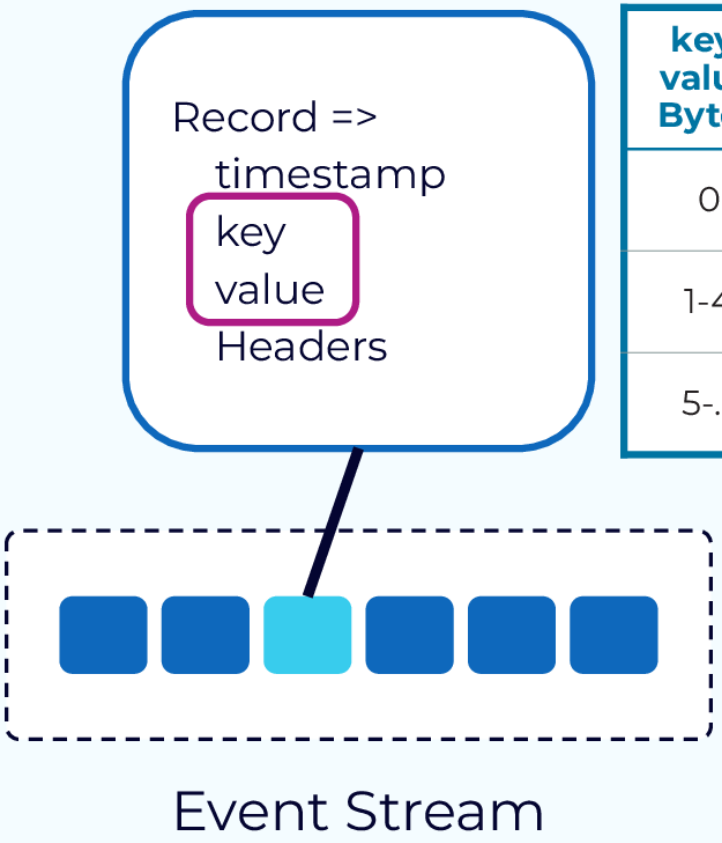
Stream Processing Operations

- **2.2 Stateful Transformations**
- Stateful transformations require maintaining intermediate results, often stored in **RocksDB-backed state stores**.
- **groupBy()** – Reorganizes data for aggregation.
- **count()** – Counts occurrences per key.
- **reduce()** – Combines records with a specified logic.
- **aggregate()** – Builds complex aggregations.

Event Processing



Event Stream



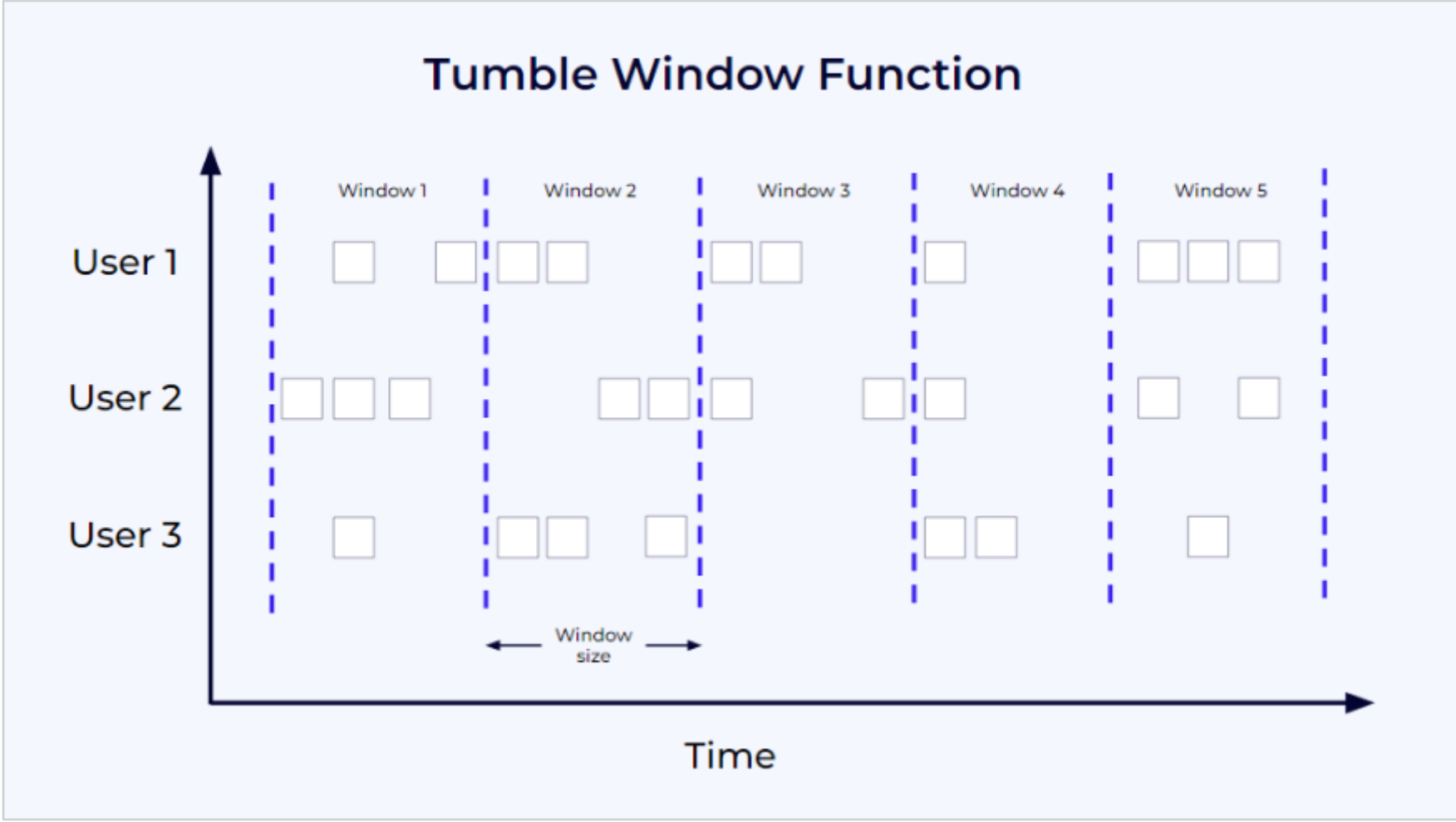
key/ value Bytes	Area	Description
0	Magic Byte	Confluent serialization format version number; currently always 0.
1-4	Schema ID	4-byte schema ID as returned by Schema Registry.
5-...	Data	Serialized data for the specified schema format.

