Design of a monitoring system for distributed data processing pipelines

ECiDA platform monitoring

Ignacio García Dachary

Distributed Systems
University of Groningen
The Netherlands
July 2023

Acknowledgements

I would like to express my gratitude to the following individuals and groups, who have played a significant role in the completion of this thesis:

Firstly, thanks to the Distributed Systems research group and everyone who is part of it for their support and collaboration. I especially want to thank Mostafa Hadadian and Alexander Lazovik for their guidance and assistance throughout the entire research process.

To the friends I made during my Erasmus exchange in Groningen, thank you for your invaluable support and help. I am grateful for the opportunity to have met you and for the unforgettable experiences we shared. Without you, this would not have been possible.

To my friends back home, thank you for always being by my side and supporting me. Your presence in my life is incredibly important, and I am grateful for your friendship and encouragement.

Lastly, I want to express my deepest gratitude to my family. Your love, support, and belief in me have been my driving force. I am grateful for the values you have taught me and for always pushing me to be my best.

Abstract

Distributed data processing pipelines allow businesses to reduce the time needed to get models in production, fostering accelerated innovation and breakthroughs in many fields. Monitoring is crucial for ensuring the health and optimal performance of these systems. This thesis aims to provide a blueprint for designing a monitoring system for distributed data processing systems, specifically targeting the ECiDA platform, which facilitates the design and deployment of such pipelines. A literature and documentation review was done to define how these systems should be monitored. Then, a software architecture for the monitoring system was defined and, by reviewing and comparing available open-source tools, a possible implementation of the defined architecture was proposed. It has been observed that monitoring is highly dependent on the specific use case, problem nature, data format, and model type, among other factors. However, the proposed architecture can serve as a foundation for developing a monitoring system for systems like ECiDA, with minor implementation details varying based on each specific problem and context.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This section serves as a gateway, inviting readers to explore the research's purpose and underlying framework. It sets the stage by providing a comprehensive overview of the context and motivation behind the project, and its objectives and scope.

1.1 Context and motivation

This project falls under the umbrella of a broader project called Evolutionary Changes in Data Analysis (ECiDA) [1, 2], currently under development at the University of Groningen.

Contemporary data analysis platforms often face limitations when it comes to adapting to changes in both the application and the underlying data flow, usually not being able to update individual components of running pipelines without requiring a restart or downtime. Additionally, they don't consider that data sources may change during the usage period. These assumptions do not align with the present reality of data science. Companies often confront a dilemma: either accept downtime during pipeline updates or invest in duplicating their expensive infrastructure to ensure uninterrupted pipeline operations.

ECiDA platform's primary objective is to bridge the gap between engineers, responsible for developing large-scale computation platforms, and data scientists, who handle vast amounts of data, all while prioritizing the ability to make changes dynamically.

With ECiDA, data scientists can build their data science pipelines on scalable infrastructures and modify them without disruption, allowing them to adjust parameters within individual pipeline components or make changes to the network topology as needed. With this flexibility, ECiDA empowers pipelines to initiate changes as part of a diagnostic response.

To ensure the platform's consistency during updates, ECiDA incorporates automatic formal verification methods such as constraint programming and AI planning. These methods transparently validate updates, ensuring the desired behaviour.

By combining the ability to make changes with scalable infrastructure and verification mechanisms, ECiDA addresses the limitations of existing data analysis platforms and provides a comprehensive solution for data scientists and companies involved in data science analyses.

To achieve this, ECiDA uses the next methods:

- **Modularity:** a module is defined as a self-contained component or unit that performs specific functions and interacts with other modules to collectively achieve system functionality. It serves for the reusability and transparency of the system.
- **AI planning:** design an application according to the given goals using existing modules. It can design one or multiple applications that fulfil the requirements.
- **Recommender system:** enables to pick the best-performing application according to the context and goals.
- **Evolution and lifecycle:** handle rapid updates and changes in the environment, data and system.

In the context of this project, it is needed to further explain in more detail the mentioned modularity approach.

Everything-as-a-Module, or XaaM, tries to apply service-oriented computing principles to machine learning development. ECiDA, which aims to integrate machine learning and service-oriented software development, places a strong emphasis on modularity as a core design principle. The platform emphasises the value of modularity and seeks to advance best practices in this area.

Software development has been significantly impacted by service-oriented computing, which is now the industry norm. It promotes flexibility, maintainability, and communication among software engineering teams through efficient component reuse and service abstraction. The newly growing subject of MLOps seeks to extend collaboration and efficiency by bringing productive DevOps practises to machine learning development.

In order to encourage service-oriented computing in machine learning, XaaM suggests a higher level of abstraction known as the ”Module”. Machine learning services are distinguished from conventional web services by this term. The encapsulation of multiple machine learning components, such as code, pipelines, and datasets, is made easier thanks to the Module abstraction.

Finally, it is worth mentioning the difference between Atomic Modules and Compound Modules:

- **Atomic Module:** self-contained module with no dependencies on other modules. This is the smallest level of abstraction.
- **Compound Module:** composed of multiple smaller modules, allowing for the construction of larger, interconnected systems.

Once defined what ECiDA aims to accomplish and the principles behind it, the focus will be set on the monitoring aspect of ECiDA, which is the main point of the current project.

1.2 Objectives and scope

The objectives of this thesis are the following:

- **Monitoring practices for ML pipelines:** define what should be monitored in a distributed ML pipeline to guarantee its correct performance and be able to detect any existing issues.
- **Architecture for monitoring of ECiDA pipelines:** propose a generic software architecture for a monitoring system for ECiDA, capable of gathering the necessary information about the running pipelines and their modules and storing it in an organised way to later be able to access it for further purposes.
- **Architecture implementation:** give an implementation proposal for the defined monitoring architecture, giving specific tools for each of its components.

It is out of the scope of the present thesis the development of the proposed solution. In the same way, the definition of the visualization, alerting and recommender system is also out of scope.

This thesis defines what data should be collected from the running pipelines and how it is managed and stored to later be able to access it.

1.3 Structure of the thesis

The thesis is structured in the following way. Chapter 2 exposes the necessary foundations and concepts needed to understand the content of the thesis. Chapter 3 reviews the related work to this thesis, analysing similar work done in the field and finding the gap where this work fits in. Chapter 4 proposes the research questions and explains the methodology followed to answer them. Chapter 5 presents the results of the research work and answers the formulated research questions. Chapter 6 discusses the results and limitations of the present research. Finally, Chapter 7 summarizes the results of the thesis and proposes possible future lines of investigation or next steps.

Chapter 2

Background

This chapter discusses the main concepts needed to understand the content of the present thesis.

2.1 DevOPS

DevOps can be defined as "*collaborative and multidisciplinary organizational effort to automate continuous delivery of new software updates while guaranteeing their correctness and reliability*" [3] or, more specifically, "*a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality*" [4]. This entails higher collaboration between operators and developers in companies, unlike traditional practices where developers continuously tried to push new versions to production while operators' main objective was to maintain the service running, blocking or delaying these new changes. Thus, the application of DevOps involves a technical and cultural transformation among companies and teams.

DevOps enables, among others, collaborative and continuous development, continuous integration and testing, continuous release and deployment, continuous infrastructure monitoring and optimization, continuous user behaviour monitoring and feedback, service failure recovery without delay, and continuous measurement [5].

In this context and in line with the topic of this thesis, it is worth mentioning microservices as one of the enablers of DevOps. Microservices are "*an architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs*" [6] and are standardizing the building of continuously deployed systems [7]. Microservices, unlike traditional monolith architectures, allow for more flexibility and scalability, fostering faster and more efficient development of applications.

2.2 Machine Learning

Machine Learning (ML) is a branch of Artificial Intelligence (AI) with the objective of enabling computers to make predictions about the future using historical data from the past [8].

The machine learning lifecycle refers to the series of steps involved in developing, deploying, and maintaining an ML system. There is no unique definition of the ML lifecycle, due to variations being subject to the specific use case, industry or application. Nevertheless, a generic ML lifecycle can be defined from the commonalities found in a diverse range of literature [9–13]. The ML lifecycle can be depicted in Figure 2.1.

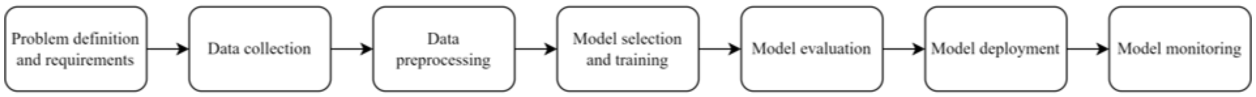


Figure 2.1: ML lifecycle

Where:

- **Problem definition and requirements:** clearly articulate the problem at hand, specifying the objectives, limitations, and criteria for measuring success.
- **Data collection:** collect relevant data necessary for training and evaluating the model in order to address the problem.
- **Data preprocessing:** cleanse and preprocess the data by addressing missing values, outliers, inconsistencies, data formats, etc. Extract and select relevant features.
- **Model selection and training:** select an appropriate model or algorithm based on the problem definition and data format. Split the data into training and testing sets and train the model, fine-tuning hyperparameters to optimize its performance.
- **Model evaluation:** evaluate the model's performance using suitable metrics and make necessary adjustments to the model if required.
- **Model deployment:** integrate the trained model into the production environment for making predictions on new data.
- **Model monitoring:** continuously monitor the deployed model's performance to identify potential issues such as model staleness or degraded performance, indicating the need for retraining.

The machine learning lifecycle is an iterative process. For this reason, phases may be reviewed or redone as necessary to enhance the model's performance and take into account new challenges or requirements [9].

2.3 MLOps

The increase in the amount of data used in ML models and the models themselves are the leading causes for the need for highly scalable ML systems [14]. This need for scalability brings new challenges compared to traditional centralized systems when designing and implementing a system's architecture. MLOps seeks to solve this issue.

By combining DevOps and ML development [10] together with Data Engineering [15] (Figure 2.2), MLOps aims to apply DevOps best practices to ML development, with the objective of deploying ML systems in a reliable and efficient way [9] while adapting to changes, increasing amounts of data and problem complexity, and the need for faster processing times and efficiency.

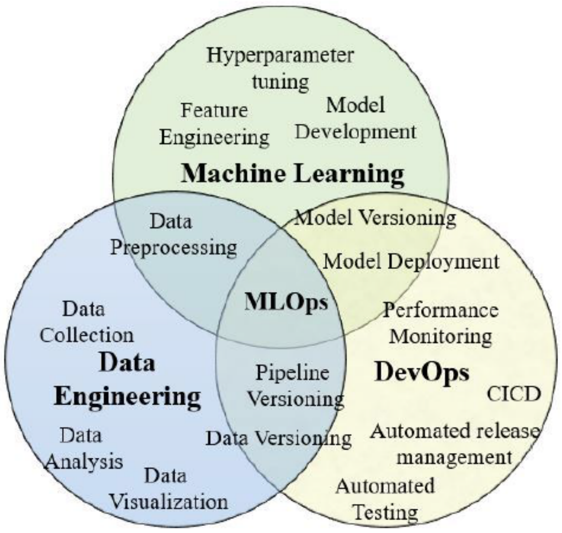


Figure 2.2: MLOps combination [15]

2.3.1 Principles

The main MLOps principles, which are the enablers for developing and deploying ML solutions flexibly and efficiently, are the following [9–11, 13, 16]:

- **Continuous Integration (CI):** guarantees the regular integration and testing of machine learning models and their related code. It involves automating the construction and verification procedures to detect integration problems, resulting in a reliable and updated codebase.
- **Continuous Delivery (CD):** ensures consistent building, testing, and deployment of machine learning models, as well as their supporting infrastructure. It helps teams in

minimizing manual mistakes, accelerating time to market, and enabling iterations and feedback loops.

- **Continuous Training (CT):** involves the automation of the retraining and updating process of models, with the aim to maintain their accuracy and performance over time. By implementing CT, model retraining is initiated whenever new data is obtained or at regular intervals, allowing models to adjust to evolving data patterns.
- **Continuous Monitoring (CM):** consists of the monitoring of deployed models to identify anomalies or performance degradation. This involves monitoring input data, model predictions, and model output to ensure the models are operating as intended. Whenever an anomaly is detected, alerts are sent, enabling teams to investigate and resolve any existing issues.

These principles entail many new challenges. Nevertheless, the focus will be set on the monitoring aspect, according to the topic of this thesis.

2.3.2 Monitoring

Monitoring needs special attention [16] and is one of the most important functions in MLOps [10]. For this reason, a monitoring system, with great visualization capabilities, is required [15] in MLOps. This is with the objective of monitoring everything related to the performance of the application, including the application's performance itself, as well as the model, data and infrastructure health [9] after the deployment phase, to be able to detect any anomalies and be able to address any related issues [15].

Chapter 3

Related work

This section discusses the related work relevant to this study in the field of monitoring distributed machine learning systems. Despite distributed systems and ML workflows being two concepts that can work together, they are not linked by default as, for instance, ML workflows can work in centralized systems. As a consequence, the literature addressing specifically the concept of (monitoring) distributed ML systems is limited. Instead, the majority of the literature available addresses MLOps, which encompasses the areas of DevOps, data engineering and machine learning. Nevertheless, this information is valuable as it can be extrapolated to distributed systems. On the other hand, literature about monitoring all-purpose distributed systems is also available.

A non-exhaustive list of relevant literature addressing the monitoring of distributed machine learning systems is reviewed below.

3.1 Importance of monitoring distributed ML systems

In an article published in 2021 by Ruf et al. [9], the different phases of an MLOps workflow are defined, as well as the needed supportive tools to guarantee a successful implementation of the system. Regarding monitoring, the article highlights its need in the training and deployment phases. Regarding the training phase, it declares that tracking model-related information, such as hyperparameters and model performance metrics, is essential to compare between runs and thus select the best model candidate. On the other hand, it states that constant monitoring of the whole system in the deployment phase is required to guarantee a robust ML product. This monitoring should cover the areas of system monitoring, to evaluate the health and status of the infrastructure and detect possible deployment errors; input data monitoring, to detect possible input-data-related issues, such as data drift or outliers; and model performance monitoring, to ensure the effectiveness of the model over time (avoiding model staleness) and detect the possible need to re-train the model.

Kreuzberger et al. [11] carry out an overview of MLOps, giving an overview of its principles, components, and roles. It defines nine principles or "best practices" needed to realize in MLOps, being one of them continuous monitoring of data, model, code, infrastructure resources, and model serving performance. It again highlights the importance of monitoring both the ML workflow and the infrastructure components to guarantee optimal performance.

Similarly, in a paper published in 2022 by Symeonidis et al. [10], an overview and definition of the phases of an MLOps system are conducted. It states that most papers and articles consider the monitoring of the ML models and every other aspect of the system, one of the most important tasks in MLOps.

Once set the importance of monitoring both ML workflows and the infrastructure where these run, a review of literature addressing both these topics is conducted.

3.1.1 Monitoring ML workflows

In an article published in 2022, Paleyes et al. [17] discuss and map the main challenges found in the deployment of ML models in production in many different use cases, industries and applications. One of the challenges found in the model deployment phase is monitoring, necessary for the maintenance of ML systems. Nevertheless, it states that the "ML community is in the early stages of understanding what are the key metrics of data and models to monitor and how to trigger system alarms when they deviate from normal behavior".

Klaise et al. [18] discuss the areas and challenges around monitoring and explaining ML models in deployment. Regarding monitoring, they identify the key areas to guarantee a successful application. These are model performance monitoring, input data metrics monitoring and detection of outliers and data drift. Considering that the monitoring solution will work alongside the running models, they highlight the importance of designing this solution in a way that does not affect the model's core performance. This article, however, does not give specific examples of ways or metrics to carry out the monitoring in each of the mentioned areas.

Schöder & Schulz [19] review the main V&V (verification and validation) challenges of ML systems. These are high dimensionality, dataset shift, model robustness, system interdependence, result communication, test design and validity. Then, monitoring is proposed as a (partial) solution for these challenges. This monitoring should cover model monitoring and data monitoring. Finally, specific examples of metrics and ways of monitoring both the model and the data are given.

3.1.2 Monitoring distributed systems

Monitoring the different components involved in a distributed system to ensure everything works as expected is a critical task [20] [21], as important as monitoring the ML workflow. Literature on this topic can be found.

Kufel [22] provides an overview of approaches for monitoring distributed systems. It enumerates the fundamental elements that need to be taken into consideration when designing a monitoring solution for a distributed system. These are:

- **Monitoring layers:** where the data is collected from
- **Events:** time-stamped information about specific metrics, such as CPU utilization
- **Thresholds:** pre-defined values to trigger alerts depending on the importance of the component
- **Polling intervals:** the interval between which monitoring events are recorded
- **Data retention time:** how long the monitoring data is stored

It then states the main areas of monitoring a distributed system are capacity and availability monitoring, being availability a percentage measure of the time that the application has been running over the sampling time, and capacity assesses the level of utilization of the system resources. This article, in spite of giving a useful overview of the elements that should be considered when designing a monitoring solution for distributed systems, does not offer insightful information about what specific metrics or measures should be monitored in these systems.

Silva et al. [20] give more detailed information about what to monitor in distributed systems and enumerate five main observations that need to be considered. These are task completion time, CPU and GPU usage, memory usage, disk input/output (IO) operations, and network traffic. It states that monitoring these concepts helps identify problems and provides valuable data to support the deployment of the models.

Ruf et al. [9] go a step further and give detailed information about what metrics to monitor and why through all phases of an MLOps workflow. Regarding the deployment infrastructure monitoring, lists some metrics matching the ones listed by Silva et al. [20], such as CPU and GPU usage, and disk utilization; and identifies some new ones, these being serving latency, throughput, system uptime and the number of API calls.

The literature addressing this matter provides a valuable overview of what should be monitored in a distributed system. Nevertheless, the monitoring of distributed systems still depends primarily on the type of distributed system, in the first place, and on the specific tools or techniques used to implement the system. Thus, a more precise literature review is conducted down below according to the scope of this project.

3.1.2.1 Monitoring microservices systems - Docker and Kubernetes

Considering what has been said, the focus is now set on monitoring microservices systems, a case of distributed systems. Particularly, a microservices system based on Docker and Kubernetes for containerization and orchestration tasks, respectively, as these are the most commonly used tools for these purposes [23] [24].

