ARCHITECUTRE FOR KAFKA:

KAFKA:

Take a look at the following illustration. It shows the cluster diagram of Kafka.



The following table describes each of the components shown in the above diagram.

|  |  |
| --- | --- |
| **S.No** | **Components and Description** |
| 1 | **Broker**  Kafka cluster typically consists of multiple brokers to maintain load balance. Kafka brokers are stateless, so they use ZooKeeper for maintaining their cluster state. One Kafka broker instance can handle hundreds of thousands of reads and writes per second and each bro-ker can handle TB of messages without performance impact. Kafka broker leader election can be done by ZooKeeper. |
| 2 | **ZooKeeper**  ZooKeeper is used for managing and coordinating Kafka broker. ZooKeeper service is mainly used to notify producer and consumer about the presence of any new broker in the Kafka system or failure of the broker in the Kafka system. As per the notification received by the Zookeeper regarding presence or failure of the broker then pro-ducer and consumer takes decision and starts coordinating their task with some other broker. |
| 3 | **Producers**  Producers push data to brokers. When the new broker is started, all the producers search it and automatically sends a message to that new broker. Kafka producer doesn’t wait for acknowledgements from the broker and sends messages as fast as the broker can handle. |
| 4 | **Consumers**  Since Kafka brokers are stateless, which means that the consumer has to maintain how many messages have been consumed by using partition offset. If the consumer acknowledges a particular message offset, it implies that the consumer has consumed all prior messages. The consumer issues an asynchronous pull request to the broker to have a buffer of bytes ready to consume. The consumers can rewind or skip to any point in a partition simply by supplying an offset value. Consumer offset value is notified by ZooKeeper. |

workflow of Kafka:

Kafka is simply a collection of topics split into one or more partitions. A Kafka partition is a linearly ordered sequence of messages, where each message is identified by their index (called as offset). All the data in a Kafka cluster is the disjointed union of partitions. Incoming messages are written at the end of a partition and messages are sequentially read by consumers. Durability is provided by replicating messages to different brokers.

Kafka provides both pub-sub and queue based messaging system in a fast, reliable, persisted, fault-tolerance and zero downtime manner. In both cases, producers simply send the message to a topic and consumer can choose any one type of messaging system depending on their need. Let us follow the steps in the next section to understand how the consumer can choose the messaging system of their choice.

Workflow of Pub-Sub Messaging

Following is the step wise workflow of the Pub-Sub Messaging −

* Producers send message to a topic at regular intervals.
* Kafka broker stores all messages in the partitions configured for that particular topic. It ensures the messages are equally shared between partitions. If the producer sends two messages and there are two partitions, Kafka will store one message in the first partition and the second message in the second partition.
* Consumer subscribes to a specific topic.
* Once the consumer subscribes to a topic, Kafka will provide the current offset of the topic to the consumer and also saves the offset in the Zookeeper ensemble.
* Consumer will request the Kafka in a regular interval (like 100 Ms) for new messages.
* Once Kafka receives the messages from producers, it forwards these messages to the consumers.
* Consumer will receive the message and process it.
* Once the messages are processed, consumer will send an acknowledgement to the Kafka broker.
* Once Kafka receives an acknowledgement, it changes the offset to the new value and updates it in the Zookeeper. Since offsets are maintained in the Zookeeper, the consumer can read next message correctly even during server outrages.
* This above flow will repeat until the consumer stops the request.
* Consumer has the option to rewind/skip to the desired offset of a topic at any time and read all the subsequent messages.

Workflow of Queue Messaging / Consumer Group

In a queue messaging system instead of a single consumer, a group of consumers having the same Group ID will subscribe to a topic. In simple terms, consumers subscribing to a topic with same Group ID are considered as a single group and the messages are shared among them. Let us check the actual workflow of this system.

* Producers send message to a topic in a regular interval.
* Kafka stores all messages in the partitions configured for that particular topic similar to the earlier scenario.
* A single consumer subscribes to a specific topic, assume Topic-01 with Group ID as Group-1.
* Kafka interacts with the consumer in the same way as Pub-Sub Messaging until new consumer subscribes the same topic, Topic-01 with the same Group ID as Group-1.
* Once the new consumer arrives, Kafka switches its operation to share mode and shares the data between the two consumers. This sharing will go on until the number of con-sumers reach the number of partition configured for that particular topic.
* Once the number of consumer exceeds the number of partitions, the new consumer will not receive any further message until any one of the existing consumer unsubscribes. This scenario arises because each consumer in Kafka will be assigned a minimum of one partition and once all the partitions are assigned to the existing consumers, the new consumers will have to wait.
* This feature is also called as Consumer Group. In the same way, Kafka will provide the best of both the systems in a very simple and efficient manner.

