FRANKFURT UNIVERSITY OF APPLIED SCIENCES

FACULTY 2: COMPUTER SCIENCE AND ENGINEERING

HIGH INTEGRITY SYSTEMS

Master Thesis

AN EVALUATION OF DIFFERENT OPEN SOURCE ESP PLATFORMS TOWARDS CONSTRUCTING A FEATURE MATRIX

Student:	Vo Duy Hieu
Matriculation number:	1148479
Supervisor:	Prof. Dr. Christian Baun
2^{nd} Supervisor:	Prof. Dr. Eicke Godehardt

February 1, 2021.

Official Declaration

I declare that this thesis has been written solely on my own. I herewith officially ensure that the work presented in the thesis is my own. I certify, to the best of my knowledge, my thesis does not violate anyone's copyright. Any external sources from the work of other people included in my thesis are fully acknowledged with citation and referencing.

Date

Signature

Acknowledgement

Abstract

Contents

1 Introduction

Nowadays, the explosion of the number of digital devices and online services comes along with an immense amount of data that is auto generated or collected from the interactions of users. For instance, from 2016, the Netflix company already gathered around 1.3 PB of log data on a daily basis [1]. With this unprecedented scale of input data, companies and organizations have tremendous opportunities to utilize them to create business values. Many trending technologies such as Big Data, Internet of Things, Machine Learning and Artificial Intelligent all involves handling data in great volume. However, this also brings about a challenge to collect these data fast and reliably.

Once the data is ingested into the organization, it needs to be transformed and processed to extract insights and generate values. In the context of enterprise applications, as the systems grows over time with more services, the need for an effective data backbone to serve these huge amount of data to these services and to integrate them together while maintaining a good level of decoupling becomes inevitable.

Moreover, all these steps of collecting, processing and transferring data must be done in real-time fashion. One of the prominent methods is Event Stream Processing (ESP) which treats data as a continuous flow of events and use this as the 'central nervous system' of the software systems with event-driven architecture.

1.1 Motivation

To develop a system evolving around streams of events, the primary basis is a central event store which can ingest data from multiple sources and serve this data to any interested consumer. Usually a ESP platform will be used as it is designed orienting to the concept of streaming. However, in order to choose the suitable platform, user will usually be burdened by a plethora of questions which need to be answered. The concerns include how well is the performance and reliability of the platform, does the platform provide necessary functionalities, will it deliver messages with accuracy that meets the requirements, can the platform integrate with the existing stream processing framework in the infrastructure, to name but a few.

As there are many platforms now available on the market both open-source and commercial with each having different pros and cons, it could be challenging and time consuming to go through all of them to choose the most suitable option that matches the requirements. It would be greatly convenient to have a single standardized evaluation of these platforms which can be used as a guideline during the decision making process. Therefore, the goal of this thesis is to derive a feature matrix to help systematically determine the right open-source ESP platform based on varying priority in different use cases.

1.2 Related Work

There are a number of articles and studies which compare and weigh different platforms and technologies. Many of them focus on evaluating the performance between platforms. There are comparisons of time and resource behavior of Apache Kafka and Apache Pulsar [2] [3], time efficiency between Apache Kafka and NATS Streaming [4].

Some other surveys cover more platforms and a wider range of evaluating aspects such as the comparisons of Apache Kafka, Apache Pulsar and RabbitMQ from Confluent [5] [6]. However, these assessments are conducted only briefly on the conceptual level. Apart from these studies, most of the scientific researches only concentrate on comparing different stream processing frameworks such as Apache Spark, Apache Flink and Apache Storm [7] [8] [9]. Therefore, in general, there is still lack of in-depth investigation into the differences of ESP platforms and their conformability with event-driven use cases and this is where the thesis will fill in.

1.3 Contribution

In this thesis, three open source ESP platforms, namely, Apache Kafka, Apache Pulsar and NATS Streaming are selected for evaluation based on preliminary measures and reasoning. Each platform is assessed against a set of criteria covering all important quality factors. The results are summarized in the form of a feature matrix with adjustable weighting factors of quality categories and features. Therefore, the matrix can be tailored to the need of user and adapted to individual use case to determine the most suitable platform for that case according to its priorities.

In the evaluation, sample implementations and code snippets are presented to illustrate the features of the platforms. Moreover, best practices for each platform in different use cases are also drawn out. These can be used as a reference for actual implementation of applications on top of these platforms.

1.4 Organization of this Thesis

The thesis is organized as follows. Chapter 2 gives a short theoretical background of the topics event-driven architecture, stream processing and ESP platforms. Chapter 3 enumerates prominent open-source ESP platforms currently available and furthermore derives criteria to choose the top three platforms which are considered in this thesis. Moreover, it also includes the elaboration of comparison metrics. After that, chapter 4 presents the evaluation of each platform against the comparison scheme and gives a discussion on the resulted feature matrix. Finally, the conclusion summarizes and proposes future improvement for the matrix.

2 Background

In order to conduct the comparison effectively, it is necessary to first lay a good theoretical basis of event-driven architecture, event stream processing and the concrete roles of an ESP platform. Based on that, a comprehensive set of evaluation metrics can be determined.

2.1 Event Driven Architecture

2.1.1 Monolithic Architecture

A monolithic application focuses on the simplicity. All components including user interface, business logic and data accessing services are developed, packaged and deployed in a single unit [10].

Monolithic applications are straightforward to develop and test since the control flow is very transparent. Deployment and scaling are also simple because there is only a single software artifact. A monolithic architecture can also result in better performance for small application with few users [11]. Therefore, when starting a new software project, monolith should be the first choice to be considered [12].

2.1.2 Microservices with Event-Driven Approach

As the application expands, monolithic architecture gradually becomes more rigid and harder to adapt to the agile development cycle. A small change in any service will lead to rebuilding and redeployment of the entire application. Moreover, developers in the team must work closely together and cannot freely establish their own paces. For that reason, the microservices architecture arises. In general, an application is disassembled into services according to different business capabilities [13]. These services are self-contained and loosely-coupled with each other. Each of them maintains a separate database and expose its data with other only via a mutually agreed contract. Since every service itself is an independent deployable, it can have its own development cycle and technology stack. Services also allow finer-grained scaling of the application. This is very useful for reducing cost when deploying the application to the cloud [14].

In microservices architecture, services need to have a mechanism to coordinate and work together to achieve end results. There are typically two approaches for this task, namely, request-driven and event-driven [15].

In the first approach, a service sends command to request for state change or queries current state in other services. This method is hard to scale because the control logic concentrates on the sender of requests. Adding a new service usually involves code change in others to include it in the control flow. For the latter approach, services communicate with each other using events.

Event
An event represents something occurred in the past and cannot be altered or deleted [16]. It is simply a fact stating what has happened in the system. Whenever a service updates its state, it sends out an event. Any other service can listen and operate on this event without the sender knowing about it. This leads to inversion of control when the receiver of events now dictates the operational logic. As a result, services of the system can be more loosely coupled. New service can be easily plugged in the system and start to consume events while other services remain unmodified. There are three common ways to use events, namely, event notification, event-carried state transfer and event sourcing [17].

Event Notification
The events are used only for notifying about a state change on the sender. Receiver decides which operation should be executed upon receiving the events. This usually involves querying for more information from either the publisher of events or another service.

The approaches of event notification and request-driven ensure a minimum level of coupling by letting each service manage its own data and only share when being requested. Another advantage is that the state of each service is consistent throughout the entire system since it only exists in one place. Nevertheless, these approaches only works at their best when services are truly independent from each other which is usually not the case in reality. It is unavoidable that services must maintain a certain level of dependency and sharing of data. When services grow with more functionalities, they need to expand the service contract to expose more data to outside which then leads to higher coupling. This is known as the dichotomy of data and services [18]. Therefore, the two following patterns tackle this problem by proactively allowing services to openly share their data instead of encapsulating it within each service.

Event-Carried State Transfer
With this pattern, each event encloses more detail information about what has been changed as well as how it has been modified. As a result, current state of any service can be reconstructed anywhere by applying its published events on the same initial state in the same order. Therefore, every subscriber can retain a local state replica and keep it synchronized with the source of events.

