DEGREE PROJECT IN TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2020

Scalable Architecture for Automating Machine Learning Model Monitoring

KTH Thesis Report

Javier de la Rúa Martínez

KTH ROYAL INSTITUTE OF TECHNOLOGY
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Authors

Javier de la Rúa Martínez <jdlrm@kth.se>
Information and Communication Technology
KTH Royal Institute of Technology

Place for Project

Stockholm, Sweden
LogicalClocks, RISE SICS, Electrum Kista

Examiner

Seif Haridi
Electrum 229, Kistagången 16, 16440 Kista, Sweden
KTH Royal Institute of Technology

Supervisor

Jim Dowling

LogicalClocks, RISE SICS, Electrum Kista

KTH Royal Institute of Technology

Abstract

Last years, due to the advent of more sophisticated tools for exploratory data analysis, data management, Machine Learning (ML) model training and model serving into production, the concept of MLOps has gained more popularity. As an effort to bring DevOps processes to the ML lifecycle, MLOps aims at more automation in the execution of diverse and repetitive tasks along the cycle and at smoother interoperability between teams and tools involved. In this context, the main cloud providers have built their own ML platforms [4, 34, 61], offered as services in their cloud solutions. Moreover, multiple frameworks have emerged to solve concrete problems such as data testing, data labelling, distributed training or prediction interpretability, and new monitoring approaches have been proposed [32, 33, 65]. Among all the stages in the ML lifecycle, one of the most commonly overlooked although relevant is model monitoring. Recently, cloud providers have presented their own tools to use within their platforms [4, 61] while work is ongoing to integrate existent frameworks [72] into open-source model serving solutions [38]. Most of these frameworks are either built as an extension of an existent platform (i.e lack portability), follow a scheduled batch processing approach at a minimum rate of hours, or present limitations for certain outliers and drift algorithms due to the platform architecture design in which they are integrated. In this work, a scalable automated cloud-native architecture is designed and evaluated for ML model monitoring in a streaming approach. An experimentation conducted on a 7-node cluster with 250.000 requests at different concurrency rates shows maximum latencies of 5.9, 29.92 and 30.86 seconds after request time for 75% of distance-based outliers detection, windowed statistics and distribution-based data drift detection, respectively, using windows of 15 seconds length and 6 seconds of watermark delay.

Keywords

Model Monitoring, Streaming, Scalability, Cloud-native, Data Drift, Outliers, Machine Learning

Abstract

Under de senaste åren har konceptet MLOps blivit alltmer populärt på grund av tillkomsten av mer sofistikerade verktyg för explorativ dataanalys, datahantering, modell-träning och model serving som tjänstgör i produktion. Som ett försök att föra DevOps processer till Machine Learning (ML)-livscykeln, siktar MLOps på mer automatisering i utförandet av mångfaldiga och repetitiva uppgifter längs cykeln samt på smidigare interoperabilitet mellan team och verktyg inblandade. I det här sammanhanget har de största molnleverantörerna byggt sina egna ML-plattformar [4, 34, 61], vilka erbjuds som tjänster i deras molnlösningar. Dessutom har flera ramar tagits fram för att lösa konkreta problem såsom datatestning, datamärkning, distribuerad träning eller tolkning av förutsägelse, och nya övervakningsmetoder har föreslagits [32, 33, 65]. Av alla stadier i ML-livscykeln förbises ofta modellövervakning trots att det är relevant. På senare tid har molnleverantörer presenterat sina egna verktyg att kunna användas inom sina plattformar [4, 61] medan arbetet pågår för att integrera befintliga ramverk [72] med lösningar för modellplatformer med öppen källkod [38]. De flesta av dessa ramverk är antingen byggda som ett tillägg till en befintlig plattform (dvs. saknar portabilitet), följer en schemalagd batchbearbetningsmetod med en lägsta hastighet av ett antal timmar, eller innebär begränsningar för vissa extremvärden och drivalgoritmer på grund av plattformsarkitekturens design där de är integrerade. I det här arbetet utformas och utvärderas en skalbar automatiserad molnbaserad arkitektur för ML-modellövervakning i en streaming-metod. Ett experiment som utförts på ett 7-nodskluster med 250.000 förfrågningar vid olika samtidigheter visar maximala latenser på 5,9, 29,92 respektive 30,86 sekunder efter tid för förfrågningen för 75% av avståndsbaserad detektering av extremvärden, windowed statistics och distributionsbaserad datadriftdetektering, med hjälp av windows med 15 sekunders längd och 6 sekunders fördröjning av vattenstämpel.

Nyckelord

Modellövervakning, Streaming-metod, Skalbarhet, Molnbaserad, Dataskift, Outlier-upptäckt, Maskininlärning

Dedication

To my family, for their strength and support in times of adversity: Cristina, Mª Teresa and Javier.

Acknowledgements

First, I would like to express my very great appreciation to my supervisor Jim Dowling, associate professor at KTH Royal Institute of Technology and CEO of LogicalClocks AB, for the opportunity to join the team and work on this project, his guidance throughout its development and for being an inspiration. Also, I wish to acknowledge the help provided by the team in getting to know Hopsworks and troubleshooting occasional problems, specially Theofilos Kakantousis, Antonios Kouzoupis and Robin Andersson.

Secondly, I wish to express my gratitude to my fellows, friends and family. Especially to those whose moral support from the distance has been essential.

Also, I wish to express my deep appreciation to Cristina for her constant attention and relentless encouragement, vital for the completion of this thesis.

Lastly, I would like to thank EIT Digital for the opportunity to accomplish this master's studies abroad, and the enriching experience it entails.

Acronyms

ADWIN ADaptive WINdowing
AKS Azure Kubernetes Service
API Application Programming Interface
AWS Amazon Web Services
CRD Custom Resource Definition
CPU Central Processing Unit
DevOps Development Operations
DDM Drift Detection Method
DNN Deep Neural Network
EDDM Early Drift Detection Method
EMD Earth Mover's Distance
EKS Elastic Kubernetes Service
FaaS Function as a Service
GKE Google Kubernetes Engine
HDDM Heoffding's inequality based Drift Detection Method
IaaS Infrastructure as a Service
IQR Interquartile Range
JS Jensen-Shannon
KL Kullback-Leibler
KVM Kernel Virtual Machine
LXC Linux Container
ML Machine Learning
MLOps Machine Learning Operations
OS Operating System
PaaS Platform as a Service
SaaS Software as a Service

SLA	Service Level Agreement
VM	Virtual Machine
VMM	Virtual Machine Monitor
XAI	Explainable Artificial Intelligence

Contents

Chapter 1

Introduction

The advancements in sophisticated ML algorithms and distributed training techniques, as well as the great progress in distributed systems technologies seeking more scalability and efficiency in the exploitation of resources – frequently influenced by the serverless paradigm – have led to the appearance of diverse and complex tools to satisfy the needs of every role across the ML lifecycle. Concepts such as feature store, which provides a centralized way to store and manage features, or frameworks such Horovod, built for efficient distributed model training, make feature engineering and model training easier and more automated.

As for model monitoring, frameworks such as Deequ [70] or Alibi [72] has been developed to address data validation and outliers detection on static data frames. They can be used to monitor inference data and predictions of productionized models in a batch-processing approach. Recently, an attempt to integrate these frameworks in open-source model serving platforms such as KFserving [38] or Seldon Core is being made. However, the architectural design of these platforms where the model is served in a decentralized fashion presents some challenges and shortages when it comes to scalable and automated monitoring in a streaming fashion. Cloud providers have implemented their solutions such as Sagemaker Monitor [4] or Azure ML Monitor [61]. These solutions follow a scheduled batch processing approach at a minimum rate of hours and commonly involve higher technical debts in terms of dependency within the system.

1.1 Background

MLOps is a set of practices seeking to improve collaboration and communication between the different teams involved in all the stages of the ML lifecycle, from data collection to model releasing into production. Bringing the application of DevOps techniques (e.g continuous development and delivery) to the cycle, it aims at providing automation, governance and agility to the whole process. Two relevant stages of this cycle are model serving and model monitoring. The former refers to making a pre-trained ML algorithm available for making predictions from given inference instances. The latter relates to the constant supervision of the model being served, analysing its performance over time and taking actions when an undesired state of the model is detected. As any other deployment in cloud-native environments, both concepts require special attention on scalability, availability and reliability.

When it comes to serving an ML algorithm, there is a wide variety of solutions available (see Section 2.2). Commonly, the model server implementation provided by the framework utilized for training is used to prepare the model for production. Productionizing an ML model involves the same burden than deploying other kinds of cloud applications, requiring infrastructure-related management such as authorization, networking, resource allocation or scalability. Recent approaches attempt to bring model serving closer to the server-less paradigm, which aims at improving the exploitation of more fine-grained resources under highly variable demands while providing an extra layer of abstraction on infrastructure management.

Regarding ML monitoring, the degradation of a model in production can be produced by multiple reasons (see Section 2.3.2). Approaches to supervise model performance include the detection of outliers, concept drift and adversarial attacks, as well as providing explainability to predictions. This supervision can be conducted via training set validation, inference data analysis or comparing inference data with a given baseline. Some of these approaches use more traditional techniques such as measuring the distance between probability distributions [69, 76], while others make use of more sophisticated algorithms [8, 13, 30]. A lot of research has been done on detecting drift [31, 52, 56, 81] and outliers [18, 42, 43] in data streams. The bulk of the work related to concept drift detection in data streams either follows a supervised approach (i.e labels are required at prediction time) or presents limitations in cloud-native architectures

when the inference data is decentralized. Moreover, a great number of these research studies aim at integrating these methods in active learning algorithms, helping to decide when to perform another training step.

More recently, research works have proposed new performance metrics [32, 33] and frameworks [65] with ML model monitoring as their main purpose, facilitating their integration in cloud-native projects as part of the ML lifecycle. However, proposals for cloud-native scalable architectures for streaming model monitoring are scarce and commonly with high technical debt regarding vendor dependencies such as DataRobot or Iguanzo platforms.

1.2 Problem

MLOps is becoming more relevant and multiple tools are being developed to improve automation and interoperability between different parts of the ML lifecycle. While solutions for model serving are multiple, solutions for model monitoring are scarce and lack maturity. The integration between current serving and monitoring tools is non-sophisticated. The majority of existent alternatives are solutions running in a batch fashion at a minimum rate of hours or solutions where existent frameworks for outlier and drift detection on static data sets have been integrated into serving platforms, commonly presenting limitations due to their architectural design. Outliers and drift detection algorithms vary in their functioning and needs. Some of them are:

- Access to descriptive statistics of the training set used for model training.
- Access to all the inference data available at a given point in time.
- Window-based operations on the inference data.
- Capacity to parallelize computations.
- Stateful operations.

Scalability and streaming processing is relevant for scenarios with huge amounts of inference data, highly variable demand peaks or workflows requiring decision-making based on the model performance (i.e declaring a model obsolete or triggering another online training step).

Therefore, the problem is twofold:

- Lack of alternatives addressing ML model monitoring in production with support for different types of algorithms.
- Need of scalable automated cloud-native architectures for model monitoring in a streaming approach with low technical debts.

Therefore, the research question can be formulated as: How can we design and evaluate an architecture that allows scalable automated cloud-native ML model serving and monitoring in a streaming fashion with support for multiple outlier and drift algorithms?

1.3 Purpose

The purpose of this thesis is to answer the research question mentioned above by designing and evaluating an architecture for scalable and automated ML model monitoring with a streaming approach. A cloud-native solution with low technical debt in terms of dependency is desirable. Lastly, an extendable design for the implementation of additional algorithms, data sources and sinks is also convenient.

1.4 Goal

The main goal of this work is to design and evaluate an architecture for scalable and automated ML model serving and monitoring. In order to succeed in that goal, the following sub-goals are pursued:

- To implement a framework that computes statistics, outliers and drift detection on top of a streaming platform.
- To design a cloud-native solution to ingest inference data, compute statistics, detect outliers and concept drift, and generate alerts for later decision-making.
- To provide a simple and abstract way to deploy and configure the behaviour of the system including data sources, sinks for generated analysis and statistics, outliers and drift algorithms to compute.
- To analyse the scalability of the solution presented.

1.5 Benefits, Ethics and Sustainability

Monitoring models in a scalable and streaming approach allows a continuous supervision of model degradation and inference data validity. Providing continuous insights of model performance can help in its maintenance and, therefore, in achieving its optimal use. By these means, the ethical sense of this work depends on the purpose of the ML algorithm being monitored.

On the other hand, model monitoring can help in deciding how veracious predictions are, which is specially relevant in some sectors such as health, where ML algorithms are becoming more relevant and used for multiple diagnosis.

From a sustainability point of view, deficient models can be detected more rapidly, helping to avoid wasting resources. These resources could be re-allocated for other purposes or the model could be updated making their consumption worthwhile.

Additionally, a continuous supervision of the inference data used for making predictions presents clear benefits in security terms. The wide adoption of ML projects in practically any sector leads to the emergence of new and more sophisticated adversarial attacks, where input data manipulation plays an important role.

1.6 Methodology

To evaluate the framework and architecture proposed, an experimentation is conducted using an HTTP load generator to make requests in a controlled manner, given the number of requests, inference instances and concurrency levels. These instances are generated with different characteristics: normal instances, instances with outliers and instances with concept drifts.

After that, a monitoring tool is used to scrape system performance metrics from the different components including resource consumption, latencies, throughput and response times. Additionally, the inference analysis obtained throughout the experimentation is examined and contrasted with the instances generated previously.

Finally, a quantitative analysis is conducted over the collected data and inductive reasoning is used to conclude if the proposal successfully provides an answer to the research question.

1.7 Delimitations

Regarding the framework implemented in this work, only distance-based outliers and data distribution-based drift detection algorithms are included. These algorithms make use of baseline (i.e training set) descriptive statistics and inference descriptive statistics to detect anomalous events. This excludes algorithms that use additional resources such as pre-trained models, as well as algorithms applied directly over windowed instance values instead of their descriptive statistics. Notwithstanding this, the framework has been developed considering these types of algorithms and designed in a way that facilitates its implementation by extending the corresponding interfaces provided. Additionally, the framework only supports continuous variables at the moment of this work.

As for the data sources and sinks supported, this work has been focused on Kafka [25] as both inference data ingestion and generated analysis storage.

Lastly, the scalability of the architecture is analyzed based on response times, latencies, throughput and resource consumption, all of these contrasted with the number of replicas created and destroyed dynamically along the experimentation. Exhaustive analysis on the number and duration of cold-starts as well as more fain-grained resource exploitation is left for future work.

1.8 Outline

Firstly, a background section is presented including a more detailed description of MLOps, infrastructure-related concepts and alternatives for model serving and monitoring. An overview of outliers and concept drift detection approaches is included in this section. Then, an explanation of the methodologies carried out in the project is provided. After introducing the concepts and methodologies needed to define the context of the rest of the report, a description of the implementations and architecture proposal is presented in the next chapter. This is followed by the experimentation conducted to collect performance metrics and inference analysis. Subsequently, these metrics are presented and analyzed. At the end of the report, a section is included where the main conclusions taken from the experimentation results are discussed, together with potential future work.

Chapter 2

Background

In this chapter, the context of the thesis is introduced, needed for the understanding of the rest of the report. Firstly, MLOps discipline and the ML lifecycle are described to better locate the stages of model serving and monitoring in ML projects.

Subsequently, ML model serving is explained in more detailed. This section includes an introduction and comparison of technologies and cloud services for that purpose as well as an exhaustive description of Kubernetes [40], a container orchestrator, and KFServing [38], a model serving tool on top of Kubernetes.

After covering model serving, the main approaches for model monitoring are introduced together with different related concepts. These concepts include causes of model degradation, approaches to analyse model performance and a classification of the most commonly used algorithms.

2.1 MLOps and the ML lifecycle

In an era where the number of companies with ML in their roadmap is increasing rapidly, and the influence of ML is almost everywhere, a new awareness emerged with a community growing at the same pace. This awareness is well exposed by Google engineers in [71] explaining that traditional software practices fall short in ML projects. The authors enumerate the deficiencies of traditional practices and complexities of these kind of projects, referring to *hidden technical debts* such as strong data dependency or eroded boundaries. Furthermore, they highlight that although developing and deploying ML code is fast, it corresponds to a small component in

the whole project and its management is more entangled than traditional software applications. This is represented in Figure 2.1.1 extracted from the aforementioned paper.

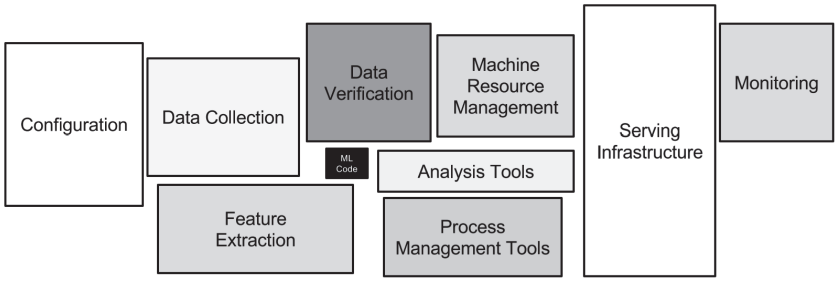

Figure 2.1.1: Role of ML code in an ML system

In this context, Machine Learning Operations (MLOps) appears as a discipline aiming at improving efficiency and automation in the management of ML projects from data collection to model deployment in production and its observability. It is a set of practices seeking to improve collaboration and communication between different teams involved in the Machine Learning lifecycle, providing automation, governance and agility throughout the process.

