Universidade de Aveiro
2023

Miguel Filipe
Rodrigues Almeida
de Matos Fazenda

Pipeline de Engenharia de Dados como serviço para iniciativas de MLOps

Data Engineering Pipeline as a service for MLOps initiatives

Universidade de Aveiro
2023

Miguel Filipe Rodrigues Almeida de Matos Fazenda

Pipeline de Engenharia de Dados como serviço para iniciativas de MLOps

Data Engineering Pipeline as a service for MLOps initiatives

"*The greatest challenge to any thinker is stating the problem in a way that will allow a solution*"

— Bertrand Russell

Universidade de Aveiro
2023

Miguel Filipe Rodrigues Almeida de Matos Fazenda

Pipeline de Engenharia de Dados como serviço para iniciativas de MLOps

Data Engineering Pipeline as a service for MLOps initiatives

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática , realizada sob a orientação científica do Doutor (Ilídio Fernando de Castro Oliveira), Professor associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Mestre (Nuno Sá Couto), Professor auxiliar convidado do Departamento de Matemática da Universidade de Aveiro.

Dedico este trabalho a Jesus Cristo por me ter permitido e capacitado chegar até aqui.

agradecimentos / acknowledgements

Agradeço toda a ajuda da minha família e amigos.

Palavras Chave AWS, Big Data, Data Engineering, DevOps, Docker, GitOps, MLOps, Pipeline, Kubernetes.

Resumo

As empresas de hoje necessitam cada vez mais de utilizar dados para fundamentar as suas decisões com informação crucial. Os dados necessitam ser processados e analisados para então ser possível extrair informação. Porém, o processamento de dados pode ser um processo longo e complexo devido à possibilidade do volume de dados ser enorme, este manuseamento de grandes volumes de dados reflete o conceito de Big Data. Para se manterem à frente da concurrência, as empresas precisam utilizar sistemas projetados de acordo com os princípios de Data Engineering para manusear grandes volumes de dados. Data Engineering é uma disciplina que se foca na construção de sistemas que ingerem, processam, armazenam e disponibilização grandes volumes de dados. O objetivo deste dissertação é a construção de um sistema, mais precisamente uma pipline, capaz de manusear grandes volumes de dados, estes relacionados a productos eletrónicos, e actuar sobre os dados através de modelos de ML de maneira a prever o próximo valor do produto. Os valores previstos devem então ser armazenados e disponibilizados a utilizadores. Este sistema serve com alternativa para as companhias initiantes de MLOps, que combinam Data Engineering, DevOps, e ML para processar dados. O sistema possui algumas limitações impostas quanto à arquitetura e ferramentas envolvidaas, deve ser baseado em microserviços, agnostico ao ambiente cloud, containerizado, baseado na SMACK stack e utilizar ferramentas grátis e de código aberto. O desenvolvimento do sistema será feito com a framework operacional GitOps, que aplica as melhores práticas de DevOps, como versionamento, compliança, colaboração e CI/CD, a automação de infraestrutura. O sistema passou por múltiplas iterações, cada uma representando um estágio do desenvolvimento. Uma iteração inicial feita em Docker Compose para servir como Prova de Conceito, uma intermediária para adaptar a pipeline ao ambiente Kubernetes e testá-la, e uma final na AWS através do EKS, representando um cenário de produção da vida real. As versões Kubernetes possuem monitorização de maneira a facilitar a observação e controlo do sistema. Em geral, este documento aborda as ferramentas escolhidas, as múltiplas versões da pipeline e os objetivos de cada uma, e os resultados obtidos e o significado por trás deles.

Keywords AWS, Big Data, Data Engineering, DevOps, Docker, GitOps, MLOps, Pipeline, Kubernetes.

Abstract

The companies of today increasingly need to use data to ground their decisions with crucial information. To make the best use of it, data needs to be processed and analyzed for information to be extracted from it. However, information extraction from data can be a long and complex process, due to the possibility of data having enormous volumes. The handling of large volumes of data represents the concept of Big Data To stay ahead of the competition, companies need to use systems designed according to Data Engineering principles in order to handle these large volumes of data. Data Engineering is a discipline that focuses on the construction of systems that can ingest, process, and store large amounts of data. The objective of this dissertation is the construction of a system, more precisely a pipeline, that can handle large volumes of data, related to electronic products, and apply ML models on top of it to predict the next value of the intended product. The predicted values should then be stored and served to users. The system has some limitations imposed regarding the architecture and tooling, it must be based on microservices, cloud-agnostic, containerized, orchestrated, based on the SMACK stack, and use free and open-source tools. This system serves as an alternative for MLOps startups, which combine Data Engineering, DevOps, and ML to process data. The development of the system will be done with the GitOps operational framework, which applies DevOps best practices, such as versioning, compliance, collaboration, and CI/CD, and applies them to infrastructure automation. The system passed through multiple iterations, each representing a stage of the development. An initial iteration made in Docker Compose to serve as Proof of Concept, a middle one to adapt the pipeline to the Kubernetes environment and test it, and a final one on AWS through EKS, representing a real-life production scenario. The Kubernetes versions have monitoring in order to facilitate testing and observation of the system. In general, this document approaches the tools chosen, the multiple versions of the pipeline and objectives of each, and the obtained results and meaning behind them.

Contents

List of Figures

List of Tables

List of Code Snippets

Glossary

GDPR	General Data Protection Regulation
HIPAA	Health Insurance Portability and Accountability Act
MLOps	Machine Learning Operations
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
CDC	Change Data capture
ETL	Extract Transform Load
PaaS	Platform as a Service
CI/CD	Continuous Integration/Continuous Deployment
AI	Artificial Intelligence
ML	Machine Learning
CRUD	create, read, update, and delete
ACID	atomicity, consistency, isolation, and durability
AWS	Amazon Web Services
EKS	Elastic Kubernetes Service
IAM	Identity and Access Management
EC2	Elastic Cloud Compute
EBS	Elastic Block Storage
VPC	Virtual Private Cloud
HDFS	Hadoop Distributed File System
RDD	Resilient Distributed Dataset
PoC	Proof of Concept
IaC	Infrastructure as Code
IoT	Internet of Things
SLA	Service Level Agreement
IEETA	Institute of Electronics and Informatics Engineering of Aveiro

CHAPTER 1

Introduction

1.1 Context

The world of Data is vast and dynamic. It encompasses many fields of study that range from science to mathematics, informatics, and many more.

Today's companies use data to make informed decisions, gain insights, and lead innovation. For this to become possible, companies must have proper infrastructure for processing and analyzing data and use statistical analysis and machine learning to extract meaningful data, all the while applying data governance throughout the process.

These actions reflect some of the most predominant concepts of the Data World: Big Data, Data Science, and Data Governance.

Big Data handles large data sets by processing them in specialized systems that distribute the work and process it in parallel. Technologies like Hadoop Distributed File System (HDFS)[1] and Apache Spark[2] provide the necessary infrastructure and processing power to build these systems.

Data Science involves statistical analysis and machine learning to extract meaningful information from data and use it for decision-making. Pytorch[3] and Tensorflow[4] are some of the most used tools in this area.

Data Governance focuses on handling data to make it as secure, private, accurate, available, and usable as possible. Complying with data policies like General Data Protection Regulation (GDPR) [1] and Health Insurance Portability and Accountability Act (HIPAA) [2] is a must. As data holds great value and influence in the technical and business world, mismanagement and ill-intended use can cause irreversible damage.

For companies that interact with data to stay in front of the competition, they need to be able to have systems that are able to ingest, process, store, and serve it, while being able to scale to workload demand in order to be as efficient as possible with regards to cost

[1]`https://hadoop.apache.org/`
[2]`https://spark.apache.org/`
[3]`https://pytorch.org/`
[4]`https://www.tensorflow.org/`

and performance. As such, Ritain.io [3] in partnership with University of Aveiro suggested a dissertation theme about creating a data engineering pipeline that is able use machine learning models to handle huge amounts of data related to electronic products in order to predict the next price. This help the company to understand price changes and trends in new products pricing by competitors.

Overall, the world of Data is a dynamic and fast-growing world. If companies want to stay competitive and use data best, Machine Learning Operations (MLOps) is a must.

1.2 ZenPrice^{TM}

Mobile phones as well as mobile plans and accessories are sold on the commercial telecomm market. Since there is a vast amount of available merchandise, analyzing this market can be difficult and time-consuming due to the daily launches of new products by multiple businesses.

A new product will be significantly more expensive than the previous model, for instance, a newly released mobile phone with only minor differences from the old model, such as more storage. As a result, businesses must come up with fresh approaches to data analysis to speed up and automate operations that are often carried out manually by marketing teams to stay ahead of their customer chains and emerge as the market's most competitive.

As a result, many businesses are placing bets on the analysis of the market and its long-term trajectory because it allows to understand price changes and trends in new products by competitors.

ZenPrice^{TM} is a Platform as a Service (PaaS), that aims to provide services that can help telecomm businesses define product prices in an automatic and reproducible way. It provides suggestions and recommendations to updates prices of products and telecomm services [4].

The platform provides multiple functionalities like data extraction, filtering, monitoring, and visualization of retrieved product prices, all of it independent of the company in question. This allows companies defining a new price for a product, understanding the most recent trends in the market, or even observing changes in the competition.

ZenPrice^{TM}'s services are achievable due to how it is implemented. Data extraction and collection are made through web crawlers that scrap websites. These web crawlers look into HTML and even images to identify products and extract values. Data storage is made with a Big Data solution to store large amounts of data to be processed in the future. Data visualization is made through a dashboard that acts on top of a data processing layer. Visualization is made through multiple graphs and charts.

1.3 Objectives

The main objective of this dissertation is to conceive and implement a data engineering pipeline that is able to run Machine Learning (ML) predictive models in order to predict the next possible value of an electronic product. The intended solution should be cloud agnostic, while being able to scale according to demand.

Following the Amazon Web Services (AWS) Well-Architected Framework [5], the pipeline must be able to ingest provided data sets, process said data through machine learning predictive models, store it in a specified user output, and serve it to the user. All the while monitoring each tool and revolving environment.

Development of the pipeline must follow a GitOps approach. The deployment environment must be treated as Infrastructure as Code (IaC), effectively creating the environment through configuration files.

CHAPTER 2

Background and State of the Art

This dissertation involves several distinct areas, each one related to Data Engineering. Since each field concerns a distinct aspect of the purpose, it becomes necessary to gain foundations and learn the fundamentals.
Not only is it necessary to understand the concepts, but to know as well how they interact with each other and are put into practice.
As such this chapter aims to give a detailed and exploratory overview of each topic related to Data Engineering, such as responsibilities, lifecycle, and architectures, followed by an overview of MLOps, and finally related work and implementations of third parties' data engineering systems.

2.1 Data Engineering

Data Engineering is a group of norms, processes, and relations that allow interaction with the Data World. It is a specific part of Data Science that focuses on the implementation and maintenance of data infrastructure to help Data Scientists and Analysts in their jobs [6] [7].

Implementing and maintaining data processing systems are made possible by applying Data Science and Software Engineering techniques principles to handle large quantities of data [8].

As Data Engineering derives from multiple areas, many undercurrents are present, namely Security, Data Management, DataOps, Data Architecture, Orchestration, and Software Engineering [7] [6].

2.1.1 The Data Engineer

The Data Engineer bridges Data Science and Software Engineering by working with both teams. Implementing and maintaining data processing systems to transform unusable data into usable and make it available for use is the goal of the Data Engineer [9] [10].

Their background and skills encompass not only technical but also business-related fields since data may affect a company's future. Usually, their roles and responsibilities vary according to the company's objectives and data maturity, categorizing a Data Engineer into two types.

Data Maturity

Data Maturity is a concept that helps measure the data utilization of companies, their capabilities, and how dependent they are on it. Maturity is not dependent on age but dependent on proficiency with it.

According to Joe Reis [9], their simplified data maturity model provides meaningful information in a simplified way without losing important details. This model has the following phases: 1. starting with data, when the company does not have clearly defined goals if any, and data infrastructure is in the early stages; 2. scaling with data, the company has formal data practices, the focus is the scalability of their systems, and planning for a data-driven future; 3. leading with data, scalability and integration of the system is seamless, future-planning is data-driven, and self-service analytics are possible;

Background & Skills

Since Data Engineering is a relatively new and expanding field, there is no defined education path to reach it. Many Data Engineers of today come from backgrounds related to Data Science or Software Engineering, which are the main influencing fields.

Although the path to Data Engineering still needs a concise structure, the knowledge necessary for it does not. The Data Engineer should possess skills that can help in the technical and business fields [9].

Technical skills should be able to handle aspects related to Software Engineering, Data Management, and Data Architecture, such as a deep understanding of data and technology, various best practices around data management, and being aware of various tool options.

Business knowledge should handle communication between technical and non-technical people, understand the impact of data across the company, and know the users' necessities.

Responsibilities

Data Engineers independent of their role in the company possess similar responsibilities since their work always carries technical and business-related responsibilities. As previously mentioned in subchapter 2.1.1 regarding skills, the responsibilities should reflect them.

Business responsibilities can be essentially resumed to the following [9]:

- communication with people from technical and non-technical fields should be clear and concise;
- awareness of the project's scope and its impact in the technical and business fields;
- adoption and practice of work culture to promote organization and communication;
- cost control permits budget awareness and facilitates decision-making;
- continuous learning and adaptation are required to accompany the everchanging data;

Technical responsibilities can be essentially resumed to the following [9]:

- write production-level code at any abstraction level;
- make proficient use of query languages, usually through SQL;

- make the bridge between Data Science and Software Engineering, usually through Python;
- have proficiency in languages open source tools are commonly written, usually Java or Scala;
- being comfortable with the terminal, it enables the use of many tools that do not have GUI and permit scripting to automate tasks;

Types of Data Engineer

Taking into account what has been referenced in 2.1.1 and according to Anderson [11] and Joe Reis [9] there are two types of jobs that the Data Engineer can perform, both are greatly influenced by the stage the data maturity the company presents.

Type A stands for "analysis" and "abstraction". The main focus is the analysis of data. Work is focused on using off-the-shelf products, services, and tools, while infrastructure is as abstract and straightforward as possible. They are present through all the Data Maturity levels [9].

Type B stands for "building". The main focus is the construction of tools and systems that scale to handle large quantities of data and satisfy the needs of Data Scientists and Analysts [9].

Conclusion

In conclusion, a Data Engineer is someone who leads companies through data. They guide the companies through the lowest to the highest level of Data Maturity by performing roles and taking responsibilities that reflect their background and skills in Data Science and Software Engineering. Their work is dependent on the Data Maturity stage. It can be divided into two, where the main focus of their work can be either the handling and processing of data through tools, frameworks, and services or the building of infrastructure to handle large amounts of data and satisfy the needs of Data Scientists and Analysts.

2.1.2 Lifecycle

The Data Engineering Lifecycle is a framework designed by Joe Reis [9] with the aim to help better understand the phases through which data passes inside data processing systems. Throughout the lifecycle, the influence of major undercurrents can be noticed as their impact is immense and act as the foundation of it.

The lifecycle is composed of 5 phases, each phase focuses on diverse and specific tasks and respective characteristics, considerations, and details. The lifecycle starts with generation, where data is created, followed by ingestion, data is inputted into the system, transformation, where data is processed, storage, data is stored inside the system, and serving, where data is made available for the user.

Figure 2.1 illustrates the lifecycle.

Aside from generation and serving, the ingestion, transformation, and storage phases can be in any order as they need to satisfy the system requirements in the best way possible.

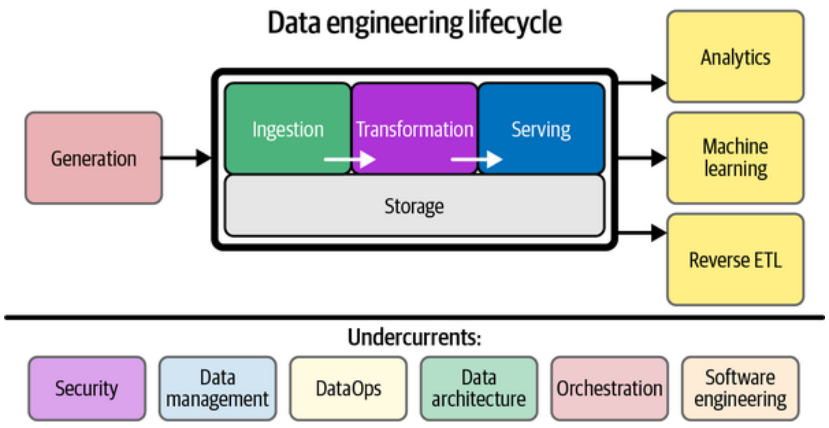


Figure 2.1: Data Engineering Lifecycle [9]

It is important to emphasize that the Data Engineering Lifecycle is just a subset of the Data Lifecycle [9].

Figure 2.2 demonstrates this relation.

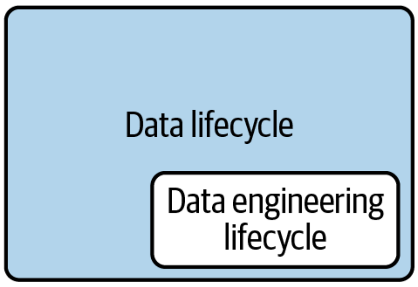


Figure 2.2: Data Engineering Lifecycle as a subset of the Data Lifecycle [9]

Generation

Data Generation is the phase where data comes into existence. Behind its origin is a system creating the data which can take many forms. By ensuring that data is properly generated, data-driven projects can provide the correct analysis and decisions to move forward.

When dealing with source systems, there is a group of main ideas that must be taken into account as at least some of them are always present through the interaction [9]. These are:

- files and unstructured data, aggregations of data that may or may not hold a specific format;
- APIs, a standard means to exchange data between two different entities;
- Online Transaction Processing (OLTP), application databases built for storing data with the capacity for high amounts of individual read and write operations, and possess characteristics like atomicity, consistency, isolation, and durability (ACID);
- Online Analytical Processing (OLAP), is a system built for running queries against large amounts of data. In contrast to OLTP, OLAP poorly handles individual records;
- change data capture, known as Change Data capture (CDC), is a method for capturing eventual data changes. These events can be related to writing, deleting, or updating data;

- logs, a method for registering every event that happens in a system. In the case of an unexpected event, logs can help identify when and what happened unexpectedly;
- create, read, update, and delete (CRUD), is a transactional pattern that represents the four basic operations of persistent storage;
- insert-only, a pattern for storing records with a timestamp in a table, where the application has access to it if need be;
- messages and streams, data exchanged between 2 systems. The difference between one and another is that a message is a single event with a start and end, while a stream is a constant append of messages;
- types of time, depending on the event in question, time can refer to generation, ingestion, or processing. In the case of streaming, the occurrence of these events can overlap each other;

Characteristics that are taken into account when considering source systems can be essentially resumed to the type of source system, data persistence, data generation, data schema, data time of arrival, and data consistency. As each source system possesses different characteristics, the Data Engineer should be able to handle them in parallel, making data acquisition as efficient as possible while taking into account the limits of each system [9].

As these characteristics are vital for the well-being of the involved systems, a line of communication with the source system owners should be maintained to promote awareness of possible changes that could hinder data processing.

Ingestion

Data Ingestion is the phase responsible for receiving data into the system and putting it to a target site for further processing. Together with Generation, they become the main bottleneck of the Data Engineering life cycle, as the rest of the lifecycle is dependent on their work [9] [12].

When defining infrastructure for the Ingestion phase, characteristics like how data is ingested and how ingested data have a big influence. In order to better define the infrastructure, key considerations can be essentially resumed to data characteristics such as reliability, destination, access frequency, arrival volume, format, and use case.

For when to ingest data, there are effectively 3 options [12]:

- streaming, data is generated and inputted constantly all the time, allowing the data to proceed to the next phase in the lifecycle in a short period of time;
- batch, data is introduced in a fixed time interval or in a fixed volume of data. Effectively, this approach immediately constrains the next stages of the lifecycle as it stipulates time periodical time intervals;
- lambda, which combines streaming and batch methods. While batching methods gather data to send, streaming methods gather data that the slower batching methods still have not collected;

When choosing between streaming, batching, and lambda, there are key considerations that should be taken which allow us to account for specific details that might not be apparent in the beginning. These considerations are related to differences between ingestion and processing capacities when streaming, weighting millisecond streaming and micro-batching, pros and cons between streaming and batching in the present context, associated costs, self-hosting or cloud services, and the response of machine learning models in streaming and batching contexts.

For obtaining data for ingestion, there are effectively 2 models, push and pull [9]. The characteristics of each model only differ in who moves the data, in push the data is moved by the entity responsible for creating it, while in the pull model, the data is fetched by an entity that needs it. Each model has different architectures and should be used accordingly to the present context.

