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
2nd Supervisor:	Prof. Dr. Eicke Godehardt

November 22, 2020.

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 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 and the form of the feature matrix. 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 other to include it in the control flow. For the latter approach, services communicate with each other using events.

Event
An event 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 where 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 without the need to modify other services. There are three common ways to use events, namely, event notification, event-carried state transfer and event sourcing [16].

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.

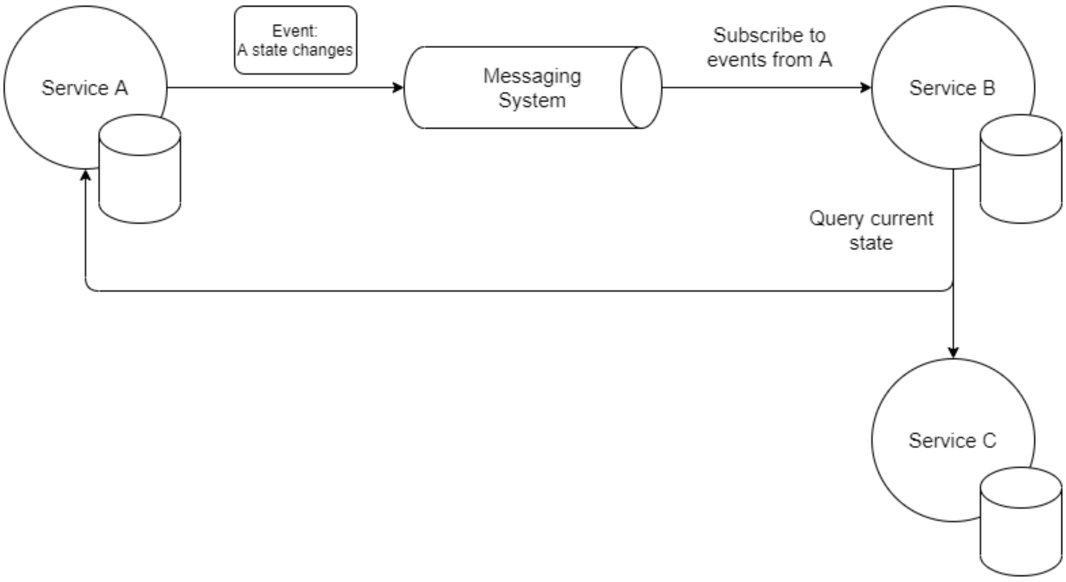

Figure 2.1: Services coordination with event notification.

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 [17]. Therefore, the two following patterns tackle this problem by actively 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 the new value. 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.

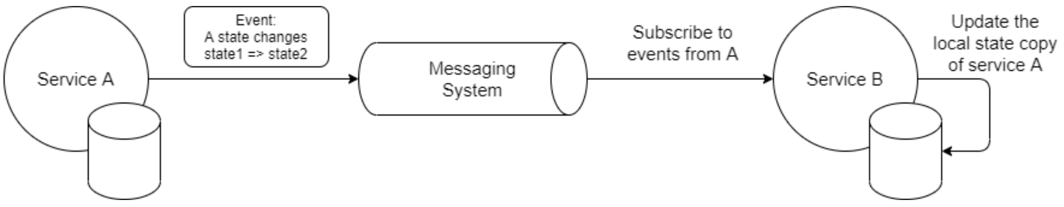

Figure 2.2: Services coordination with event-carried state transfer.

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 [18].

Event Sourcing and Command Query Responsibility Segregation (CQRS)

2.2 Event Stream Processing

With the increasing amount of data, the demand about how data is processed and analyzed also evolves over time. In the early day, data is usually collected over a 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 the organization. However, this type of data processing gradually cannot keep up with the need of faster analysis allowing companies to response more timely to change. 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. 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 an uncountable number of 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 [19]. 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 [20]. As stream processing engines grow more mature and accurate, the demand for such system is lessened [21].

2.2.1 Stream Processing and Event-Driven Architecture

Moreover, stream processing is not merely a data processing paradigm to achieve low latency result. Applying the concept of streaming on the architectural level in event-driven systems also has the potential to help build more scalable and resilient applications. The problem of coupling emerges not only between services of applications but also between data systems in general. A sophisticated software system can comprise of multiple data systems with different functionalities such as relational database, document store, cache, monitoring system, data warehouse. Data needs to be shared and synchronized between these components in an efficient way. This can quickly turns the entire system into a big tangled mesh. As an example, this problem was experienced at LinkedIn as their system became increasingly complex [22].