Mart et al. [25] addresses the topic of automatic anomaly detection in a Kubernetes cluster. To do so, it first presents the concept of observability in distributed systems and a Kubernetes cluster in particular. It defines observability as "a measure of how well internal states of a system can be inferred based on external outputs". In the case of distributed systems, observability is based on three different measures: logs, metrics and traces, these being:

- **Logs:** timestamped records of the system events that have occurred
- **Metrics:** numerical measurements that track the performance of the system over time
- **Traces:** a detailed record of the path that a particular request or transaction has taken through the system

It sets metrics as the main piece of monitoring, as they allow set alerts and notifications about possible issues present in the system. However, this article fails to describe the specific metrics needed to monitor a Kubernetes cluster and evaluate its health and performance, as well as the components and strategy needed to monitor Kubernetes metrics in the first place.

Hassan & Abdullah [26] go a step further and centre their work on the design of a specific solution for monitoring a Kubernetes cluster using Prometheus, Grafana, and other tools. They explicitly define the 15 resource metrics considered when designing the monitoring solution, divided into pod-level and node-level metrics. Regarding the monitoring solution itself, each of the components involved and the strategy to connect them is explained. Despite this article showing a monitoring solution, it is a specific solution with programming behind it and does not explicitly present the code, and how everything is connected and works together.

Alternatively, Großmann & Klug [27] provide a lightweight prototype solution called *Pymon* for monitoring resource consumption, CPU and memory usage and network traffic, of Docker cluster components.

3.2 Open-source monitoring tools

Finally, a review of the literature on solutions based on open-source tools for the monitoring of ML workflow and distributed systems is conducted.

3.2.1 ML workflow monitoring

Ruf et al. [9] conduct an overview and comparison between many different open-source tools available to help automate each phase of the MLOps workflow. It does a benchmarking of them based on the level of fulfilment (none, partial or complete) of the requirements set for each phase of the MLOps workflow: data processing, model training, model management, model deployment, operations and monitoring, and general requirements. Finally, some guidelines for the correct choice of tools are given.

Symeonidis et al. [10] present a list of available tools (private, public, and open-source) to carry out the different tasks involved in the data processing, modelling, and operationalization or deployment phases. Regarding monitoring, it highlights whether the modelling tools offer model tracking or not and whether the operationalization tools offer model monitoring or end-to-end features. It again gives guidelines on how to choose the right tools for a given project.

Idowu et al. [28] carry out an in-depth review and comparison of 18 state-of-research and 12 state-of-practice tools for ML asset management.

Kufel [22] gives guidelines to select a suitable monitoring toolset for monitoring distributed systems performance and conducts a review of available open-source and proprietary tools to carry out this task.

Kreuzberger et al. [11] give some examples of open-source and non-open-source monitoring tools that can be implemented to execute this task.

3.2.2 Distributed system monitoring

Mart et al. [25] focuses on the designing of an automatic model for alerting errors, using Prometheus [29] and Grafana [30], based on the forecasting of values, instead of the usually used hard-coded thresholds.

Hassan & Abdullah [26] take a similar approach and design a monitoring solution for a Kubernetes cluster based on Prometheus, Grafana and other tools.

3.2.3 Tools review and comparison

In this context, an own search and review of the available tools to monitor a distributed ML system are conducted. This is with the objective of finding a gap in currently available tools and, thereafter, defining a software architecture to have as a base for designing and implementing a monitoring system for this work.

There are plenty of available tools for managing and monitoring an ML workflow, as well as for monitoring the resource usage of a (distributed) system. Nevertheless, in accordance with this project, the considered tools must not be proprietary tools, paid-per-use tools, or any other paid option. They should be publicly available and free to use including all their features.

It is worth noting that the following comparison contains a non-exhaustive list of tools. Only tools with remarkable maturity, active community, and available documentation are considered.

To be able to compare them easily, the next features are considered to define each tool in a standardized way:

F1. **Input data monitoring in the deployment phase:** the capability to monitor the input data to detect possible data drift and outliers and evaluate the data consistency and quality.

F2. **Model tracking in the training phase:** the ability to organize, compare and visualize different runs and experiments using metrics and parameters.

F3. **Model performance monitoring in the deployment phase:** the capability to monitor the performance of a model through performance metrics (e.g.: ROC curve, AUC, confusion matrix), concept drift or model bias, between others.

F4. **Resource/Service health monitoring:** the capability to monitor the resource usage (CPU, GPU, memory, etc.) of the distributed system where the ML workflow is running to ensure its correct performance.

F5. **Data streaming support:** continuously created data by different data sources can be processed and analysed.

F6. **Threshold alerting:** alert and notification setting is supported based on pre-defined conditions and thresholds.

F7. **Library agnostic:** most used ML libraries and frameworks are supported.

F8. **Hyper-parameter tunning engine:** selecting the optimal settings for hyperparameters, which are pre-set parameters for machine learning algorithms that affect model training and performance.

F9. **CI/CD availability:** support of CI/CD integration for the whole pipeline.

F10. **Model deployment:** support for different deployment patterns (cloud, containers, etc.)

F11. **Code versioning:** support for model's code versioning.

F12. **User-friendliness / Ease of set-up:** an intuitive interface and not-complex way of set-up and configuration, as many actors are present in the MLOps workflow.

The fulfilment or not of these features will be checked for each tool. Following, a review of the considered tools can be found:

MLFLOW

https://github.com/mlflow/mlflow

Mlflow is an open-source platform to manage the end-to-end ML lifecycle. It provides tools for tracking experiments, packaging code, and sharing models across different environments. It consists of four main components [31]:

- **Tracking:** experiment tracking to compare parameters, metrics, and results.
- **Projects:** packaging ML models code in a reusable and reproducible way.
- **Models:** deploying ML models from various ML libraries to many serving platforms.
- **Registry:** central storage to manage the full lifecycle of an ML model.

Regarding monitoring, it has some features for the tracking and deployment phase. It lets organise runs in bigger sets, called experiments. It also lets one compare different runs, comparing metrics and parameters via a parallel coordinates graph. It also has the possibility to build other graphs like bar charts scatter charts or contour charts. Nevertheless, the monitoring aspect of Mlflow is not as advanced as its other features.

AIM

https://github.com/aimhubio/aim

Aim is an open-source platform for managing and visualizing machine learning experiments. It is designed to help data scientists and machine learning engineers track, compare, and reproduce different machine learning experiments, and to gain insights into the performance and behaviour of their models [32].

It consists of a server to manage the storage and retrieval of experimental data; and a web UI, to inspect individual runs, and compare different runs and experiments through dashboards.

Some of the key features and benefits of Aim include:

- Automatic tracking and logging of experiment data
- Visualization of experiment data through charts and graphs
- Comparison of experiment results across different metrics and runs
- Integration with popular machine learning frameworks and tools

It integrates seamlessly with Mlflow, creating Aimlflow [33]. This solution sets the base with the tracking and deploying features of Mlflow and takes the monitoring aspect of Mlflow to the next level using Aim.

TFX (TENSORFLOW EXTENDED)

https://github.com/tensorflow/tensorflow

https://github.com/tensorflow/tfx

TFX (TensorFlow Extended) is a TensorFlow-based end-to-end platform for deploying machine learning models in production. It offers a selection of libraries, instruments, and best practices for developing scalable and robust machine learning (ML) pipelines that can manage large amounts of pre-process data, train models, and deploy them in production. TensorFlow Data Validation (TFDV), TensorFlow Transform (TFT), and TensorFlow Model Analysis (TFMA) are TFX components that, respectively, simplify the processes of data pre-treatment and validation, transformation and feature engineering, and model evaluation. TFX also allows model versioning, distributed training, and integration with multiple cloud platforms. TFX is an effective tool for developing and implementing scalable, reliable machine learning systems.

Although it offers several interfaces and APIs to enable integration with other machine learning libraries, TFX is primarily designed to support TensorFlow-based machine learning pipelines. However, it can be tailored to work with any machine learning framework that can be used with Python. When integrating with non-TensorFlow libraries, some TFX components might need to be modified or replaced, and other TFX functionalities might not be available or require additional work.

To help monitor the effectiveness and general condition of machine learning models in use, TFX offers several monitoring options. The essential monitoring elements in TFX include:

- **TensorBoard:** a web-based visualisation tool to present real-time performance metrics and graphs of model training and evaluation.
- **TensorFlow Model Analysis (TFMA):** offers metrics for evaluating models, such as precision, recall, accuracy, F1 score, and more. TFMA can produce reports, visualise model performance, and contrast the performance of several models.

KUBEFLOW

https://github.com/kubeflow/kubeflow

Kubeflow is an open-source platform "dedicated to making deployments of machine learning (ML) workflows on Kubernetes simple, portable and scalable" [34].

Kubeflow provides monitoring features that enable users to track and monitor the health and performance of machine learning models in production. These are:

- **Metrics and logging:** built-in support for metrics and logging, allowing users to collect and visualize data on model performance, system utilization, and other metrics. This helps identify issues with model training or deployment.
- **Visualization:** visualization tools that allow users to monitor and visualize the performance of ML models in real time. For example, Kubeflow TensorBoard provides a rich set of visualizations that enable users to track model performance, analyse training data, and diagnose issues.

POLYAXON

https://github.com/polyaxon/polyaxon

Polyaxon is an open-source framework for maintaining and deploying machine learning models on Kubernetes. From data preparation and model training to model versioning and deployment, it offers a variety of tools and capabilities to optimise the machine learning workflow.

It is intended to make the process of designing, training, and deploying machine learning models at scale easier, by offering a web-based dashboard for managing experiments, monitoring parameters, and displaying outcomes.

One of the most important characteristics of Polyaxon is its support for hyperparameter tuning, which is the process of determining the best values for the parameters that constitute a machine learning model.

Additionally, it offers several tools for model deployment and versioning. It encourages the development of reproducible model pipelines, allowing users to replicate and test models in various settings.

Overall, Polyaxon is a capable framework for maintaining and deploying machine learning models on Kubernetes.

EVIDENTLY AI

https://github.com/evidentlyai/evidently

Evidently AI is an open-source Python library for automated machine learning (AutoML) model monitoring and analysis. It provides a range of tools and features for detecting and diagnosing issues with machine learning models, such as data drift, concept drift, and model performance degradation.

It can work alongside other end-to-end ML tools, such as Mlflow.

ZABBIX

https://github.com/zabbix/zabbix

Zabbix is an open-source network monitoring and management platform that provides a range of tools and features for monitoring the performance and availability of IT infrastructure components such as servers, applications, networks, and databases.

It integrates with Kubernetes clusters, allowing users to monitor the performance and availability of their Kubernetes infrastructure in real time. Users can leverage the Zabbix Agent to collect performance data from Kubernetes nodes and use it to generate alerts and reports. The Zabbix Agent can collect a wide range of metrics, including CPU usage, memory usage, network traffic, and disk utilization. It also provides support for the Kubernetes API, allowing users to monitor Kubernetes resources such as pods, containers, and services.

It is possible to create customized dashboards to monitor specific metrics or resources and generate reports to analyse performance trends and identify areas for improvement.

Overall, Zabbix provides a powerful and flexible platform for monitoring Kubernetes clusters. With support for the Kubernetes API, Kubernetes events and logs, and a wide range of performance metrics, Zabbix allows users to monitor their Kubernetes infrastructure in real time and troubleshoot issues quickly and efficiently.

PROMETHEUS & GRAFANA

https://github.com/prometheus/prometheus

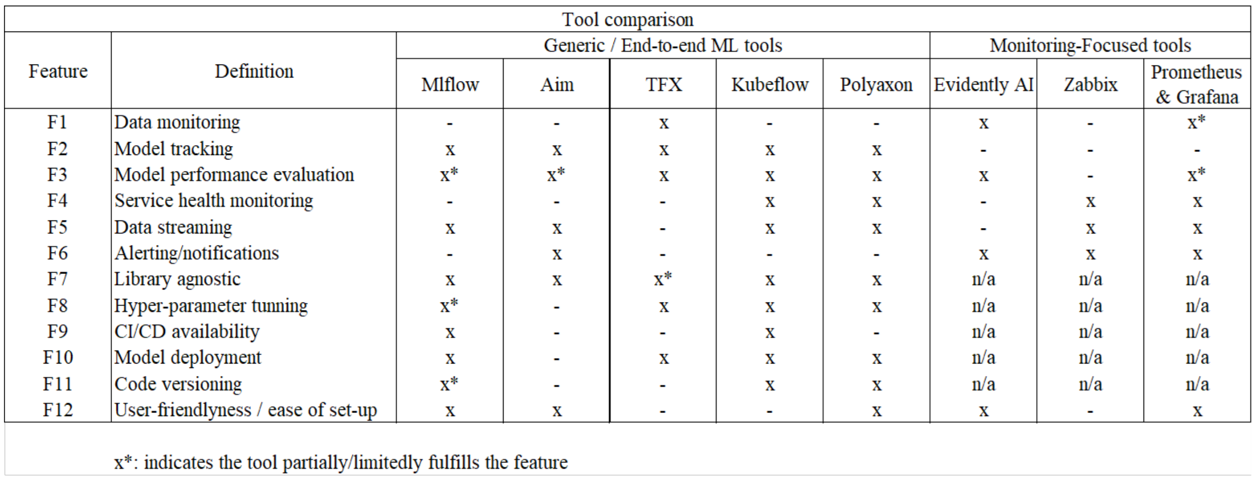
https://github.com/grafana/grafana

Prometheus and Grafana are popular open-source tools with a large and active community used for monitoring and visualizing metrics. Prometheus is a monitoring and alerting system designed to collect and store time-series data from various sources, including applications, databases, and infrastructure components. It provides a flexible query language and powerful data processing capabilities. On the other hand, Grafana is a data visualization tool that provides a variety of tools and features for visualizing and analysing time-series data. It allows users to create custom dashboards and reports, making it easier to visualize and analyse data.

When used together, Prometheus and Grafana provide a complete monitoring and visualization solution. Prometheus collects and stores metrics from various sources and makes them available for Grafana to visualize and analyse. For example, in a Kubernetes environment, Prometheus can collect and store metrics from Kubernetes components, such as nodes, pods, and containers and Grafana can then create custom dashboards and alerts to monitor the performance and availability of the Kubernetes infrastructure.

FINAL COMPARISON

Next, a summary table remarking whether each of the reviewed tools has the features defined at the beginning of this section is shown. The table has been done consulting various academic articles reviewing all or some of these tools [9, 10, 13, 20, 35–38], as well as looking into the available documentation of each tool [29–34, 39–41].



Tool comparison									
Feature	Definition	Generic / End-to-end ML tools					Monitoring-Focused tools		
		Mlflow	Aim	TFX	Kubeflow	Polyaxon	Evidently AI	Zabbix	Prometheus & Grafana
F1	Data monitoring	-	-	x	-	-	x	-	x*
F2	Model tracking	x	x	x	x	x	-	-	-
F3	Model performance evaluation	x*	x*	x	x	x	x	-	x*
F4	Service health monitoring	-	-	-	x	x	-	x	x
F5	Data streaming	x	x	-	x	x	-	x	x
F6	Alerting/notifications	-	x	-	-	-	x	x	x
F7	Library agnostic	x	x	x*	x	x	n/a	n/a	n/a
F8	Hyper-parameter tunning	x*	-	x	x	x	n/a	n/a	n/a
F9	CI/CD availability	x	-	-	x	-	n/a	n/a	n/a
F10	Model deployment	x	-	x	x	x	n/a	n/a	n/a
F11	Code versioning	x*	-	-	x	x	n/a	n/a	n/a
F12	User-friendlyness / ease of set-up	x	x	-	-	x	x	-	x

x*: indicates the tool partially/limitedly fulfills the feature

Figure 3.1: Tool comparison table

It must be made clear that there is the possibility that a certain tool in reality fulfils a feature marked as "-" in the previous table. An extensive and hands-on review should be carried out to ensure the exact capabilities of each tool. However, this overview should provide enough information and insights into the different tools to be able to propose a software architecture for an ML monitoring solution.

Finally, a list of discarded but overviewed tools can be found below. These tools have not been considered due to being proprietary, paid or not fully open-source tools:

END-TO-END / ML WORKFLOW MANAGEMENT TOOLS:

- WEIGHTS & BIASES / WANDB ( )
- NEPTUNE AI ( )
- H2O AI ( )
- COMET ML ( )
- VALOHAI ( )
- AMAZON SAGEMAKER ( )
- AZURE MACHINE LEARNING ( )
- GOOGLE CLOUD VERTEX AI ( )
- DATA ROBOT AI CLOUD ( )

MONITORING-FOCUSED TOOLS:

- ARIZE (↗)
- WHYLABS (↗)
- QUALDO (↗)
- FIDDLER (↗)
- CENSIUS (↗)
- NAGIOS XI (↗)

3.3 Summary

After reviewing the available literature addressing the topic of monitoring distributed ML systems, it is determined that monitoring is considered an important part of both ML workflows and distributed systems. The majority of articles share similar principles and measures on what should be monitored in order to guarantee optimal performance.

Nevertheless, in accordance with the reviewed literature, it is concluded that there is a need to address the topic of distributed ML systems as a whole and propose a solution to monitor both the ML workflow and the infrastructure behind it, as they are equally important tasks to ensure the correct performance of the system. Many articles focus on either the monitoring of an ML workflow or the monitoring of the computing infrastructure. Limited literature addressing the monitoring of a distributed ML system holistically has been found.

Thus, this project aims to fill this gap by focusing on a solution for monitoring a distributed ML system as a whole, taking into consideration the monitoring of the ML workflow to guarantee optimal performance of the system. Additionally, a solution based on open-source tools will be proposed.

Chapter 4

Research methodology

This section focuses on the formulation of the research questions of this project and the methodology followed to answer them.

4.1 Research questions

Considering the motivation and objectives of this project and the insights obtained from the literature review, as well as the gap found in it, the following research questions are proposed:

- **RQ1:** What needs to be monitored in a distributed ML system to guarantee its correct performance?
- **RQ2:** How to capture the results of RQ1? Proposing a software architecture and technical details with no limitations.
- **RQ3:** What are the main available tools and their features suitable for implementing the software architecture proposed in RQ2?
- **RQ4:** Based on the results of RQ2 and RQ3, to what extent can the proposed architecture for the monitoring system be implemented using open-source tools?

With these questions, the main objectives are to clearly define what and how should be monitored in a distributed ML system and to narrow down the number of open-source tools to choose between to design a monitoring system for this type of system.