Windowing

- Windowing is used for time-based aggregations in Kafka Streams. Windows define **how long** records should be grouped together.
- **Types of Windows**

1. **Tumbling Window**: Fixed-size, non-overlapping time windows.
2. **Hopping Window**: Overlapping windows (fixed interval + hop size).
3. **Sliding Window**: Overlapping windows where each event can belong to multiple windows.
4. **Session Window**: Based on user activity (closes when no new events appear).

Tumbling Windows



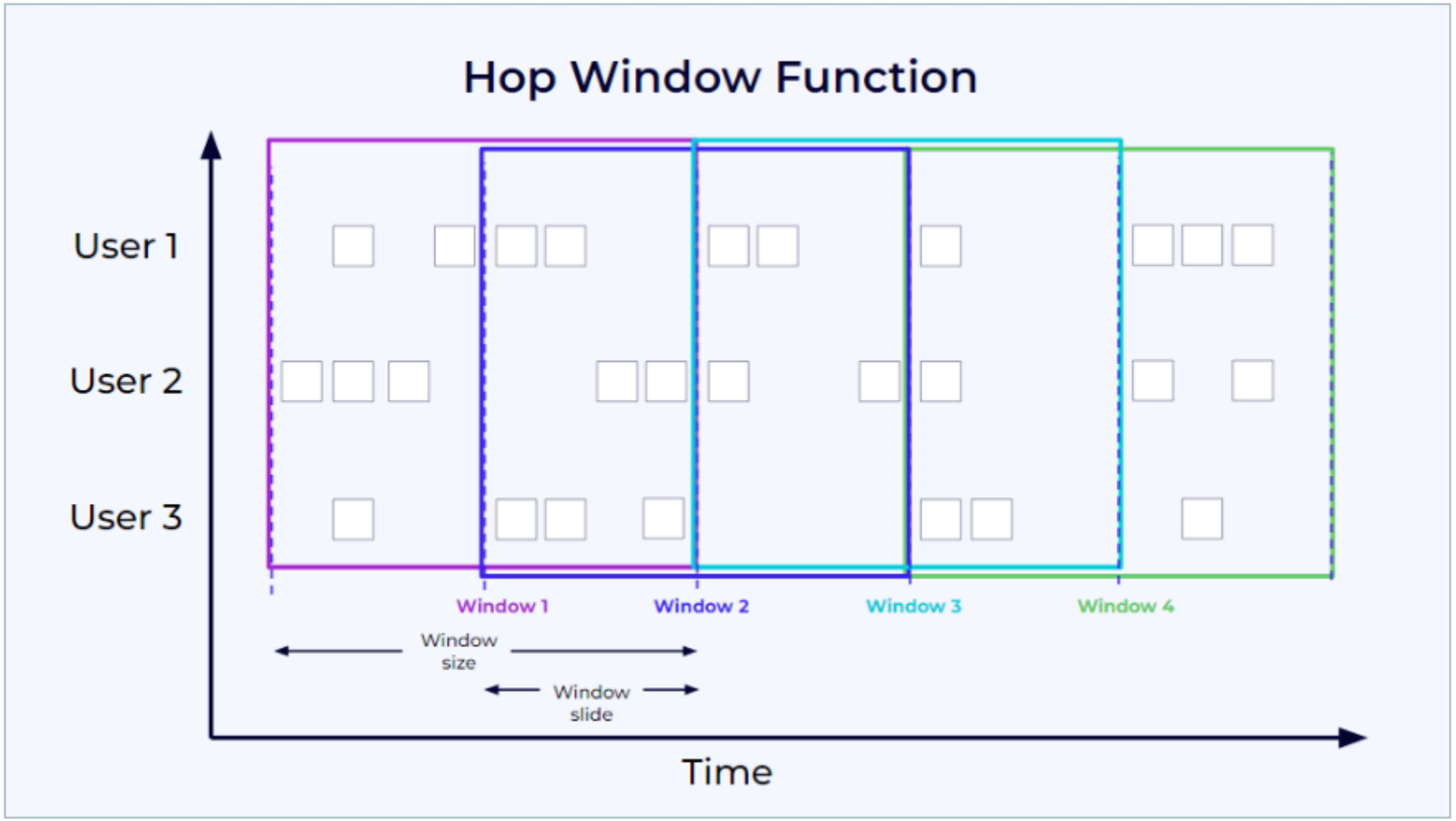
Tumbling Windows

kafka

- SELECT * FROM TABLE(
- TUMBLE(TABLE `examples`.`marketplace`.`orders`, DESCRIPTOR($rowtime), INTERVAL '10' MINUTES))

- -- or with the named params
- -- note: the DATA param must be the first
- SELECT * FROM TABLE(
- TUMBLE(
- DATA => TABLE `examples`.`marketplace`.`orders`,
- TIMECOL => DESCRIPTOR($rowtime),
- SIZE => INTERVAL '10' MINUTES));

Hopping Windows



Hopping Windows

kafka

- SELECT * FROM TABLE(
- HOP(TABLE `examples`.`marketplace`.`orders`, DESCRIPTOR($rowtime), INTERVAL '5' MINUTES, INTERVAL '10' MINUTES))

- -- or with the named params
- -- note: the DATA param must be the first
- SELECT * FROM TABLE(
- HOP(
- DATA => TABLE `examples`.`marketplace`.`orders`,
- TIMECOL => DESCRIPTOR($rowtime),
- SLIDE => INTERVAL '5' MINUTES,
- SIZE => INTERVAL '10' MINUTES));

Session Windows


```
SELECT * FROM TABLE(
  SESSION(TABLE `examples`.`marketplace`.`orders` PARTITION BY product_id, DESCRIPTOR($rowtime), INTERVAL '1' MINUTES));


-- or with the named params
-- note: the DATA param must be the first
SELECT * FROM TABLE(
    SESSION(
      DATA => TABLE `examples`.`marketplace`.`orders` PARTITION BY product_id,
      TIMECOL => DESCRIPTOR($rowtime),
      GAP => INTERVAL '1' MINUTES));
```