Role of ZooKeeper

A critical dependency of Apache Kafka is Apache Zookeeper, which is a distributed configuration and synchronization service. Zookeeper serves as the coordination interface between the Kafka brokers and consumers. The Kafka servers share information via a Zookeeper cluster. Kafka stores basic metadata in Zookeeper such as information about topics, brokers, consumer offsets (queue readers) and so on.

Since all the critical information is stored in the Zookeeper and it normally replicates this data across its ensemble, failure of Kafka broker / Zookeeper does not affect the state of the Kafka cluster. Kafka will restore the state, once the Zookeeper restarts. This gives zero downtime for Kafka. The leader election between the Kafka broker is also done by using Zookeeper in the event of leader failure.

# Streams Architecture

This section describes how Kafka Streams works underneath the covers.

Kafka Streams simplifies application development by building on the Apache Kafka® producer and consumer APIs, and leveraging the native capabilities of Kafka to offer data parallelism, distributed coordination, fault tolerance, and operational simplicity.

Here is the anatomy of an application that uses the Kafka Streams API. It provides a logical view of a Kafka Streams application that contains multiple stream threads, that each contain multiple stream tasks.

Diagram

Description automatically generated

Tip

To learn how Kafka transactions provide you with accurate, repeatable results from chains of many stream processors or microservices, connected via event streams, see [Building Systems Using Transactions in Apache Kafka](https://developer.confluent.io/learn/kafka-transactions-and-guarantees/).

## Processor Topology

A **processor topology** or simply **topology** defines the stream processing computational logic for your application, i.e., how input data is transformed into output data. A topology is a graph of [stream processors](https://docs.confluent.io/platform/current/streams/concepts.html#streams-concepts-processor) (nodes) that are connected by [streams](https://docs.confluent.io/platform/current/streams/concepts.html#streams-concepts-stream) (edges) or shared [state stores](https://docs.confluent.io/platform/current/streams/architecture.html#streams-architecture-state). There are two special processors in the topology:

* **Source Processor**: A source processor is a special type of [stream processor](https://docs.confluent.io/platform/current/streams/concepts.html#streams-concepts-processor) that does not have any upstream processors. It produces an input stream to its topology from one or multiple Kafka topics by consuming records from these topics and forward them to its down-stream processors.
* **Sink Processor**: A sink processor is a special type of stream processor that does not have down-stream processors. It sends any received records from its up-stream processors to a specified Kafka topic.

A stream processing application – i.e., your application – may define one or more such topologies, though typically it defines only one. Developers can define topologies either via the [low-level Processor API](https://docs.confluent.io/platform/current/streams/developer-guide/processor-api.html#streams-developer-guide-processor-api) or via the [Kafka Streams DSL](https://docs.confluent.io/platform/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl), which builds on top of the former.

[Diagram

Description automatically generated](https://docs.confluent.io/platform/current/_images/streams-architecture-topology.jpg)

A processor topology is merely a logical abstraction for your stream processing code. At runtime, the logical topology is instantiated and replicated inside the application for parallel processing (see [Parallelism Model](https://docs.confluent.io/platform/current/streams/architecture.html#streams-architecture-parallelism-model)).

## Parallelism Model

### **Stream Partitions and Tasks**

The messaging layer of Kafka partitions data for storing and transporting it. Kafka Streams partitions data for processing it. In both cases, this partitioning is what enables data locality, elasticity, scalability, high performance, and fault tolerance.

Kafka Streams uses the concepts of **stream partitions** and **stream tasks** as logical units of its parallelism model. There are close links between Kafka Streams and Kafka in the context of parallelism:

* Each **stream partition** is a totally ordered sequence of data records and maps to a Kafka topic partition.
* A **data record** in the stream maps to a Kafka **message** from that topic.
* The **keys** of data records determine the partitioning of data in both Kafka and Kafka Streams, i.e., how data is routed to specific partitions within topics.