There is no standardized lifecycle for all ML projects. The way these practices and other procedures are adopted in each project can variate from one another. Nevertheless, they can generally be placed under the umbrella of MLOps which tend to include the following stages:

- **Business understanding.** In this stage, the goal is to clarify business needs for an ML project, ensure the expertise in the topic, establish priorities and requirements for the rest of the stages and consider risks.
- **Data preparation.** Data collection, data analysis, data transformation and feature engineering are practices involved in this stage.
- **Data modeling.** It mainly focus on model training, testing and validation. Techniques included in this phase are hyper-parameter tuning, neural architecture search (NAS), model selection or transfer learning.
- **Model serving.** In this stage, the candidate model resulting from the previous stage is made accessible to be consumed. Generally, DevOps practices are involved in this stage such as model versioning, canary deployments or A/B testing, as well as infrastructure concerns such as networking, access-control or

scalability.

- **Model monitoring.** It refers to monitoring model performance over time, commonly measured by model accuracy if the ground-truth labels are available at prediction time, or using techniques such as outlier and data drift detection.
- **Maintenance and decision-making.** This stage is dependent of the monitoring analysis obtained by its predecessor. It relates to decision making such as updating a model using active learning techniques, training a new model with fresh data or triggering other business-related actions.

This thesis focuses in model **serving** and **monitoring**, although practices concerning data preparation and model training are inevitably mentioned at some points.

2.2 ML model serving

Model serving is one of the main concerns of ML systems. As mentioned before, it involves the application of DevOps practices and IT operations processes with the objective of bringing ML models into production and making them accessible and available for making predictions on demand. Among other concepts, the following are commonly managed in this stage:

- **Scalability.** Deployments must scale on demand either horizontally or vertically depending on infrastructure limitations and platform constraints.
- **Availability.** Deployed models have to be available for consumption.
- **Reliability.** A certain degree of confidence in successful use of productionized models has to be ensured. Together with availability, reliability is commonly agreed beforehand by Service Level Agreements (SLAs).
- **Access control.** Authentication and authorization must be validated before model consumption.
- **Canary deployments.** Deploying a new model version gradually by forwarding a proportion of requests to the new model and validating its performance before completely replacing the old version.
- **A/B tests.** Serving multiple versions of a model and comparing their performance to select the most desirable one.

Faced with these needs, new software architectures have emerged to replace traditional software architectures where scalability typically referred to improving server resources (i.e scaling vertically) while flexibility and development agility was limited due to their monolithic (i.e centralized) design. One of the most promising ones is microservice architecture which aspires to more independence between different parts of a system, separating concerns in smaller deployable services and, therefore, improving scalability and availability. Additionally, microservice architecture considerably enhances DevOps processes, facilitating the development of independent functionalities, bugs fixing, versioning and faster releases into production. This type of architecture is exploited in cloud-native solutions (i.e container-based) which facilitates the management of these single-purpose services, making it very suitable for ML projects.

2.2.1 Cloud-native infrastructure

Scalability, computing speed and security concerns have certain relevance in the development of new infrastructure alternatives for adopting cloud-native solutions. For a better understanding of these alternatives, it is worth introducing the following technologies:

- **Containers.** Standard units of software that virtualize an application in a lightweight manner by packaging the application code and its dependencies, and running in an isolated way while sharing the host OS kernel. Container runtimes can be divided into low-level and high-level. Among the former are Linux Container (LXC)[1], lmctfy[2], runC[3] or rkt[4]. On the other hand, some high-level runtimes are dockerd[5], containerd[6], CRI-O[7] and, again, rkt.

- **Virtual Machine (VM)** VMs are virtual software computers running an Operating System (OS) and applications on a physical computer in an isolated manner as if it was a physical computer itself. Research on specialized lightweight VMs to run containers more efficiently than with traditional VMs has

[1]Linux Container (LXC): https://linuxcontainers.org/lxc/introduction/
[2]lmctfy: https://github.com/google/lmctfy
[3]runC: https://github.com/opencontainers/runc
[4]RKT: https://coreos.com/rkt/
[5]dockerd: https://docs.docker.com/engine/reference/commandline/dockerd/
[6]containerd: https://containerd.io/docs/
[7]CRI-O: https://cri-o.io/

led to the appearance of projects such as Kata Containers[8], used in OpenStack [62] or gVisor[9], used in Google Functions [35]. Also, Azure Functions [59] implements a similar approach.

- **Unikernels [57]** (aka library OS). They refer to specialized OS kernels acting as individual software components including the OS and application, in an unmodifiable, lighter, single-purpose piece of software. They can improve performance and security in the execution of containers and are used in serverless services such as Nabla Containers[10], used in IBM Cloud Functions [45].

- **Hypervisors** (aka Type I Virtual Machine Monitor (VMM)). They are software programs behind a Virtual Environment (VE) in charge of VMs management including VM creation, scheduling and resource management. Hypervisors have demonstrated to increase efficiency in Cloud Computing [51]. Some hypervisors are Kernel Virtual Machine (KVM)[11], Xen[12], VMWare[13] and Hyper-V[14]. Additionally, dedicated VMMs has been developed such as Firecracker [2] which is built on top of KVM for a better container support, backing services like AWS Lambda [6] or AWS Fargate.

The differences between containers, unikernels and VMs mainly fall on their virtualization type. While VMs run a whole guest OS, unikernels use lighter kernels and containers share the host OS. A representation of the different virtualization types is shown in the Figure 2.2.1, extracted from [79].

The use of container technologies for cloud solutions has been demonstrated a good alternative [78] and at least as good as traditional VMs in most of the cases [16, 24]. Also, the arrival of container management tools facilitates the convergence towards a universal deployment technology for cloud applications [11].

The wide variety of technologies, each with advantages and drawbacks, has led to the offering of multiple cloud services. These services can be classified into the following categories, depending on their abstraction level:

[8] Kata Containers: https://katacontainers.io/docs/
[9] gVisor: https://gvisor.dev/
[10] Nabla Containers: https://nabla-containers.github.io/
[11] KVM: https://www.linux-kvm.org/page/Main_Page
[12] Xen: https://xenproject.org/
[13] VMWare: https://docs.vmware.com/es/VMware-vSphere/index.html
[14] Hyper-V: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/mt169373(v=ws.11)

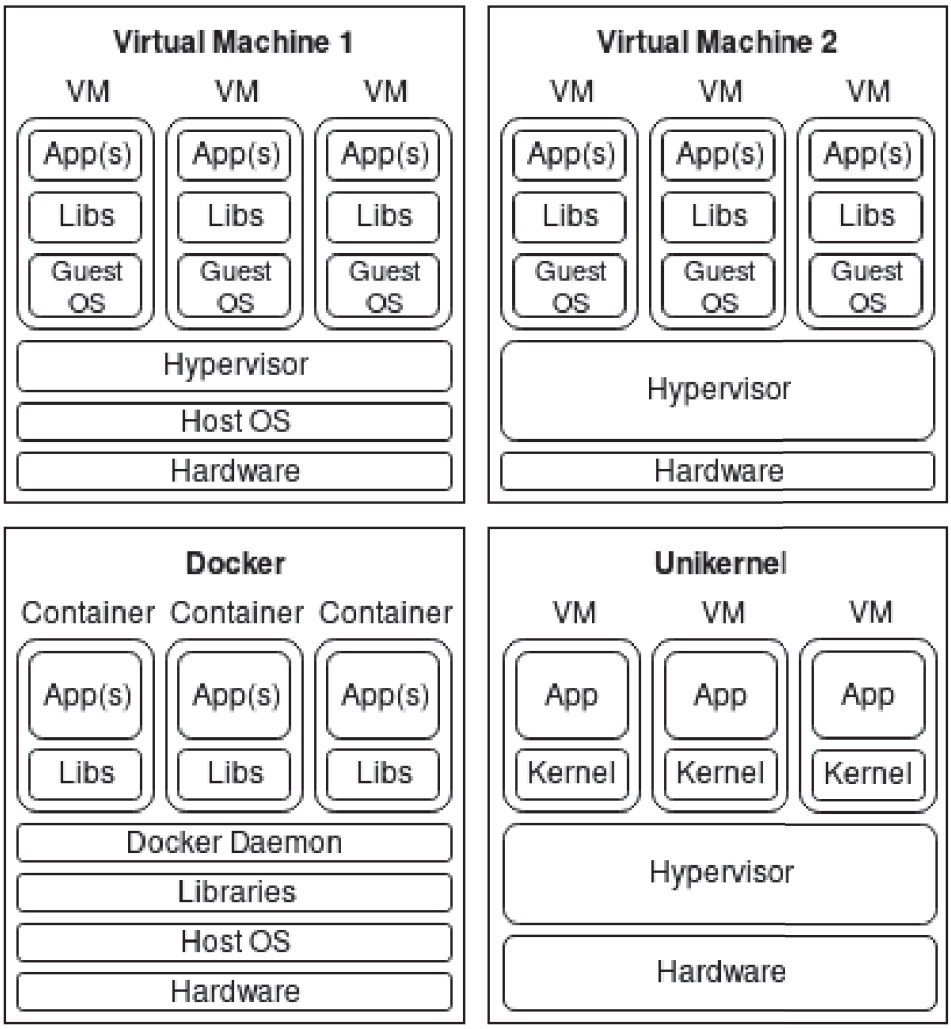

Figure 2.2.1: Types of virtualization

- **Infrastructure as a Service (IaaS)**. It refers to services where configuration and maintenance of networks, storage and other infrastructure resources is responsibility of the user, as well as OS updates or its applications. For instance, services such as AWS EC2, Google Compute Engine or Azure VMs allows the creation of clusters with multiple nodes but networks, storage, access control or maintenance need to be managed.

- **Platform as a Service (PaaS)**. It abstracts most of the complexity of building and maintaining the infrastructure, letting the user focus on developing applications. Concerns such as scalability and availability are managed to a large extend. AWS Elastic Kubernetes Service (EKS) [5], Azure Kubernetes Service (AKS) [60] or Google Kubernetes Engine (GKE) [36] are examples of this kind of services.

- **Software as a Service (SaaS)**. It comprehends feature rich applications hosted by third-party providers and accessible through the Internet. Every technical concern about maintenance, scalability or availability is managed by the provider. Examples of cloud-based ML applications are Amazon Sagemaker [4], Azure ML [61] or Google AI Platform [34].
- **Function as a Service (FaaS)**. Adding an extra layer of abstraction, these services also administrates applications, runtimes and other dependencies, allowing the user to focus on the application code. Examples of FaaS offerings are AWS Lambda [6], Azure Functions [59], Google Cloud Functions [35] and IBM Cloud Functions [45].

Server-less computing (aka Function-as-a-Service) is a relatively new paradigm that abstract server management from tenants. It is drawing the attention of more an more researchers who compare the implementations and efficiency between cloud providers [80], and attempt to find solutions [75] to overcome its main challenges [23, 74]. Also, recent research evaluates the use of server-less computing for ML model serving [15, 47, 83].

All these services offer alternatives for ML model serving with different levels of management, flexibility, performance and vendor dependency. Thus, it is worth analysing common decisive factors in ML productionizing.

ML model serving needs

As introduced in the previous section, there are multiple alternatives for model serving. For a long time, the most common approach for ML productionizing was using SaaS alternatives which provides support for end-to-end ML lifecycle management in the same application. However, the emergence of cloud-native ML platforms [39] or improved model serving tools suitable for server-less computing [10, 12, 55] facilitates model deployment using other approaches.

The decision about using server-less services (i.e FaaS), container-based solutions (i.e PaaS or IaaS) or feature rich software applications running in the cloud (i.e SaaS) depends mainly on pre-defined ML project guidelines, deployment requirements, desired flexibility and expected demand. Some of the main technical concerns to consider when serving ML models are:

- **Scalability:** The solution needs to scale dynamically on-demand by adjusting the number of instances (i.e horizontally) or upgrading them (i.e vertically).
- **Cold-starts:** At the moment of an inference request, the model and its dependencies can happen to be already loaded in memory (warm-start) or not (cold-start), implying an additional latency for loading these resources before inference.
- **Memory limits:** While the model size can vary from MBs to GBs depending on the problem type and training approach, it is not the only component to consider. Runtimes, configuration files or additional data might be needed for the correct functioning of the model, and all of them must fit in memory during inference.
- **Execution time limits:** Although model execution is commonly in the range of milliseconds, it can be considerably increased when loading additional external resources or performing transformations on the inference data.
- **Bundle size limits:** There are services where the size of the deployed bundle its limited, more commonly in FaaS solutions.
- **Billing system:** Depending on the type of service, different billing systems are applied addressing the trade-off between resource exploitation and application readiness distinctively. Hence, it also affects to other concepts such as cold-starts or scalability. For instance, FaaS are cheaper and more effective solutions for small sized models and unexpected high demand peaks while PaaS or IaaS are more suitable for big sized models and more constant compute-demanding workloads.
- **Portability:** Some of the alternatives for model serving, especially SaaS applications, might be tightly coupled to or better optimized for provider's infrastructure. This difficulties the portability and flexibility of the solution.

Furthermore, considering the principles of MLOps, smoothness and automation are also relevant terms to take into account for ML projects, as mentioned in Section 2.1. For example, IaaS alternatives, where server management is responsibility of the user, can slow the process of productionizing a new models.

Lastly, when the flexibility of IaaS solutions is not a requirement, these services generally involve too much complexity and PaaS alternatives are preferable. Also, the

advancements in container management tools [40] and cloud-native ML frameworks [38, 39] make PaaS solutions more appealing. They include features such as autoscaling or failure recovery, abstracting the bulk of cluster management to the user, and can be easily migrated to other clouds or an on-premise cluster.

2.2.2 Kubernetes: a container orchestrator

Given the advancements in container technologies and the advantages of cloud-native solutions as introduced in the previous section, the need of tools to build, deploy and manage containers became noticeable [64]. Given this need, new platforms known as *container orchestrators* were developed. Among the most widely adopted orchestrators are Kubernetes [40], Docker Swarm [21], Marathon, Helios or Nomad, being Kubernetes the most popular one and with the most potential.

Architecture

When running in the cloud, a Cloud Controller Manager (CCM) acts as the control plane for Kubernetes, separating the logic of the Kubernetes cluster from that of the cloud infrastructure. In the Figure 2.2.2, a Kubernetes cluster with a CCM and three nodes is represented.

Kubernetes was designed using declarative Application Programming Interfaces (APIs). This means that the creation, update or deletion of objects is purely declarative, and represents the desired states of the corresponding objects. Then, attempts to reconcile these states are made by the object controllers inside never-ending control loops. The main components of Kubernetes are:

- **Controller.** A controller backs an object API. It runs a control loop watching the state of the cluster using the API Server and applying changes to reconcile the desired states of the objects.
- **API Server.** It is used for the controllers to manage the state of the cluster. Also, it configures and validates the states for the objects.
- **Controller Manager.** It is a daemon that manages the control loops of the fundamental controllers.
- **Scheduler.** It is a policy-rich topology-aware component in charge of matching containers with nodes (i.e scheduling) by analyzing nodes capacity, availability

and performance.

- **kubelet.** It is an agent running on every node of the cluster, in charge of registering the node in the Kubernetes cluster and ensuring the execution and health of containers.
- **kube-proxy.** It is a network proxy running together with the agent on each node of the cluster, mirroring the Kubernetes API.

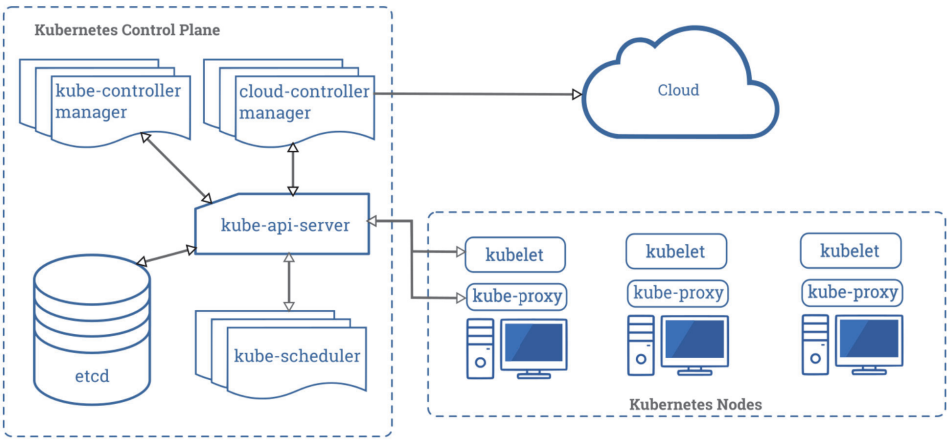

Figure 2.2.2: Kubernetes architecture with Cloud Controller Manager

Terminology

To better comprehend how Kubernetes works, it is worth describing some of the most commonly used concepts in Kubernetes terminology:

- **Custom Resource Definition (CRD).** It is an extension of the Kubernetes API, enabling the management of a custom type of object. A controller (i.e control loop) is generally implemented, backing the CRD and ensuring the reconciliation of the object states.
- **Pod.** It refers to the smallest deployable unit in Kubernetes, composed of one or more containers sharing network, storage and the desired state specification that led to its creation.
- **Service.** A service makes applications running in a pod accessible, abstracting the burden of network configuration required due to pod mortality.
- **ReplicaSet.** It is responsible of ensuring a stable set of pods running across time by creating and destroying them, given a desired state specification.

- **Deployment.** It manage dynamic creation and destruction of pods, given a specific policy, by creating, deleting or modifying replicaSets or other deployments.
- **StatefulSet.** It functions similarly than deployments, with the extra feature of establishing an identifier to every pod. This identifier makes pods not interchangeable, therefore maintaining their own state, and facilitates linking with specific volumes.

