Streaming

Reference Specification

Contents

[Streaming 1](#_Toc535920478)

[Reference Specification 1](#_Toc535920479)

[Goal 3](#_Toc535920480)

[Guiding Principles/Demand: 3](#_Toc535920481)

[Audience 4](#_Toc535920482)

[Key Terminology 5](#_Toc535920483)

[Specification Overview 6](#_Toc535920484)

[Technology 8](#_Toc535920485)

[Global Implications (Confluent Kafka v11) 9](#_Toc535920486)

[Design Patterns 13](#_Toc535920487)

[Design Anti-Patterns 19](#_Toc535920488)

[Appendix 20](#_Toc535920489)

[Definitions 22](#_Toc535920490)

[Document History 22](#_Toc535920491)

## Goal

The goal of this Reference Specification is to provide guidance and direction for selecting the appropriate designs for Event Streaming/ Event Backplane. In the context of this document, the concept of *appropriate design* means selecting patterns that align to the [Integration Reference Architecture](https://pages.github.nwie.net/Nationwide/Architecture-Standards/ref-arch/ra-integration.html) and/or the [IoT Reference Architecture](https://pages.github.nwie.net/Nationwide/Architecture-Standards/ref-arch/ra-iot.html).

**Out of Scope:**

Databases – refer to the [Polyglot Persistence Statement of Direction](https://pages.github.nwie.net/Nationwide/Architecture-Standards/sod/sod-polyglot.html)

The Streaming Platform is not a drop-in tool replacement i.e. MQ, IIB, JMS, etc. (Synchronous Messaging)

The Streaming Platform is not an Analytical Model

## Guiding Principles/Demand:

|  |  |
| --- | --- |
| Guiding Principles | Notes |
| Portability | The technology must be portable to have a consistent deployment model regardless of location (Cloud, On-Premises) |
| Loosely Coupled | Less Application integration, more business integration |
| Cloud First | Egress charges and performance implications will be evaluated to determine when we require an on-premises deployment |
| Self-Service | Producers will have the capability to create and manage their own Kafka topics |
| Always appending, immutable | But with-in a bounded context, with a defined retention period |
| Security | It is expected that security mechanisms are in place and follows our standard guidelines with least privileged access |
| Open Standards | We have a biased for Open Standards based software, especially for highly-integrated, core, systems |

|  |  |
| --- | --- |
| Demand | Notes |
| IoT needs to do their own Reference Specs | This reference specification addresses business event backplane. Additional work will need to be completed to address IoT specification |
| Analytics needs to do their own Reference Specs | We loosely touch on this with the lambda pattern, but a deeper dive into analytical patterns may be required |
| Governance Guidance | We will need to determine how governance of the platform and pattern alignment will be accomplished |
| Standard Message Format/ Standard Naming Conventions | For integration simplicity a common data format with common naming conventions is recommended |
| Data Models (e.g., what is a claim?) | We are not defining data models as a part of this effort, but the aggregation of streams will have dependency on them |
| Training | Training will be provided as a part of a larger initiative |
| Platform Owner | It is important that the Platform/Product Owner is well defined and communicated. We also need to determine if a higher-level Capability Stakeholder is also required |

## Audience

|  |  |
| --- | --- |
| User | Primary Use |
| Architects | Adoption |
| Application Developers | Inform & Guide |
| Infrastructure Engineers | Inform & Guide |
| CTOs | Inform & Guide |
| Application Owners | Inform & Guide |
| BSA Run Teams | Inform & Guide |
| Extended Project/Development Team Members | Inform & Guide |

## Key Terminology

**Event:** An occurrence of a change of state. In all cases, events in the context of Data Streaming refer to definable occurrences of "things" which may or may not have immediate use cases for observing. The definition of an event only goes so far as to describe the event which happened and related metadata (time, origin, changed state, etc). The event does not describe what actions should occur as a result of the event. Events can be broken down into the following categories and definitions, useful when describing use cases for Event Streaming.

Note: Real events will almost always conform to some structured message format. Examples given below are for illustration only.

* Business Event
  + A Business Event is a definable occurrence in a business scenario.
* System Event
  + System Events originate from the applications or underlying systems on which IT solutions are built. Events here have no business context attached to them, however could result in further business events being emitted or actions being taken. For example, observing a System Event indicating "shell session started" on a production system could trigger an Information Risk Management Business Event that an investigation into why has begun.

It is important to note that message streaming platforms add to the potential system analytics capabilities provided by APM or Log Aggregations systems in use today, but do not replace them.

* IoT Event (we only care about the events other business events have to respond to)
  + Borrowing from the IoT definition contained in the Nationwide IoT Statement of Direction, IoT Events are those events which stream off of physical devices, vehicles, home appliances and other items embedded with electronics, software and sensors. Events in this realm typically originate from changes in the surrounding physical environment the IoT device resides in. The amount of metadata about the event will vary depending on the anonymity requirements of the solution.

<https://pages.github.nwie.net/Nationwide/Architecture-Standards/sod/sod-iot.html>

* Human/Behavioral Event
  + Human or Behavioral Events are those which originate from human interaction with technology, or human behavior detected by technology. Events in this category provide a more aggregated and meaningful context to other streams of events. For example, multiple "motion detected" or "coordinates changed" events from a vehicle IoT Event Stream could result in a single "arrived home" event. They can also be simple events that occur with direct interaction with technology such as "button clicked" which can be used to analyze marketing campaigns.

**Stream:** A stream is a sequence of data elements made available over time.

**Streaming:** Streaming is data that is generated continuously by thousands of data sources, which can send in the data records simultaneously, and in varying sizes. Streaming data includes a wide variety of data such as business event data, log files, website clicks and impressions, ecommerce purchases, in-game player activity, information from social networks, geospatial services, and telemetry from connected devices or instrumentation in data centers.

**Topic:** A topic is a category or feed name to which records are published. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it.

**Managed**: Managed Topics are meant to be shared with the Enterprise outside the bounded context, with self-governance of content and SLAs.

**Un-Managed:** Streams that are used within a bounded context need not be exposed on the enterprise event backplane, as it may not require publishing to service directory for general consumption.

**Partition:** A partition is an ordered, immutable sequence of records that is continually appended to a structured commit log. The records in the partitions are each assigned a sequential id number called the *offset* that uniquely identifies each record within the partition.

**Producer:** A Producer is any system that publishes data to the stream. Producers publish data to the topics of their choice. The producer is responsible for choosing which record to assign to which partition within the topic. This can be done in a round-robin fashion simply to balance load or it can be done according to some semantic partition function (say based on some key in the record).

**Consumer:** A Consumer is any system that subscribes to topics in the stream for consumption by their system or analytic engine.

**Processor**: A Consumer that may; transform, extract or filter existing streams into a new stream for a business purpose.