Windowing

Feature/Type	Tumbling Window	Hopping Window	Session Window	Sliding Window
Definition	A type of window that has a fixed size and does not overlap. Each record belongs to exactly one window.	Similar to tumbling windows but can overlap. It is defined by two parameters: the window's size and its advance interval (or hop).	Dynamic windows that group events by activity sessions. Windows are defined by periods of activity and inactivity.	A type of window that allows for overlapping windows where each window is defined by a fixed duration and can slide over the data stream.
Window Size	Fixed	Fixed	Dynamic, determined by inactivity gap	Fixed
Overlap	No	Yes, if the advance interval is smaller than the window size	No, but windows are not fixed and can vary in length	Yes, windows can overlap due to the sliding nature
Use Cases	Aggregations over a uniform time interval, such as hourly metrics.	Aggregations where more frequent updates are needed within a fixed period, allowing overlap.	Aggregations where the activity is bursty or irregular, such as user sessions in web analytics.	Fine-grained aggregations where updates are needed for every event or at short intervals, capturing trends over time.
Key Characteristics	Simple and non-overlapping, making it suitable for clear-cut time intervals.	Provides flexibility with overlapping windows, useful for more nuanced time-based aggregations.	Ideal for handling irregular or event-driven data, adapting to the data's inherent patterns.	Offers the most granular control over windowing, allowing for continuous updates within overlapping periods.

Joins

4.1 Types of Joins

Join Type	Description
KStream-KStream Join	Joins two KStreams on a key within a time window.
KStream-KTable Join	Enriches a stream with data from a table.
KTable-KTable Join	Merges two tables on matching keys.

Confluent Kafka using Local

- Refer F:\Local disk\Docker\kafkahelm\docker

- https://developer.confluent.io/tutorials/change-topic-partitions-replicas/ksql.html

Scaling Kafka Streams

- Kafka Streams applications scale **horizontally** across multiple instances.
- Each **Kafka Streams instance** runs multiple **tasks**.
- Each task processes a subset of **Kafka partitions**.
- More instances = More parallelism.
- To scale, just deploy multiple instances of the application.

Kafka Streams vs Other Stream Processing Frameworks

Feature	Kafka Streams	Apache Flink	Apache Spark Streaming
Processing Model	Event-driven	Batch + Event-driven	Micro-batch
Stateful Processing	Yes (RocksDB)	Yes	Yes
Scalability	High (Parallel per partition)	High	Moderate
Deployment	Embedded (microservices)	Clustered	Clustered
Latency	Low (ms-level)	Low (ms-level)	High (sec-level)

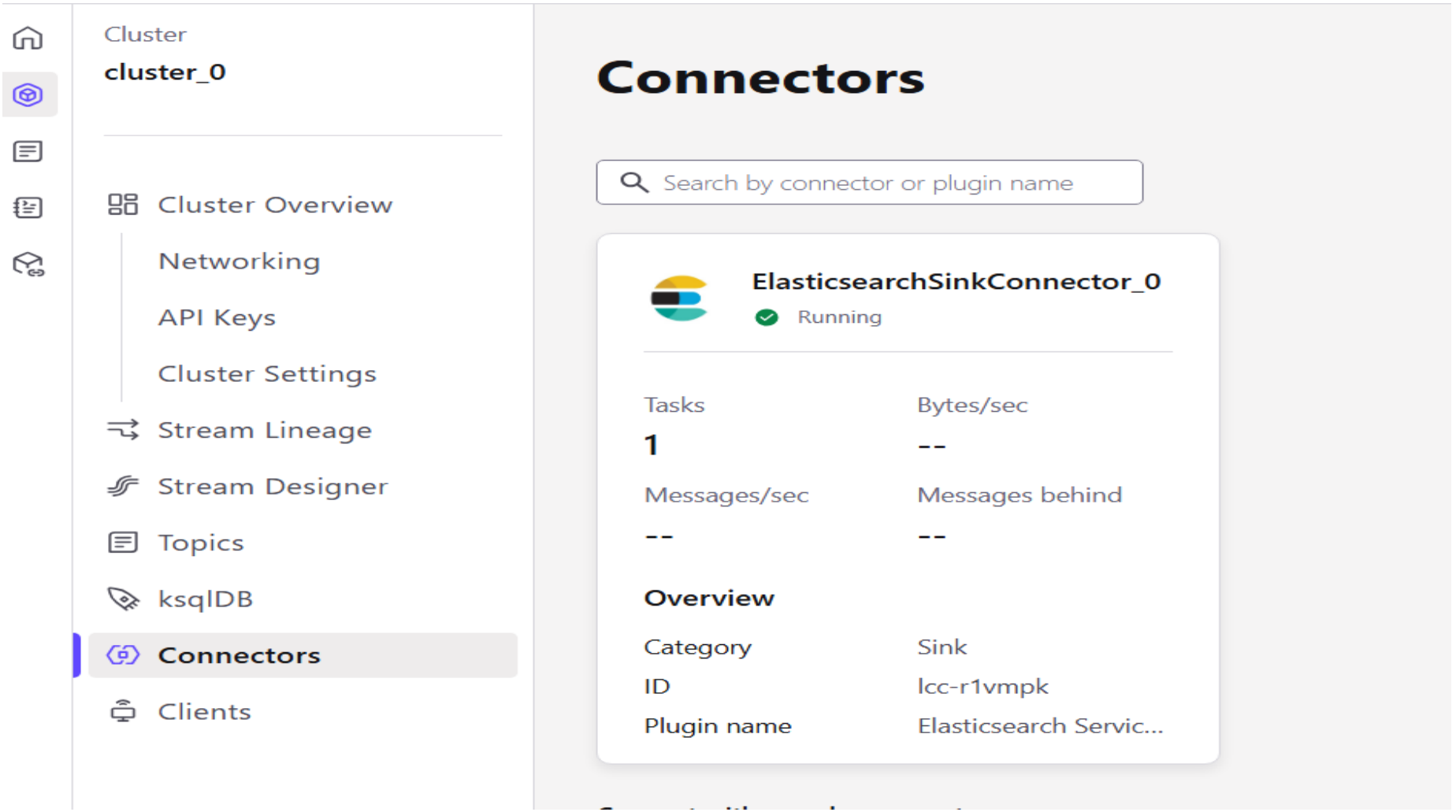
Kafka Streaming vs Traditional Messaging (Message Queues)

Feature	Kafka Streaming	Traditional Messaging (MQ)
Processing Model	Continuous stream processing (real-time)	Message-based (discrete events)
Data Flow	Continuous & event-driven	Request-response, point-to-point
Message Storage	Persistent log-based storage (Kafka)	Queue-based, messages are removed after consumption
Scaling	Distributed, parallel processing	Limited to broker-defined consumers
State Management	Supports stateful processing via RocksDB	Stateless (or requires an external DB)
Fault Tolerance	Built-in state recovery and replay	Message loss possible if not handled properly
Replayability	Supports replaying past messages	Messages are removed after consumption
Latency	Low latency (ms-level)	Moderate latency (depends on broker performance)

