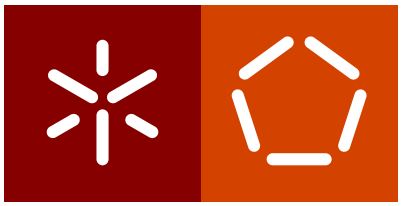


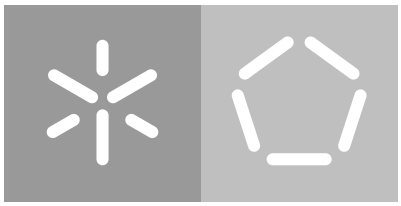
Universidade do Minho
Escola de Engenharia
Departamento de Informática

Rui Fernando Carvas dos Santos

Microservices architectures in healthcare with Apache Kafka

Relatório de Pré-Dissertação

Fevereiro 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Rui Fernando Carvas dos Santos

Microservices architectures in healthcare with Apache Kafka

Relatório de Pré-Dissertação

Dissertação de Mestrado
Mestrado em Engenharia Informática

Dissertação realizada sob a orientação do Professor
António Carlos da Silva Abelha
Hugo Daniel Abreu Peixoto

Fevereiro 2021

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

API Application Programming Interface i, ii, 2, 8, 10, 15, 16
App Application 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 15, 16, 17

CD Continuous Delivery 17
CI Continuous Integration 17
CQRS Command Query Responsibility Segregation i, ii, 16, 17
CRUD Create, Read, Update and Delete 16

ESB Enterprise Service Bus 7, 9

gRPC Google Remote Procedure Calls 13

IP Internet Protocol 15
IPC Inter-Process Communication i, 13, 14

JS JavaScript 14
JSON JavaScript Object Notation 19

REST Representational State Transfer 13

SOA Service-Oriented Architecture ii, 5, 7, 8, 9
SOAP Simple Object Access Protocol 7

UI User Interface 6, 10, 11

VM Virtual Machine 18

WS Web Service 7

1

INTRODUCTION

The evolution of technology in recent years combined with greater demand on the part of users both in the launch of Software products and with the introduction of new updates to correct possible errors/flaws, and the incorporation of new features has made companies and institutions reflect the way they develop their Application (App)s. In addition to the requirement from users for continuous innovation, there is currently an enormous amount of requests made to the back-end services, causing data failures and momentarily overloads, thus possibly losing vital information. [1]

One of the methods found by companies and institutions to solve the problems described above, was to change their systems based on monolithic architectures to systems based on microservices architectures, which has been gaining enormous popularity in scientific articles, blogs and conferences. [2] An architecture based on microservices, is based on a distributed solution where an application is divided into smaller services that communicate with each other through Application Programming Interface (API)'s. Each service can be developed, implemented and scaled completely independently without directly affecting other services in the application, thus making the application as uncoupled as possible, therefore preventing a single failure from taking the system offline. [2]

In microservices architectures, there are several problems that can become complex, namely the issue of communication between the services of an application. There are several forms of communication, however, a bad approach can cause an application based on microservices to become a distributed monolithic solution, not enjoying the advantages of said architecture. Of all forms of communication, there is one that stxtands out, Messaging, which is based on messages exchanged through a Publisher and a Subscriber through a Message Broker. [23]

Apache Kafka is one of the most widely used Message Brokers. Around 35% of Fortune 500 Companies employ this platform. Initially developed by Linkedin, Apache Kafka is an open-source platform for event streaming. It allows for communications in latencies below 2ms, which is tolerant to more diverse failures that can occur during the communication and that can be scaled horizontally. Apache Kafka has evolved and today has a complete ecosystem, which can be used to assist in the most diverse areas, such as Apache Kafka Streams and Apache Kafka Connect. [9]

Currently, with the Covid-19 pandemic induced stress in the healthcare system, plus the already existing issues, the need for modernization and update is crucial, in order to be able

to deal with the most diverse of problems, as well as assisting healthcare professionals in decision-making.

In this sense, this dissertation aims to study and develop an application using an architecture based on microservices using a Message Broker, namely Apache Kafka. The goal is to understand the advantages and disadvantages of the platform, and whether it may be an adequate fit for healthcare systems.

1.1 MOTIVATION

Locally and globally microservice architectures have been adopted by large companies, such as Netflix, Amazon and Uber. The investment of these companies in this type of architecture has translated to an increased focus and attention in Microservices by the software architecture community and acknowledge writers in the area such as Martin Fowler[1] and Chris Richardson[2].

That being said, the motivation for this dissertation comes from understanding this innovative software architecture and how it can best be applied to areas of need, such as the healthcare system.

1.2 MAIN GOALS

It is proposed, with this dissertation, to address the following objectives:

- Study of the Microservices architectural style;
- Study of the Apache Kafka Ecosystem;
- Design and plan an App based on Microservices using Apache Kafka;
- Treatment of App results like fail tolerance and maintainability.

1.3 PRE DISSERTATION STRUCTURE

This pre dissertation is divided into chapters in order to facilitate its reading. These are:

1. Introduction

   In this chapter an introduction to the dissertation is made, addressing its context and motivation,main goals, as well as an explanation of the structure of the pre dissertation.

2. State of the Art

   This Chapter presents the state of the art that will involve the entire dissertation, addressing the main concepts and approaches used in the topic in question.

[1] https://martinfowler.com/aboutMe.html
[2] https://www.chrisrichardson.net/about.html

3. Methodology Here, the various methods of analysis, documentation and research, as well as guidance meetings, employed in writing this dissertation are described, as well as the related Gantt Diagram.

2

STATE OF THE ART

Throughout this section, the main concepts of what is a microservices based architecture will be presented, alongside different factors such as the types of communication between services; the main characteristics associated with a microservice based App; and the different ways to do the deployment. Furthermore, Apache Kafka will be presented, and its main concepts.

2.1 MICROSERVICES ARCHITECTURE

For a better understanding of this type of architecture, as well as the intrinsic characteristics, that will be exposed in the document, it is relevant to explain two other different types of styles. One, monolithic, more traditional, and Service-Oriented Architecture (SOA), a style often referred to as the predecessor of microservices.

2.1.1 *Monolithic Architecture*

The main programming languages used in the development of modern App's such as Python and Java, provide abstractions in order to divide the project in modules, facilitating the developers. However, these same languages are projected for the creation of single executable, commonly designed as monolithics. These are characterized for sharing all the modules in the same machine that is running the App, turning the App extremely coupled. [6]

Using a monolithic architecture in the development of apps can be beneficial, mainly due to the following advantages: [14]

- Easy and fast initial development of an App;
- Single location for management of resources;
- Universal programming language (no need for additional programming skills);
- Simple communication between the various modules;
- Easy deploy.