Transformation

Data Transformation is the phase responsible for transforming raw data, which is devoid of meaning and logic, into processed data with meaning and logic. Transformed data provide many benefits for the company, them being [13]:

- higher data quality, which greatly affects decision-making by providing better and more correct information to be used as decision ground. This can be acquired by removing duplicates, deleting null values, and reducing inconsistencies;
- improve data management, as huge amounts of data are generated, they become harder to handle and store. Transformed data become simpler and possesses a lower volume;
- seamless data integration, as companies use several tools and interconnect with each other, data integration becomes harder. A common data format provides easier integration;
- obfuscate sensitive data, as data awareness and security become an increasingly concerning issue, abiding by regulations such as GDPR [1] and HIPAA [2] involve masking or removing sensitive data;

When defining the infrastructure for the transformation phase, key considerations can be essentially resumed [9] [13]:

- data preparation, details like data format before and after cleansing, applied transformations, and isolation from other phases, so it can be properly transformed and produce the desired results;
- tracking, being aware of the impact of every transformation helps gain a deeper understanding of the impact of data, whether technical or business-related and its value before and after transformation;
- business rules associated with transformation, by standardizing business logic, through data modeling it becomes easier to understand the meaning of data;

Storage

Data Storage is the phase where data is stored in a central repository. Whether it is for processing or querying, the type of repository can vary and each one has specific characteristics

Storage is usually organized using a hierarchy of levels of abstraction, which refers to the different ways data can be represented and accessed. Each level of abstraction focuses on specific details. These levels are basic components, storage systems, storage abstractions.

Basic components are what constitute the lowest abstraction level of storage. As data passes through the data engineering lifecycle, it also passes through different storage media. While most modern storage solutions abstract the developers from lower-level complexities, it is essential to know their characteristics, performance, durability, cost, and any extra detail for the use case. Usually related to hardware like HDD, SSD, RAM, network, and CPU. However, some software techniques in tandem with the hardware enable many possibilities such as serialization, compression, and caching [9].

Storage systems are an abstraction above basic components. They provide meaningful features and capabilities like data management and protection. They enable the full potential of the basic elements. When choosing a storage system type, details like data type, data access frequency, data volume, and many other necessary details must be known first. As such, many storage solutions exist to accommodate the existing necessities, them being [9]:

- file storage, a system designed for storing a wide range of file types in a hierarchical structure organization. Features like versioning, recovery, replication, and access control can also be provided;
- block storage, a system designed for storing data in chunks with a fixed size. It allows fine control of storage, scalability, and durability of data beyond the capacities of basic components;
- object storage, in the context of Data Engineering, object storage is designed for storing specific data formats (TXT, CSV, JSON, images, videos, audio) in a key-value format, with large batch read and write performant operations;
- cache & memory-based storage, volatile memory designed to provide fast access to data that is usually queried a significant amount of times in small intervals of time;
- Hadoop distributed file system, a system designed for storing and processing data in a distributed architecture by breaking large files into blocks and distributing them through the nodes. Data is also replicated through the nodes to provide fault tolerance and data availability;
- streaming storage, a system designed for storing and serving short-lived data with a constant influx. However, most systems nowadays provide more consistent and lengthy storing capabilities;

Storage abstractions are organizations and patterns built on top of storage systems. They provide a unified way of accessing data regardless of the underlying hardware and software infrastructure without worrying about its details. There are many storage abstractions, each with its appropriate characteristics and use case. The use of a storage abstraction is essentially composed of 4 considerations, them being purpose and use case, update patterns, cost, and separate storage and computation: Knowing the considerations that take part in the abstractions, there are only a few that fit in the field of Data Engineering, them being [9] [14]:

- data lake, a unified repository for storing structured and unstructured data. It supports querying and processing of data at large scales;
- data warehouse, designed for storing and querying processed data. Its main purposes are related to analytics like business intelligence and reporting;

- data lakehouse, a hybrid architecture of the data lake and the data warehouse to provide the benefits of both;
- data platform, tightly coupled tools and frameworks strongly connected to the storage layer that form an ecosystem. It can be composed of every type of tool for each task related to Data Engineering.
- stream-to-batch storage architecture, an abstraction with the intent to receive streaming data and make it available to every type of consumer, either stream or batch;

Serving

Serving is the last phase of the lifecycle. Commonly there are three different use cases, depending on the final objective, them being analytics, machine learning, and reverse Extract Transform Load (ETL).

Analytics is the center of many data-related projects. They depend on data analysis to support many objectives through different types of analysis such as [9] [15]:

- business intelligence, related to data processing with the aim to lead a company. As the company grows in terms of Data Maturity, its proficiency and dependency on data grow as well. As it grows, the data access model changes from an ad-hoc style (use case specific) to a self-service model (free access). Data is consumed from a centralized repository that aggregates data in general;
- operational, which focuses on operation details like resource management, cost control, or application monitoring. In contrast to business intelligence where data is aggregated and analyzed throughout time, operational consumes data directly from data sources in real-time, allowing for instantaneous answers according to needs;
- embedded, considered a middle ground between business intelligence and operational. Analytic functionalities are made available as a service to users outside the company;

Machine Learning is an area of Artificial Intelligence (AI) that focuses on data processing through various algorithms that "learn". These algorithms construct models that allow the prediction or decision without being explicitly programmed for such [16].

Although the Data Engineer does not need to know about Machine Learning, at least having a notion of the revolving concepts certainly does help, mainly operational concepts like foundations, data processing, and adequate model use cases.

Key considerations that should be taken into account while serving data for machine learning involve aspects such as data quality for feature engineering, facility in finding important data, technical and organizational limitations between data engineering and machine learning engineering, and biases present in the data set.

Reverse ETL is the process of re-introducing data that has been processed from the system, into the system again [17]. This approach allows the data to be stored inside the repositories and compared against the previously present data, instead of just presenting it through a BI tool.

When serving data, there are some key considerations that should be taken into account [9]. These can be essentially resumed to the following:

- trust, one of the most important aspects for the user. Trust is achieved by validating and observing the data provided in order to guarantee the desired results;
- user use cases, data is in the best state when it leads to action. However, the action depends on the users' use case, which in turn involves questions such as "what action", "who performs the action" and "is the action automatable". As a general rule, when these questions have an acceptable answer, the focus should always be on which use case has the highest return on investment;
- data product, the product that is sold or provided must only have features that satisfy the needs of the user. Unnecessary features that the user can not use can affect trust;
- self-service, it depends entirely on the target audience. As it is a "niche" feature, controlling and accessing it for people who do not need the feature can make its implementation difficult because of access control and authority;
- data definition and logic, ultimately the meaning and value of data depend on the definition stipulated throughout the entire company;
- data mesh refers to the data sharing structure between teams. Teams ingest, prepare, and serve data to and from other teams, forming a peer-to-peer data repository rather than a centralized one;

2.1.3 Major Influences

With the evolution of Data Engineering, there has been an increase in tools, responsibilities, and skills that a Data Engineer must have. This is the reflection of many undercurrents that leave their influence. These influences are Security, Data Management, DataOps, Data Architecture, Orchestration, and Software Engineering.

Security

Security should be the first influence on the data engineer's work. Essentially, it boils down to controlling access to data and systems while taking into account people, processes, and technology [9] [18].

People represent the weakest link in security. Many times errors happen by human origin due to lack of care. As such, security must be taken into account to always expect the worst possible case and always be suspicious of requests for credentials.

Processes allow structuring and procedure definition to establish active and passive security. These procedures consider aspects such as active security, the principle of least privilege, shared responsibility when in the cloud, and backing up data.

Technology deals with aspects such as updating security systems, encrypting data at all stages of the lifecycle, logging, monitoring and alerting, and network access control.

Data Management

Data Management reflects the ability to handle data while taking into account quality, integrity, security, and usability. It is a concept that can be applied to almost anything related to data, whether business or technical-related.

Henceforth, Data Management can be reflected in the abilities and characteristics of companies when handling data. When handling data companies need to follow certain concepts such as Data Governance, data modeling, and design, data lineage, data integration and interoperability, data lifecycle management, and ethics and privacy.

Data Governance effectively is the capacity of a company to handle data, understand and transform it can be used as intended, and extract as much value as possible. Discoverability, quality assurance, metadata gathering, and accountability are tasks that reflect Data Governance [18].

Data Modeling and Design specifically handle data formats. As there are formats desired for specific use cases, data needs to be transformed to be used effectively [19].

Data Lineage focuses on keeping track of data. Not necessarily the data, but what happens to data, where is data, what is the objective of data, and anything else that can provide value for tasks such as error tracking, accountability, and debugging data processing systems [20].

Data Integration and Interoperability aims to input new data into the system to integrate it with already existing data. This way, updated results can be obtained.

Data lifecycle management focuses on providing structure and organization to the data engineering lifecycle by providing processes and rules that benefit streamlining and compliance of existing methods, controlling costs, and assuring data usability.

Ethics and privacy are vital to earning the trust of users. As data mismanagement may lead to disastrous situations and negatively impact companies and their users, a higher focus has been given to preventing such repercussions.

DataOps

DataOps is a group of practices that focus on automating data projects through Agile methodologies, DevOps, and statistical process control. Through these, it becomes possible to automate, observe, monitor, and act upon data projects to improve quality, speed, and collaboration to promote a culture of continuous improvement [21].

The use of Continuous Integration/Continuous Deployment (CI/CD) practices allow improvements to the velocity, quality, predictability, and scalability of work. The key difference is that the focus is to bring these improvements to data analytics.

Data Architecture

Data Architecture combines the business and technical requirements that dictate the architecture of the system. These details influence the lifecycle through which data passes, and the Data Architect is responsible for this [22].

The Data Architect is responsible for gathering the needs and requirements for new data use cases. Although the Data Engineer does not necessarily handles these tasks, it is good to be aware of them as the work between the two is done in tandem.

This is approached more deeply in chapter 2.1.4.

Orchestration

Orchestration focuses on managing and aligning tasks and jobs synchronized and organized based on a schedule [23].

Besides organization and scheduling of jobs, it is also possible to do logging, visualizations, and alerting, depending on the tool used for Orchestration.

As jobs are usually executed until the end in Orchestration, the work follows the batch model. However, some investment has been made to make streaming jobs easier in orchestration.

Software Engineering

Software Engineering is a systematic approach to software development [24]. This approach greatly influences the work of the data engineer since some areas are common, such as:

- core data processing code, when needed, the data engineer should be able to delve into the lower levels of abstraction to understand how tools fundamentally work;
- development of open source frameworks, usually the work of the data engineer revolves around using already built tools and frameworks, however when needed, data engineers should be able to contribute the tools and frameworks to accommodate their needs better;
- streaming, data processing in real-time is more demanding than batch processing. When needed, if a solution can not be found, then an alternative must be taken to achieve as close to the desired result as possible;
- IaC, through the abstraction of code, it becomes possible to set up and configure systems. It enables version controlling and repeatability in a controlled environment;

2.1.4 Architectural Patterns

As was mentioned in sub-chapter 2.1.3, Data Architecture is a discipline that focuses on gathering technical and business-related requirements to help define a proper architecture for data processing systems.

A Data Architect, through accumulated expertise in data handling, databases, operating systems, security, and data architecture, builds the blueprints for data processing systems while considering technical and business-related requirements. They might have started as a data scientist, engineer, or analyst, but through years of expertise, they become leaders of data teams responsible for the companies data strategy [25] [26].

Evaluating Data Architectures

Since data processing systems have diverse requirements and objectives, Data Architecture is considered a rather abstract discipline. No specifications are provided when defining an architecture. However, Joe Reis [9] defined some critical principles based on the AWS Well-Architected Framework [5], and Google Cloud's Five Principles for Cloud-native Architecture [27] to help evaluate architectural decisions and practices. These are:

- common components, by choosing components present in several teams, knowledge sharing enables faster movement of teams;

- planning for failure, building reliable and fault tolerant systems that can recover and help prevent or handle failure scenarios;
- scalability, scale systems accordingly to workload, up or down to zero, or even during workload spikes. Scaling considering workload helps control costs;
- leading through architecture, as Data Architects are responsible for technology decisions and communicating them effectively to the team, they should be technically competent and have strong leadership skills;
- continuous architecture, architects should constantly be developing in response to business and technical changes through an Agile approach to enable companies to keep up with data;
- build loosely coupled systems, whether technical or business-related, loose coupling enables independence, allowing for easier and more efficient work and collaboration;
- reversible decisions, as the data world is unpredictable, committing to a decision may lead to irreversible outcomes. Unless the outcome is certain, definitive decisions should not be made;
- security, as data may hold sensitive information, if not protected properly, can lead to disastrous outcomes. As such, access control, data modeling, and monitoring are some of the necessary precautions when designing systems;
- FinOps, fiance operations enable companies to obtain the best possible value from using cloud-based solutions by combining DevOps culture with finances;

Types of Architectures and Use Cases

There are several types of data architecture, each with its characteristics and use cases. Data architectures also come and go because the data world is dynamic and unpredictable.

Following are some predominant architectures, accompanied by implementation and their use cases.

Lambda Architecture is an architecture for data processing systems. It provides both low-latency and high-throughput data processing by combining batch and real-time data processing, being able to handle larger amounts of structured and unstructured data. [28].

The architecture is comprised of three layers, the batch layer, the streaming layer, and the speed layer [28] [29] [30].

- batch layer, commonly implemented with Apache Hadoop, processes batches of data through pre-computing;
- serving layer, commonly implemented with Apache Cassandra and Apache Hbase, stores and serves data from the processing layer;
- speed layer, commonly implemented with Apache Kafka or Apache Flink, processes and serves data in real time. Served data can also be from the storage layer;

The batch and storage layer work in tandem, while the speed layer works alone. This allows the separation of data processing according to needs.

Figure 2.3 illustrates the Lambda architecture.

It is important to denote that common critiques focus on the potential maintenance footprint, and inherent complexity of layers with duplicated functionality [30].

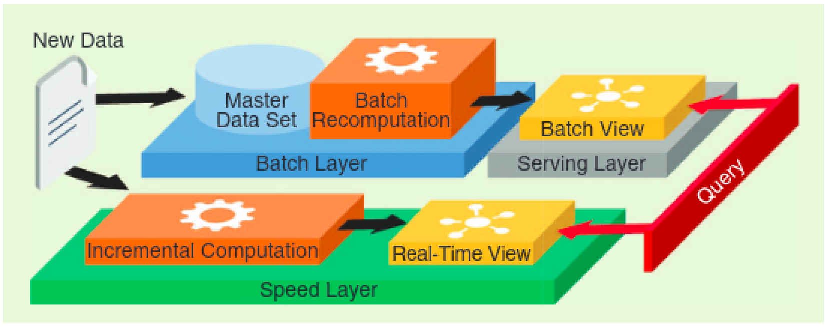


Figure 2.3: Lambda Architecture Diagram [30]

Kappa architecture is a Lambda architecture variation that aims to simplify the data processing pipeline by eliminating the need for separate batch, and real-time processing layers [31].

In Kappa Architecture, all data is processed in a single streaming system rather than separated [28] [29] [30]. The data is ingested and processed in real-time and stored in an append-only manner, allowing the same data to be used for real-time and batch processing, eliminating the need for separate data storage and processing systems.

The role of streaming computation is commonly performed by tools like Amazon Kinesis, Apache Spark, and Kafka Streams, to name a few. The role of storing data is performed by tools like Amazon Quantum Ledger Database, Apache Kafka, and Apache Pulsar, to name a few [31].

Important to denote that high costs can be associated if high volume and network capacity are needed for processing data [28].

Figure 2.4 illustrates the Kappa architecture.

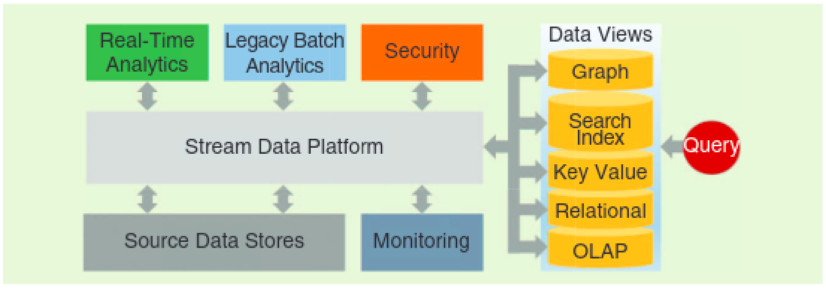


Figure 2.4: Kappa Architecture Diagram [30]

Event-driven architecture is a type of data processing system architecture that enables the processing of eventual data streams in real-time. This architecture encompasses scalability, loose coupling, asynchronous eventing, and fault tolerance. It is composed of three core concepts [32]:

- event producers, responsible for generating or detecting events and transmitting them to managers;
- event managers, act as middleware and are responsible for asynchronous filtering, processing, and routing of received events;

- event consumers, which receive events and act upon them;

Figure 2.5 illustrates a generic event-driven architecture.

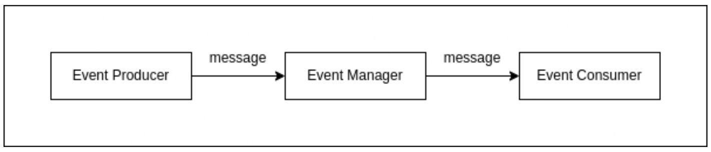


Figure 2.5: Generic Event-Driven Architecture Diagram

There are two similar yet different approaches to event-driven architecture: the message queue and the event-streaming platform. The difference lies in the event manager component [9] [33].

The message queue implements a First In First Out architecture for messages. It routes messages and guarantees they are delivered to some degree [9]. RabbitMQ, Redis, and ActiveMQ are commonly used message queues [34].

Figure 2.6 illustrates the message queue's event manager.

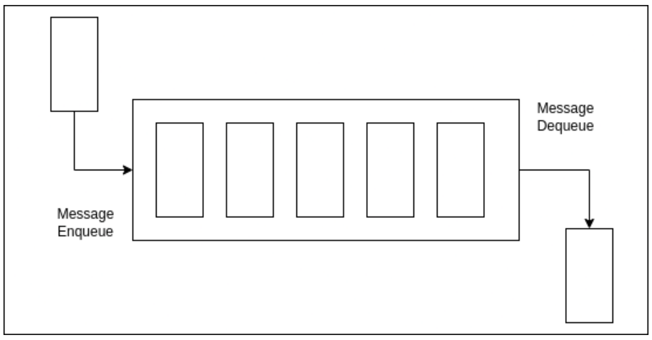


Figure 2.6: Event Manager's Message Queue Architecture Diagram

The event-streaming platform ingests and processes data in an ordered log of records. Its implementation relies on topics, a collection of related events, and stream partitions, division of a stream into multiple streams [9] [32]. Apache Kafka, Apache Pulsar, and AWS Kinesis are commonly used event-streaming platforms [34].

Figure 2.7 illustrates the event-streaming platform's event manager.

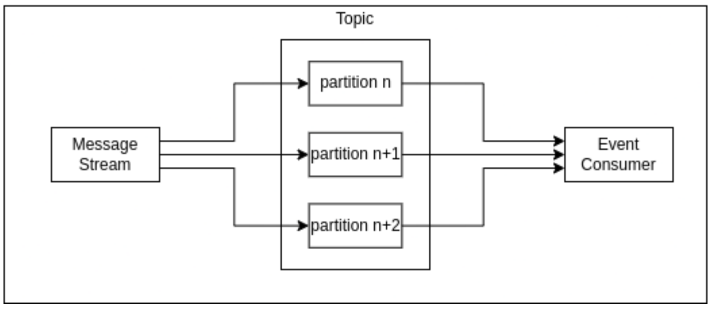


Figure 2.7: Event Manager's Event-Streaming Platform Architecture Diagram

Modern Data Stack refers to the technology and tools commonly used to collect, store, process, and analyze data in modern data architecture. It typically includes a combination of open-source and proprietary cloud-based tools and technologies where each technology holds a specific role in the infrastructure [35] [36].

The stack provides many benefits in the technical and business fields. Characteristics like cloud-based, ease of use, modularity, and tool usage flexibility reflect some technical advantages.

The modern data stack possesses a modular architecture composed of off-the-shelf tools that focus on automating data processing to drive business decisions. It bolsters high performance and low costs due to being cloud-based [37] [36].

The main functionalities of the modern data stack revolve around: ingesting, storing, transforming, data analysis, and data governance [36] [37] [35]. As the architecture is modular, each tool performs a functionality. Listing some of the most common tools for each functionality [38]: Fivetran, Airbyte, and Twillio Segment for ingestion, Snowflake, Databricks, and Amazon Redshift for storage, Apache Airflow, dbt, and LookMl for transformation, Tableau, Looker, and Hex for analysis, Atlan, Datafold, and Monte Carlo for governance.

Due to its characteristics, this architecture provides many benefits, such as an operating model focused on business, tool flexibility, operational business intelligence and AI, and data governance [37] [35].

Figure 2.8 illustrates the generic architecture of a modern data stack.

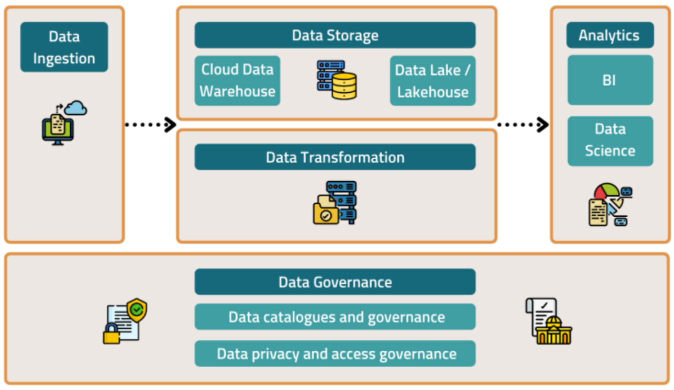


Figure 2.8: Modern Data Stack Architecture Diagram [37]

Data Hub is a type of infrastructure that serves as a unified entry point to multiple data sources [39] [40]. On a technical level, it is not a single tool but a group that makes it up.

Data hubs are commonly mentioned, along with data lakes and data warehouses. However, as similar as the context may be, they differ in terms of functionality and purpose. As mentioned in 2.1.2, a data lake and a data warehouse are centralized repositories. One serves processed data for analytical jobs, while the other serves data for processing jobs. However, the purpose of a data hub is to unify data storage through data integration, and orchestration with existing data sources [39].

