Apache Kafka Essentials

Building Scalable and Reliable Stream Processing Applications With Apache Kafka and Flink

SUDIP SENGUPTA
TECHNICAL WRITER, BROLLYCA

CONTENTS

With the changing tech landscape, several trends have emerged in recent years. First, with the increasing adoption of the cloud-native framework, microservices architectures are now a preferred choice for application development. These loosely coupled microservices enable enhanced scalability, fault isolation, and rapid deployment across hybrid cloud environments. Second, the diversity, volume, and velocity of the data that an enterprise wants to collect for decision making continues to grow.

Additionally, there is a growing need for an enterprise to make real-time decisions on collected data. Undoubtedly, the increasing application of high-performance data analytics to support modern businesses is considered central to their rapid decision making, augmented predictive analyses, and factual strategic initiatives.

Apache Kafka is a streaming engine for collecting, caching, and processing high volumes of data in real time. The distributed event store typically serves as a central data hub that accepts all enterprise data for aggregation, transformation, enrichment, ingestion, and analysis. The data can then be used for continuous processing or fed into other systems and applications in real time.

In this Refcard, we'll explore the fundamentals of Apache Kafka and delve into how Apache Flink complements Kafka as a powerful stream processing framework.

ABOUT APACHE KAFKA

Kafka was originally developed at LinkedIn in 2010 and later open sourced via the Apache Software Foundation in 2012. The platform comprises three main components: Pub/Sub, Kafka Connect, and Kafka Streams.

The role of each component is summarized in the table to the right.

COMPONENT	ROLE
Pub/Sub	Storing and delivering data efficiently and reliably at scale
Kafka Connect	Integrating Kafka with external data sources and data sinks
Kafka Streams	Processing data in Kafka in real time

The main benefits of Kafka are:

- **High throughput** – Kafka delivers high throughput, with clusters capable of handling terabytes of data per second.
- **High availability** – Redundancy is ensured through data replication across multiple brokers, allowing the system to tolerate failures without data loss.
- **High scalability** – New servers can be added over time to scale out the system.

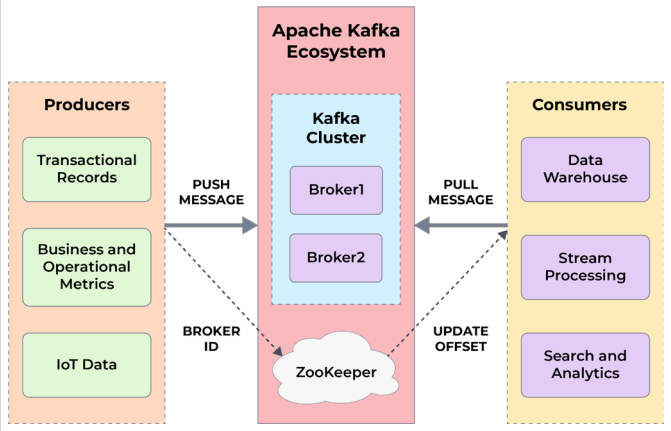
- **Easy integration** – Seamless integration with external data sources or data sinks.
- **Real-time processing** – Kafka Streams provides a native stream processing library for building real-time applications.
- **Fault isolation** – Kafka's distributed architecture ensures that failures in one part of the system don't impact others.
- **Client library availability** – Kafka allows for event stream processing in multiple coding languages.

While Kafka Streams has traditionally been a popular choice for real-time data processing within Kafka, the industry is shifting toward alternative stream processing frameworks like Apache Flink. We'll delve into this trend in more detail in the upcoming sections.

APACHE KAFKA FUNDAMENTALS

Apache Kafka runs in distributed clusters, with each cluster node being referred to as a broker. Kafka Connect integrates Kafka instances on brokers with producers and consumers — clients that produce and consume event data, respectively. All these components rely on the publish-subscribe durable messaging ecosystem to enable instant exchange of event data between servers, processes, and applications.