**Bounded Context:** A business context is a logical boundary that encompasses applications/systems that satisfies a specific business service function. For example, ECIF is a specific business capability that provides customer information for any enterprise system that needs it, could be a bounded context. This bounded context may comprise of several applications/systems in order to serve such a function, where sub-systems are usually highly integrated. [https://pages.github.nwie.net/Nationwide/Architecture-Standards/ref-arch/ra-integration.html#definitions](https://pages.github.nwie.net/Nationwide/Architecture-Standards/ref-arch/ra-integration.html)

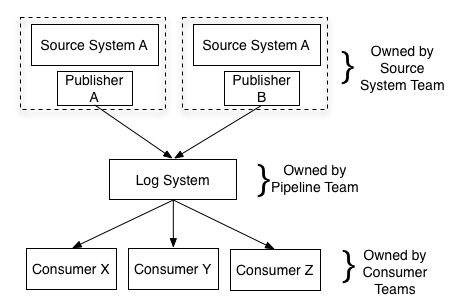
**Batch:** A Unit-of-work where all data elements emitted by the producer are validated/accepted as complete by a bounded check-point message as defined by the use case needs; whether that boundary is date/time driven, user context or arbitrary frame.

## Specification Overview

In general, to provide a specification we need to scope the problem space down to a specific implementation. The [Stream Platform Reference Architecture](https://pages.github.nwie.net/Nationwide/Architecture-Standards/ref-arch/ra-integration.html) provides a conceptual architecture. Work going on in the Microservices space indicates an Enterprise Event Backplane (Backplane), which serves “receive and deliver business events between [software] product.” The Backplane is a hub where Nationwide’s Domain Systems (bounded contexts) communicate events asynchronously in real-time using the Nationwide’s domain language, as opposed to the synchronous communication handled via APIs.

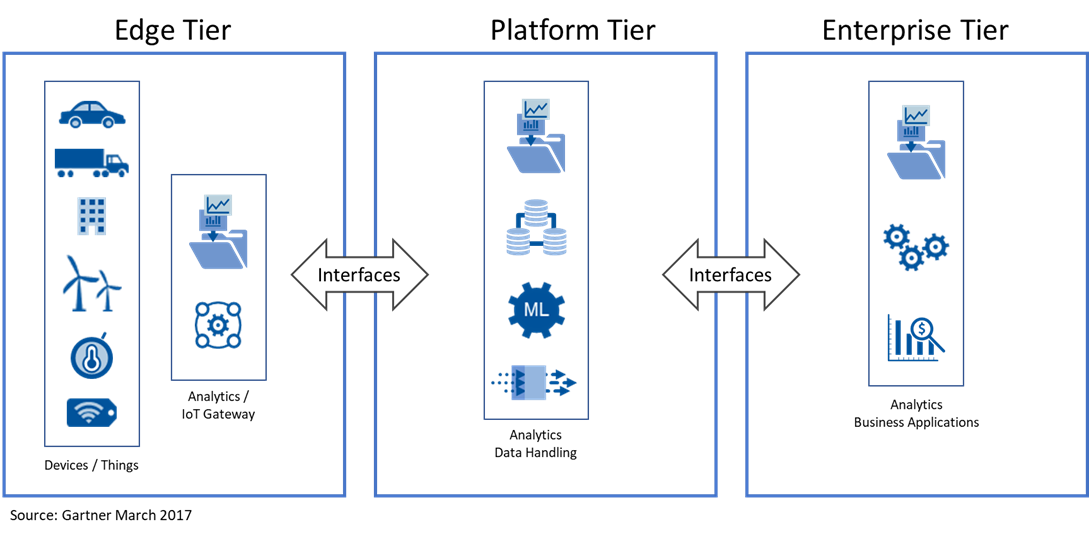
The Backplane aligns with Kappa Architecture and provides a good model for streaming at Nationwide. “Kappa Architecture is a software architecture pattern. Rather than using a relational DB…or a key-value store…the canonical data store in a Kappa Architecture system is an append-only immutable log. From the log, data is streamed through a computational system and fed into auxiliary stores for serving.” [See: [Kappa Architecture](http://milinda.pathirage.org/kappa-architecture.com/)]

The “log” in Kappa Architecture “...is an append-only, totally-ordered sequence of records ordered by time.” [See: [“The Log”](https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying)]

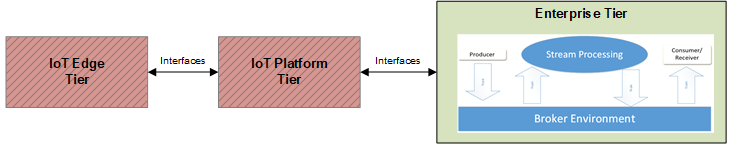


If we consider the Backplane as the place where events that provide meaningful state changes to the enterprise and need to be detected from outside the bounded context must be published, then the Backplane is the Log System from Kappa Architecture.

However, the Backplane is not an all-encompassing solution for all things streaming. For instance, the [IoT Reference Architecture](https://pages.github.nwie.net/Nationwide/Architecture-Standards/ref-arch/ra-iot.html) makes heavy use of streams as well. While the Backplane would participate in Nationwide’s I0T solutions, it would not expect to handle all the various streams of data needed to make a fully-functional IoT platform. The IoT Reference Architecture references three tiers in its [Logical Architecture](https://pages.github.nwie.net/Nationwide/Architecture-Standards/ref-arch/ra-iot.html): Edge, Platform, and Enterprise.



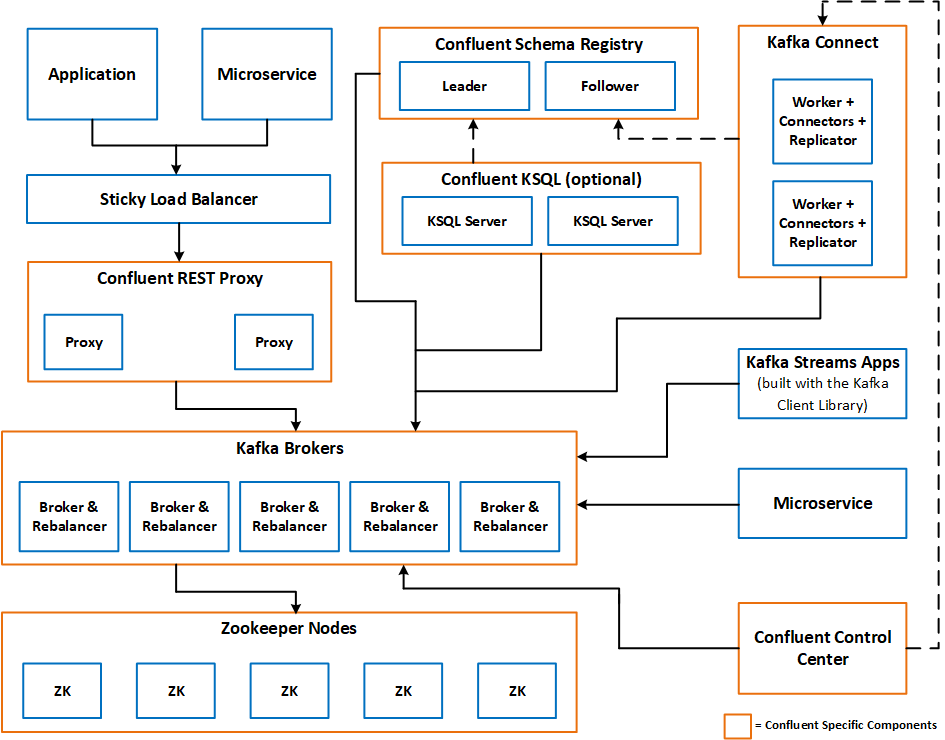
In the IoT context, the Backplane exists in and only addresses IoT streaming needs within the Enterprise Tier.