This problem can be tackled by using stream processing in event-driven systems. In such systems, every operation, state change or any information can be recorded in form of an event. Events from all services are gathered and written orderly to an append-only log called event stream. On the contrary to the previous approach where data is encapsulated in each service, this stream is used as the single source of truth shared by all services. Any data service in the system can consume and

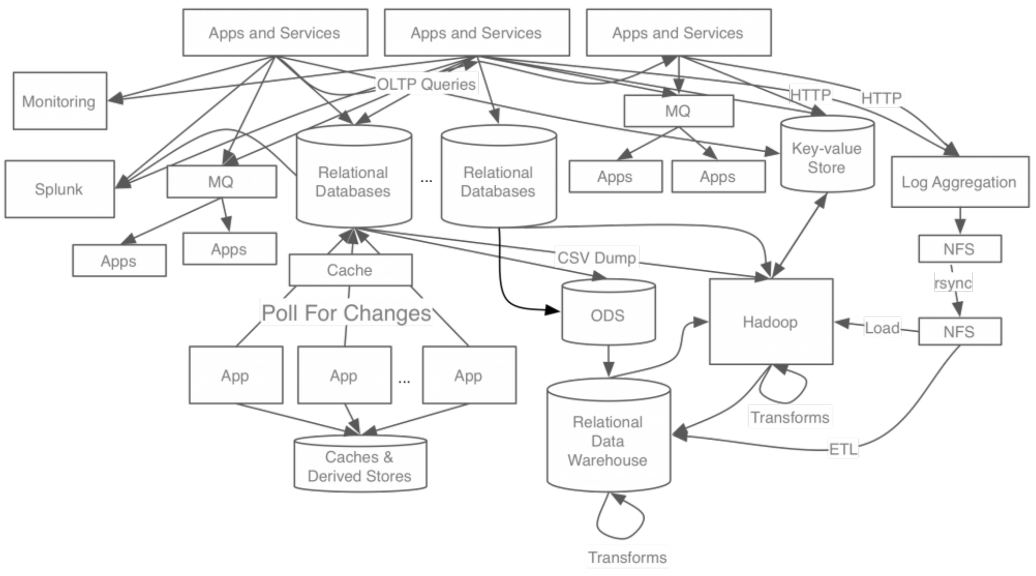

Figure 2.3: The tangled data systems at LinkedIn in the old day [22].

process this log of raw events using stream processing to generate a local replication of the current system state. This log-centric design fits seamlessly into a distributed environment with numerous moving components to allow data to be replicated among services with minimal coupling [23]. Moreover, services also have the flexibility to derive different data structures from the raw events to match their specific access pattern. The task of data representation is now done at individual data service instead of in the central data store. This is referred by Martin Kleppmann as turning the database inside out [24]. The result is a more neatly organized system with every services synchronize and communicate via the event streams.

2.3 Event Stream Processing Platform

An ESP platform must facilitate the construction of software systems revolving around streams of events. Therefore, it must have a number of fundamental capabilities. Firstly, it must provide the mechanism for events storage. 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 the entire system depends on. Accessing interface must be provided for applications to publish and consume events. The platform should also enable the processing on the events streams either by providing a native stream processing tool or integrating with external stream processing framework. Moreover, the platform should come with ready-to-be-used tools to integrate with a wide range of existing data systems effortlessly including also legacy

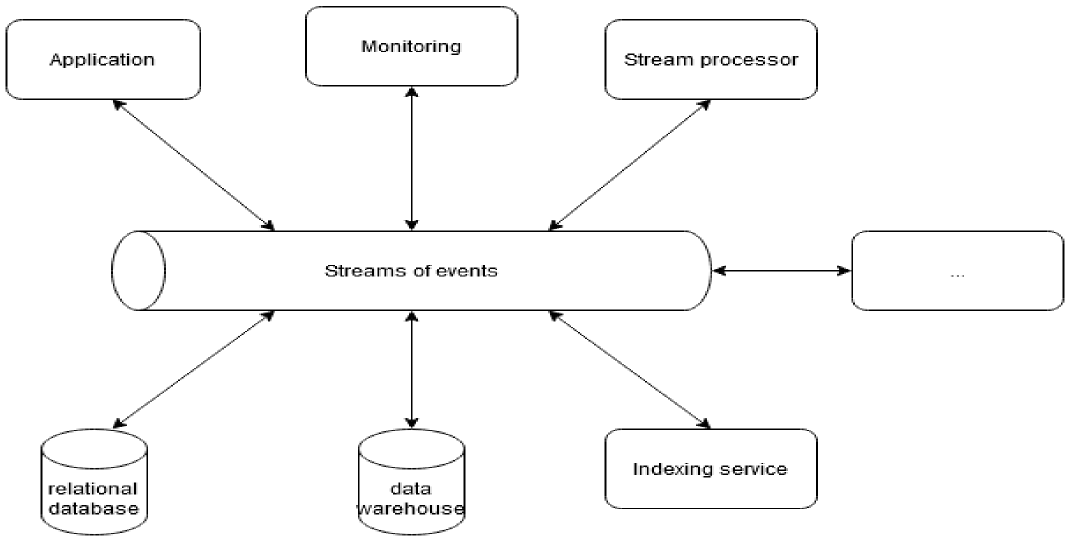

Figure 2.4: System with streams of events as the single source of truth.

systems.

The order of events must be preserved by the platform throughout their entire life cycle: in storage, transit and under process. 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 system. Finally, the platform must have a rich set of utility tools for monitoring and management.

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

[17] 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.

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

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

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

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

[22] 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.

[23] 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.

[24] 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.