4.2 Literature review

To answer the previously formulated questions, a mixed approach methodology is proposed. Firstly, a non-exhaustive literature review is conducted to gather information and try to answer the research questions. In the literature review, both academic and grey literature are considered. On the other hand, especially for the evaluation and selection of open-source tools, a documentation review together with a practical review of the main features of the different open-source tools is done.

4.2.1 Academic literature review

A non-exhaustive academic literature review is proposed for answering the formulated research questions. A systematic literature review (SLR) is dismissed due to its complexity, the need for more than one expert on the matter and for being a highly time-consuming task. Nevertheless, some best practices of an SLR can be used to carry out a non-exhaustive literature review to gain valuable insights to be able to answer the formulated research questions.

SOURCE SELECTION

Multiple electronic databases have been considered when searching for relevant information. The next sources have been used for the academic literature review:

- **Google Scholar:** https://scholar.google.com/
- **IEEE Xplore:** https://ieeexplore.ieee.org
- **ACM Digital Library:** https://dl.acm.org
- **dblp computer science bibliography:** https://dblp.org/
- **ScienceDirect - Elsevier:** https://www.sciencedirect.com

A search string has been used in all sources to collect the first relevant publications for this thesis. Advanced search mechanisms, using boolean connectors, have been used in all of them to narrow down the number of publications according to their title and content. This first search string has been:

monitoring "machine learning" mlops performance

After retrieving the publications matching this search method, inclusion and exclusion criteria have been used to decide whether a certain paper is relevant or not to answer the research questions.

INCLUSION AND EXCLUSION CRITERIA

For a paper to be considered had to fulfil, in the first place, the following requirements:

- **Language:** English
- **Publication date:** between 2015 and 2023
- **Open access:** it had to be accessible at no cost

Once an article fulfilled the previous requirements, its title and abstract were reviewed to decide if it was relevant. Furthermore, a quick keyword scanning through the article was carried out to see if it discussed the desired concepts.

SNOWBALLING

After having a first set of articles related to the thesis, forward and backwards snowballing [42] is conducted, to enlarge the number of articles to consult beyond the ones found during the initial search and expand the available information regarding the different topics to discuss and investigate through the thesis. These techniques consist of the following:

- **Backward snowballing:** consists of reviewing the reference list of the articles included in the initial search. By doing this, relevant articles related to them that have not been retrieved during the initial search can be found.
- **Fordward snowballing:** consists of looking for articles that have cited the already included articles. This may help identify newer articles that again have not been found during the initial search.

In the same way, as in the initial search, the inclusion and exclusion criteria already explained were applied to decide whether to consider a certain article or discard it.

With these practices, it is considered that enough information will be available to answer the formulated research questions.

4.2.2 Grey literature

Nevertheless, due to the topic of this thesis being relatively new, the inclusion of grey literature has also been considered, as a way of finding up-to-date information and complementing the academic literature [42].

SOURCE SELECTION

For the grey literature review, the next source has been used:

- **Google:** https://www.google.com/

INCLUSION AND EXCLUSION CRITERIA

The following requirements must be fulfilled to include a source of information for answering the research questions:

- **Language:** English
- **Formats:** 1st-grade Grey Literature according to Garousi et al.: books, magazines, government reports, companies' white papers.

Including grey literature may help fill gaps in the existing literature, as it may contain information, case studies, or research findings on topics that are not extensively covered or addressed in any formal academic publications.

4.2.3 Tools evaluation

For deciding which tools to review and evaluate for the implementation proposal of the monitoring system, both academic and grey literature have been considered. This is because, although many existing tools can be found in academic literature, such as tools review and comparison papers, many other emerging and relatively new tools may not appear in academic literature but could be useful for this use case.

On the other hand, tools' webpages, repositories, if existing, and documentation have been considered when reviewing such tools.

Chapter 5

Results

This chapter presents the results achieved in the research process and it answers the research questions RQ1, RQ2, RQ3 and RQ4 formulated in the previous chapter.

5.1 Monitoring a distributed ML system (RQ1)

When monitoring a distributed ML system, different aspects should be considered. This monitoring can be divided into two main areas, necessary to guarantee the correct performance of the whole solution [9]:

- **ML workflow monitoring:**
  - Model monitoring
  - Data monitoring
- **Resource usage monitoring**

Regarding the first point, tracking model parameters, code versions, metrics, and output files during the training phase and visualising the results allows for comparing different runs and deciding the most suitable model. Monitoring the model performance during the deployment phase also helps ensure the correct performance of the model. Finally, ensuring data quality and consistency is also necessary to ensure the models work with the correct data.

On the other hand, making sure the distributed infrastructure uses the available resources (CPU, GPU, memory, etc.) properly is another vital point to consider and to constantly monitor as it is the enabler of the ML workflow running and working in the first place. Nonetheless, this aspect of monitoring is out of the scope of the present study.

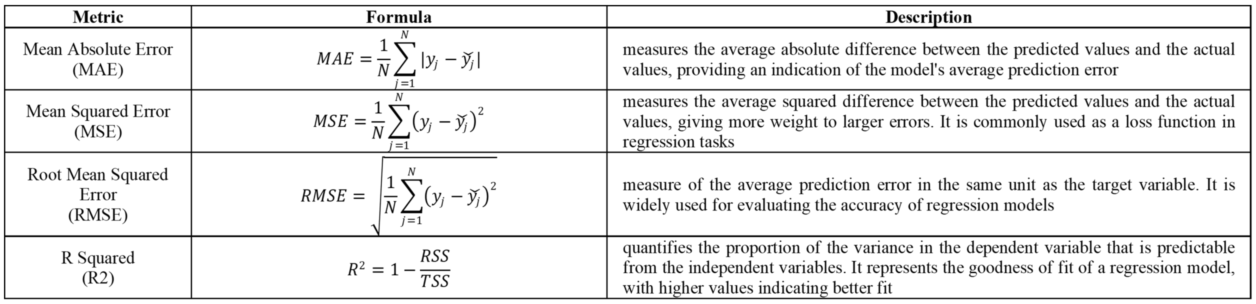
Now, a more in-depth analysis of the potential metrics needed to monitor a distributed ML system, focusing on the ML workflow, is conducted. The following list of monitoring metrics has been made by reviewing various articles [9–11, 17–19, 43] and sources [44–48] discussing this topic, and selecting the most commonly mentioned ones. Although it is not an exhaustive list, the below-mentioned metrics should provide enough information to evaluate an ML system's health and performance.

5.1.1 Model monitoring

When developing, training or using any model in inference, its monitoring is needed to evaluate how well-built or suited the model is. In this context, model performance metrics and concept drift detection are commonly referred to as ways to monitor a model.

MODEL PERFORMANCE METRICS

Performance metrics provide information about how well-fitted and how good the prediction is in a model in the training phase, where labelled data or ground truth is usually available. These metrics are different depending on the nature of the problem, whether a regression or a classification model is used [49]. Although there are plenty of performance metrics for each case, a reduced number of these form the most popular and used ones and, usually, provide enough information for evaluating any model. Some examples of these metrics are shown below [49–51], in Figures 5.1 and 5.2:

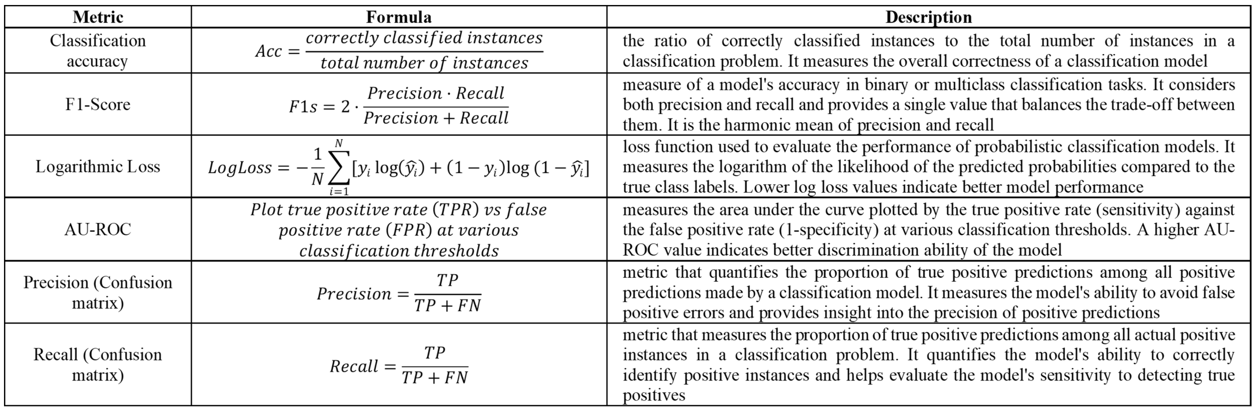


Metric	Formula	Description
Mean Absolute Error (MAE)	$MAE = \frac{1}{N}\sum_{j=1}^{N} \lvert y_j - \hat{y}_j \rvert$	measures the average absolute difference between the predicted values and the actual values, providing an indication of the model's average prediction error
Mean Squared Error (MSE)	$MSE = \frac{1}{N}\sum_{j=1}^{N} (y_j - \hat{y}_j)^2$	measures the average squared difference between the predicted values and the actual values, giving more weight to larger errors. It is commonly used as a loss function in regression tasks
Root Mean Squared Error (RMSE)	$RMSE = \sqrt{\frac{1}{N}\sum_{j=1}^{N} (y_j - \hat{y}_j)^2}$	measure of the average prediction error in the same unit as the target variable. It is widely used for evaluating the accuracy of regression models
R Squared (R2)	$R^2 = 1 - \frac{RSS}{TSS}$	quantifies the proportion of the variance in the dependent variable that is predictable from the independent variables. It represents the goodness of fit of a regression model, with higher values indicating better fit

Figure 5.1: Performance metrics for regression models

As said, the previous tables are non-exhaustive lists of performance metrics. A more extensive review of the existing metrics can be found in articles by Rácz et al. [52] and by Botchkarev [53].

Regarding the evaluation of model performance in production, an important aspect to mention is the dependence these performance metrics have on label availability. In a production environment, where new not-labelled streaming data comes into a model for it to make predictions, it is sometimes not possible to calculate the previously mentioned performance metrics to evaluate the model, primarily due to significant time delays after



Metric	Formula	Description
Classification accuracy	$Acc = \frac{correctly\ classified\ instances}{total\ number\ of\ instances}$	the ratio of correctly classified instances to the total number of instances in a classification problem. It measures the overall correctness of a classification model
F1-Score	$F1s = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$	measure of a model's accuracy in binary or multiclass classification tasks. It considers both precision and recall and provides a single value that balances the trade-off between them. It is the harmonic mean of precision and recall
Logarithmic Loss	$LogLoss = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$	loss function used to evaluate the performance of probabilistic classification models. It measures the logarithm of the likelihood of the predicted probabilities compared to the true class labels. Lower log loss values indicate better model performance
AU-ROC	$Plot\ true\ positive\ rate\ (TPR)\ vs\ false\ positive\ rate\ (FPR)\ at\ various\ classification\ thresholds$	measures the area under the curve plotted by the true positive rate (sensitivity) against the false positive rate (1-specificity) at various classification thresholds. A higher AU-ROC value indicates better discrimination ability of the model
Precision (Confusion matrix)	$Precision = \frac{TP}{TP + FN}$	metric that quantifies the proportion of true positive predictions among all positive predictions made by a classification model. It measures the model's ability to avoid false positive errors and provides insight into the precision of positive predictions
Recall (Confusion matrix)	$Recall = \frac{TP}{TP + FN}$	metric that measures the proportion of true positive predictions among all actual positive instances in a classification problem. It quantifies the model's ability to correctly identify positive instances and helps evaluate the model's sensitivity to detecting true positives

Figure 5.2: Performance metrics for classification models

a prediction is made or high cost for the labels to be produced, resulting in delayed or no ground truth. In other words, it is sometimes not possible to know how a model should have performed in production right after it has been performed (no feedback) [54], thus not being able to calculate such metrics.

Despite these challenges, it is crucial to ensure the deployed model is performing adequately. Label-independent metrics are frequently employed as proxy metrics for model performance when labels are hard to get. The best metrics to use depend on the type of data being analysed, such as photos, text, or tabular data. Even within tabular data, different metrics will be used depending on whether the features are numerical or categorical [18]. These metrics will also depend on the specific use case, as they will be calculated using the available data with the objective of correlating them with the unavailable ground truth to have a basic understanding of how the model is performing [54].

Aside from proxy metrics, there are other options to evaluate if there is no degradation in a model's performance, such as measuring statistical bias in predictions, i.e. the average of predictions in a particular slice of data [55], or comparing the distribution of predictions over time using statistical metrics such as Hellinger Distance, Kullback-Leibler Divergence, and Population Stability Index (PSI) [56].

CONCEPT DRIFT

Concept drift, sometimes referred to as concept shift or model drift, is a phenomenon that appears when the statistical characteristics or relationships between input features and output variables of the model alter with time in a machine learning problem [57]. In other words, the distribution or concept from which the model is trying to learn and infer can change or evolve, causing the performance of the model to deteriorate over time (model staleness), as well as the prediction accuracy.

Maintaining the quality and dependability of machine learning models requires the capacity to recognise and respond to concept drift. It is crucial to regularly evaluate and track the model's performance on new data to spot instances of concept drift and evaluate how well drift detection and adaption strategies are working.

In terms of speed, concept drift can be classified as [58]:

- **Sudden drift:** the concept change happens abruptly, causing the model performance to decline suddenly.
- **Gradual drift:** the new concept appears gradually and overlaps with the previous concept for some time, until the new concept becomes stable.
- **Recurrent drift:** when a recurring change happens in the concept ocasionally.

The previous concept drift types can be depicted in Figure 5.3.

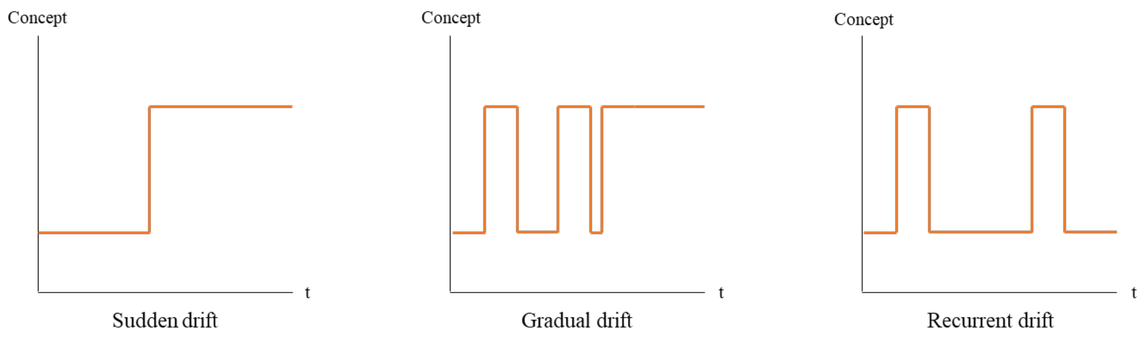


Figure 5.3: Drift types in terms of speed

Once again, it is important to consider the fact that the streaming data coming into the model will most probably be unlabelled data. Due to this fact, unsupervised concept drift detection techniques are required, as traditional supervised techniques used with labelled data are not suitable for this scenario.

There is no unique technique or algorithm to detect concept drift in streaming data problems. Plenty of algorithms have been developed and their suitability and performance depend on the data's characteristics and the problem requirements. There are several methods, that could be classified, according to how they look for concept drift, in similarity and dissimilarity-based methods, sequential analysis-based methods, window-based methods, statistical-based methods, significance analysis-based methods, data distribution-based methods, decision boundary-based methods, model-dependent based methods [59]. However, although these techniques have different steps and procedures, most of them base concept drift detection on the detection of changes in the data distribution.

5.1.2 Data monitoring

Outlier detection and data drift detection are the two most mentioned aspects regarding data monitoring.

OUTLIER DETECTION

An outlier can be defined as "*an observation which deviates so much from other observations as to arouse suspicions that it was generated by a different mechanism*" [60]. Measurement errors, exceptional but true values, misreporting or sampling errors are just a few of the possible causes of outliers generation. In many applications, spotting and comprehending outliers is crucial because they can offer insightful information, show potential issues, or indicate unexpected phenomena. In some cases, one would want to detect outliers to remove them so that the following data analysis does not get affected; whereas in other cases, one would want to detect them as they might provide useful information (e.g. by detecting an outlier in a breast cancer dataset, one could discover that someone potentially has a malign tumour) [61].

An important concept to note is the difference between local outliers and global outliers. For local outliers, only a small subset of the data is considered and the probability of a data point being an outlier as compared to its neighborhood is addressed. While, on the other hand, for detecting a global outlier, all data is considered and a point is considered an outlier if it is far away from all other points. This idea can be depicted in Figure 5.4, taken from an article by Alghushairy et. al. [62].

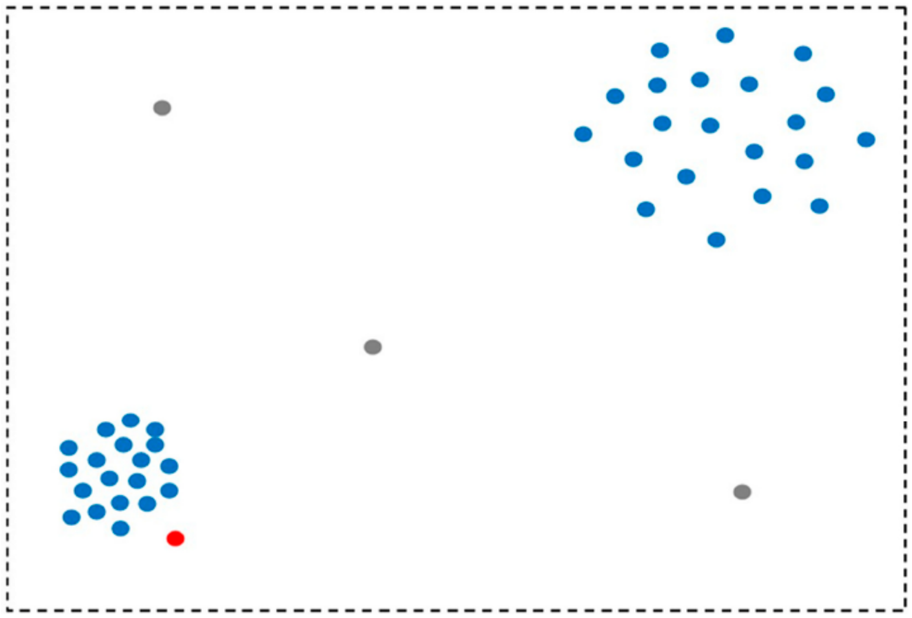


Figure 5.4: Outlier types. Grey points are global outliers and the red point is a local outlier

There are several techniques and algorithms, normally applied in the preprocessing phase, that can help to detect outliers in a dataset. Some examples of these are the use of Z-Score, Local Outlier Factor (LOF), geometric models such as Angle-Based Techniques (ABOD) and Depth-Based Techniques(Convex Hull), Isolation Forest, Autoencoder, and Outlier Detection using In-degree Number (ODIN) [63]. Nevertheless, in line with the scope of this project, it is necessary to review the existing techniques and procedures to detect outliers in data streaming processes, where a continuous and real-time flow of data from various sources is generated and fed to the system and needs to be analysed in real-time or near real-time with the objective of detecting outliers.