Dependencies

Additionally, Kubernetes relies on third-party software such as Istio [20], etcd [19] and cert-manager [48] for the configuration and management of storage, networks and certificates, respectively. Istio is a software implementation that facilitates connection, security, control and observation of services, via a service mesh, in a microservices architecture. As for etcd, it is a distributed key-value store for data accessed by distributed systems, reliable for storing state and configuration files. Lately, cert-manager is a certificate manager built on top of Kubernetes.

2.2.3 KFServing: an ML serving tool

When it comes to deploying ML models into a Kubernetes cluster, there are several alternatives available that differ on their terminology, scalability support, metrics collection or ML frameworks supported. They can be roughly classified into two categories: ML-specific and server-less functions. The former refers to alternatives designed with ML requirements in mind and, therefore, normally have better support for a wider range of ML frameworks. Among them are KFServing, Seldon Core or BentoML. The latter are implementations to run generic functions in a server-less approach such as OpenFaas or Kubeless.

Although model serving can be carried out with both alternatives, ML-specific implementations provide additional abstractions and dedicated features. Among these implementations, KFServing stands out for its potential, active community and wide catalog of features such as transformations or predictions explainability.

KFServing extends the Kubernetes API with objects of kind *InferenceService*. In order to serve an ML model, a new object needs to be created including required

information in its specification. This required information concerns the predictor component and includes the location where the model is stored and the model server used to run the model. Additionally, other parameters can be included to configure the rest of components. Figure 2.2.3, extracted from [37], shows the architecture of the KFServing data plane and how its components interconnect. These components are:

- **Transformer.** They can be used to apply transformations on the inference data before reaching the predictor, or processing the predictions before returning to the endpoint. They can be configured using out-of-the-box implementations or a given container.
- **Explainer.** They provide a way to compute explanations of the predictions using out-of-the-box frameworks or a given container.
- **Predictor.** They use the model server defined in the object specification to expose and run the model.
- **Endpoints.** Each endpoint expose a group of components including the three above. It supports strategies such as canary deployment, A/B testing or BlueGreen deployment. Moreover, it implements a logger component for inference logs forwarding as CloudEvents [27] using HTTP protocol.

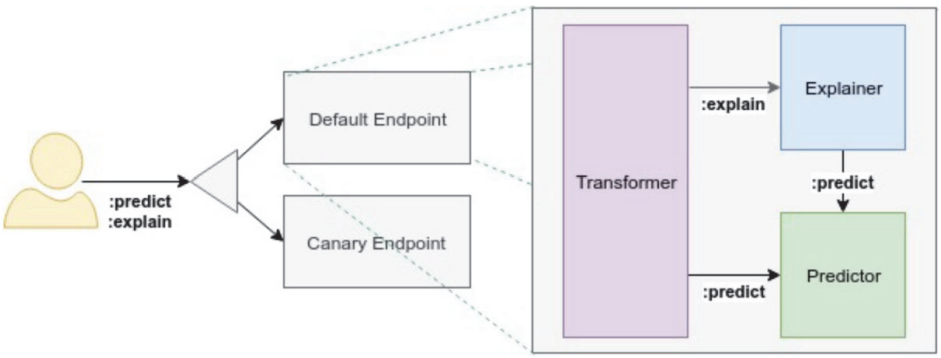

Figure 2.2.3: KFServing architecture

2.3 ML model monitoring

ML model monitoring is another relevant stage in the ML lifecycle, although it is frequently paid less attention than other stages. There are multiple factors that can affect the performance of the model and, therefore, the functioning of other parts of

the system dependent on its predictions. Examples of these factors are a change of data context (e.g culture, location or time), anomalous events produced by a malfunctioning device or inference requests with adversarial intentions.

Once an ML model is released into production, its continuous supervision is essential. This supervision can be conducted using different approaches. The main approaches for model monitoring correspond to outlier detection, drift detection, prediction explainability and adversarial attacks detection.

2.3.1 Outlier detection

Outlier detection (aka anomaly detection) is a kind of analysis aiming at detecting anomalous observations whose statistical characteristics differ from those of the data set as a whole. Numerous research [3, 18, 42, 43] have extensively analyzed their possible sources, types and approaches to detect them. In [18, 43] outliers are classified in three types based on their composition and relation to the dataset. Considering an additional one (i.e Type 0) for the simplest outlier detection approach, these types are:

- **Type 0:** It comprehends outliers detected by comparing the instance values with the statistics of a given baseline, such as maximum, minimum, expected value or standard deviation.
- **Type 1:** These outliers refer to individual instances that show different statistical characteristics than normal instances. Analysis for this type compare individual instances against the rest of the data without prior knowledge of the dataset.
- **Type 2:** Outliers of this type are context-specific Type 1 outliers. In other words, they are individual instances that might correspond to outliers in a specific context but not in general terms. This context is defined by the structure of the data.
- **Type 3:** These outliers are part of a subset of the data that without being individually considered outliers, the statistical characteristics of the subset are considered anomalous.

Furthermore, approaches to detect outliers have been classified in multiple ways. The most common ones are:

- **Distance-based:** (aka nearest neighbor based) This approach makes use of distance metrics to compute the distance between an individual instance with its neighbors. The greater the distances to its neighbors or the less number of neighbors closer than a predefined distance, the higher probabilities to be an outlier. In [42], these outliers are further classified into global or locals. While in the former all neighbors are considered, the latter considers a reduced number of neighbors, commonly applying Local Outlier Factor (LOF) algorithm.

- **Statistical-based:** Categorized into parametric and non-parametric, this approach consists of building a generative probabilistic model that captures the distribution of the data and detect outliers by analysing how well they fit the model.

- **Classification-based**: Divided into supervised and semi-supervised, this approach involves training a model with either normal instances or known outliers as labels, and using it to classify new instances. These outliers can have been detected previously or be manually generated. Among the most common techniques are Neural Networks (NN), Bayesian Networks, Support Vector Machines (SVMs) and rule-based methods.

- **Clustering-based:** This method consists of using cluster analysis techniques to differentiate groups of similar instances and detect outliers under the assumption that these do not belong to any group or belongs to a small one.

Additionally, some studies add several sub-approaches contained in one or more of the alternatives mentioned above. These include density-based, depth-based, set-based, model-based or graph-based.

2.3.2 Drift detection

The conditions in which a model is trained and in those which the same model is consumed are prone to change. Assuming that the data used for model training and inference is generated by the same function and follows the same distribution commonly leads to a poor model performance. This is due to the fact that environments are typically non-stationary and, therefore, also the distributions characterizing new generated data streams. In other words, the function that generates new data typically changes between environments.

Detecting change in data streams has been far studied in the last two decades [31, 44, 56, 66]. A data stream can be defined as a data set that is generated progressively and, hence, a timestamp is assigned to each item. The phenomenon of a data stream generator function changing over time is known as drift or shift.

More specifically, two terms can be differentiated: data drift and concept drift. While the former refers to change on the statistical properties of the input data used for either training or model inference, the latter relates to a change of the interpretation of the concept itself. In the literature, this differentiation is commonly omitted and concept drift is generally used. Therefore, to better understand the different sources and types of drift, it is worth defining what is a concept first.

Over time, a concept has been defined in various ways, being the most recent and lately used the one provided in [31] with a probabilistic approach: the joint distribution $P(X, Y)$, uniquely determined by the prior class probabilities $P(Y)$ and the class conditional probabilities $P(X|Y)$. In an unsupervised scenario (i.e there are no labels) and considering streaming environment, a concept can be defined as $P_t(X)$. Consequently, concept drift can be expressed as $P_{t1}(X) \neq P_{t2}(X)$. Given this definition, possible sources of concept drift [31, 44, 50, 56] are:

- **Covariate shift:** (aka virtual concept drift) When the non-class attributes $P(X)$ or class-conditional probabilities $P(X|Y)$ change but the posterior probabilities $P(Y|X)$ remain the same. Hence, the boundaries of the concept don't *really* change.
- **Prior probability shift:** (aka real concept drift, class drift or posterior probability shift) When the prior probabilities $P(Y)$ or posterior probabilities $P(Y|X)$ (i.e the concept) change. Hence, the boundaries of the concept change.
- **Class prior probabilities shift:** Suggested in [50], it refers to unique change in the prior probabilities $P(Y)$.
- **Sample selection bias:** When the sample data used to train the model does not accurately represent the population due to certain dependency on the labels (Y) by the generator function.
- **Imbalanced data:** It consists of concept misinterpretation when the classes of the training set have considerably different frequencies, making difficult to correctly predict infrequent classes.

- **Domain shift:** It refers to semantic change of concepts out of a specific domains (e.g culture, location or time period)
- **Source shift:** Directly related to the non-stationary sense of environments, it is produced by variation in the characteristics of the different data sources.

Additionally, concept drift can be classified in several ways depending on its characteristics. The most common ones refer to:

- **Concept boundaries:** As introduced before, concept drifts can be real or virtual, depending on the class boundaries.
- **Speed:** In terms of speed, concept drifts can be abrupt (aka sudden) or extended.
- **Transition:** Drift can happen gradually (i.e gradual progression towards the new concept) or incrementally (i.e steady progression).
- **Severity:** It refers to drift partially (i.e local) or completely (i.e global) affecting the instance space.
- **Recurrence:** Drift can reoccurred over time at different rates.
- **Complex drift:** It refers to more than one type of drift occurring simultaneously.

Once concept drift is defined together with possible sources and types, it is possible to classify different approaches and algorithm for its detection.

Algorithms classification

In recent research [56], an exhaustive classification of drift detection methods based on implemented test statistics has been carried out, dividing them into three categories:

- **Error rate-based:** These algorithms analyse the change in the error of base classifiers over time (i.e online error), declaring drift when it is statistically significant. Among them, the most popular are Drift Detection Method (DDM) [30], Early Drift Detection Method (EDDM) [8], Heoffding's inequality based Drift Detection Method (HDDM) [29] and ADaptive WINdowing (ADWIN) [13].
- **Data distribution-based:** These algorithms use distance metrics to measure dissimilarity between training and inference data distributions, declaring drift

when it is statistically significant. Among these algorithms are kdqTree and Relativized Discrepancy (RD), which use Kullback-Leibler (KL) divergence and Competence distance metrics, respectively.

- **Multiple hypothesis tests:** These algorithms apply analogous techniques than the other two categories, with the particularity that they use hypothesis tests either in parallel or in a hierarchical basis. Among the most popular ones are Just-In-Time (JIT) adaptive classifiers and Linear Four Rates (LFR).

Additionally, authors in [44] provide a classification based on the approach for overcoming drift, differentiating three categories:

- **Adaptive base learners:** These algorithms adapt to new upcoming data, relying on reducing or expanding the data used by the classifier, in order to re-learn contradictory concepts. They can be subdivided into three categories: decision tree-based, k-nearest neighbors (kNN) based and fuzzy ARTMAP based.

- **Learners which modify the training set:** Algorithms in this category modify the training set seen by the classifier, using either instance weighting or windowing techniques. Previous mentioned algorithms DDM, EDDM, HDDM and ADWIN are examples of window-based algorithms.

- **Ensemble techniques:** These algorithms use ensemble methods to learn new concepts or update their current knowledge, while efficiently dealing with re-ocurring concepts since historical data is not discarded. They can be subdivided into three categories: accuracy weighted, bagging and boosting based, and concept locality based.

Two main insights can be taken from these classifications: (1) the close relationship between drift detection and online learning, being some algorithms lately adjusted for that scenario (i.e ADWIN2 [13]); and (2) the majority of these algorithms use windowing techniques to partition the data stream.

As for ML model monitoring, it is important to consider that in productionized ML systems the labels are rarely available at near prediction-time. This means that error-based algorithms are not suitable for all scenarios.

Windowing and window size

Early proposed drift detection algorithms use fixed size windows over the data stream. This approach shows several limitations in the exploitation of the trade-off between adaptation speed and generalization. While smaller windows are more suitable for detecting sudden concept drift and offers a faster response due to a shorter duration of the window, larger windows are suited for detecting extended drift (i.e either gradual or incremental).

More recent studies focus on algorithms using dynamic size windows that offer more flexibility to detect different types of drift. The criteria for adjusting the window size can be trigger-based [8, 13, 30] (aka drift detection mechanisms) or using statistical hypothesis test [50]. In the former, the window size is re-adjusted when drift is detected, which is more suitable when information such as detection time or occurrence is relevant.

The motivation of using multiple adaptive windows for concept drift detection has been demonstrated an effective approach to deal with the complex and numerous types of drift prone to occur [13, 41, 52, 53]. The most challenging ones are extended drifts (i.e either gradual or incremental) with long duration and small transitive changes.

2.3.3 Explainability

Faced with the far studied problem of models seen as *black boxes*, specially concerning Deep Neural Network (DNN), the conceptualization of Explainable Artificial Intelligence (XAI) and the need to provide explanations of predictions have led to extensive research in the field [1, 9]. Some of the most popular approaches are LIME [67], Anchors [68] and Layer-wise Relevance Propagation (LRP) [7].

LIME is a model-agnostic interpretation method which provides local approximations of a complex model by fitting a linear model to samples from the neighborhood of the prediction being explained. After further research, the same authors presented Anchors [68], which instead of learning surrogate models to explain local predictions, it uses *anchors* or scoped IF-THEN rules that are easier to understand.

LRP [7] is a methodology that helps in the understanding of classifications by visualizing heatmaps of the contribution of single pixels to the predictions.

2.3.4 Adversarial attacks

Another concern in ML models in production is the robustness of the models. Recent research [14, 17, 63] has extensively analyzed the vulnerability of DNN against adversarial examples. These are special input data that affect model classification but remain imperceptible to the human eye. Adversarial attacks can be classified in three different scenarios [17]:

- **Evasion attack:** The goal of these attacks is inciting the model to make wrong predictions.
- **Poisoning attack:** They focus on contaminating the training data used to train the model.
- **Exploratory attack:** These attacks focus on gathering information about the model and training data.

From a model monitoring perspective, (1) evasion attacks might trigger the detection of outliers or concept drift, (2) poisoning attacks can affect model monitoring when the baseline used for outlier and drift detection has been contaminated; and (3) exploratory attacks can easily go unnoticed since they do not influence training data and might not represent neither outliers nor concept drift.

2.3.5 Related work

Several frameworks have been already implemented for outlier and drift detection, explainability and adversarial attacks detection with different approaches, platform support and programming language. Concerning the monitoring implementation in this thesis, there are two frameworks that are more similar to the work presented in Chapter 4, under different perspectives.

On the one hand, Alibi-detect [72] shares the same purpose, focusing on monitoring productionized ML models. Although it can be integrated in a streaming platform, its lack of native streaming platform support leads to integration procedures that can impact the design and performance of the system.

On the other hand, StreamDM [73] is the most similar in design. It is built on top of Spark Streaming and implements multiple data mining algorithms in a streaming fashion. It focuses on data mining instead of model monitoring and lacks support for

Spark Structured Streaming.

Lastly, KFServing [38] is recently adding support for some features of Alibi-detect including outlier detection and prediction explainability algorithms. Due to the inherited decentralized architecture of KFServing, these algorithms run on isolated sub-streams of the whole inference data stream. This presents additional concerns about the accuracy of these algorithms, specially drift detectors, for a couple of reasons: (1) gaps of temporal data are prone to appear among the sub-streams and (2) comparing non-consecutive subsets of a data difficulties the detection of outliers and drift, specially for algorithms using windowing techniques.

Chapter 3

Methodologies

After introducing necessary definitions and concepts for a better comprehension of this thesis, the methodologies conducted throughout this work are presented in this chapter. First, the architecture decisions needed beforehand are justified together with the streaming platform used for performing monitoring-related computations. The next two chapters describe how the architecture design is evaluated including the inference data generated for the experimentation as well as how performance metrics and inference analysis are collected. Lastly, information about the open-source character of the implementations in this work close this chapter.

3.1 Architecture decisions

As described in Section 2.2, there are multiple alternatives for ML model serving. Due to the burden of infrastructure management in IaaS solutions, this option adds complexity in terms of automation, which is one of the target characteristics of the architecture evaluated in this work. As for SaaS applications, they are typically used for the whole ML lifecycle. Although they abstract all the infrastructure-related concerns, they lack flexibility and portability, and do not commonly provide model monitoring in a streaming fashion, hence conflicting with the thesis goals.

Regarding PaaS platforms, they are feasible alternatives given the advancements in container technologies and the maturity of existent container orchestrators which facilitate the automation of infrastructure management tasks. Something similar happens with server-less services (i.e FaaS). A big effort is being made to productionize

ML models using these kind of services. However, when it comes to model monitoring with a streaming approach, server-less services are not suitable for several reasons. They are not suitable for continuous jobs, lack windowing techniques support and its portability is also limited since they tend to require additional services for adding functionalities as the complexity of the system increase (i.e load balancers or API gateways).

Thus, the architecture proposal in this work is implemented on top of a container-based PaaS platform, leveraging its flexibility and the use of available automated tools such as container orchestrators.

As for the container orchestrator, Kubernetes is the one selected for this thesis. This decision is due to its wide catalog of features, extensibility support, active community, potential and great support among cloud providers.

3.2 Model monitoring on streaming platforms

Model inference is intrinsically related to data streams since the inference logs are produced progressively and have a timestamp assigned to them. As mention in the Section 2.3.2, the use of windowing techniques for data stream analysis has been demonstrated an effective approach. For instance, it benefits concept drift detection for dealing with the complex and numerous types of drift prone to occur.

Therefore, streaming platforms are perfectly suitable tools for model monitoring since one of their most common features is windowing support. Additionally, they typically provide other important features such as fault-tolerance, delivery guarantee, state management or load balancing. The number of available streaming processing platforms is multiple, each with advantages and disadvantages due to their processing architecture. Multiple research studies have conducted exhaustive comparisons between their weaknesses and strengths [46, 49, 58]. The most popular ones are Spark Streaming, Flink, Kafka Streams, Storm and Samza, being the first two the most feature rich.