Based on *Data hub purpose and architecture overview | AltexSoft* [39], a data hub comprises five layers: the source system, data integration, data storage, data access, and orchestration.

Following what [40] said about the critical components of the data hub are the data storage, integration, and access since they provide the desired functionalities.

Some of the most common tools for data integration are Apache Kafka, Azure Data Factory, and AWS Glue. OrientDB, ArangoDB, and Apache Ignite are the most common for data integration, while data access depends on databases' respective APIs or Apache Kafka [39].

Figure 2.9 illustrates a data hub:

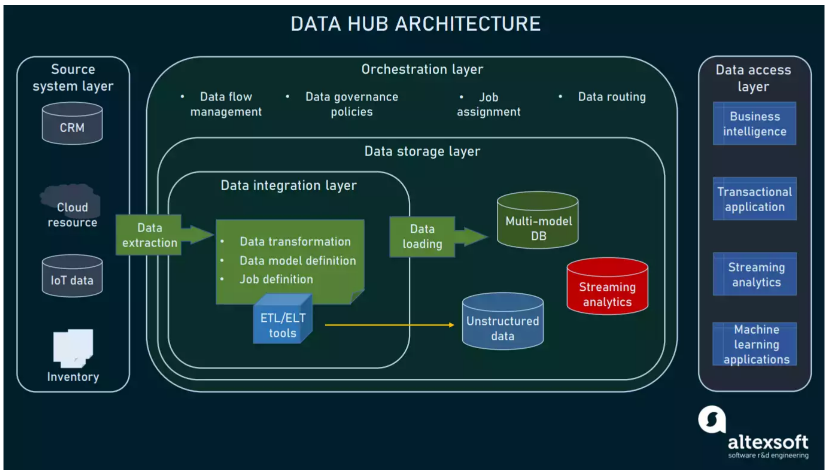


Figure 2.9: Data Hub Architecture Diagram [39]

Data Fabric is a type of data infrastructure that seeks to unify how data is stored, accessed, and processed across the system. This unification is made possible through the creation of an abstraction layer [41] that handles all technical details. Important to denote that a data fabric is not a single tool but a group of technologies, namely machine learning, micro-services, cloud solutions, orchestration, and virtualization [42] [43].

To ensure data fabric delivers the intended value, Gupta [44] defines some key considerations to ensure the data fabric has the intended value. Collecting and analyzing all forms of metadata, converting passive to active metadata, creating and curating knowledge graphs, and possessing robust data integration are critical for a data fabric.

Since the data fabric does not have a specific implementation, it has at least a defined group of components, them being a data catalog and knowledge graph for data aggregation and organization, metadata activation for data governance and security, a recommendation engine to provide better analysis, data delivery enables data usage, and Orchestration handles all transformation, integration, and cleansing work [42].

Figure 2.10 illustrates the generic architecture of a data fabric.

Data Mesh is a rather abstract concept. It focuses on averting centralizations, like data lakes and data warehouses, by dividing a system into small, independently-deployable data services owned and operated by small, cross-functional teams.

This approach allows for a more flexible and scalable architecture, as teams can evolve their services independently and at their own pace. It also promotes autonomy, ownership,

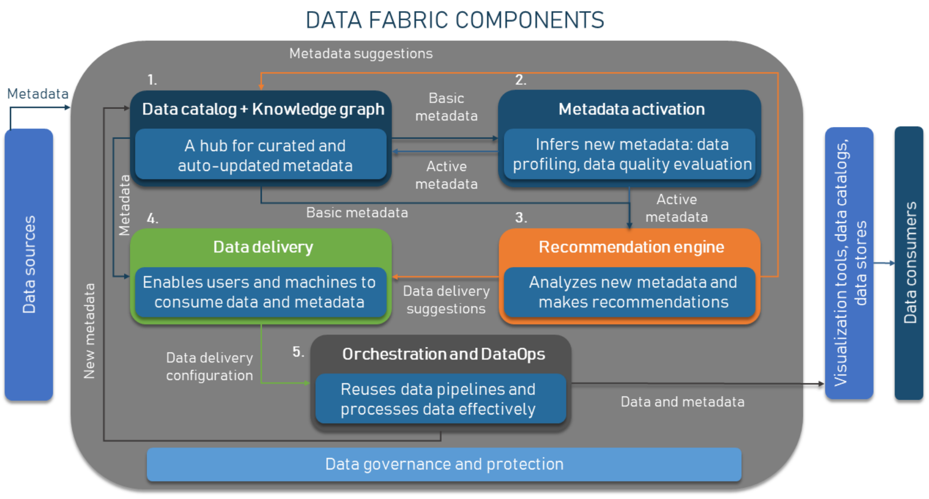

DATA FABRIC COMPONENTS
Metadata suggestions
Metadata
1.
Data catalog + Knowledge graph
A hub for curated and auto-updated metadata
Basic metadata
Active metadata
2.
Metadata activation
Infers new metadata: data profiling, data quality evaluation
Basic metadata
Active metadata
Metadata
4.
Data delivery
Enables users and machines to consume data and metadata
Data delivery suggestions
3.
Recommendation engine
Analyzes new metadata and makes recommendations
Data sources
New metadata
Data delivery configuration
5.
Orchestration and DataOps
Reuses data pipelines and processes data effectively
Data and metadata
Visualization tools, data catalogs, data stores
Data consumers
Data governance and protection

Figure 2.10: Data Fabric Architecture Diagram [42]

and self-service for data access and management, which can help reduce the dependency on centralized teams and improve the overall system's performance [45].

Dehghani [46] defines a set of principles that define a data mesh. These principles are:

- decentralized ownership by domain, teams own, provision, and govern data related to a specific business domain. This allows for team independence and flexibility;
- data as a product, treating data as a valuable and marketable asset enforces data quality and usability, allowing companies to use regular data beyond their normal purpose;
- self-serve data infrastructure as a service, to aid teams with their product while abstracting from technological complexity;
- federated computation governance, as the previous three principles need to work seamlessly, a cross-domain committee assures that standards are met;

Important to denote that a data mesh is an architecture that is still new, and since it focuses more on the business field instead of the technical one, there are still no commonly used tools.

Figure 2.11 shows the interactions between each principle.

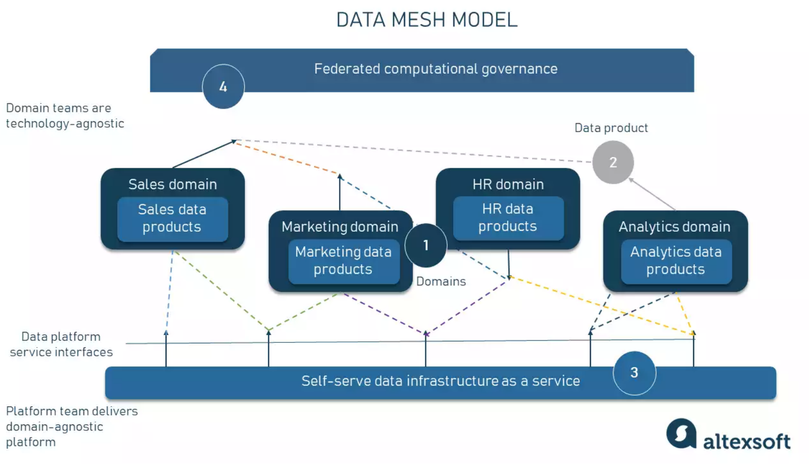

DATA MESH MODEL
Federated computational governance
4
Domain teams are technology-agnostic
Data product
2
Sales domain
Sales data products
Marketing domain
Marketing data products
1
Domains
HR domain
HR data products
Analytics domain
Analytics data products
Data platform service interfaces
Self-serve data infrastructure as a service
3
Platform team delivers domain-agnostic platform
altexsoft

Figure 2.11: Data Mesh Architecture Diagram [45]

2.2 Machine Learning Operations

Before addressing what MLOps is and what it stands for, it is first necessary to understand where it comes from and what it tries to solve. Only after contextualizing MLOps and understanding what it provides to help solve the problem at hand, will focus be given to the use and practice of MLOps.

MLOps aims to solve a common problem in the ML community, the need for ML models to be adapted to production environments [47]. In some cases where companies have the resources, Data Scientists are responsible for manually managing ML workflows, although prone to problems due to human interaction.

The application of the DevOps methodology solves this question. Since DevOps focuses on delivering production-ready products in an automated way, the the introduction of DevOps facilitates the process of preparing and deploying ML products to the production environments of Data Engineering pipelines.

According to Kreuzberger *et al.* [47], MLOps is an interdisciplinary development culture that combines a set of practices and concepts from ML, DevOps, and Data Engineering to deliver ML products in an automated, efficient, and reliable way to production environments. MLOps comprises three key aspects: principles, components, and roles.

Figure 2.12 demonstrates the intersections between each discipline.

Principles act as guidelines for correctly realizing things, these are:

1. CI/CD automation, through automation of integration, deployment, and delivery processes, it becomes faster to receive feedback on specific steps, thus increasing overall productivity;
2. Workflow orchestration, focuses on the automation and scheduling of tasks, limiting manual labor and increasing the speed of tasks;
3. reproducibility, repeating ML tasks to achieve the same results;
4. versioning, allows version control of data, code, and model to provide reproducibility and traceability;

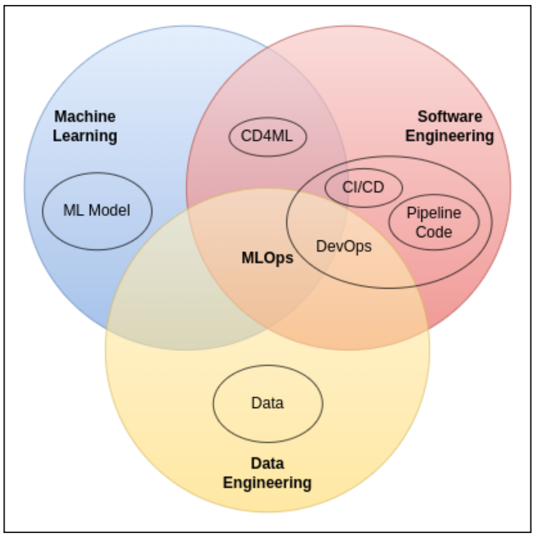


Figure 2.12: Venn diagram of MLOps' base disciplines. Adapted from [47]

5. collaboration, collaborative work on data, code, and models between teams to promote a better work culture and communication;
6. continuous ML training and evaluation, regular training of the model with new data promotes quality in tandem with data and model quality checks;
7. ML metadata tracking/logging, allows tracking of model metadata, providing meaningful insights (traceability) on model quality and characteristics;
8. continuous monitoring, enables error detection and prevention through cyclic assessments to data, models, code, infrastructure resources, and model serving performance;
9. feedback loops, allow for integration of obtained insights into the pipeline;

Components provide the necessary functionalities to the pipelines, while ensuring at least one principle. These are as follow:

- CI/CD, reflects principles 1, 6, and 9. The tools commonly chosen for CI/CD are GitHub Actions, Gitlab CI/CD, and Jenkins [48] [49] [50];
- source code repository, reflect principles 4 and 6. Common choices of tools are GitHub, Gitlab, and Gitea[48] [51] [52];
- workflow orchestration, reflects principles 2, 3, and 6. Usual tools for workflow automation are Apache Airflow, Kubeflow Pipelines, and AWS SageMaker Pipelines [51] [52] [53];
- feature store system, reflect principles 3 and 4. Common choices of tools are Google feast, AWS Feature Store, and Tecton.ai [51] [52] [53];
- model training infrastructure, reflects principle 6. Tools commonly chosen are Kubernetes and Red Hat Openshift [49] [52];
- model registry, reflects principles 3 and 4. Common choices are AWS SageMaker Model Registry, Microsoft Azure ML Model Registry, and Neptune.ai [48] [53];
- ML metadata store, reflects principles 4 and 7. Predominant choices are Kubeflow Pipelines, AWS SageMaker Pipelines, and Azure ML [49] [50];

- model serving, reflects principle 1. Tools commonly chosen are Microsoft Azure ML REST API, AWS SageMaker Endpoints, and Google Vertex AI prediction service [50] [53];
- monitoring, reflects principle 8 and 9. The usual choices of tools are Prometheus, Grafana, and TensorBoard [48] [50];

Roles represent specific tasks handled by persons that interacts with the pipeline, these are as follow:

1. business stakeholder, responsible for defining the goal of the project and handle communication of the business;
2. solution architect, responsible for designing the architecture and defining the technologies composing the pipeline;
3. data scientist, responsible for creating a solution to the problem through data and ml, while leveraging performance and choosing the best hyper-parameters;
4. data engineer, responsible for building and managing data processing and feature engineering pipelines, while also handling databases;
5. software engineer, responsible for ensuring software design patterns to the ML model in order to prepare the model for production environments;
6. devops engineer, responsible for bridging development and operations while automating CI/CD and orchestration;
7. ML/MLOps engineer, is responsible for bridging every role thus working along every role and being present in the whole pipeline;

Figure 2.13 demonstrates the interactions between each role in the MLOps paradigm.

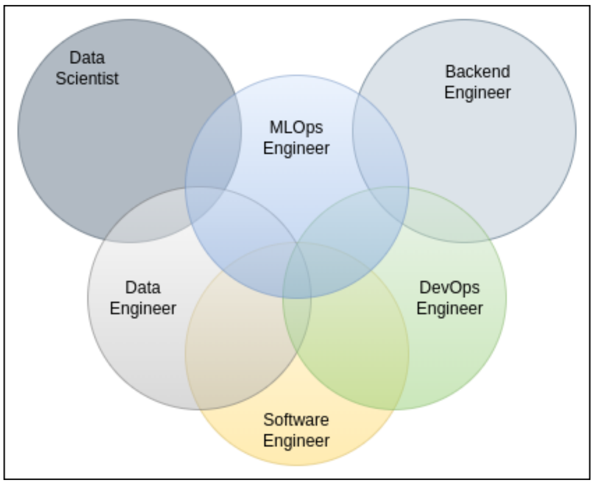


Figure 2.13: Venn diagram of roles and intersections, contributing to the MLOps paradigm. Adapted from [47]

Figure 2.14 represents an ideal workflow encompassing every principle, component, and role [47]. The workflow provides a functional end-to-end and step-by-step template that any company can use, and its choice of components effectively demonstrates every crucial phase and its interactions.

The ideal pipeline is composed of phases A, B1, B2, C, and D.

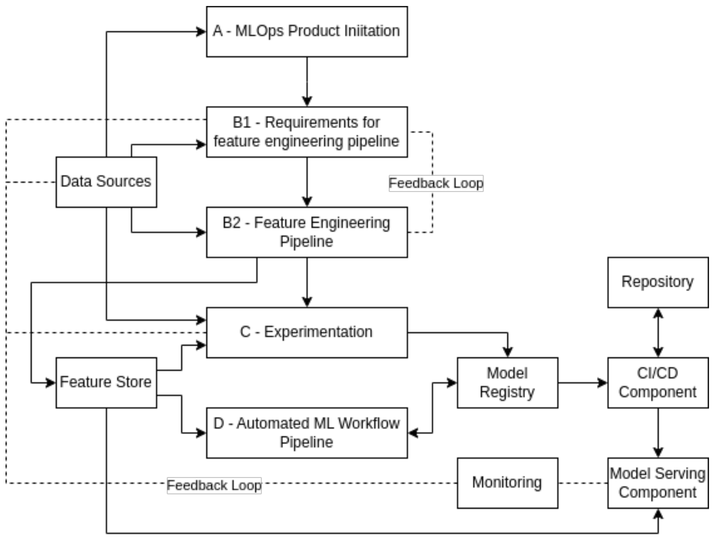

A - MLOps Product Iniitation
B1 - Requirements for feature engineering pipeline
Data Sources
Feedback Loop
B2 - Feature Engineering Pipeline
Repository
C - Experimentation
Feature Store
Model Registry
CI/CD Component
D - Automated ML Workflow Pipeline
Feedback Loop
Monitoring
Model Serving Component

Figure 2.14: Ideal MLOps pipeline. Adapted from [47]

A represents the product initiation phase, where the groundwork is laid for subsequent phases. The business stakeholder 1 analyzes the problem at hand and derives a solution through ML, the solution architect 2 designs the ML system choosing its components, and the data scientist 3 formulates the ML problem choosing either regression or classification, finally the data engineer 4 and data scientist 3 work in tandem to understand and verify the data at hand.

B1 represents the requirements for the feature engineering pipeline, which enhances the quality and effectiveness of the features used during model training and prediction. The data engineer 4 establishes data transformation rules to transform raw data into usable data. The data scientist 2 and the data engineer 4 then work in tandem to define feature engineering rules to reach better and improved features. Feature engineering rules pass through an iterative process to reach the best value.

B2 represents the feature engineering pipeline, using the model to act upon received/extracted data to calculate new features and store them in a database. The data engineer 4 and software engineer 5 use the previously defined requirements from phase A to choose components and build the pipeline. One of the foundational requirements is code for CI/CD and orchestration, both handled by the Data Engineer. The pipeline passes through an iterative improvement process to further hone the results.

C represents the experimentation phase, where the model and features are tested and fine-tuned to obtain the best predictions/classifications possible. Most of the tasks are led by the data scientist 3 with the support of the software engineer 5. These iterative tasks revolve around adjustments to data, models, parameters, and code until the model presents the desired performance. The model is then committed to a repository by the data scientist 3 and later reviewed by the devops engineer 6 and ml engineer 7 to deploy it to the automated ML pipeline through CI/CD after automated testing.

D represents the automated ML pipeline, where tasks are automated to refine the model until it is ready for production environments. The devops 6 and ml 7 engineers maintain the pipeline through metric collection and logging, ensuring tasks are executed as intended. When the pipeline is triggered, tasks are executed in the following order: extract data for preparation and validation, model training and validation, model export, and store it in a model registry. After the model push to the registry, CI/CD pipelines are triggered to deploy the model to a production environment under the supervision of either a DevOps or ML Engineer.

While MLOps tries to solve the problems of adapting ML products to production environments, there are some challenges that need to be tackled [47]. These can be categorized as the following:

- organizational challenges, which originate from the need for more skilled professionals proficient in MLOps. Data Scientists can only cover a fraction of those skills, implying a need for additional individuals to cover the remaining skills, thus leading towards a multi-disciplinary team. Another important aspect is the communication between the individuals. Since each discipline has its terminology and area of expertise, some shared knowledge ground is necessary to ensure an acceptable degree of communication and the whole group strives in the same direction;
- ML system challenges, which are directly related to model training and serving, more precisely, resource provisioning for training and serving. Model training is challenging due to the dynamic volume of data, leading to more complex estimates for the ideal hardware. Model serving implies a necessity for flexible and scalable infrastructure;
- operational challenges, which mainly originate from the selection and interaction of hardware and software choices, which can become more complex if managed manually. Automation and monitoring of the pipeline in conjunction with versioning and governance of created artifacts lead to robustness and reproducibility of the whole system. It effectively reduces the chance of problems and provides the necessary tools to aid in solving one if it happens;

2.3 Related Work

There are some third party implementations of data engineering pipelines that reflect current approaches to processing data. These implementations can range from more local systems to cloud systems, thus demonstrating that a data processing system can be built in multiple different environments with a variety of different tools.

Important to denote that the finality of these systems can change since their objective correlates to the type of data processing executed. However, what makes these systems relatable to each other is the features they provide, which effectively classify them as a data engineering pipeline.

Baranda *et al.* [54] build a system to be used as a service for Service Level Agreement (SLA) management of a Digital Twin Virtual Network Service. While their system is complex regarding many aspect, focus will only be given to the data processing part of the system.

Data processing is handled by a YARN[1] cluster along HDFS and multiple ML libraries such as Spark ML[2], BigDL[3], and Ray[4].

Mäkinen [55] provides a generic MLOps pipeline that is Kubernetes[5] based cloud-native and production ready environment for CI/CD and monitoring of machine learning systems. The system is composed of three different pipelines, one for ETL of data, another for training and fine tuning of models, and a final one for serving the trained model. As the system is intended to be generic, future models to be executed are assumed by the author to be in the format of Docker[6] images.

Li and Zou [56] build a system entirely on the AWS cloud to serve as a data engineering pipeline. The pipeline receives data from multiple Internet of Things (IoT) devices that store data in a local MySQL[7] database to later be queried an IoT Core service that sends data to an S3[8] service bucket. A IoT Analytics[9] service then loads data from the S3 bucket into a DynamoDB[10] service instance to later be queried by a SageMaker[11] service to act upon it through the use of ML models. If there is any anomaly detected, a Lambda[12] service instance sends an email with the anomaly data.

[1]https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html
[2]https://spark.apache.org/docs/latest/ml-guide.html
[3]https://www.intel.com/content/www/us/en/developer/tools/bigdl/overview.html
[4]https://www.ray.io/
[5]https://kubernetes.io/
[6]https://www.docker.com/
[7]https://www.mysql.com/
[8]https://aws.amazon.com/s3/
[9]https://aws.amazon.com/iot-analytics/
[10]https://aws.amazon.com/dynamodb/
[11]https://aws.amazon.com/sagemaker/
[12]https://aws.amazon.com/lambda/

CHAPTER 3

Business Scenario

The business scenario aims to detail the problem in question and the environment in which it is inserted. In the context there is a problem that needs to be solved. A solution is proposed, planned, and analyzed in order to solve the problem at hand in the best way possible. This chapter will approach the following aspects that contextualize this dissertation. These are:

- business scenario, the context where the main objective revolves around and what it aims to solve;
- main objective, characteristics of the solution in order to solve the problem in the best way possible;
- business use cases, how the solution will be used in order to solve the problem at hand;

3.1 Context

Making judgments on product offerings and price strategies can be difficult, particularly in highly competitive e-commerce marketplaces where decisions dictate the future of an organization. Manual pricing tracking and analysis are labor-intensive, error-prone methods that prevent organizations from keeping track of what their rivals are doing.