IoT Streaming will likely vary between Edge and Platform Tier as well as between different IoT solutions and will need to be addressed in an IoT Reference Specification or in specific IoT Solutions. However, all solutions will leverage the Backplane to publish events to consumers outside the bounded context.

## Technology

Confluent Technology: We have decided on Confluent Kafka as the underlying technology for the Event Backplane. Below is the logical architecture. We will provide additional detail in the Streaming Platform Reference Implementation.



**Note:** See Appendix for a description of each component

**Confluent Recommendations:**

1. Limit the number of clusters (Fewest number of clusters may not be one cluster):
   1. The recommendation is to connect to a cluster “locally” and mirror the data across disparate locations.
   2. In some cases, based on security requirements, we may need to physically segregate data.
   3. Kafka supports isolating applications within a single cluster, but there will be times we want to separate the cluster to reduce the “blast radius”.
2. Pick a single data format:
   1. Kafka does not enforce a single data format for events beyond a simple key/serialized value model.
   2. The overall simplicity of integration is not only having stream data in a single system, but also making all data look similar and follow similar conventions (JSON as example).
3. Model events not commands.
   1. An event states something has occurred whereas a command tells a particular system to do some particular work.
4. Use Kafka Connect for connecting existing systems and applications.
   1. The connectors have a built-in scale-out model so you can easily connect very large-scale systems like Hadoop or Cassandra without the integration itself becoming a bottleneck. You can do this dynamically without stopping your running connectors.
   2. The connectors are fault-tolerant: if one instance of a connector fails data won’t stop flowing, the other instances will detect this and pick up the work.
   3. Connect allows you to manage many such connections simply with an easy-to-use REST api. For example, if you have connectors running for many database instances you can do this without having to manually run processes for each of these.
   4. Connect helps you to capture whatever metadata is present about data format. If your data is unstructured strings or bytes that is fine, but if you have richer structure such as you might find with a relational database this will be preserved by the connect framework.
5. Consider a Stream Processer (E.g. Flink, Storm, Spark)
   1. In the 0.10 release of Kafka we added the streams api which brings native stream processing capabilities to Kafka. This is a bit different from the existing frameworks. Rather than being a MapReduce-like framework for distributing and executing stream processing jobs, it is instead a simple library that brings state-of-the-art stream processing capabilities to normal Java applications. Applications that use this library can do simple transformations on data streams that are automatically made fault-tolerant and are transparently and elastically distributed over the instances of the application.

## Global Implications (Confluent Kafka v11)

**Behaviors and Implications of Streams vs Queues**

The following are scenarios which are common in asynchronous messaging systems, but where the behavior in implementation can differ between Streams and Queues. These are called out for those familiar with Queuing based platforms like IBM MQ and are looking to migrate to an alternate messaging platform. The Apache Kafka implementation of Streams is used as an example where Streams are mentioned, however the reader should consult the documentation of the streaming product in mind when designing their solution. Amazon Kinesis for example shares high level concepts with Apache Kafka, however the finer details of design and implementation will vary. While you can and should use Streaming for asynchronous messaging, there is coding/refactoring that will need to be done for streaming vs queuing.

**Message Retry and Poison Message Handling Semantics**

Errors of some kind can occur during the production or consumption of any message. The message itself may be malformed, or the invocation of a dependent downstream system involved in the processing of a message may fail. In asynchronous messaging systems there is the benefit of being able to retry that message at some point in the future (waiting on a downstream to come back online for example), where in synchronous systems you would likely respond with a failure immediately.

**Queues**

Queuing systems typically have semantics for tracking Retry Counts and Backout Thresholds. A common technique is to define a Backout Threshold for a Queue, and then track the Retry or Backout count when processing the message. If processing fails, increment the count and put it back on the Request Queue for processing again. Some systems will allow for a delay to be placed on that message before it can be processed again under the assumption that whatever failed a second ago is likely to fail again now.

When the Backout Count matches the Backout Threshold, the Consumer of the message will place it on a Backout Queue to be diagnosed at a later time. Some client API's like JMS can perform this behavior by convention without any additional coding needed.

**Streams**

Depending on the client API in used, there may be more handholding required for what to do with streamed messages which cannot be processed. One fact always remains true which is a stream is an unbounded, continuously updating, ordered, replayable sequence of immutable data records. The "immutable" part of that statement means that you cannot remove a bad record from the stream. It will be there in the future during replays, and will have to be handled the same.

What is possible, through additional logic or built in help from a client library, is to write that bad record yourself to another stream. In this way, you can treat a separate stream like you might a Backout Queue in that it can be inspected, and potentially replayed, at a later time.[[1]](#footnote-2)

The key callout here is that the concept of a "Retry" within the same stream has no correlation to how it works in Queues. To go backwards in a stream is to reprocess that event and every event that came after it. This is potentially possible if following Event Sourcing patterns, but the results of replaying messages could be disastrous if that level of maturity is not there.

Further discussion and examples of retry logic in Kafka are provided in the following article:

<https://blog.pragmatists.com/retrying-consumer-architecture-in-the-apache-kafka-939ac4cb851a>

**Horizontal Scaling of Consumers**

A common technique for increasing the throughput of messages consumption is to horizontally scale your consumer application via threads or new processes. This scales so long as your messaging system can keep up with message delivery. How it is achieved differs between Queues and Streams and will impact your design.

A deeper description of the difference between Streams and Queues is described in [Kafka as a Messaging System] (https://kafka.apache.org/documentation/#kafka\_mq).

**Queues**

The technique of horizontally scaling your consumers works because of the delivery semantics of Queue based messaging. Only if there are strict ordering guarantees will a Queue limit it's consumer channel to a single connection. When ordering is not required, multiple channels and consumers are allowed, and the number of current messages being processed can be as high as the number of consumer application instances or threads.

**Streams**

The delivery semantics of messages to a Stream impact how many consumers can process that stream. In contrast to Queues where a consumer pops a message off the top, a Stream consumer locates the next message by an offset. The offset only increases no matter the age, depth, or purge strategy of the Stream. Ignoring Stream Partitions for the moment, if there were multiple consumers of a single Stream, they would need to come to consensus among one another what the next offset to retrieve is before reading the next message. Without doing so, they would all perform duplicate processing on the same Stream. The benefits of parallelism is offset by the expense of coming to that consensus, and therefore Streaming platforms leverage Partitioning (Kafka) or Shards (Kinesis) to enable parallel processing of a Stream.

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 in the FAQ. For an example of this, see [link](https://docs.confluent.io/current/streams/architecture.html)**.**

The following two articles describe this concept further.

[https://docs.confluent.io/current/streams/architecture.html#parallelism-model](https://docs.confluent.io/current/streams/architecture.html)

[https://kafka.apache.org/documentation/#intro\_consumers](https://kafka.apache.org/documentation/)

**Ordering Guarantees**

Ordering Guarantees become important if there are strict requirements for processing messages in the exact order they arrived. The difference between Streams and Queues are not that different when it comes to strategies for how to ensure strict ordering guarantees, but additional scaling opportunities exist for Streams under the right conditions which are worth calling out.