Key Considerations for Choosing

Criteria	Kafka Streaming	Traditional Message Queue
Use Case	Real-time stream processing	Point-to-point messaging
Data Persistence	Stored in Kafka for replayability	Typically removed after processing
Ordering Guarantees	Partition-level ordering	FIFO (depending on broker)
Throughput	High	Moderate
Scaling	Horizontally scalable	Limited scaling based on queues

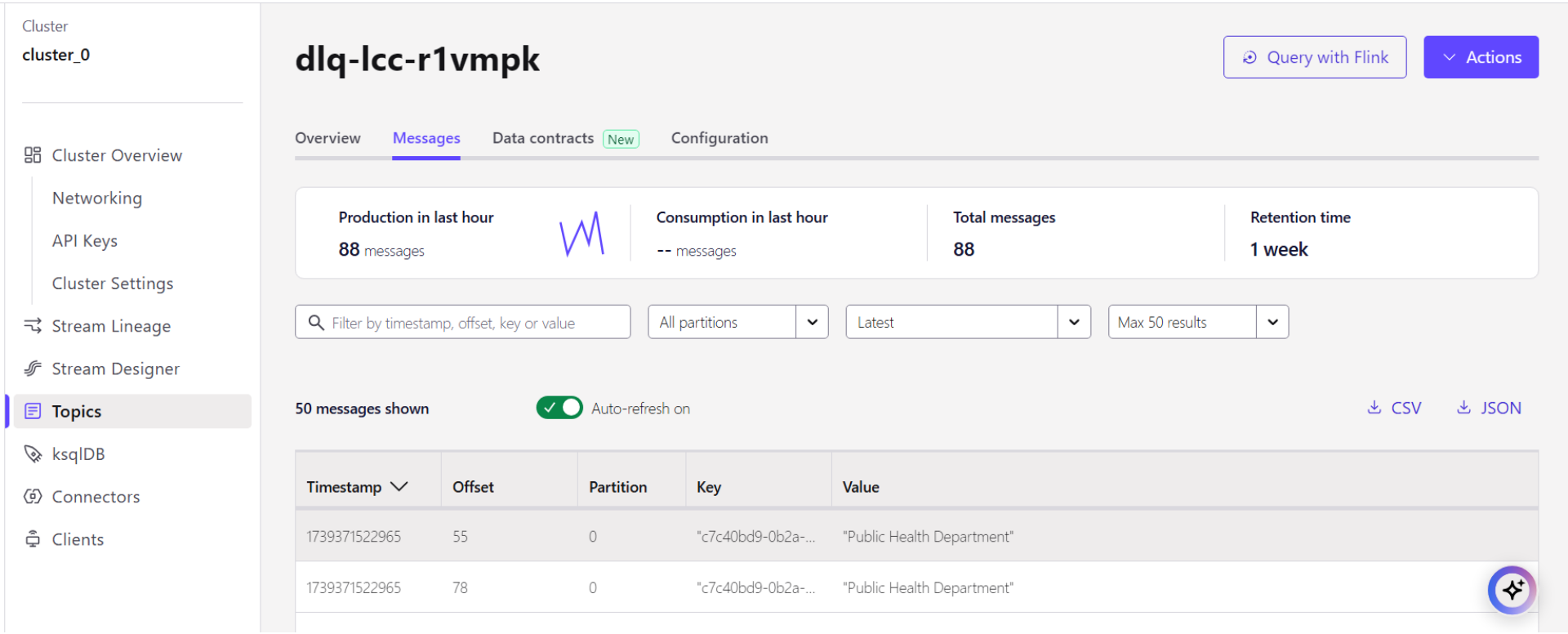
Kafka elastic Connector



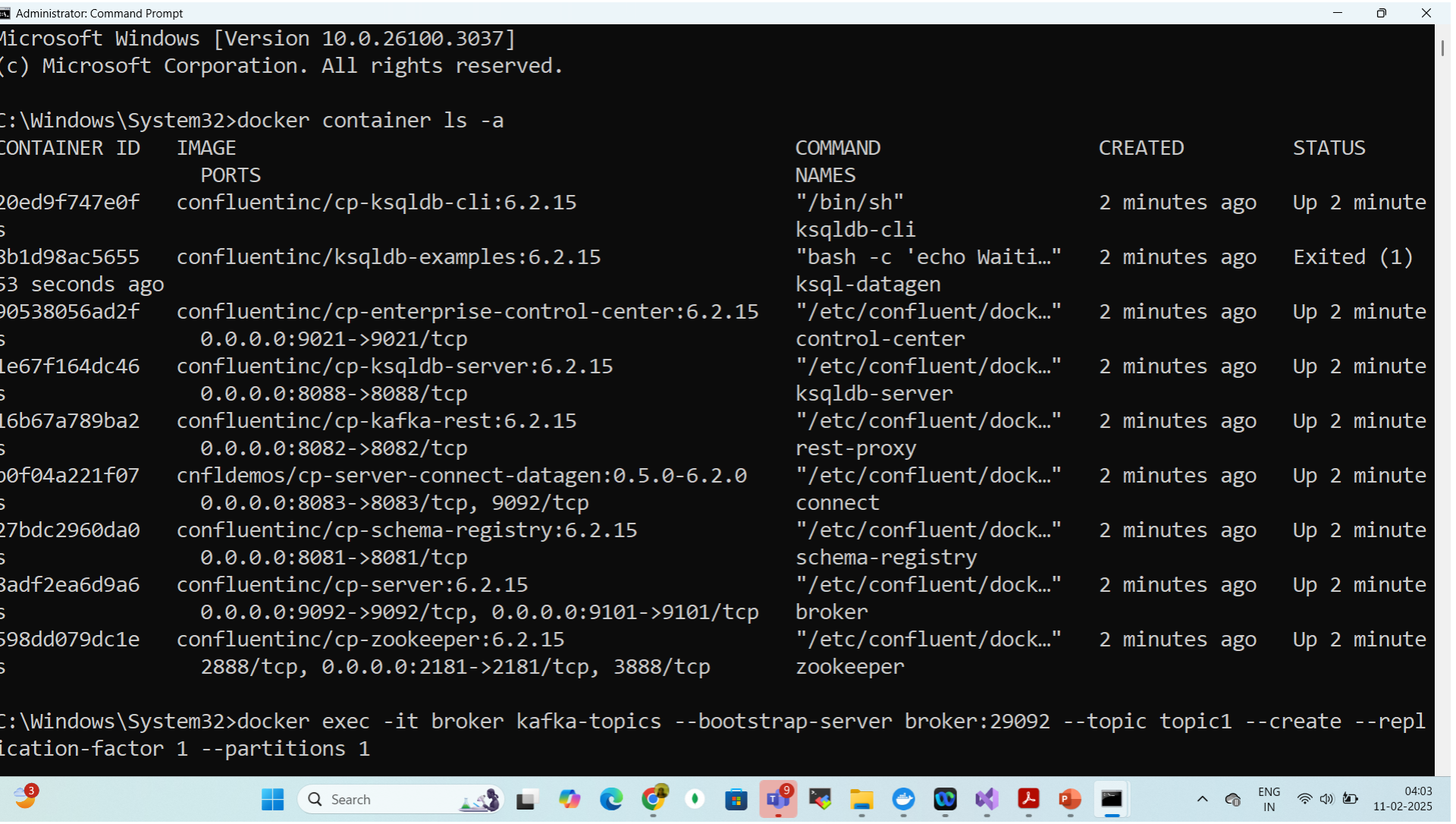
Kafka Connector sends data to elastic index