Ritain.io, an organization that focuses on automation and digital transformation in order to help business keep up with next technologies, provides a PaaS denominated ZenPrice^{TM}, a competitive intelligence pricing service that provides real-time market data collecting and analysis with the help of AI/ML algorithms. By helping businesses make quicker and more informed pricing decisions, they can better serve their clients and ensure that their strategies are being followed.

In the telecommunications sector, ZenPrice^{TM} helps organizations stay ahead of the competition and increase their agility by optimizing price and brand positioning. More specifically, some of the key situations that ZenPrice^{TM} can show it's value are:

- pricing optimization, as the main functionality is to predict prices with the least margin of error;

- competitive analysis, providing consistent monitoring of rival companies' product prices;
- brand positioning, by defining prices according to rival companies positioning, it becomes possible for companies to position themselves in the best way possible;
- market insights, it allows for observation of trends, costumer preferences, and market dynamic allowing for a better adaptation of the market's dynamism;
- agility and decision making, through the leverage of real-time data and machine learning algorithms, which enables businesses to make faster and more informed pricing decisions;

3.2 System requirements

The main objective of this dissertation is the creation of a data processing pipeline that runs machine learning predictive models on-demand. This section will cover important aspects such as intended features, system constrains, and possible integrations.

As any well designed data engineering pipeline, the pipeline needs to have a high degree of data maneuverability. This maneuverability is reflected through the following features:

- data ingestion, the pipeline has to be able to ingest large amounts of data and in parallel;
- data storing, the pipeline has to be able to store data for indefinitely amounts of time;
- data processing, the pipeline has to be able to process data through a various ways;
- data serving, the pipeline has to be able to serve data to multiple clients in parallel;

Ritain.io specified some constrains by which the pipeline has to abide, which will impact the system requirements and shape the implementation of it. These can either be non-functional or functional requirements and are as such:

- distributed system;
- micro-service based;
- container based;
- cloud agnostic;
- based of the SMACK stack [57];
- integratable with multiple systems;
- serve data through a GraphQL API;

Taking into account the constraints previously mentioned, it is expectable for the the pipeline to have multiple concurrent users, which translates to multiple datasets being processed in parallel, at any possible time. Thus, the pipeline should be able to handle high data IO operations and processing in parallel while providing a high level of availability and fault tolerance.

In order to better define what the functional and non-functional requirements are, we can separate them as the following:

- non-functional requirements:
  - scalability, being able to scale according to user demand;
  - high availability, being available whenever the user needs to use the pipeline;
  - high throughput, being able to handle large amounts of data IO operations;

  - fault tolerance, being able to tolerate any error in case such happens, thus allowing the remaining of the system to continue execution;
  - data retention, being able to store data for unspecified amounts of time;
  - extensibility, being able to extend it's functionalities to accommodate new user needs;
- functional requirements:
  - ingest, process, and store unstructured data;
  - serve data through a GraphQL API;
  - distributed system;
  - container based;
  - cloud agnostic;
  - integratable with multiple systems;

Taking everything into consideration, the pipeline should be able to handle Big Data needs while also ensuring the user's needs are met. This includes ingestion, processing, and storing of structured and unstructured data according to user specifications in order to ensure the pipeline's effectiveness and reliability. The pipeline's additional characteristics, such as scalability, high availability, fault tolerance, and extensibility, only add value to it by allowing it to handle more workload in parallel while simultaneously handling and adapting to new trends in the dynamic world of Big Data.

CHAPTER 4

Design and Implementation

The process of designing and implementing the pipeline will be the main focus of this section. The entire process will be approached in broader terms, followed by addressing each component separately, and finally focusing on the pipeline itself as a whole and deployment environments.

4.1 General Overview

When designing the pipeline, multiple considerations were taken into account, which are mainly related to the defined requirements 3.2 and the data engineering lifecycle 2.1.2. In more general terms, the defined requirements mainly affect the type of architecture used for the pipeline, these being a micro-service based architecture using a containerization and orchestration technologies to run and manage each service, while the data engineering lifecycle relates each micro service to a different phase of the data engineering lifecycle.

Figure 4.1 represents a generic architecture of the pipeline. It is separated into multiple parts, each corresponding to a specific phase of the data engineering lifecycle and a key functionality.

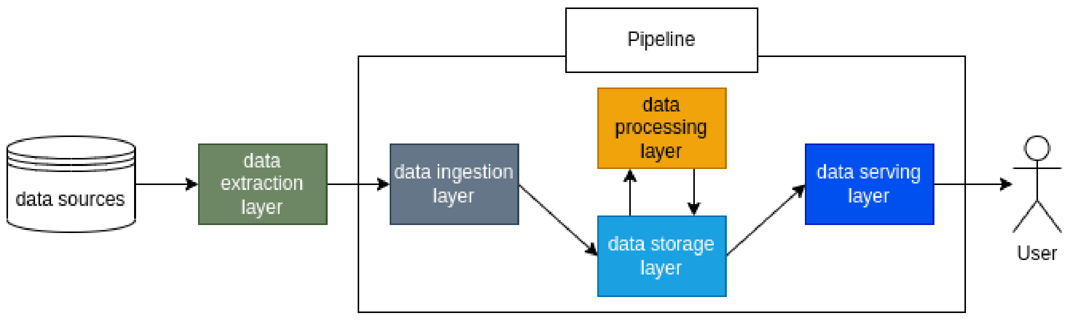


Figure 4.1: General Architecture Diagram

Going into further detail for each part present in architecture:

- data sources represent any possible data generating system. Not exactly part of the pipeline, but created data will be processed in the pipeline. Data sources can vary in type such as a IoT device always exporting measures, a database which a client can query to then send to the pipeline, or even a file that contains values.

- data extraction layer, responsible for extracting data from a data source and moving it into the pipeline. Some level of rearrange to the data may be done if needed, although minimal as the main data processing task is done in the next phase;
- data ingestion layer, responsible for introducing data into the pipeline. Needs to ingest large amounts of data and scale according to current throughput. Some level of data retention capacity, while not necessary, is valued;
- data processing layer, responsible for processing and rearranging data from the original to the desired structure. Models will then be trained on the data and predict the next value. After data processing and model prediction, obtained results are sent to the next phase to be stored;
- data storing layer, responsible for receiving and storing data from the data processing layer. As data can vary in multiple aspects like size, structure, accessibility, availability, and many more, this layer should be able to handle all these aspects. While this phase can interchange order with the data extraction and data processing layer, in this case, data storing will be the last of these 3;
- data serving layer, responsible for providing the data it is tasked to retrieve. It works on top of the data storing layer, and should be able to integrate well with it in order to provided the requested data as fast as possible;

Similarly to ZenPrice^{TM} [4], the pipeline is a new approach to a data enginnering pipeline from a different perspective. One that provides additional benefits such as scalability, fault tolerance, and extensibility, which allows it to adapt to new trends and handle larger amounts of work in parallel.

4.2 Implementation of Components

The pipeline is composed of several components, each responsible for providing a specific functionality to the pipeline. While some implement a specific functionality, others provide the required environment or some secondary functionality in order to achieve the desired main functionalities.

As such, each component will be addressed in detail regarding it's role and configuration. Starting by the deployment environment, and then proceeding with each part, which corresponds to a specific feature.

4.2.1 Deployment Strategy

Taking into account the system requirements 3.2, the deployment strategy is intended to favor environments that are cloud agnostic, are container-based, and can be managed as IaC. As such, the whole pipeline will be developed to a Kubernetes environment with Docker as container runtime.

Docker[1] is a platform that allows the construction, sharing, and execution of software in a container format. A container is an Operating System level virtualization option that

[1] `https://www.docker.com/`

creates an isolated space with only the necessary dependencies, configurations, and libraries for a program to run separated from the rest of the system.

Docker enables a controlled and portable environment, effectively providing reproducibility in any system as long as Docker is installed.

Taking into account the characteristics of Docker, the development and testing of applications becomes simpler since it provides reproducibility, flexibility, and dynamism that speed up the development cycle of applications and reduce the chance of possible mistakes.

When handling multiple containers at the same time, Docker Compose[2] is a tool that comes integrated with Docker and provides a way to handle application stacks consistently.

As such, Docker and Docker Compose will be the platform of choice for implementing and deploying our micro-services. It will enable a controlled and simplified way of creating, testing and deploying our micro-services.

Kubernetes[3] is a platform that orchestrates containers in a cluster. It operates on top of container technology to provide availability, scalability, and management of groups of containers.

Kubernetes works as a cluster, a cluster composed of a control pane, responsible for managing all the tasks inside the cluster, and a group of either virtual or physical nodes, where container workloads are executed. Containers are encapsulated into Pods, the smallest resource of Kubernetes, and scheduled by the control pane to be executed on available nodes. Effectively, Kubernetes provides an environment that allows for container deployment, scaling, and management while abstracting the underlying infrastructure.

As Kubernetes also provides an API for controlling containerized applications and their resources, developers can benefit from the management of network and storage resources, secret and configuration files, while also benefiting from fault tolerance, high availability, and on-demand scaling.

There are many distributions of Kubernetes, each with a specific objectives which can range from a cloud production grade environment, to local testing environment. Two of these distributions are **Minikube**[4] and **Elastic Kubernetes Service (EKS)**[5], both of which will be addressed further ahead.

As such, Kubernetes will be the choice for managing our micro-services in a controlled and provisioned environment, while making sure scalability and availability are maintained. Further details regarding the implementation will be discussed further ahead.

4.2.2 Data Ingestion Layer

The data ingestion layer is responsible for ingesting data. It has to be able to scale according to demand and handle multiple producers and consumers. Decoupling between producers and consumers is also desirable, for such, some level of data retention facilitates

[2]`https://docs.docker.com/compose/`
[3]`https://kubernetes.io/`
[4]`https://minikube.sigs.k8s.io/docs/`
[5]`https://aws.amazon.com/eks/`

this aspect. Due to these aspects, the choice for implementing the data ingestion layer is Kafka, Zookeeper, and Strimzi.

Apache Kafka is a streaming event platform built for data processing pipelines and streaming applications. It is designed to handle large-scale, high-throughput, and low-latency data streams, while providing scalability, fault tolerance, and data retention and replication to some degree. Besides the vantages it provides to the pipeline, what makes it valuable in a data processing pipeline is the decoupling it provides between data producers and consumers, making the data writing/reading independently from each other.

Kafka can scale by forming clusters, composed of Kafka instances called "brokers", and **Apache Zookeeper** used for distributed systems' synchronization.

Due to its scalable and flexible nature in handling multiple producers and consumers in parallel, Kafka is the choice for implementing the data ingestion layer.

Strimzi is a Kubernetes operator. It simplifies the deployment and management of Kafka and Zookeeper on Kubernetes, leveraging Kubernetes' native features and providing a declarative approach to managing Kafka and Zookeeper resources. It enables users to seamlessly run Kafka in a cloud-native environment, benefiting from Kubernetes' scalability, resilience, and ecosystem of tools.

For the cluster to achieve high throughput while making sure no data is lost when ingesting data, the ingestion layer needs to work in tandem with the data generation layer. As such, producers, brokers, and topics need to be configured properly.

Code snippet 1 demonstrates a broker configuration that aims to ensure a balance between high availability and high throughput while providing data integrity and minimizing data loss. For a cluster composed by 3 brokers, each using eight threads for IO operations and three threads for network operations, this will ensure at least 2 brokers are always synchronized when replicating data across all three.

```yaml
config:
    offsets.topic.replication.factor: 3
    transaction.state.log.replication.factor: 3
    transaction.state.log.min.isr: 2
    default.replication.factor: 3
    num.partitions: 2
    min.insync.replicas: 2
    inter.broker.protocol.version: '3.4'
    num.network.threads: 3
    num.io.threads: 8
    socket.send.buffer.bytes: 102400
    socket.receive.buffer.bytes: 102400
```

Code 1: Broker configuration defined in YAML format

The reasoning behind this configuration it that it prioritizes data durability and high availability via replication, in-sync replicas, and specific partitioning strategies. It also takes performance into account by establishing thread amount and socket buffer sizes. The selection of replication factors and the minimum number of in-sync replicas are critical for fault tolerance and data integrity.

Code snippet 2 demonstrates a configuration for a topic, which focuses on providing high availability while providing some level of data retention. While many of these configurations can be configured by the broker, it is possible to create topics with custom configurations which differ from the broker's default.

```yaml
spec:
    partitions: 2
    replicas: 3
    config:
        cleanup.policy: delete
        compression.type: uncompressed
        retention.ms: 7200000
        segment.bytes: 1073741824
```

Code 2: Topic configuration defined in YAML format

The reasoning behind this topic configuration is that it provides data replication across 3 brokers and parallelism for IO operations through the two partitions. Data is stored in an uncompressed state with a retention period of 2 hours, which allows for a faster data transference. The segment size defines the size of each log segment, impacting disk usage and segment management, the default value was used.

4.2.3 Data Generation Layer

The data generation layer is responsible for sending data to the pipeline, wether created or exported from a file or a databse, for it to be later processed and stored. The component handling this task is denominated **Producer** and it's intended purpose is to read data from a file and sending the content to the pipeline.

While this layer is not exactly part of the pipeline, it is an important phase of the data engineering lifecyle 2.1.2. As such, in order for the pipeline to work as intended, data must be present, and the Producer is the component responsible for this task.

The layer will be implemented with Python[6], a general purpose programming language that provides a plethora of libraries which provide additional functionalities like data management, database drivers, and many other depending on the use case. For this specific case, the Producer will be using the **pandas library**[7] for handling data from the file and **confluent-kafka library**[8] for sending the data to the data ingestion layer 4.2.2.

Important to denote that this component is custom built for this scenario in question. Any other different scenario would involve a different way to handle and send data.

Since this layer needs to work in tandem with the data ingestion layer, it needs to be configured appropriately to not have a negative impact. Code snippet 3 demonstrates a generic configuration for producers, which aims to ensure high availability and minimize data loss through properties like retries, idempotent messaging, and acknowledgement settings. Performance is leveraged by batching messages, being able to have up to 5 messages sent concurrently and waiting for acknowledgement. If batches do not reach the max size, the

[6]https://www.python.org/
[7]https://pandas.pydata.org/
[8]https://docs.confluent.io/kafka-clients/python/current/overview.html

producer may linger up to a predefined amount of time waiting for additional messages in order to increase the batch a bit more.

```
bootstrap.servers=192.168.67.6:9095
retries=2147483647
acks=all
enable.idempotence=true
max.in.flight.requests.per.connection=5
batch.size=16384
linger.ms=5
buffer.memory=33554432
```

Code 3: Producer configuration provided in INI format

4.2.4 Data Processing Layer

The data processing layer is responsible for processing the data inside the pipeline. It is responsible for establishing a connection between the data processing layer and the data storage layer. It interacts by querying for data, processing said data through the use of ML models, and storing obtained results back to the data storage layer. This component is denominated as **Producer**.

Selected tools for implementing the data processing layer are Apache Spark[9], Spark-On-K8s-Operator[10], and Python.

Apache Spark is a unified analytics engine for large-scale data operations. Spark presents functionalities for batch processing, stream processing, exploratory data analysis, and machine learning with high performance and scalability.

Spark is designed to run as a cluster of nodes, its structure implements the actor model, effectively allowing for tasks to be scheduled and executed in tandem while scaling with ease. This is made possible due to the fact that Spark is built on top of the HDFS and Resilient Distributed Dataset (RDD), which allow for distributed computation. Clusters are comprised of two types of nodes, master nodes which are responsible for delegating tasks for execution, and worker nodes which are responsible for executing the scheduled tasks.

Spark-on-Kubernetes-Operator allows users to leverage the benefits of Kubernetes for running Spark workloads. It provides an efficient and scalable way to run Apache Spark workloads on Kubernetes clusters. It simplifies the deployment process, takes advantage of Kubernetes' resource management capabilities, and enables seamless integration with other Kubernetes ecosystem tools. By using the operator, users can leverage the benefits of both Apache Spark and Kubernetes, combining the data processing capabilities of Spark with the scalability and flexibility of Kubernetes for their big data workloads.

Python, as it has been previously presented, is a programming language that provides, through a plethora of libraries, multiple functionalities, including the libraries used in the ML models and connecting to data storage layer.

While Spark-on-k8s-operator and Apache Spark were expected to implement the processing layer, due to limitations in DNS name resolution and the intended machine learning models

[9]`https://spark.apache.org/`
[10]`https://github.com/GoogleCloudPlatform/spark-on-k8s-operator`

not using the Spark machine learning library, these components were tested but not chosen for the final product. Combined with the fact that the models are implemented in Python with the tensorflow[11] and keras[12] libraries, Python proves to be the best choice, providing only the necessary functionalities for executing the models and establishing the connection between the data processing and data storage layers.

4.2.5 Data Storage Layer

The data ingestion layer is responsible for storing data, either structured, semi-structured, or unstructured, in an efficient way. It has to be able to scale according to data volume while providing data persistence, integrity, and replication.

Selected tools for implementing the data storage layer are Apache Cassandra, K8ssandra, and Python.

Apache Cassandra[13] is a NO-SQL, column-type database able to handle large amounts of data. It provides efficient writing and reading performance, can handle high writing throughput, and scale horizontally with ease.

Cassandra is able to create circular clusters, a circle-like architecture that provides high availability, fault tolerance, data replication, and data performance. The cluster is composed of two types of nodes, seed nodes which are responsible for bootstrapping new nodes into the cluster, and normal nodes which hold data. Inside the cluster, data is stored in tables, whose schema defines the layout. In turn, tables belong to a keyspace, a group of tables that abide by a common configuration denominated as replication strategy.

K8ssandra[14] simplifies the deployment and management of Cassandra on Kubernetes, allowing users to leverage the benefits of both technologies. It provides a streamlined approach to running Cassandra clusters, automates common operational tasks, integrates with popular monitoring tools, and offers additional features to enhance the Cassandra deployment experience. K8ssandra is well-suited for managing large-scale, distributed, and fault-tolerant Cassandra deployments in a Kubernetes environment.

Python, as it has been previously mentioned, will be used to implement the **Importer** component. This component is responsible for moving data from the data ingestion layer, rearrange data into the desired structure, and then store it in the database.

Taking into account Cassandra's characteristics which get along with cloud and micro-service environments, besides providing the wanted functionalities, Cassandra will be the choice for implementing the data storage layer. However, due to K8ssandra not being compatible with Kubernetes versions +1.25, its inability to work on single node clusters, and its additional functionalities which are not necessary for the pipeline, K8ssandra was not chosen along Cassandra.

The Cassandra cluster will be composed by a single datacenter, which in turn is composed by a single rack of three nodes. Knowing the architecture of the database cluster is necessary

[11]`https://www.tensorflow.org/learn`
[12]`https://keras.io/`
[13]`https://cassandra.apache.org/_/index.html`
[14]`https://k8ssandra.io/`

to configure Cassandra appropriately. Aspects like security, network, and replication. COde snippet 4 demonstrates the some key configurations:

```
authenticator: PasswordAuthenticator
authorizer: CassandraAuthorizer
role_manager: CassandraRoleManager
network_authorizer: AllowAllNetworkAuthorizer
seed_provider:
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
          - seeds: 'cassandra-0.cassandra.cassandra.svc.cluster.local'
ideal_consistency_level: LOCAL_QUORUM
```

Code 4: Cassandra configuration defining security, network, and replication aspects

These configurations are crucial for a Cassandra cluster's security, scalability, and consistency. Authentication and authorization mechanisms are required to secure and control database access, while the seed provider is necessary for new nodes to join the cluster. The consistency level is important as it impacts the consistency and availability of the cluster.

4.2.6 Data Serving Layer

The data serving layer is responsible for serving the stored data in the data storage layer. Data serving is made through queries while the data can be visualized through dynamic and customizable graphs, charts, tables, heatmaps, etc.

The tool chosen for implementing the data serving layer is Stargate.

Stargate[15] is a data gateway that provides a unified way of accessing the Apache Cassandra cluster through multiple APIs.

It presents a modular structure, composed of a coordinator node that integrates the Cassandra cluster, which does not hold data for faster access, and API nodes. Each API can be developed independently from the other, allowing for ease of deployment, integration, and scalability. Currently, there are five types of interactions with the Cassandra cluster, client drivers for direct interaction with the cluster, and four different APIs, REST, gRPC, Graphql, and Document, that communicate with the cluster through the coordinator.

Knowing the Stargate structure, which seems to interact well with cloud and micro-service-based environments, and the integration with Cassandra, Stargate will be the choice for implementing the data serving layer.

4.2.7 Monitoring

Monitoring is a crucial aspect of pipelines that allows for observation of executing processes, resource utilization, and anomaly detection through scrapping of metrics. These metrics facilitate prevention and handling of undesirable application states.

Tools chosen for implementing Monitoring are Prometheus, Prometheus Pushgateway, and Grafana.

[15] https://stargate.io/

Prometheus[16] is an open-source monitoring and alerting system. It integrates well with cloud and micro-service based environments, making it the go-to choice for monitoring.