Nonetheless these advantages can in reality turn to disadvantages with time. New functionalities and updates add a set of obstacles to this architecture, as such: [6]

- Extensive Source-Code are increasing harder to maintain;
- A single failure point;
  - If there is a problem that hasn't been detected in the compilation or realization of tests, the app can turn offline.
- A single alteration implies an restart of the entire monolithic, therefore increasing the time of development/tests;
- Complete deployment in each update of the source-code, even in modules that have not suffered any alterations;
- Limited scalability of the App;
  - Usually to solve the overload of the App, a new instance of the same App is created, dividing the load into two parts. However, this excess of load can occur in a specific part of the App and not in its entirety, causing, nonetheless, the replication of the App in its entirety.
- Using a single technology can be seen as a limitation from a development standpoint, when, at times, the best technology for a certain requirement is not used.

In figure 1 one example of how an App can be developed using a monolithic architecture is presented, where a single executable, contain the layer of business logic, User Interface (UI) and data interface. In addition, the data is all stored in the same database.

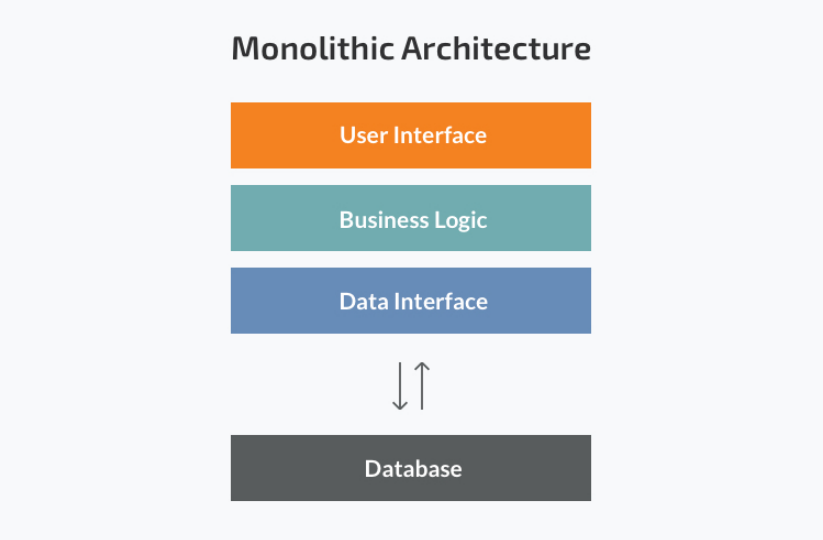


Figure 1: Monolithic Architecture. Source: [12]

2.1.2 *Service-Oriented Architecture*

SOA is an architecture style based on a distributed system that allows the construction of complex apps through services that communicate between each other via a mutual language. Typically, Simple Object Access Protocol (SOAP) or Web Service (WS) is used. [7]

A service, in this software architecture, is an independent unit that is projected to conclude a complete commercial function, having at its disposal all the data needed to do so. [7]

The employment of Enterprise Service Bus (ESB) is fairly typical in SOA. It allows for integration between the several services of an App. Furthermore, it is commonly used one or more databases shared amongst the several services of the App. [13]

Some of the main advantages of this architecture are: [7]

- Recycling of services;
- Easy maintenance;
- Better availability and scalability;
- Increased productivity;
- Platform independence.

Despite the previously shared advantages, this architecture presents some disadvantages, such as: [30]

- Single failure point;
- Shared databases increase coupling of App's;
- Services can become large monolithics.

In figure 2 an example of how an App can be developed using a SOA is presented.

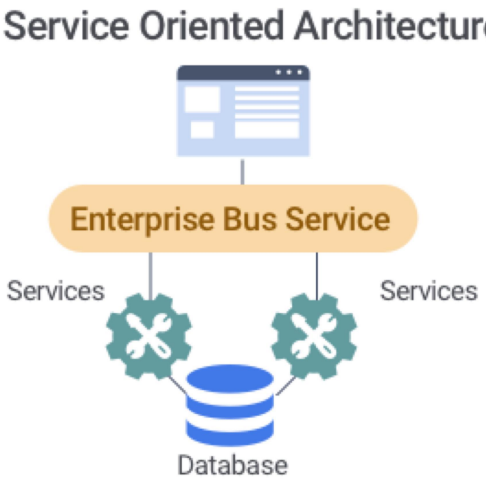


Figure 2: Service-Oriented Architecture (SOA). Source: [31]

2.1.3 *Microservices Architecture*

The term microservices, that ultimately has gained traction from companies and organizations such as Netflix, Amazon and UBER, etc, and the scientific community, is, in fact, related to two key definitions presented below. [2]

Definition 1 (Microservice) - *"A microservice is a minimal independent process interacting via messages."* [6]

Definition 2 (Microserive Architecture) - *"A microservice architecture is a distributed application where all its modules are microservices."* [6]

Having as a basis the previous definitions is possible to comprehend that a microservices based architecture is the decomposition of an entire App by basic functions, where each function is named microservice and can be created and run in a totally independent manner.

Making each microservice autonomous, allows for the same to be developed, implemented, run and scaled without affecting the functioning of the other services of the App. Moreover there isn't an excess of coupling, since the other services do not share any type of codes amongst themselves. The communications through API's are well defined.

Another important fact is that microservices are specialized, meaning each service is projected for a specific functionality dedicated to the solution of a problem. In case there is a need to add new functionalities to a specific service increasing its complexity, the same can be after divided into smaller services allowing for a horizontal scaling.

In figure 6, the example of how an App can be developed using a microservice architecture is presented.