health	status	index	uuid	pri	rep	docs.count	docs.deleted	store.size	pri.store.size	dataset.size
green	open	.internal.alerts-transform.health.alerts-default-000001	fBk0rj8bR0Kjo1iBCe6J3A	1	1	0	0	498b	249b	249b
green	open	.internal.alerts-observability.logs.alerts-default-000001	xwu1_nuLRzmlv19VvMTBCQ	1	1	0	0	498b	249b	249b
green	open	.ds-logs-enterprise_search.audit-default-2025.02.12-000001	-tuXVmXkRCS2k2viIGm1Sg	1	1	9	0	159.4kb	79.6kb	79.6kb
green	open	.ds-logs-enterprise_search.api-default-2025.02.12-000001	PSI3DEe5TVGEud760eAo9Q	1	1	2	0	62.8kb	31.4kb	31.4kb
green	open	.internal.alerts-observability.uptime.alerts-default-000001	fbhmjnC3RtSnzKxp3a-3pw	1	1	0	0	498b	249b	249b
green	open	.internal.alerts-ml.anomaly-detection.alerts-default-000001	j13byq0lRCe7_6vOJY_HSw	1	1	0	0	498b	249b	249b
green	open	.internal.alerts-observability.slo.alerts-default-000001	75VABmOWR3GVCXxtBbKTiw	1	1	0	0	498b	249b	249b
green	open	.ds-metrics-fleet_server.agent_status-default-2025.02.12-000001	LznAEXkaSw6jbKT6yPAwCw	1	1	63	0	857.1kb	428.5kb	428.5kb
green	open	.internal.alerts-default.alerts-default-000001	boo0zM9RTQeTjzmpOPxTzw	1	1	0	0	498b	249b	249b
green	open	.internal.alerts-observability.apm.alerts-default-000001	nZXXAa7VRqOWt-1AbrZDQw	1	1	0	0	498b	249b	249b
green	open	.elastic-connectors-v1	pX2Q6RMDSsKS6RuGWylPGg	1	1	0	0	3.8kb	247b	247b
green	open	.internal.alerts-observability.metrics.alerts-default-000001	LdM--gqdRmqgFEQQ0giWGg	1	1	0	0	498b	249b	249b
green	open	ordersv2	EVfwxB7zRM-A770M5X-ItQ	1	1	6	0	24.5kb	12.2kb	12.2kb
green	open	.ds-metrics-fleet_server.agent_versions-default-2025.02.12-000001	eX4hR9WURa6zwjMLgcWYSg	1	1	63	0	604.6kb	302.3kb	302.3kb
green	open	.internal.alerts-ml.anomaly-detection-health.alerts-default-000001	a6S9hX6RQLKbFtLNOJyfrw	1	1	0	0	498b	249b	249b
green	open	.internal.alerts-observability.threshold.alerts-default-000001	z6ZOm-kTS1qCtJef0MWGBw	1	1	0	0	498b	249b	249b
green	open	.elastic-connectors-sync-jobs-v1	fjMmjY0OSECbARBQEe-xrA	1	1	0	0	3.8kb	247b	247b
green	open	.internal.alerts-security.alerts-default-000001	V5DyZT4YQt2VIg2Wcvc0jg	1	1	0	0	498b	249b	249b
green	open	.internal.alerts-stack.alerts-default-000001	Tgis9mUPTLyynBaJUn6s7Q	1	1	0	0	498b	249b	249b

Kafka Connector elastic keeps in dlq (error data)