Figure 1: Apache Kafka as a central real-time hub



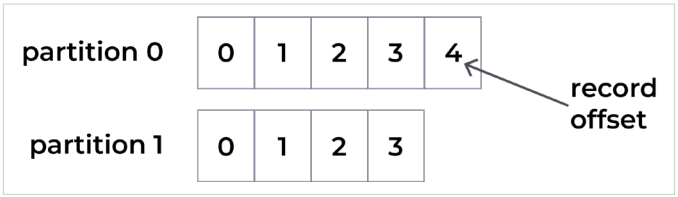
PUB/SUB IN APACHE KAFKA

The first component in Kafka deals with the production and consumption of the data. The following table describes a few key concepts in Kafka:

CONCEPT	DESCRIPTION
Topic	Defines a logical name for producing and consuming records
Partition	Defines a non-overlapping subset of records within a topic
Offset	A unique sequential number assigned to each record within a topic partition
Record	Contains a key, value, timestamp, and list of headers
Broker	Server where records are stored; multiple brokers can be used to form a cluster

Figure 2 depicts a topic with two partitions. Partition 0 has 5 records, with offsets from 0 to 4, and partition 1 has 4 records, with offsets from 0 to 3.

Figure 2: Partitions in a topic



The following code snippet shows how to produce records to the topic, `"test"`, using the Java API:

```
Properties props = new Properties();
   props.put("bootstrap.servers",
      "localhost:9092");
props.put("key.serializer",
   "org.apache.kafka.common.serialization.
StringSerializer");
props.put("value.serializer",
   "org.apache.kafka.common.serialization.
StringSerializer");
Producer<String, String> producer = new
   KafkaProducer<>(props);
producer.send(
    new ProducerRecord<String, String>("test",
"key", "value"));
```

In the example above, both the key and value are strings, so we are using a `StringSerializer`. It's possible to customize the serializer when types become more complex. The following code snippet shows how to consume records with key and value strings in Java:

```
Properties props = new Properties();  props.
put("bootstrap.servers", "localhost:9092");
props.put("key.deserializer",
   "org.apache.kafka.common.serialization.
StringDeserializer");
props.put("value.deserializer",
   "org.apache.kafka.common.serialization.
StringDeserializer");
KafkaConsumer<String, String> consumer =
   new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("test"));
while (true) {
   ConsumerRecords<String, String> records =
      consumer.poll(100);
 for (ConsumerRecord<String, String> record :
records)
      System.out.printf("offset=%d, key=%s,
value=%s",
         record.offset(), record.key(), record.
value());
   consumer.commitSync();
}
```

Records within a partition are always delivered to the consumer in offset order. By saving the offset of the last consumed record from each partition, the consumer can resume from where it left off after a restart. In the example above, we use the `commitSync()` API to save the offsets explicitly after consuming a batch of records. Users can also save the offsets automatically by setting the property, `enable.auto.commit`, to `true`.

A record in Kafka is not removed from the broker immediately after it is consumed. Instead, it is retained according to a configured retention policy. The following are two common retention policies:

- `log.retention.hours` – number of hours to keep a record on the broker
- `log.retention.bytes` – maximum size of records retained in a partition

While Kafka Streams could be used to process these consumed records, Apache Flink provides a compelling alternative with a richer set of features. Flink excels in state management and windowing operations, and it offers more flexible deployment options compared to Kafka Streams.

Here's an example of how to consume data from the `"test"` topic using Apache Flink:

```
import org.apache.flink.api.common.eventtime.
WatermarkStrategy;
import org.apache.flink.api.common.serialization.
SimpleStringSchema;
import org.apache.flink.connector.kafka.source.
FlinkKafkaConsumer;
import org.apache.flink.connector.kafka.source.
enumerator.initializer.OffsetsInitializer;
import org.apache.flink.streaming.api.datastream.
DataStream;
import org.apache.flink.streaming.api.environment.
StreamExecutionEnvironment;

public class FlinkKafkaConsumerExample {

    public static void main(String[] args) throws
Exception {
        // Set up the streaming execution
environment
        final StreamExecutionEnvironment
env = StreamExecutionEnvironment.
getExecutionEnvironment();

        // Configure the Kafka consumer
        FlinkKafkaConsumer<String> consumer = new
FlinkKafkaConsumer<>(
                "test", new SimpleStringSchema(),
properties)

               .setStartFromEarliest(); // Read
from the beginning of the topic
```

CODE CONTINUES IN NEXT COLUMN

```
        // Create a DataStream from the Kafka
consumer
        DataStream<String> stream = env.
addSource(consumer);

        // Process the stream (e.g., print the
messages)
        stream.print();

        // Execute the Flink program
        env.execute("Flink Kafka Consumer Example");
    }
}
```

Note that in contrast to the Kafka consumer code, which focuses on basic consumption, the above example using Flink demonstrates a more streamlined approach to creating a data stream from Kafka. It is Flink's `DataStream` API that provides a powerful foundation for building complex stream processing pipelines.