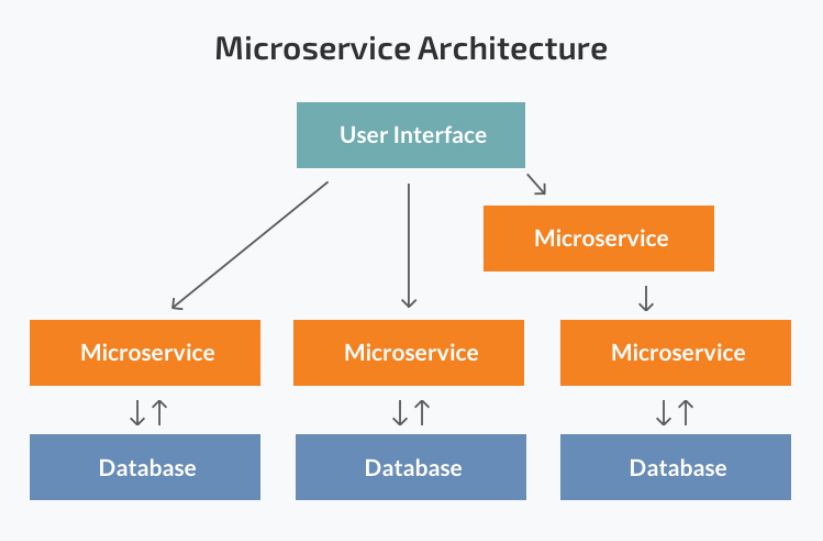


Figure 3: Microservices Architecture. Source: [12]

Choreography vs. Orchestration

Associado ao termo microservices existe sempre a discução do tipo de lógica de coordenação que deve ser utilizado, nomeadamente a questão da Choreography e a questão da Orchestration.

Orchestration

Orchestration has the logic of coordination the communication of the App in a single service(Orchestrator). An orchestrator sends messages to other services stating which operations must perform and for who. This technique is common in SOA with the use of ESB. [27]

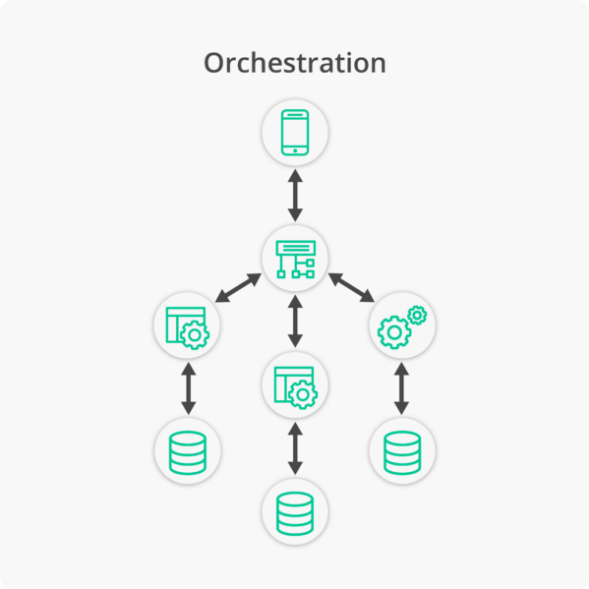


Figure 4: Microservices Orchestration. Source: [27]

Choreography

Choreography is a saga for the composition of services in a distributed and decentralized way, where under a global perspective of the application, there is no coordinating service. Therefore, each application service knows how to collaborate with the other services. For this purpose it is common to use an event broker. [27]

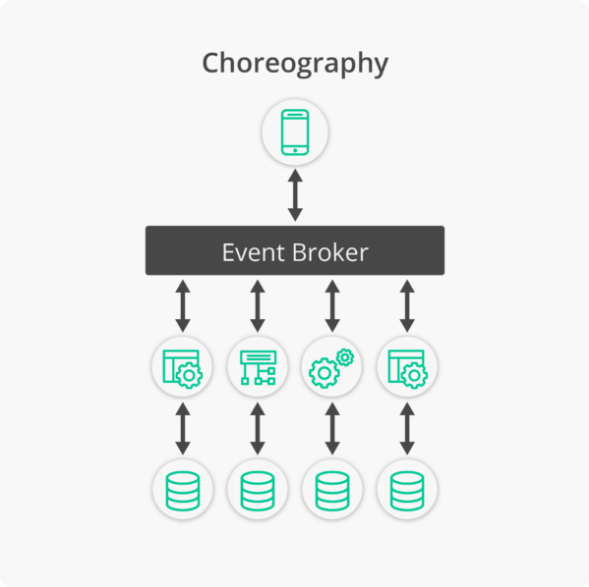


Figure 5: Microservices Choreography. Source: [27]

Characteristics of a Microservice Architecture

There is a set of characteristics that are directly linked with microservices architecture, even though not all App's that employ this architecture share all the characteristics detailed below. [22]

Componentization via Services

One of the main characteristics of microservices architecture is the componentization via services, meaning, the possibility to create a complete app composed by microservices ,totally independent, that communicate amongst each other through well defined API's. These services are divided in such a way that each of them perform a functionality, and at the same time, cooperating to run a complete App. [22]

Organized around Business Capabilities

During the construction of an app, it is usual to separate the app in different development parts. The most common are: User Interface (UI) team; server-side logic team; and the database team, meaning, the architecture of the App organized around the technological resources. [15]

In a microservices based approach, the division must be made around the business. A development team should be able to alone be responsible for one or more products, through all stages of development (UI, Server-Side Logic, Database, amongst others). Each team member should have a specific ability, managing to supply a finished product. The development team needs to be multidisciplinary. [15]

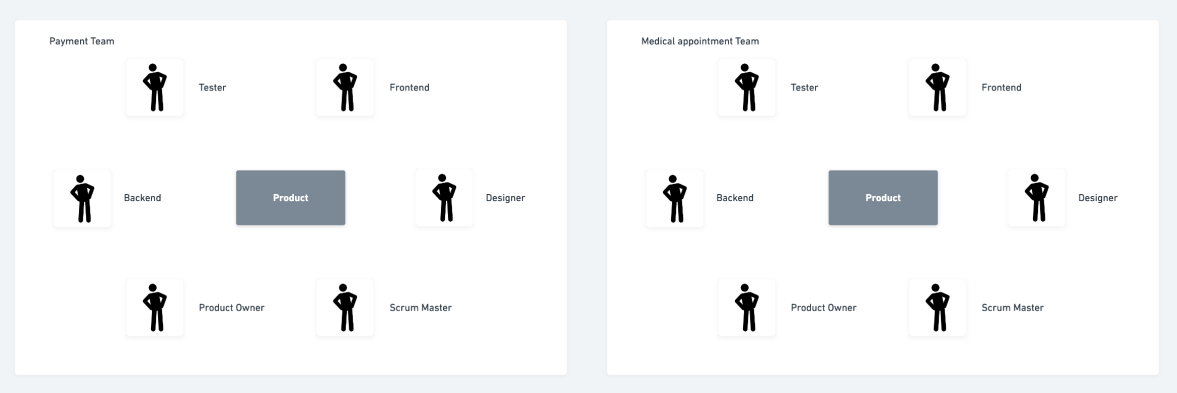


Figure 6: Organized around Business Capabilities

Products not Projects

It is common in the development of an App, to employ a project model. The main goal of the team is to deliver an operating software that meets the required demands. After this stage, the software is managed by a team in charge of its maintenance. [22]

In a microservice based approach, the same tends to not happen. Following Amazon's slogan "You build, you run it", a team is in charge of a product during its development and operating process, remaining in permanent contact with the product to improve it for the final clients. [22]

Smart Endpoints and Dumb Pipes

The communication between services of a microservice based App should be simple, quick and uncomplicated. To do so this architecture's communication is inspired in the UNIX classic system. The goal is: receive a requisition; process it; and respond. Simple synchronous or asynchronous protocols such as Request/Response and Publish/Subscribe are employed. [15]

By using this communication methods, in which the containers that send and receive messages are smart and the pipes are dumb, it allows for a decentralization of the architecture.