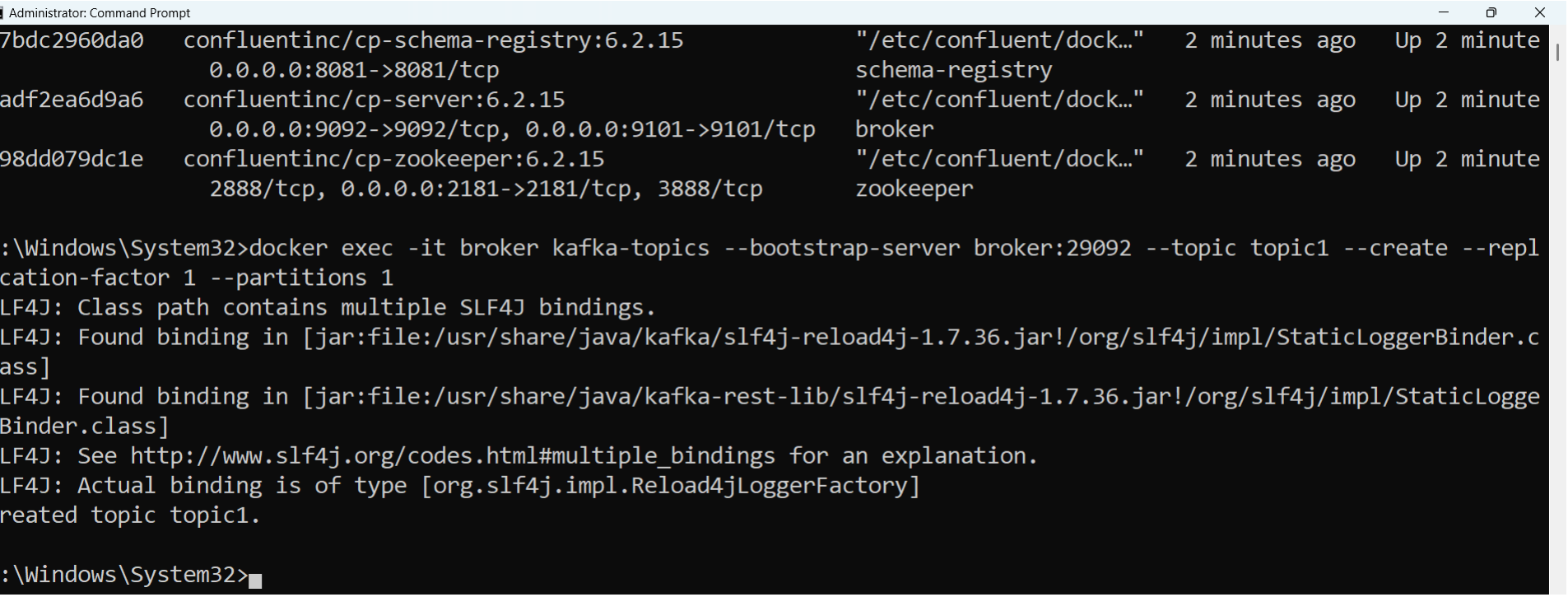


Confluent Kafka using Local



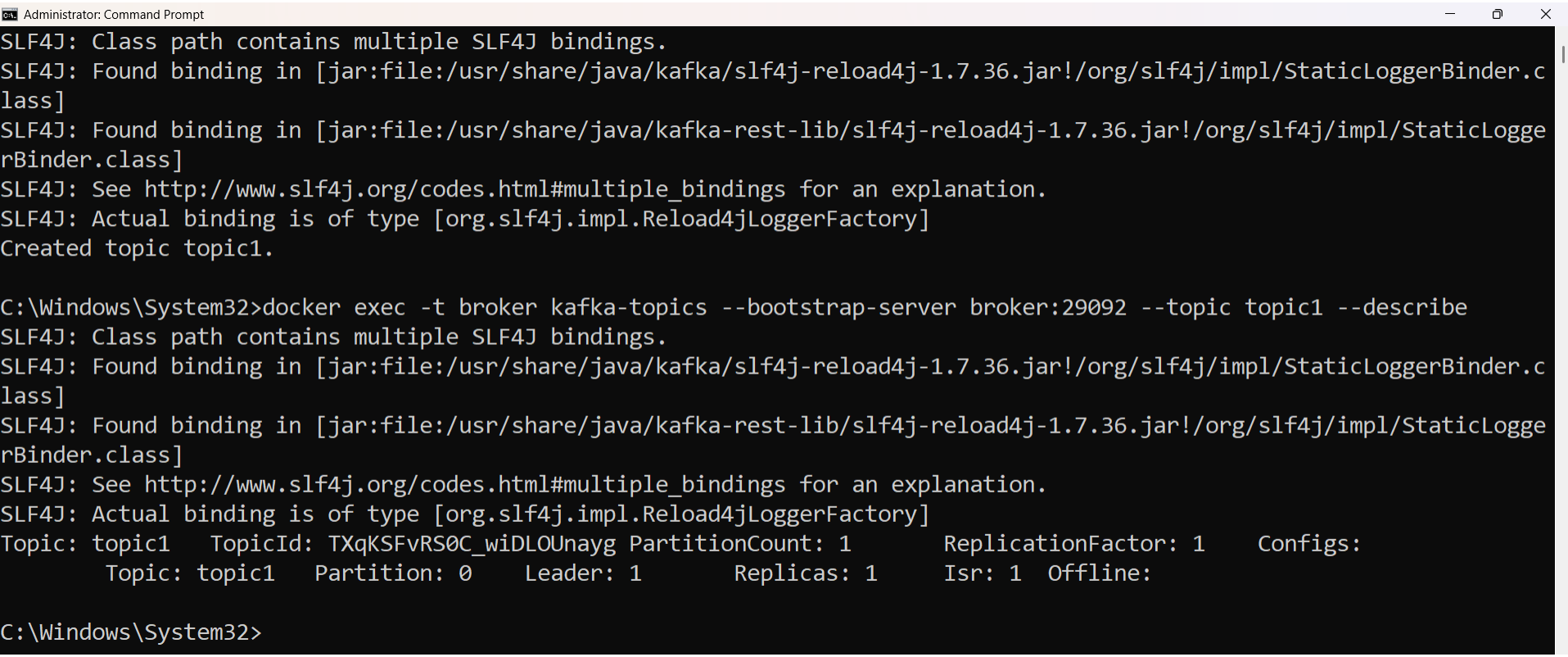
Confluent Kafka using Local

- docker exec -it broker kafka-topics --bootstrap-server broker:29092 --topic topic1 --create --replication-factor 1 --partitions 1