KAFKA CONNECT

The second component is Kafka Connect, a framework that makes it easy to stream data between Kafka and other systems. The framework uses connectors as pre-built components to integrate Kafka with various external systems for handling data transfer. Users can deploy a Connect cluster and run various connectors to import data from different sources into Kafka (through Source Connectors) and export data from Kafka further (through Sink Connectors) to storage platforms such as HDFS, S3, or Elasticsearch.

The benefits of using Kafka Connect are:

- Parallelism and fault tolerance
- Avoidance of ad hoc code by reusing existing connectors
- Built-in offset and configuration management

QUICK START FOR KAFKA CONNECT

The following steps show how to run the existing file connector in standalone mode to copy the content from a source file to a destination file via Kafka:

1. Prepare some data in a source file:

```
> echo -e \"hello\nworld\" > test.txt
```

2. Start a file source and a file sink connector:

```
> bin/connect-standalone.sh
config/connect-file-source.properties
config/connect-file-sink.properties
```

3. Verify the data in the destination file:

```
> more test.sink.txt
hello
```

4. Verify the data in Kafka:

```
> bin/kafka-console-consumer.sh
      --bootstrap-server localhost:9092
      --topic connect-test
      --from-beginning
   {"schema":{"type":"string",
         "optional":false},
      "payload":"hello"}
   {"schema":{"type":"string",
         "optional":false},
      "payload":"world"}
```

In the example above, the data in the source file, `test.txt`, is first streamed into a Kafka topic, `connect-test`, through a file source connector. The records in `connect-test` are then streamed into the destination file, `test.sink.txt`. If a new line is added to `test.txt`, it will show up immediately in `test.sink.txt`. Note that we achieve this by running two connectors without writing any custom code.

Connectors are powerful tools that allow for integration of Apache Kafka into many other systems. There are many open-source and commercially supported options for integrating Apache Kafka — both at the connector layer as well as through an integration services layer — that can provide much more flexibility in message transformation.

TRANSFORMATIONS IN CONNECT

Connect is primarily designed to stream data between systems as-is, whereas Kafka Streams is designed to perform complex transformations once the data is in Kafka. That said, Kafka Connect provides a mechanism used to perform simple transformations per record. The following example shows how to enable a couple of transformations in the file source connector:

1. Add the following lines to `connect-file-source.properties`:

```
transforms=MakeMap, InsertSource
transforms.MakeMap.type=org.apache.kafka
   .connect.transforms.HoistField$Value
transforms.MakeMap.field=line
transforms.InsertSource.type=org.apache
   .kafka.connect.transforms
   .InsertField$Value
transforms.InsertSource.static.field=
   data_source
transforms.InsertSource.static.value=
   test-file-source
```

2. Start a file source connector:

```
> bin/connect-standalone.sh
config/connect-file-source.properties
```

3. Verify the data in Kafka:

```
> bin/kafka-console-consumer.sh
      --bootstrap-server localhost:9092
   --topic connect-test
```

CODE CONTINUES IN NEXT COLUMN

```
{"line":"hello","data_source":"test
   -file-source"}
{"line":"world","data_source":"test
   -file-source"}
```

In step one above, we add two transformations, `MakeMap` and `InsertSource`, which are implemented by the classes, `HoistField$Value` and `InsertField$Value`, respectively. The first one adds a field name, `line`, to each input string. The second one adds an additional field, `data_source`, that indicates the name of the source file.