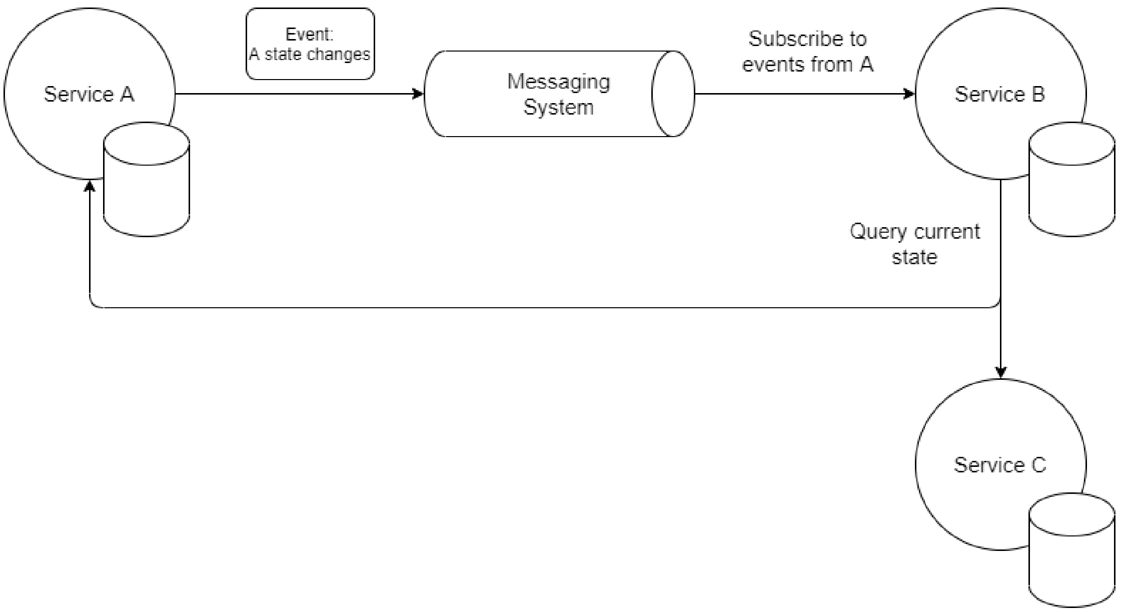

Figure 2.1: Services coordination with event notification.

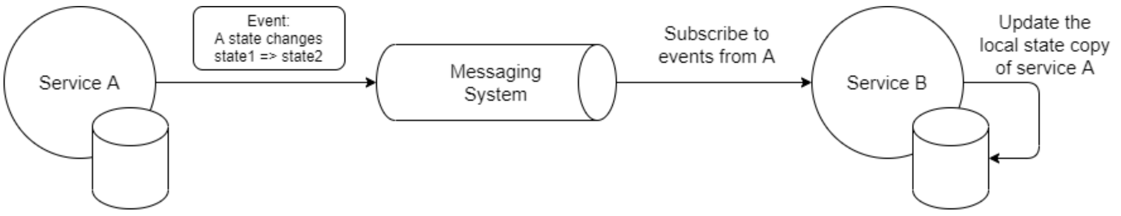

Figure 2.2: Services coordination with event-carried state transfer.

When a service keeps a state copy of another locally, it can access this data faster and becomes independent of the online status of the source of data. Nevertheless, having multiple copies of data across the system also means that the system can be in inconsistent state temporarily or even worse permanently if it is not designed carefully. This concern is closely related to the problem of how to atomically update the local state and publish a corresponding event [19].

Event Sourcing and Command Query Responsibility Segregation (CQRS)
The event sourcing pattern takes one step further. Events are not used to only notify and help replicate states among services but they are now the primitive form of data store [20]. Instead of updating the current state, services record every state change as events which are stored in the order of occurring in append-only fashion. This stream of events becomes the source of truth for the entire system. Services can derive the state of any entity from the events stream and keep it in memory or a cache to query when needed. Snapshot of the current state can be periodically created to avoid processing through all events after every restart.

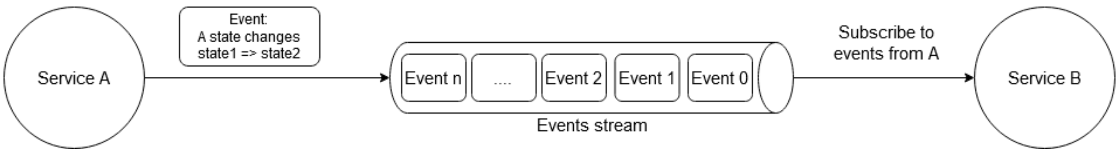

Figure 2.3: Services coordination with event sourcing.

This approach eliminates the need of atomically updating state and generating event as in event-carried state transfer since there is now only publishing of events. Apart from providing a loosely-coupled and scalable way for coordination among services, event sourcing comes with additional benefits. The stream of events retains the entire history of the system and therefore can be used for auditing, extracting more value from past events, troubleshooting problem during production as well as for testing and fixing errors by simply replaying the events with the new system [21].

Event sourcing is often used together with CQRS [16]. The CQRS pattern promotes the idea of having two independent sides of command and query corresponding to two separate models for writing and reading data respectively. By doing this, each model can scale independently based on the load of each side. Moreover, this gives the flexibility to derive multiple ways to query data on the reading side without being tightly coupled to the data model of the input side.

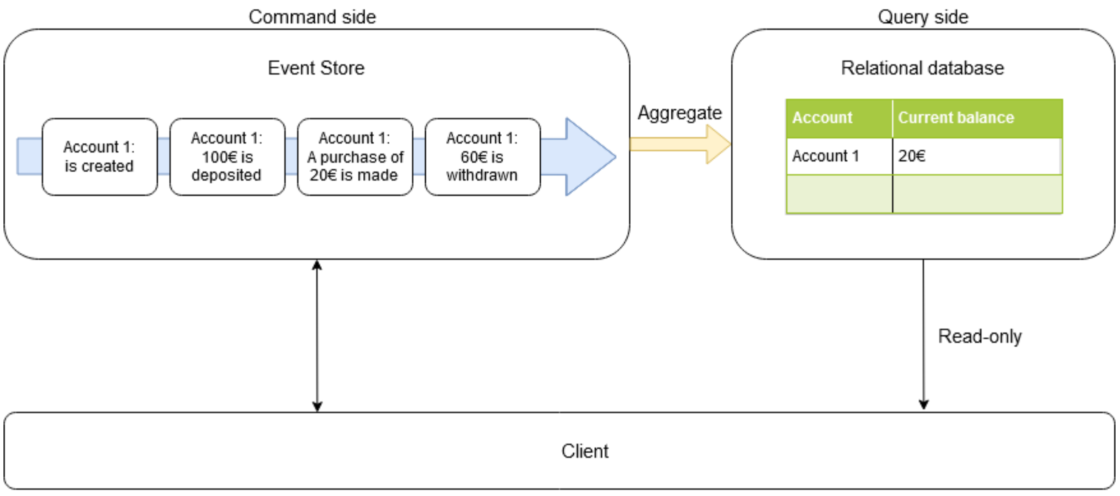

Figure 2.4: Event sourcing and CQRS.

In the context of event sourcing, the command side of CQRS is the stream of events. Greg Young pointed out a number of reasons why this is a good writing model [16]. The append-only operation to add new events makes it more scalable. Moreover, there is no impedance mismatch between domain model of the application and the storage model as in the case of relational database because event is also a domain concept. On the query side, the representation of current state are created with the underline events stream. Figure 2.4 illustrates an example of a simple bank account management system. Transactions of users are recorded as events. The queryable current account balance is derived by aggregating the history of transactions and writing it to a relational database which is solely for reading.

Event-driven architecture and its related patterns are useful but it also comes with a cost of increasing complexity in the system [22] [23]. As can be seen from the above short summary, one of the most significant added values of this approach is to reduce the coupling and lay the foundation for systems with high demand for scalability. If this is not the case, a simpler approach should be chosen to not overcomplicate the system.

2.2 Event Stream Processing

With the increasing amount of data, the demand about how data is processed and analyzed also evolves over time. With the batch processing paradigm, data is collected over a predefined period of time and stored in a big bounded batch in a data warehouse. Some scheduled batch jobs will then go over the entire batch of data to generate insights and reports tying to the needs of organizations. However, this type of data processing gradually cannot keep up with the need of faster analysis to allow companies to response more timely to changes. Therefore, the concept of stream processing begins to emerge.

Unlike its batch counterpart, stream processing aims at handling unbounded data which is a better form for representing the flow of events in real world given their continuous and boundless nature. Stream processing frameworks are designed to handle this type of data with high throughput and horizontal scalability. A stream processor can ingest one or more input streams and execute its business logic. It can also generate new data to other output streams which subsequently can be consumed by other stream processors. The data flow is continuous and new stream processors can always be added to incorporate new processing logic and generate more results.