The existing outlier detection techniques can be divided, according to their dependence on labels, into supervised methods, semi-supervised methods, and unsupervised methods [64]. Due to this being a streaming data problem, the aim is to analyse the data as it is fed into the system. Thus, it is assumed that the labels for the incoming data are not immediately available, as getting these labels is a time-consuming and costly task. Accordingly, unsupervised methods are considered in this section.

The main existing unsupervised methods for outlier detection in data streams can be categorized into statistical, distance-based, density-based, and cluster-based methods, ensemble-based, graph-based, and learning-based. [65].

Detecting outliers in a data stream brings new challenges. In the first place, due to a data stream being a continuous sequence of data, it is not possible to store the whole stream to later apply an outlier detection method to the dataset. Secondly, due to the necessity of many streaming problems to detect outliers on the fly, the used methods impose an efficiency requirement [64]. To address these problems, windowing is commonly used, where a segment of the data stream is used to build incremental models to detect outliers in incoming and evolving data. There are four types of windowing techniques [66], according to the specific characteristics of the data stream and the requirements of the outlier detection algorithm: landmark window, sliding window, damped window, and adaptive window.

Although all these methods could theoretically support data stream outlier detection, some perform better than others in terms of memory usage, execution time and detection accuracy. There are plenty of different algorithms that have been developed for handling outlier detection in data streams. Which algorithm to use depends on the use case, data format and structure, and specific needs of the problem [62]. Thus, the choice of what algorithm to use is up to the user according to its needs.

DATA DRIFT / DATA DISTRIBUTION CHANGES

Data drift happens when the statistical characteristics or distribution of data vary over time. It happens when the underlying processes that produce the data change or when conditions or characteristics of the data collection environment change, resulting in different data distribution between the training and deployment data. This can influence the effectiveness and dependability of models and algorithms [67].

Detecting data drift in machine learning algorithms is essential for ensuring:

- **Model performance:** data drift has a big impact on how well machine learning models function. When used with new data that has drifted, models that were trained on prior data may become less accurate or even stop producing accurate predictions.
- **Fairness and bias:** data drift can create biases, producing discriminatory or unfair predictions. Results may be biased if the drift has an impact on how certain traits are distributed.

By detecting data drift, one may be able to determine when model performance is declining and take the necessary steps to restore it, such as retraining the model or changing its parameters.

Regarding unsupervised data drift detection techniques, they are similar to the ones used to detect concept drift, as both of them try to detect changes in the probability distribution of data. Nevertheless, while concept drift detection focuses on the relationship between these changes and a decline in the model performance or accuracy, data drift detection is limited to detecting these distribution changes. Thus, detecting data drift serves as an alert to look for concept drift in advance and carry out the necessary actions to prevent this from affecting negatively the model's performance [46].

5.2 Monitoring system architecture proposal (RQ2)

Once defined what the different metrics needed to monitor a generic distributed ML system are, it is required to define a software architecture and its technical details on how to capture and store this information to be later able to do the following tasks:

- **Visualization:** to visualize the data in real-time or near real-time through dashboards containing the needed graphs and panels to get to know the status of the pipeline and its components in a visual and simple way.
- **Alerting & Notifying:** to set conditions and thresholds to given values or metrics so that, whenever these trigger conditions are met or these thresholds surpassed, an alert is risen and a notification to a certain user/channel is sent.
- **Recommender system:** to design a recommender system that, by having a register of the behaviour, logs and outputs of the different modules of the pipeline according to its inputs and configs, gives recommendations about how a certain module should be configured in each situation, according to the given inputs (i.e. *when the input of module M is X, use config A; when the input is Y, use config B*)
- **Store data for other purposes:** to develop any other further system.

Furthermore, this architecture definition should serve as a base to later carry out an implementation of the monitoring system with specific tools and procedures.

5.2.1 Generic module-based ML pipeline example definition

To do so, firstly, a generic ML pipeline is defined as a base to describe the monitoring software architecture. This pipeline will be based on modules, a higher abstraction of services.

Following the concept explained in Chapter 1, a module can be described as a self-contained unit within a containerized environment. A generic module would have the following structure (Figure 5.5):

Where:

- **Input(s):** information or data provided to the module from an external source, which the module uses to perform its operations or calculations.
- **Output(s):** results or data produced by the module after processing the inputs. These results are typically sent to an external destination or used by other system parts.

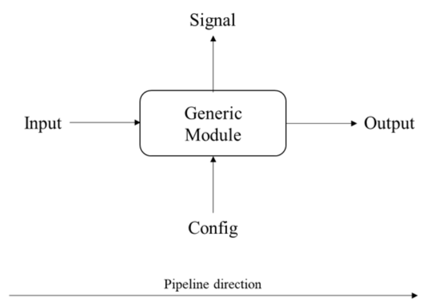


Figure 5.5: Generic module defintion

- **Configuration(s):** are the settings or parameters that can be adjusted to customize the behaviour or functionality of the module. Configurations allow users to define specific options to adapt the module to their specific requirements. For example, training datasets, hyperparameters, models or algorithms can be configs, among others. These configs will be fed to a module to perform its assigned tasks in a certain way.

- **Signal(s):** intermediate or internal data flows within the module that carries information between different components or stages of the module's processing. Signals often represent the state or progress of the module's operations (logs). They can also be seen as non-functional outputs, as it is not necessary to check their compatibility between modules when connecting them to one another.

One important point to consider is that it is not mandatory for a module to have all four communications. For example, a training dataset, which would exist in the form of a module, could have no inputs or configs, but just an output. This is because the training dataset module's output (the dataset itself) would be fed to another module, a model training module, for instance, as a config. This idea is depicted in Figure 5.6.

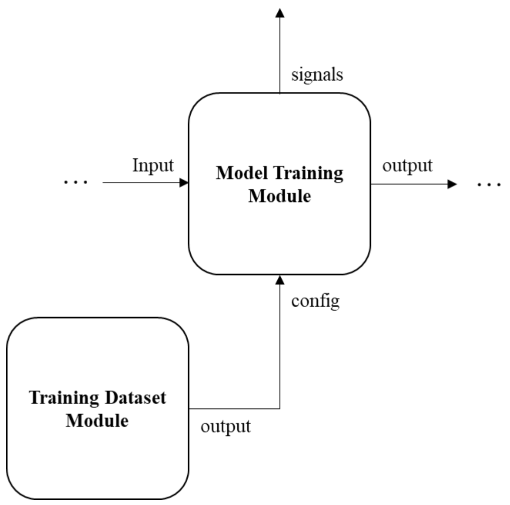


Figure 5.6: Module with no inputs or configs

Another important aspect to consider is the fact that, due to this being a data stream processing problem, the pipeline is intended to work with streaming data. The communication between modules and data transfer will be done through event logs, where modules read (module's input and configs) and write (module's output and signal) said data in them as records.

The combination of these modules would give rise to a simple generic pipeline such as the following (Figure 5.7):

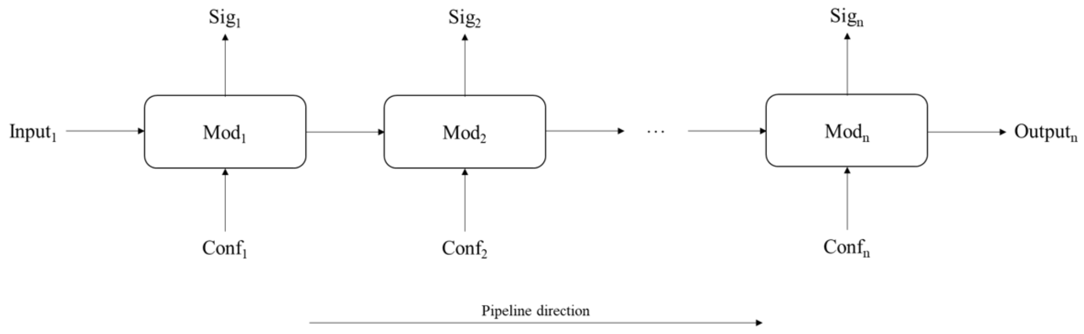


Figure 5.7: Pipeline structure definition

The aim of proposing a software architecture for a monitoring system is to define what information from these modules should be captured, how to actually capture it, and how and where to store this information to later be accessible for other purposes.

It is worth noting that, depending on the final use of the ML model, the real ML pipeline would have different structures and components. This is caused by the existence of different data types and formats, as well as different ML models and algorithms. Nevertheless, many ML pipelines can include similar general steps, resulting in similar workflows. These basic tasks are data ingestion, data cleaning, data preprocessing, model training, model evaluation, model deployment, and performance monitoring.

Taking this into consideration and following the module-based topology described above, a generic ML pipeline example can be defined using the following modules:

- **Data collection module:** for gathering the required data coming from various sources, like databases, APIs or files.
- **Data cleaning module:** for cleaning and transforming the incoming data to the desired and required format for future use. It involves tasks such as handling missing values, outliers or scaling.
- **Data validation module:** for ensuring the quality and correctness of the processed data.
- **Model building module:** for choosing and trying different models and algorithms based on the problem requirements and the data format.

- **Model training module:** for the training of the chosen model using training data and adjusting hyperparameters to optimize its performance.
- **Model evaluation module:** for evaluating the trained model using the test data using performance metrics.
- **Model serving module:** for the deployment of the trained model in a production environment to make predictions on new unseen data.
- **Monitoring modules:** for monitoring different modules of the pipeline to evaluate the performance of the modules and define thresholds for alerting and notifying.

A diagram representing this generic pipeline and its modules and connections is presented in Figure 5.8.

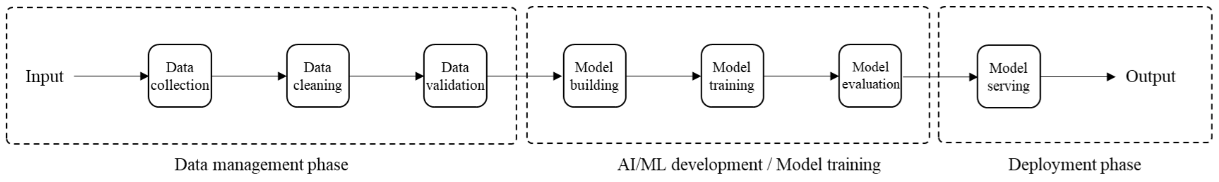


Figure 5.8: Generic pipeline defintion

Having defined the generic pipeline, it can be discussed in more detail how the monitoring would be done and how the monitoring system is structured.

5.2.2 Monitoring principles

Before start defining the monitoring architecture, some principles and possible approaches need to be addressed.

MONITORING PURPOSES

Two purposes can be defined when monitoring the ML pipeline:

- **Pipeline performance:** firstly, the aim is to get to know the status, behaviour and performance of the pipeline and its components to ensure it works as expected and to detect possible issues, visualizing this data through dashboards and setting an alerting and notifying system.
- **Data record of the pipeline:** secondly, the aim is to have a record of how each module performs and behaves according to its inputs and configs, tracking down which module's version works better and associating it with its inputs and outputs. In the same way, the performance of the pipeline with respect to the modules that make it up

can be evaluated. With this information, a data scientist or a recommender system, for instance, could either analyse the performance of the pipeline using this information and decide the optimal configuration for each module of the pipeline and could also compare multiple pipelines to know which one performs better than the rest.

Thus, all inputs, outputs, signals and configs should be taken into consideration when defining the monitoring architecture.

MODULARITY APPROACHES

Firstly, regarding monitoring modularity, there are several possible approaches:

- **Monitoring as part of the platform:** considering monitoring as a holistic solution, defining a single standardized cluster-wide monitoring interface on top of which the user can implement their own monitoring logic/controllers, that collects all information offered by modules of the pipelines and processes and responds to them with the user-defined logic (Figure 5.9).

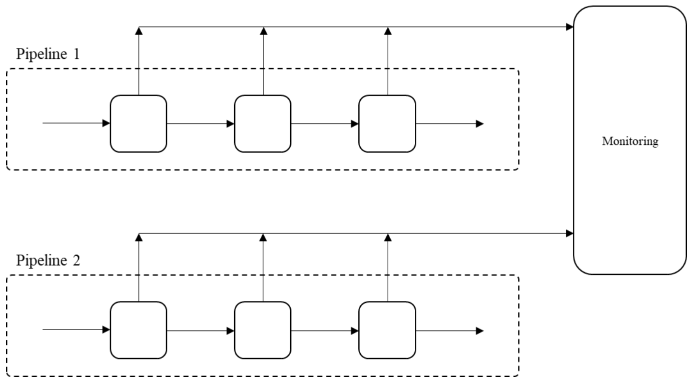


Figure 5.9: Cluster-wide monitoring approach

- **Monitoring-as-a-Module (MaaM):** in line with the XaaM vision explained in Section 1.1, this approach allows users to define custom monitoring modules, attachable to the running pipelines' modules, with various module implementations that allow selecting which modules to monitor, allowing to create a more flexible monitoring solution according to the requirements of the user (Figure 5.10).

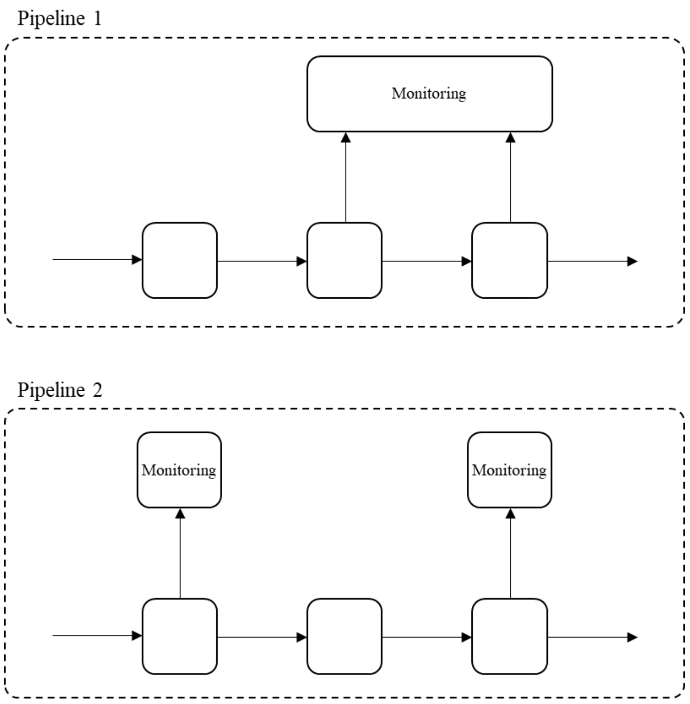


Figure 5.10: MaaM approach

- **Mixed approach:** both previously explained approaches could coexist in a mixed solution. It may be beneficial to have the flexibility that MaaM offers when deciding which modules to monitor and which data to collect from them but centralise the gathering of all this data and its management and storage for later processing (Figure 5.11).

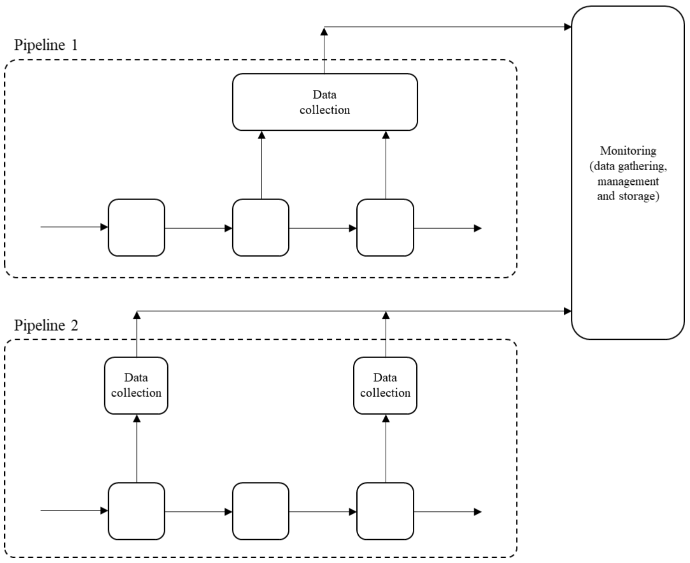


Figure 5.11: Mixed modularity approach

Taking into consideration the possible approaches for the design of the monitoring system, it is considered that a mixed approach is the best option. MaaM offers more flexibility when it comes to deciding which modules the user wants to monitor, considering that maybe not all modules require monitoring. However, it may not be practical to have an independent data management and storage system, as well as a potential visualizing and alerting system, for each of the monitored modules. The cluster-wide approach solves this problem, gathering all information coming from the selected modules to monitor in a single place and centralizing the management and storage of all data, as well as the implementation of a visualization and alerting system.

Now, the focus will be set on how the monitoring of the pipeline is carried out. This means defining what data should be captured, where and how it is captured, and how it is managed to be able to later access it.

Considering the final purpose of the monitoring system is to be able to evaluate the performance of each pipeline to decide which one performs better, it is necessary to define how this performance will be evaluated.

Due to the fact that the pipeline is not a module itself, its performance evaluation is subject to the performance of the modules that make it up. Thus, in the first place, one may want to monitor a specific module of a given pipeline.

5.2.3 Module Monitoring

In the first place, when creating and developing ML pipelines, an iterative process may be needed to come up with a well-performing solution. Thus, several changes in code, models, algorithms, training datasets, hyperparameters, seed values, etc. may be needed to accomplish this. These changes entail the appearance of module versions, where each new version behaves in a certain and different way according to the changes made. To be able to properly track the performance of modules and pipelines and evaluate their evolution, it is necessary to have a record of all created modules and their versions.