An application’s processor topology is scaled by breaking it into multiple stream tasks. More specifically, Kafka Streams creates a fixed number of stream tasks based on the input stream partitions for the application, with each task being assigned a list of partitions from the input streams (i.e., Kafka topics). The **assignment of stream partitions to stream tasks never changes**, hence the stream task is a fixed unit of parallelism of the application. Tasks can then instantiate their own processor topology based on the assigned partitions; they also maintain a buffer for each of its assigned partitions and process input data one-record-at-a-time from these record buffers. As a result stream tasks can be processed independently and in parallel without manual intervention.

Slightly simplified, the **maximum parallelism** at which your application may run is bounded by the maximum number of stream tasks, which itself is determined by maximum number of partitions of the input topic(s) the application is reading from. For example, if your input topic has 5 partitions, then you can run up to 5 applications instances. These instances will collaboratively process the topic’s data. If you run a larger number of app instances than partitions of the input topic, the “excess” app instances will launch but remain idle; however, if one of the busy instances goes down, one of the idle instances will resume the former’s work. We provide a more [detailed explanation and example](https://docs.confluent.io/platform/current/streams/faq.html#streams-faq-scalability-maximum-parallelism) in the FAQ.

Note

**Sub-topologies (also called sub-graphs):** If there are multiple processor topologies specified in a Kafka Streams application, each task only instantiates one of the topologies for processing. In addition, a single processor topology may be decomposed into independent sub-topologies (or sub-graphs). A sub-topology is a set of processors, that are all transitively connected as parent/child or via state stores in the topology. Hence, different sub-topologies exchange data via topics and don’t share any state stores. Each task may instantiate only one such sub-topology for processing. This further scales out the computational workload to multiple tasks.

[Diagram

Description automatically generated](https://docs.confluent.io/platform/current/_images/streams-architecture-tasks.jpg)

*Two tasks each assigned with one partition of the input streams.*

It is important to understand that Kafka Streams is not a resource manager, but a library that “runs” anywhere its stream processing application runs. Multiple instances of the application are executed either on the same machine, or spread across multiple machines and tasks can be [distributed automatically by the library](https://docs.confluent.io/platform/current/streams/architecture.html#streams-architecture-threads) to those running application instances. The assignment of partitions to tasks never changes; if an application instance fails, all its assigned tasks will be [restarted on other instances](https://docs.confluent.io/platform/current/streams/architecture.html#streams-architecture-fault-tolerance) and continue to consume from the same stream partitions.

Note

Topic partitions are assigned to tasks, and tasks are assigned to all threads over all instances, in a best-effort attempt to trade off load-balancing and stickiness of stateful tasks. For this assignment, Kafka Streams uses the [StreamsPartitionAssignor](https://github.com/apache/kafka/blob/trunk/streams/src/main/java/org/apache/kafka/streams/processor/internals/StreamsPartitionAssignor.java" \t "_blank) class and doesn’t let you change to a different assignor. If you try to use a different assignor, Kafka Streams ignores it.

### **Threading Model**

Kafka Streams allows the user to configure the number of **threads** that the library can use to parallelize processing within an application instance. Each thread can execute one or more stream tasks with their processor topologies independently.

[Diagram

Description automatically generated](https://docs.confluent.io/platform/current/_images/streams-architecture-threads.jpg)

*One stream thread running two stream tasks.*

Starting more stream threads or more instances of the application merely amounts to replicating the topology and having it process a different subset of Kafka partitions, effectively parallelizing processing. It is worth noting that there is no shared state amongst the threads, so no inter-thread coordination is necessary. This makes it very simple to run topologies in parallel across the application instances and threads. The assignment of Kafka topic partitions amongst the various stream threads is transparently handled by Kafka Streams leveraging [Kafka’s server-side coordination](https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Client-side+Assignment+Proposal) functionality.

As we described above, scaling your stream processing application with Kafka Streams is easy: you merely need to start additional instances of your application, and Kafka Streams takes care of distributing partitions across stream tasks that run in the application instances. You can start as many threads of the application as there are input Kafka topic partitions so that, across all running instances of an application, every thread (or rather, the stream tasks that the thread executes) has at least one input partition to process.