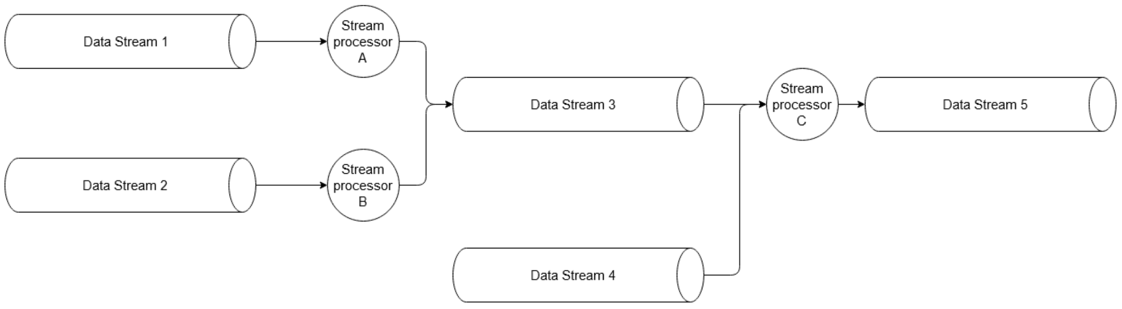

Figure 2.5: Stream processing concept.

By processing this influx of data continuously as they arrives, events and patterns can be detected with low latency making stream processing more suitable for real-time use cases. Moreover, the input data can come from numerous sources with varying transmission rates. Therefore, data may arrive late and out of order with respect to the time it is generated at the source. In this case, for it sees data in an endless fashion, stream processing gives more tolerance for late data and more flexibility to assemble data into windows and sessions to extract more useful information. It is even suggested that a well-designed stream processing system with guarantee of correctness and the capability to effectively manage the time semantics could surpass batch processing system [24]. Back in the time when using stream processing was a trade-off between accuracy and low-latency, it was a popular idea to run two batch and stream processing pipelines in parallel to achieve both fast and correct results [25]. As stream processing engines grow more mature and accurate, the demand for such system is lessened [26].

2.2.1 Stream Processing and Event-Driven Architecture

Stream processing is not merely a data processing paradigm to achieve low latency result. Applying the concept of streaming and event-driven architecture on the organizational level also has the potential to help build more scalable and resilient software infrastructure for organizations. A big institution using software in every operational aspect usually has a sophisticated software infrastructure providing different functionalities such as business applications, query and analytical services, monitoring system, data warehouse. In this case, the problem of coupling emerges not only between services of applications but also between these processing and data systems on a bigger scale. Data needs to be shared and synchronized between these components. Without an efficient way to integrate system-wide data, the entire system can quickly turns into a big tangled mesh. As an example, this problem was experienced at LinkedIn as their system became increasingly complex [27].

Jay Kreps described in his article a log-centric infrastructure that can solve this problem [28]. Every event recorded by any data system in the organization, is written orderly to an append-only log. This becomes the backbone for all data systems in the organization similar to using event-sourcing in a distributed system but on a bigger scope. Each system has the flexibility to derive different data structures from the raw events to match its specific access pattern. The task of data representation is now done at individual system instead of in the central data store. This is referred by Martin Kleppmann as turning the database inside out [29]. With this approach, an entire software infrastructure of organization now turns into a gigantic distributed database with the log of events lying at the center and all systems are simply indexes on the raw data in the form of events [28].

Processing streams of events on an organizational scale is a non-trivial task. This is where stream processing fits in by providing a scalable tool to handle such amount of data. Applications and their decoupled services can use stream processing to continuously

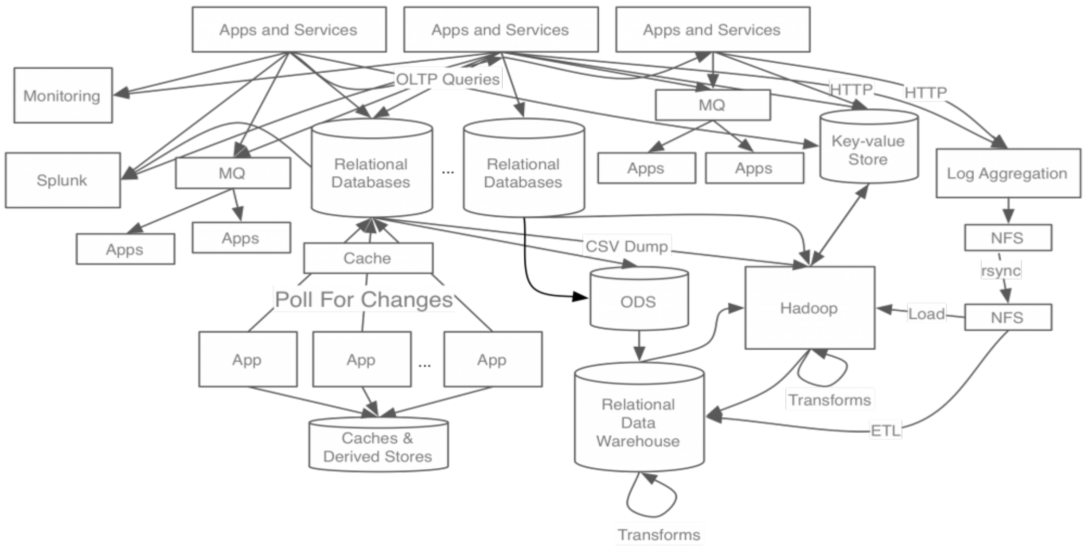

Figure 2.6: The tangled data systems at LinkedIn in the old day [27].

consume, process and generate new events while data systems can use streaming tool to continuously integrate new data from the streams to their data representation. The result is a more neatly organized infrastructure with every system synchronizes and communicates via the event streams using stream processing.

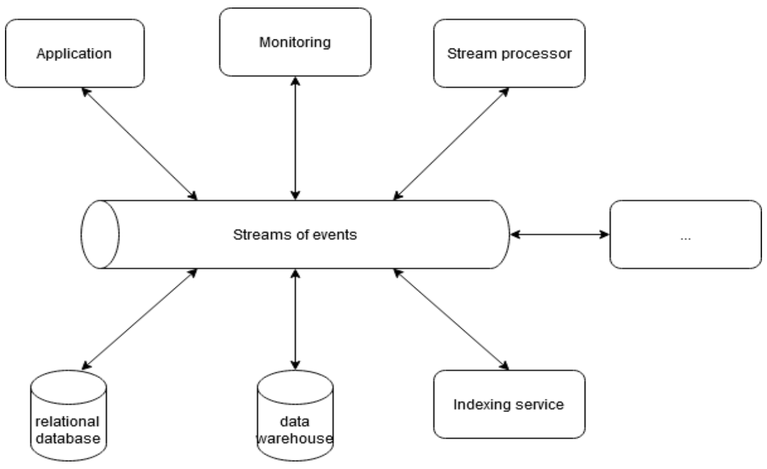

Figure 2.7: System with streams of events as the single source of truth.

2.3 Event Stream Processing Platform

An ESP platform must facilitate the construction of software systems revolving around streams of events. According to Jay Kreps, it has two main uses [27]. Firstly, it must provide the necessary infrastructure and tools for applications to work and coordinate with each other on top of events streams. Secondly, the platform can serve as the integration point where various data systems can attach themselves into and synchronize data among them continuously.

To fulfill these two uses, the following fundamental capabilities are required. A platform must have an events storage layer. Optimally, it should also support the option to persist events for an infinite period of time since this will be the single source of truth that all data systems depend on. Accessing interface must be provided for applications to publish and consume events. Based on this, custom applications and services can already be self-implemented to do stream processing as well as integrating data to different destinations. However, such platform only provides minimal functionalities and burdens developers with many low-level implementation tasks.

In order to be more useful and easier to be integrate into the infrastructure of organizations, an ESP platform must provide more supporting tools. More particularly, the platform should also support the processing of events streams either by providing a native stream processing tool or being able to integrate with external stream processing framework. For the data integration, the platform should come with ready-to-be-used tools to integrate with a wide range of existing data systems effortlessly including also legacy systems. Moreover,the platform should have a rich set of utility tools for monitoring and management.