Taking this into consideration, one may want to have a register of the different created modules and their versions to track their performance in relation to the modules' versions and configs used in each pipeline.

MODULES AND CONFIGS VERSION TRACKING

Due to the modularity approach for building the pipelines, each module may vary in structure, which will be different, for instance, for a model training module (model as module's core) and a training dataset module (dataset as module's core). Due to the existence of different formats (i.e. code, values for hyperparameters, files for models or algorithms, etc.), it is considered that a NoSQL database is the best option for storing all this information regarding the modules' versions. A NoSQL database offers the flexibility, scalability and performance needed for storing unstructured and heterogeneous data.

Regarding how to actually store this information, a new document should be created in the NoSQL database for each module and version when created. One approach to achieve this and have all the needed information is to store this information as a document in a JSON or similar format, for example, as follows:

```json
{
  "module": "module_x",
  "version": "version_y"
}
```

Furthermore, if required, the actual content of the modules, such as models or algorithms code, datasets, parameters, values, etc. could be also stored in the same NoSQL database. Nevertheless, to be able to track and link them with the information from the modules' documents, a reference id should be defined. One way of doing this is by using a one-to-many reference id. A unique id is given to the module document and this id is written in each file regarding this module stored in the database. Then, the document for the module would look like the following:

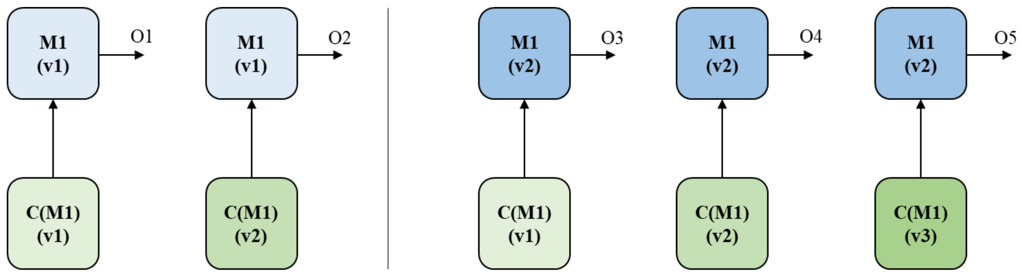
```json
{
  "id": "unique_id",
  "module": "module_x",
  "version": "version_y"
}
```

Where the *unique_id* is also present in the corresponding files and, when querying the document for the module version information, it is possible to make a second query and get the associated files.

This approach may be enough for storing versioning information about modules responsible for carrying out tasks directly involved in the ML pipeline, like the ones shown in Figure 5.8. Nevertheless, considering what has been said about certain modules only having an output, like config modules (Figure 5.6), it may be useful to link the information of such modules (and their versions) with the module they are feeding the config to. In other words, link the information about config modules with the modules that receive these configs. By doing this, it would be possible to evaluate how a certain module version behaves with different config versions. For example, how the output of a given version of a model training module behaves in relation to different training datasets (different config module versions).

A more visual representation of this concept can be depicted in Figure 5.12. In it, there can be found two versions of a certain module M1 (i.e. M1(v1) and M1(v2)). For version v1 of M1, two versions of a config module C are fed to it, C(M1)(v1) and C(M1)(v2); on the other hand, for version 2 of M1, three versions of the same config module are fed to it, C(M1)(v1), C(M1)(v2), and C(M1)(v3). In each of the five cases, the output (O1 to O5) may be different, due to the different combinations of M1 versions and C versions.

To track these combinations, it is necessary to link the version of a config module with



Module	Version	Config	Output
M1	V1	C(v1)	O1
M1	V1	C(v2)	O2
M1	V2	C(v1)	O3
M1	V2	C(v2)	O4
M1	V2	C(v3)	O5

Figure 5.12: Example: combinations of module versions and config versions

the version of a module that is consuming this config. In the NoSQL database, this can be achieved in a similar way as the one described for linking a module's name and version with its actual content. The proposal is to write the unique id, defined in the main module document, in the document used to store the config module's information, such as:

```
{
  "id": "config_unique_id",
  "module": "config_module_x",
  "version": "config_version_y",
  "main_module_id": "unique_id"
}
```

It can be seen in the previous document example that, on the one hand, the *"main_module_id": "unique_id"* allows linking the config module information with the main module that it is feeding; and, on the other hand, like with the main module, the *"id": "config_unique_id"*, enables the link between the config module document and the actual content of this module (e.g. training dataset), also stored in the database. This reference structure can be depicted in Figure 5.13, where there is a main module document and its content (ML model), as well as two different config documents (Config 1 Module document and Config 2 Module document), used to feed this module, and the actual content of these two config modules (Training dataset 1 and Training dataset 2, respectively).

Once defined how created modules and their versions are registered and stored, and how the relationships between modules and config modules are tracked, it is necessary to define how to evaluate the performance of these modules to be able to determine which module's version performs better. To do so, let's define now how a module operates.

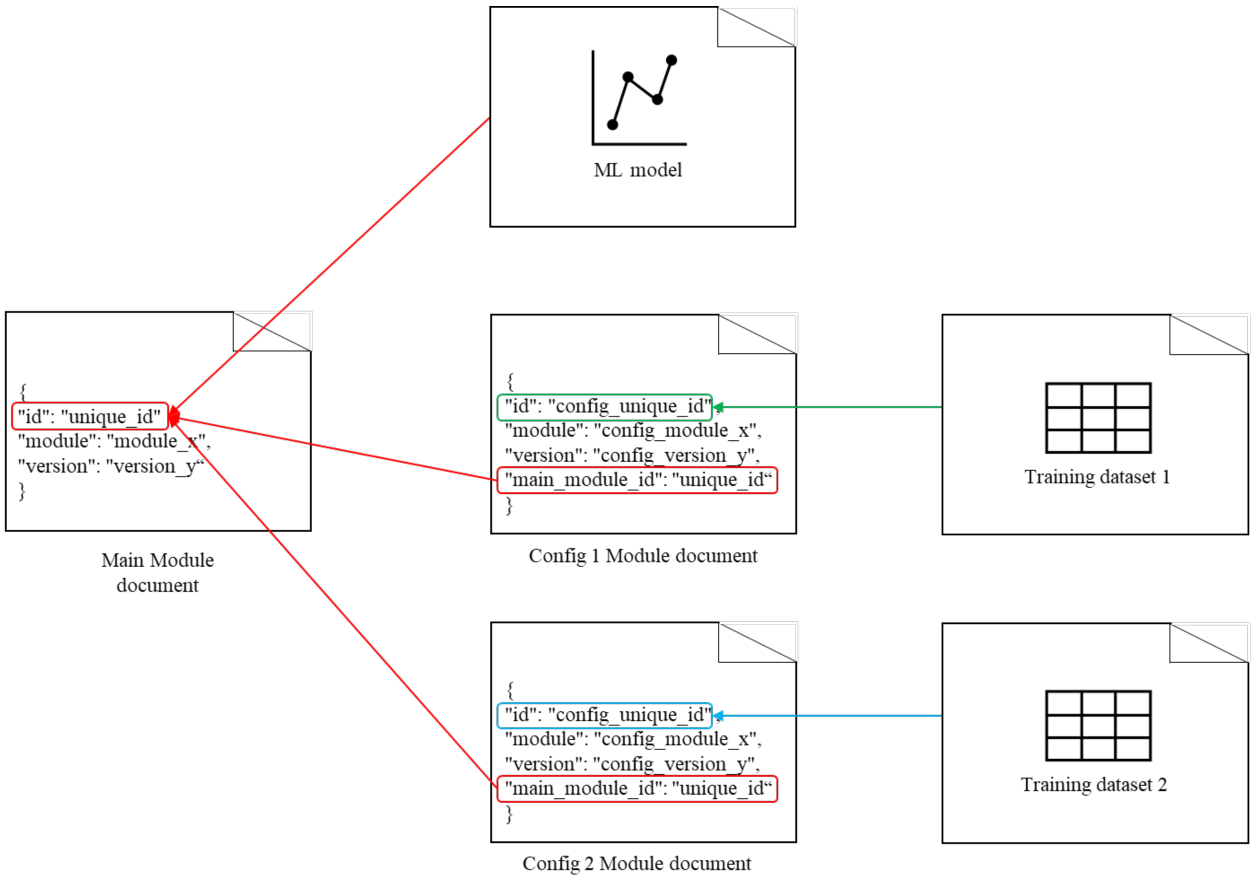


Figure 5.13: NoSQL database reference linking structure example

MODULE'S INPUTS, OUTPUTS AND SIGNALS TRACKING

Due to this being a streaming data problem, the inputs going into a module and the outputs and signals coming out of it get registered as records in event logs containing, for instance, at least a unique key, the record timestamp, its payload, and other metadata.

Depending on the task that is carried out by a certain module, it will not be convenient to calculate certain outputs in the module itself, aside from the primary outputs, needed for evaluating the module's performance, i.e. an output whose generation may impact the performance of the module itself (e.g. certain model performance metric calculation after a prediction that takes a considerable time to be calculated or detecting outliers based on the incoming data).

In this case, one may want to delegate this task to another module, working in parallel as the rest of the pipeline keeps running seamlessly, and whose execution time is not critical and does not affect the rest of the pipeline. This module can be called a 'Module Evaluator' module and would be attached to the desired module and would consume the records from the event logs coming in and out of it. This is with the aim to collect the primary outputs, as well as the inputs of the module, to carry out the process to generate the needed output/metric to evaluate the module's performance that has not been possible to be generated in the module in the first place. This idea is shown in Figure 5.14.

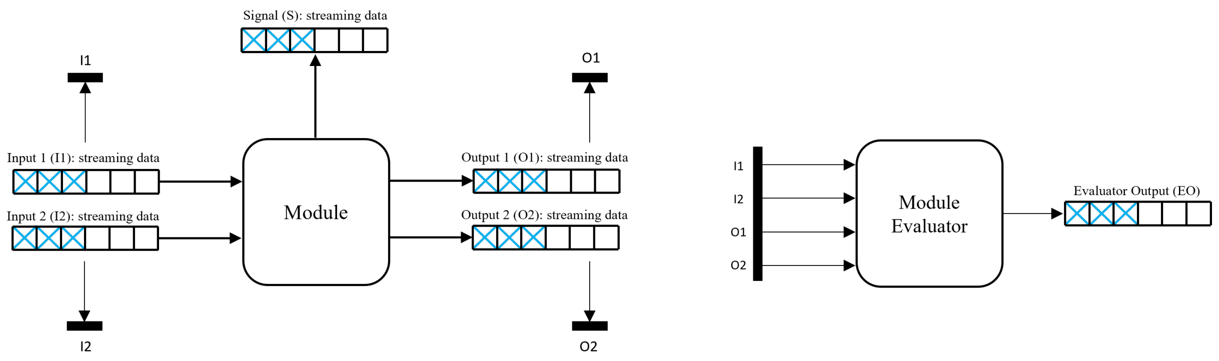

Signal (S): streaming data
I1
O1
Input 1 (I1): streaming data
Input 2 (I2): streaming data
Module
Output 1 (O1): streaming data
Output 2 (O2): streaming data
I2
O2
I1
I2
O1
O2
Module Evaluator
Evaluator Output (EO)

Figure 5.14: Module Evaluator module

It can be seen that the output of the Module Evaluator module (EO) is again a streaming output and is published to a new event log, containing the necessary information to evaluate the performance of the module. It is also worth noting that the development of these modules would be subject to each use case and would depend on various factors, such as the module to which it is attached, the calculation needed, and the data format, among other factors.

Regarding the storage of this information, a good choice would be to store it in the same NoSQL database as the modules, versions and configs, for easier linking and retrieval of all needed information. To be able to link the data of the event logs for the inputs, outputs, signals and evaluator output of the module with the module, module version and config version information stored in the NoSQL database, it would be necessary to write, in the metadata of said records, the necessary additional fields to enable this correlation between both sources of data.

In the first place, the proposal is to create separate documents for inputs, outputs, signals and evaluator module output. Log information would be stored in each of these documents as an array containing all the records. It will be then necessary to write fields for the config module id in these records, as well as the type of information they carry (inputs, output, etc.). By linking the log information to the config module document, and considering the config module is already linked to the main module, all information gets stored in an organized way.

For example, a single record for the input data of a module would look like the following:

```
{
  "id": "unique_log_id",
  "type": "input",
  "logdata": "...",
  "config_module_id": "config_unique_id"
}
```

By doing this, one could link the data from the event logs with the specific module and version (using certain configs) that consumed and produced them. Joining this information would allow one to evaluate the performance of the given module through the *"congig_module_id": "config_unique_id"*, present in both the config module document and the log data documents. The storage structure described so far can be depicted in Figure 5.15.

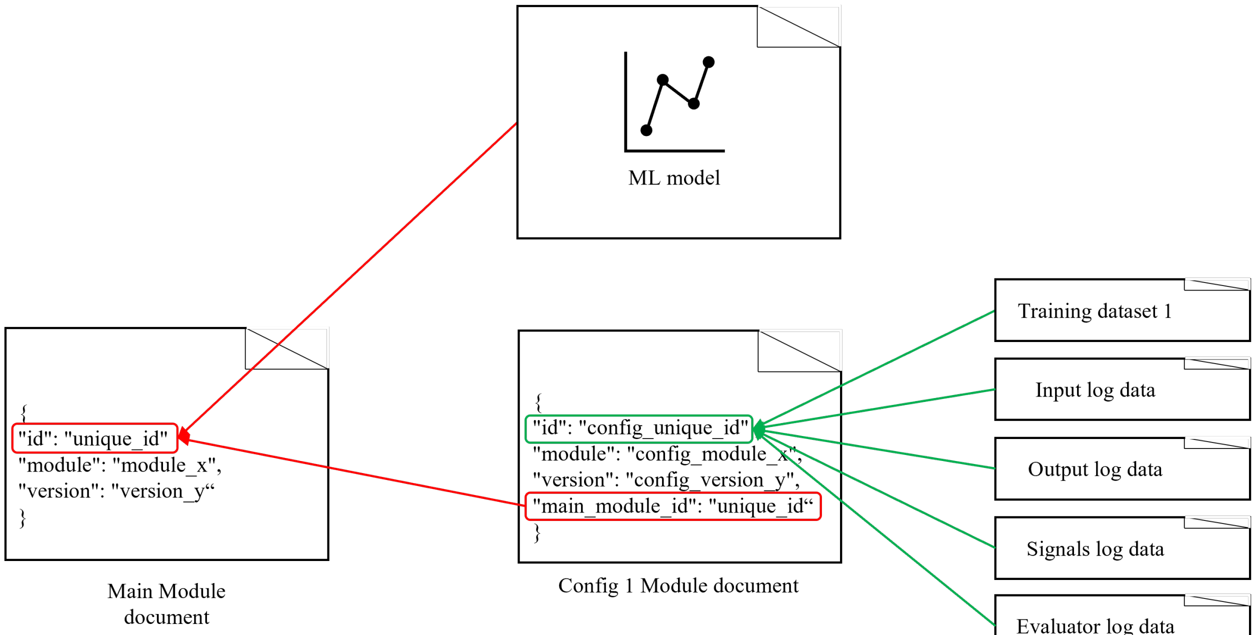


Figure 5.15: Log data linkage to config module information in NoSQL database

Nonetheless, aside from monitoring a single module, the aim is to monitor the performance of the running pipelines and to be able to compare them. Thus, the proposed architecture must be extrapolated so that all the info regarding the running pipelines is available.

5.2.4 Pipeline Monitoring

Now, let's imagine there are multiple pipelines running simultaneously. Each of these pipelines is made up of several modules. It is worth noting that the same module could be part of more than one pipeline and, in the same way, the same module but with a different version could be in different pipelines. Moreover, different configs for the same module and version could also coexist in the system.

Let's illustrate this context with a simple example of four running pipelines (A, B, C and D), each of them made of a single module, where the same module (M1) is present in all pipelines, as well as different versions of this module (v1 for pipelines A, B and C; and v2 for pipeline D) and with different configs (C1 for pipelines A, B and D; and C2 for pipeline C) (Figure 5.16).

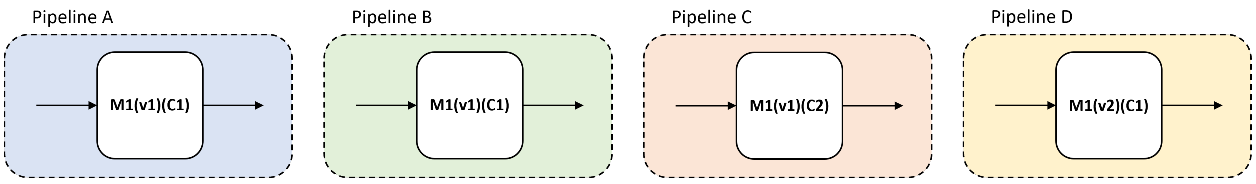


Figure 5.16: Multiple running pipelines example

Considering this example, it can be seen that it is necessary to organise all the log available data (input, output, signals and module evaluator output) in an organised and traceable way, regarding the modules, versions, configs and pipelines information.

This could be achieved by adding a new field to the metadata of the records of the event logs for linking the log data of a certain module with the pipeline it is operating in, like shown below:

```
{
  "id": "unique_log_id",
  "type": "input",
  "logdata": "...",
  "config_module_id": "config_unique_id",
  "pipeline": "pipeline_x"
}
```

With this structure, one would be able to retrieve the desired records from the event logs of the desired module (using a certain config) of a given pipeline

For example, to monitor module M1 of Pipeline A, one would retrieve the records from the event logs (inputs, outputs, signals, or, if there is one, the evaluator module) that, in their metadata, have the *"pipeline"* field equal to *"pipeline_a"* and the *"config_module_id": "config_unique_id"* field equal to the *"id": "config_unique_id"* of the config document, that at the same time, have the *"main_module_id": "unique_id"* equal to the *"id": "unique_id"* of the main module document. By doing this, one would retrieve the logs for module M1 in version v1, using config C1, and running in pipeline A.

Following the same procedure, each module of each pipeline could be monitored independently, thus becoming a flexible solution allowing one to decide which modules and which information of such modules to monitor.