Decentralized Governance

Another main characteristic of this architecture is the division of the App in small services. By doing this division there is no longer the need to restrict technologies that happens in other architectures such as monolithic architectures, where all logic is implemented in a single programming language. [22]

With this freedom of choice, the development teams can then adapt to each service the appropriate technology for the functionality.

Decentralized Data Management

Each microservice should have its own database, in opposition to a centralized database that is used in monolithic architectures. [15]

The use of a database per service, makes it so that there are no shared tables. Consequently it increases cohesion and diminishes coupling, since alterations in the databases will not affect the data model of other services. [15]

Furthermore, it allows to use the type of storage best indicated for the service. E.g., relational databases, Document Store Database, Graph Database, among others. [15]

Infrastructure Automation

When working with microservices architecture, the process should be agile and quick, including the DevOps parts. Since there are many microservices in an App, the manual deployment of each one can be arduous. [22]

To make the process more agile and quick at the DevOps level, it is common to employ varied techniques such as: [22]

- Cloud Computing;
- Automated testings;
- Continuous Integration;
- Continuous Delivery;
- Load Balancer / Auto Scaling.

Design for failure

Since an App is composed of several services, the App should be prepared for eventual failures in one or more services, therefore augmenting potential clients confidence. For such, secondary plans must exist to come into action when one or more services fail. After the occurrence of fails, the same should be quickly detected, and if possible, the service should restart automatically. To allow for a quick detection and correction, the use of logging systems and real time monitoring is common. [15]

The characteristics mentioned above are only possible if the organization is composed by more disciplined and qualified teams. In addition, a very coherent company organization is important.

2.2 Inter-Process Communication (IPC)

Since microservices architecture is a distributed system, one of the focal points is precisely the communication between the various services. In monolithic services, the communication is made through call functions between modules. The same isn't possible for microservices.

Microservices need to communicate amongst each other through a mechanism of communication between processes (IPC). There are several approaches that can be used for the interaction between microservices. They can be classified resorting to a two dimensions table. [2]

Table 1: "Inter-Process Communication Styles"[2]

	One to One	**One to Many**
Synchronous	Request/Response	-
Asynchronous	Request/Async Response	Publish/Subscribe

One of the dimensions refers to the number of destination services that should receive a solicitation.

- One to One;
  - Each client request is processed by exactly one service instance.
- One to Many.
  - Each request is processed by multiple service instances.

The other dimension is related to the answer time of the solicitation.

- Synchronous;
  - The service that made a request blocks waiting for a response.
- Asynchronous.
  - The service that made a request does not block waiting for a response.

2.2.1 *Synchronous Communication*

As previously described, synchronous communication has as a goal to assure that the service that made the request is blocked until the service to which it was sent, answers. To do so, and as demonstrated in the Table 1, there is a way to employ a synchronous communication:

Request/Response

A service makes a request to another service, blocks, and waits for a possible answer. An example of a Request/Response synchronous request is Representational State Transfer (REST) or Google Remote Procedure Calls (gRPC), when done synchronously. [2]

Figure 7: Request/Response Communication

2.2.2 *Asynchronous Communication*

On the contrary of what is verified in a synchronous communication, a asynchronous communication implies that the service that made the request, doesn't block whilst it waits for an answer. It may not even bother if the operation was or not concluded.

Request/Async Response

A service makes a request to another service that will answer asynchronously. The service that makes the request does not block whilst it waits for the answer, and will receive the answer later on. Example: JavaScript (JS) Callbacks. [2]

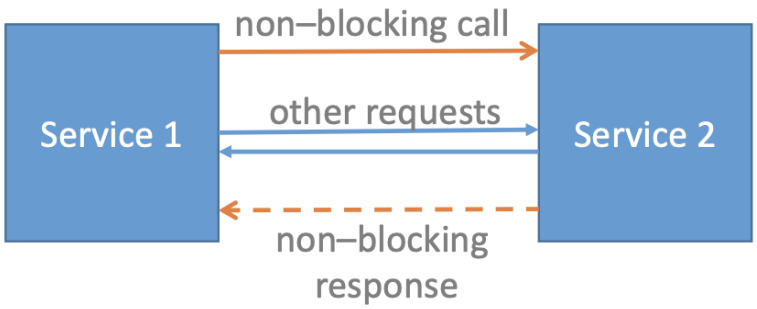


Figure 8: Request/Async Response Communication

Event based

A service publishes a message that will be consumed by one or more services. An example of this type of communication is RabbitMQ and Apache Kafka. The later one will be discussed in the second section of this dissertation. [2]

As seen in figure 9, when Microservice A wants to send information to two other microservices, namely B and C, it publishes a message in a Broker. Later it will then be consumed by Microservice B and C. In this way, it is possible to guarantee that, even if microservice B and C are offline, the message will not be lost, being consumed when the microservices are back online.