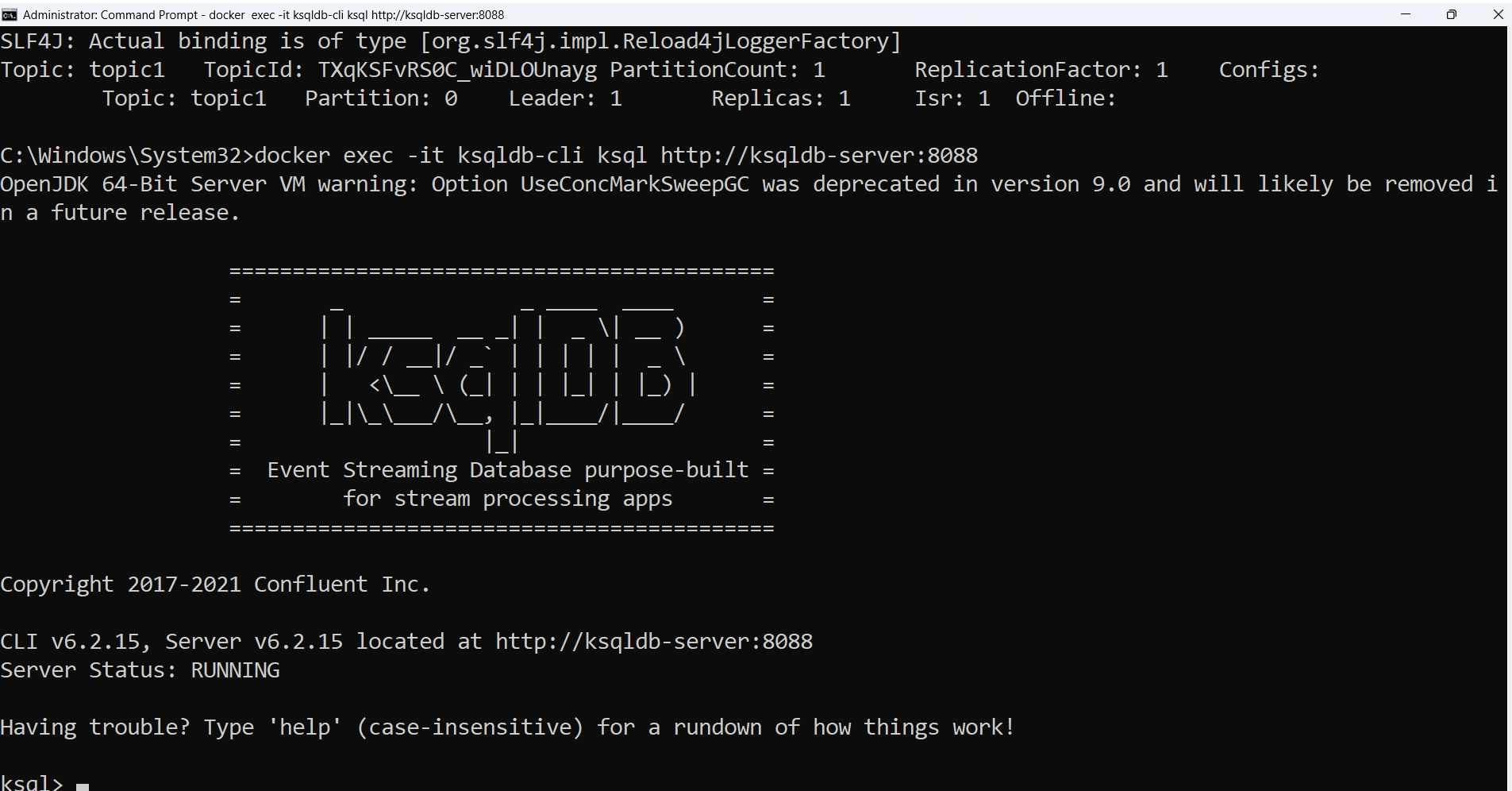
Confluent Kafka using Local

- docker exec -t broker kafka-topics --bootstrap-server broker:29092 --topic topic1 --describe