It is worth mentioning that the proposed structure for storing the data from the running pipelines in the NoSQL database is intended to minimize the storage capacity needed by avoiding storing multiple times the same information. For example, if a dedicated database had been assigned to each running pipeline and, in these pipelines, the same module with the same version, or the same config module had been used, this information would have been stored multiple times, one time for each running pipeline containing said module or config. With the proposed structure, a module and its content, as well as the configs used and its content, are stored only once. The way in which log information is stored is what allows for differentiating the information for each pipeline. To better understand this structure, how the data in a context like the one shown in Figure 5.16 would be stored in the NoSQL database can be depicted in Figure 5.17.

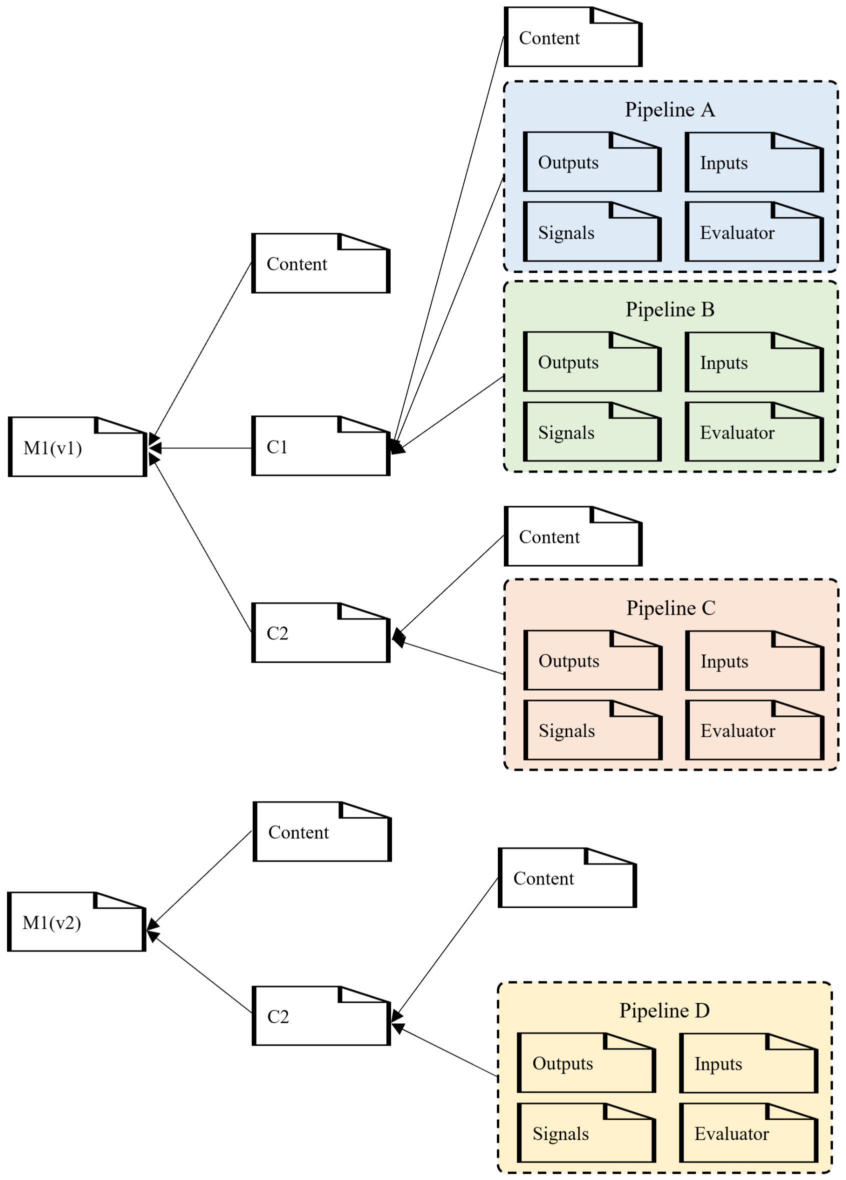


Figure 5.17: NoSQL database data storage structure

5.2.5 Data Management and Storage

Once defined how to capture all the data coming in and out of the modules, and how this information could be linked with the specific module (and version) and pipeline where it came from, it is necessary to define how all this data would be managed and stored.

On the one hand, the data stored in the NoSQL database regarding the versioning of modules and configs does not need further management, as it is already accessible when needed. On the other hand, the data in the event logs need to be further managed. It is worth noting that many streaming data applications have data retention times policies that determine for how long the data is retained or stored in the system before it is discarded. Some of these applications may let the user define these policies and, even more, some may allow one to retain data for a long period or even indefinitely. Nevertheless, storing the data in a more persistent way for later access may be more convenient.

The proposed solution is to use a data ingestion tool that ingests the event logs from the streaming data application, transforming them into a structured format that helps in standardizing the data for further processing and analysis and connects to a storage system that allows querying this data efficiently. It is necessary to choose a compatible data storage system to store the ingested event logs.

Having the streaming data correctly stored persistently, now it is possible to access the data needed to implement new systems, such as visualization, alerting and recommender systems.

The proposed architecture can be depicted in Figure 5.18. Although there are only two pipelines, the architecture would be the same with several running pipelines: the event logs ingestion module would ingest all the desired logs and store them in the NoSQL storage system. In the same way and as said before, the NoSQL database would have a register of the created modules and their versions and configs used.

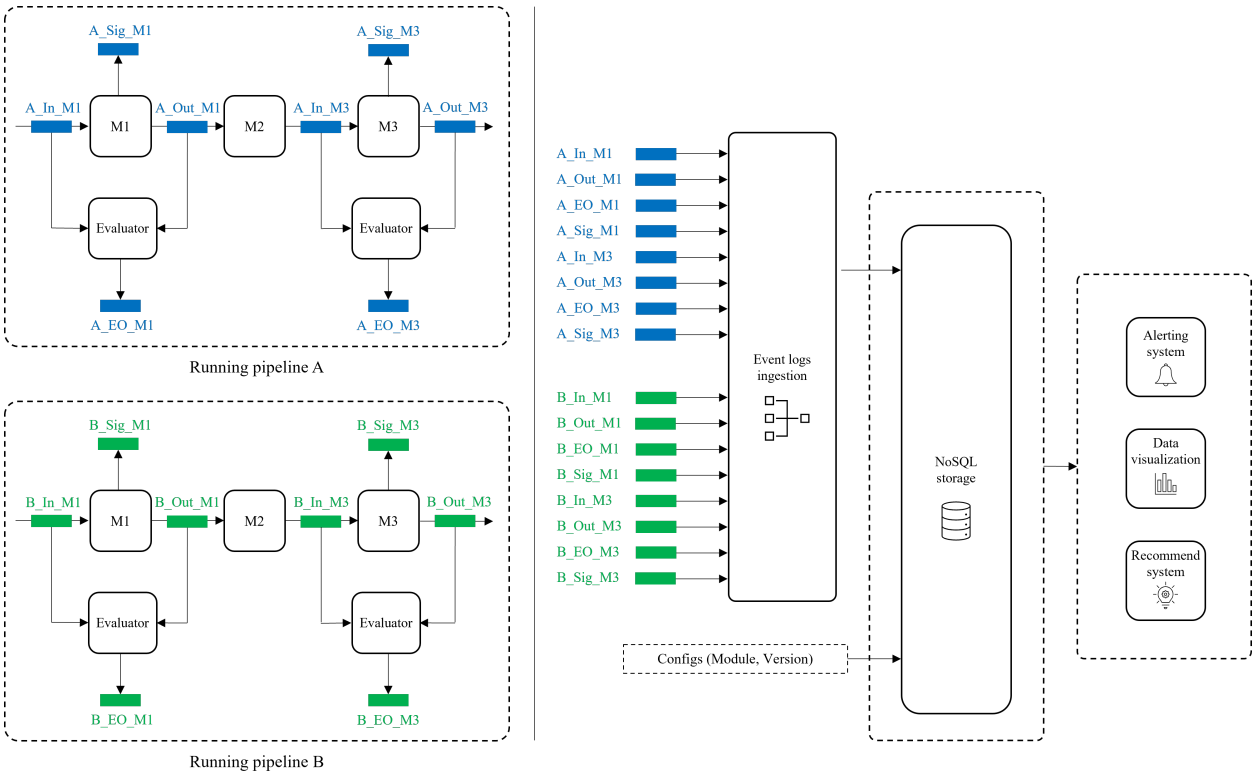


Figure 5.18: Monitoring system architecture

5.3 Available tools review (RQ3)

Before proposing an implementation with specific tools for the software architecture defined in RQ2, a search and review of the available and potentially suitable tools to carry out the different tasks are conducted. Thereafter, a study of the feasibility of implementing the proposed software architecture proposed in RQ2 is done and the most suitable toolset for building the monitoring system is provided in RQ4, alongside the architecture itself and how everything would work together. This is with the aim to define a software example to have as a base for further development of the monitoring system.

5.3.1 Tasks definition

Firstly, according to the software architecture proposed in RQ2, tools responsible for the following tasks must be chosen:

- **Message management system:** responsible for ingesting and processing all the messages responsible for the continuous communication between modules. It is responsible for the collection of all data from these messages in an organised way so that it can be later stored in a persistent storage system.
- **Logs storage system:** for indexing and storing all data from event logs, including input, output and signals information, in an optimized way for efficient data storage and retrieval.
- **NoSQL database:** system for storing in a flexible and scalable manner all information regarding configs of modules and versions. It will be stored both the information about the used configs and the config files/values/models themselves.

It is worth noting that, when choosing the set of tools, it must be ensured that the different tools are compatible with each other and have the necessary connection capabilities to be able to work together seamlessly.

On the other hand, it should be noted that an all-in-one-tool solution would be ideal, but this might not be feasible with the available tools and, thus, the final proposal might be a multiple-tool solution.

5.3.2 Tools basic requirements

For this project's purpose, certain fundamental criteria must be met by these tools to be eligible for designing the proposed software architecture for the monitoring system. The following prerequisites need to be satisfied:

- **Fully open-source:** they must be publicly available and completely free to use.
- **Kubernetes compatibility:** must be compatible with Kubernetes as it is the base of the containerized distributed system. There is no need for the tools to be considered Kubernetes-native, as long as they are compatible with it.

Only tools that already fulfil these basic requirements will be considered in the following subsections.

5.3.3 Tools review and comparison

Next, several tools are reviewed for the mentioned tasks for the monitoring system. This is a non-exhaustive review, each list being a non-exhaustive list of tools that could fulfil the needed requirement for each task. The reviewed tools have been selected by reviewing several articles and websites reviewing tools for these tasks. Moreover, the maturity, size and activity of the community and available documentation have been key aspects to consider when choosing which tools to review.

5.3.3.1 Message management system

Regarding event logs management, there are many available tools for specific tasks under this umbrella. Nevertheless, a more holistic and general solution is preferred, allowing the log lifecycle to be managed by a single tool. This involves providing the infrastructure needed to be able to publish and consume data from and to event logs and store this data in a way so that it can be retrieved in an organised way.

For this purpose, by reviewing several articles [68–70] and resources [71–73], Apache Kafka [74] and RabbitMQ [75] are the two most referenced tools, both with an active community and available documentation and information, thus these being the two candidates for message management.

APACHE KAFKA

Apache Kafka is an open-source distributed streaming platform that acts as a high-throughput, fault-tolerant messaging system. It serves as a reliable enabler for real-time data streaming and processing. With Kafka, different systems and applications can exchange data efficiently and in a scalable manner [74].

Kafka communications are based on the concept of topics, which can be seen as channels for data streams. Producers publish records (messages) to specific topics, while consumers subscribe to those topics to consume the records, allowing for communications between applications or parts of an application. It also uses a partitioning mechanism, where each topic is divided into multiple partitions, allowing for parallel processing and high throughput [76].

Moreover, it provides fault tolerance by replicating data across multiple brokers in a cluster, ensuring data durability and availability. These features make Kafka a powerful and widely adopted solution for building real-time data pipelines and streaming applications [74].

RABBITMQ

RabbitMQ is an open-source messaging system for reliable communication between different components of a distributed system. It serves as a message broker, enabling the exchange of data and messages between producers and consumers. With RabbitMQ, applications can easily communicate and share information in a decoupled and asynchronous way [75].

Message producers send messages to specific queues, while consumers retrieve and process those messages from the queues. This decoupling allows applications to operate independently, focusing on their specific tasks without direct knowledge of each other. RabbitMQ ensures message delivery by employing various messaging patterns like point-to-point and publish-subscribe.

Overall, it allows efficient and resilient communication within distributed applications and smooth data flow and integration between different components.

COMPARISON

The comparison shown in Table 5.1 between both tools has also been made by aggregating and selecting the desired information from the previously mentioned sources, as well as by looking into the available documentation for each tool. The features considered for the comparison and the reason why they have been considered are the following:

- **Message handling:** designates how messages are handled in the messaging system, once they have been consumed.
- **Throughput:** indicates the rate at which messages can be processed. Given an application with increasing workloads and amounts of data, this is a crucial point to consider.
- **Message ordering:** indicates the ability of the tool of maintaining the order in which messages were produced or received. For certain applications and post-processing purposes, ensuring this order may be necessary.
- **Scalability:** is the ability to handle increasing workloads by adding resources or distributing the load across multiple machines or nodes.
- **Most common use cases:** they help evaluate a tool's suitability for specific application requirements and scenarios.

- **Ecosystem & integration:** due to the use of multiple tools for building an application like the one in this project, it is important that a tool offers great connection and compatibility capabilities with other tools, intending to build a robust and functional ecosystem.
- **Github starts & forks:** these two numbers allow comparing the projects' popularity, community support, and active development, as they provide insights into the traction and community engagement.

Apache Kafka vs RabbitMQ		
Feature	**Apache Kafka**	**RabbitMQ**
Description	distributed streaming platform	message broker
Message handling	persistent (retention time / size) - replayability	deleted when consumed - no replayability
Throughput	very high	high
Message ordering	partition-order guarantee	singie-queue-order guarantee
Scalabillity	horizontally - good for throughput	vertically
Most common uses	real-time event streaming / log aggregation / streaming data pipeline	asynchronous common patterns
Ecosystem & integration	excellent	good
Github stars	25.2k	10.8k
Github forks	12.7k	3.9k

Table 5.1: Message management system tools comparison

Taking this comparison into consideration, Apache Kafka seems a more suitable choice for the current project, due to its focus on data streaming processes, its scalability and throughput, and excellent integration with other tools and systems.

5.3.3.2 NoSQL database

As said, a NoSQL database storage system is needed for storing the information regarding the configs of the newly created modules and versions, as well as the event logs information. With this information, one should be able to evaluate the performance of modules and pipelines according to their version.

Once again, the non-exhaustive list of candidates for this system has been elaborated from reviewing several data sources [77–79] and selecting the most mentioned ones. In this case, the candidates for the NoSQL database are MongoDB [80], Apache Cassandra [81] and CouchDB [82].

MONGODB

MongoDB is a popular open-source NoSQL database. It stores data in a flexible, document-based way in a BSON (Binary-JSON) format. This format allows for storing more data types than JSON format, and makes data manipulation more easy, becoming a more flexible solution and letting store more types of data.

It is a distributed database that ensures high availability, fault tolerance, and scalability. These features, together with its flexibility and advanced querying capabilities make it a powerful tool for developing applications with high volume and speed needs [80].

APACHE CASSANDRA

Apache Cassandra is another open-source NoSQL database. However, unlike MongoDB, it is a wide-column store or family database, which means that the formats of the columns can vary from row to row, even in the same table. This allows for storing all sorts of data, including semi-structured and unstructured data [81].

It offers great scalability, availability and fault-tolerance capabilities, making it a popular NoSQL database.

COUCHDB

Finally, Apache CouchDB is an open-source NoSQL database. It has similar characteristics to MongoDB, as it is a JSON-like document-based storage. One of the main differences is that it supports offline access and data synchronization [82].

COMPARISON

As in the previous cases, the comparison shown in Table 5.2 has been done by selecting information from the reviewed sources and tools' documentation.

In this case, all three options could be eligible. Nonetheless, MongoDB is chosen due to its flexible schema handling, powerful querying capabilities, and being a mature tool with a big and active community.

5.3.3.3 Selected tools compatibility

As has been said, the compatibility and integration of the selected tools is also an important aspect to consider, to be able to build a robust and functional solution. This compatibility has already been checked while carrying out the review, comparison and selection of tools done in this section. Nevertheless, the specific details on how these tools can be

MongoDB vs Apache Cassandra vs Apache CouchDB			
Feature	**MongoDB**	**Cassandra**	**CouchDB**
Data model	Document-based	Wide-column storage	Document-based
Scalability	Horizontal	Horizontal	Horizontal
Data model	JSON-like documents	Wide-column	JSON-like documents
Secondary indexes	Supported (multiple types)	Supported (on columns)	Supported
Data schema	Schema-free	Schema-free	Schema-free
Most common uses	Web apps real-time analytics	Time-series data, high write throughput	Offline support
Github stars	24.0k	8.0k	5.7k
Github forks	5.6k	3.4k	1.0k

Table 5.2: NoSQL database tools comparison

connected to each other and work together are explained in the next section, where the exact implementation proposal using these tools is described.

5.4 Implementation of proposed architecture using open-source tools (RQ4)

In line with the architecture defined in RQ2, and considering the tools reviewed and compared in RQ3, now an implementation of such architecture using open-source tools is now defined. Firstly, a holistic overview of the architecture will be defined and, secondly, a more specific example will be set to better understand how it operates.

5.4.1 Modules and configs version storage

On the one side, there is all the information regarding the created modules and versions and the used configs for such modules that form the pipelines. Let's imagine three running pipelines A, B and C (Figure 5.19). Each one of these pipelines will be made of several different modules (and versions) using different configs. For this reason, it is necessary to store all information about the created modules, as well as the used configs for these modules, to link this data with the information contained in the records from the event logs and, by doing this, be able to evaluate the performance of the modules and, with this, the running pipelines.

For this purpose, MongoDB is the NoSQL database chosen. Firstly, when creating a new module or a new version of an existing module, this information will be written and stored in MongoDB. In the form of a unique document, in a JSON object, for instance, information about the module name that has been created, as well as its version, will be stored. The actual content of the module (code, model, values, etc.) will also be stored in the same NoSQL database but in a separate document. To link the content of the module with the document containing its name and version, a one-to-many relationship [83] between the module document (parent) and its corresponding content documents (child) is proposed. To do this, a *"id": "unique_id"* is defined in the main module document, which is also present in the child document(s) that contains the content of the module, allowing the link between both documents.