After applying the transformation logic, the data in the input file is now transformed to the output in step three. Because the last transformation step is more complex, we implement it with the Streams API (covered in more detail below):

```
final Serde<String> stringSerde = Serdes.String();
final Serde<Long> longSerde = Serdes.Long();
StreamsBuilder builder = new StreamsBuilder();
// build a stream from an input topic
KStream<String, String> source = builder.stream(
   "streams-plaintext-input",
   Consumed.with(stringSerde, stringSerde));
KTable<String, Long> counts = source
   .flatMapValues(value -> Arrays.asList(value.
toLowerCase().split(" ")))
   .groupBy((key, value) -> value)
   .count();
// convert the output to another topic
counts.toStream().to("streams-wordcount-output",
   Produced.with(stringSerde, longSerde));
```

While Kafka Streams could be used for further processing, Flink provides a richer API and a broader ecosystem for implementing complex transformations on this data. It's also important to understand why Flink is gaining traction. Flink is a true stream processing engine that handles events individually as they arrive, enabling low-latency operations and real-time results. It also excels at state management, allowing you to maintain and query application state for tasks like sessionization and windowing.

CONNECT REST API

In production, Kafka Connect usually runs in distributed mode and can be managed through REST APIs. To manage Connect in these environments, a REST API allows you to perform actions such as:

- Monitoring connector status
- Creating new connectors
- Updating connector configurations

The table on the following page lists the common APIs. See the Apache Kafka documentation for more information.

CONNECT REST API	ACTION
`GET /connectors`	Return a list of active connectors
`POST /connectors`	Create a new connector
`GET /connectors/{name}`	Get information for the connector
`GET /connectors/{name} / config`	Get configuration parameters for the connector
`PUT /connectors/{name} / config`	Update configuration parameters for the connector
`GET /connectors/{name} / status`	Get the current status of the connector

FLINK

Kafka offers the capability to perform stream processing on data stored within its topics. While Kafka Streams is a traditional, native library for this purpose, Apache Flink has emerged as a popular choice due to its advanced features and performance benefits. Flink offers several advantages over Kafka Streams for stream processing, including:

- High performance and scalability
- Rich API for complex transformations
- Support for stateful operations
- Exactly-once semantics for data accuracy
- Flexible deployment options

```
import org.apache.flink.api.common.functions.
FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.
DataStream;
import org.apache.flink.streaming.api.environment.
StreamExecutionEnvironment;
import org.apache.flink.util.Collector;

public class WordCount {

    public static void main(String[] args) throws
Exception {
        // Set up the streaming execution
environment
        final StreamExecutionEnvironment
env = StreamExecutionEnvironment.
getExecutionEnvironment();

        // Create a DataStream from the input source
(e.g., Kafka topic)
        DataStream<String> text = env.
fromElements("hello world", "flink is great"); //
Replace with Kafka source
        DataStream<Tuple2<String, Integer>> counts =
                // Split the lines into words
                text.flatMap(new Tokenizer())
                // Group by word and count
```

CODE CONTINUES IN NEXT COLUMN

```
                .keyBy(value -> value.f0)
                .sum(1);

        // Print the results
        counts.print();

        // Execute the program
        env.execute("Word Count Example");
    }

    public static final class Tokenizer implements
FlatMapFunction<String, Tuple2<String, Integer>> {
        @Override
        public void flatMap(String value,
Collector<Tuple2<String, Integer>> out) {
            String[] tokens = value.toLowerCase().
split("\\W+");
            for (String token : tokens) {
                if (token.length() > 0) {
                    out.collect(new Tuple2<>(token,
1));
                }
            }
        }
    }
}
```

The above Flink code snippet reads a stream of text, splits it into individual words, and then counts the occurrences of each word. To run this Flink example:

1. Download and install Apache Flink.
2. Package the Flink application: Package the `WordCount.java` code into a JAR file.
3. Submit the JAR file to the Flink cluster using the `flink run` command. (You'll need to configure the Flink application to *read-from* and *write-to* the appropriate Kafka topics.)

STREAM PROCESSING WITH FLINK: DATASTREAMS AND TABLES

Apache Flink provides two core abstractions for working with streaming data: DataStreams and Tables. These abstractions offer different ways to represent and process streaming data, catering to various use cases and programming styles.

- A **DataStream** represents a continuous flow of data records. Each record is treated as an individual event, similar to how records are handled in a KStream. DataStreams are well suited for scenarios where you need to process events in the order they arrive and perform operations such as filtering, mapping, and windowing.
- A **Table** represents a dynamic collection of records with a schema. Unlike DataStreams, Tables focus on the relational aspect of data, allowing you to express transformations using SQL-like queries. Tables are ideal for scenarios where you need to perform aggregations, joins, and other relational operations on streaming data.