**Queues**

Put simply, to ensure strict processor ordering in Queue based messaging systems, you can only have one consumer of that Queue. Any attempt to scale horizontally through processes or threads will result in the unordered processing of messages.

**Streams**

The natural delivery semantics of Streams imply that messages will be processed in the order they are received. Consumers maintain an offset value which is used to find the next sequential message in the stream. This is always true within a single Stream Partition; however a Stream can be broken into multiple Partitions. The implication is that within a Partition, a consumer can depend on the message order, whereas a consumer cannot depend on the message order across partitions within a stream.

When strict ordering is required in a Stream, it is also good to consider exactly where order matters. For example, is it important that every Insurance Policy event be processed in the order they were received, or is it only important that events of an Insurance Policy in the State of Ohio be processed in order? Understanding where ordering is important can lead to good Stream Partition Index design, and allow for more concurrent processing by consumer applications.

The following article describes extra considerations with respect to At-least-once and Exactly-once message delivery semantics, and Out-of-Order Handling of messages.

[https://docs.confluent.io/current/streams/concepts.html#processing-guarantees](https://docs.confluent.io/current/streams/concepts.html)

**Correlation Identifiers**

The [Correlation Identifier Pattern](https://www.enterpriseintegrationpatterns.com/patterns/messaging/CorrelationIdentifier.html) is commonly used when Synchronous communication behavior is required over asynchronous messaging systems like Queues. As stated elsewhere in this document, applications should not use Streams for synchronous communication. The current preference is to use APIs for synchronous communication. See the [Integration Reference Architecture](https://pages.github.nwie.net/Nationwide/Architecture-Standards/ref-arch/ra-integration.html) for more information regarding APIs. The reason is simply that Streams are not optimized for this pattern, as they require a messaging system with capabilities to efficiently query a specific message out of a larger list.

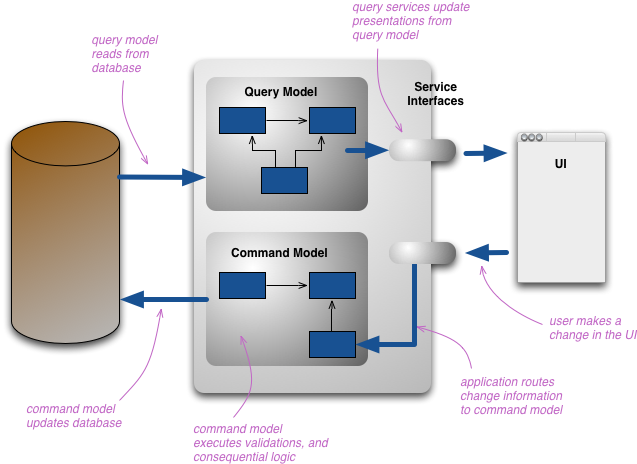
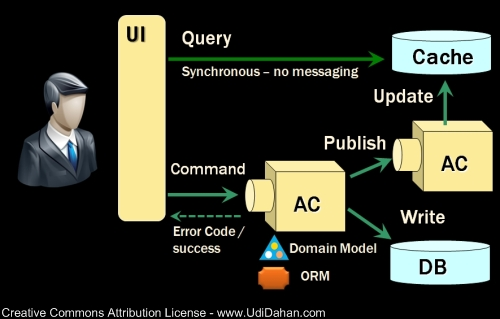
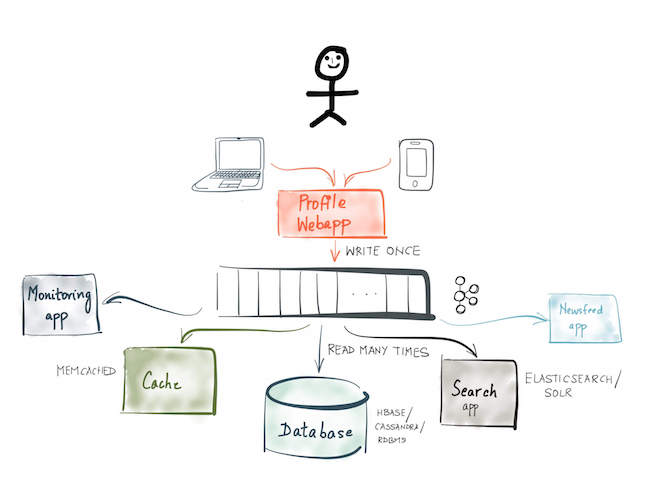
Queuing systems typically have some mechanism by where you can specify some attribute to search on when opening a session. In JMS, one might search on the JMSCorrelationID message when opening a new session, and the JMS Session will block on a GET operation until that message can be delivered to the consumer.

QueueReceiver receiver = session.createReceiver(myQueue, "JMSCorrelationID='12345'");

Streaming platforms (Kafka included) typically do not provide this indexing capability, and instead a routine would need to scan the entire Stream looking for the message in question. This is similar to the "full table scan" issue one might encounter when querying a database table where an index was not used.

This is not to say that Correlation ID's of some sort cannot exist in Stream messages. The pattern is only to avoid scenarios where your logic needs to find a specific message in a Stream, as this is not what Streams are optimized for. If a message is written to Stream B as a result of processing a message on Stream A, there can be cases where correlating the two is beneficial. Even in this scenario however, the goal should be to shift to event-based architectures in which the event provides all the necessary context of what just occurred. The presence of Correlation ID's in messages can be a symptom of tight coupling between systems which should be avoided in streaming architecture.

## Design Patterns

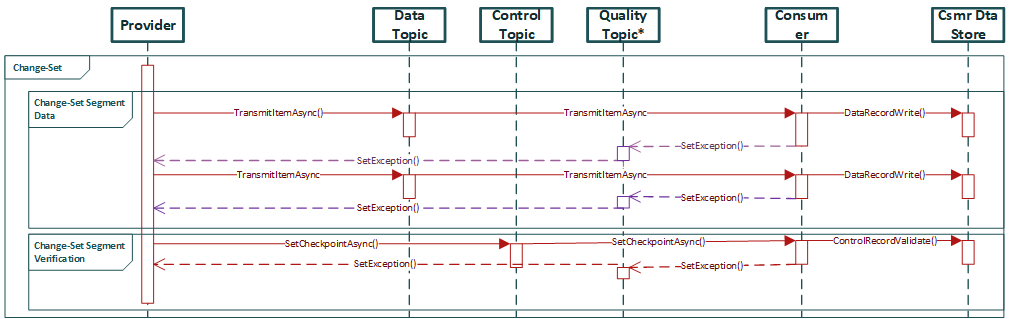
1. Command Query Responsibility Segregation (CQRS)
   1. **Description:** Command Query Responsibility Segregation (CQRS) uses different models to separate read operations (Query) from write operations (Command) based on the idea that having a single model for both leads to a complex model that handles neither reads nor writes particularly well. By separating these responsibilities, read operations can be tuned and scaled independently of write operations. However, since these rely on separate models, and the read model is derived from the write model, the models are eventually consistent rather than strictly consistent.  
      