Regarding the config modules, following the architecture defined in RQ2 and considering the concept shown in Figure 5.6, a new document will be created for storing this information. Again, for linking the config module document (child) information, containing the config module's name and version, with the main module document (parent), the same *"id": "unique_id"*, present in both documents, will be used. In a similar way, the content of the config files (e.g. training dataset, hyperparameter values, seed values, etc.) is also stored in the same database. Once again, the link between the config module document and its content will be done using a *"id": "config_unique_id"* present in both the config module document and the document containing its content. All this structure is depicted and can be better understood in Figure 5.19.

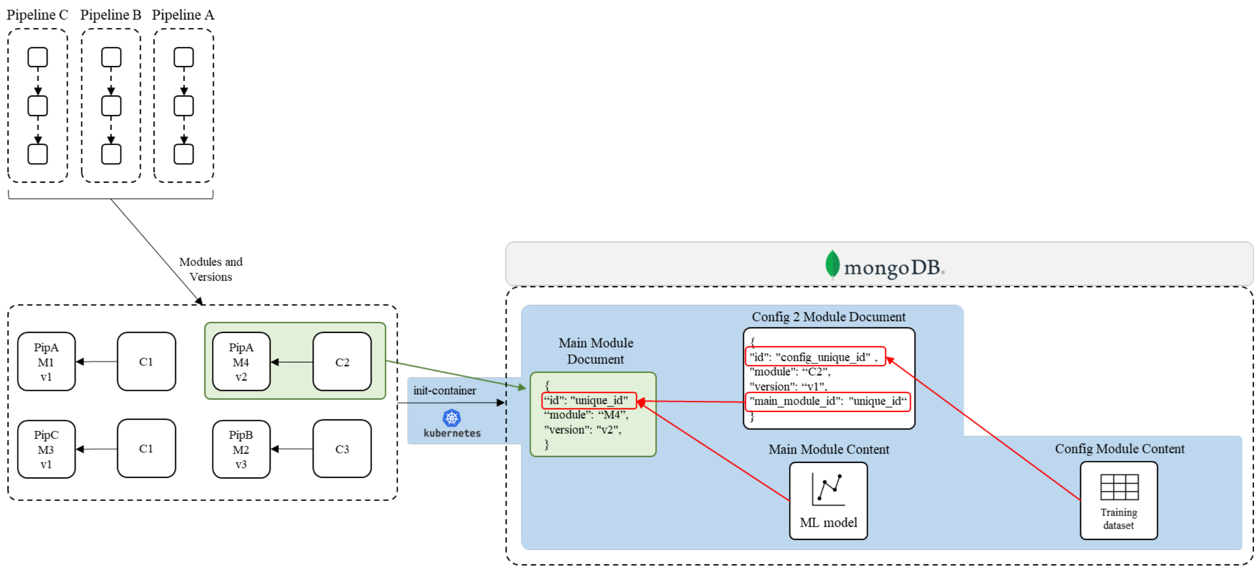


Figure 5.19: Modules (and versions) configs data storage in MongoDB

Now, it is necessary to define how this information will actually be stored in MongoDB when creating a new module. As the objective is to store this data when the module is created, a one-time procedure to do so must be triggered when the module is created. To achieve this, the use of Kubernetes' init-container feature [84] is proposed. An init-container is a container within a pod that runs to completion before the main application containers start. They are usually used for initialization tasks.

In this context, an init-container can help store the config information and config files of a new module or module version before it starts running, by executing the necessary code to retrieve the config information and files and store them in MongoDB. To achieve this, the following steps should be followed:

- **Define init-container:** an init-container should be defined at the end of the module YAML file, following the syntaxis found in the Kubernetes documentation [84]. This container will run before the module starts operating and will store the information regarding the module's information in MongoDB.
- **Connect to MongoDB:** the init-container must have the necessary access to connect to the MongoDB database (collection, credentials, port, etc.).
- **Extract module information:** to store the module name and version, some data needs to be extracted from the YAML file. To do so, the proposal is to parse the YAML file in a way that the needed information can be read and extracted.
- **Extract config information:** to store the config information, a similar procedure is proposed. Parsing the YAML file should allow one to extract the configs of the module, as well as the location of the files/code/parameters that must be saved in MongoDB.

- **Generate a unique identifier:** a *"unique_id"* for the main module document and a *"config_unique_id"* for the config module document must be defined by the init-container before storing everything in MongoDB, as a way of linking the module document with the associated saved files/values, as well as the config module document with the main module document, and the config module document with its content, also saved in MongoDB.
- **Create the module document:** create the MongoDB document for the main module with the unique id and the corresponding information.
- **Create the config document:** create the MongoDB document for the config module with the unique id and the corresponding information, as well as the reference id of the main module document.
- **Save the documents in MongoDB:** store the created documents in MongoDB.
- **Save the associated main module and config module content in MongoDB:** store the corresponding files/code/values in MongoDB.
- **Finalise init-container:** once everything has been correctly saved in MongoDB, the init-container can exit and the actual module must start operating.

Once again, with this structure, all information regarding the modules, versions, and configs created is available and organized and can be used for further purposes.

5.4.2 Logs ingestion and storage

On the other hand, there are the streaming data event logs, containing information about inputs, outputs, signals, and evaluator outputs of modules. As it has previously been said, data comes into the system in the form of continuous data streams and the communication between modules is done through event logs, where modules can write (producers) and read (consumers) in and from them, respectively.

Apache Kafka has been chosen as the log ingestion tool for managing all the information coming into the event logs and storing them in a centralized and accessible way. For this purpose, a module reads events or record(s) from Kafka one or multiple certain topics (module's input(s)) and, once the module has carried out its task, publishes new record(s) (module's output) in different Kafka topics. By doing this the continuous flow of data between the modules of a pipeline is guaranteed.

Kafka ensures scalability and fault tolerance by using partitions [76]. Each topic is broken into multiple partitions across several Kafka brokers, making it possible to read the same information about a topic from different servers at the same time, allowing numerous consumers to read the same information simultaneously. Nevertheless, Kafka automatically manages everything related to partitions by default. Thus, when defining a topic's producer or consumer, the user does not need to worry about partition management.

Let's consider the example in Figure 5.20 of a pipeline made of three modules in which communications are made through four Kafka topics. Topic 2 (T2), for instance, is spread over three partitions. In partition 2 (P2), there is a set of records, each one with the shown structure. It can be seen that each record is unique due to the *topic+partition+offset* combination [76], where offset denotes the position of a message inside a partition. Moreover, the record contains information regarding the *"type"* of the record info (input, output, signal, evaluator), *"logdata"* (record data), and other needed metadata.

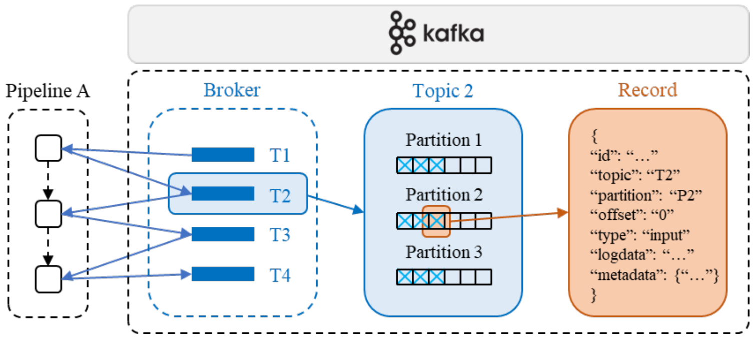


Figure 5.20: Module communication via Kafka and topic/record structure

This example can be extrapolated to the rest of Kafka's topics and partitions. With this, all records from all topics can be identified uniquely. In other words, all information coming in and out of all modules of a pipeline is traceable.

All this data could be accessible through Kafka if an indefinite data retention time was set. However, it may be more convenient to store this information persistently in a storage system that allows for optimized storage of log information. For this implementation proposal, and considering the same NoSQL database will be used to store all data, MongoDB is the log storage system of choice.

According to the architecture defined in RQ2, the information regarding the records of the event logs will be stored in documents in MongoDB. One document will be created for each type of record (input, output, signal, and evaluator output) for each module version using certain configs. By doing this, all the log information regarding a module and version will be linked to the document where the configs used by this main module are found. By doing this, if a certain version of a module is being used with different configs, the log information about this version of the module can be differentiated according to the config used. This structure allows storing all information without duplicating anything. Finally, a field for the *"pipeline"* can be found in the metadata of each record, to be able to know which pipeline each record belongs to. This could be useful in the case of the same module and version using the same config being present in multiple pipelines. This structure can be depicted in Figure 5.21.

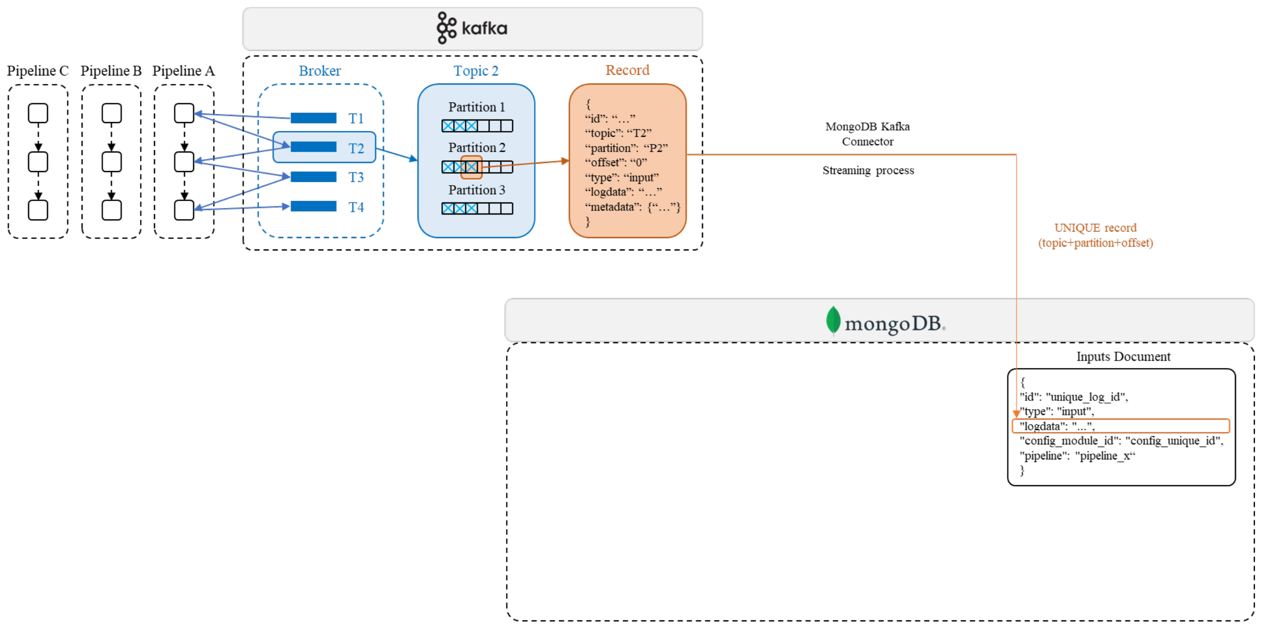


Figure 5.21: MongoDB structure and connection to Kafka

The communication between Kafka and MongoDB is possible using the MongoDB Kafka Connector [85]. It lets write data from a Kafka topic to MongoDB.

With this structure, all data from Kafka topics are stored persistently and in an organised and accessible way in MongoDB. From this point, MongoDB can act as the data source for implementing other systems, such as visualization and alerting systems.

5.4.3 Connection to other tools

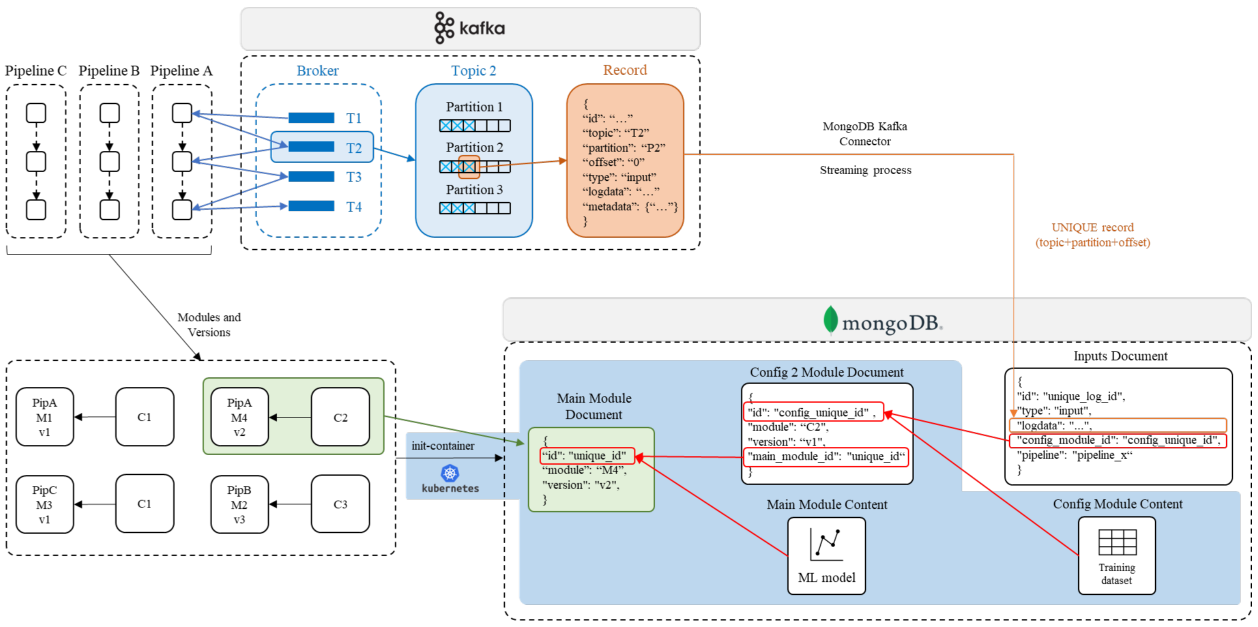
Having the data storage system defined, all data regarding the running pipelines is stored in an organised way and is accessible to other tools for other purposes.

As has been mentioned in this document, the main systems to implement from this data are a visualization system and an alerting system. For this purpose, Grafana [30] is a perfect choice, as it has both visualization and alerting capabilities. To be able to implement these systems in Grafana, a connection between it and the data storage system, MongoDB, is required. Luckily, Grafana does support connection for MongoDB [86]. Nonetheless, the configuration of this connection and the development of such systems is out of the scope of this project.

Aside from Grafana for monitoring and alerting capabilities, MongoDB is a mature solution with a big community of users and, thus, many other tools support connection to it.

5.4.4 Architecture implementation diagram

Taking into consideration everything said in this section, the whole architecture is defined and can be depicted in Figure 5.22.



```
{
“id”: “…”
“topic”: “T2”
“partition”: “P2”
“offset”: “0”
“type”: “input”
“logdata”: “…”
“metadata”: {“…”}
}
```

```
{
“id": "unique_id"
“module”: “M4”,
"version": "v2",
}
```

```
{
"id": "config_unique_id" ,
"module": “C2",
"version": “v1",
"main_module_id": "unique_id“
}
```

```
{
"id": "unique_log_id",
"type": "input",
"logdata": "...",
"config_module_id": "config_unique_id",
"pipeline": "pipeline_x“
}
```

Figure 5.22: Monitoring architecture implementation diagram

The proposed architecture lets one develop the desired additional systems having MongoDB as the main data source, with all the needed information about pipelines, their modules and their inputs, outputs, signals and configs.

Chapter 6

Discussion

In this section, the research findings are summarised and their implications, and limitations, are explored in a comprehensive manner.

There are plenty of areas where MLOps applications are used. This entails many different environments, data sizes and formats, ML models and algorithms, performance requirements, etc. Due to this heterogeneous landscape, when trying to define what should be monitored in a distributed machine-learning system, there is no unique answer. As it has been discussed in Section 5.1, regarding data and model monitoring, there are some concepts that are present in nearly all scenarios. These concepts (i.e. data drift, concept drift, performance metrics, and outlier detection) may be enough to ensure the correct performance of many ML solutions. However, this may not be the case in certain critical applications, where ensuring optimal performance is indispensable. For this reason, the reviewed concepts for monitoring both data and models should not be taken as an exhaustive list and the specific metrics should be defined for each use case, according to its needs.

Defining a software architecture for the monitoring system is subject to the streaming data nature of the problem. This context entails an inevitable coexistence of two types of data: streaming data and batch data. On the one hand, regarding streaming data, the design of the software architecture presents more challenges than conventional systems. Challenges like how to calculate certain metrics needed to evaluate the performance of modules on the run (a Module Evaluator module has been proposed for this aim), or how to manage and store continuous streams of data in an organized way (a dedicated log ingestion tool has been proposed). On the other hand, storing versioning information about the created modules in the pipelines is also necessary for monitoring the system, as it allows the linkage between the streaming data and the module information about its version and the used configs. Storing all this data in the NoSQL database permits retrieval of the desired data. For example, one could monitor the accuracy of a regression model module that is working on version 2 using certain hyperparameters and seed values for the model (Configs).

Moreover, a mixed modularity approach has been chosen when defining the software architecture. This may help towards gaining flexibility and major observability of arbitrary parts of the system. Nonetheless, this may also entail more challenges when configuring and setting the tools intended to carry out the needed tasks for the monitoring of the system, as they have to handle connections between a distributed set of monitoring modules (evaluator modules) and a centralized system for gathering and storing all the data.

It is also worth mentioning the benefits of using open-source tools for the design of the monitoring system. The wide range of open-source available options makes creating a software solution based on this type of tool a viable option. Using these tools allows for creating a cost-effective and flexible solution, with the possibility to customize and extend the tool's capabilities. However, they may entail more challenges and programming knowledge than using a paid solution. For the purpose of this project, there have been found suitable tools that fulfil the needed requirements, and also with a big and active community. Other tools could also be used for developing a similar solution, although the most suitable have been chosen for this case.