The streaming-related implementations in this work are built on top of Spark Structured Streaming. Apart from its higher popularity and adoption for ML workflows, although Flink shows lower ingestion time with high volumes of data, Spark tends to have lower total computation latencies with complex workflows [58]. Since

one of the purposes of this thesis is provide support of different algorithms for model monitoring, it is assumed that their complexity can widely variate.

3.3 ML lifecycle

In order to test the scalability of the presented solution, a ML model is served using KFServing (described in Section 2.2.3) and used for making predictions over a generated data stream containing outliers and data drift.

Being the evaluation of the architecture design the main purpose in this thesis, the problem to solve via a ML algorithm is kept simple by addressing the Iris species classification problem. This problem refers to the classification of the species of flowers based on the length and width of the sepals and petals.

The different stages of the ML lifecycle has been carried out in Hopsworks[1], an open-source platform for developing and operating ML models at scale, and leveraging the Hopsworks Feature Store for feature management and storage. Additionally, the framework used for model training is Tensorflow.

3.3.1 Training set

The training [2] and test sets [3] used for model training are provided by Tensorflow. The data contains four features and the label.

- **Petal length:** Continuous variable representing the length of the petal in centimeters.
- **Petal width:** Continuous variable representing the width of the petal in centimeters.
- **Sepal length:** Continuous variable representing the length of the sepal in centimeters.
- **Sepal width:** Continuous variable representing the width of the sepal in centimeters.

[1]Hopsworks: Data-Intensive AI platform with a Feature Store: https://github.com/logicalclocks/hopsworks

[2]Training set can be found here: http://download.tensorflow.org/data/iris_training.csv

[3]Test set can be found here: http://download.tensorflow.org/data/iris_test.csv

- **Species:** Categorical variable indicating the specie of the flower. Possible species are: Setosa, Versicolour and Virginica.

The training dataset statistics are presented in Table 3.3.1, as they were computed automatically by the feature store. Also, the Figure 3.3.1 shows the distributions of the features.

Table 3.3.1: Training set descriptive statistics.

Feature	Count	Avg	Stddev	Min	Max
species	120.0	1.0	0.84016806	0.0	2.0
petal_width	120.0	1.1966667	0.7820393	0.1	2.5
petal_length	120.0	3.7391667	1.8221004	1.0	6.9
sepal_width	120.0	3.065	0.42715594	2.0	4.4
sepal_length	120.0	5.845	0.86857843	4.4	7.9

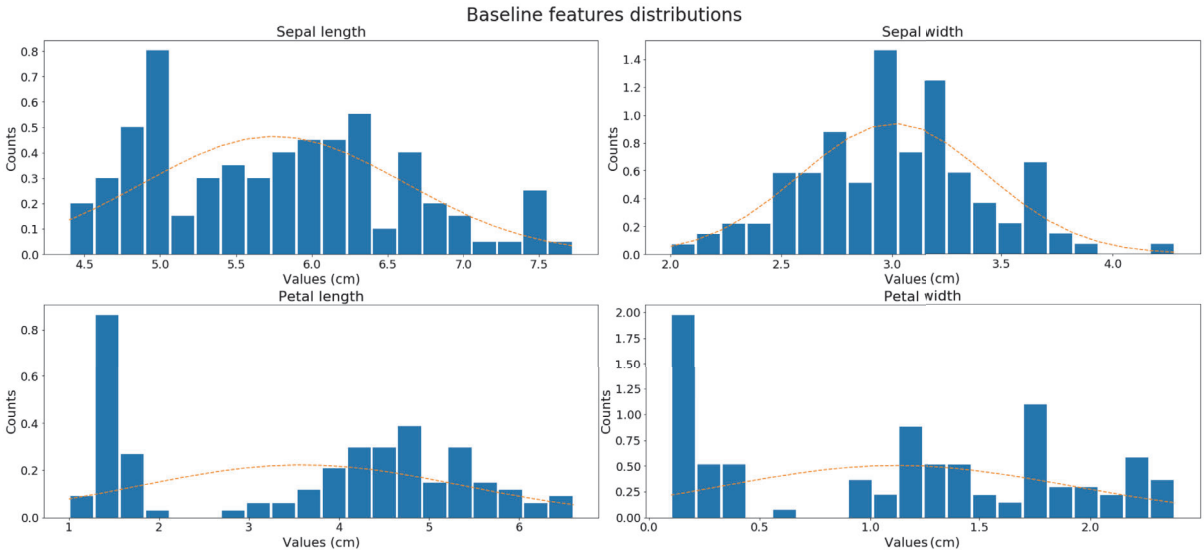

Figure 3.3.1: Training set features distributions.

3.3.2 Model training

The algorithm trained to solve the classification problem is a Neural Network (NN) composed of two hidden layers of 256 and 128 neurons, respectively, and a fully connected layer. Since the label is a categorical variable provided in string format, one hot encoding is applied to turn it into binary variables. The weights for the layers are randomly initialized following a normal distribution.

The algorithm is trained by running experiments with different hyper-parameter settings to maximize the accuracy (i.e target metric), using Grid Search hyper-parameter tuning. These settings come from the combination of three parameter: (1) a learning rate of 0.01, 0.001 and 0.02, (2) a batch size of 32, 64 and 128, and (3) 50, 75 and 100 epochs. After running the experiments, the selected model achieved an accuracy of 82%. Even though more accurate models can be trained to solve the Iris classification problem, this is out of the scope of this work.

3.3.3 Model inference

The inference data stream used for making predictions and then comparing the resulting inference analysis is generated based on the training set statistics, but slightly modified to contain outliers and concept drift. This data stream is generated with Numpy [4] using the baseline mean and standard deviation. After that, requests are sent at different rates and concurrency level using a custom adaptation of Hey [5], an http load generator, avoiding repetition of instances.

3.4 Metrics collection

During model inference, the proposed architecture is expected to auto-scale. In order to measure this behavior, metrics are scraped from different sources using Prometheus, an open-source monitoring solution, and visualized with Grafana, a metrics analytics and visualization tool. These sources include Kubernetes metrics-server, Knative serving and Kafka.

Kubernetes metrics-server provides information of the whole cluster from a Kubernetes perspective. This means that it provides metrics related to deployments, statefulsets, pods or cluster nodes. For instance, Spark driver and executors metrics are collected from a pod perspective, including memory, CPU and network consumption. As for Knative Serving and Kafka, they provide metrics related to their respective contexts such as the response times or message ingestion rates.

Additionally, the inference analysis also provides relevant metrics apart from those regarding statistics, outliers or concept drift detection. These metrics contains the

[4]Numpy, a package for scientific computing with Python: https://numpy.org/
[5]Hey. A tiny program that sends load to a web application. https://github.com/rakyll/hey

number of instances per window, delays between request containing outliers and its detection, or delays in the computation of statistics or drift analysis.

3.5 Open-source project

All implementations in this work have been developed under the GNU Affero General Public License v3.0 [6]. GNU AGPLv3 is a strong copyleft license whose permissions are conditioned on making available complete source code of licensed works and modifications, which include larger works using a licensed work, under the same license. Copyright and license notices must be preserved. Contributors provide an express grant of patent rights. When a modified version is used to provide a service over a network, the complete source code of the modified version must be made available.

These implementations can be found at `https://github.com/javierdlrm`.

[6] A detailed description of GNU AGPLv3 can be found at https://choosealicense.com/licenses/agpl-3.0/

Chapter 4

Architecture and implementations

In this chapter, a detailed description of the architecture proposal and corresponding implementations is provided. First, an overview of the solution is presented describing the main components and how they interact with each other. Subsequently, the next two sections explain the design and implementation details of the components developed throughout this work. Lastly, a section dedicated to metrics collection conclude this chapter enumerating the different sources and metrics available.

4.1 Overview

Since the architecture proposed to answer the research question is cloud-native, it can run on any cluster as long as it has the required third-party software previously installed. Therefore, the only requirements are those established by them.

The Figure 4.1.1 shows a representation of the whole solution for one ML model deployment, including the main components and their interactions. These are either implemented or third-party software.

On the one hand, among the components specifically created for this work (i.e green elements) there are three main components:

- **Model Monitoring Operator:** A Kubernetes controller in charge of managing and configuring the rest of the components in the workflow. It extends the Kubernetes API via CRDs, providing a way to define the specification for the deployment.

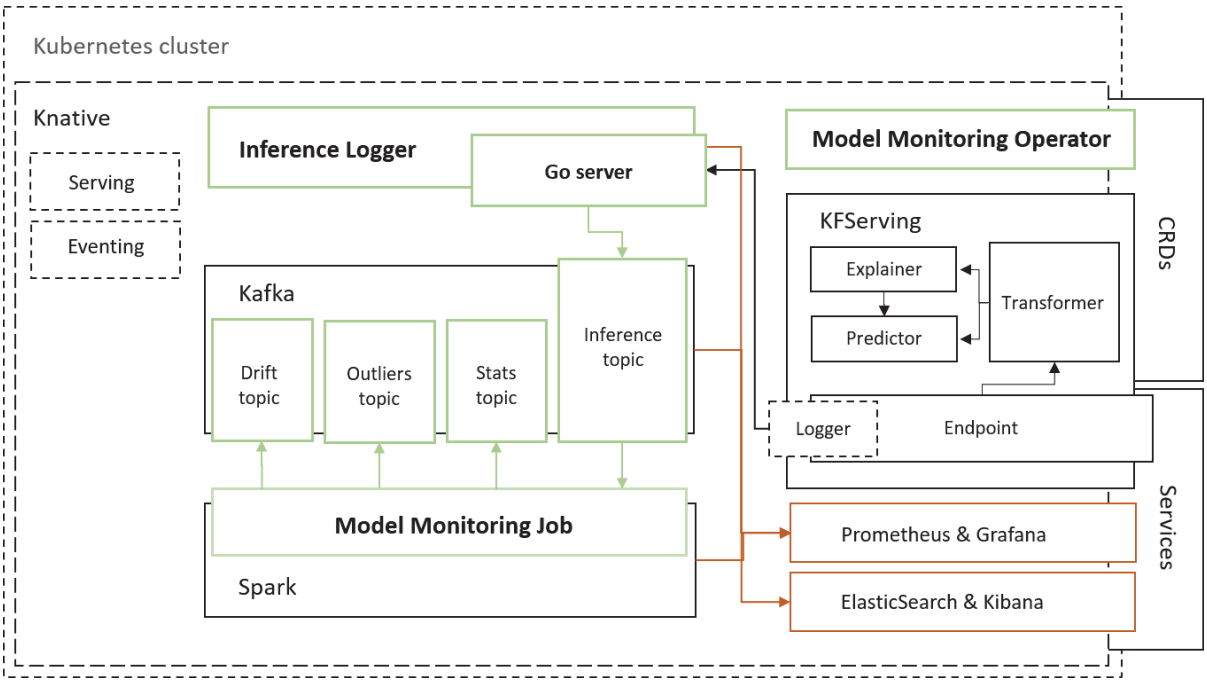

Figure 4.1.1: Architecture proposal. Green figures represents implemented components. Orange figures represent metrics-related components. The rest of the components (i.e third-party software and dependencies) are coloured black.

- **Model Monitoring Job:** A generic implementation of the Model Monitoring framework (see Section 4.2) running in a Spark Streaming job. It is configured via environment variables by the Model Monitoring Operator (described in Section 4.3).
- **Inference Logger (http server):** A lightweight server in Golang deployed using Knative service that expose an endpoint for inference logs ingestion, transform those logs into Kafka messages and send them to the corresponding Kafka topic.

On the other hand, among the third-party software used in the solution (i.e black elements) the main components are:

- **Kubernetes:** A container orchestrator in charge of the creation, scheduling, allocation and replication of the containers constituting the solution. It is extensively described in Section 2.2.2.
- **KFServing:** An ML serving tool built on top of Kubernetes for server-less model inferencing. It includes a logger for inference logs forwarding.
- **Knative:** A platform built on top of Kubernetes to manage server-less workflows. It is used to deploy the Inference Logger service. Moreover, it is used

by KFServing to serve ML models.

- **Kafka:** A distributed streaming platform mainly used for messaging in distributed systems [25]. Either inference logs and monitoring analysis (i.e statistics, outliers and drift detections) are stored in Kafka topics.
- **Spark:** A distributed cluster-computing framework for large-scale data processing [26]. It is used to compute analytics over the inference data in a streaming fashion, using the Model Monitoring framework.

Moreover, there are two channels for interacting with the system: Services and Custom Resource Definition (CRD). Services expose applications inside the cluster to be accessible from the outside. In this case, these are the model inference and metrics services. CRDs, as introduced in Section 2.2.2, are extensions of the Kubernetes API that provides a place for storing and managing CRD objects. A CRD object constitutes a specification of a particular *kind*. In this scenario, CRDs are used for specifying objects of the kinds *InferenceService* and *ModelMonitor*, enabled by KFServing and Model Monitoring Operator respectively. As the names suggest, these objects define the inference service and model monitoring system to be deployed.

Ultimately, in order to gather metrics from the system, two optional components can be installed in the cluster. These components do not depend on each other and, therefore, can be installed independently. One is Prometheus, an open-source systems monitoring and alerting toolkit [28], and the other one is ElasticSearch, a distributed, open source search and analytics engine for all types of data [22] built on top of Apache Lucene. The former is used to collect performance-related metrics of the whole system (see Section 4.4) such as latencies, number of instances or resource use. The latter can be used for indexing traces related to the behavior of the system.

4.1.1 Deployment process

In order to deploy the model monitoring system for a served model, and considering that Model Monitoring Operator is already installed in the cluster, two steps are required: (1) configuring inference logs forwarding to the endpoint expose by the Inference Logger and (2) creating a *ModelMonitor* specification (see Section 4.3). A representation of the steps involved in the deployment is provided in Figure 4.1.2.

Once a *ModelMonitor* object is created, the Model Monitoring Operator deploys and

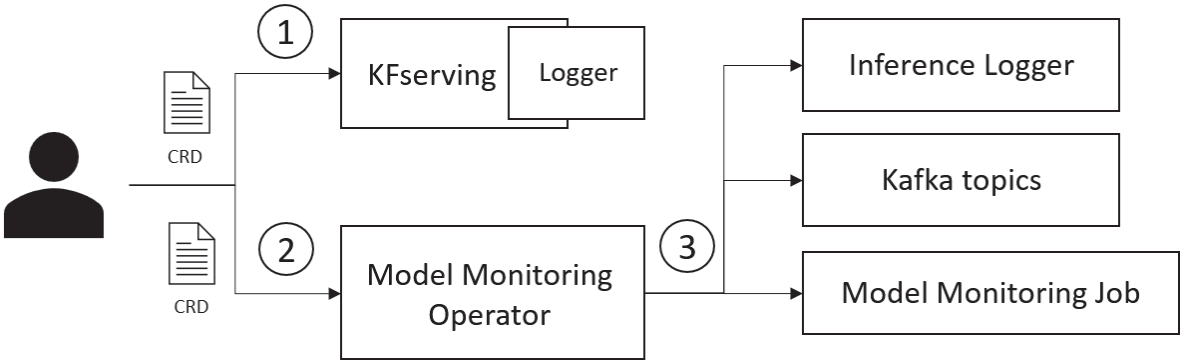

Figure 4.1.2: Steps involved in the deployment of the model monitoring system. (1) Inference logs are forwarded to the Inference Logger. (2) A ModelMonitor object is created. (3) The Model Monitoring Operator deploys the necessary components using the specification provided in the step 2.

configures the necessary components for monitoring the model (3). As explained in Section 4.3, a *ModelMonitor* object contains the specification for the deployment. It includes information such as the inference schema and algorithms used for outlier and drift detection.

Because of the cloud-native design of Kubernetes, the installation of the Model Monitoring Operator as well as the deployment of the different components consists of creating and managing containers, roughly speaking. Therefore, three docker images have been built for this work with names: model-monitoring-job, model-monitoring-operator and inference-logger, available at `https://hub.docker.com/u/javierdlrm`.

4.2 Model Monitoring framework

Model Monitoring framework is an open-source software implementation for inference data monitoring built on top of Spark Streaming [26]. More specifically, it has been developed using Scala 2.11 and the version 2.4.5 of Spark Structured Streaming.

The framework has been built with ML monitoring as the main purpose. It includes a core set of classes for designing pipelines (i.e data flows) with a *fluent* pattern [82]. These pipelines compute statistics and perform outliers and concept drift detection over the incoming data streams (i.e inference data). The framework leverages windowing techniques to partition the input data streams. Additionally, the core of the

framework is implemented using traits (i.e interfaces) to add a level of abstraction over the definition of statistics, outliers and drift algorithms. These traits can be extended with custom implementations, which also facilitates the extension of the framework with new algorithms in future work.

4.2.1 Terminology

For a better understanding of how the frameworks works, the definition of its core concepts is convenient. These concepts are:

- **Pipes:** They are scoped data processing stages that can be concatenated one after the other. The framework includes pipes for windowing (i.e WindowPipe), data source and sink definition (i.e SourcePipe and SinkPipe), and the computation of statistics, outliers and drift detection (i.e StatsPipe, StatsOutliersPipe or StatsDriftPipe).
- **PipeJoints:** They provide interfaces for joining two pipes. Each pipe defines a pipe joint that needs to be satisfied (i.e inherited) by the preceding pipe.
- **Pipelines:** They are groups of concatenated pipes that represent complete data flows, from a data source to one or multiple sinks. They can be thought as the equivalent to Spark Streaming queries.
- **PipelineManager:** It manages and supervises the execution of pipelines.
- **StatDefinition:** A trait for defining statistics including name, an aggregator and dependencies with other statistics (e.g average depends on the sum and count statistics).
- **StatAggregator:** A trait for implementing a statistic aggregator. There are three types of aggregators executed at different iterations: StatSimpleAggregator, StatCompoundAggregator and StatMultipleAggregator. For instance, the sum of instances (i.e simple) is used for computing the average (i.e compound) which is required for computing correlation (i.e multiple).
- **OutliersDetector:** A trait for defining outliers detection algorithms applied directly to instances values, without windowing.
- **WindowOutliersDetector:** A trait for defining non-statistical-based outliers detection algorithms executed directly over windowed instance values.