   2. **Streaming Impact:** Outside the context of Event Streaming, an application or microservice might implement the read and write models using a single data store. However, once an application or microservice start publishing its internal events outside its bounded context, the write operations (Command) must support writing to an external event store. The read model (Query) will need to be synchronized with the write model (Command), likely creating an eventual consistency situation.  
      
   3. **Use Case Example**: Need help here. I think almost any application / microservice implementing event sourcing will need to implement this pattern. Potentially use Data Warehouse or Whoville as examples.
   4. **Links**:
      1. [Event sourcing, CQRS, stream processing and Apache Kafka](https://www.confluent.io/blog/event-sourcing-cqrs-stream-processing-apache-kafka-whats-connection/) by Neha Narkhede
      2. [CQRS](https://martinfowler.com/bliki/CQRS.html) by Martin Fowler
      3. [Clarified CQRS](http://udidahan.com/2009/12/09/clarified-cqrs/) by Udi Dahan
      4. [1 Year of Event Sourcing and CQRS](https://hackernoon.com/1-year-of-event-sourcing-and-cqrs-fb9033ccd1c6) by Teiva Harsanyi
      5. [Command and Query Responsibility Segregation](https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs) at Azure Cloud Design Patterns
      6. [CQRS](https://microservices.io/patterns/data/cqrs.html) at Microservice Architecture
2. Compensating Transactions
   1. **Description:** Compensation Transactions is a method of undoing work performed by a series of steps, possibly by independent autonomous components, which define an eventually consistent operation.
   2. **Streaming Impact:** As Nationwide embraces a cloud-first mentality, applications and microservices will adopt cloud behaviors, which include eventual consistency. Event Streaming encourages an eventual-consistency model and will require applications / microservices to coordinate undo actions through compensation transactions.
   3. **Use Case Examples**: Correcting Transactions in distributed accounting systems.
   4. **Links**:
      1. [Compensating Transaction](https://docs.microsoft.com/en-us/azure/architecture/patterns/compensating-transaction) at Azure Cloud Design Patterns
      2. [Saga](https://microservices.io/patterns/data/saga.html) at Microservice Architecture
3. Event Sourcing – being able to restore current state from playback  
   
   1. **Description**: By expressing the user intent as an ordered log of immutable events, event sourcing gives the business an audit and compliance log. It enables resilient applications; rolling back applications amounts to rewinding the event log and reprocessing data. It has better performance characteristics; writes and reads can be scaled independently. It enables a loosely coupled application architecture; one that makes it easier to move towards a microservices-based architecture. But most importantly, event sourcing enables building a forward-compatible application architecture — the ability to add more applications in the future that need to process the same event but create a different materialized view.
   2. **Streaming Impact:** Applications should move away from providing multiple views of data tailored to individual downstream applications and, instead, provide a single stream of events that downstream applications can use to create a materialized view specific to their purpose.
   3. **Use Case Examples**:
   4. **Links:**
      1. [Event sourcing, CQRS, stream processing and Apache Kafka: What’s the connection?](https://www.confluent.io/blog/event-sourcing-cqrs-stream-processing-apache-kafka-whats-connection/)
      2. [Stream processing, Event sourcing, Reactive, CEP… and making sense of it all](https://www.confluent.io/blog/making-sense-of-stream-processing/)
4. Pub/Sub Pattern
   1. **Description:** Publish/subscribe messaging is a form of asynchronous service-to-service communication used in serverless and microservices architectures. Providers process events and publish the event to the stream for a specific topic. It is immediately available for all the subscribers to the topic to pull the message and process based on their application needs.
   2. **Streaming Impact:** Current Implementation utilizes IBM message queues and IIB in a centralized model. It is a push of the message from a single provider queue to multiple subscriber queues. This changes to self-service development replacing queues with topic on stream. The push becomes a pull by the subscribers from a single topic. All data manipulation including transformation will occur outside of stream and is the responsibility of the Provider and Consumers. MQ or custom shared code to read and write to queues is replaced with adapters, Kafka Connect, Confluent Rest Proxy, API’s or “bridge” solutions as defined by the application. This solution is not a drop and replacement of IIB or MQ for pub-subs. Additional development will be needed by the publisher and subscriber to migrate to streams.
   3. **Use Case Example:**  eCIF publishing changes (events) in Customer Preferences to be consumed by Billing, Claims, Agency applications, etc...
   4. **Links:**

<https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html>

<https://onyourside.sharepoint.com/sites/SOAPlaybook/SitePages/Pub%20Sub%20Messaging%20Integration%20Pattern.aspx>

1. Batch
   1. **Description:** Provider to Consumer interaction where the Consumer can process events with in a bounded frame; where that frame may be business transaction, time-frame or any other unit-of-work set boundary and must be consumed with-in that boundary in order to complete a logical transaction-set.
   2. **Streaming Impact:** Current Implementation is implemented in file-based transfers with data quality header and footers. This changes to a self-service development pattern; replacing file transfers with file detail records streamed as individual records to data topics and the data quality records shared to control topics with both holding a globally unique unit of work id shared by unit of work set. It will change from multiple ETL service(s) to multiple topics to a single data and control topic per business data implementation. File input code needs to produce a globally unique ID per file; read the details and emit the records to the correct topics. System integration work-sets may emit the data records as generated with a scheduled or business triggered checkpoint that emits the control record completing the unit of work. This pattern is required for the producer, but each consumer’s use case may or may not implement the checkpoint emission validation. See also Pub/Sub, Lambda patterns.
   3. **Use Case Example:**  External provider(s) change-sets, Retirement Plans inbound payroll intents, ‘End of Day’ or ‘Intra-Day-Checkpoint' cycle processing.
   4. **Links:**

Replaces: [https://www.enterpriseintegrationpatterns.com/patterns/messaging/FileTransferIntegration.](https://www.enterpriseintegrationpatterns.com/patterns/messaging/FileTransferIntegration.html)html  
Implements: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageChannel.html>Implements: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/DatatypeChannel.html>Implements: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/ControlBus.html>



<https://onyourside.sharepoint.com/sites/SOAPlaybook/SitePages/Pub%20Sub%20Messaging%20Integration%20Pattern.aspx>

1. Analytics
   1. **Description:** As more and more data are streamed from a variety of sources such as devices, sensors, web sites, social media, and other applications, transforming this data into actionable insights and predictions in real-time or near real-time becomes an operational necessity.

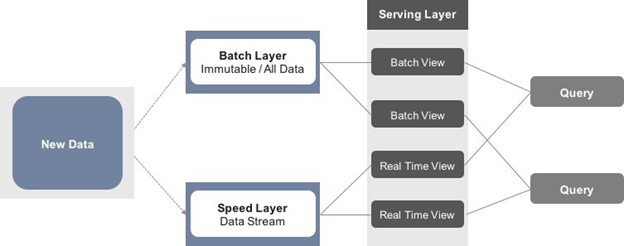
Streaming Analytics is the ability to constantly calculate statistical analytics while moving with the stream of data. Streaming Analytics allows management, monitoring, and real-time analytics of live streaming data. Streaming Analytics involves knowing and acting upon events happening in your business at any given moment. Since Streaming Analytics occurs immediately, we must act on the analytics data quickly within a small window of opportunity before the data loses its value.