As of Kafka 2.8 you can scale stream threads much in the same way that you can scale your Kafka Streams clients. Simply add or remove stream threads and Kafka Streams takes care of redistributing the partitions. You may also add threads to replace stream threads that have died, eliminating the need to restart clients to recover the number of running threads.

### **Example**

To understand the parallelism model that Kafka Streams offers, let’s walk through an example.

Imagine a Kafka Streams application that consumes from two topics, A and B, with each having 3 partitions. If we now start the application on a single machine with the number of threads configured to 2, we end up with two stream threads instance1-thread1 and instance1-thread2. Kafka Streams will break this topology into three tasks because the maximum number of partitions across the input topics A and B is max(3, 3) == 3, and then distribute the six input topic partitions evenly across these three tasks; in this case, each task will process records from one partition of each input topic, for a total of two input partitions per task. Finally, these three tasks will be spread evenly – to the extent this is possible – across the two available threads, which in this example means that the first thread will run 2 tasks (consuming from 4 partitions) and the second thread will run 1 task (consuming from 2 partitions).

[Diagram

Description automatically generated](https://docs.confluent.io/platform/current/_images/streams-architecture-example-01.png)

Now imagine we want to scale out this application later on, perhaps because the data volume has increased significantly. We decide to start running the same application but with only a single thread on another, different machine. A new thread instance2-thread1 will be created, and input partitions will be re-assigned similar to:

[Diagram

Description automatically generated](https://docs.confluent.io/platform/current/_images/streams-architecture-example-02.png)

When the re-assignment occurs, some partitions – and hence their corresponding tasks including any local state stores – will be “migrated” from the existing threads to the newly added threads (here, stream task 2 from instance1-thread1 on the first machine was migrated to instance2-thread1 on the second machine). As a result, Kafka Streams has effectively rebalanced the workload among instances of the application at the granularity of Kafka topic partitions.

What if we wanted to add even more instances of the same application? We can do so until a certain point, which is when the number of running instances is equal to the number of available input partitions to read from. At this point, before it would make sense to start further application instances, we would first need to increase the number of partitions for topics A and B; otherwise, we would over-provision the application, ending up with idle instances that are waiting for partitions to be assigned to them, which may never happen.

## State

Kafka Streams provides so-called state stores, which can be used by stream processing applications to store and query data, which is an important capability when implementing [stateful operations](https://docs.confluent.io/platform/current/streams/concepts.html#streams-concepts-stateful-processing). The [Kafka Streams DSL](https://docs.confluent.io/platform/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl), for example, automatically creates and manages such state stores when you are calling stateful operators such as count() or aggregate(), or when you are [windowing a stream](https://docs.confluent.io/platform/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl-windowing).

Every stream task in a Kafka Streams application may embed one or more local state stores that can be accessed via APIs to store and query data required for processing. These state stores can either be a [RocksDB](http://rocksdb.org/" \t "_blank) database, an in-memory hash map, or another convenient data structure. Kafka Streams offers [fault-tolerance](https://docs.confluent.io/platform/current/streams/architecture.html#streams-architecture-fault-tolerance) and automatic recovery for local state stores.

[Diagram

Description automatically generated](https://docs.confluent.io/platform/current/_images/streams-architecture-states.jpg)

*Two stream tasks with their dedicated local state stores*

A Kafka Streams application is typically [running on many application instances](https://docs.confluent.io/platform/current/streams/developer-guide/running-app.html#streams-developer-guide-execution). Because Kafka Streams [partitions the data for processing it](https://docs.confluent.io/platform/current/streams/architecture.html#streams-architecture-tasks), an application’s entire state is spread across the local state stores of the application’s running instances. The Kafka Streams API lets you work with an application’s state stores both locally (e.g., on the level of an instance of the application) as well as in its entirety (on the level of the “logical” application), for example through stateful operations such as count() or through [Interactive Queries](https://docs.confluent.io/platform/current/streams/developer-guide/interactive-queries.html#streams-developer-guide-interactive-queries-discovery).

## Memory management

### **Record caches**

With Kafka Streams, you can specify the total memory (RAM) size that is used for an instance of a processing topology. This memory is used for internal caching and compacting of records before they are written to state stores, or forwarded downstream to other nodes. These caches differ slightly in implementation in the [DSL](https://docs.confluent.io/platform/current/streams/developer-guide/memory-mgmt.html#streams-developer-guide-memory-management-record-cache) and [Processor API](https://docs.confluent.io/platform/current/streams/developer-guide/memory-mgmt.html#streams-developer-guide-memory-management-state-store-cache).