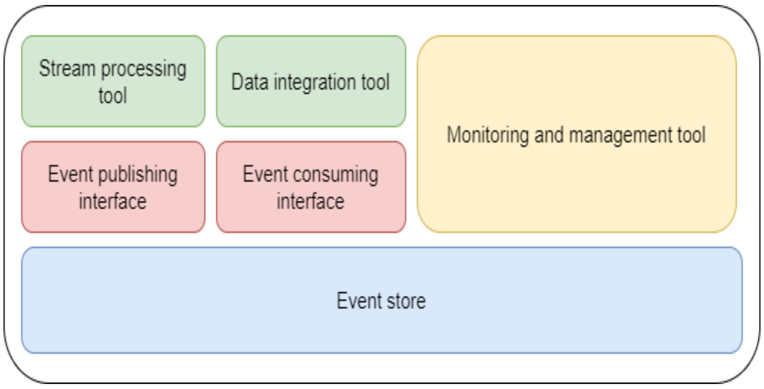
Stream processing tool
Data integration tool
Monitoring and management tool
Event publishing interface
Event consuming interface
Event store

Figure 2.8: Capabilities of an ESP platform.

All of these capabilities should be in a real-time, high throughput, scalable and highly reliable fashion so that the platform would not become the bottleneck in the infrastructure.

3 Evaluation Scheme

3.1 Considered Platforms

As the concept of using stream as a source of truth gains more attention and becomes more popular, many projects aiming at creating a processing platform evolving around streams started to take shape. Many companies first started their projects as in-house products and then later open-sourced them to enhance the development pace with the help of community. Kafka, which was first developed at LinkedIn, is the first prominent name in the field [30]. It was later open sourced to the Apache Software Foundation. Yahoo! also created their own stream processing platform named Pulsar and it is now also an Apache project [31]. The company Alibaba also joins the trend by open sourcing their RocketMQ to Apache Foundation [32]. In addition, there is the NATS streaming server, which is developed and maintained by the company Synadia on top of NATS messaging system to provide stream processing capability [33]. It is currently an incubating project of Cloud Native Computing Foundation. Pravega is also a quite new open-source project in the field [34].

Since an adequate evaluation for all these platforms could not be contained within the scope of the thesis, only three platforms will be selected based on a set of criteria indicating the maturity, the size of the active community, the popularity of the platform and the quality of documentation.

Maturity is evaluated based on the project stage in an open source foundation if it belongs to one and the date of the first release. Because each project has different in-house phase before open-sourcing, the date of release on GitHub is chosen as the standardized criterion.

The size of active community is determined by the number of contributors of the project on GitHub and the number of related questions on Stackoverflow.

To assess the popularity, the number of GitHub stars and Google trend points of each project in the last 12 months are used.

The result of the preliminary evaluation is summarized in Table 3.1. The data and number presented in the table were collected in 11/2020.

According to these criteria, Apache Kafka and Apache Pulsar are the leaders in all preliminary categories. Among the three remaining platforms, RocketMQ is the most popular in the community. However, the document page of RocketMQ is unfortunately

Table 3.1: Preliminary evaluation of 5 open-source ESP platforms.

		Apache Kafka	Apache Pulsar	RocketMQ	NATS Streaming	Pravega
Maturity	Project stage in open-source foundation	Graduated project of Apache Software Foundation	Graduated project of Apache Software Foundation	Graduated project of Apache Software Foundation	Incubating project of Cloud Native Computing Foundation	Sandbox project of Cloud Native Computing Foundation
	The first release date on GitHub	11/2012	09/2016	02/2017	06/2016	09/2017
Active community	Number of contributors on GitHub	727	327	211	34	65
	Number of related questions on Stackoverflow	21 447	144	57	47	0
Popularity	GitHub stars	17 300	6800	12700	2100	1300
	Google trend points	40	72	5	2	2
Documentation	Quality of the content	Very Good Update regularly with each release Cover in detail design and features Blogposts and books from Confluent, a company founded by the creators of Kafka	Good Update regularly with each release Cover in detail design and features	Bad Do not update regularly, some content is even from 2016 Cover only a few examples	Medium Update regularly Good coverage of general concept and features	Medium Update regularly with each release Good coverage of general concept and features

very inadequate and outdated. This could become a great problem during the evaluation. Therefore, RocketMQ will not be considered in the thesis. Regarding Pravega, despite the promising set of rich features, it is not mature enough since it is still in the sandbox stage at the Cloud Native Computing Foundation. Moreover, the community of Pravega is still too small. On the other hand, although NATS streaming has a moderate community, it has a reasonable maturity and quality of documentation. Moreover, it is built on top of NATS messaging system which is used in industry by many companies such as Siemens, Pivotal, GE and therefore can inherit its stability. The evaluation of NATS streaming can be greatly beneficial for organizations which already have NATS messaging in the infrastructure to integrate NATS streaming more easily.

As a result, the three platforms which will be inspected in depth are Apache Kafka, Apache Pulsar and NATS streaming.

3.2 Evaluation Metrics

The evaluation categories for distinct functionalities of an ESP platform are derived based on the necessary capabilities in section 2.3. Since there are not many major scientific works on evaluating ESP platforms, concrete criteria which are directly related to ESP platforms cannot be found. Therefore, criteria in these categories are determined mainly by self-deducing from the literature research of related technology to each capability of an ESP platform or event-driven concept and from interviewing with experts at the company Novatec who work intensively with ESP platforms.

For general functionalities such as security and non-functional criteria, the comparison categories are adapted from ISO25010 software quality model to have a good coverage of main quality aspects [35]. However, non-applicable characteristics from the standard such as maintainability which refers to the inner structure and complexity of the system will not be included. As a result, 12 main comparison categories are determined.

1: License:

1.1: Open-Source license: there are many open-source licenses available and each of them can impose some restriction on the use of the platforms. Users can have different intentions when using an open-source ESP platform such as for internal use or to build a commercial product on top of it. Therefore, this criterion is added to help users quickly decide which platform has the suitable license for their business model. It will not contribute to the points of the platforms in the evaluation.

2: Event storage: this category is directly corresponding to the event store capability of the platforms. In this category are the criteria to evaluate this storage functionality.

2.1: Durable storage: with event-driven application, the event stream is the backbone that all application and services depend on. Therefore, events on an ESP platform must be stored in non-volatile storage. It must be guaranteed that once an event is confirmed to be persisted, it must survive system crash or power outage.

2.2: Flexible data retention policy: For event-driven application, events are the source of truth of the systems. They can be consumed, replayed by numerous services and applications at different rates and times. Therefore, the platform must support a long data retention period to give services enough time for events consumption. The platform should also support selective retention of data since this can be useful for certain use cases to save disk storage.

2.3: Data archive in cheap storage (hot/cold storage): For use cases where the entire history of events must be retained, it is also desirable to have the option to automatically offload old and low-demanded data to cheap storage to save cost. On the other hand, newer data can still be retained in hot storage on the platform to be served to clients faster.

In order to evaluate the capabilities to allow publishing and subscribing events of the platform, two evaluation categories are derived, namely, messaging patterns and messaging semantics. These categories include evaluation criteria taken from the concept of messaging systems which are corresponding to this functionality on the ESP platforms.

3: Messaging patterns: In this category, the evaluation of various possibilities to deliver events from producer to consumer with the platform as the messaging middleware is considered. This is directly related to the concept of asynchronous messaging and is dictated by the messaging patterns supported by the platform [36]. Therefore, most common and related patterns of message delivery will be used as evaluating criteria for these platforms.

3.1: Publish-Subscribe [37]: With this pattern, a message is delivered to multiple consumers, each of which will receive the same set of messages. This pattern is very relevant in the context of event stream processing since one event stream can be consumed by numerous independent receivers and each of them requires an entire history of events for a different processing logic. Therefore, it must be supported by the platform.

3.2: Competing consumers [**?**]: This pattern is very important to allow multiple consumers to consume a messaging channel to balance the load and increase throughput. Each message will be delivered to only one of the competing consumers. This is suitable for the event-driven use case where each event in the stream is self-contained and can be processed separately such as using event to trigger a reaction. The number of events to be processed can be significant, the capability to scale with competing consumer is very important and therefore this pattern is included in the evaluation.