It boasts capacities like: • dynamic service discover-ability which interacts well with micro-service based environments; • data collection through the pull model which scraps provided endpoints; • data model based on time-series to store and order metrics; • querying metrics through PromQL, Prometheus own query language for querying collected metrics; • alerting based on rules that act upon collected metrics; • visualization and graphing, although it is rather limited; • ease of integration with other tools that provide additional functionalities;

Prometheus is unable to scrape ephemeral jobs since it relies on constant endpoints. As a result, the **Prometheus Pushgateway**[17] offers a reliable endpoint for metrics aggregation where ephemeral jobs can transmit their metrics in order for Prometheus to scrape them.

Grafana[18] is an open-source data visualization and monitoring tool which is frequently used in conjunction with other data sources. It features: • interactive dashboards with rich visualizations like graphs, charts, tables, and heatmaps for dynamic and customizable data presentations; • integration with multiple data sources; • alerting based on rules; • extensibility, which allows for the addition of new features or the expansion of current ones through the use of plugins; • role-based access control (RBAC), which allows for regulated access;

Together, Prometheus and Prometheus Pushgateway will be the choice for monitoring the pipeline while Grafana will assist Prometheus in monitoring the pipeline as it connects nicely with already-existing data sources and offers extra visualization capabilities.

4.3 Pipeline Deployment

During the development of the pipeline, three deployments were made, each with a specific objective in mind. Each of these deployments have an objective and will be addressed in the following subsections. Additionally to the objectives , the problems faced and solutions found during the deployment of each version will be addressed. The deployments, along their respective objectives, are the following:

- Docker Compose deployment, a small PoC version that ensures components can be integrated in the pipeline and features work as intended;
- Minikube Deployment, a Kubernetes version of the pipeline that allows testing of the pipeline in a similar environment ;
- EKS deployment, a production grade Kubernetes version in the cloud. This will test the pipeline in a scenario equal to a

[16] `https://prometheus.io/`
[17] `https://github.com/prometheus/pushgateway`
[18] `https://grafana.com/`

4.3.1 Docker Proof of Concept

The pipeline's implementation with Docker and Docker-Compose[19] is a small-scale Proof of Concept (PoC). Building the pipeline in a Docker-Compose environment enables simplicity, reproducibility, portability, efficient resource utilization, and service isolation, everything necessary to have a small-scale micro-service environment.

Figure 4.2 provides an overview of the pipeline implemented in the docker-compose platform.

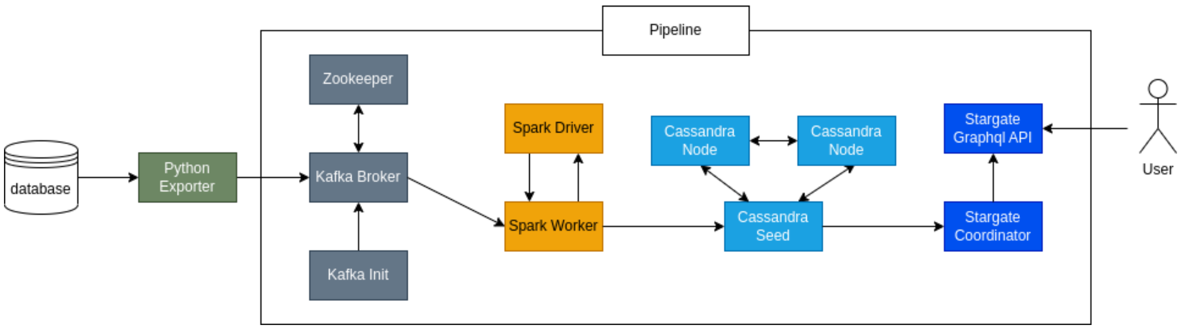


Figure 4.2: Docker Architecture Diagram

The deployment is composed of the following services:

- Apache Zookeeper coordinator for broker synchronization;
- Apache Kafka broker for ingesting data;
- Apache Kafka broker topic-initiator for topic creation;
- Apache Spark master for distributing tasks;
- Apache Spark worker for task execution;
- Apache Cassandra seed for cluster synchronization, data replication, and storage;
- Apache Cassandra node x2 for data replication and storage;
- Stargate Coordinator for API coordination;
- Stargate Graphql API for querying data to Cassandra through the coordinator;

During the development of the pipeline, some difficulties were found and addressed accordingly in order to overcome the situation. These difficulties relate to Docker container initialization, Kafka connection, Cassandra's node collisions, and Sparks DNS resolution.

Docker container initialization, docker considers the container ready when it starts running, even if the startup configuration of the container is on-going. This characteristic of Docker may lead to containers failing to connect to containers that are "running", yet still preparing for the main task. In order to prevent this, Docker provides a way to check if containers are ready for work called **healthchecks** [20]. Healthchecks periodically run a command as a check. It is run until successful execution or a specified timeout. Together with the **depends_on** [21] configuration, which allows for a docker-compose service to wait for another service until it is considered "healthy", it becomes possible to start services only when the dependent services are ready.

[19]`https://docs.docker.com/compose/`
[20]`https://docs.docker.com/compose/compose-file/compose-file-v3/#healthcheck`
[21]`https://docs.docker.com/compose/compose-file/compose-file-v3/#depends_on`

Kafka client connection, in a normal scenario, a Kafka client establishes a connection with a Kafka broker through a bootstrapping port, which is responsible for providing metadata about an actual address for data transfer to the client. This configuration is done through the property “advertised.listeners”, which holds all the broker’s addresses [58].

A monolithic connection between Kafka and a client can be seen in figure 4.3:

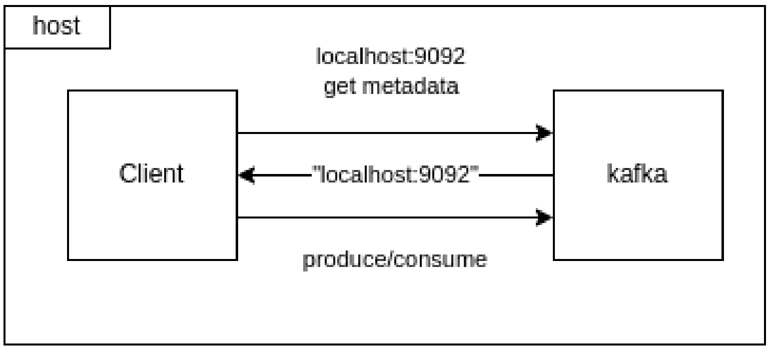


Figure 4.3: Kafka connection on same machine

However, due to the use of Docker for implementation of the pipeline in a micro-service oriented architecture, each component is isolated in containers, thus adding complexity in configuring addresses to establish connections. Connection through “localhost” is not possible. It is necessary specify the “advertised.listeners” with the container’s address 4.4.

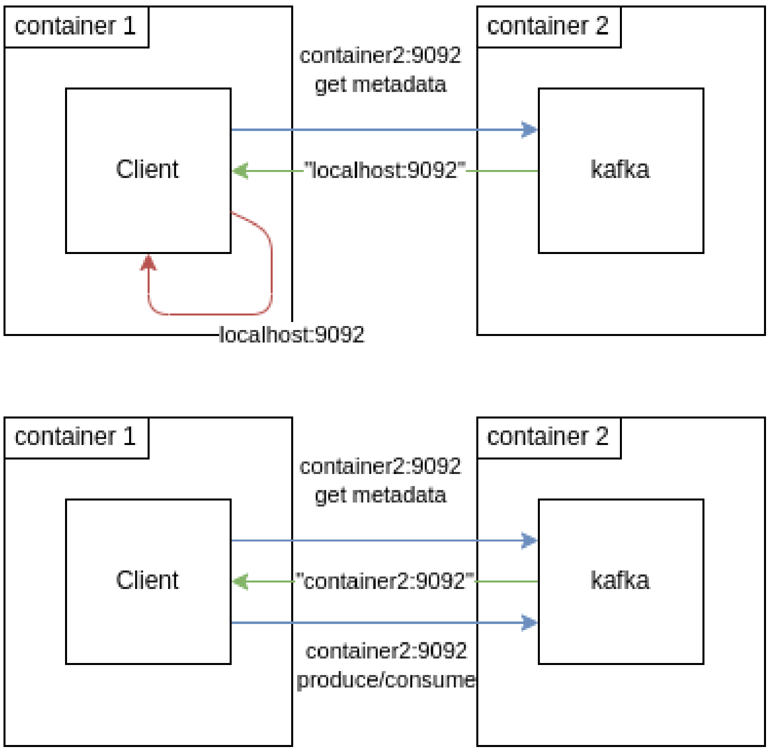


Figure 4.4: Kafka connection between containers

Another scenario is when testing the connections from the host into the Kafka container, thus turning the previous solution ineffective. An exemplification can be seen in 4.5.

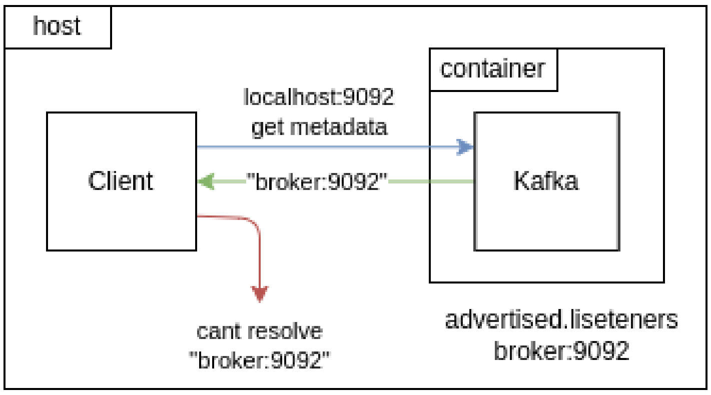


Figure 4.5: Kafka connection between host and container

In this situation, it becomes necessary to define two addresses in the "advertised.listeners" property, one for inter-container communication, and another for host to container communication. It also becomes necessary to configure a new "KAFKA_LISTENER_SECURITY_PROTOCOL_MAP" for Kafka, effectively distinguishing connections from the host and containers.

The result can be seen in figure 4.6.

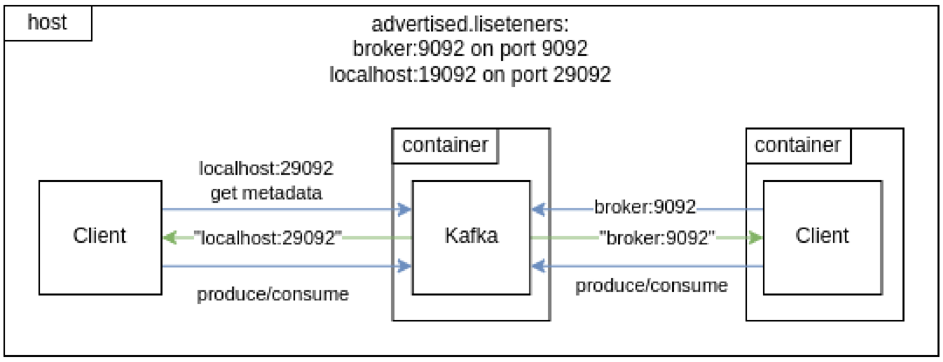


Figure 4.6: Kafka connection between host and container

This final configuration is a combination of both, thus allowing for communication between containers and the host to Kafka.

Code snippet 5 demonstrates the configuration done in order to achieve this.

Apache Cassandra Node's bootstrap collision, if nodes initialise at the same time, they end up having token collisions with each other when bootstrapping into the cluster, as Cassandra seems to use the same default node by default when initialized. The workaround was to implement "healthcheks" and "depends_on" for each node, thus controlling the bootstrapping order of each node.

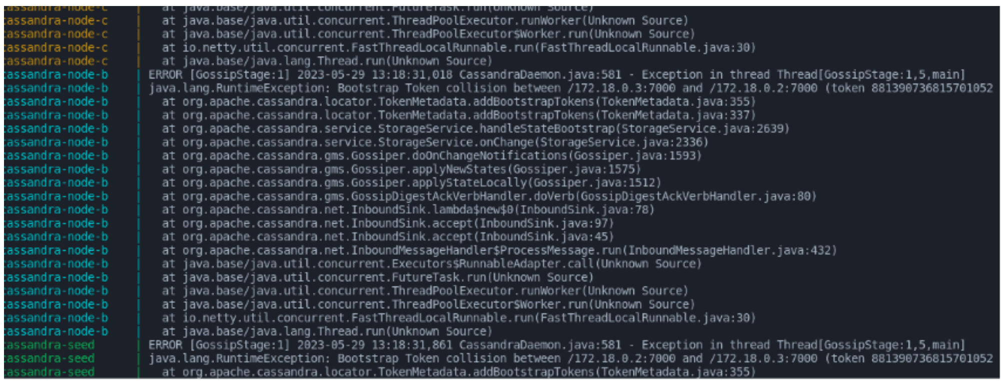
Some logs of the token collision can be seen in figure 4.7.

Due to Docker after failure, this problem eventually solves itself, although it harms startup time notably. This can be observed in figure 4.8.

Apache Spark worker's DNS resolution problem, as Spark is responsible for moving and processing data, it needs to be able to communication with the data ingestion and data

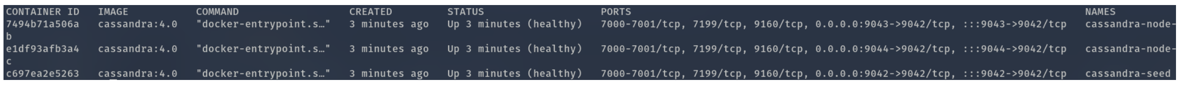
```yaml
broker:
    image: confluentinc/cp-kafka:7.3.0
    hostname: broker
    container_name: broker
    ports:
        - '29092:29092'
    expose:
        - '9092'
    depends_on:
        - zookeeper
    networks:
        - smack
    environment:
        KAFKA_BROKER_ID: 1
        KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
        KAFKA_ADVERTISED_LISTENERS: |
            PLAINTEXT://broker:9092,
            PLAINTEXT_HOST://localhost:29092
        KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: |
            PLAINTEXT:PLAINTEXT,
            PLAINTEXT_HOST:PLAINTEXT
        KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
        KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
        KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
    healthcheck:
        test: nc -z broker 29092 || exit -1
        start_period: 5s
        interval: 5s
        timeout: 10s
        retries: 10
```

Code 5: Docker-Compose Kafka setup



```
cassandra-node-c     |   at java.base/java.util.concurrent.FutureTask.run(Unknown Source)
cassandra-node-c     |   at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source)
cassandra-node-c     |   at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source)
cassandra-node-c     |   at io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)
cassandra-node-c     |   at java.base/java.lang.Thread.run(Unknown Source)
cassandra-node-b     | ERROR [GossipStage:1] 2023-05-29 13:18:31,018 CassandraDaemon.java:581 - Exception in thread Thread[GossipStage:1,5,main]
cassandra-node-b     | java.lang.RuntimeException: Bootstrap Token collision between /172.18.0.3:7000 and /172.18.0.2:7000 (token 8813907368157010522
cassandra-node-b     |   at org.apache.cassandra.locator.TokenMetadata.addBootstrapTokens(TokenMetadata.java:355)
cassandra-node-b     |   at org.apache.cassandra.locator.TokenMetadata.addBootstrapTokens(TokenMetadata.java:337)
cassandra-node-b     |   at org.apache.cassandra.service.StorageService.handleStateBootstrap(StorageService.java:2639)
cassandra-node-b     |   at org.apache.cassandra.service.StorageService.onChange(StorageService.java:2336)
cassandra-node-b     |   at org.apache.cassandra.gms.Gossiper.doOnChangeNotifications(Gossiper.java:1593)
cassandra-node-b     |   at org.apache.cassandra.gms.Gossiper.applyNewStates(Gossiper.java:1575)
cassandra-node-b     |   at org.apache.cassandra.gms.Gossiper.applyStateLocally(Gossiper.java:1512)
cassandra-node-b     |   at org.apache.cassandra.gms.GossipDigestAckVerbHandler.doVerb(GossipDigestAckVerbHandler.java:80)
cassandra-node-b     |   at org.apache.cassandra.net.InboundSink.lambda$new$0(InboundSink.java:78)
cassandra-node-b     |   at org.apache.cassandra.net.InboundSink.accept(InboundSink.java:97)
cassandra-node-b     |   at org.apache.cassandra.net.InboundSink.accept(InboundSink.java:45)
cassandra-node-b     |   at org.apache.cassandra.net.InboundMessageHandler$ProcessMessage.run(InboundMessageHandler.java:432)
cassandra-node-b     |   at java.base/java.util.concurrent.Executors$RunnableAdapter.call(Unknown Source)
cassandra-node-b     |   at java.base/java.util.concurrent.FutureTask.run(Unknown Source)
cassandra-node-b     |   at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source)
cassandra-node-b     |   at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source)
cassandra-node-b     |   at io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30)
cassandra-node-b     |   at java.base/java.lang.Thread.run(Unknown Source)
cassandra-seed       | ERROR [GossipStage:1] 2023-05-29 13:18:31,861 CassandraDaemon.java:581 - Exception in thread Thread[GossipStage:1,5,main]
cassandra-seed       | java.lang.RuntimeException: Bootstrap Token collision between /172.18.0.2:7000 and /172.18.0.3:7000 (token 8813907368157010522
cassandra-seed       |   at org.apache.cassandra.locator.TokenMetadata.addBootstrapTokens(TokenMetadata.java:355)
```

Figure 4.7: Token collision of the Cassandra cluster on Docker-Compose



```
CONTAINER ID   IMAGE           COMMAND                  CREATED         STATUS                   PORTS                                                                       NAMES
7494b71a506a   cassandra:4.0   "docker-entrypoint.s…"   3 minutes ago   Up 3 minutes (healthy)   7000-7001/tcp, 7199/tcp, 9160/tcp, 0.0.0.0:9043->9042/tcp, :::9043->9042/tcp   cassandra-node-
b
e1df93afb3a4   cassandra:4.0   "docker-entrypoint.s…"   3 minutes ago   Up 3 minutes (healthy)   7000-7001/tcp, 7199/tcp, 9160/tcp, 0.0.0.0:9044->9042/tcp, :::9044->9042/tcp   cassandra-node-
c
c697ea2e5263   cassandra:4.0   "docker-entrypoint.s…"   3 minutes ago   Up 3 minutes (healthy)   7000-7001/tcp, 7199/tcp, 9160/tcp, 0.0.0.0:9042->9042/tcp, :::9042->9042/tcp   cassandra-seed
```

Figure 4.8: Cassandra container statuses some time after token collision occurs

storage layers. Due to the architecture being implemented in Docker for it to be micro-service oriented, each service lifecycle is ephemeral, thus making the address ephemeral as well. To overcome the ephemeral address, DNS resolution is possible. Unfortunately, Spark is not able to communicate with both layers through DNS, relying on "localhost" for communication. This conclusion was reached when using a simple Python image for running the intended task with the DNS names of both layers and completing it successfully, thus providing a reason for switching Apache Spark for Python.

4.3.2 Kubernetes Cluster

The Kubernetes implementation is a more robust and closer to production-grade deployment. It solves Docker-Compose problems like scalability, fault tolerance, network limitations, monitoring, and management.

The cluster is created through **Minikube**, a single-node Kubernetes implementation the eases development and testing. The pipeline will be separated by namespaces, a virtual separation that separates groups of resources, each representing a phase of the pipeline.

In figure 4.9 it is possible to observe the cluster structure and how each component interacts.

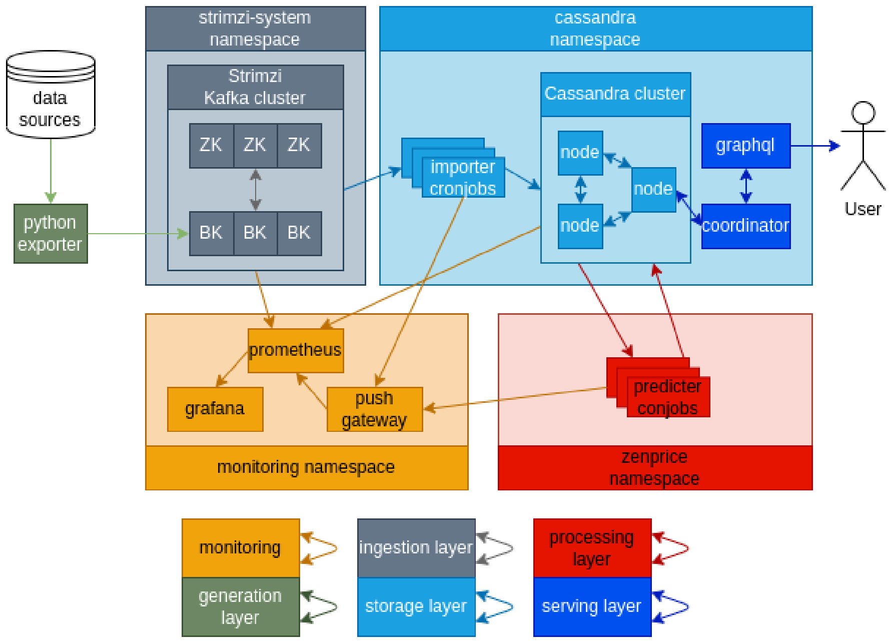


Figure 4.9: Kubernetes Architecture Diagram

Following, specific focus will be given to the deployment steps of the pipeline in order to make sure the pipeline in a functional state.