- **StatsOutliersDetector:** A trait for defining statistical-based outliers detection algorithms (see Section 2.3.1). Statistics are computed on windowed instance values.
- **DriftDetector:** A trait for defining drift detection algorithms applied directly to instance values, without windowing.
- **WindowDriftDetector:** A trait for defining non-statistical-based drift detection algorithms executed directly over the windowed instance values.
- **StatsDriftDetector:** A trait for defining statistical-based drift detection algorithms such as distribution-based ones (see Section 2.3.2). Statistics are computed on windowed instance values.
- **StatValue:** A special data type in the Model Monitoring framework that represents either a double value or a map of string-double key-value pairs. It is mainly used for strong typing during data frame row generation.
- **Baseline:** It defines the baseline data to be used for outlier and drift detection algorithms, including descriptive statistics and the distributions of the features.

4.2.2 Dataflow design

As mentioned before, the Model Monitoring framework allows the design of pipelines for inference data processing with a *fluent* design [82]. These pipelines are built by concatenating pipes one after the other. The Figure 4.2.1 shows the different combinations available.

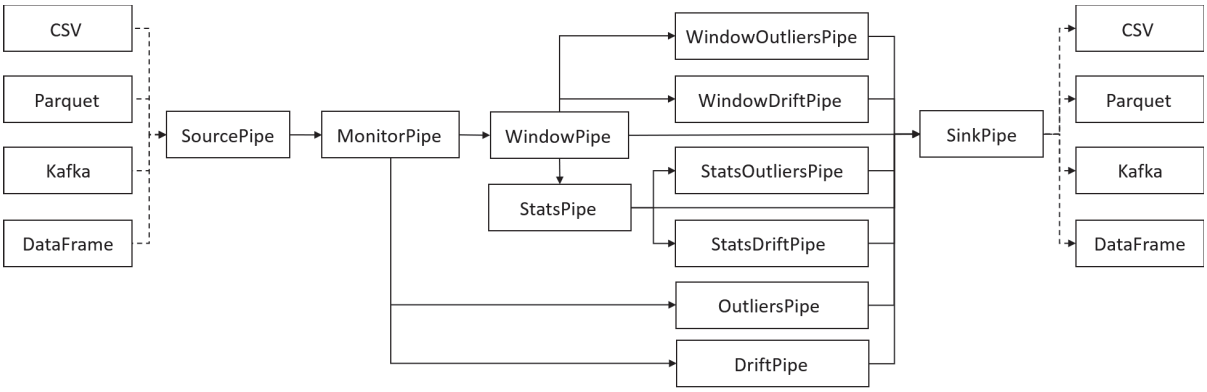

Figure 4.2.1: Pipe combinations for designing data processing pipelines.

The available stream sources and sinks correspond to the ones provided by Spark Structured Streaming. The starting point of every pipeline is a source pipe. After

reading data from the source, the next pipe is a monitor pipe where custom schemas can be applied over the data. Subsequently, either outliers/drift detection can be conducted over the instances or a window pipe can be joint which partitions the data given a predefined time-based window configuration including duration, slide duration and watermark delay. Moreover, a window pipe can be appended with a statistics pipe or the corresponding pipes for outliers and drift algorithms computed directly over the windowed instance values. In case of the statistics pipe, it can be extended by outliers and drift pipes which are computed over the windowed statistics.

As observed in the figure, all the pipes can be forwarded to a sink. A pipeline is considered as a data flow from source to sink. Therefore, a new pipeline is defined each time a sink is appended, but shared pipes are not duplicated. This allows to forward results to different pipes and sinks at the same time. For instance, statistics can be stored in parquet files while concept drift detections over these statistics are sent to a Kafka topic just by adding the corresponding sinks to the statistics and drift pipes.

4.2.3 Statistics, Outliers and Concept drift

As already mentioned, the framework provides traits for implementing statistics, outlier and drift detection computations. These traits are: (1) StatDefinition and StatAggregator for statistics, (2) StatsOutliersDetector and WindowOutliersDetector for outliers, and (3) StatsDriftDetector and WindowDriftDetector for concept drift. These traits are executed in the pipes with the same name.

The Model Monitoring framework already contains several implementations. Regarding statistics, the framework includes: maximum, minimum, count, average, mean, standard deviation, correlation, co-variance, percentiles and feature distributions (i.e histograms). The Table A.0.3 shows optional parameters for these statistics. For instance, specific percentiles or distribution bounds can be indicated.

Concerning outlier detectors, the framework includes an implementation for detecting distance-based outliers given baseline statistics. Instance values more than three standard deviations from the mean are assumed to be anomalous.

Respecting data drift detectors, three distribution-based approaches are included in

the framework. These are Earth Mover's Distance (EMD) [69], KL divergence [76] and Jensen-Shannon (JS) divergence.

- **EMD** (aka Wasserstein distance): A measure of dissimilarity between two multi-dimensional probability distributions in a feature space. It is defined by the following equation.

$$EMD(P,Q) = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} f_{i,j}d_{i,j}}{\sum_{i=1}^{m}\sum_{j=1}^{n} f_{i,j}}$$

- **KL divergence:** An asymmetric measure of distance from a probability distribution to a second one. It is defined by equation below.

$$D_{KL}(P||Q) = \sum_{x \in X} P(x) log(\frac{P(x)}{Q(x)})$$

- **JS divergence:** A symmetric measure of distance between two probability distributions, based on Kullback-Leibler divergence. It is defined by the following formula.

$$D_{JS}(P||Q) = \frac{1}{2}D_{KL}(P||M) + \frac{1}{2}D_{KL}(Q||M)$$

  where $M = \frac{1}{2}(P+Q)$

4.2.4 Model Monitoring Job

The Model Monitoring job is a Spark Streaming job using a generic implementation of the Model Monitoring framework that receives the required configuration via environment variables. This configuration includes the specification of statistics, outliers and drift to compute as well as the sources and sinks for each of the pipelines. The environment variables store json string values with the corresponding fields. A representative summary of these variables and their first-level fields is shown in the Table 4.2.1.

A detailed description of all configuration fields is included in Appendix A.

Table 4.2.1: Configuration for the Model Monitoring job. Each environment variable contains a json string with the corresponding fields.

Env var	Fields	Description	Required
MODEL_INFO	name	Model name	X
	id	Model id	
	version	Model version	
MONITORING_CONFIG	trigger	Trigger configuration	X
	stats	Statistics definitions	X
	outliers	Outliers detectors	
	drift	Drift detectors	
	baseline	Baseline data	
STORAGE_CONFIG	inference	Data source specification	X
	analysis	Sinks specification	X
JOB_CONFIG	timeout	Job timeout	

4.3 Model Monitoring Operator

The Model Monitoring Operator is a Kubernetes controller developed in Golang that backs the objects API for the *ModelMonitor* kind. More specifically, it is developed in Golang 1.13 and tested on Kubernetes 1.16. Just like every Kubernetes operator, the installation consists of applying several object specifications of different kinds: namespaces, services, service accounts, roles and CRDs, together with the container running the controller. A yaml file containing all these specifications is available in the code project.

The operator constantly watches the state of three kind of objects in the system: *ModelMonitor*, Spark Applications (i.e Model Monitoring Job) and Knative Services (i.e Inference Logger). When a *ModelMonitor* is created, changed or deleted, the operator proceeds with the deployment or deletion of the corresponding Inference Logger (described in Section 4.3.2) and Model Monitoring Job. As for the observation of the last two objects, it is mainly a supervision task. If any of these objects is deleted or modified by external reasons, the operator ensures the components are created and properly configured again.

4.3.1 ModelMonitor object

Just like with any object, those of kind *ModelMonitor* can be defined using yaml files. Each *ModelMonitor* includes the specification for monitoring one ML model. This specification combines Kubernetes-related settings (e.g minimum number of replicas or resource allocation) and general configuration for the system (e.g Kafka topics or inference schemas).

The main structure of a *ModelMonitor* yaml file is shown in Figure 4.3.1, including top-level fields. The field *spec* (line 6) accepts the same configuration structure as the Model Monitoring Job (see Section 4.2.4), simply changing the data format from json to yaml. Moreover, the specification accepts additional parameters such as a field for inference logger-related configuration (line 11) (see Section 4.3.2) and extra fields for the Spark job configuration (line 10) (see Section 4.3.3).

```
1  apiVersion: monitoring.hops.io/v1beta1
2  kind: ModelMonitor
3  metadata:
4    name: <object-name>
5    namespace: <object-namespace>
6  spec:
7    model: <model-info>
8    monitoring: <monitoring-config>
9    storage: <storage-config>
10   job: <job-config>
11   inferencelogger: <inferencelogger-config>
```

Figure 4.3.1: Structure of a ModelMonitor object specification.

4.3.2 Inference logger

Inference Logger is a Knative service running a lightweight server implemented in Golang 1.13. It uses the Go SDK for CloudEvents [27] to create an HTTP client that listens to `http://<service-name>.<namespace>.default:8080` for receiving inference logs in CloudEvent format. After handling an event, it transforms it into a Kafka message and forwards it to the corresponding Kafka topic specified in its configuration. For that purpose, it employs the Sarama library for Kafka [77]. In case a Kafka topic does not exists, it is automatically created.

The configuration parameters for this service are shown in Table 4.3.1.

Table 4.3.1: Inference Logger service configuration. A field in the ModelMonitor object specification.

Field	Description	Type	Required
autoscaler	Autoscaler to use: kpa.autoscaling.knative.dev or hpa.autoscaling.knative.dev	String	
metric	Autoscaler metric: concurrency, rps or cpu	String	
window	Autoscaling interval	String	
panicWindow	Panic window size	Int	
panicThreshold	Panic threshold	Int	
minReplicas	Minimum replicas	Int	
maxReplicas	Maximum replicas	Int	
target	Autoscaling target	Int	
targetUtilization	Autoscaling target utilization	Int	

4.3.3 Spark Streaming job

As mentioned before, the Model Monitoring Job is a Spark Streaming job with a generic implementation of the Model Monitoring framework. The configuration related to the framework is already described in Section 4.2.4. As for the spark job specific settings, they are collected in Table 4.3.2.

Table 4.3.2: Spark job configuration. It is merged with JOB_CONFIG settings in the ModelMonitor object specification.

Field	Description	Type	Required
exposeMetrics	Expose job metrics	Boolean	
driver	Driver configuration	Object	
driver.cores	Number of cores	Int	
driver.coreLimit	CPU resources	String	
driver.memory	Memory resources	String	
executor	Executor configuration	Object	
executor.cores	Number of cores	Int	
executor.coreLimit	CPU resources	String	
executor.memory	Memory resources	String	
executor.instances	Number of instances	Int	

4.4 Observability

Regarding observability, the installation of Prometheus [28] in the Kubernetes cluster is required. Once it is installed, all the available metrics are automatically collected except for the Spark Streaming job. In order to gather spark metrics the parameter *exposeMetrics* needs to be set in the *ModelMonitor* object specification (as shown in Table 4.3.2).

The available sources are Kubernetes metrics-server, Knative Serving and Kafka. The former provides information of the whole cluster from a Kubernetes perspective. This means that it provides metrics regarding running Kubernetes objects such existing deployments, statefulsets or pods. For instance, the resource consumption of the Spark driver or executors is available but from a pod perspective.

Regarding Knative Serving and Kafka, they provide metrics related to their respective contexts such as latencies, throughput or message byte rate..

Chapter 5

Experimentation

In this section, the experimentation conducted to test the scalability and performance of the proposed architecture is described in detail. The different phases of the experimentation, as well as data collection and visualization, are performed using a Jupyter Notebook to provide automation and flexibility to the process.

5.1 Cluster configuration

As mentioned before, a Kubernetes cluster has been deployed using AWS EKS in the Amazon Cloud. It is composed of 7 nodes with the same instance specification: t3.medium. This specification corresponds to (1) 2 virtual CPUs, (2) 4 GiB of memory, (3) up to 5 Gbps of network performance and (4) 20 GB of disk backed by AWS Elastic Block Storage (EBS) for each node.

The following software has been previously installed on the cluster:

- **Model Monitoring Operator:** The Kubernetes operator implemented in this work, detailed in Section 4.3. It is used for managing the different components for monitoring by creating a *ModelMonitor* object whose specification can be found in the next section.
- **KFserving:** As introducted in Section 2.2.3, it is a tool for ML model serving on top of Kubernetes. The pre-trained model for the experimentation has been served using this tool by creating an object of kind *InferenceService*. More details about this specification are included in Section 5.3.

- **Knative:** A platform built on top of Kubernetes to manage server-less workflows. It is used for both KFServing and Model Monitoring Operator to create server-less services.
- **Istio:** A software implementation that provides a service mesh for microservices architectures, facilitating connection, security, control and observation of services. It is a dependency of KFServing.
- **Kafka:** A Kafka cluster is configured using Strimzi operator. It is composed of three Zookeper replicas (i.e quorum size of three) and five brokers with 6 GiB of storage backed by AWS Elastic Block Storage (EBS) per each. Two of these brokers are used by the Inference Logger for inference logs ingestion and the remaining three are used for the Model Monitoring Job, one per topic.
- **Spark:** It is deployed using spark-on-k8s-operator by Google. The configuration of drivers and executors is specified in the *ModelMonitor* object.
- **Prometheus:** An open-source monitoring solution used for collecting metrics from both Kubernetes resources and the different software tools installed in the cluster.

5.2 ModelMonitor object

In order to deploy the different components for the monitoring workflow, a ModelMonitor object is applied and reconciled by the Model Monitoring Operator. A detailed description of the available fields is included in Appendix A. The specification for this experimentation is the following.

5.2.1 Inference data analysis

As for inference analysis, all the statistics available are computed over tumbling windows of **15 seconds** of duration and slide, with a watermark delay of **6 seconds**. This means that late data within the last 6 seconds since the last event received are included in the window. Instances are not duplicated among windows. The purpose of this choice is simply to reduce the complexity and amount of results for an easier evaluation of the resulting inference analysis. The statistics computed are all the available in the framework: minimum, maximum, count, average, mean, sample

standard deviation, sample correlation, sample covariance, feature distribution, percentiles 25_{th}, 50_{th} and 75_{th}, and the interquartile range.

Regarding outliers detection, feature values which are further than three standard deviations from the mean are declared anomalous (i.e distance-based outlier detection). Since these values are commonly greater than the maximum or lower than the minimum values seen during model training, the detection of unseen data based on maximum and minimum thresholds is not included in the experimentation for the same reason as the window slide, to reduce the number of outputted results.

When it comes to concept drift detection, all the distribution-based detectors implemented in the framework are included, which are: Wasserstein distance, KL divergence and JS divergence. The reason of including all the algorithms is merely comparative, although KL and JS divergences might not be applied together in a real scenario due to their similarity. In their specification, the detectors accept two fields: *threshold* and *showAll*. While the former relates to the threshold value over which drift is considered to have occurred and the corresponding trace outputted, the latter indicates whether all the computed coefficients should be outputted independently of over-passing the threshold. For the sake of the comparison, all the detectors have a *True* value in this field.

In general terms, except for unseen events over maximum or above minimum baselines, all the computations available in the framework are performed to test the performance of the solution at its highest computation demand.

5.2.2 Kafka topics

As mentioned in previous sections, as for this work Kafka is the only fully supported storage for inference logs and analysis, although other sources and sinks can be easily included such as csv and parquet files, or other external storages via extension of Spark interfaces.

In this case, the *ModelMonitor* object includes the specification of four Kafka topics for the inference logs, computed statistics, outliers detected and concept drift analysis, respectively. The former is defined with three partitions and a replication factor of one. The other three topics are defined with only one partition and a replication factor of one as well.

5.2.3 Inference Logger

Concerning the Inference Logger, the autoscaler provided by Knative (KPA) is configured to scale the service with a target concurrency of 80. Also, resources of *100m* (0.1 units) CPU and *56Mi* of memory, expandable up to *68Mi* if needed, are dedicated to it. These values are Kubernetes-specific measures. One CPU unit is equivalent to 1 vCPU/Core for cloud providers or 1 hyper-thread on bare-metal Intel processors, as specified in the official documentation. As for memory, it is specified in mebibytes.

It is important to note that these resources act as soft limits, which means that they are considered as a reference for resource allocation and scaling purposes but they can be over-passed if needed. By contrasts, the use of hard limits means the termination of containers which attempt to exceed allocated resources. These are not used for this experimentation. The rest of possible configurations are kept with default values.

5.2.4 Model Monitoring Job

Regarding the Model Monitoring Job, three executors are configured to perform the inference analysis. For each of them, 1 CPU and *1024m* of memory are allocated. As for the driver, these resources are 1 CPU and *512m* of memory. In contrast with the Inference Logger, in this case memory is specified in megabytes. The rest of possible configuration are kept with default values.

5.3 ML model serving

As described in Section 3, a model for the Iris classification problem has been trained using Hopsworks [54] and its Feature store. This model is deployed using KFserving. Autoscaling for this service is configured using the autoscaler provided by Knative (KPA) with a target concurrency of 80. The resources allocated for this service are *100m* CPU and *256Mi* of memory.