3.3: Publish-Subscribe + Competing consumers (Consumer group) [**?**]: With Competing consumers pattern, a message can be delivered to any available consumer to maximize concurrency and throughput. In this scenario, each message can be interpreted and processed individually. For certain event-driven patterns such as event-sourcing, it is essential to have the entire history of events to reconstruct the system state. The Publish-Subscribe pattern is more suitable for this purpose. However, an event stream can retain large data volumes with events of many different entities which could overwhelm the processing capacity of a subscriber. Therefore, it can be very useful if the platform supports the combination of two patterns Pub-Sub and Competing consumer to scale each subscriber of an event stream. In this way, events can be consumed by several competing consumers where each consumer will receive only events from a subset of the entities belong to that event stream. This pattern is sometimes referred as consumer group.

3.4: Event playback: This is not directly related to asynchronous messaging but is very important in event-driven architecture. With event streams as the source of truth, it is also very important for any consumer to be able to re-consumes older events. The ability to replay events is indeed one of the key features of Event Sourcing pattern [20]. Therefore, the platform must support this access pattern for replaying past events with various starting points, namely, specific position in the event streams or specific time point.

3.5: Content-based routing [38]: With this pattern, the consumer of a message channel can selective receive based on some information in the messages. With event-driven approach, each event stream usually retains event of the same type but from multiple source entities. It is very common that some downstream consumer only wants to receive events coming from a specific source. If the pattern is supported on the platform, it can be very useful in such scenarios.

4: Messaging semantics: This category focuses on the concern of correctness of delivered data. Forwarding message from producer to consumer alone is not enough. The messages must also arrive on the consumer side correctly with a certain level of delivery guarantee and be interpretable by the consumer.

4.1: Strong ordering guarantee: For events, the order is very important to correctly reproduce the system state from event streams because events represent state changes in the. Events in different orders will result in different system states. However, in distributed systems, order guarantee is very hard. Failures can happen anytime causing delays, retries and hence out-of-order. To really achieve order guarantee requires the cooperation between the platform and clients and also compromise on throughput. The platform must lay a good basis to allow collaboration to guarantee order of events traversing from producer, through the platform and to the end consumer.

Message delivery guarantee is another hard problem in distributed systems. In an unreliable environment, failures are unavoidable which can lead to connection disruption while sending messages. There are three different levels of guarantee in such scenarios a messaging system can provide, namely, at-least-once, at-most-once and exactly-once with different tradeoffs between performance and reliability. An ESP platform should allow users to choose different tradeoffs depending on their use cases. Therefore, these guarantee levels are adapted as evaluation criteria for the platforms.

4.2: at-most-once delivery semantics: In case of failures, the message can simply be dropped which results in message loss but there will be no duplication on the consumer side.

4.3: at-least-once delivery semantics: In case of failures, message can be resent until success which may lead to message duplication. The consumer is guaranteed to receive each message at least once.

4.4: exactly-once semantics: exactly once delivery on the other hand cannot be physically achieved because systems communicating over an unreliable channel like network will never be ensured about the status of the published message [39]. This criterion actually evaluates seemingly exactly-once processing capability instead of exactly-once delivery guarantee. More particularly, it is possible that a message can be redelivered and reprocessed by consumer multiple times. However, the result of processing should be the same as when message is received and processed exactly once. This feature is very important in mission-critical application such as financial transaction.

4.5 and 4.6: Support schema records, Schema management and evolution: events or messages in general usually have certain structure with specified fields. This is very important for consumer of messages to be able to interpret them. Therefore, the platform should support sending and receiving schema records with various common schema data serializing systems such as Avro, Protobuf, Json. In distributed systems where producers and receivers of data are independent, it is very important that they are in agreement about the schema of records to avoid mismatch interpretation which can lead to non-processable records. Moreover, the structure of the messages and their schema can be upgraded over time to adapt new requirements. Therefore, it is necessary to have a schema management system which handles the evolution of schema overtime and enforce validation rules to ensure the compatibility of new schemas to both producer and consumer. Thus, platforms should also support schema management mechanism.

5: Stream processing: The criteria in this category are used to evaluate how well the platform provide the stream processing capability on its persisted event streams.

5.1: Native stream processing: This can be a very useful feature to develop stream processing applications that integrate better with the platform. If it is provided by the platform, detail elaboration and evaluation will be conducted to see if important functionalities are supported such as: windowing function, time semantics, aggregation functions.

5.2: Integration with external stream processing frameworks: Apart from native stream processing, it should be also possible to integrate the platform with existing and mature stream processing frameworks such as Apache Spark, Apache Flink. This criterion will help determine the available integrable frameworks of each platform.

5.3: Simple, high-level stream processing: In addition to native stream processing and external stream processing frameworks, it would be useful if the platform also supports stream processing capability in the form of simple query to help people with little software development background quickly gain insight into the events streams.

6: Data integration and Interoperability: in this category, the ability of the platform to integrate and share data with different types of systems and clients are evaluated.

6.1: Connectors to external systems: since the streams will be the data backbone of the system, the platform should have a standard framework to ingest and export data with various data systems. Moreover, a wide range of ready-to-be-used connectors should also be supported to ease the need of self-implementing integration service.

6.2: Supported programming languages for client: to help services and applications integrate easier, an ESP platform should support a wide range of clients in different programming languages. This factor could also be important for the decision making of a company to use it or not. This criteria will be evaluated based on number of currently supported clients for each platform.

7: Monitoring and Management: in this category, the set of operational tools provided by each platform will be inspected.

7.1: Technical monitoring: during operation of the platform, it is very important to keep track of the health of nodes in the cluster via metrics such as: CPU usage, Memory used, messages throughput so quickly react to changes. Therefore, it is very important to be able to set monitoring system for the platforms.

7.2: Event tracing monitoring: in addition to technical monitoring, it can also be very useful to have monitoring systems on a higher level to give an overview and quick inspection of the flow of events through the platform and also the content of each event.

7.3: Admin API: to help user manage and configure the platform, administration API should be exposed and rich set of managing tools should be provided.

7.4: Professional support: although the evaluated technologies are all open-source, it is not always convenient for users to self-manage everything to maintain and operate with the platform. Users might want to delegate the tasks of deployment, management of the platform to service providers to focus more on developing their business logic. Therefore, the number of available managed services of these platforms is also a good evaluation criterion to consider.

8: Scalability: as the data backbone, an ESP platform will usually have to handle a huge amount of data. Therefore, it should be easily scalable to quickly adapt to new demand of data volumes.

8.1: Scalability of storage and computing server: the two fundamental functionalities of an ESP platform are persisting events and serving read/write requests from clients. The demands for storage and messaging consumption could vary greatly. Therefore, these two layers should be scalable and optimally can be scaled independently with minimum manual administering from users.

9: Security: this category of criteria is adapted from the ISO25010 quality model. This is very important for any data system in general. In case of ESP platform, security is a very important factor since the platform is the data backbone for the organization which can retain many sensitive data which needs to be protected.

9.1: Authentication mechanism: this is one of the fundamental security mechanisms to verify the legitimacy of clients connected to a system. The platform should support authentication mechanisms. They could be built-in mechanisms which only work within the platform. More optimally, the platform should support pluggable mechanisms of common authentication scheme such as Simple Authentication and Security Layer (SASL) so that it can be integrated more easily to the existing security infrastructure in the organization. 9.2: Authorization mechanisms: different clients of the platform can have different levels of access rights. Therefore, the platform should support authorization mechanisms to control access of clients. There are two common types of access control, namely, Role-Based Access Control (RBAC) which defines access rights based on business roles, and Access Control List (ACL) which allows more fine-grained control on the level of individual clients. The platform should support these mechanisms.

9.3: Encryption: this criterion covers the confidentiality of data on the platform. There are different levels of encryption, namely, encrypted transmission, encryption of data at rest and end-to-end encryption. The evaluation will be conducted based on the number of supported encryption levels on each platform.

9.4: Multi-tenant: In case multiple teams and departments rely on an ESP platform to operate, multi-tenant feature needs to be supported. Otherwise, different cluster will have to be set up for each individual team which then increase the cost of operating and maintenance. Therefore, this is a nice feature to have on an ESP platform.

10: Reliability/Recoverability: this non-functional category is adapted from the quality model and aims at evaluating the failover mechanisms of the platforms in case of failure of different components, namely, data store, client, serving layer, and the entire data center.

10.1: Event storage is fault tolerant: As the data integration point for the system, there must be no single of point failure for the event storage. Therefore, the platform should provide failover in case of failure of data storage so that the events can continue to be served to applications and services.

10.2: Consumer group failover mechanism: If the platform supports the consumer group in criterion 3.3, there should be failover mechanism within the group in case of one or more consumers fail to avoid disrupted consumption.

10.3: Broker failover mechanism: the platform should handle the case when one or more server instances fail. In this case, the requests from clients should not be disrupted.