Volume provisioning can be done through the creation of PersistentVolumeClaim and PersitentVolume resources. However, volume provisioning was done with Rancher's **local-path-provisioner**[22], a custom Kubernetes storage class resource that eases provisioning of volumes by handling PersistentVolume creation automatically, thus only requiring the

[22]`https://github.com/rancher/local-path-provisioner`

creation of PersistentVolumeClaim resources for the desired Kubernetes adapted component in question.

Scalling is made through the HorizontalPodAutoscaler, a Kubernetes resource that scales other Kubernetes resources responsible for executing workloads. It scales them by monitoring the CPU and memory usage, if it is near the defined limit then new Pods are created in order to satisfy the demand. Pod amount is based on the minimum and maximum defined for the HorizontalPodAutoscaler.

Strimzi deploys the Kafka cluster, configuration was needed in order for it to oversee and act only upon the "strimzi-system" namespace. Strimzi already provides the deployment files for a version of the cluster with three brokers, three zookeepers, storage handled by the operator itself, associated services for connection, and a metric scrape point. Besides the default services for communication with the cluster, an additional service of type "load-balancer" was added in order to connect directly to the broker in question and facilitate testing. Topic creation can be handled through the client or either through a YAML file, which is then handled by the Topic Operator. Data replication between clusters is handled through a Strimzi resource denominated MirrorMaker. It possesses 2 versions, the later having ease of deployment and configuration. While Strimzi also provides a Prometheus deployment, this was not used due to the fact that the deployment was handled by Strimzi itself instead of creating a proper YAML deployment file.

Importer and **Predicter** Python jobs were deployed as a CronJob, a Kubernetes resource, in the "cassandra" and "zenprice" namespaces respectively. Due to the repetitive and ephemeral nature of the tasks, the CronJob is the ideal resource to adapt them to the Kubernetes environment. Configuration is made through environment variables, thus simplifying the deployment process for the prediction of different products or importing data from different Kafka topics onto different Cassandra tables. Deployment is done to the "zenprice" namespace. Endpoint exposure for metric scraping is not possible due to the ephemeral life of CronJobs, as such, metrics as pushed onto a **Prometheus Pushgateway** deployment that aggregates all the metrics pushed and provides a steady endpoint for it to be scraped.

Cassandra is deployed through a StatefulSet resource, a component that eases scaling and saves the application state if the container resets. It is deployed to the "cassandra" namespace. Connecting to the cluster can be made through two "headless" services, one required by the StatefulSet to identify Pods, and the second for exposing a metric endpoint for scraping present on each node. An important aspect of the "headless" service is that it performs a DNS lookup and returns every Pod IP, which interacts nicely with the monitoring layer [59]. It scales through the HorizontalPodAutoscaler and assigns a specific network name to each Pod based on the existing amount, thus easing the identification of seed nodes. The same configuration from the Docker-Compose version is used, although it needs to be deployed as a ConfigMap into into the same namespace. Volume provisioning is done with Rancher's "local-path-provisioner". An endpoint is exposed for metric scraping, while metrics are generated through

Stargate components, Coordinator and Graphql API, are deployed as separate Deployments, a Kubernetes resource for workloads, in the “cassandra” namespace. The Coordinator connects to the Cassandra seed node’s FQDN (fully qualified domain name), possible through the “headless” service that allows for specific selection of a Pod DNS, while the Graphql API connects to the Coordinator’s associated service.

Prometheus & Grafana are deployed as a Kuberentes Deployment resource, as it provides the desired availability. Storage is provided with Rancher’s “local-path-storage” while the configuration is through a mounted ConfigMap. Metric scrapping is usually directly to the micro-service in question, however, Importer and Predicter make use of **Prometheus Pushgateway** due to their ephemeral nature. It provides a way to aggregate cronjob metrics and a steady point for Prometheus to scrape, made available through a service.

Additional Minikube plugins were needed in order for the cluster to function optimally, since Minikube is a minimal implementation of Kubernetes, some functionalities are not present. These plugins are maintained either from Google itself or from 3rd party entities. The installed plugins are as follow:

- **MetalLB** is an open-source implementation of a load balancer for Kubernetes clusters. It enables the exposure of “load-balancer” type services by providing a static IP to them, thus facilitating and distributing external connections to multiple instances of a target component running in the cluster;
- **Metrics-Server** is a Kubernetes resource that collects metrics related to resource consumption and utilization of nodes and workloads in order to control scaling of said workloads. While it provides metrics, these metrics are not meant for monitoring purposes;

4.3.3 AWS EKS Cluster

AWS is a cloud based service provider that provides a pay-as-you-go model to individual clients, companies, and governments.

One of the services it provides is the EKS. It is an AWS managed service that provides a cloud based Kubernetes environment able to scale according to demand. Storage and networking are also handled by AWS since it is tightly coupled with other AWS services,

Through this service, a production-grade deployment was made. The contrast between Minikube and EKS deployments lies in the environment itself, and not the pipeline itself. EKS effectively provides a more robust and mature environment.

Table 4.1 compares both Minikube and EKS on some key characteristics:

EKS’s tightly coupled nature to other AWS services is what enables it to provide production-grade Kubernetes environments. In order to do so, it interacts with the following services:

- Identity and Access Management (IAM), provides a central and fine-grained control of access to users and resources. This service will allow the creation of roles, which can hold specific accesses, which will then be given to the EKS service in order to create the cluster;
- Elastic Cloud Compute (EC2), is the main service for computing in the cloud. Through this service it will be possible to provide nodes the cluster;

Table 4.1: Comparison between Minikube and EKS regarding multiple important aspects.

Product	**Minikube**	**EKS**
Deployment Environment	local and testing environments	cloud based production environments
Scalablity	limited to one node	scale to demand
Service Management	self managed	AWS managed
Complexity	simple and concise	complex with additional AWS configuration
Network & Integration	limited to local machine specs	strongly coupled to other services
Availablity	limited due to being local	supported by default due to multiple availability zones
Security & Compliance	local envrionment dependent	benefits from AWS compliance standards
Cost	local machine dependent	cloud resources costs

- Elastic Block Storage (EBS), is a service that provides block storage to other services available on AWS. This service will provide the desired storage for the cluster;
- Virtual Private Cloud (VPC), is a service that provides a logically isolated virtual network with all the utilities a network can provide. This will allow the communication between the control pane and the nodes;

There as two ways to interact with the AWS services, the management console and the AWS CLI. While both allow interaction, the management console provides an easier interaction and allows for a more exploratory view of the services in a more user friendly way, while the CLI provides a more powerful yet complex and precise way of interaction, providing power users a way to solve their specific needs that the console can't solve.

The whole process of creating and configuring the EKS cluster was done mainly through the management console, however, there are some steps further ahead that require the CLI due to their intrinsic nature and complexity. For the creation of the EKS itself, the management console was enough. The steps needed to create it revolve around creating IAM roles with the appropriate permissions as well as the components themselves. The steps are the following:

- creation of IAM role for EKS control pane, this will ensure the EKS service has the necessary permissions to create the cluster, since it needs to leverage other services like EC2 and EBS in order to create it;
- creation of EKS control pane and installation of addons, it used the previously mentioned IAM role to create the control pane which is responsible for controlling the whole cluster.
- creation of IAM role for EKS node group, this will ensure the node group used by the EKS service creates nodes based of EC2 instances;
- creation of node group and nodes, which uses the previously mentioned IAM role to create EC2 based nodes for the cluster;

The cluster is comprised of a single EC2 based node, one that uses a **t3.2xlarge**[23] instance tyep which boasts eight CPUs and a total of 32GB of memory.

After the creation of the components, it is necessary to configure the cluster in order to use it. The configuration process focuses on the installation of addons that provide crucial functionalities for the Kubernetes cluster as well as configuring access for other users/roles. The addons necessary for the proper functioning of the cluster are the following:

- coreDNS, a DNS server responsible for providing name resolution and service discovery for applications inside the cluster;
- kube-proxy, reponsible for enabling communication between services and pods inside the cluster;
- VPC CNI[24], provides management to networking aspects inside the VPC allowing for efficient communication of pods;
- EBS CSI[25], provides EBS capabilities to the pods inside the cluster for persistent storage;

To configure access to the cluster, it is necessary to add a new Kubernetes Role or ClusterRole along it's respective RoleBinding or ClusterRoleBinding to the "aws-auth" ConfigMap, a Configmap used by EKS to authenticate access to the cluster. Important to denote that after creating the cluster, the only account with access to it is the one that created it in the first place, hence the need to configure access. In order to configure access, it is necessary to use the AWS CLI along eksctl[26], a CLI tool to manage EKS clusters.

Code snippet 6 demonstrates the necessary script to configure access through eksctl is the following:

```
# update kubeconfig file to have access to EKS cluster
aws eks update-kubeconfig --region region-code --name cluster-name

# create identify mapping for new user/role
eksctl create iamidentitymapping \
--cluster cluster-name \
--region=region-code \
--arn arn:aws:iam::############:role/role-name \
--username username \
--group group-name \
--no-duplicate-arns

# apply role and rolebinding
kubectl apply -f xpto_role_n_rolebinding.yaml
```

Code 6: EKS commands to provide access to a specific IAM role

After creating the control pane and node with the desired configuration, the pipeline can be deployed just as it is since the Kubernetes environment is identical to the one Minikube provides.

[23]https://aws.amazon.com/ec2/instance-types/
[24]https://docs.aws.amazon.com/eks/latest/userguide/managing-vpc-cni.html
[25]https://docs.aws.amazon.com/eks/latest/userguide/ebs-csi.html
[26]https://eksctl.io/

4.4 GitOps & CI/CD

Taking into account the main task of creating a pipeline through configuration files as well as automating the deployment process, GitOps seems to be the best suited work methodology for the whole process.

It combines DevOps best practices with version control systems in order to provision infrastructure, usually cloud environments, through configuration files, similarly as to how software development teams use source code.

GitOps is flexible and as such, there is no defined way of applying it to every team, as each has their own set of situations and characteristics. However, there are 3 core components that are always present:

- git repository, a single repository managed by a git versioning system that works as single source of truth for everything, effectively presenting an immutable and versioned system state that is desired;
- change mechanism, use merge and pull requests as a trigger to control and log changes collaboratively, setting the target environment to the desired state;
- CI/CD, continuous integration and deployment that automatically applies the desired system state present on the git repository to the target environment;

GitLab[27] has it's own CI/CD tools that provide all the functionalities for extensively using GitOps as the work methodology. It provides a repository for tracking and versioning all the code that has been done, issue tracking and monitoring, and automation for CI/CD tasks.

In order to make the best use of the Gitlab CI/CD features, the repository will be separated into two important branches that will facilitate execution of CI/CD tasks in an ordered and predetermined time:

- development branches, where features are developed and tested according to needs, and are named according to the feature in development;
- main, where features are deployed according to the most recent state of the repository;

Automation of tasks were separated into two categories, each reflecting the integration and deployment aspects.

For Continuous Integration, a runner was installed locally. This runner handles tasks that are responsible for creating docker images used in the pipeline and pushing them to Docker Hub. It is triggered when a new commit is pushed to the development branch in question.

Code snippet 7 represents the script responsible for building the Importer Docker image, executed by the local runner.

Code snippet 8 represents the script responsible for building the Predicter Docker image, also executed by the local runner.

The runner uses the "Docker in Docker" executor, effectively running tasks inside a Docker container that possesses all Docker tools. This executor proves effective at building images and pushing them to Docker Hub. Important to denote that these tasks do not run on the "main" branch.

[27]`https://about.gitlab.com/`

```
importer-build:
  stage: build
  image: docker:24.0.4
  tags:
    - image.builder
  services:
    - docker:24.0.4-dind
  before_script:
    - |+
        echo $DATA_ENG_DOCKER_PASS | docker login -u $DATA_ENG_DOCKER_USER \
        --password-stdin
  script:
    - |+
        docker build -f ./k8s/importer/importer.dockerfile \
        -t d1scak3/importer:1.0 ./k8s/importer/
    - docker push d1scak3/importer:1.0
  except:
    - main
```

Code 7: Continuous Integration - Importer build

```
predicter-builds:
  stage: build
  image: docker:24.0.4
  tags:
    - image.builder
  services:
    - docker:24.0.4-dind
  before_script:
    - |+
        echo "$DATA_ENG_DOCKER_PASS" | docker login -u $DATA_ENG_DOCKER_USER \
        --password-stdin
  script:
    - |+
        docker build -f ./k8s/predicter/predicter_simple.dockerfile \
        -t d1scak3/predicter:lstm_simple ./k8s/predicter/
    - docker push d1scak3/predicter:lstm_simple
    - |+
        docker build -f ./k8s/predicter/predicter_conv.dockerfile \
        -t d1scak3/predicter:lstm_conv ./k8s/predicter/
    - docker push d1scak3/predicter:lstm_conv
  except:
    - main
```

Code 8: Continuous Integration - Predicter build

For Continuous Deployment, a runner was installed on an EC2 **t2.micro**[28] instance. Access was configured in order allow deployment of CronJobs only to the “zenprice” namespace, as this task is triggered when there is an event in the main branch, usually from a merge request with a new developed feature.

The runner uses the “shell” executor, thus executing bash commands as provided. First the task updates the “kube-config” file to ensure connectivity to the cluster, following by deleting all existing CronJobs deployed in the “zenprice” namespace, and finally deploying the new CronJobs. This option proves to be the most straightforward when compared to other executors that need additional configuration or tools.

To add an additional layer of security to all tasks that need sensitive information, variables are used in order to hide said information.

Code snippet 9 demonstrates the script executed on the auxiliary EC2 instance.

```
jobs-deployment:
  stage: deploy
  tags:
    - jobs.deployer
  before_script:
    - aws eks update-kubeconfig --region $AWS_DEFAULT_REGION --name $AWS_DATA_ENG_PIPELINE
  script:
    - export KUBECONFIG=~/.kube/config
    - |+
      CRONJOB_NAMES=$(
      kubectl get cronjobs -n "$AWS_DATA_ENG_NAMESPACE" -o jsonpath='{.items[*].metadata.name}'
    )
    - IFS=" " read -ra CRONJOB_ARRAY <<< "$CRONJOB_NAMES"
    - |+
      if [ -z "${CRONJOB_ARRAY[@]}" ]; then
        echo "No cronjobs found. Deploying new cronjobs."
        kubectl apply -f ./k8s/importer/cron-importer.yaml -n "$AWS_DATA_ENG_NAMESPACE"
        kubectl apply -f ./k8s/predicter/cron-predicter.yaml -n "$AWS_DATA_ENG_NAMESPACE"
        echo "Cronjobs deployed."
      else
        echo "Deleting cronjobs."
        for CRONJOB_NAME in "${CRONJOB_ARRAY[@]}"; do
          kubectl delete cronjob "$CRONJOB_NAME" -n "$AWS_DATA_ENG_NAMESPACE"
        done
        echo "Deploying new cronjobs."
        kubectl apply -f ./k8s/importer/cron-importer.yaml -n "$AWS_DATA_ENG_NAMESPACE"
        kubectl apply -f ./k8s/predicter/cron-predicter.yaml -n "$AWS_DATA_ENG_NAMESPACE"
        echo "Cronjobs deployed."
      fi
  only:
    - main
```

Code 9: Continuous Deployment task

4.5 Chapter Summary

This chapter starts with section 4.1, providing a general overview of the pipeline’s architecture. The architecture consists of multiple parts, each responsible for a specific functionality

[28] https://aws.amazon.com/ec2/instance-types/

of the pipeline, them being: • data generation; • data ingestion; • data storage; • data processing; • data serving; Each of these parts represents a phase of the data engineering lifecycle, which has been mentioned in 2.1.2.

Section 4.2 approaches the component choices for each part of the pipeline. Due to the pipeline being based of the SMACK stack, and intended to be deployed to a Kubernetes environment, many of the choices were constrained due to previously defined requirements. Some components were subject to change, due to situations regarding the development environment or even failing to reach the defined requisites, these were switched for other ones that proved to be better. Cases such as Apache Spark being switched by Python, due to having DNS resolution problems even on an local environments, as well as K8ssandra being switched by a simple Apache Cassandra cluster, due to not even working on Minikube environments.

Section 4.3 addresses the deployment of the pipeline into multiple environments, namely Docker with Docker Compose, Kubernetes with Minikube, and Kubernetes with EKS. Subsection 4.3.1 approaches the development of a proof of concept built in Docker Compose. It allows for the development and testing of a small scale version of the future pipeline, allowing for extraction of insights regarding its behavior and possible problems. This version of the pipeline proved to be helpful, since it allowed for a timely switch of Python instead of Apache Cassandra. Subsection 4.3.2 approaches the development of a more robust version of the pipeline. While it is a Kubernetes version that is closer to production, the environment used, Minikube, was intended to test and debug the pipeline on a Kubernetes environment. This lead to the exclusion of the K8ssandra operator due to not even working on single node clusters. This version also allowed to handle problems like volume provisioning, monitoring, and scalability. Subsection 4.3.3 approaches a production-grade deployment of the pipeline to a cloud environment, namely AWS. It is much more robust and mature environment when compared to Minikube, since it boasts security, scalability, and configurations managed by AWS itself, letting the developer focus more on the product itself and less on hardware hassles. Due to tight integration between EKS and other AWS services, configuration of IAM roles to control access and provide authorization for the creation of the cluster were necessary.

Lastly, section 4.4 approaches the work methodology and it's application in building the pipeline. GitOps is the chosen work methodology to develop and deploy the pipeline, as it encompasses various critical areas such as automation, versioning, infrastructure, continuous integration, and continuous deployment, all concisely consolidated through DevOps principles. Regarding implementation of GitOps, the integration and deployment tasks are handled by GitLab runners. Integration tasks are handled by a local runner that uses the "Docker in Docker" executor in order to build Docker images and push them to Docker Hub. Deployment tasks are handled by an EC2 instance with the "shell" executor, which was configured to only access and deploy CronJobs the "zenprice" namespace.

CHAPTER 5

Testing & Results discussion

This chapter addresses the testing of the pipeline and obtained results during this phase. The chapter will focus first on the necessary setup considerations that can highly affect the results, aspects like cluster hardware, component configuration, and such will be discussed in detail. Following, an overview of the data used during the testing phase, namely the structure, size, and amount of measures. Finally, each testing scenario will be addressed along the obtained results. Each scenario aims to mimic a real-life production grade environment, testing a specific functionality of the pipeline such as ingestion, storing, processing, and serving, and even the automation aspect of it, effectively proving that the pipeline works as intended.

5.1 Test conditions

When setting the pipeline for the test scenarios, some considerations must be taken into account before proceeding, since these can highly influence either how the testing process is carried or the obtained results. These considerations are as follow:

- **data**, the pipeline will use 4610 product measures. Although small in amount, it is condensed enough to prove it's value in each testing scenario. This data was provided by Ritain.io in collaboration with Institute of Electronics and Informatics Engineering of Aveiro (IEETA) [4];
- **data ingestion's parallelism** will be tested with the help of GNU Parallel, a Unix tool that allows for the execution of processes in parallel [60];
- **machine-learning models**, the models used in question are two LSTM models, a simple and a convolutional versions, both provided by Ritain.io and IEETA, and adapted in order to be executed from inside a containerized environment. Important to denote that results of the models are not important for conclusions, and are only being used to prove that the pipeline is able to run them;
- **Grafana's monitoring dashboards** will be used to observe the behaviour or each component. These dashboards where created by the community and are free and open-source for use. Dashboards where used for Kubernetes cluster metrics [61], Strimzi [62], and Cassandra [63];

Important to denote that after every component is deployed to the pipeline, with the exception of Importer and Predicter instances, the pipeline has 2 CPUs and 14.7Gb of memory as available resources.

5.2 Data analysis

The data used in the testing process of the pipeline was provided by Ritain.io, it is data collected by Zenprice and refers to multiple products sold by multiple companies, all of them originating from Chile. It is important to denote that this data is not recent, it is from a time-span of two year, between 2019 and 2021, and holds no value for any other situation besides testing.

The data is present across 292 files, representing 220 different products across 367215 measures in total. In the following table 5.1 we can observe how the data is composed by eight columns.

Table 5.1: Structure of the files

	product_id	**timestamp**	**product**	**country**	**company**	**product_group_id**	**offer_type**	**price**
418936	3200	2019-12-21	motorola Moto Z3 Play 128GB + Gamepad	CL	movistar	951	unlocked	272.891840
418937	3200	2019-12-22	motorola Moto Z3 Play 128GB + Gamepad	CL	movistar	951	unlocked	272.891840
418938	3200	2019-12-23	motorola Moto Z3 Play 128GB + Gamepad	CL	movistar	951	unlocked	272.891840
418939	3200	2019-12-24	motorola Moto Z3 Play 128GB + Gamepad	CL	movistar	951	unlocked	272.891840
418940	3200	2019-12-25	motorola Moto Z3 Play 128GB + Gamepad	CL	movistar	951	unlocked	272.891840
...	...	...	...	...	...	...	...	...
420294	3201	2021-10-24	motorola Moto Z3 Play 128GB + Gamepad	CL	claro	951	unlocked	227.408351
420295	3201	2021-10-25	motorola Moto Z3 Play 128GB + Gamepad	CL	claro	951	unlocked	227.408351
420296	3201	2021-10-26	motorola Moto Z3 Play 128GB + Gamepad	CL	claro	951	unlocked	227.408351
420297	3201	2021-10-27	motorola Moto Z3 Play 128GB + Gamepad	CL	claro	951	unlocked	227.408351
420298	3201	2021-10-28	motorola Moto Z3 Play 128GB + Gamepad	CL	claro	951	unlocked	227.408351

The first column represents the a unique ID to identify the product, each product corresponds to a combination of product, company, and offer type. The second column corresponds to the timestamp of when the measure was taken. The third column represents the product in question. The forth column is the country where the product is sold. The fifth is the company selling the product. The sixth is the product group id, a unique ID that represents the file that holds the data. The seventh is the type of offer. The eight is the price of the product when it was measured. In general, each measure represents the price evolution of the product until it stops being sold. If the product sale resumes, it will appear again.