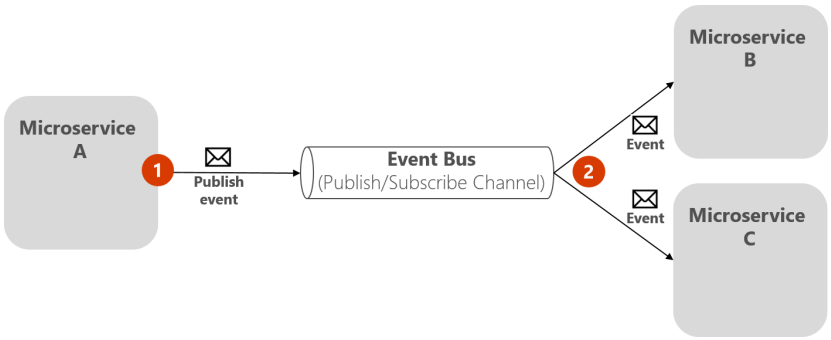


Figure 9: Event based Communication. Source: [5]

2.3 PATTERN'S

2.3.1 *API Gateway*

A microservices based architecture is a distributed system composed of several services. Each one possesses its own API and Internet Protocol (IP) address. For an user interface to be able to present all the information, it needs to resort to several services, in which case it needs to know all the IP addresses. [3] Figure 10 represents this scenario:

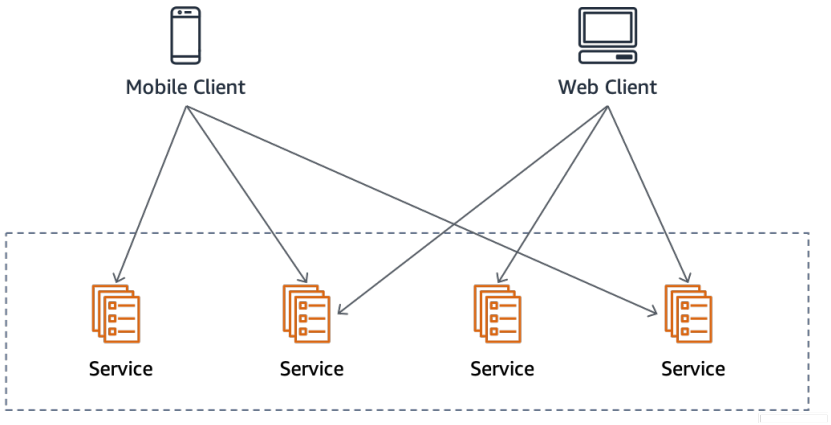


Figure 10: Microservices without API Gateway. Source: [3]

To simplify this process the use of an API gateway is fairly common. This is a middle layer through which the user interface can interact with the services of an App, through calls to only one service - in this case, API Gateway. Furthermore, it can supply an added level of granularity for the various types of clients, such as mobile and web. [3]

The various services of an App are totally independent, Each has its own authentication logic, authorization, among others. By using an API Gateway, all the processes can be shared in the Gateway. [3]

Figure 11 represents the benefits of using the API gateway pattern.

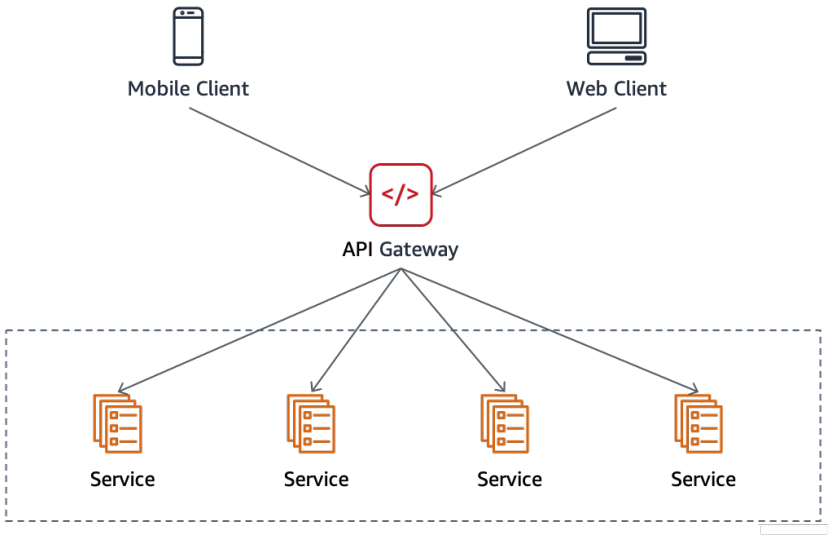


Figure 11: Microservices with API Gateway. Source: [3]

2.3.2 *Command Query Responsibility Segregation (CQRS)*

By far, the most common approach in the development of apps is the use of the Create, Read, Update and Delete (CRUD). All the data manipulations are performed in the same process, using the same database. As the needs become more sophisticated - adding logic to the CRUD's methods - CRUD starts to deter from its original simplicity, possibly slowing the system when used on a large scale. [11]

To solve the problem presented above, the CQRS was created. Its main goal is to separate the parts and responsibilities of writing and reading in the system, in such a way that the system is more efficient. To do so very restricted limits are imposed on the App. [11]

The CQRS pattern is directed to support a great number of users, and to deal with this affluence in such a way that a system can continue to grow without an overwhelming increase in complexity and deteriorating the system. [4]

CQRS is typically used in conjunction with another pattern, Event Sourcing. Figure 12 represents how the two work together, one App is in charge of the reading part and another the writing part. When a user makes a change, a message is sent to the writing service that will update the database. After that, it will write a message asynchronously, that later on will be consumed by the reading service that will synchronize the database with the writing service. When another user needs an information, it will then request it to the reading service, making the processor faster and not overloading the writing database. [29]

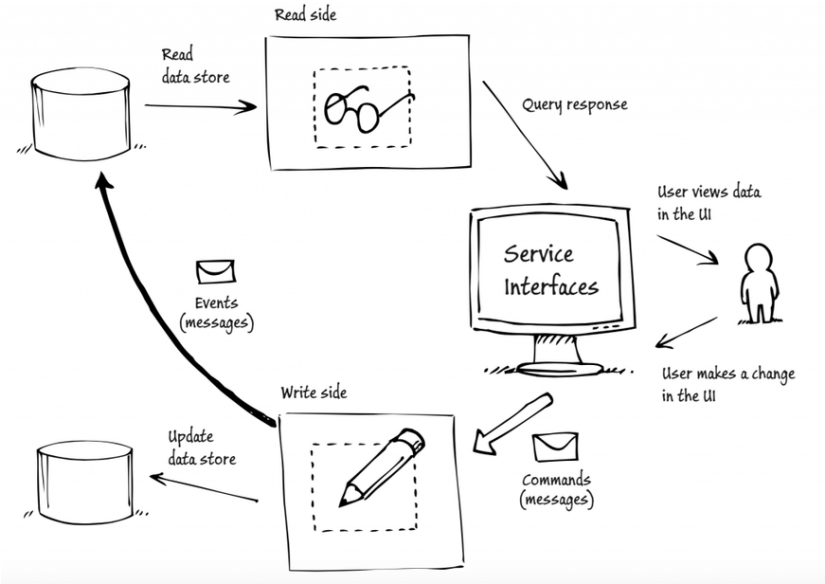


Figure 12: CQRS - Overview. Source: [29]

Another important aspect of the CQRS pattern is that it can use relational databases for the writing part, and NoSQL databases for the reading part. It increases the simplicity of the search and the quickness in reading. [4]

Despite the advantages of CQRS, it can not always be resorted to, since it adds an additional layer of complexity to the system. Furthermore, its procedure implies, that databases can become assynchronized at times. [4]

2.4 DEPLOYMENT

Associated with the deployment process of a microservice architecture based App, there is a set of good practices that should be implemented, such as Continuous integration and Continuous delivery.