* 1. **Streaming Impact:** The key need for analytic data processing, in a streaming data context, where large volumes of data is streamed continually and unbounded, is to process this big data in real-time or near real-time. This imposes the need for qualities such as scalability, fault-tolerant, predictability, resiliency against stream imperfections, and a platform that is extensible.

The Lambda data processing architecture attributed to Nathan Marz, is one of the more common architectures designed to address robustness, scalability and fault-tolerance (human and/or machine) in big data processing.

Lambda Architecture tends to achieve these goals by balancing latency (timeliness of results from data processing) with accuracy of the results

In this pattern, the data stream entering the system is dual fed into both a **batch** and **speed** layer



The **Batch Layer** stores the raw data as it arrives, untouched, immutable, in an append only fashion. The result of the batch processing yields in creating batch views for consumption in the service layer. Batch processes will occur on some interval and the results will be long-lived. The retention scope of data is anywhere from hours to years.

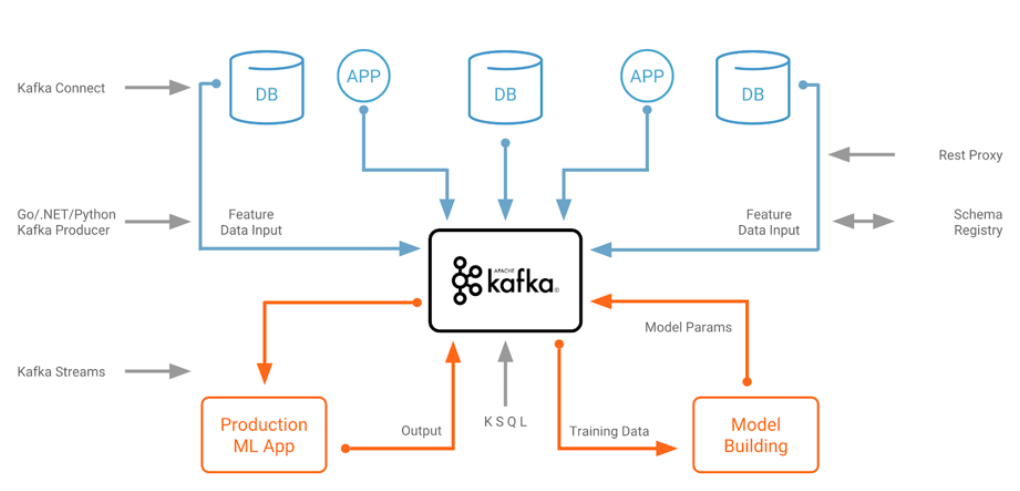
The **Speed Layer** is used to compute the real-time views to compliment the batch views, it is temporal in nature, meaning we can only keep limited information in the streams known history, which is typically held in-memory, before we make way for new data

**Service Layer** holds the views to both batch and real-time processed views.   
Any query may get a complete picture by retrieving data from both the batch views and the real-time views. The queries will get the best of both worlds. The batch views may be processed with more complex or expensive rules and may have better data quality and less skew, while the real-time views give up to the moment access to the latest possible data. As time goes on, real-time data expires and are replaced with data in the batch views.

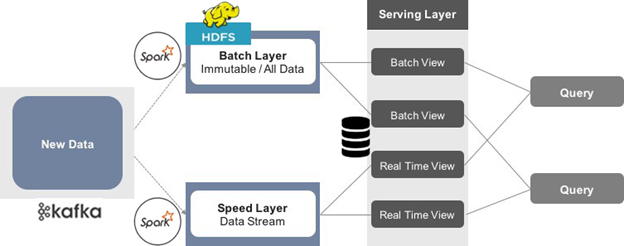
An additional benefit to this architecture is that we can replay the same incoming data and produce new views in case code or formula changes

* 1. **Use Case Examples:** Many real-time use cases fit well with Lambda Architecture. In some cases, however, having access to a complete set of data in a batch window may yield certain optimizations that would make Lambda better performing and perhaps even simpler to implement.

Eg: Build/Train Machine Learning Models, ML Model Execution (Real-time), Dashboards, Metrics



Technology Considerations:

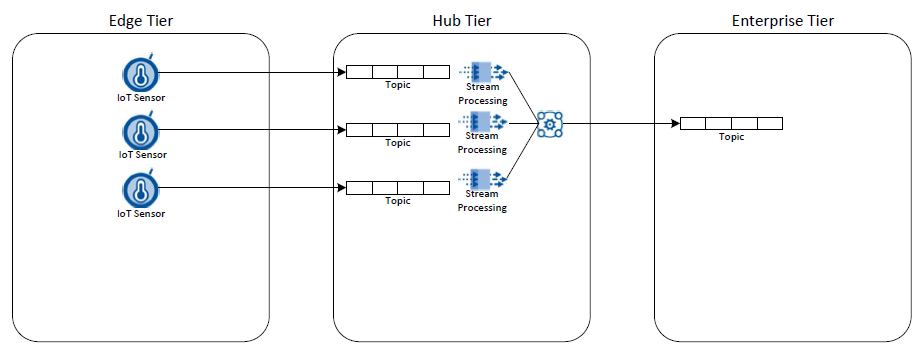


* 1. **Links:**

Lambda Architecture: <http://lambda-architecture.net/>

Plural Site (Lambda Implementation): <https://app.pluralsight.com/library/courses/spark-kafka-cassandra-applying-lambda-architecture/table-of-contents>

1. Event Aggregation/Reduction
   1. **Description:** Modification of an inbound stream(s) creating a new stream. Reduction can apply filtering of streams message content or filtering of number of messages by a selection criterion, creating a new stream. Aggregation can add content to stream or concatenate streams, creating a new stream.
   2. **Streaming Impact:** Aggregation prior to streaming was typically through composite services or Experience APIs. Modifications to a stream requires creation of a new stream and is done by BSA either through stateful or stateless processing in Kafka, Lambda function, BSA application or other application specific implementation.
   3. **Use Case Example:** 
      1. Aggregation:
         1. Aggregating multiple policy admin system “cancel policy by *system*” topics to single enterprise “cancel policy” topic.
         2. Adding customer name based on enterprise customer number to a stream from a “table” lookup.
      2. Reduction:
         1. A stream for registration of new customers. An event processor could and then filter the topic creating a new stream for Life Insurance Customers only.
         2. IoT Events from home thermostat / nest published to a stream. An event processor filters the stream and creates a new stream of Increase in temperature over 80 degrees.