Key differences are summarized in the table below:

FEATURE	DATASTREAM	TABLE
Data model	Unbounded sequence of individual records	Dynamic table with rows and columns
Processing	Record-at-a-time processing	Relational operations (e.g., aggregations, joins)
Use cases	Event processing, real-time analytics	Data analysis, continuous queries

Flink allows you to seamlessly switch between DataStreams and Tables, enabling you to leverage the strengths of both abstractions within your stream processing applications. Using DataStreams, you can process each event individually to track user sessions or detect anomalies. On the other hand, with Tables, you can aggregate events to calculate metrics such as the number of active users or the average session duration.

Let's consider a stream of events with a key and a numeric value to understand their differences better:

KEY	VALUE
`"k1"`	`2`
`"k1"`	`5`

When processed as a **DataStream**, both records are treated as independent events. If we were to sum the values, the result would be `7` (`2` + `5`).

When processed as a **Table**, the second record with key `"k1"` is treated as an update to the existing `"k1"` record. Therefore, if we were to sum the values in the table, the result would be `5` (the latest value associated with `"k1"`).

FLINK DATASTREAM API AND OPERATORS

Flink's DataStream API provides a rich set of operators for transforming and processing streaming data. These operators allow you to perform various operations on DataStreams, such as filtering, mapping, joining, and aggregating. Here are some commonly used Flink DataStream operators:

1. `filter(Predicate)`: Create a new DataStream consisting of all records of this stream that satisfy the given predicate. Example:

```
DataStream<Integer> input = env.fromElements(1, 2, 3, 4, 5, 6, 7);
DataStream<Integer> output = input.filter(value -> value > 5); // Output: 6, 7
```

2. `map(MapFunction)`: Transform each record of the input stream into a new record in the output stream. Example:

```
DataStream<Integer> input = env.fromElements(1, 2, 3); DataStream<Integer> output = input.map(value -> value * 2); // Output: 2, 4, 6
```

3. `keyBy(KeySelector)`: Group the records by a key. This is a crucial step for performing aggregations and other stateful operations. Example:

```
DataStream<Tuple2<String, Integer>> input = env.
fromElements(

Tuple2.of("a", 1),

Tuple2.of("b", 2),

Tuple2.of("a", 3));
KeyedStream<Tuple2<String, Integer>, String>
keyedStream = input.keyBy(value -> value.f0); // Key by the first element (String)
```

4. `reduce(ReduceFunction)`: Combine the values of records in a keyed stream. Example:

```
DataStream<Tuple2<String, Integer>> input = // ... (keyed stream as in the previous example)
DataStream<Tuple2<String, Integer>> output = keyedStream.reduce(

(value1, value2) -> Tuple2.of(value1.f0, value1.f1 + value2.f1)
                                                    );
// Output: ("a", 4), ("b", 2)
```

5. `window(WindowAssigner)`: Group records into windows for performing aggregations or other operations over a specific time period or number of events. Example:

```
DataStream<Tuple2<String, Integer>> input = // ... (keyed stream)
DataStream<Tuple2<String, Integer>> output = keyedStream

.window(TumblingEventTimeWindows.of(Time.
seconds(5)))

.sum(1); // Sum the second element (Integer) within 5-second windows
```

6. `join(DataStream, KeySelector, KeySelector, ValueJoiner)`: Join records from two DataStreams based on their keys. Example:

```
DataStream<Tuple2<String, Integer>> stream1 = // ...
DataStream<Tuple2<String, String>> stream2 = // ...
DataStream<Tuple3<String, Integer, String>> output = stream1

.join(stream2)

.where(value -> value.f0) // Key selector for stream1

.equalTo(value -> value.f0) // Key selector for stream2
```

CODE CONTINUES ON NEXT PAGE

```
.window(TumblingEventTimeWindows.of(Time.
seconds(10)))

.apply(

(value1, value2) -> Tuple3.of(value1.f0, value1.f1,
value2.f1)

); // Join within 10-second windows
```

The above is just a small selection of the many operators available in Flink's DataStream API. Flink also provides operators for connecting to various data sources and sinks, performing stateful operations, and handling event time. Additional details on the respective set of operations and examples can be found in the Apache Flink DataStream API documentation.