Continuous Integration (CI) is the process of integrating source-code modifications in a continuous automatic way, in a single software project. One of the check ups that can be performed is the execution of tests and the verification of the quality of the code, thus preventing human error of validation, and guaranteeing more agility and safety in the process of software development. Currently, CI is a valuable practice, of high performance, and well established in modern organizations. [25]

Another good practice in the DevOps process is the Continuous Delivery (CD). The development teams produce a valid software in short cycles. Allows to guarantee that the software can be continuously deployed in a trusted way. [24]

According to Chris Richardson, there are four types of patterns associated with the deployment of a microservices based App. Each present its own advantages and disadvantages:

- Multiple service instances per host;
- Service instance per host;
- Service instance per Virtual Machine (VM);
- Service instance per Container.

2.5 APACHE KAFKA

Apache Kafka is an open-source platform of distributed transmission of events, developed in the programming languages Java and Scala. Initially developed by Linkedin, with the goal of processing 1.4 trillion message a day, it was launched in January 2011. Currently, it is kept and explored by the company Confluent, under the stewardship of the Apache foundation. [9]

Of the many characteristics of the Apache Kafka, four can be highlighted: **High Performance**, since the delivery of messages has a latency of around 2 milliseconds; **Horizontal Scalability**, it is possible to expand up to hundreds of brokers and process millions of messages by the second; **Failure tolerant**, thanks to its distributed system, in which messages are duplicated and stored in different locations; and **High availability**, since it is possible to have clusters in several parts of the world. [9]

Apache Kafka supports different types of use cases categories, such as: the *publish-subscribe message system*; *Website Activity Tracking; Gather Metrics from many different locations; Log Aggregation; Stream Processing; Event Sourcing; and Commit Log.* [9]

Due to the previously mentioned characteristics, Apache Kafka is present in the most diverse business areas such as Banks, hospitals and telecommunications. Currently, it is employed by more than 2000 companies, including 80% of all Fortune 100 companies. Companies such as Netflix, Uber, Airbnb and PayPal all use Apache Kafka. [9]

2.5.1 *Key Concepts*

Kafka Cluster and Kafka Broker

A Kafka cluster consists of one or more Kakfa' servers (also denominated Kafka brokers). Despite being possible to initiate a Kafka cluster with only one Kafka broker, it is not recommended since it does not allow to take advantage of all the qualities of Apache Kafka. [16]

Inside the Kafka brokers, are located the topics. Producers publish messages in these topics so that later on consumers can read them.

To initiate a Kafka cluster, at least one Zookeeper is needed, that will have as main function the management of all the Kafka brokers of one cluster. [16]

Zookeeper

In order for a Kafka cluster to run properly, at least one zookeeper is needed. The zookeeper's main goal is to manage the several brokers of a cluster.

Besides managing the several brokers, the zookeeper has other functionalities, such as:

- Aiding in the election of a partition leader in a topic;
- Send notification to Kafka when:
  - A new topic is created;
  - An existing topic is eliminated;
  - A broker becomes unavailable;
  - A broker becomes available again.

The existence of more than one Zookeeper is possible. However, only one will be the leader. The others will act as alternate, and can only exist in odd numbers. For example, there can be 1,3,5,7 etc zookeepers.

Messages

Messages is the main resource of Apache Kafka. It's where the data that is published by the producers and consumed by the Consumers, is located. A Message is composed by a Key, that identifies in which partition the message will be stored, a Value, where the content is stored (it can be a simple number or a string to an object in JavaScript Object Notation (JSON), and a timestamp, that identifies the hour in which a message was sent to a specific topic.

Topic, Partitions and Offset

The term topic in Apache Kafka signifies a category or a name of a flow in which messages are published. It is identified through a single name that usually references the flow of data for which it was created. It is composed by a specific number of partitions that should be introduced by the user when it creates the topic. The messages are stored in the partitions in an ordained and immutable way, where each is attributed a single incremental identifier called Offset. [20]

When a message arrives to a topic, it is sent randomly to a partition, unless it carries a specific key. In this case all the messages with the same key will be stored in the same partition. [20]

By definition of the Apache Kafka, a message is retained in a topic during a week. However, the time period can be altered. For example, one day or one year. [28]

The topics may possess a replication factor that makes it so that partitions are replicated by the several brokers of a cluster. In this sense, if a broker has problems or is unavailable, the

topic messages can be consulted through other brokers, therefore increasing the availability of the software. [20]

Apache Kafka uses the concept of partition leader. There is only one leader partition, independently of the quantity of replicas. At the same time, it is exclusively responsible for receiving and providing the various messages. The replicas should only have as function to synchronize the information in order to prevent errors. [20]

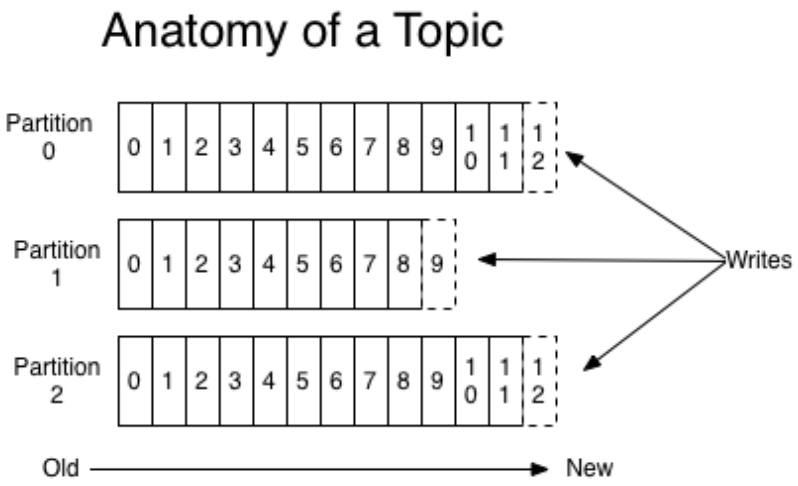


Figure 13: Anatomy of a Topic in Apache Kafka. Source: [10]

Producer

In Apache Kafka, the producers are responsible for publishing messages in a topic. A producer knows which broker to communicate to when it first calls (this is done automatically through Apache Kafka). In case the broker is unavailable, Kafka will redirect to another broker. [18]

By definition, a producer can not choose which topic partition it wants to publish the message. This can be altered by adding a key in the message. A key can be a number, a set of characters, etc..