Additionally, the *InferenceService* kind of KFServing provides two fields for configuring a logger service that send the requests, responses or both to a pre-defined endpoint in CloudEvents [27] format. The endpoint needs to be filled with the endpoint of the Inference Logger service deployed which transform them into Kafka messages and send them to the corresponding

Kafka topic. This endpoint follows the format *http://[modelmonitor_name]-inferencelogger.[namespace]*, where *[modelmonitor_name]* and *[namespace]* refers to the name of the *ModelMonitor* object and its namespace, respectively. Additionally, this endpoint url can be found in the description of the Inference Logger service deployed by the Model Monitoring Operator.

Lastly, only request logs are stored and processed since there are not prediction analysis algorithms implemented as for this work.

5.4 Inference data streams

An inference data stream of 250.000 instances has been generated in five different stages, for each of the four available features as shown in Figure 5.4.1. Each stage is composed of 50.000 instances. The first and last stages corresponds to normal instances compared with the training data. This is, it strictly follows the same distribution as the one seen by the trained model. The second and fourth stages includes a 10% of outliers, which for this experimentation means instances further than three standard deviations from the baseline mean. The third stage contains instances with concept drift. This is achieved by displacing the location of the instances as well as modifying its variation.

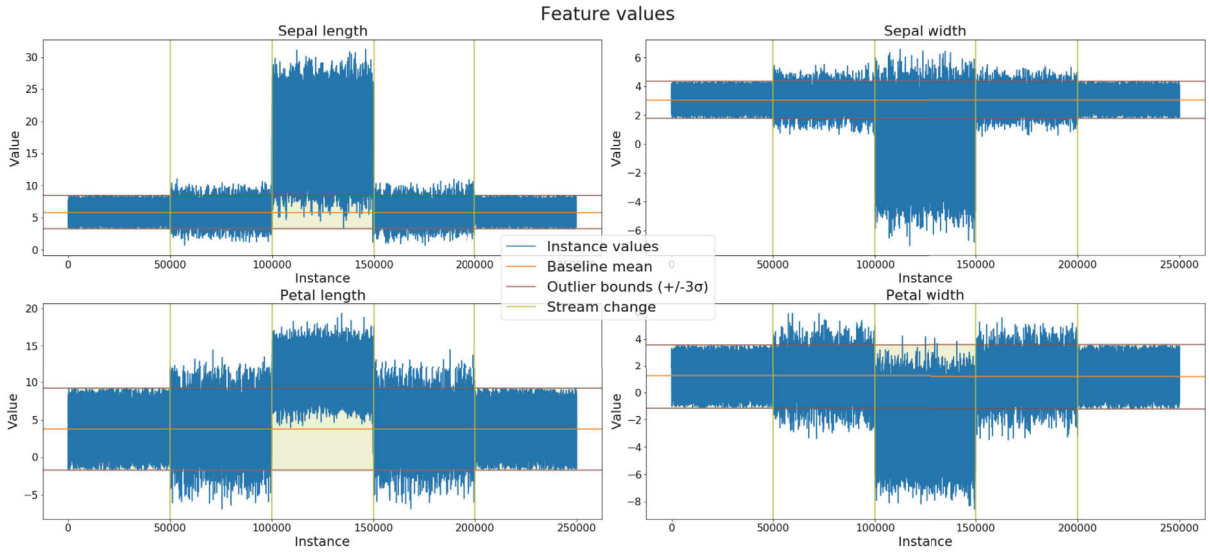

Figure 5.4.1: Data streams used as inference data for each feature. Each stream is divided into five stages (green vertical lines). They refer to normal data, presence of outliers and concept drift, respectively.
The green area represents data seen during model training.

These stages have been clearly delimited for a better interpretation and visualization of the results along the experimentation process.

5.5 Inferencing

In order to analyse the solution scalability, the experimentation consists of making predictions at different rates using expected instances seen during model training, instances representing outliers and instances with concept shift as described in the previous section.

To send these requests at different concurrency levels, the http load generator tool called Hey [1] has been modified for avoiding duplication of instances. Originally, this tool use the same input stream for every request sender, which means sending the same instances multiple times. In Table 5.5.1 the different inference stages and concurrency levels are presented. The experimentation has been conducted in a computer with an Intel Core i7-8750H CPU with 6 cores, which supposes 12 logical cores, and 16Gb of RAM.

Table 5.5.1: Inference stages rates.

Stages	Nº of requests	Concurrency	Instance characterization
1st	50.000	48	Normal instances.
2nd	50.000	96	Normal instances with 10% of outliers
3rd	50.000	144	Instances with concept drift, which in turn implies the presence of outliers.
4th	50.000	96	Normal instances with 10% of outliers
5th	50.000	48	Normal instances.

[1] Hey. A tiny program that sends load to a web application. https://github.com/rakyll/hey

Chapter 6

Results

In this section, an evaluation of performance metrics and inference analysis obtained after running the experimentation is conducted. These metrics include the number of replicas, latencies, throughput or resource consumption of the different components. After that, the results of the inference analysis conducted by the monitoring job are presented in three sub-sections corresponding to statistics, outliers and drift detected.

6.1 Experimentation

Concerning the http load generator, Figure 6.1.1 visualizes different statistics of the response times along the experimentation. These response times are grouped by the request offset time, which is the number of seconds elapsed since the beginning of the experimentation. In other words, response times of requests that started within the same second are binned together.

As it can be observed, response times during the first and last stages remain quite stable without big gaps between maximum, minimum and mean values. With regards to the second and third stage, it can be seen that during the first seconds where up-scaling occurs due to an increase in traffic load, the response times increase considerably and then stabilize until the end of the stages. The reason of the sudden increment in response times is due to temporary insufficient instances available in the cluster at that moment to satisfy the new demand. These latencies until a new instance is ready is considered a cold-start. Then, the availability of sufficient instances leads to the

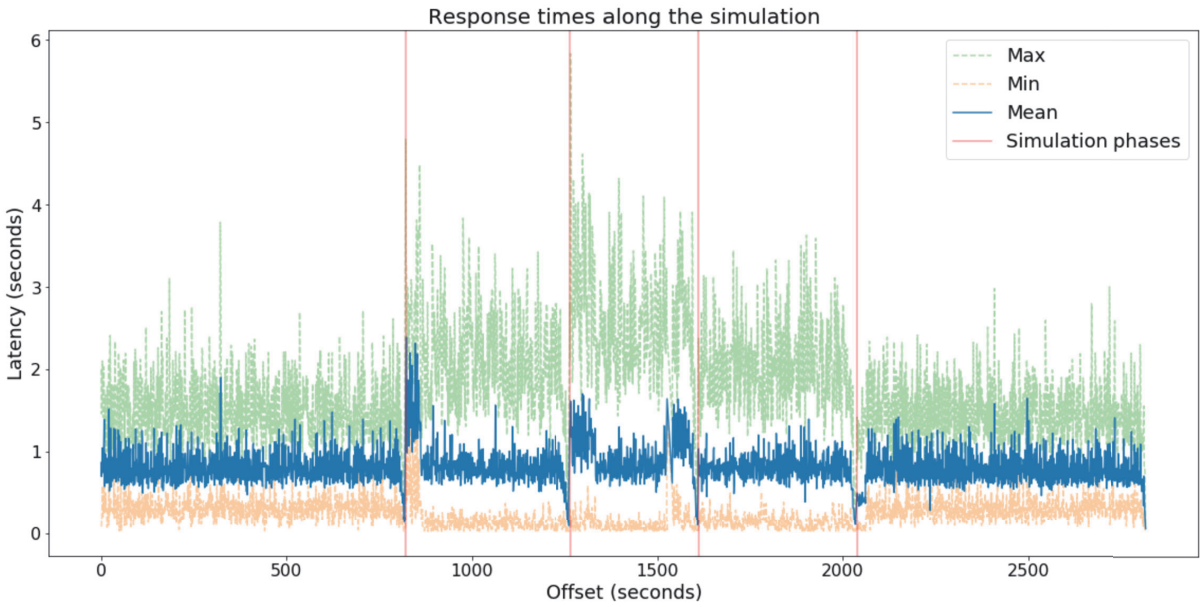

Figure 6.1.1: Response times measured by the http load generator along the experimentation. Requests are grouped by offset, which is the number of seconds since the beginning of the experimentation. Red vertical lines represent the beginning of a new experimentation stage.

decrease and stabilization of the response times. As for the third stage, there is also an increase of response times in the last seconds due to an inappropriate revision that led to the momentary removal of one instance. This can be contrasted with Figure 6.2.3 in the next section.

Additionally, it can be observed a slightly reduction at the end of each stage. This is due to the fact that the http load generator is executed once per stage, which means that there is a minimal delay between the last requests of one stage and the first of the next one. This leads to faster responses at the end of each stage due to a brief decrease of concurrent requests.

In the Figure 6.1.2, a histogram of the response times along the experimentation is presented. The bulk of response times are less than one second, being the average time 0.822 seconds and the 75_{th} percentile 1.04 seconds.

6.2 Architecture behavior

As mentioned before, cluster performance have been monitored using metrics from different components of the architecture such as Kubernetes metrics-server or Knative serving. These metrics are scraped using Prometheus [28].

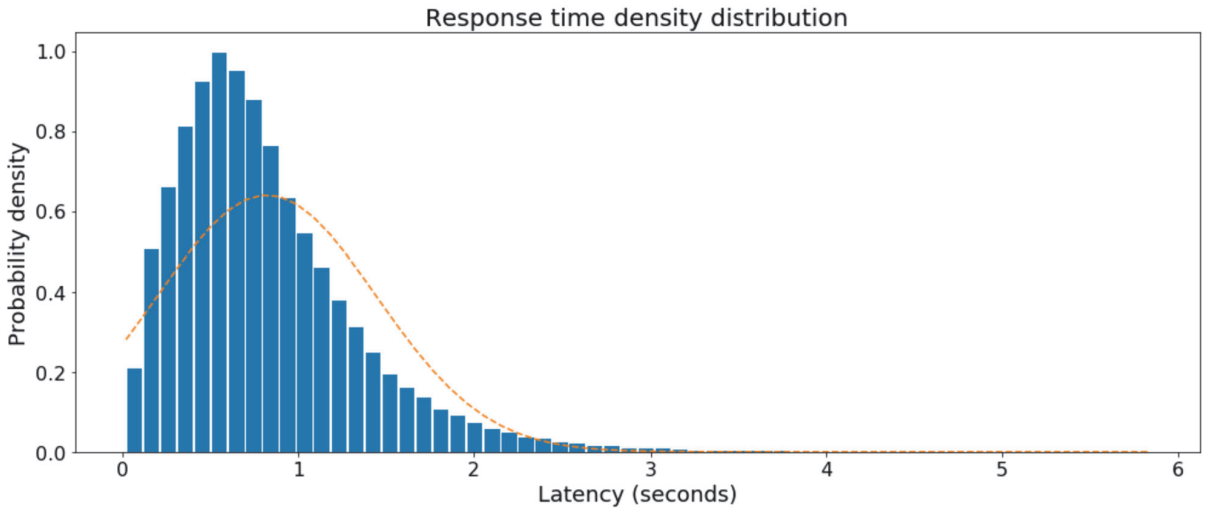

Figure 6.1.2: Density distribution of the response times measured by the http load generator.

6.2.1 Control plane

In the Figure 6.2.1 it is shown the CPU usage of the system backbone components, considered as control plane, which are kube-system, Istio and Knative. The former can be seen as the main Kubernetes implementation for cluster and resources management. Istio is the service mesh used for networking and services discovery. Knative is composed of three independent implementations that can be installed separately: eventing, serving and monitoring. Knative eventing is not required in this experimentation, therefore it does not appear in the visualization. As for Knative serving, it is used for deploying services with a server-less approach, providing auto-scaling and networking management. The model inference service and the Inference Logger implemented in this work are deployed using Knative services. Furthermore, Knative monitoring uses Prometheus to collect service performance metrics.

While CPU usage remains considerably stable around 0.17 cores for kube-system and Knative monitoring along the experimentation, it increases for Knative serving and Istio system. These increments coincide with the experimentation stages, where a heavier traffic load is generated due to higher concurrency levels.

The evolution of memory usage by the same components is represented in Figure 6.2.2. Contrary to CPU usage, it remains quite stable for all the components but for Knative monitoring which is constantly gathering metrics so increasing over time until reaching 2.6 GB at the end of the experimentation.

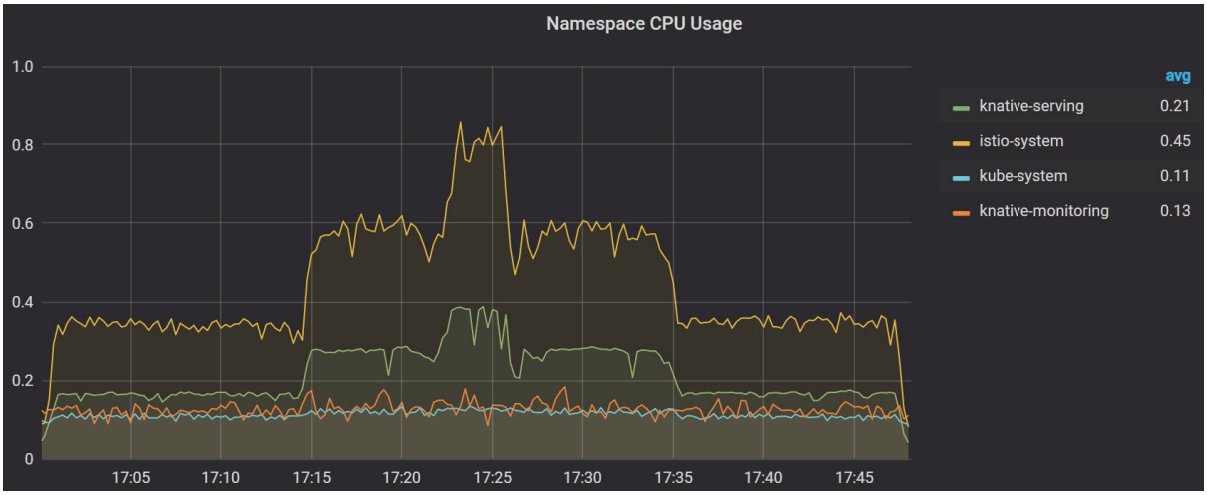

Figure 6.2.1: Control plane CPU usage by system (namespace). The evolution of CPU usage for Knative serving and Istio match the traffic loads generated along the experimentation stages.

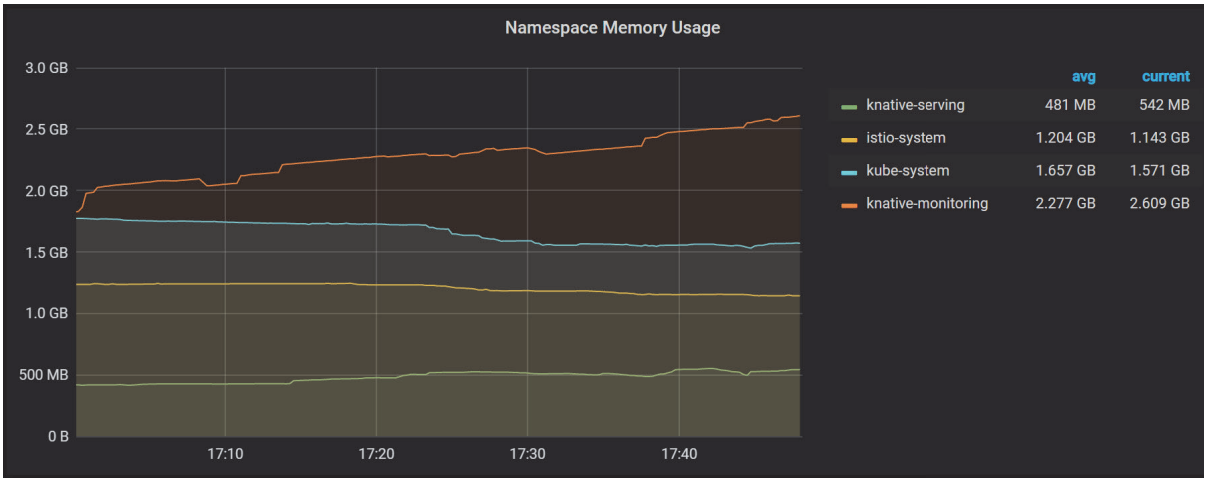

Figure 6.2.2: Control plane memory usage by system (namespace). Only Knative monitoring shows an increase in memory usage due to its continuous metrics collection over time.

6.2.2 Inference serving

As mentioned before, KFServing uses Knative serving to serve the model via the deployment of an *InferenceService* object. The total success rate achieved by the Inference Service is 99.4%, which means that 0.6% of the requests failed during the experimentation. As discussed later, these failures occur during up-scaling the number of pods.

Knative serving scale the number of pods dynamically, with regards to a target metric and value specified during its creation. In this case, it scales based on concurrency level with a target of 80. The increase in the number of pods along the different

experimentation stages can be seen in the Figure 6.2.3. Being one pod the minimum possible number, the inference service did not need to scale until the second and third stages where one pod was added in each of them. During the second half of the experimentation, a downscale was conducted with the same magnitude.