Finally, a software implementation for the proposed architecture has been defined. Although it has been explained how the different tools and parts of the system would be connected and work together, many issues and challenges may arise when actually putting what has been stated into practice. The final configuration of the chosen tools and the proposed architectural behaviour would need to be tested in a testing environment before developing a full monitoring system based on the blueprint provided.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The current research project aimed to design a monitoring system for distributed data processing pipelines, in the context of ECiDA and machine learning applications.

To do so, firstly, an academic review has been conducted to define what must be monitored in these systems. Data, model and resource usage are the three main actors to take into account. While data and model monitoring help to understand how the ML solution performs, resource usage monitoring allows for the detection of any existing problems in the underlying infrastructure that enables the ML solution to run in the first place.

Once defined what must be monitored in these pipelines, a software architecture proposal for the monitoring system has been provided. A NoSQL storage system has been proposed for storing all data needed to monitor the pipelines in an organized and accessible way. On the one hand, there is data concerning the versioning of the pipelines and their components; on the other hand, there is data containing all communications and information exchange between the components that make up the pipelines.

A review and comparison of open-source tools potentially suitable for implementing the proposed software architecture have then been carried out, defining the needed features for the potential tool and comparing their degree of fulfilment for the different candidates. Next, a toolset has been chosen and the compatibility of such tools has been checked. Finally, having the toolset defined, a possible implementation of the software architecture has been given, providing details on how the data would be gathered, managed, and stored, and how the different tools would connect and work together.

The monitoring system has proven to be a critical part of distributed ML pipelines, as it enables gaining visibility over these pipelines and ensuring their health and desired performance, thus allowing the development of robust and trustworthy applications, with no or minimal downtimes. Nevertheless, its design is a complex task, as it entails most of the components of the system and it can be challenging to gather and manage all needed data from them, without compromising other components or negatively affecting the performance of the application.

The proposed design for the monitoring system could serve as a base for the development of monitoring systems for similar distributed systems. Regarding the current project, it could be used as a blueprint for developing the monitoring system needed for the ECiDA platform.

7.2 Future Work

This section presents potential future work that can extend the findings and contributions of this research:

- Develop further needed systems, such as visualization, alerting, or an AI recommender system, by using all the data regarding the running pipelines and their components stored in a centralized way.
- Define and explore in more detail how the Module Evaluator module would work, seeing how monitoring concepts such as data drift and concept drift could be detected in such modules. Nonetheless, and considering what has been said about drift detection techniques being subject to data formats and model types, this would be a use-case-specific task.
- Further explore the capabilities of the init-container feature of Kubernetes and fully optimize the process of storing all desired data regarding modules' information and version, to minimize the time needed to store this information in the storage system before the actual module starts operating.

Bibliography

[1] Distributed Systems. Distributed Systems — cs.rug.nl. `https://www.cs.rug.nl/ds/Research/ECiDA`. [Accessed 06-Jun-2023].

[2] ECiDA Core team. ECiDA - Evolutionary Changes in Data Analysis — ecida.nl. `https://ecida.nl/`. [Accessed 06-Jun-2023].

[3] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A survey of devops concepts and challenges. *ACM Computing Surveys (CSUR)*, 52(6):1–35, 2019.

[4] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[5] Mali Senapathi, Jim Buchan, and Hady Osman. Devops capabilities, practices, and challenges: Insights from a case study. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*, pages 57–67, 2018.

[6] What are Microservices? — AWS — aws.amazon.com. `https://aws.amazon.com/microservices/?nc1=h_ls`. [Accessed 12-Jun-2023].

[7] Liming Zhu, Len Bass, and George Champlin-Scharff. Devops and its practices. *IEEE software*, 33(3):32–34, 2016.

[8] Yalin Baştanlar and Mustafa Özuysal. Introduction to machine learning. *miRNomics: MicroRNA biology and computational analysis*, pages 105–128, 2014.

[9] Philipp Ruf, Manav Madan, Christoph Reich, and Djaffar Ould-Abdeslam. Demystifying mlops and presenting a recipe for the selection of open-source tools. *Applied Sciences*, 11(19):8861, 2021.

[10] Georgios Symeonidis, Evangelos Nerantzis, Apostolos Kazakis, and George A Papakostas. Mlops-definitions, tools and challenges. In *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0453–0460. IEEE, 2022.

[11] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. Machine learning operations (mlops): Overview, definition, and architecture. *arXiv preprint arXiv:2205.02302*, 2022.

[12] Rob Ashmore, Radu Calinescu, and Colin Paterson. Assuring the machine learning lifecycle: Desiderata, methods, and challenges. *ACM Computing Surveys (CSUR)*, 54(5):1–39, 2021.

[13] Nipuni Hewage and Dulani Meedeniya. Machine learning operations: A survey on mlops tool support. *arXiv preprint arXiv:2202.10169*, 2022.

[14] Ruben Mayer and Hans-Arno Jacobsen. Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Computing Surveys (CSUR)*, 53(1):1–37, 2020.

[15] Amitabha Banerjee, Chien-Chia Chen, Chien-Chun Hung, Xiaobo Huang, Yifan Wang, and Razvan Chevesaran. Challenges and experiences with mlops for performance diagnostics in hybrid-cloud enterprise software deployments. In *2020 USENIX Conference on Operational Machine Learning (OpML 20)*. USENIX Association, 2020.

[16] Tuomas Granlund, Aleksi Kopponen, Vlad Stirbu, Lalli Myllyaho, and Tommi Mikkonen. Mlops challenges in multi-organization setup: Experiences from two real-world cases. In *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*, pages 82–88. IEEE, 2021.

[17] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D Lawrence. Challenges in deploying machine learning: a survey of case studies. *ACM Computing Surveys*, 55(6):1–29, 2022.

[18] Janis Klaise, Arnaud Van Looveren, Clive Cox, Giovanni Vacanti, and Alexandru Coca. Monitoring and explainability of models in production. *arXiv preprint arXiv:2007.06299*, 2020.

[19] Tim Schröder and Michael Schulz. Monitoring machine learning models: a categorization of challenges and methods. *Data Science and Management*, 5(3):105–116, 2022.

[20] Lucas Cardoso Silva, Fernando Rezende Zagatti, Bruno Silva Sette, Lucas Nildaimon dos Santos Silva, Daniel Lucrédio, Diego Furtado Silva, and Helena de Medeiros Caseli. Benchmarking machine learning solutions in production. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 626–633. IEEE, 2020.

[21] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchỳ. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review*, 52:77–124, 2019.

[22] Łukasz Kufel. Tools for distributed systems monitoring. *Foundations of Computing and Decision Sciences*, 41, 12 2016.

[23] Marek Moravcik and Martin Kontsek. Overview of docker container orchestration tools. In *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 475–480. IEEE, 2020.

[24] Isam Mashhour Al Jawarneh, Paolo Bellavista, Filippo Bosi, Luca Foschini, Giuseppe Martuscelli, Rebecca Montanari, and Amedeo Palopoli. Container orchestration engines: A thorough functional and performance comparison. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2019.

[25] Octavian Mart, Catalin Negru, Florin Pop, and Aniello Castiglione. Observability in kubernetes cluster: Automatic anomalies detection using prometheus. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 565–570. IEEE, 2020.

[26] Balqees Talal Hasan and Dhuha Basheer Abdullah. Real-time resource monitoring framework in a heterogeneous kubernetes cluster. In *2022 Muthanna International Conference on Engineering Science and Technology (MICEST)*, pages 184–189. IEEE, 2022.

[27] Marcel Großmann and Clemens Klug. Monitoring container services at the network edge. In *2017 29th International Teletraffic Congress (ITC 29)*, volume 1, pages 130–133. IEEE, 2017.

[28] Samuel Idowu, Daniel Strüber, and Thorsten Berger. Asset management in machine learning: State-of-research and state-of-practice. *ACM Computing Surveys*, 55(7):1–35, 2022.

[29] Prometheus. Prometheus - Monitoring system & time series database — prometheus.io. `https://prometheus.io/`. [Accessed 08-Apr-2023].

[30] Grafana: The open observability platform — Grafana Labs — grafana.com. `https://grafana.com/`. [Accessed 08-Apr-2023].

[31] MLflow - A platform for the machine learning lifecycle — mlflow.org. `https://mlflow.org/`. [Accessed 01-May-2023].

[32] Home — AimStack — aimstack.io. `https://aimstack.io/`. [Accessed 01-May-2023].

[33] GitHub - aimhubio/aimlflow: aim-mlflow integration — github.com. `https://github.com/aimhubio/aimlflow`. [Accessed 01-May-2023].

[34] Kubeflow — kubeflow.org. `https://www.kubeflow.org/`. [Accessed 01-May-2023].

[35] Łukasz Kufel. Tools for distributed systems monitoring. *Foundations of computing and decision sciences*, 41(4):237–260, 2016.

[36] Gilberto Recupito, Fabiano Pecorelli, Gemma Catolino, Sergio Moreschini, Dario Di Nucci, Fabio Palomba, and Damian A Tamburri. A multivocal literature review of mlops tools and features. In *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 84–91. IEEE, 2022.

[37] Anders Köhler. Evaluation of mlops tools for kubernetes: A rudimentary comparison between open source kubeflow, pachyderm and polyaxon, 2022.

[38] René Peinl, Florian Holzschuher, and Florian Pfitzer. Docker cluster management for the cloud-survey results and own solution. *Journal of Grid Computing*, 14(2):265–282, 2016.

[39] Open source Machine Learning at scale with Kubernetes - MLOps Lifecycle for data-scientists & machine-learning engineers - Model & Data Centric AI - Polyaxon — polyaxon.com. `https://polyaxon.com/`. [Accessed 01-May-2023].

[40] Evidently AI - Open-Source Machine Learning Monitoring — evidentlyai.com. `https://www.evidentlyai.com/`. [Accessed 01-May-2023].

[41] Zabbix :: The Enterprise-Class Open Source Network Monitoring Solution — zabbix.com. `https://www.zabbix.com/`. [Accessed 01-May-2023].

[42] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and software technology*, 106:101–121, 2019.

[43] I Sessione di Laurea. *Mlops-standardizing the machine learning workflow*. PhD thesis, University of Bologna Bologna, Italy, 2021.

[44] Monitoring ML systems in production. Which metrics should you track? — evidentlyai.com. `https://www.evidentlyai.com/blog/ml-monitoring-metrics`. [Accessed 16-May-2023].

[45] ML Model Monitoring — Fiddler AI — fiddler.ai. `https://www.fiddler.ai/ml-model-monitoring`. [Accessed 16-May-2023].

[46] Stephen Oladele. A Comprehensive Guide on How to Monitor Your Models in Production — neptune.ai. `https://neptune.ai/blog/how-to-monitor-your-models-in-production-guide`. [Accessed 16-May-2023].

[47] A Guide to Monitoring Machine Learning Models in Production — NVIDIA Technical Blog — developer.nvidia.com. `https://developer.nvidia.com/blog/a-guide-to-monitoring-machine-learning-models-in-production/`. [Accessed 16-May-2023].

[48] Emeli Dral. Monitoring Machine Learning Models in Production: How to Track Data Quality and Integrity? — towardsdatascience.com. `https://towardsdatascience.com/monitoring-machine-learning-models-in-production-how-to-track-data-quality-and-integrity-391435c8a299`. [Accessed 16-May-2023].

[49] Aayush Bajaj. Performance Metrics in Machine Learning [Complete Guide] - neptune.ai — neptune.ai. `https://neptune.ai/blog/performance-metrics-in-machine-learning-complete-guide`. [Accessed 15-May-2023].

[50] Aditya Mishra. Metrics to Evaluate your Machine Learning Algorithm — towardsdatascience.com. `https://towardsdatascience.com/metrics-to-evaluate-your-machine-learning-algorithm-f10ba6e38234`. [Accessed 15-May-2023].

[51] Songhao Wu. What are the best metrics to evaluate your regression model? — towardsdatascience.com. `https://towardsdatascience.com/what-are-the-best-metrics-to-evaluate-your-regression-model-418ca481755b`. [Accessed 15-May-2023].

[52] Anita Rácz, Dávid Bajusz, and Károly Héberger. Multi-level comparison of machine learning classifiers and their performance metrics. *Molecules*, 24(15):2811, 2019.

[53] Alexei Botchkarev. Performance metrics (error measures) in machine learning regression, forecasting and prognostics: Properties and typology. *arXiv preprint arXiv:1809.03006*, 2018.

[54] Aparna Dhinakaran. The Playbook to Monitor Your Model's Performance in Production — arize.com. `https://arize.com/blog/monitor-your-model-in-production/`. [Accessed 24-May-2023].

[55] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D Sculley. The ml test score: A rubric for ml production readiness and technical debt reduction. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 1123–1132. IEEE, 2017.

[56] Stephen Oladele. A Comprehensive Guide on How to Monitor Your Models in Production — neptune.ai. `https://neptune.ai/blog/how-to-monitor-your-models-in-production-guide`. [Accessed 24-May-2023].

[57] Yifei Yuan, Zhixiong Wang, and Wei Wang. Unsupervised concept drift detection based on multi-scale slide windows. *Ad Hoc Networks*, 111:102325, 2021.

[58] What is Concept Drift: Causes and How to Deal with it? — censius.ai. `https://censius.ai/blogs/what-is-concept-drift-and-why-does-it-go-undetected`. [Accessed 05-Jun-2023].

[59] Supriya Agrahari and Anil Kumar Singh. Concept drift detection in data stream mining: A literature review. *Journal of King Saud University-Computer and Information Sciences*, 34(10):9523–9540, 2022.

[60] Douglas M Hawkins. *Identification of outliers*, volume 11. Springer, 1980.

[61] Abir Smiti. A critical overview of outlier detection methods. *Computer Science Review*, 38:100306, 2020.

[62] Omar Alghushairy, Raed Alsini, Terence Soule, and Xiaogang Ma. A review of local outlier factor algorithms for outlier detection in big data streams. *Big Data and Cognitive Computing*, 5(1):1, 2020.

[63] Prakhar Mishra. 5 Outlier Detection Techniques that every "Data Enthusiast" Must Know — towardsdatascience.com. https://towardsdatascience.com/5-outlier-detection-methods-that-every-data-enthusiast-must-know-f917bf439210. [Accessed 16-May-2023].

[64] Azzedine Boukerche, Lining Zheng, and Omar Alfandi. Outlier detection: Methods, models, and classification. *ACM Computing Surveys (CSUR)*, 53(3):1–37, 2020.

[65] Hongzhi Wang, Mohamed Jaward Bah, and Mohamed Hammad. Progress in outlier detection techniques: A survey. *Ieee Access*, 7:107964–108000, 2019.

[66] Mahsa Salehi and Lida Rashidi. A survey on anomaly detection in evolving data: [with application to forest fire risk prediction]. *ACM SIGKDD Explorations Newsletter*, 20(1):13–23, 2018.

[67] Samuel Ackerman, Eitan Farchi, Orna Raz, Marcel Zalmanovici, and Parijat Dube. Detection of data drift and outliers affecting machine learning model performance over time. *arXiv preprint arXiv:2012.09258*, 2020.

[68] Guo Fu, Yanfeng Zhang, and Ge Yu. A fair comparison of message queuing systems. *IEEE Access*, 9:421–432, 2020.

[69] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. A survey of distributed data stream processing frameworks. *IEEE Access*, 7:154300–154316, 2019.

[70] Apostolos Lazidis, Konstantinos Tsakos, and Euripides GM Petrakis. Publish–subscribe approaches for the iot and the cloud: Functional and performance evaluation of open-source systems. *Internet of Things*, 19:100538, 2022.

[71] yasir saeed. Top 5 Open Source Message Queue (MQ) Software In 2021 — blog.containerize.com. https://blog.containerize.com/top-5-open-source-message-queue-software-in-2021/. [Accessed 24-Jun-2023].

[72] What are the best Message Queue Tools? — stackshare.io. https://stackshare.io/message-queue. [Accessed 24-Jun-2023].

[73] Kafka vs Pulsar - Performance, Features, and Architecture Compared — confluent.io. https://www.confluent.io/kafka-vs-pulsar/#:~:text=Kafka%20is%20a%20pure%20distributed,it%20synthesizes%20some%20similar%20properties.

[74] Apache Kafka — kafka.apache.org. https://kafka.apache.org/. [Accessed 17-Jun-2023].

[75] Messaging that just works &2014; RabbitMQ — rabbitmq.com. https://www.rabbitmq.com/. [Accessed 17-Jun-2023].

[76] Apache Kafka — kafka.apache.org. https://kafka.apache.org/documentation/. [Accessed 20-Jun-2023].

[77] Adity Gupta, Swati Tyagi, Nupur Panwar, Shelly Sachdeva, and Upaang Saxena. Nosql databases: Critical analysis and comparison. In *2017 International conference on computing and communication technologies for smart nation (IC3TSN)*, pages 293–299. IEEE, 2017.

[78] Yishan Li and Sathiamoorthy Manoharan. A performance comparison of sql and nosql databases. In *2013 IEEE Pacific Rim conference on communications, computers and signal processing (PACRIM)*, pages 15–19. IEEE, 2013.

[79] Bogdan George Tudorica and Cristian Bucur. A comparison between several nosql databases with comments and notes. In *2011 RoEduNet international conference 10th edition: Networking in education and research*, pages 1–5. IEEE, 2011.

[80] MongoDB: The Developer Data Platform — mongodb.com. `https://www.mongodb.com/`. [Accessed 24-Jun-2023].

[81] Apache Cassandra — Apache Cassandra Documentation — cassandra.apache.org. `https://cassandra.apache.org/_/index.html`. [Accessed 24-Jun-2023].

[82] Apache CouchDB — couchdb.apache.org. `https://couchdb.apache.org/`. [Accessed 24-Jun-2023].

[83] Model One-to-Many Relationships with Document References &x2014; MongoDB Manual — mongodb.com. `https://www.mongodb.com/docs/manual/tutorial/model-referenced-one-to-many-relationships-between-documents/`. [Accessed 18-Jun-2023].

[84] Init Containers — kubernetes.io. `https://kubernetes.io/docs/concepts/workloads/pods/init-containers/`. [Accessed 21-Jun-2023].

[85] MongoDB Kafka Connector x2014; MongoDB Kafka Connector — mongodb.com. `https://www.mongodb.com/docs/kafka-connector/current/`. [Accessed 05-Jul-2023].

[86] MongoDB plugin for Grafana — Grafana Labs — grafana.com. `https://grafana.com/grafana/plugins/grafana-mongodb-datasource/`. [Accessed 18-Jun-2023].