The specified cache size is divided equally among the Kafka Stream threads of a topology. Memory is shared over all threads per instance. Each thread maintains a memory pool accessible by its tasks’ processor nodes for caching. Specifically, this is used by stateful processor nodes that perform aggregates and thus have a state store.

[Diagram

Description automatically generated](https://docs.confluent.io/platform/current/_images/streams-record-cache.png)

The cache has three functions. First, it serves as a read cache to speed up reading data from a state store. Second, it serves as a write-back buffer for a state store. A write-back cache allows for batching multiple records instead of sending each record individually to the state store. It also reduces the number of requests going to a state store (and its changelog topic stored in Kafka if it is a persistent state store) because records with the same key are compacted in cache. Third, the write-back cache reduces the number of records going to downstream processor nodes as well.

Thus, without requiring you to invoke any explicit processing operators in the API, these caches allow you to make trade-off decisions between:

* When using smaller cache sizes: larger rate of downstream updates with shorter intervals between updates.
* When using larger cache sizes: smaller rate of downstream updates with larger intervals between updates. Typically, this results reduced network IO to Kafka and reduced local disk IO to RocksDB-backed state stores, for example.

The final computation results are identical regardless of the cache size (including a disabled cache), which means it is safe to enable or disable the cache. It is not possible to predict when or how updates will be compacted because this depends on many factors, including:

* Cache size.
* Characteristics of the data being processed.
* Configuration parameters, for example commit.interval.ms.

For more information, see [Kafka Streams Memory Management](https://docs.confluent.io/platform/current/streams/developer-guide/memory-mgmt.html#streams-developer-guide-memory-management) in the Developer Guide.

## Fault Tolerance

Kafka Streams builds on fault-tolerance capabilities integrated natively within Kafka. Kafka partitions are highly available and replicated; so when stream data is persisted to Kafka it is available even if the application fails and needs to re-process it. Tasks in Kafka Streams leverage the fault-tolerance capability offered by the [Kafka consumer client](http://www.confluent.io/blog/tutorial-getting-started-with-the-new-apache-kafka-0.9-consumer-client) to handle failures. If a task runs on a machine that fails, Kafka Streams automatically restarts the task in one of the remaining running instances of the application.

In addition, Kafka Streams makes sure that the local state stores are robust to failures, too. For each state store, it maintains a replicated changelog Kafka topic in which it tracks any state updates. These changelog topics are partitioned as well so that each local state store instance, and hence the task accessing the store, has its own dedicated changelog topic partition. [Log compaction](https://docs.confluent.io/platform/current/kafka/design.html#log-compaction) is enabled on the changelog topics so that old data can be purged safely to prevent the topics from growing indefinitely. If tasks run on a machine that fails and are restarted on another machine, Kafka Streams guarantees to restore their associated state stores to the content before the failure by replaying the corresponding changelog topics prior to resuming the processing on the newly started tasks. As a result, failure handling is completely transparent to the end user.

Tip

**Optimization:** The cost of task (re)initialization typically depends primarily on the time for restoring the state by replaying the state stores’ associated changelog topics. To minimize this restoration time, you can configure your applications to have standby replicas of local states, which are fully replicated copies of the state. When a task migration happens, Kafka Streams assigns a task to an application instance where such a standby replica already exists, to minimize the task (re)initialization cost. For more information, see num.standby.replicas at [Optional configuration parameters](https://docs.confluent.io/platform/current/streams/developer-guide/config-streams.html#streams-developer-guide-optional-configs) in the Developer Guide. Starting in 2.6, Kafka Streams guarantees that a task is assigned to an instance with a fully caught-up local copy of the state exists, if such an instance exists. Standby tasks increase the likelihood that a caught-up instance exists in the case of a failure.