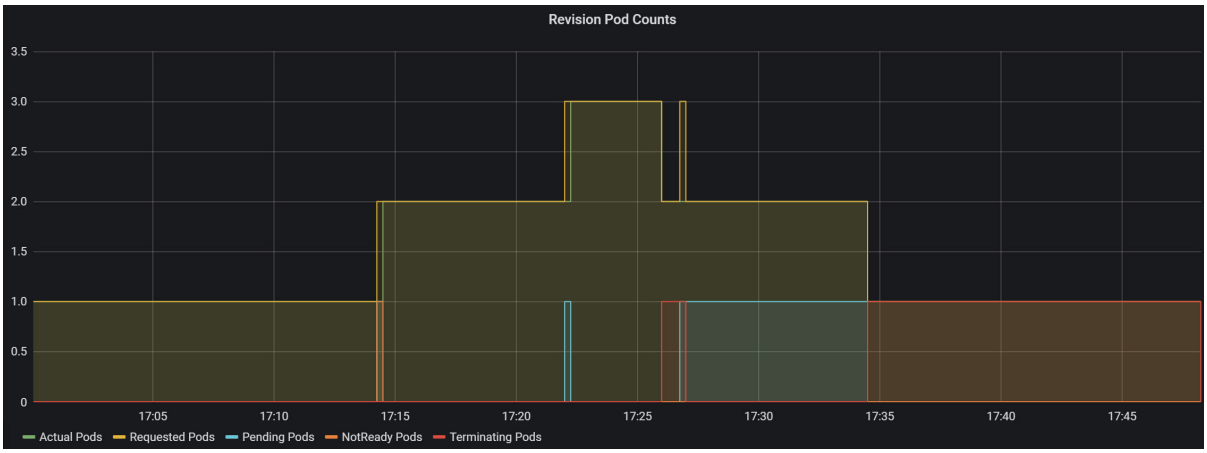

Figure 6.2.3: Number of Inference Service pods along the experimentation.

As discussed in Section 6.1, at the end of the third stage there is a slight peak in response times. This is caused by an inappropriate momentary down-scale that might occur when used resources oscillate very close to requested resources, leading to short-term ups and downs in the number of pods.

It is clear that the increment in the number of pods is due to an increase in the request volume received by the Inference Service. Request counts per last minute are shown in the Figure 6.2.4, aggregated by response http code. As for successful requests, the average counts are close to 3.850 and 7.250 for the first and second stages, respectively, reaching up to 10.700 in the third stage. The second half of the experimentation shows similar request counts.

As indicated above, successful requests correspond to 99.4% of the total requests received. It can be observed that failed responses occur when up-scaling is required to satisfy higher demands. When a new pod is created, Kubernetes has to allocate the corresponding resources, set up containers and configure additional requirements such as volumes or connectivity among other things. Also, container images or any of their dependencies might not be loaded in memory already. This is considered as a cold-start and involves a varying delay of milliseconds or seconds until the new pod is ready. During this time, some requests happen to fail due to insufficient resources.

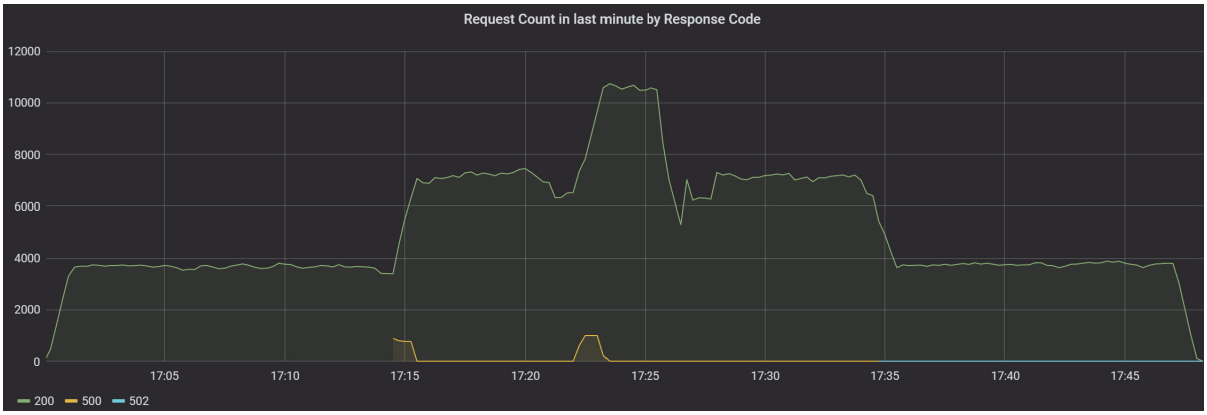

Figure 6.2.4: Request count per last minute received by the Inference Service during the experimentation. When up-scaling is required, some requests fail until more pods are created and ready (yellow line).

It is important to mention that although Knative Pod Autoscaling (KPA) includes mechanisms to scale pods before reaching the maximum capacity such as panic windows, these are more effective when demand changes gradually. In other words, when sudden changes in demand appear, such as in this work, there are less chances of over-passing the panicking threshold before reaching the capacity limit.

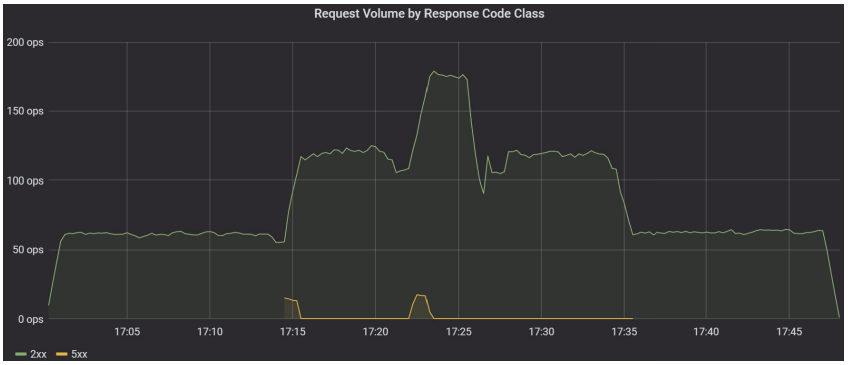

Figure 6.2.5: Request volume per http code received by the Inference Service during the experimentation. When up-scaling is required, some requests fail until more pods are created and ready (yellow line).

In the Figure 6.2.5, the number of operations per second shows the same pattern than request counts along the experimentation, which means that the service manages to adapt its throughput under new demands. The average throughput is 64, 122 and 175 OPS during each experimentation stage, respectively.

Looking at the average response times per last minute (Figure 6.2.6), it can be observed that except for the peaks during up-scaling, response times remain stable throughout the experimentation, regardless the request volume received. The average response times correspond to 585 milliseconds and 1.9 seconds considering the 50_{th} and 95_{th}

percentile of quicker responses.

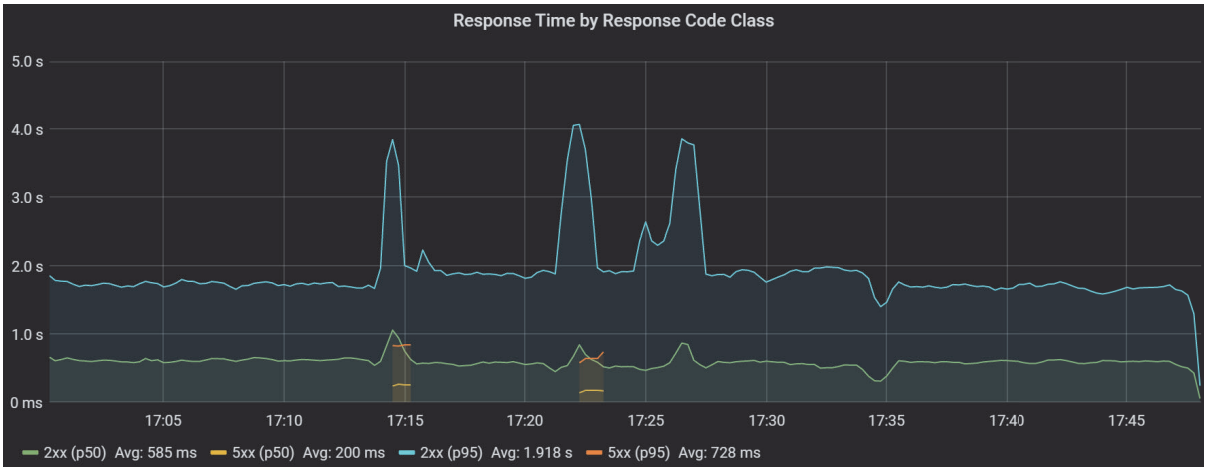

Figure 6.2.6: Response times in the last minute by the Inference Service during the experimentation. The different lines corresponds to 50_{th}, 90_{th}, 95_{th} and 99_{th} percentiles, including the fastest responses.

As an attempt to reduce response times, it is possible to adjust the configuration for the auto-scaling tool, making it more prone to scale by decreasing the periodicity when the state of the pod is checked or providing lower threshold values. However, this can lead to a worse exploitation of resources for each pod. Additional information concerning a specific scenario such as expected demand, usage patterns or which response times are considered acceptable should be taking into account at this point. As mentioned before, for this experimentation only the target concurrency was specified leaving thresholds and other parameters with default values.

Finally, Figure 6.2.7 shows the total resources usage for one pod. As explained in Section 2.2.2, each pod can contain multiple containers, volumes or other resources co-existing within the same context. In this case, the figure shows the resource usage by each of the containers in the pod which are three: kfserving-container, inferenceservice-logger and queue-proxy. The first and second ones are for model inference and inference logging, respectively. Queue-proxy container acts as a proxy, making the pod discoverable within the cluster.

The container in charge of model inference is comprehensively the more resource-demanding one. It requires the trained model, serving runtime and other dependencies to be loaded in memory. As for CPU usage, it reached 0.087, 0.175 and 0.261 cores in the different stages of the experimentation. Memory usage hit 198, 392 and 656 MB, respectively.

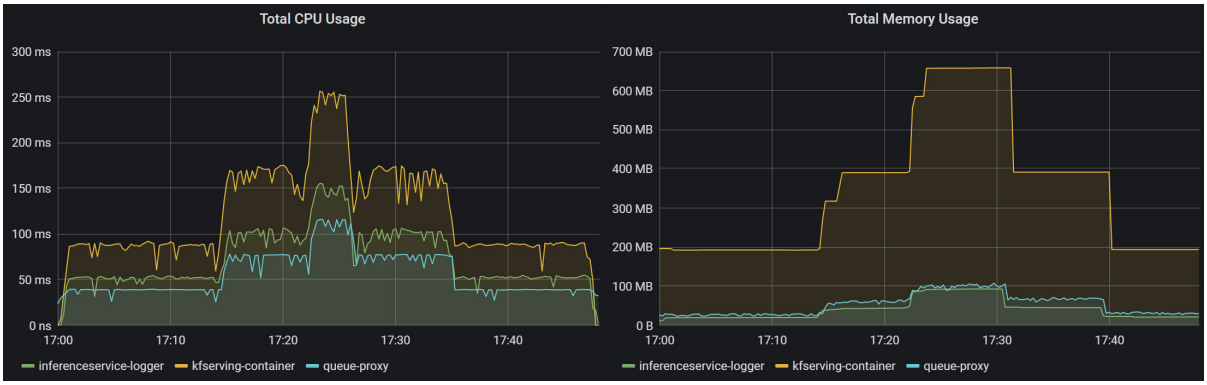

Figure 6.2.7: Resources usage by the Inference Service during the experimentation. Each pod contains three containers: kfserving-container for model inference, inferenceservice-logger for inference logging and queue-proxy for service discovery and networking.

6.2.3 Inference logger

As explained in previous sections, Inference Logger is an http server deployed using Knative serving that ingests logs in CloudEvents format, transform them into Kafka messages and send them to the corresponding Kafka topic. The total success rate achieved by the Inference Logger is 100%, which means that all logs received were successfully deliver to Kafka (i.e inference topic). Given the request volume received and the low computation needs of the service, only one instance sufficed along the whole experimentation.

An increase in the request volume received by the Inference Service means an increase in the number of logs transmitted to the Inference Logger. Figure 6.2.8 shows the number of requests received per minute by the Inference Logger. As expected, it also varies over time, being around 3.900, 7.600 and 10.600 request per minute in average during the first three stages, respectively. The second half of the experimentation shows similar request counts.

This variance in the number of request per minute coincide with the number of operations per second (OPS) (i.e throughput) among the stages, which indicates absence of bottlenecks in the Inference Logger during the experimentation. This is visualized in Figure 6.2.9, which shows how it is stabilized around 63, 122 and 177 OPS in the first three stages, respectively.

Figure 6.2.10 shows the average response times per last minute. In this case, it can be observed that as traffic load increases in each stage, so does response times. This is due to the fact that only one pod was used for the whole experimentation. No

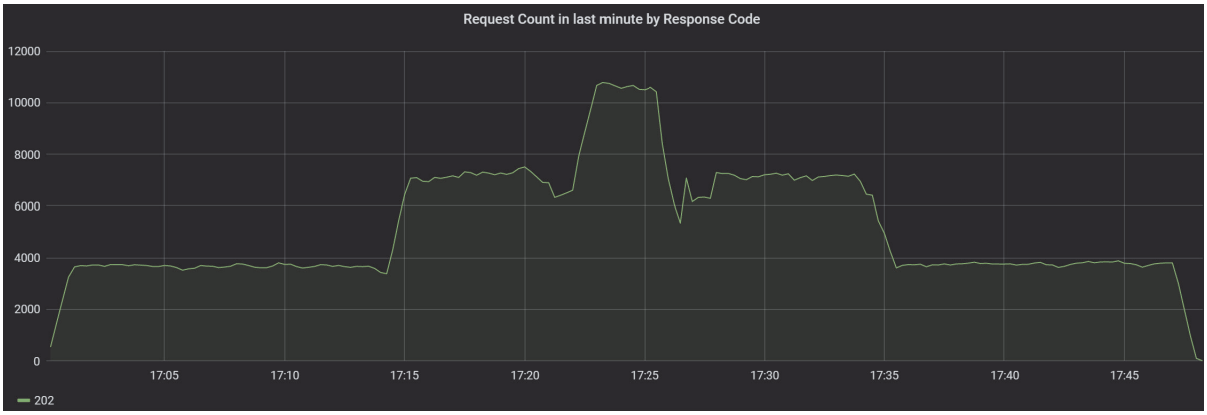

Figure 6.2.8: Request count per last minute received by the Inference Logger during the experimentation.

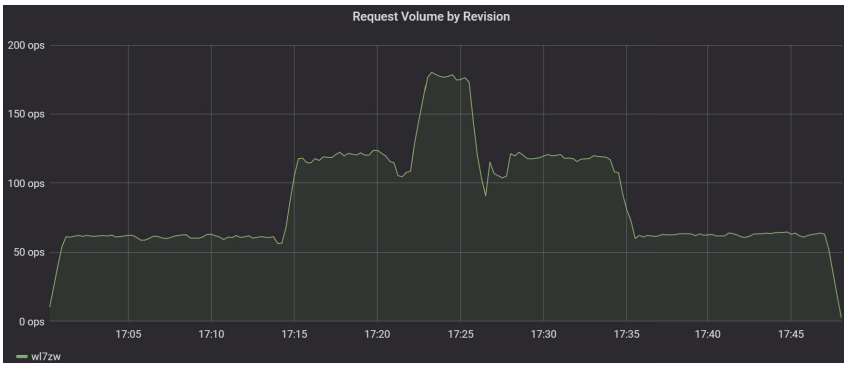

Figure 6.2.9: Operations per second (OPS) of the Inference Logger during the experimentation.

up-scaling was conducted since default thresholds were not over-passed. Similarly than with the Inference Service, the only auto-scaling parameter overwritten for this experimentation is the concurrency target with a value of 80. Again, the criteria for auto-scaling might be adapted to optimize the trade-off between resource exploitation and acceptable latencies according to a real scenario.

Excluding the 10% of the slowest responses (i.e 90_{th} percentile), response times stay close to 16, 30 and 64ms in average for the first three stages, respectively. The differences between percentiles remain quite stable within the same stage over time, which means that variance in the response times stays considerably constant. These latencies were considered acceptable for this experimentation so no threshold adjustment was conducted.

Since the delivery of Kafka messages does not involved changes in the Kafka producers that could prevent them from being reused among threads in the Inference Logger, those are instantiated at server starting phase, kept in memory and shared across

threads and request handling. This helps to quicken response times by avoiding opening and closing connections to Kafka constantly in each request handling.

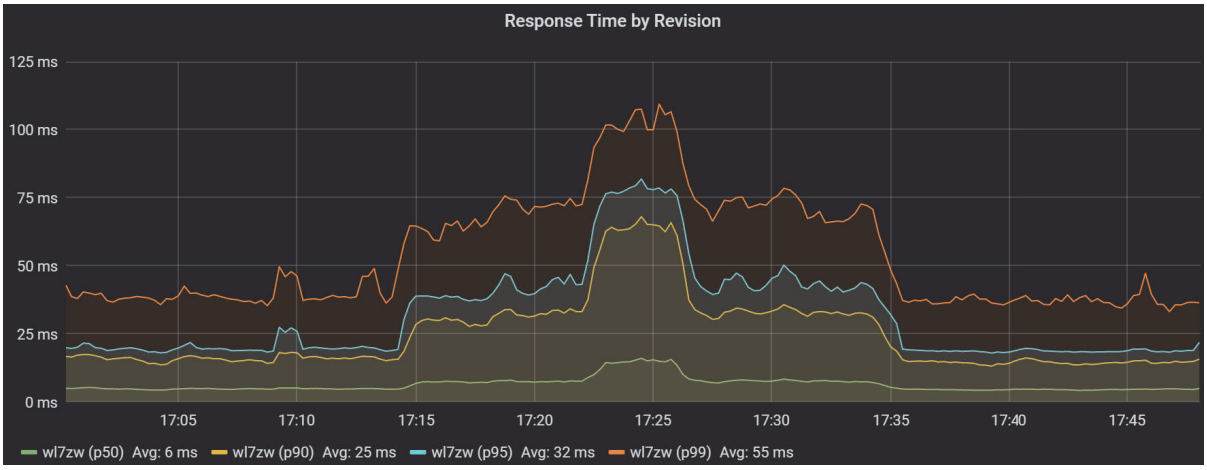

Figure 6.2.10: Average of response times per last minute of the Inference Logger during the experimentation.