* 1. **Links:**

<https://onyourside.sharepoint.com/:b:/r/sites/IT-Architecture/ALT/Shared%20Documents/02%20-%20Capability%20Uplift/ALT%20Initiatives/2017%20ALT%20Decision%20Velocity/Ref%20Spec%20-%20Event%20Streaming/Making_Sense_of_Stream_Processing_Confluent_1.pdf?csf=1&e=hXu7E5>

<http://kafka.apache.org/documentation/streams/>

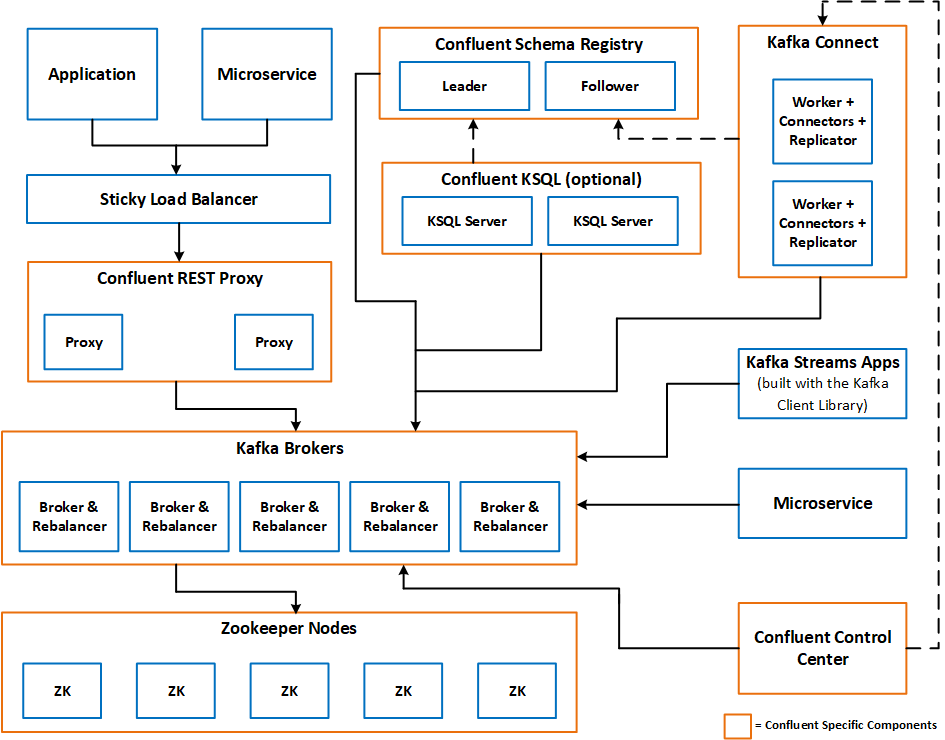
[https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Stream+Usage+Patterns#KafkaStreamUsagePatterns-Howtoaggregatedatafromallcurrentlyactivesessions](https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Stream+Usage+Patterns)?

<https://docs.confluent.io/current/streams/concepts.html>

## Design Anti-Patterns

1. Many Providers/Publishers to single stream topic
   1. **Description:** This is situation where we have multiple providers publishing the same event to a single topic/stream.
   2. **Alternatives**: The approved pattern is to have individual topic/stream for each provider and if needed have an aggregation of the individual streams into a single topic/stream at enterprise business event.
   3. **Use Case Example:**  Service Advantage and PolicyCenter both publish “Cancel Policy” events.
   4. **Justification**:
      1. Variable message formats
      2. Decoupling of providers
      3. Providers able to size and implementation that is best fit for their application
      4. Audit and Logging at granular level
   5. **Link**:
2. Request-Reply “Command” pattern
   1. **Description:** Providers utilizes topic on stream to create a request with the expectation of a response on another topic to complete the operation.
   2. **Alternatives**: The approved pattern is to utilize APIs or queues.
   3. **Use Case Example:** PolicyCenter order underwriting report but requires report from B2B partner to complete processing.
   4. **Justification**:
      1. Modern day synchronous patterns are best achieved through APIs. Streaming platform capabilities are not well-suited for this pattern.
      2. APIs should be utilized for synchronous
   5. **Links**: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/RequestReply.html>

## Appendix



### Components

ZooKeeper

ZooKeeper is a centralized service for managing distributed processes and is a mandatory component in every Apache Kafka cluster.

Kafka Brokers

Kafka brokers are the main storage and messaging components of Apache Kafka. Kafka is a streaming platform that uses messaging semantics. The Kafka cluster maintains streams of messages called topics; the topics are sharded into partitions (ordered, immutable logs of messages) and the partitions are replicated and distributed for high availability. The servers that run the Kafka cluster are called brokers.

Kafka Connect Workers

Kafka Connect is a component of Apache Kafka that allows it to integrate with external systems to pull data from source systems and push data to sink systems. It works as a pluggable interface, so you plug in Connectors for the systems you want to integrate with. For example, you deploy Kafka Connect with JDBC and Elasticsearch Connectors to copy data from MySQL to Kafka and from Kafka to Elasticsearch. The full list of available Connectors can be found here: <http://www.confluent.io/product/connectors>

Kafka Clients

Apache Kafka clients are used in the applications that produce and consume events. The Apache Kafka’s Java client JARs are included in the Confluent Platform Kafka packages and are installed alongside Kafka brokers, but they are typically deployed with the application that imports them by adding the client libraries as application dependencies using a build manager such as Apache Maven.

Kafka Streams API

Kafka Streams, a component of open source Apache Kafka, is a powerful, easy-to- use library for building highly scalable, fault tolerant, stateful distributed stream processing applications on top of Apache Kafka. It builds upon important concepts for stream processing such as properly distinguishing between event-time and processing-time, handling of late-arriving data, and efficient management of application state.

Confluent KSQL Server

Confluent KSQL is an open source streaming SQL engine that implements continuous queries against Apache Kafka. It allows you to query, read, write, and process data in Apache Kafka in real-time, at scale using SQL-like semantics. The KSQL Command Line Interface (CLI) allows you to interactively write KSQL queries, this CLI acts as a client and can run on any machine (server or laptop) with access to the KSQL server. The KSQL server runs the engine that executes KSQL queries, which includes the data processing as well as reading data from and writing data to the target Kafka cluster. (Describe/Provide Valid Use-cases for KSQL Server deployment)

Confluent REST Proxy

The Confluent REST Proxy is an open source HTTP server that provides a RESTful interface to a Kafka cluster. It makes it easy to produce and consume messages, view the state of the cluster, and perform administrative actions without using the native Kafka protocol or clients. The REST Proxy is not a mandatory component of the platform — you will choose to use the REST Proxy if you wish to produce and consume messages to/from Kafka using a RESTful HTTP protocol. If your applications only use the native clients (mentioned above), you can choose not to deploy the REST Proxy.

Confluent Schema Registry

Confluent Schema Registry is an open source serving layer for your metadata. It provides a RESTful interface for storing and retrieving Avro schemas. It stores a versioned history of all schemas, provides multiple compatibility settings, and allows evolution of schemas according to the configured compatibility setting. The Confluent Schema Registry packages also include serializers that plug into Kafka clients and automatically handle schema storage and retrieval for Kafka messages that are sent in the Avro format.

Confluent Replicator

Confluent Replicator is a new component added to Confluent Enterprise to help manage multi-cluster deployments of Confluent Platform and Apache Kafka. It provides a centralized configuration of cross-cluster replication. Unlike Apache Kafka’s MirrorMaker, it replicates topic configuration in addition to the messages in the topics.

Confluent Auto Data Balancing

Confluent Auto Data Balancing is a new component added to Confluent Enterprise to optimize resource utilization and help scale Kafka clusters. Auto Data Balancing evaluates information on the number of brokers, partitions, leaders and sizes of partitions to decide on a balanced placement of partitions on brokers and modify the replicas assigned to each broker to achieve a balanced placement. For example, when a new broker is added to the cluster, Auto Data Balancing will move partitions to the new broker to balance the load between all brokers available in the cluster. To avoid impact on production workloads, the rebalancing traffic can be throttled to a fraction of the available network capacity.