When the producer publishes a message, an acknowledgement of receipt is sent to assure there aren't failures. There can be three types of confirmations: [18]

- Acknowledgment = 0;
  - Producer won't wait for acknowledgment;
  - Possible data loss;
  - Faster mode.
- Acknowledgment = 1;
  - Producer will wait for leader acknowledgment;

  - Subject to the possibility of loss of information if the process fails in the synchronization of replicas. It is, however, safer than in Acknowledgment =0.

- Acknowledgment = All.
  - Producer will wait for leader and replicas acknowledgment;
  - No data loss;
  - Slower process compared to Acknowledgment = 0 and = 1.

Consumer and Consumer Groups

The main goal of a consumer in Apache Kafka is to consume the data of a topic, with the role of treating and/or analysing. As it happens with the Producers, the consumers know, right at the initial communication, to which broker they should communicate. [17]

A consumer can subscribe to one or more topics, that are identified by their name, and consume stored messages in the various partitions in the order through which it was produced by the Producers. Despite the messages being consumed by order within a partition, it is not possible to guarantee the order of consumption through the several partitions of a topic. [17]

In the case of an overwhelming demand of messages that the Consumer can not keep up with there is an additional feature in the Apache Kafka that allows for scaling and following of the execution of the publication of messages, called Consumer Group. A consumer group is formed by consumers in which the various partitions of a topic are divided through the members of the group. In case there are more consumers in a consumer group then partitions in a topic, one of the members of the group will have to be inactive. It will act as an alternate, so in case one fails, it can immediately substitute it, without losing productivity in the software. [17]

Similarly to Producers, Apache Kafka has a mechanism to safeguard eventual failures during the consumption of the various messages. As soon as a consumer receives a message, it automatically communicates to Apache Kafka the Offset that it just received. If In case of failure the consumer becomes unavailable, the Apache Kafka will know what message it was in and as soon as the consumer becomes available, it will continue reading the mentioned message. There are three ways to do the Safety Commit of the reception of a message: [8]

- At Most Once;
  - Commit is realized as soon as the message is received;
  - In case something goes wrong after the reception, in other words, in its possible storage, the message will be marked as read, and it will not be read again.

- At Least Once;
  - Commit is realized after the reception and processing of the message;
  - In case something goes wrong after the message is received, there won't be a problem, since the message will be consumed again;

  - There is a possibility of the duplication of messages, in which case it should be saved by the source-code.

- Exactly Once.
  - Used by Apache Kafka on the Apache Kafka Streams.

2.5.2 *Apache Kafka Ecosystem*

Initially Apache Kafka was essentially, what was previously described, there being, the existence of Brokers, Producers and Consumers. However, throughout its existence it has evolved to a complete ecosystem, built and sustained to help companies that use this system. Some of the products that have helped build the ecosystem are the Apache Kafka Connect and the Apache Kafka Streams. [26]

The Apache Kafka Connect was created in November 2015 during an update of the Apache Kafka. It's goal is to simplify what, often times, is very repetitive. It allows to fetch data from a source (e.g. MongoDB, MySQL, Twitter API), and publish in one or more Kafka topics, without there being a need to type a single line of code. Furthermore, another product was created, Apache Kafka Sinks, that does the exact opposite. It fetches data from a topic and places it in a Sink, that can be anything from an ElasticSearch or a text file. [26]

Figure 14: Apache Kafka Connect. Source: [21]

There are several connectors developed by Apache Kafka and Confluent. However, and similarly to what happens with Docker, companies and developers are free to provide its own connectors, creating a hub of Connectors.[1]

Besides Apache Kafka Connect there is also the Apache Kafka Streams, a library produced in Java and launched in 2016. It aims to help in the processing and transformation of data in real time. In other words, Apache Kafka Streams is a library capable of consuming data from a topic, processing it, and, per a condition, sending it a specific topic. [26]

[1] https://www.confluent.io/hub/

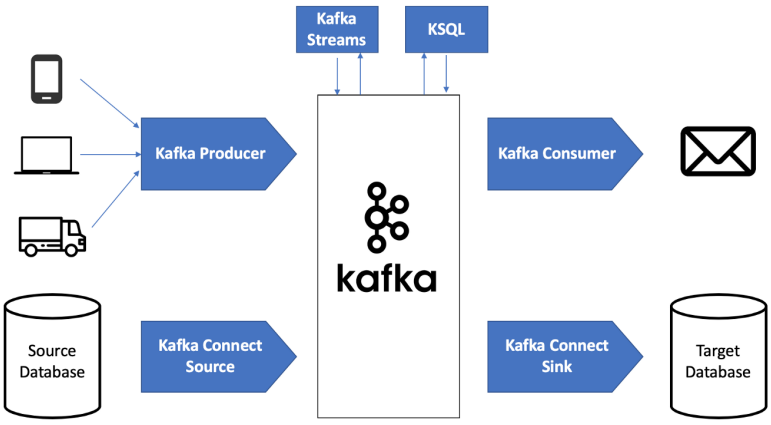


Figure 15: Apache Kafka Streams and Apache Kafka Connect. Source: [19]

Besides the two variables presented previously, Apache Kafka contains a wide range of options that can help companies in the most varied of occasions.[2]

[2]The complete ecosystem of Apache Kafka can be consulted in https://cwiki.apache.org/confluence/display/Kafka/Ecosystem.

3

METHODOLOGY

In order to carry out an organized work for this dissertation, the work methodology follows the following steps:

- Research and acquisition of bibliography and webliography on the two main concepts of the dissertation (Microservices, Apache Kafka);
- Multi-level study of Microservices and Apache Kafka;
- Analysis and exploration of existing solutions in the healthcare area;
- Meetings with the supervisor and co-supervisor.

3.0.1 *Phases*

- Phase I
  - Bibliography acquisition, research and Reading on the main and related areas;
- Phase II
  - State of the Art: study and definition. Description and critical discussion of related scientific work. Study and definition of the results and approaches that have already been presented in this or related areas;
- Phase III
  - Study existing software;
- Phase IV
  - Modeling and planning software: architectural decisions and proposed changes;
- Phase V
  - Start of the main developement cycle, software implementation;
- Phase VI
  - Software debugging and testing;

- Phase VII
  - Thesis development, chapters writing and consolidation ideas;
- Phase VIII
  - Review of the final thesis. Consolidation and delivery of the final document.