10.4: Geo-replication: for significant systems that span across the globe, an ESP platform should support out-of-the-box data replication between different data center not only to increase the response time for client but also for fault-tolerance and high availability in case one of the data center is down.

11: Usability and Community: this category is also derived from the ISO25010 standard. This refers to how fast and easy for users to get used to the platform and start to integrate it into their system. This can be a very important factor when choosing an ESP platform.

11.1: Community size: for usability and learnability, the size of the active community is very important. A community of many users and active developers could be a great source of support. Moreover, apart from learnability, a platform with a big community can gain more contributions such as detecting issues, adding new features. This can help speed up the development and enhance the quality of the platform.

11.2: Available training courses: in addition to available document, there should be a good selection training courses available.

11.3: The ease to start the development: It should be easy for developers to quickly start development, build prototype applications with the platform and so forth.

12: Performance: this category is taken from the ISO25010 quality model. This focuses on how good the platform can perform its functions while guaranteeing an acceptable processing speed.

12.1 and 12.2: End-to-end latency, Throughput: since the demand for low latency in stream processing is very high, the platforms should have reasonable end-to-end latency to deliver messages from producer to consumer. Moreover, as the data backbone, huge data volumes from all kinds of systems and applications will pass through the platform and therefore, high throughput must be guaranteed. These two criteria will be evaluated based on literature research since there are already many works on comparing time behavior of these ESP platforms. The platforms will be sorted based on their time behaviors with the platform with the best performance in the first place and the platform with the lowest performance in the third place. The grading will be done accordingly.

After deriving this set of criteria, a discussion was held with experts at Novatec to determine which criteria in each category are more essential for an ESP platform or of greater importance in their daily works with an ESP platform. These criteria are marked as high priority. In the scope of the thesis, only these criteria will be the focus and be evaluated in-depth.

For different use cases, the priority might be different. Therefore, despite not being considered in detail, the other criteria could serve as a general guideline to compare and evaluate in different scenarios.

Table 3.2: Considered evaluation criteria in the thesis.

No.	**Evaluation category / Evaluation criteria**
1	**License**
1.1	Open-source license
2	**Event storage**
2.1	Durable storage
2.2	Flexible data retention policy
3	**Messaging patterns**
3.1	Publish-Subscribe
3.2	Competing consumers
3.3	Publish-Subscribe + Competing consumers (Consumer group)
3.4	Event playback
4	**Messaging semantics**
4.1	Strong ordering guarantee
4.2	At-most-once delivery semantics
4.3	At-least-once delivery semantics
4.4	Exactly-once semantics
5	**Stream processing**
5.1	Native stream processing
6	**Data integration and Interoperability**
6.1	Connectors to external systems
6.2	Supported programming languages for client
7	**Monitoring and Management**
7.1	Technical monitoring
8	**Scalability**
8.1	Scalability of storage and computing server
9	**Security**
9.1	Authentication mechanisms
9.2	Authorization mechanisms
9.3	Encryption
10	**Reliability / Recoverability**
10.1	Event storage is fault tolerant
10.2	Consumer group failover mechanism
10.3	Broker failover mechanism
11	**Usability and Community**
11.1	Community size
12	**Performance**
12.1	End-to-end latency
12.2	Throughput

4 Evaluation of Platforms

4.1 General concepts

Before going into comparison, the general concepts of the three platforms will be quickly recapped to provide a basis about the working mechanism of these platform. More detail elaboration of their features will be presented during the evaluation.

Apache Kafka

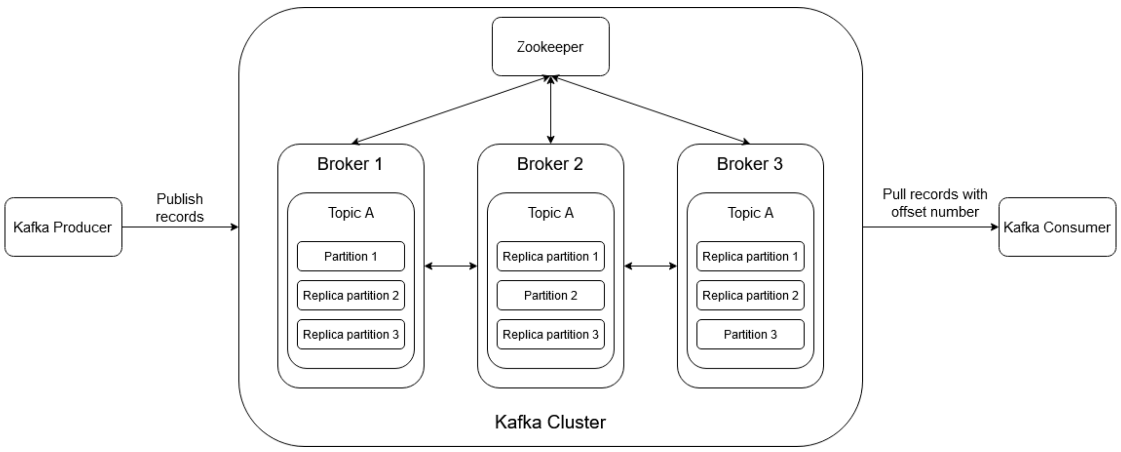

Figure 4.1: General concept of Apache Kafka.

Apache Kafka is a platform designed specifically for event streaming. There are a number of fundamental concepts and core components of Kafka:

- Record: this is the name for message published to Kafka. Each record can have a key, a value, and some metadata. This is also usually referred as event or message.
- Kafka broker: this is the heart of Kafka. A broker is in charge of serving read/write requests and persisting published messages from clients to its disk. Kafka usually runs in cluster with three or more broker. In the current release of Kafka, the cluster also include one or more Zookeeper nodes to maintain metadata of these brokers. However, Zookeeper will soon be removed completely from Kafka in later release and metadata will be maintained natively on Kafka broker instead [40].

- Topic and partition: Records are organized into different topics on Kafka. A topic further comprises of multiple partitions, each of which is an append-only and immutable log. New records will be appended to the end of the log. Each record in a partition is uniquely identified by an incremental offset number. There are two types of partition, namely, leader and follower. Read and write operations will be done on the active leader partition. Follower partitions are replicas of the leader which reside on different brokers in the cluster and cannot serve requests. Since Kafka 2.4, it is possible to read records from follower replicas as well [41]. Nevertheless this feature is disabled by default.
- Kafka clients: There is Producer API which is used to publish records to Kafka topics. On the other hand, Consumer API is used to read records from a Kafka topic.

In this thesis, the current release 2.6.0 of Kafka will be evaluated.

Apache Pulsar

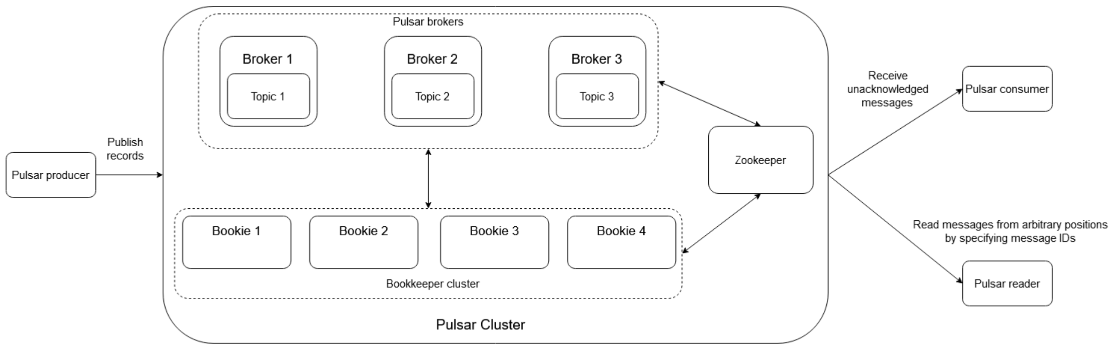

Figure 4.2: General concept of Apache Pulsar.

Apache Pulsar is designed as a multiple-purposed platform by combining the concept of traditional messaging and event streaming. Following are the main concepts of Pulsar:

- Message: a message published to Pulsar has a key-value format along with some metadata.
- Pulsar broker: this component is responsible for serving read/write requests from clients and send persisting request of messages to the persistence layer. There are usually many brokers run together in a cluster.
- Bookkeeper: this is the persistence layer of Pulsar. Bookkeeper handles the durable storage of messages upon receiving requests from the Pulsar broker. Messages are stored on Bookkeeper ledgers which are immutable logs with new records being appended to the end. The Bookkeeper usually runs in a cluster with multiple nodes which are called Bookies.