## Flow Control with Timestamps

Kafka Streams regulates the progress of streams by the timestamps of data records by attempting to synchronize all source streams in terms of time. By default, Kafka Streams will provide your application with [event-time processing semantics](https://docs.confluent.io/platform/current/streams/developer-guide/config-streams.html#streams-developer-guide-timestamp-extractor). This is important especially when an application is processing multiple streams (i.e., Kafka topics) with a large amount of historical data. For example, a user may want to re-process past data in case the business logic of an application was changed significantly, e.g. to fix a bug in an analytics algorithm. Now it is easy to retrieve a large amount of past data from Kafka; however, without proper flow control, the processing of the data across topic partitions may become out-of-sync and produce incorrect results.

As mentioned in the [Concepts](https://docs.confluent.io/platform/current/streams/concepts.html#streams-concepts-time) section, each data record in Kafka Streams is associated with a timestamp. Based on the timestamps of the records in its stream record buffer, stream tasks determine the next assigned partition to process among all its input streams. However, Kafka Streams does not reorder records within a single stream for processing since reordering would break the delivery semantics of Kafka and make it difficult to recover in the face of failure. This flow control is best-effort because it is not always possible to strictly enforce execution order across streams by record timestamp; in fact, in order to enforce strict execution ordering, one must either wait until the system has received all the records from all streams (which may be quite infeasible in practice) or inject additional information about timestamp boundaries or heuristic estimates such as [MillWheel’s watermarks](https://www.oreilly.com/radar/the-world-beyond-batch-streaming-102/" \t "_blank).

## Backpressure

Kafka Streams does not use a backpressure mechanism because it does not need one. Using a depth-first processing strategy, each record consumed from Kafka will go through the whole processor (sub-)topology for processing and for (possibly) being written back to Kafka before the next record will be processed. As a result, no records are being buffered in-memory between two connected stream processors. Also, Kafka Streams leverages Kafka’s consumer client behind the scenes, which works with a pull-based messaging model that allows downstream processors to control the pace at which incoming data records are being read.

The same applies to the case of a processor topology that contains multiple independent sub-topologies, which will be processed independently from each other (cf. [Parallelism Model](https://docs.confluent.io/platform/current/streams/architecture.html#streams-architecture-parallelism-model)). For example, the following code defines a topology with two independent sub-topologies:

stream1.to("my-topic");

stream2 **=** builder.stream("my-topic");

Any data exchange between sub-topologies will happen through Kafka, i.e. there is no direct data exchange (in the example above, data would be exchanged through the topic “my-topic”). For this reason there is no need for a backpressure mechanism in this scenario, too.

KAFKA STEAMS:

# Introduction

This section provides a quick introduction to the Streams API of Apache Kafka®.

## The Kafka Streams API in a Nutshell

The Streams API of Apache Kafka®, available through a Java library, can be used to **build highly scalable, elastic,** **fault-tolerant, distributed applications and microservices**. First and foremost, the Kafka Streams API allows you **to create real-time applications that power your core business**. It is the easiest yet the most powerful technology to process data stored in Kafka. It builds upon important [concepts](https://docs.confluent.io/platform/current/streams/concepts.html#streams-concepts) for stream processing such as efficient management of application state, fast and efficient aggregations and joins, properly distinguishing between event-time and processing-time, and seamless handling of [out-of-order data](https://docs.confluent.io/platform/current/streams/concepts.html#streams-concepts-out-out-order-handling).

A unique feature of the Kafka Streams API is that the applications you build with it are **normal Java applications**. These applications can be packaged, deployed, and monitored like any other Java application – there is **no need to install separate processing clusters or similar special-purpose and expensive infrastructure**!

[Diagram

Description automatically generated](https://docs.confluent.io/platform/current/_images/streams-introduction-your-app.png)

*An application that uses the Kafka Streams API is a normal Java application. Package, deploy, and monitor it like you would do for any other Java application. Even so, your application will be highly scalable, elastic, and fault-tolerant.*

## Use Case Examples

The Kafka Streams API is applicable to a wide range of use cases and industries.

* Travel companies can build applications with the Kafka Streams API to make real-time decisions to find best suitable pricing for individual customers, to cross-sell additional services, and to process bookings and reservations.
* The finance industry can build applications to aggregate data sources for real-time views of potential exposures and for detecting and minimizing fraudulent transactions.
* Logistics companies can build applications to track their shipments fast, reliably, and in real-time.
* Retailers can build applications to decide in real-time on next best offers, personalized promotions, pricing, and inventory management.
* Automotive and manufacturing companies can build applications to ensure their production lines perform optimally, to gain real-time insights into their supply chains, and to monitor telemetry data from connected cars to decide if an inspection is needed.
* And many more.