Confluent Control Center

Confluent Control Center is Confluent’s web-based tool for managing and monitoring Apache Kafka. It is part of Confluent Platform Enterprise and provides two key types of functionality for building and monitoring production data pipelines and streaming applications:

* **Data Stream Monitoring and Alerting:** You can use Control Center to monitor your data streams end to end, from producer to consumer. Use Control Center to verify that every message sent is received (and received only once), and to measure system performance end to end. Drill down to better understand cluster usage and identify any problems. Configure alerts to notify you when end-to-end performance does not match SLAs or measure whether messages sent were received.
* **Multi-cluster monitoring and management:** A single Control Center node can monitor data flows in multiple clusters and manage data replication between the clusters.
* **Kafka Connect configuration:** You can also use Control Center to manage and monitor Kafka Connect: the open source toolkit for connecting external systems to Kafka. You can easily add new sources to load data from external data systems and new sinks to write data into external data systems. Additionally, you can manage, monitor, and configure connectors with Confluent Control Center.

## Definitions

* Complex Event Processing & Complex Event Processing Engines
  + Event processing is a method of tracking and analyzing (processing) streams of information (data) about things that happen (events) and derive a conclusion from them. Complex event processing, or CEP, is event processing that combines data from multiple sources to infer events or patterns that suggest more complicated circumstances. The goal of complex event processing is to identify meaningful events (such as opportunities or threats) and respond to them as quickly as possible.
  + <https://www.quora.com/How-is-stream-processing-and-complex-event-processing-CEP-different>
* Event Stream Processing & Stream Processing Engines
  + Event stream processing, or ESP, is a set of technologies designed to assist the construction of event-driven information systems. ESP technologies include event visualization, event databases, event-driven middleware, and event processing languages, or complex event processing.
  + <https://www.quora.com/How-is-stream-processing-and-complex-event-processing-CEP-different>
  + <https://www.bmc.com/blogs/event-stream-processing/>
* Broker
* Record
  + Producer sends messages to Kafka in the form of records. A record is a key-value pair. It contains the topic name and partition number to be sent. Kafka broker keeps records inside topic partitions. Records sequence is maintained at the partition level. You can define the logic on which basis partition will be determined.
* Topic
  + Producer writes a record on a topic and the consumer listens to it. A topic can have many partitions but must have at least one.
* Partition
  + A topic partition is a unit of parallelism in Kafka, i.e. two consumers cannot consume messages from the same partition at the same time. A consumer can consume from multiple partitions at the same time.
* Offset
  + A record in a partition has an offset associated with it. Think of it like this: partition is like an array; offsets are like indexes.
* Producer
  + Creates a record and publishes it to the broker.
* Consumer
  + Consumes records from the broker.
* Commands
  + In Kafka, a setup directory inside the bin folder is a script (kafka-topics.sh), using which, we can create and delete topics and check the list of topics. Go to the Kafka home directory.

## Links

* Related SOD and Reference Architectures
  + [Cloud Native Platform Reference Architectecture](https://pages.github.nwie.net/Nationwide/Architecture-Standards/ref-arch/ra-cnp.html)
  + [Integration Reference Architecture](https://pages.github.nwie.net/Nationwide/Architecture-Standards/ref-arch/ra-integration.html)
  + [Integration Statement of Direction](https://pages.github.nwie.net/Nationwide/Architecture-Standards/sod/sod-integration.html)
  + [Cloud Infrastructure Reference Architecture](https://pages.github.nwie.net/Nationwide/Architecture-Standards/ref-arch/ra-ci.html)
  + Add Data Classification and Data Reference
* Enterprise Message Integration
  + [ETS Message Integration](https://onyourside.sharepoint.com/sites/SOAPlaybook/SitePages/Integration%20Architecture.aspx)
  + [Enterprise Integration Patterns](https://www.enterpriseintegrationpatterns.com/index.html)
* Streaming Platforms and Tools
  + [Kafka](https://kafka.apache.org/)
  + [Confluent Docs](https://docs.confluent.io/current/?_ga=2.77658132.1811372741.1548254355-1588546750.1546463151&_gac=1.90543208.1547235405.Cj0KCQiAmuHhBRD0ARIsAFWyPwgv1LtCBDkIZ0IAOb-fu9-CzOWaHljXswmNfolCiSfzoYVGMCHLpqwaAiISEALw_wcB)
  + [Amazon Managed Streaming for Kafka](https://aws.amazon.com/msk/) (preview)
  + [Amazon Kinesis Data Firehose](https://aws.amazon.com/kinesis/data-firehose/)
  + [Apache Storm](https://storm.apache.org/)
* Streaming Architecture
  + [O’Reilly Streaming Architecture](https://mapr.com/ebooks/streaming-architecture/chapter-01-why-event-streaming.html)
  + [Lambda](http://lambda-architecture.net/)
  + [Kappa](http://milinda.pathirage.org/kappa-architecture.com/)
* Streaming General Info
  + [InfoQ-eMag-Streaming](https://onyourside.sharepoint.com/:b:/r/sites/IT-Architecture/ALT/Shared%20Documents/02%20-%20Capability%20Uplift/ALT%20Initiatives/2017%20ALT%20Decision%20Velocity/Ref%20Spec%20-%20Event%20Streaming/InfoQ-eMag-Streaming-Architecture-1515445122463.pdf?csf=1&e=4tr10s)
  + [Intro Stream Processing](https://medium.com/stream-processing/what-is-stream-processing-1eadfca11b97)
  + [Design Patterns for Streaming Data Analytics](https://www.slideshare.net/Hadoop_Summit/design-patterns-for-real-time-streaming-data-analytics)
  + [Streaming Analytics 101](https://www.dataversity.net/streaming-analytics-101/)
  + [DZone Kafka Reference Card](https://onyourside.sharepoint.com/:b:/r/sites/IT-Architecture/ALT/Shared%20Documents/02%20-%20Capability%20Uplift/ALT%20Initiatives/2017%20ALT%20Decision%20Velocity/Ref%20Spec%20-%20Event%20Streaming/7714451-dzone-rc254-apachekafka.pdf?csf=1&e=ccYW0C)
  + [DZone Stream Processing Reference Card](https://onyourside.sharepoint.com/:b:/r/sites/IT-Architecture/ALT/Shared%20Documents/02%20-%20Capability%20Uplift/ALT%20Initiatives/2017%20ALT%20Decision%20Velocity/Ref%20Spec%20-%20Event%20Streaming/9223150-dzone-refcard265-streamprocessing-0523.pdf?csf=1&e=F0B7VH)
  + [What is Stream Processing](https://medium.com/stream-processing/what-is-stream-processing-1eadfca11b97)
  + [Amazon Streaming Data](https://aws.amazon.com/streaming-data/)
  + [Event Driven Architecture Pattern](http://radar.oreilly.com/2015/02/variations-in-event-driven-architecture.html)

# Document History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Summary of Changes |
| 0.9.1 | 01/17/19 | Stream Team | Initial Draft |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

1. See also Patterns: Batch, Compensating Transactions [↑](#footnote-ref-2)