Table 5.2 demonstrates the amount of different products and total amount of measures by company.

Table 5.3 demonstrates the sum of measures by type, notice that the majority of measures come from **unlocked** offers, with a small group of **postpaid_new_line**, and an almost negligible amount of **postpaid_portability**.

As it has been previously mentioned on 5.1, the data used will be a single file with **4610 measures** related to a **single product** and **unlocked offer type**, namely the **samsung Galaxy A51 128GB**. While representing only a fraction of all available data, this proves to be enough to test the pipeline as intended. Table 5.4 demonstrates the amount of measures by company and offer type:

Table 5.2: Amount of distinct products and total measures by company

company name	distinct products	amount of measures
Movistar	159	116368
Claro	110	64330
Abcdin	103	22217
Entel	88	34058
Ripley	124	31140
Paris	129	32953
Falabella	110	31205
WOM	52	15217
Lider	81	14289
VTR	8	4629
MacOnline	5	809

Table 5.3: Amount of measures by offer type

type of offer	unlocked	postpaid new line	postpaid portability
amount of measures	354887	11261	1067

Table 5.4: Amount of measures by company on the "long_product_group_id_23" file

company name	movistar	Abcdin	entel	Ripley	Paris	claro	Falabella	Lider	wom
measure amount	670	477	655	433	474	636	511	305	449

5.3 Test Scenarios

The test scenarios will take place in the AWS EKS version of the pipeline, as this version is considered the final one. The previous versions such as the Docker PoC and Kubernetes do not have all the functionalities and do not represent a real production grade environment, respectively. There is one exception, which is related to one of the scenarios that compares the performance of a component on the pipeline and locally. This test scenario will be addressed further ahead.

Every scenario will be observed through metric observation, made possible with Prometheus, Grafana, and DataGrip[1]. Prometheus and Grafana are the components used for monitoring the pipeline, while Datagrip is a cross-platform IDE for interacting and handling databases.

5.3.1 Local vs Pipeline Scenario

This scenario aims to compare the behaviour and performance of the Predicter component when being executed locally and on the pipeline. The model used in this test was the simple LSTM version.

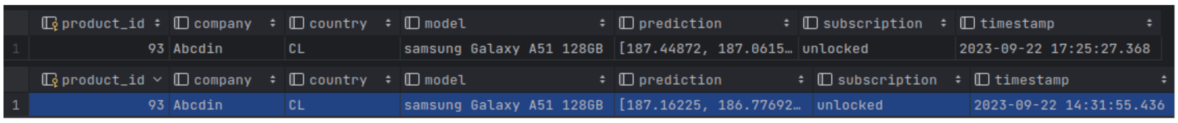
[1] `https://www.jetbrains.com/datagrip/`

Locally, the Predicter is executed as a Python script on an Intel i7-8750H CPU able to run from 2.20GHz up to 4.10GHz, along 16GB of available RAM [64]. This allows for the model to be executed with a **7.7483s +- 1.1027s average execution time and standard deviation**.

In the pipeline, the Predicter is executed on a t3.2xlarge EC2 instance able to run at a 3.10GHz along 32GB of available RAM [65]. This allows for the model to be executed with a **13.045 +- 0.289s average execution time and standard deviation**.

Both results are the average of 15 executions and are executed with no restriction to the available resources.

Figure 5.1 demonstrates results processed from the local and pipeline versions respectively. The dark line represents results from the local version, while the blue line represents values from the pipeline version.



	product_id	company	country	model	prediction	subscription	timestamp
1	93	Abcdin	CL	samsung Galaxy A51 128GB	[187.44872, 187.0615…	unlocked	2023-09-22 17:25:27.368

	product_id	company	country	model	prediction	subscription	timestamp
1	93	Abcdin	CL	samsung Galaxy A51 128GB	[187.16225, 186.77692…	unlocked	2023-09-22 14:31:55.436

Figure 5.1: Predicter obtained results from local and pipeline versions, seen through Datagrip.

Each column presented in the screenshot represents a a specific characteristic of the measure. It is also possible to notice the difference in predicted values in the "prediction" column, while the values are indeed deterministic in their own environment, the values seem to defer from environment to environment. The models were configured to be deterministic, according to the documentation of tensorflow [66] and keras [67]. The difference between environments might be due to different hardware every possible configuration in the software was done in order to obtain deterministic results.

Taking into account the obtained results, we can safely conclude that the pipeline does indeed behave as intended. The increase in execution time is expected as the CPU used on the pipeline is weaker when compared to the one used locally.

5.3.2 Data Ingestion Scenario

This scenario tests the ingestion of data into the pipeline by running multiple instances of data Producers and sending data to a Kafka Broker. Data will be visualized inside the Broker through the Kafka plugin from DataGrip [68].

According to what has been defined in 3.2 about the requirements, the configurations of the components have been previously mentioned on sections 4.2.2 and 4.2.3 about the broker and the topic, as well as the Producer configuration.

When sending data to Strimzi for ingestion, we can observe through the Grafana dashboards the increase in message throughput for each topic. Figure 5.2 represents the amount of messages that Strimzi is handling at a certain time:

Figure 5.3 demonstrates, through Datagrip's Kafka plugin [68], all the measures received.

Table 5.5 demonstrates the time taken for Strimzi to receive the same 4610 measures through two different ways, the first when exporting to a single topic, and the second when

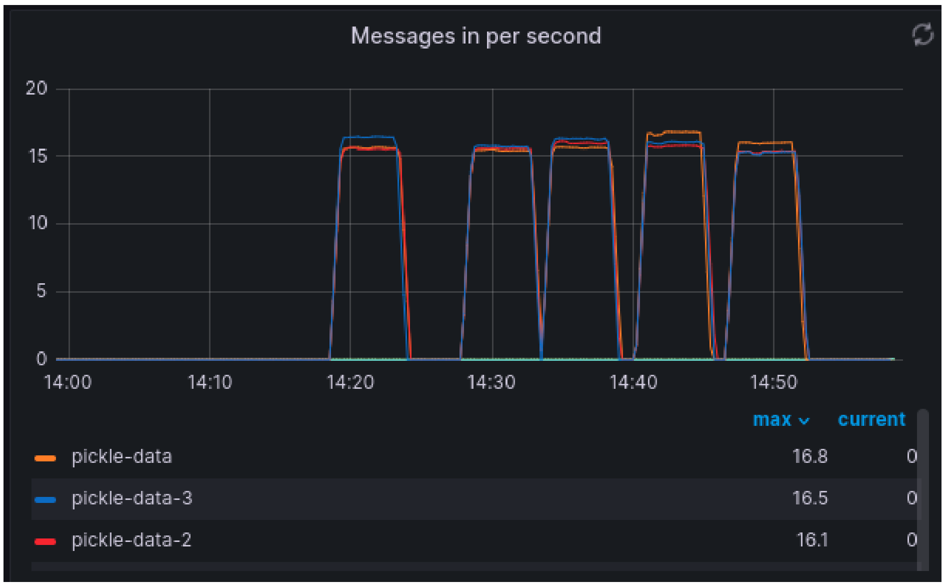
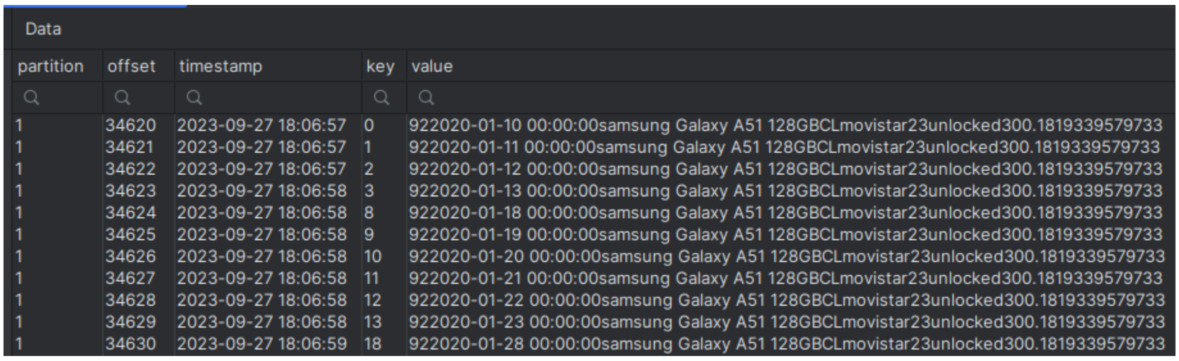


Figure 5.2: Grafana dashboard of received messages per topic



Data

partition	offset	timestamp	key	value
1	34620	2023-09-27 18:06:57	0	922020-01-10 00:00:00samsung Galaxy A51 128GBCLmovistar23unlocked300.1819339579733
1	34621	2023-09-27 18:06:57	1	922020-01-11 00:00:00samsung Galaxy A51 128GBCLmovistar23unlocked300.1819339579733
1	34622	2023-09-27 18:06:57	2	922020-01-12 00:00:00samsung Galaxy A51 128GBCLmovistar23unlocked300.1819339579733
1	34623	2023-09-27 18:06:58	3	922020-01-13 00:00:00samsung Galaxy A51 128GBCLmovistar23unlocked300.1819339579733
1	34624	2023-09-27 18:06:58	8	922020-01-18 00:00:00samsung Galaxy A51 128GBCLmovistar23unlocked300.1819339579733
1	34625	2023-09-27 18:06:58	9	922020-01-19 00:00:00samsung Galaxy A51 128GBCLmovistar23unlocked300.1819339579733
1	34626	2023-09-27 18:06:58	10	922020-01-20 00:00:00samsung Galaxy A51 128GBCLmovistar23unlocked300.1819339579733
1	34627	2023-09-27 18:06:58	11	922020-01-21 00:00:00samsung Galaxy A51 128GBCLmovistar23unlocked300.1819339579733
1	34628	2023-09-27 18:06:58	12	922020-01-22 00:00:00samsung Galaxy A51 128GBCLmovistar23unlocked300.1819339579733
1	34629	2023-09-27 18:06:58	13	922020-01-23 00:00:00samsung Galaxy A51 128GBCLmovistar23unlocked300.1819339579733
1	34630	2023-09-27 18:06:59	18	922020-01-28 00:00:00samsung Galaxy A51 128GBCLmovistar23unlocked300.1819339579733

Figure 5.3: Messages present on a Kafka topic, seen through Datagrip

exporting to separated topics. The times obtained are the average and standard deviation of 10 executions, made possible with the help of GNU Parallel to run the exporter processes in parallel [60].

Table 5.5: Comparison of data ingestion performance when writing data to Strimzi to a single and multiple topics.

	1 Exporter	3 Exporters	5 Exporters
1 topic for all exporters	293.980 +- 9.930s	290.640 +- 8.429s	289.856 +- 11.539s
1 topic per exporter	293.980 +- 9.930s	292.497 +- 7.621s	294.246 +- 9.263s

Taking into account the obtained results, we can safely conclude that the pipeline can easily handle data ingestion without any noticeable drawbacks. Data is received, independently of data exporter amount or topic amount through a single topic or multiple topics, weather writing data consecutively or in parallel.

5.3.3 Data Storing Scenario

This test scenario focuses on the data storing functionality of the pipeline. The data that is temporarily stored on Strimzi will be retrieved by the Importer component and redirected to Cassandra to be stored.

Figure 5.4 demonstrates the Grafana dashboard for Strimzi, enabling to observe messages read by second to Strimzi.

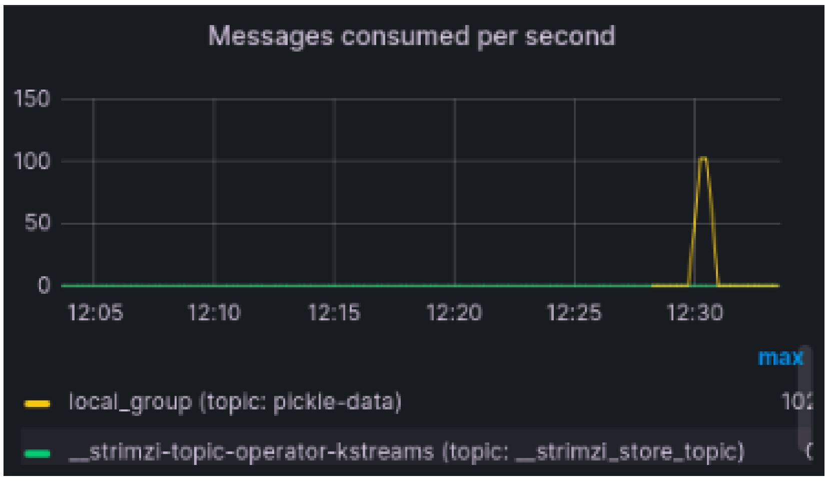


Figure 5.4: Grafana dashboard of messages being read from a topic.

Table 5.6 demonstrates the times taken for the Importer component to import data from Strimzi into Cassandra. Each column and line combination corresponds to a resource configuration related to the CPU and memory amount. The times correspond to the average and standard deviation of the execution time of a single running instance ten times.

Table 5.6: Comparison of Importer performance with different resource configurations.

	1Gib	0.5Gib	0.25Gib
1 CPU	22.537 +- 1.262s	22.396 +- 1.205s	22.877 +- 1.879s
0.5 CPU	23.102 +- 2.090s	23.034 +- 1.774s	22.557 +- 1.459s
0.25 CPU	26.858 +- 1.465s	26.962 +- 1.343s	26.905 +- 1.547s

Table 5.7 demonstrates the execution time taken the Importer component to import data, when running with multiple instances in parallel.

Table 5.7: Comparison of Importer performance with multiple parallel instances. Each instance is configured to use 0.25 CPU and 512Mib of memory.

Instance amount	1	3	5
Execution time	24.412 +- 1.277s	26.114 +- 2.214s	32.073 +- 5.159s

Figure 5.5 demonstrates, through the Grafana Cassandra dashboard, Cassandra handling write operations made by Importer instances.

Taking into account the presented screenshots and tables, we can conclude that the pipeline can store data. Importer instances do not seem to be greatly affected by resources, this

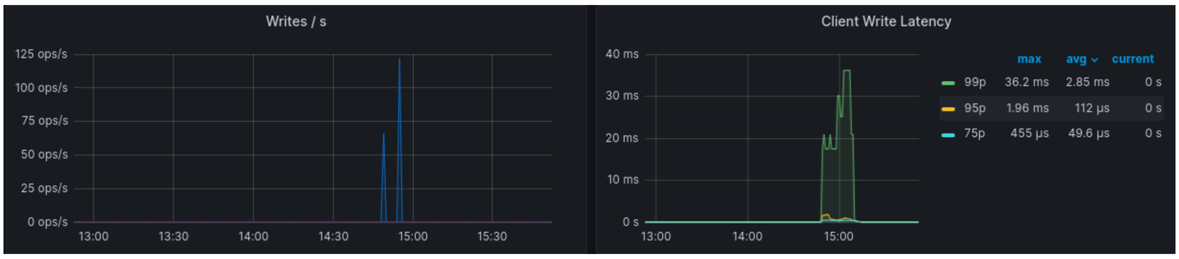


Figure 5.5: Casandra handling write operations made by Importer instances

might be due to the fact that the Importer is mainly executing IO operations from Strimzi to Cassandra, thus not needing much processing power. However, deploying multiple instances seem to affect to some degree the performance, this might be due to CPU sharing since each parallel instance was configured to use 0.25 CPU.

Combining every information taken, we can conclude that the Importer is mainly impacted by the time the CPU dedicates to each instance instead of the available resources directly, while this directly relates to an increase in execution time, the work done is much higher for the total time taken when compared to the time work done sequentially would take.

5.3.4 Data Processing Scenario

This scenario aims to test the processing phase of the pipeline. The data is queried to Cassandra by the Predicter component, to be processed, and then stored back to Cassandra.

Table 5.8 demonstrates the performance of a Predicter instance running the LSTM simple model, varying in resource configuration.

Table 5.8: Comparison between different configurations of Predicter instances using the simple LSTM model.

CPU/Memory	1Gib	0.5Gib	0.25Gib
1	13.7112 +- 0.2411s	13.1936 +- 0.2329s	12.9303 +- 0.1418s
0.5	26.3653 +- 0.1601s	27.3676 +- 0.1959s	26.6845 +- 0.4756s
0.25	61.2682 +- 9.7783s	61.3421 +- 9.1907s	62.0867 +- 9.4638s

Table 5.9 demonstrates the execution time of the Predicter instances running in parallel. Each instance was configured to use 0.25 CPU and 512Mib of memory.

Table 5.9: Comparison of Predciter performance with multiple parallel instances. Each instance is configured to use 0.25 CPU and 512Mib of memory.

Instance amount	1	3	5
Execution time	61.268 +- 9.778s	69.573 +- 10.866s	71.414 +- 15.121s

Through figure 5.6 we can observe the messages being read from Cassandra to write back to it the obtained results. We can observe the read and write operations occurring on the same time interval.

Taking every thing into account we can say with certain that the pipeline is indeed able to process data in parallel. In contrast with the Importer performance, the Predicter

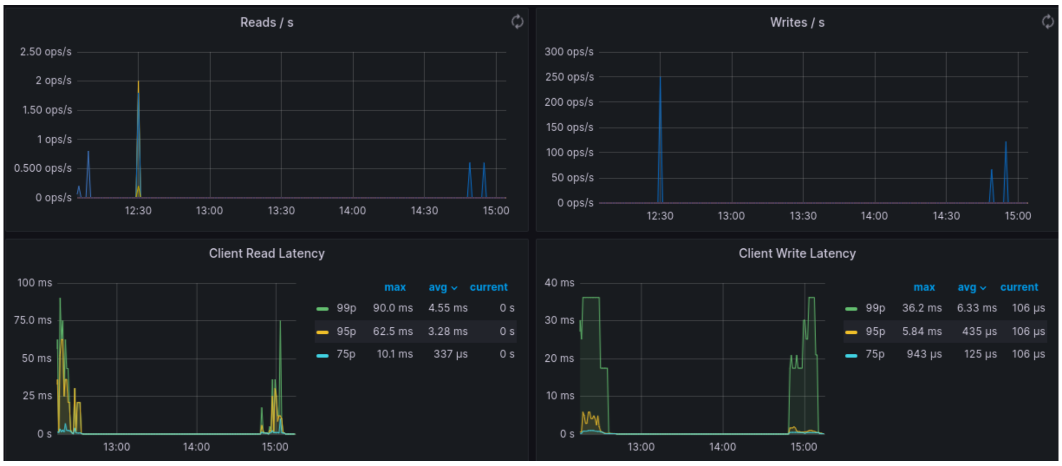


Figure 5.6: Cassandra handling read operations made by Predicter instances.

demonstrates a much higher impact on execution time depending on the available resources. As for the amount of parallel instances, similarly to the Importer, there is also a noticeable increase in the execution time, presumably due to CPU sharing. In conclusion, while there is an increase in execution time when running multiple instances in parallel, when taking into account the amount of work executed in the same time interval, parallel execution proves to be worth even when it increases execution time.

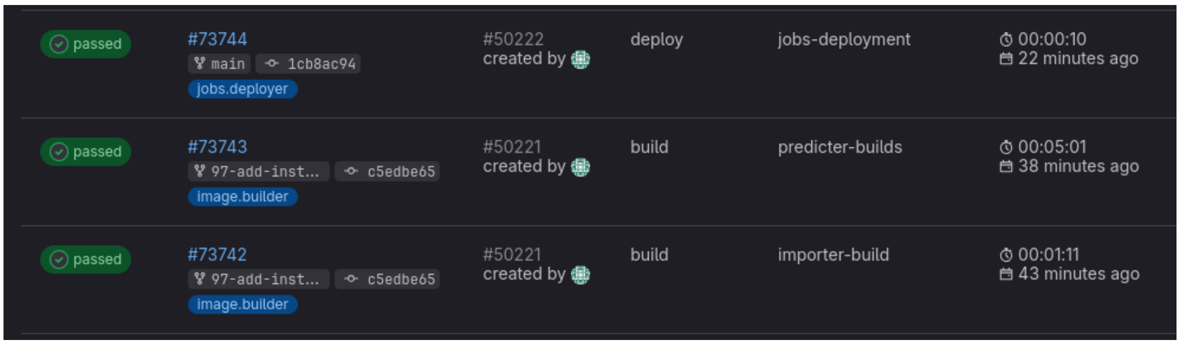
5.3.5 CI/CD of new models Scenario

This scenario tests the automation aspect of integration and deployment of new models.

A new model was adapted from the ones provided by Ritain.io and IEETA, in order to test the tasks. The new model is a convolutional LSTM. For it to be tested accordingly, the defined CI/CD tasks, previously mentioned on section 4.4, will be triggered on two occasions:

- continuous integration will be triggered on every branch besides the main. Only after the model, the CI/CD task defined, and the images ready to be build, will the task be executed;
- continuous deployment will be triggered on the main branch. Any deployment defined on the deployment task will be deployed to the pipeline;

Figure 5.7 demonstrates the CI/CD tasks being executed successfully.