STATE MANAGEMENT AND FAULT TOLERANCE IN FLINK

While processing data in real time, a Flink application often needs to maintain state. State is information that is relevant to the processing of events, such as aggregations, windowed data, or machine learning models. For example, in a word count application, the state would be the current count of each word encountered.

As a key feature over KStreams, Flink excels at state management, providing efficient ways to store, access, and update this state. It offers various state back ends to suit different needs, such as in-memory state for high performance, file-system-based state for durability, and RocksDB state for handling large state sizes.

Figure 3: RocksDB in a Flink cluster node

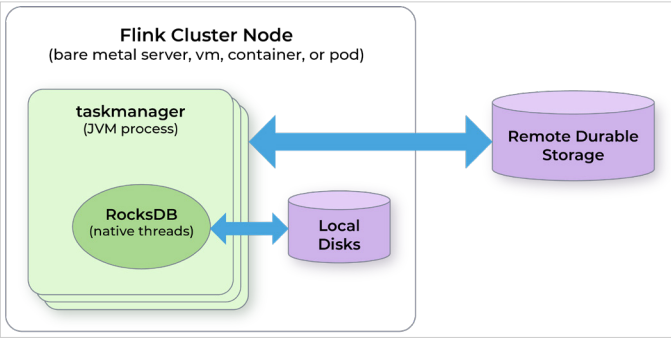


Image adapted from Apache Flink docs

To mitigate potential failures in the system, Flink guarantees *exactly-once* processing semantics. As a result, each record in the stream will be processed exactly once, and the results will be consistent, regardless of any disruptions. A typical way of achieving this is through a combination of techniques, including checkpointing (periodically saving the state of the application), robust state management, and transactional sinks (ensuring that output data is written only once).

Governing data in motion is often a shared responsibility between developers, DataOps, and business development teams. Flink provides built-in observability metrics and monitoring capabilities that allow you to track key metrics such as throughput (number of records processed per second), latency (time taken to process a record), and resource utilization (CPU, memory). You can also integrate Flink with external monitoring tools like Prometheus and Grafana for more comprehensive monitoring and visualization.

Though it is important to note that Flink, on its own, doesn't include a built-in event portal to catalog event streams and visualize the complete topology of data pipelines. However, it offers features and integration points that enable you to achieve this functionality. To understand the flow of data, there are various tools and APIs for tracking data lineage to trace the origin and transformations of data as it moves through your pipelines. If you detect an anomaly in the output data, data lineage can help you pinpoint the source of the issue and the transformations applied.

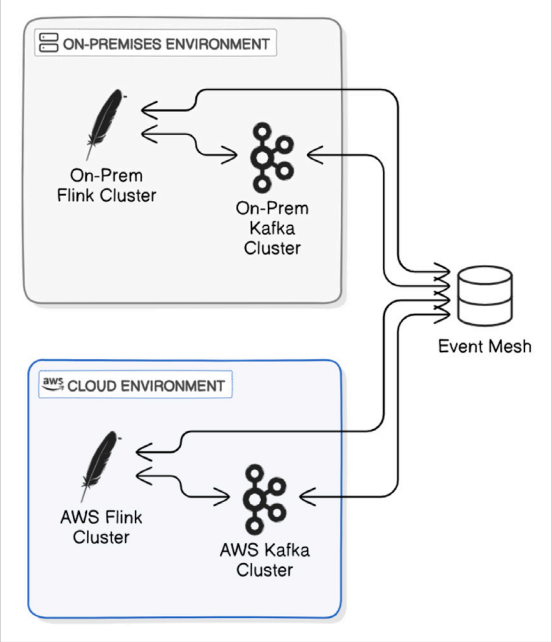
For example, you can use OpenLineage as a standard for capturing and sharing data lineage information or Marquez as a lineage tracking system that can be integrated with Flink.