- A Pulsar cluster is made from a cluster of Pulsar brokers, a cluster of Bookkeeper nodes and a few Zookeeper nodes for metadata management of these clusters.
- Topic and partition: Apache Pulsar organize records into different topics. Each broker is responsible for read/write request of a different subset of topics. A topic can also be split further into multiple partitions. However, a partition is internally also a normal Pulsar topic which is managed transparently to user by Pulsar. Records in a Pulsar partition or a non-partitioned topic are uniquely identified by message IDs.
- Pulsar clients: Messages can be published to Pulsar topic with Pulsar Producer API. For messages consumption, there are two different client APIs, namely, Consumer API and Reader API. Each of these consumption clients has different level of flexibility and can be used in different cases. The detail comparison of Consumer and Reader is given in the evaluation section of messaging patterns.

The current release 2.7.0 of Pulsar will be evaluated.

NATS Streaming

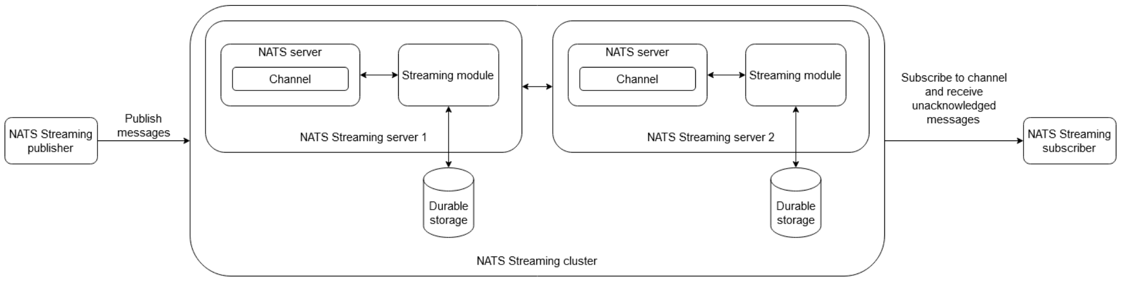

Figure 4.3: General concept of NATS Streaming.

NATS Streaming is an event streaming add-on built on top of NATS server which is a messaging system without persistent layer. It has some core concepts:

- Message: a message published to NATS Streaming contains a value and some metadata.
- NATS Streaming server: A NATS Streaming server comprises of a normal NATS server as the messaging layer and a streaming module to receive and persist message on NATS server to a pluggable durable storage. NATS Streaming comes with an embedded NATS server but can also be configured to work with an existing NATS server. A number of NATS Streaming servers can be grouped together into a cluster in two modes: clustering and fault tolerance. In the first mode, each server retains a full copy of all messages in a separated data store. In the latter mode, nodes in the cluster share a single data store.

- Channel: On NATS Streaming, messages are organized into channels which internally are made from append-only message logs. A message in a channel can uniquely be identified with an incremental sequence number.
- NATS Streaming client: NATS streaming provides client API to publish and subscribe to messages on channels.

In the thesis, the latest version 0.19.0 of NATS Streaming is used for assessment.

Moreover, each platform provides different implementations of its clients in different programming languages. Nevertheless, to have a uniform benchmark, when the evaluation involves comparing clients, the Java implementations will be used since this programming language is officially supported by all three platforms.

4.2 Event storage

Apache Kafka Durable storage - criterion 1.1

In Kafka, records are stored in partitions each of which is stored on the disk as a number of segment files [42]. The broker receives records and writes them sequentially to these files. When a segment file reaches its configurable maximum size, a new file will be created for writing new records. Since all records are only appended to these files, the read/write of records only requires sequential I/O on the disk. This is one of the key design features of Kafka to help maintain fast performance even on cheap hard disks.

However, a Kafka broker can already acknowledge writing requests when records are written to I/O buffer and not necessarily when records are persisted to disk. Therefore, durability is not guaranteed, and message loss can still happen when the broker fails before flushing records to disk. User can force disk flush to ensure durability whenever a message is received but this is not recommended by Kafka since it can reduce the throughput of the system [42]. To achieve durability, a more common approach is combining this unflushed write feature with redundant write on other brokers in the cluster which is the fault-tolerance feature for storage provided by Kafka. This is elaborated in more detail in the next section.

Event storage is fault tolerant - criterion 10.1
As briefly mentioned in the general concept, there are two types of Kafka partitions, namely, leader and follower. The leader partition is active and can serve read/write requests from clients while followers are standby replicas of the leader. For fault tolerance, Kafka supports data replication among brokers in the cluster [43]. For each topic, user can specify a replication factor which determines the number of existing copies of records on Kafka. When replication factor is 2 or more, every partition of the topic will have 1 leader and 1 or more followers. For each partition, each of its copies will be distributed on a different broker in the cluster. Therefore, there will be no single point of failure

for record storage. The replication factor must be equal or smaller than the number of brokers in the Kafka cluster.

However, enabling only data replication on the broker cannot guarantee that all records are safely replicated on the Kafka cluster. By default, a Kafka producer sends a record and only waits for the acknowledgement of successful write from the leader partition. If the broker with the leader partition goes down before the record is flushed to its disk and replicated to other follower replications, the record may be lost without the producer knowing about it for resending. Therefore, the producer must be strictly configured to wait for acknowledgements from the leader partition as well as other follower replicas. In this case, the leader partition will also wait for acknowledgements from its followers before confirming with client. By having the redundant acknowledgements, durability is guaranteed even when messages are not yet persisted to disks given that all brokers retain the partition do not fail simultaneously.

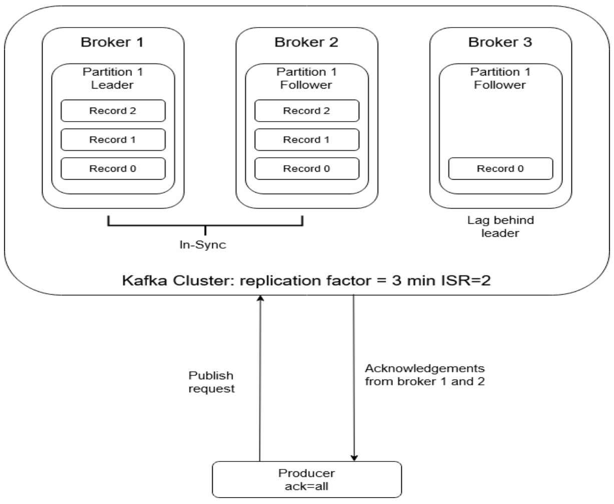

Figure 4.4: Data replication model for fault tolerance of event storage on Kafka.

For each replicated partition, the follower periodically fetch data from leader to stay in-sync. If the producer waits for acknowledgements from all replicas, some followers may fall too far behind the leader for instance due to network connection and increase the waiting time of the producer. Therefore, the leader of the partition dynamically maintains a list of in-sync replicas (ISR) which contains itself and all followers which currently stays synchronized with it. This list is stored on Zookeeper. When a follower does not catch up with the leader by sending fetch request after a configured amount of time, it will be removed from the ISR of the partition. The slow follower can rejoin ISR later when it has fully caught up with the leader partition. In practice, the producer will only wait for acknowledgements from the ISR instead of all replicas. This aims at balancing between the durability, fault-tolerance of published records and the latency for acknowledgment. As a result, a message acknowledged to producer can survive up to ISR-1 failed nodes and still be available to consumer.

It could be possible that all followers of a partition are out-of-sync with the leader. In this case, producer only receives one acknowledgment from the leader which brings back the problem of losing messages. Therefore, Kafka also provides the option to configure the minimum number of in-sync replicas. If the ISR falls below this number, new writing requests will be rejected, and availability is compromised to ensure the durability.

Durability and fault tolerance of data storage on Kafka are closely related to each other and can only be achieved with the right configurations on both Kafka brokers and Kafka producers. In addition, Kafka provides many configuration options to give users the flexibility to choose different priorities for their systems such as availability, durability, latency.

Flexible data retention policy - criterion 1.2
All records published to Kafka will be retained. Old data can be cleaned up with different cleanup policies [42]. These policies can be configured on the broker level which will then be applied to all topics or they can be configured differently for individual topic. There are two basic strategies to retain data:

- Delete: All records are retained for a period of time or based on a maximum size and then they will be deleted.
- Compact: This strategy is only applicable to records with key values. The topic will be compacted. Only the latest record of each key value is retained.