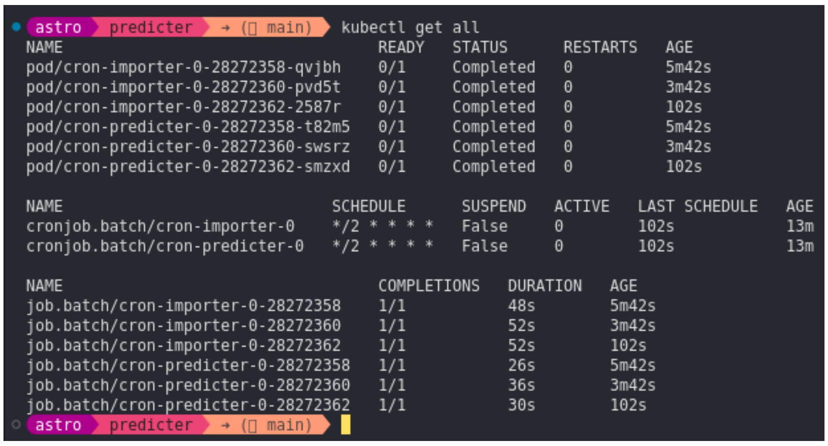
```
passed   #73744                  #50222        deploy   jobs-deployment    00:00:10
         main  1cb8ac94          created by                                22 minutes ago
         jobs.deployer

passed   #73743                  #50221        build    predicter-builds   00:05:01
         97-add-inst...  c5edbe65  created by                              38 minutes ago
         image.builder

passed   #73742                  #50221        build    importer-build     00:01:11
         97-add-inst...  c5edbe65  created by                              43 minutes ago
         image.builder
```

Figure 5.7: Successful execution of integration and deployment tasks

More specifically, it demonstrates three successfully executed tasks. Addressing each task in ascending order:

- task with id #73742, with the “build” tag, and labelled as “importer-build”, is a continuous integration task responsible for building a new Importer Docker image and pushing it to Docker Hub. This task is triggered when there is a push to a development branch;
- task with id #73743, with the “build” tag, and labelled as “predicter-build”, is a continuous integration task responsible for building a new Predicter Docker image and pushing it to Docker Hub. This task is triggered when there is a push to a development branch;
- task with id #73744, with “deploy” tag, and labelled as “jobs-deployment”, is a continuous deployment task is responsible for removing old cronjobs and deploying new ones. This task is triggered when there is a merge request or a push to the main branch;

Figure 5.8 demonstrates the cronjobs deployed in by the continuous deployment task.



```
● astro  predicter  → (□ main)  kubectl get all
  NAME                                 READY   STATUS      RESTARTS   AGE
  pod/cron-importer-0-28272358-qvjbh   0/1     Completed   0          5m42s
  pod/cron-importer-0-28272360-pvd5t   0/1     Completed   0          3m42s
  pod/cron-importer-0-28272362-2587r   0/1     Completed   0          102s
  pod/cron-predicter-0-28272358-t82m5  0/1     Completed   0          5m42s
  pod/cron-predicter-0-28272360-swsrz  0/1     Completed   0          3m42s
  pod/cron-predicter-0-28272362-smzxd  0/1     Completed   0          102s

  NAME                             SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
  cronjob.batch/cron-importer-0    */2 * * * *   False     0        102s            13m
  cronjob.batch/cron-predicter-0   */2 * * * *   False     0        102s            13m

  NAME                                 COMPLETIONS   DURATION   AGE
  job.batch/cron-importer-0-28272358   1/1           48s        5m42s
  job.batch/cron-importer-0-28272360   1/1           52s        3m42s
  job.batch/cron-importer-0-28272362   1/1           52s        102s
  job.batch/cron-predicter-0-28272358  1/1           26s        5m42s
  job.batch/cron-predicter-0-28272360  1/1           36s        3m42s
  job.batch/cron-predicter-0-28272362  1/1           30s        102s
○ astro  predicter  → (□ main)
```

Figure 5.8: Cronjobs deployed and in execution on the cluster.

Taking into account the passed tasks and the cronjobs being deployed effectively, we can safely conclude that the CI/CD of the pipeline works as intended.

CHAPTER 6

Conclusion

Recalling the initial goal, the objective of this dissertation was the creation of a scalable data engineering pipeline that can be used by MLOps initiatives in order to face the hardships of the Big Data world, all the while taking into account the best practices of data engineering.

The final iteration of the pipeline is a Kubernetes based pipeline that is able to ingest, process, store, and serve unstructured data. More precisely, the pipeline has the following features:

- ingest structured and unstructured data through Strimzi, an operator that handles Kafka clusters;
- store data, through a Cassandra database cluster;
- pre-process and process data, through the use of Python components that connect to Strimzi and Cassandra;
- serve data, through the use of Stargate GraphQL API to query for data;
- innate scalability, fault tolerance, and availability through the Kubernetes environment and innate affinity of components with it;
- monitoring through Prometheus, Prometheus Pushgateway, and Grafana to store and demonstrate collected metrics through multiple graphs and charts;

The pipeline passed through multiple iterations, each addressing a specific aspect of it. During these iterations multiple problems were found which led to a solution or an entire different path for the pipeline. The problems found, along their respective answer, are the following:

- Apache Kafka needing a specific configuration in order for a client to reach it, when being deployed through Docker Compose;
- Apache Spark being unable to resolve DNS names, which led to choosing a new component to replace it. Due to this problem being found during the Docker Compose version, which could occur again on further versions of the pipeline, Python was chosen to replace it;
- K8ssandra not being deployable to a single node Kubernetes cluster, which led to exclusion of the component in turn for a simple Apache Cassandra cluster;

- Apache Cassandra not having metrics by default, thus leading to a custom Docker image in order to add JMX metrics;

When comparing the features provided by Zenprice and the pipeline, the pipeline offers all of the features with the exception of data collection through web scraping. However, the pipeline provides additional features such as high availability, fault tolerance, scalability, and parallel data handling such as ingestion, storage, processing, and serving. Section 5.3 proves through the obtained results that the pipeline does indeed benefit from parallelism as the data it handles in parallel, while taking an additional amount of time when compared to single instance processing, the amount of data processed by amount of time is higher.

6.1 Future Improvements

After analysing the pipeline and taking into account the developed features, we can safely say that there are no features that were left to implement.

However, there are some possible improvements that can benefit to the pipeline:

- according to section 2.2, the current pipeline could benefit from being integrated into an ideal MLOps pipeline;
- switching custom Python components responsible for moving data, such as Exporter and Importer, for Benthos[1], a data streaming service;
- switching Cassandra for a more performant database, which also implies a different GraphlQL API instead of Stargate;

The integration into an ideal MLOps pipeline 2.14 could help in automating the whole process. As the pipeline includes both the team and the components of the pipeline through automation and workflows, the pipeline could benefit from better and improved models through feature engineering, model versioning, and overall automation. This however implies that there is an extensive team that is responsible for handling every aspect of the pipeline.

Benthos could help in multiple scenarios in the pipeline as it is able to extract data from multiple sources, process it with multiple different processing operations, and send it to a designated destination, all configured through a declarative configuration file. This would simplify the generation and ingestion layers of the pipeline by using the same tool and only needing a specific configuration file for each layer.

Switching Cassandra for a better database performance wise, could result in a better and more reserved use of cluster resources. According to [69], Discord managed to switch from Cassandra to Scylla[2], a C++ implementation of a No-SQL, column type database, that has highly compatible with Cassandra. The problem at hand was due to performance issues when there was a spike in their service usage, which led to a specific Cassandra node to lag behind the others due to high IO operations and garbage collection, causing the whole cluster to slow. Discord managed to overcome this by switching to Scylla, which allowed Discord to migrate from a 177 node Cassandra cluster to a 72 node Scylla cluster. Besides other improvements

[1] `https://www.benthos.dev/`
[2] `https://www.scylladb.com/`

made by Discord, which are not relevant to the context of this dissertation, Scylla seems to be a potential upgrade.

References

[1] «General data protection regulation», European Comission. (May 2018), [Online]. Available: https://gdpr-info.eu/.

[2] «Health information privacy», United States Department of Health & Human Services. (Aug. 1996), [Online]. Available: https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/index.html.

[3] *Home - ritain.io.* [Online]. Available: https://ritain.io/.

[4] *Universidade de aveiro.* [Online]. Available: https://www.ua.pt/pt/noticias/16/80206.

[5] AWS. «Aws well-architected framework». Last Accessed 6 January 2023. (Oct. 2022).

[6] «What is data engineer: Role description, responsibilities, skills, and background». Last Accessed 9 January 2023. (Apr. 2020), [Online]. Available: https://www.altexsoft.com/blog/what-is-data-engineer-role-skills/.

[7] «Data engineering and its main concepts: Explaining the data pipeline, data warehouse, and data engineer role». Last Accessed 6 January 2023. (Aug. 2021), [Online]. Available: https://www.altexsoft.com/blog/datascience/what-is-data-engineering-explaining-data-pipeline-data-warehouse-and-data-engineer-role/.

[8] M. Beauchemin. «The rise of the data engineer». Last Accessed 6 January 2023. (Jan. 2017), [Online]. Available: https://medium.com/free-code-camp/the-rise-of-the-data-engineer-91be18f1e603.

[9] M. H. Joe Reis, *Fundamentals of Data Engineering.* O'Reily Media Inc., Jun. 2022, Last Accessed 10 January 2023.

[10] B. Lutkevich. «What is a data engineer and what do they do?» (Mar. 2020), [Online]. Available: https://www.techtarget.com/searchdatamanagement/definition/data-engineer.

[11] J. Anderson. «The two types of data engineering». Last Accessed 6 January 2023. (Jun. 2018), [Online]. Available: https://www.jesse-anderson.com/2018/06/the-two-types-of-data-engineering/.

[12] J. Kutay. «What is data ingestion and why this technology matters». (May 2021), [Online]. Available: https://www.striim.com/blog/what-is-data-ingestion-and-why-this-technology-matters/.

[13] J. Kutay. «Data transformation 1010: The what, why, and how». (Aug. 2021), [Online]. Available: https://www.striim.com/blog/data-transformation-101-the-what-why-and-how/.

[14] J. Kutay. «Data wharehouse vs data lake vs data lakehouse». (Nov. 2021), [Online]. Available: https://www.striim.com/blog/data-warehouse-vs-data-lake-vs-data-lakehouse-an-overview/.

[15] P. Russom *et al.*, «Big data analytics», *TDWI best practices report, fourth quarter*, vol. 19, no. 4, pp. 1–34, 2011.

[16] T. M. Mitchell and T. M. Mitchell, *Machine learning.* McGraw-hill New York, 1997, vol. 1.

[17] K. Kirwan. «What is reverse etl? a complete guide + best tools | twilio segment blog». (Jul. 2022), [Online]. Available: https://segment.com/blog/reverse-etl/.

[18] E. Eryurek, U. Gilad, V. Lakshmanan, A. Kibunguchy, and J. Ashdown, *Data governance: The definitive guide: People, processes, and tools to operationalize data trustworthiness.* O'Reilly Media, Inc., 2021.

[19] P. B. Team. «What is data modeling: Microsoft power bi». (), [Online]. Available: `https://powerbi.microsoft.com/en-us/what-is-data-modeling/`.

[20] «What is data lineage?» (), [Online]. Available: `https://www.ibm.com/topics/data-lineage`.

[21] J. Ereth, «Dataops-towards a definition.», *LWDA*, vol. 2191, pp. 104–112, 2018.

[22] «What is a data architecture?» (), [Online]. Available: `https://www.ibm.com/topics/data-architecture`.

[23] «What is orchestration?» (Apr. 2022), [Online]. Available: `https://www.databricks.com/glossary/orchestration`.

[24] T. Contributor. «What is software engineering?: Definition from techtarget». (Nov. 2016), [Online]. Available: `https://www.techtarget.com/whatis/definition/software-engineering`.

[25] P. Ghosh. «Data architect vs data engineer». Last Accessed 6 January 2023. (Nov. 2021), [Online]. Available: `https://www.dataversity.net/data-architect-vs-data-engineer/`.

[26] J. Kutay. «Data architect vs. data engineer: An overview of two in-demand roles». (Nov. 2021), [Online]. Available: `https://www.striim.com/blog/data-architect-vs-data-engineer-an-overview-of-two-in-demand-roles/`.

[27] T. Grey. «5 principles for cloud-native architecture - what is and how to master». Last Accessed 6 January 2023. (Jun. 2019).

[28] W. Abramowicz and R. Corchuelo, Eds., *Business Information Systems*. Springer International Publishing, 2019, vol. 353, check parts<br/> decision-support for selecting big data reference architectures<br/> an ICT project case study from education<br/> towards an optimized cloud-agnostic deployment of hybrid applications, ISBN: 978-3-030-20484-6. DOI: `10.1007/978-3-030-20485-3`. [Online]. Available: `http://link.springer.com/10.1007/978-3-030-20485-3`.

[29] M. Volk, S. Bosse, D. Bischoff, and K. Turowski, «Decision-support for selecting big data reference architectures», in *Business Information Systems*, W. Abramowicz and R. Corchuelo, Eds., Cham: Springer International Publishing, 2019, pp. 3–17, ISBN: 978-3-030-20485-3.

[30] D. Nelson, «Distributed computing: The unsung hero of the modern global economy», *IEEE Potentials*, vol. 37, pp. 12–16, 4 Jul. 2018, ISSN: 15581772. DOI: `10.1109/MPOT.2018.2824378`.

[31] «Kappa architecture - where every thing is a stream». (), [Online]. Available: `http://milinda.pathirage.org/kappa-architecture.com/`.

[32] D. Dhanushka. «Anatomy of an event streaming platform — part 1 | by dunith dhanushka | event-driven utopia | medium». (Apr. 2021), [Online]. Available: `https://medium.com/event-driven-utopia/anatomy-of-an-event-streaming-platform-part-1-dc58eb9b2412`.

[33] D. Dhanushka. «Comparing enterprise messaging vs event streaming | event-driven utopia». (Jan. 2021), [Online]. Available: `https://medium.com/event-driven-utopia/comparing-enterprise-messaging-and-event-streaming-e714f7b5fc40`.

[34] «Event-driven architecture and pub/sub pattern explained | altexsoft». (2021), [Online]. Available: `https://www.altexsoft.com/blog/event-driven-architecture-pub-sub/`.

[35] C. Wang. «What is the modern data stack? | blog | fivetran». (Aug. 2021), [Online]. Available: `https://www.fivetran.com/blog/what-is-the-modern-data-stack`.

[36] «What is a data stack? | modern data stack explained | mongodb». (), [Online]. Available: `https://www.mongodb.com/basics/data-stack`.

[37] M. Wu. «What's so modern about the modern data stack? - neptune.ai». (Nov. 2022), [Online]. Available: `https://neptune.ai/blog/modern-data-stack`.

[38] T. Jaipuria. «Understanding the modern data stack - by tanay jaipuria». (Jun. 2022), [Online]. Available: `https://tanay.substack.com/p/understanding-the-modern-data-stack`.

[39] «Data hub purpose and architecture overview | altexsoft». (2021), [Online]. Available: `https://www.altexsoft.com/blog/data-hub/`.

[40] A. B. MIT, A. J. E. U. Chicago, D. K. MIT, *et al.*, «Collaborative data analytics with datahub harihar subramanyam mit», 2150.

[41] J. Serra. «Data fabric defined | james serra's blog». (2020), [Online]. Available: `https://www.jamesserra.com/archive/2021/06/data-fabric-defined/`.

[42] «Data fabric, explained | altexsoft». (2022), [Online]. Available: `https://www.altexsoft.com/blog/data-fabric/`.

[43] N. G. Kuftinova, O. I. Maksimychev, A. V. Ostroukh, A. V. Volosova, and E. N. Matukhina, «Data fabric as an effective method of data management in traffic and road systems», Institute of Electrical and Electronics Engineers Inc., 2022, ISBN: 9781665406352. DOI: `10.1109/IEEECONF53456.2022.9744402`.

[44] A. Gupta. «Using data fabric architecture to modernize data integration». (May 2015), [Online]. Available: `https://www.gartner.com/smarterwithgartner/data-fabric-architecture-is-key-to-modernizing-data-management-and-integration`.

[45] «Data mesh concept and principles | altexsoft». (2022), [Online]. Available: `https://www.altexsoft.com/blog/data-mesh/`.

[46] Z. Dehghani. «Data mesh principles and logical architecture». (Mar. 2020), [Online]. Available: `https://martinfowler.com/articles/data-mesh-principles.html`.

[47] D. Kreuzberger, N. Kühl, and S. Hirschl, *Machine learning operations (mlops): Overview, definition, and architecture*, 2022. DOI: `10.48550/ARXIV.2205.02302`. [Online]. Available: `https://arxiv.org/abs/2205.02302`.

[48] I. Karamitsos, S. Albarhami, and C. Apostolopoulos, «Applying devops practices of continuous automation for machine learning», *Information*, vol. 11, no. 7, p. 363, Jun. 2020, ISSN: 2078-2489. DOI: `10.3390/info11070363`. [Online]. Available: `http://dx.doi.org/10.3390/info11070363`.

[49] L. E. Lwakatare, I. Crnkovic, E. Rånge, and J. Bosch, «From a data science driven process to a continuous delivery process for machine learning systems», in *Product-Focused Software Process Improvement*, M. Morisio, M. Torchiano, and A. Jedlitschka, Eds., Cham: Springer International Publishing, 2020, pp. 185–201, ISBN: 978-3-030-64148-1.

[50] L. E. Lwakatare, I. Crnkovic, and J. Bosch, «Devops for ai - challenges in development of ai-enabled applications», in *2020 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2020, pp. 1–6. DOI: `10.23919/SoftCOM50211.2020.9238323`.

[51] A. M. Domenech and A. Guillén, «Ml-experiment: A python framework for reproducible data science», *Journal of Physics: Conference Series*, vol. 1603, no. 1, p. 012 025, Sep. 2020. DOI: `10.1088/1742-6596/1603/1/012025`. [Online]. Available: `https://dx.doi.org/10.1088/1742-6596/1603/1/012025`.

[52] O. Spjuth, J. Frid, and A. Hellander, «The machine learning life cycle and the cloud: Implications for drug discovery», *Expert Opinion on Drug Discovery*, vol. 16, no. 9, pp. 1071–1079, 2021, PMID: 34057379. DOI: `10.1080/17460441.2021.1932812`. eprint: `https://doi.org/10.1080/17460441.2021.1932812`. [Online]. Available: `https://doi.org/10.1080/17460441.2021.1932812`.

[53] Y. Zhou, Y. Yu, and B. Ding, «Towards mlops: A case study of ml pipeline platform», in *2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE)*, 2020, pp. 494–500. DOI: `10.1109/ICAICE51518.2020.00102`.

[54] J. Baranda, J. Mangues-Bafalluy, E. Zeydan, *et al.*, «Demo: Aiml-as-a-service for sla management of a digital twin virtual network service», Institute of Electrical and Electronics Engineers Inc., May 2021, ISBN: 9781665404433. DOI: `10.1109/INFOCOMWKSHPS51825.2021.9484610`.

[55] S. Mäkinen, *Designing an open-source cloud-native mlops pipeline*, 2021. [Online]. Available: `http://www.cs.helsinki.fi/`.

[56] X. Li and B. Zou, «An automated data engineering pipeline for anomaly detection of iot sensor data», Sep. 2021. [Online]. Available: `http://arxiv.org/abs/2109.13828`.

[57] *The smack stack - o'reilly*. [Online]. Available: `https://www.oreilly.com/radar/the-smack-stack/`.

[58] *Why can't i connect to kafka? ||troubleshootconnectivity.* [Online]. Available: https://www.confluent.io/blog/kafka-client-cannot-connect-to-broker-on-aws-on-docker-etc/.

[59] *Kubernetes in action.* [Online]. Available: https://learning.oreilly.com/library/view/kubernetes-in-action/9781617293726/.

[60] Tange, *Gnu parallel 20230922 ('derna')*, Sep. 2023. DOI: 10.5281/zenodo.8374296. [Online]. Available: https://doi.org/10.5281/zenodo.8374296.

[61] *Kube-state-metrics-v2 ||grafanalabs.* [Online]. Available: https://grafana.com/grafana/dashboards/13332-kube-state-metrics-v2/.

[62] *Strimzi kafka exporter ||grafanalabs.* [Online]. Available: https://grafana.com/grafana/dashboards/11285-strimzi-kafka-exporter/.

[63] *Cassandra dashboard ||grafanalabs.* [Online]. Available: https://grafana.com/grafana/dashboards/12086-cassandra-dashboard/.

[64] *Intel core i78750h processor 9m cache up to 4.10 ghz product specifications.* [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/134906/intel-core-i78750h-processor-9m-cache-up-to-4-10-ghz.html.

[65] *Compute - amazon ec2 instance types - aws.* [Online]. Available: https://aws.amazon.com/ec2/instance-types/.

[66] *Tf.config.experimental.enable_op_determinism ||tensorflowv*2.14.0. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/config/experimental/enable_op_determinism.

[67] *Reproducibility in keras models.* [Online]. Available: https://keras.io/examples/keras_recipes/reproducibility_recipes/.

[68] *Kafka | datagrip documentation.* [Online]. Available: https://www.jetbrains.com/help/datagrip/big-data-tools-kafka.html.

[69] *How discord stores trillions of messages.* [Online]. Available: https://discord.com/blog/how-discord-stores-trillions-of-messages.