EXTENDING APACHE KAFKA WITH FLINK

While Kafka and Flink provide a powerful foundation for event streaming and stream processing, you might need to extend their capabilities to support specific requirements of modern data-driven applications. Here are some approaches to consider:

- **Low-code stream processing** – To reduce manual effort redundancy, consider using low-code approaches that require minimal to no coding for redundant tasks. Flink SQL, for example, allows you to express transformations and analysis using SQL queries, minimizing the need for complex Java or Scala code. The approach can be particularly useful for common tasks like filtering, aggregations, and joins.
- **Linking on-prem and cloud-based clusters** – As your streaming infrastructure grows, you might need to connect Flink and Kafka deployments across different environments, such as on-premises data centers and multiple cloud providers. Consider solutions like an event mesh to create a unified fabric for seamless data flow and interoperability across these environments.
- **Filtering data streams** – While Flink provides a rich set of operators for filtering and transforming data, you might encounter scenarios that require more specialized or complex logic. In such cases, you can leverage Flink's extensibility to implement custom functions or integrate with external libraries to achieve the desired functionality.
- **Data sharing and replication** – To ensure high availability, disaster recovery, or data synchronization across different Kafka clusters, consider using tools and techniques for data replication. MirrorMaker 2 is a Kafka-native tool that can replicate topics between clusters. You can also explore Kafka's active/active replication feature for more advanced replication scenarios.

Figure 4: Hybrid/multi-cloud deployment with event mesh



For more information on low-code stream processing options, including Flink SQL access to Apache Kafka, please see the additional resources below.

SCALING DATA STREAMS

With the increasing popularity of real-time stream processing and the rise of event-driven architectures, a number of alternatives have started to gain traction for real-time data distribution. Apache Kafka is the flavor of choice for distributed high-volume data streaming; however, many implementations commonly struggle with building solutions at scale when the application's requirements go beyond a single data center or single location.

While Apache Kafka is purpose-built for real-time data distribution and stream processing, it may not fit the requirements of every enterprise application due to its constraints with architecture, throughput, and amount of data ingested. Some alternatives to Apache Kafka include:

- Apache Pulsar – a lower-latency, higher-throughput event management platform that relies on geo-replication for enhanced performance.
- Amazon Kinesis – an AWS streaming service that supports Java, Android, .NET, and Go SDKs.
- Apache Spark – a unified, open-source analytics engine that is mainly built for streaming data in ML and AI applications.
- Google Cloud Pub/Sub – a scalable message queue and stream analytics platform with enterprise support for event-driven applications.

For more information on comparisons between Apache Kafka and other data distribution solutions, please see the additional resources.

CONCLUSION

Apache Kafka has become the de facto standard for high-performance, distributed data streaming. It has a large and growing community of developers, corporations, and applications that are supporting, maintaining, and leveraging it. Flink complements Kafka by providing a robust and scalable platform for performing complex stream processing tasks. If you are building an event-driven architecture or looking for a way to stream data in real time, Apache Kafka is a clear leader in providing a proven, robust platform for enabling stream processing and enterprise communication.

Additional resources:

- Apache Kafka Documentation – http://kafka.apache.org/documentation/
- Apache NiFi website – http://nifi.apache.org/
- "Real-Time Stock Processing With Apache NiFi and Apache Kafka, Part 1" – https://dzone.com/articles/real-time-stock-processing-with-apache-nifi-and-ap
- Apache Kafka Summit website – http://kafka-summit.org/
- Apache Kafka Mirroring and Replication – http://cwiki.apache.org/confluence/display/KAFKA/KIP-382%3A+MirrorMaker+2.0
- Apache Flink Documentation - https://flink.apache.org/documentation/
- Real-time sessionization pipeline using Apache Flink – https://github.com/vraja2/flink-sessionization
- *Open-Source Data Management Practices and Patterns Refcard* – https://dzone.com/refcardz/open-source-data-management-practices-and-patterns

WRITTEN BY SUDIP SENGUPTA,
PRINCIPAL ARCHITECT & TECHNICAL WRITER, JAVELYNN

Sudip Sengupta is a TOGAF Certified Solutions Architect with more than 19 years of experience working for global majors such as CSC, Hewlett Packard Enterprise, and DXC Technology. Sudip now works as a full-time tech writer, focusing on Cloud, DevOps, SaaS, and cybersecurity. When not writing or reading, he's likely on the squash court or playing chess.

DZone
3343 Perimeter Hill Dr, Suite 100
Nashville, TN 37211
888.678.0399 | 919.678.0300

At DZone, we foster a collaborative environment that empowers developers and tech professionals to share knowledge, build skills, and solve problems through content, code, and community. We thoughtfully — and with intention — challenge the status quo and value diverse perspectives so that, as one, we can inspire positive change through technology.