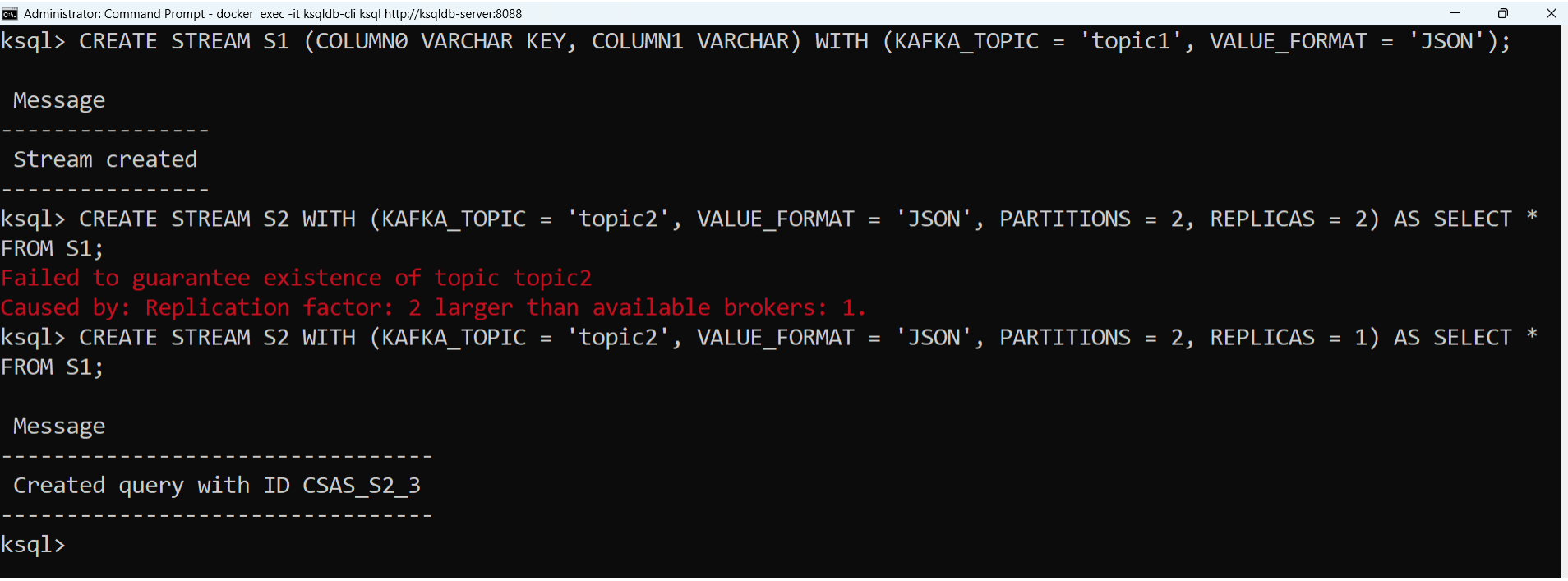
KSQLDB

- docker exec -it ksqldb-cli ksql http://ksqldb-server:8088



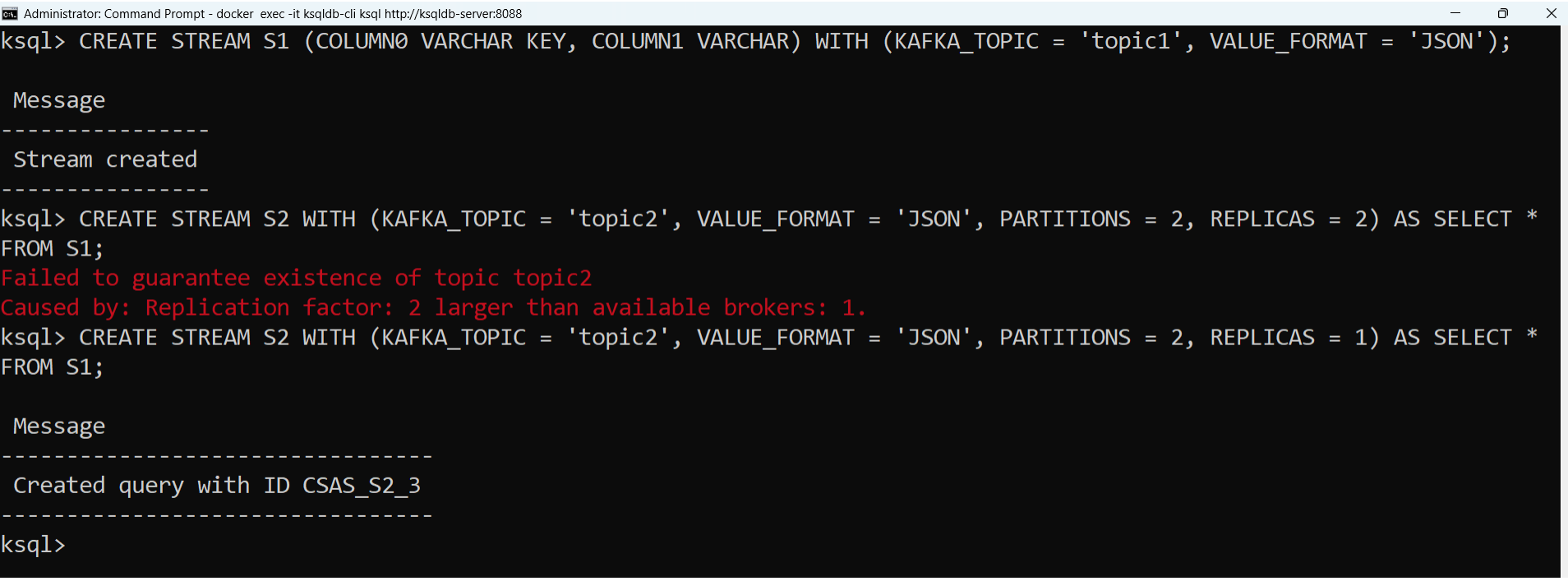
KSQLDB

- CREATE STREAM S1 (COLUMN0 VARCHAR KEY, COLUMN1 VARCHAR) WITH (KAFKA_TOPIC = 'topic1', VALUE_FORMAT = 'JSON');



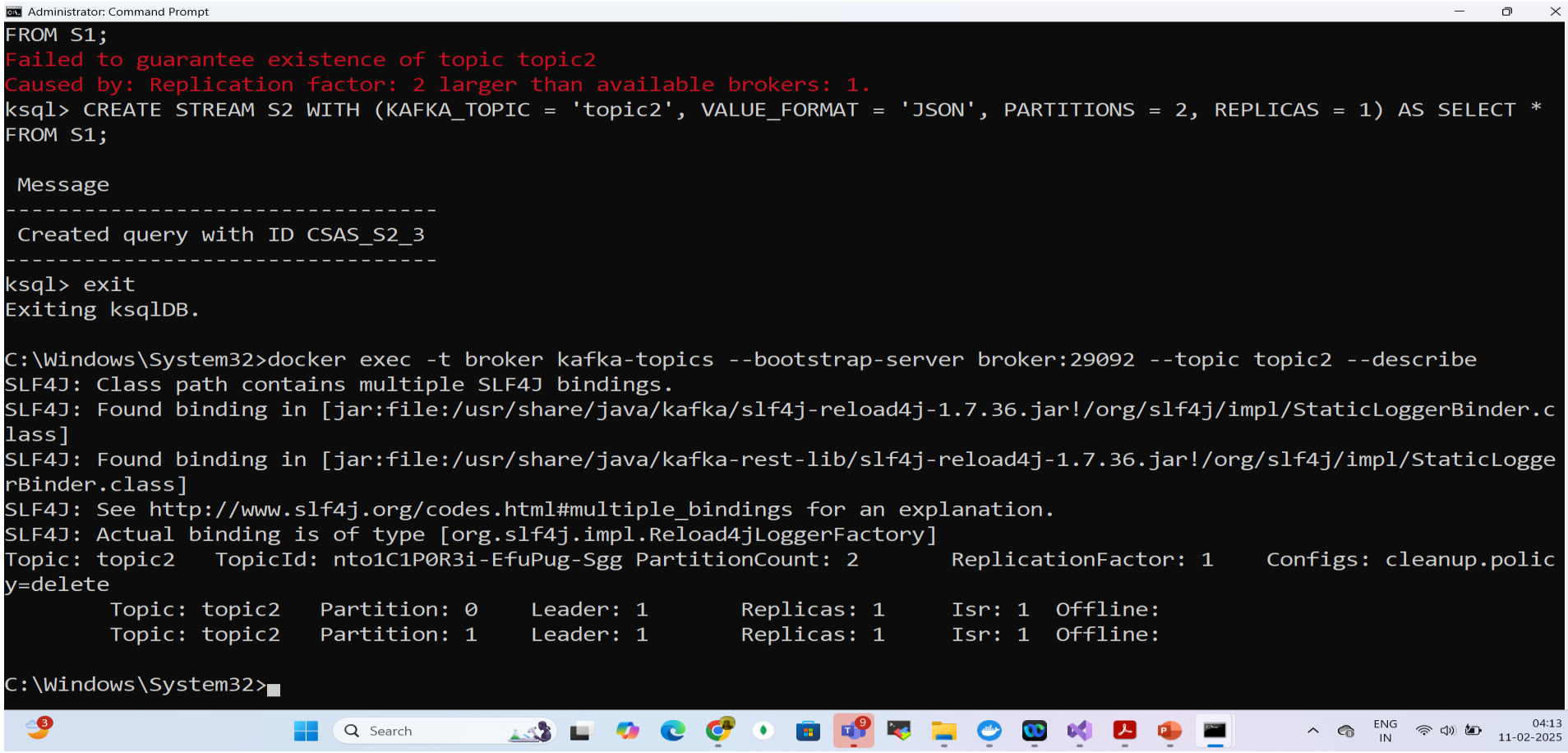
KSQLDB

- CREATE STREAM S2 WITH (KAFKA_TOPIC = 'topic2', VALUE_FORMAT = 'JSON', PARTITIONS = 2, REPLICAS = 2) AS SELECT *
- FROM S1;



Confluent Kafka using Local

- docker exec -t broker kafka-topics --bootstrap-server broker:29092 --topic topic2 --describe



Questions

Module Summary

- What is Grafana
- How Grafana Works
- Grafana Installation
- DataSources
- Dashboards