## A Closer Look

Before we dive into the [Concepts](https://docs.confluent.io/platform/current/streams/concepts.html#streams-concepts) and [Architecture](https://docs.confluent.io/platform/current/streams/architecture.html#streams-architecture) or get our feet wet by walking through the [Quick Start](https://docs.confluent.io/platform/current/streams/quickstart.html#streams-quickstart), let us take a first, closer look.

A key motivation of the Kafka Streams API is to bring stream processing out of the Big Data niche into the world of mainstream application development, and to radically improve the developer and operations experience by [making stream processing simple and easy](http://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple). Using the Kafka Streams API you can implement standard Java applications to solve your stream processing needs – whether at small or at large scale – and then run these applications on client machines at the perimeter of your Kafka cluster. Your applications are fully elastic: you can run one or more instances of your application, and they will automatically discover each other and collaboratively process the data. Your applications are also fault-tolerant: if one of the instances dies, then the remaining instances will automatically take over its work – without any data loss! Deployment-wise, you are free to choose from any technology that can deploy Java applications, including but not limited to Puppet, Chef, Ansible, Docker, Mesos, YARN, Kubernetes, and so on. This lightweight and integrative approach of the Kafka Streams API – “Build applications, not infrastructure!” – is in stark contrast to other stream processing tools that require you to install and operate separate processing clusters and similar heavy-weight infrastructure that come with their own special set of rules on how to use and interact with them.

The following list highlights [several key capabilities and aspects](https://docs.confluent.io/platform/current/streams/architecture.html#streams-architecture) of the Kafka Streams API that make it a compelling choice for use cases such as microservices, event-driven systems, reactive applications, and continuous queries and transformations.

**Powerful**

* Makes your applications highly scalable, elastic, distributed, fault-tolerant
* Supports exactly-once processing semantics
* Stateful and stateless processing
* Event-time processing with windowing, joins, aggregations
* Supports [Kafka Streams Interactive Queries](https://docs.confluent.io/platform/current/streams/developer-guide/interactive-queries.html#streams-developer-guide-interactive-queries) to unify the worlds of streams and databases
* Choose between a [declarative, functional API](https://docs.confluent.io/platform/current/streams/developer-guide/dsl-api.html#streams-developer-guide-dsl) and a lower-level [imperative API](https://docs.confluent.io/platform/current/streams/developer-guide/processor-api.html#streams-developer-guide-processor-api) for maximum control and flexibility

**Lightweight**

* Low barrier to entry
* Equally viable for small, medium, large, and very large use cases
* Smooth path from local development to large-scale production
* No processing cluster required
* No external dependencies other than Kafka

**Fully integrated**

* 100% compatible with Kafka 0.11.0 and 1.0.0
* Easy to integrate into existing applications and microservices
* No artificial rules for packaging, deploying, and monitoring your applications
* Runs everywhere: on-premises, public clouds, private clouds, containers, etc.
* Integrates with databases through continuous change data capture (CDC) performed by [Kafka Connect](https://docs.confluent.io/platform/current/connect/index.html#kafka-connect)

**Real-time**

* Millisecond processing latency
* Record-at-a-time processing (no micro-batching)
* Seamlessly handles out-of-order data
* High throughput

**Secure**

* Supports [encryption of data-in-transit](https://docs.confluent.io/platform/current/streams/developer-guide/security.html#streams-developer-guide-security)
* Supports [authentication and authorization](https://docs.confluent.io/platform/current/streams/developer-guide/security.html#streams-developer-guide-security)

In summary, the Kafka Streams API is a compelling choice for building mission-critical stream processing applications and microservices. Give it a try and run your first [Hello World application](https://docs.confluent.io/platform/current/streams/quickstart.html#streams-quickstart)! The next sections [Quick Start](https://docs.confluent.io/platform/current/streams/quickstart.html#streams-quickstart), [Concepts](https://docs.confluent.io/platform/current/streams/concepts.html#streams-concepts), [Architecture](https://docs.confluent.io/platform/current/streams/architecture.html#streams-architecture), and the [Developer Guide](https://docs.confluent.io/platform/current/streams/developer-guide/index.html#streams-developer-guide) will get you started.

Tip