For the first strategy, user can choose different retention policies for a partition based on maximum size or for a segment file of a partition based on retention period. Once the retention limit is exceeded, oldest segment file of the partition will be deleted. By default, there is no limit on the size of a partition and the retention period is 7 days. User can also configure infinite retention period if necessary.

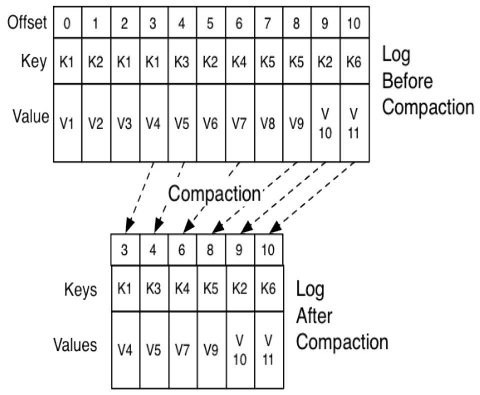

Figure 4.5: Log compaction on Kafka (taken from Kafka documentation [42]).

For the second strategy, the background cleaner thread of Kafka will regularly scan the segment files and keep only the latest record for each key value (Figure 4.5). Therefore, this only works with records with non-empty key value. Writing request of record without a key to a topic configured with compact cleanup policy will be rejected.

The two strategies can also be combined. With this setup, topics will be compacted regularly but when the retention maximum size or maximum retention period is reached, the data will also be removed regardless of being compacted or not. For example, in case of an online shopping application where each order is kept track by a sequence of events published to Kafka as records, when users are interested in the latest status of an order but only if the order is not older than 3 months, it is reasonable to this combination of retention strategies.

To sum up, Kafka provides a flexible way to retain all records or only selective data. Users can choose the appropriate strategy based on their use cases.

Acronyms

CQRS	Command Query Responsibility Segregation.
ESP	Event Stream Processing.

List of Figures

List of Tables

List of Listings

Bibliography

[1] "Evolution of the netflix data pipeline." Netflix Technology Blog: `https://netflixtechblog.com/evolution-of-the-netflix-data-pipeline-da246ca36905`. Accessed: 2020-11-14.

[2] S. Intorruk and T. Numnonda, "A comparative study on performance and resource utilization of real-time distributed messaging systems for big data," in *2019 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pp. 102–107, IEEE, 2019.

[3] C. Bartholomew, "Performance comparison between apache pulsar and kafka: Latency." `https://medium.com/swlh/performance-comparison-between-apache-pulsar-and-kafka-latency-79fb0367f407`. Accessed: 2020-11-12.

[4] T. Treat, "Benchmarking nats streaming and apache kafka." `https://dzone.com/articles/benchmarking-nats-streaming-and-apache-kafka`. Accessed: 2020-11-12.

[5] "Kafka vs. pulsar vs. rabbitmq: Performance, architecture, and features compared." `https://www.confluent.io/kafka-vs-pulsar/`. Accessed: 2020-11-12.

[6] V. C. Alok Nikhil, "Benchmarking apache kafka, apache pulsar, and rabbitmq: Which is the fastest?." `https://www.confluent.de/blog/kafka-fastest-messaging-system/`. Accessed: 2020-11-12.

[7] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream data processing systems," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 1507–1518, IEEE, 2018.

[8] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan, "A survey of distributed data stream processing frameworks," *IEEE Access*, vol. 7, pp. 154300–154316, 2019.

[9] M. A. Lopez, A. G. P. Lobato, and O. C. M. Duarte, "A performance comparison of open-source stream processing platforms," in *2016 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, IEEE, 2016.

[10] C. Richardson, "Pattern: Monolithic architecture." `https://microservices.io/patterns/monolithic.html`. Accessed: 2020-11-19.

[11] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pp. 000149–000154, IEEE, 2018.

[12] M. Fowler, "Monolithfirst." `https://martinfowler.com/bliki/MonolithFirst.html`. Accessed: 2020-11-18.

[13] J. Lewis and M. Fowler, "Microservices." `https://martinfowler.com/articles/microservices.html`. Accessed: 2020-11-18.

[14] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, *et al.*, "Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 179–182, IEEE, 2016.

[15] B. Stopford, *Designing Event-Driven Systems Concepts and Patterns for Streaming Services with Apache Kafka*, ch. 5, pp. 31–38. O'Reilly Media, Inc., 2018.

[16] G. Young, "Cqrs documents." `https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf`, 2010. Accessed: 2020-11-23.

[17] M. Fowler, "What do you mean by "event-driven"?." `https://martinfowler.com/articles/201701-event-driven.html`. Accessed: 2020-11-19.

[18] B. Stopford, *Designing Event-Driven Systems Concepts and Patterns for Streaming Services with Apache Kafka*, ch. 8, pp. 79–81. O'Reilly Media, Inc., 2018.

[19] C. Richardson, "Pattern: Event-driven architecture." `https://microservices.io/patterns/data/event-driven-architecture.html`. Accessed: 2020-11-22.

[20] M. Fowler, "Event sourcing." `https://martinfowler.com/eaaDev/EventSourcing.html`. Accessed: 2020-11-23.

[21] D. Betts, J. Dominguez, G. Melnik, F. Simonazzi, and M. Subramanian, *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure*, ch. Reference 3: Introducing Event Sourcing. Microsoft patterns & practices, 2013.

[22] H. Rocha, "What they don't tell you about event sourcing." `https://medium.com/@hugo.oliveira.rocha/what-they-dont-tell-you-about-event-sourcing-6afc23c69e9a`, 2018. Accessed: 2020-11-23.

[23] C. Kiehl, "Don't let the internet dupe you, event sourcing is hard." `https://chriskiehl.com/article/event-sourcing-is-hard`, 2019. Accessed: 2020-11-23.

[24] T. Akidau, "Streaming 101: The world beyond batch." `https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101/`. Accessed: 2020-11-17.

[25] N. Marz, "How to beat the cap theorem." http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html. Accessed: 2020-11-18.

[26] J. Kreps, "Questioning the lambda architecture." https://www.oreilly.com/radar/questioning-the-lambda-architecture/. Accessed: 2020-11-18.

[27] J. Kreps, "Putting apache kafka to use: A practical guide to building an event streaming platform (part 1)." https://www.confluent.io/blog/event-streaming-platform-1/. Accessed: 2020-11-18.

[28] J. Kreps, "The log: What every software engineer should know about real-time data's unifying abstraction." https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying. Accessed: 2020-11-18.

[29] M. Kleppmann, *Making Sense of Stream Processing: The Philosophy Behind Apache Kafka and Scalable Stream Data Platforms*, ch. 5, pp. 160–165. O'Reilly Media, 2016.

[30] https://kafka.apache.org/. Accessed: 2020-11-19.

[31] https://pulsar.apache.org/. Accessed: 2020-11-19.

[32] https://rocketmq.apache.org/. Accessed: 2020-11-19.

[33] https://docs.nats.io/nats-streaming-concepts/intro. Accessed: 2020-11-19.

[34] https://pravega.io/. Accessed: 2020-11-19.

[35] "Iso/iec 25010." https://iso25000.com/index.php/en/iso-25000-standards/iso-25010. Accessed: 2020-11-22.

[36] G. Hohpe and B. Woolf, "Enterprise integration patterns: Messaging patterns." https://www.enterpriseintegrationpatterns.com/patterns/messaging/Messaging.html. Accessed: 2020-11-23.

[37] G. Hohpe and B. Woolf, "Enterprise integration patterns: Messaging patterns: Publish-subscribe channel." https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html. Accessed: 2020-11-23.

[38] G. Hohpe and B. Woolf, "Enterprise integration patterns: Messaging patterns: Content-based router." https://www.enterpriseintegrationpatterns.com/patterns/messaging/ContentBasedRouter.html. Accessed: 2020-11-23.

[39] T. Treat, "You cannot have exactly-once delivery." https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/. Accessed: 2020-11-24.

[40] "Kip-500: Replace zookeeper with a self-managed metadata quorum." https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum. Accessed: 2020-11-26.

[41] "Kip-392: Allow consumers to fetch from closest replica skip to end of metadata." https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica. Accessed: 2020-11-26.

[42] "Kafka documentation." https://kafka.apache.org/documentation/. Accessed: 2020-11-26.

[43] "Kafka replication." https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Replication. Accessed: 2020-11-27.