The following Gantt diagram, shown in figure 16, represents the work plan estimated for the conclusion of the dissertation. The green boxes represent the completed tasks, thus far.

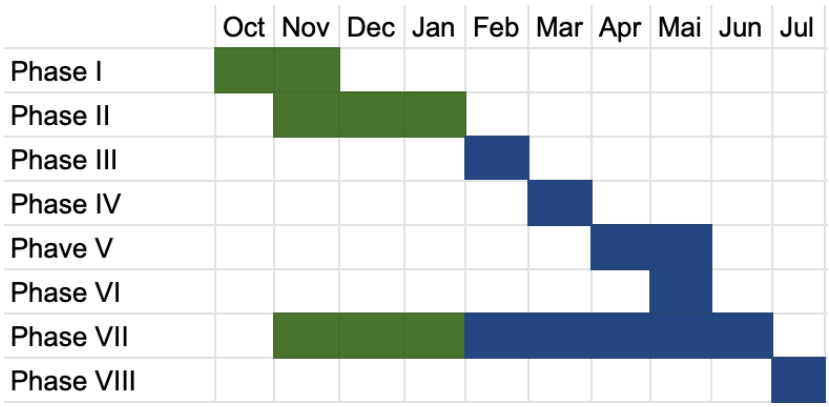


Figure 16: Dissertation's Gantt Diagram

BIBLIOGRAPHY

[1] André Vasconcelos Carrusca. Gestão de micro-serviços na Cloud e Edge. Master's thesis, Faculdade de Ciências e Tecnologia, 2018.

[2] Floyd Smith Chris Richardson. *Microservices - From Design to Deployment*. NGINX, 2016.

[3] Amazon Docs. API Gateways, 2019. URL https://docs.aws.amazon.com/whitepapers/latest/modern-application-development-on-aws/api-gateways.html.

[4] Amazon Docs. Modern Application Development on AWS, 2019. URL https://docs.aws.amazon.com/whitepapers/latest/modern-application-development-on-aws/command-query-responsibility-segregation.html.

[5] Microsoft Docs. Implementing event-based communication between microservices (integration events), 2021. URL https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/integration-event-based-microservice-communications.

[6] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. 06 2016.

[7] IBM Cloud Education. SOA (Service-Oriented Architecture), 2019. URL https://www.ibm.com/cloud/learn/soa.

[8] Apache Kafka Foundation. Apache Kafka Documentation - Message Delivery Semantics, 2017. URL https://kafka.apache.org/documentation/#semantics.

[9] Apache Kafka Foundation. Apache Kafka: A Distributed Streaming Platform., 2017. URL https://kafka.apache.org/.

[10] Apache Kafka Foundation. Apache Kafka Documentation - Topics and Logs, 2017. URL http://kafka.apache.org/090/documentation.html#intro_topics.

[11] Martin Fowler. CQRS, 2011. URL https://martinfowler.com/bliki/CQRS.html.

[12] Romana Gnatyk. Microservices vs Monolith: which architecture is the best choice for your business?, 2018. URL https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/.

[13] Red Hat. What is service-oriented architecture (SOA)?, 2018. URL https://www.redhat.com/en/topics/cloud-native-apps/what-is-service-oriented-architecture.

[14] Marilena Ianculescu and Adriana Alexandru. Microservices - a catalyzer for better managing healthcare data empowerment. *Studies in Informatics and Control*, 29:231–242, 07 2020. doi: 10.24846/v29i2y202008.

[15] Nathan Khan, Asif; Steckmeyer, Pierre; Peck. Running Containerized Microservices on AWS. *Aws*, (November):21, 2017. URL https://d0.awsstatic.com/whitepapers/microservices-on-aws.pdf.

[16] Stephane Maarek. Kafka brokers and data replication explained, 2017. URL https://www.youtube.com/watch?v=ZOU7PJWZU9w&ab_channel=StephaneMaarek.

[17] Stephane Maarek. Kafka consumer and consumer groups explained, 2017. URL https://www.youtube.com/watch?v=lAdG16KaHLs&ab_channel=StephaneMaarek.

[18] Stephane Maarek. Kafka producers explained, 2017. URL https://www.youtube.com/watch?v=lh_tjm0yPz4&ab_channel=StephaneMaarek.

[19] Stephane Maarek. The Kafka API Battle: Producer vs Consumer vs Kafka Connect vs Kafka Streams vs KSQL!, 2018. URL https://medium.com/@stephane.maarek/the-kafka-api-battle-producer-vs-consumer-vs-kafka-connect-vs-kafka-streams-vs-ksql

[20] Stephane Maarek. Kafka topics, partitions and offsets explained, 2019. URL https://www.youtube.com/watch?v=_q1IjK5jjyU&ab_channel=StephaneMaarek.

[21] Akhlaq Malik. ETL with Kafka, 2018. URL https://blog.codecentric.de/en/2018/03/etl-kafka/.

[22] James Lewis Martin Fowler. Microservices, 2014. URL https://martinfowler.com/articles/microservices.html.

[23] Andre Newman. Is your microservice a distributed monolith?, 2020. URL https://www.gremlin.com/blog/is-your-microservice-a-distributed-monolith/.

[24] Max Rehkopf. Continuous Delivery Principles, . URL https://www.atlassian.com/continuous-delivery/principles.

[25] Max Rehkopf. What is Continuous Integration?, . URL https://www.atlassian.com/continuous-delivery/continuous-integration.

[26] Matthias Sax. *Apache Kafka*, pages 1–8. 01 2018. doi: 10.1007/978-3-319-63962-8_196-1.

[27] Jonathan Schabowsky. Microservices Choreography vs Orchestration: The Benefits of Choreography, 2019. URL https://solace.com/blog/microservices-choreography-vs-orchestration/.

[28] Liron Shapira. What is Kafka Retention Period?, 2018. URL https://www.cloudkarafka.com/blog/2018-05-08-what-is-kafka-retention-period.html.

[29] Liron Shapira. When logic programming meets CQRS, 2019. URL https://hackernoon.com/when-logic-programming-meets-cqrs-1137ab2a5f86.

[30] Phil Wittmer. Microservices vs SOA: What's the Difference?, 2020. URL https://www.tiempodev.com/blog/microservices-vs-soa/.

[31] XenonStack. Service-Oriented Architecture vs. Microservices | The Comparison, 2020. URL https://www.xenonstack.com/insights/service-oriented-architecture-vs-microservices/.