Lastly, Figure 6.2.11 represents the CPU and memory usage for one pod along the experimentation. Each pod contains two containers: modelmonitor-container and queue-proxy. The former implements the http server, while the latter makes the pod discoverable and reachable within the cluster. As it can be observed, both containers are CPU and memory efficient, being queue-proxy the one with higher resource needs.

As for the http server, CPU usages throughout the experimentation stages stay close to 0.037, 0.068 and 0.089 cores, respectively, while memory usage remains almost constant around 8 MB for the whole experimentation.

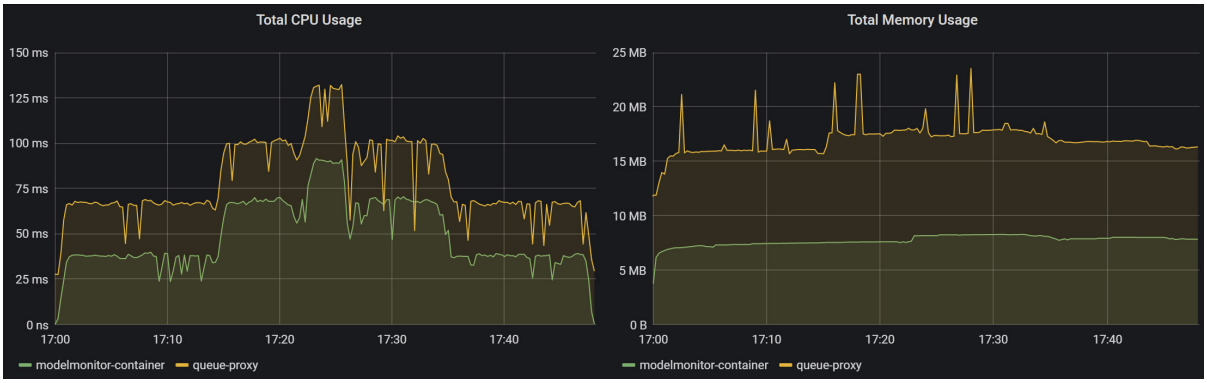

Figure 6.2.11: CPU and memory usage by the Inference Logger along the experimentation. Each pod contains two containers: modelmonitor-container with the http server and queue-proxy for service discovery and networking.

6.2.4 Model Monitoring job

As for the Model Monitoring job, one driver and three executors were defined in the *ModelMonitor* specification, as mentioned in the previous chapter. Spark uses a mechanism called dynamic resource allocation for auto-scaling executors. On Kubernetes, spark-on-k8s-operator is an operator that facilitates the deployment and configuration of a driver controller which, in turn, is in charge of the creation and supervision of the executor pods. At the time of this work, dynamic resource allocation is not supported by the operator. Hence, three executors have been specified in advance to prevent shortage of capacity in this experiment.

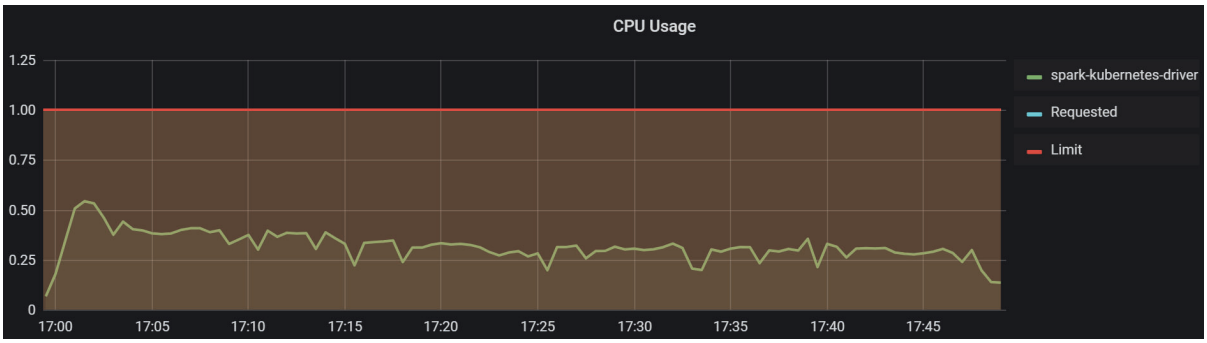

Figure 6.2.12: CPU usage of the Spark's driver running the Model Monitoring Job.

In the specification, 1 core is defined as both requested and limit CPU resources. The Figure 6.2.12 shows the CPU consumption by the driver throughout the experimentation. It can be seen that it generally stays in the range between 0.25 and 0.5 cores, slightly decreasing over time. It is assumable that at the start of the job there is an additional CPU load for setting up the streaming queries, establishing connection with the Kafka brokers and other pre-execution preparations.

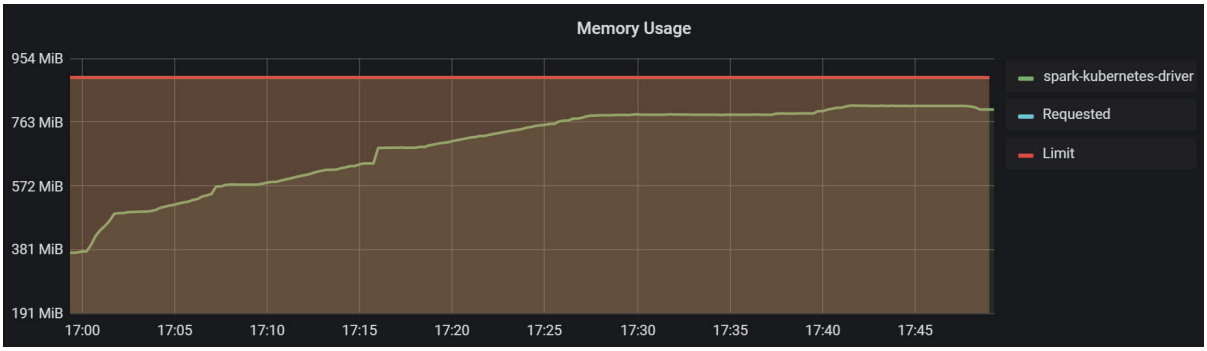

Figure 6.2.13: Memory usage of the Spark's driver running the Model Monitoring Job.

Driver memory consumption increases over time as shown in Figure 6.2.13. It is important to mention that both memory consumption and limit is represented at a

pod level. Therefore, it also includes resources of other containers and dependencies coexisting in the same pod such as the sidecar proxy. Specifically for the driver container, a limit of *512m* (Megabytes) was defined in the ModelMonitor specification. At the point of greatest consumption, the pod reaches 808 MiB of memory usage.

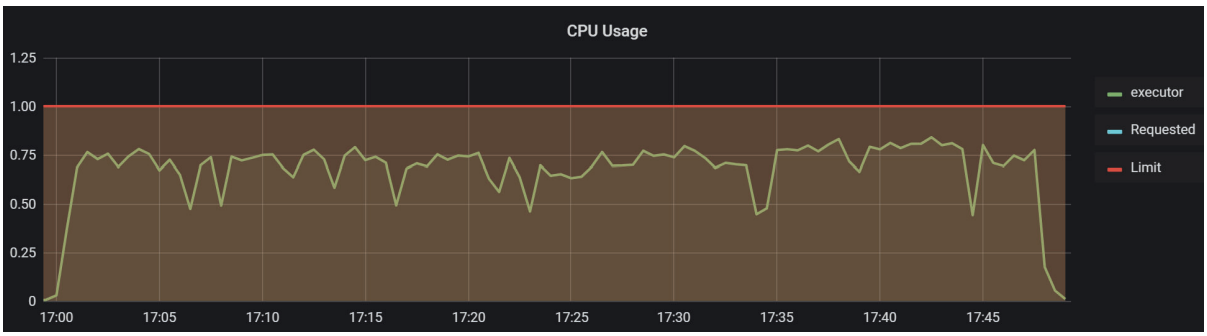

Figure 6.2.14: CPU usage of one of the Spark's executors behind the Model Monitoring Job. CPU consumption is practically identical for all the executors.

As for the executors, they were configured to use 1 core, for both requested and limit CPU resources, and 1024m (Megabytes) of memory. The performance of all the executors is practically identical. The evolution of CPU and memory consumption of one executor can be seen in Figures 6.2.14 and 6.2.15, respectively. The former remains close to 0.75 cores throughout the experimentation, with occasional troughs down to 0.5 cores.

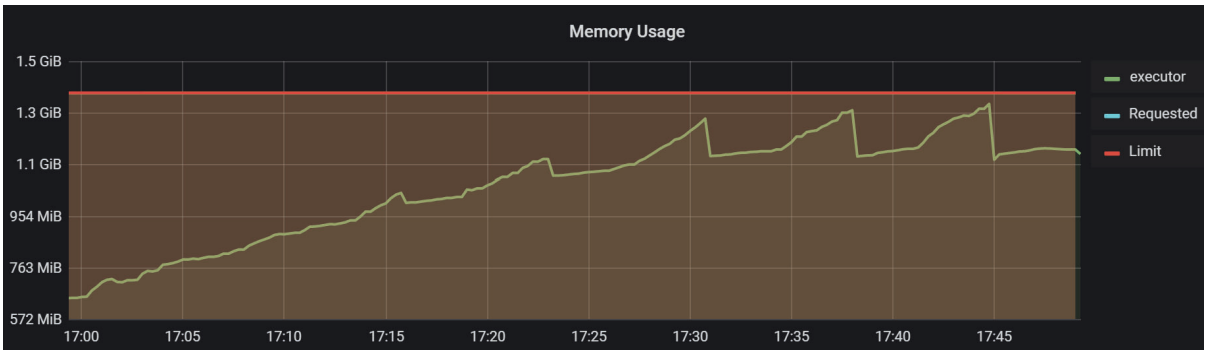

Figure 6.2.15: Memory usage of one of the Spark's executors behind the Model Monitoring Job. Memory consumption is practically identical for all the executors.

Memory consumption by the executor also increases gradually. In this case, it stabilizes when it reaches around 1177 MiB, showing periodic peaks up to 1327 MiB of memory.

6.2.5 Kafka cluster

As already mentioned, the Kafka cluster configured for this experimentation consists of three Zookeper replicas (i.e quorum size of three) and five brokers with 6 GiB of storage backed by AWS Elastic Block Storage (EBS) per each. Two of these brokers are used by the Inference Logger for inference logs ingestion and the remaining three are used for the Model Monitoring job, one per topic. Table 6.2.1 contains configuration details of each topic as well as their final log size after the experimentation.

Table 6.2.1: Kafka topics information including partitions (P), replication factor (RF) and log size (LS).

Name	Content	P	RF	LS
iris-inference-topic	Request instances	3	1	38.5 MiB
iris-inference-stats-topic	Windowed instances statistics	1	1	781 KiB
iris-inference-outliers-topic	Outliers detected	1	1	30.5 MiB
iris-inference-drift-topic	Concept drift coefficients	1	1	454 KiB

Figures 6.2.16 and 6.2.17 show the volume of messages ingested per topic and measured in total log size per second. It can be observed how the message rate of iris-inference-topic match the same pattern seen previously, due to differences in traffic load during each experimentation stage, reaching a maximum of 158 bytes per second. As for iris-inference-outliers-topic, it starts to receive messages in the second stage where a 10% of outliers are included in the inference stream and reaches a maximum of 564 bytes per second in the third stage. In this stage there is a large increment of received messages per second. The reason of this is the data drift added in this stage, which leads to a vast number of anomalous observations due to their distance with the baseline mean. Additionally, while each inference message contains four values (i.e one per feature), a new outlier message is generated per feature value detected anomalous. Therefore, there are four potential outlier messages per instance.

Regarding the topics for inference statistics and concept drift coefficients, the message sizes ingested per second stays constant within the range between 0.720 and 0.960 for the former, and between 0.240 and 0.320 bytes per second for the latter. This is due to the windowing technique used by the Model Monitoring job, which maintains certain pace in the computation of instances and the output of results.

Lastly, Figure 6.2.18 shows the total incoming and outgoing byte rates achieved

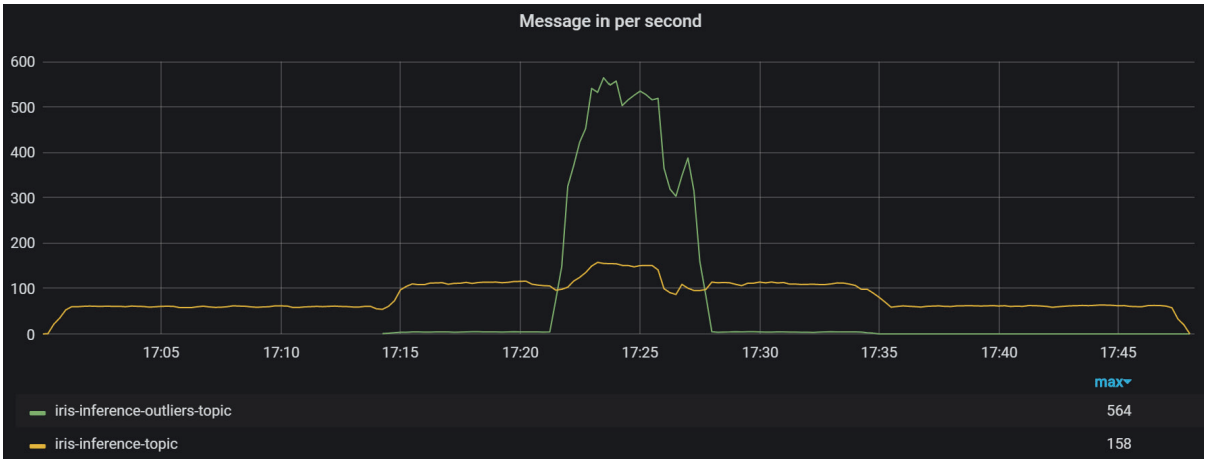

Figure 6.2.16: Instances and outliers messages ingested by Kafka throughout the experimentation. They are measured in bytes per second.

Figure 6.2.17: Instance statistics and data drift coefficients ingested by Kafka throughout the experimentation. They are measured in bytes per second.

in Kafka. As expected, it also matches the traffic load generated among stages. It is important to mention the existence of an additional internal topic called *__consumer_offsets* which is used internally by Kafka to keep track of two types of offsets: current and committed. The former indicates which records have been already consumed, while the later indicates which records have been processed by the consumer. The incoming and outgoing byte rates of this topic are also included in the figure.

6.3 Statistics

The Model Monitoring job groups inference logs in tumbling (i.e not overlapped) windows of 15 seconds length using the request time field included in each log. The

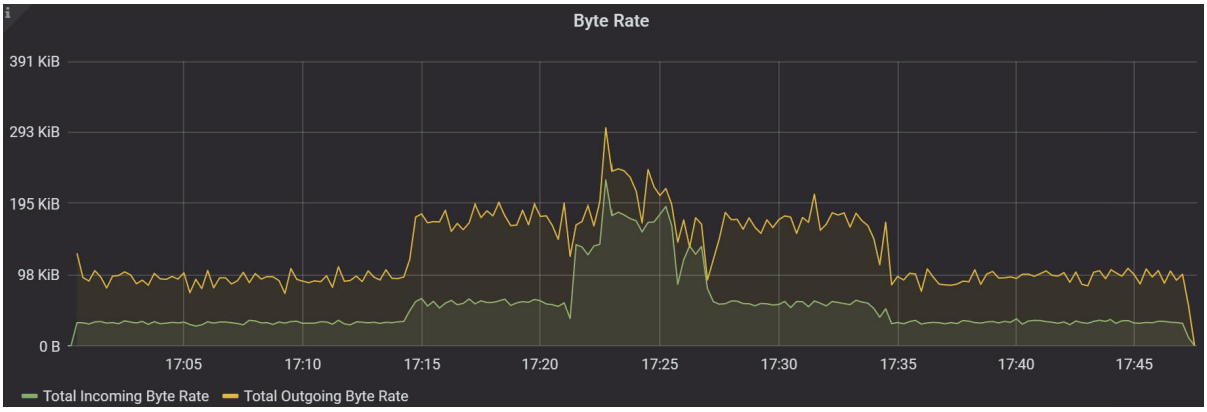

Figure 6.2.18: Total incoming and outgoing byte rates in Kafka along the experimentation.

watermark delay is set to 6 seconds, which refers to how late an instance can arrived so as not to be discarded. As mentioned before, the use of tumbling windows instead of sliding windows facilitates the visualization and interpretation of the results since their size is considerably reduced.

Additionally, larger window lengths imply longer computation delays. Therefore, it is important to keep a balance in the trade-off between acceptable computation delays and number of instances per window since shorter windows might contain less logs. Likewise, a small number of logs per window leads to more inconsistent statistics and, therefore, more uncertainty in the information obtained about the data stream.

Figure 6.3.1 shows the number of instances per window processed by the Model Monitoring job throughout the experimentation. During the first stage, windows contain around 987 logs each. In the second and third stages this number increases up to 1798 and 2379 instances, respectively. The second half of the experimentation shows similar counts. Additionally, the slight delays among stages can also be appreciated in the troughs of instance counts at the beginning of each stage.

As previously mentioned, each instance contains four features: sepal length, sepal width, petal length and petal width. All the statistics available in the framework implemented in this work are computed on each of the features in a windowed basis. These are minimum, maximum, average, mean, count, sample standard deviation, sample correlation, sample co-variance, density distributions, percentiles 25_{th}, 50_{th}, 75_{th} and the interquartile range (IQR). The maximum, minimum, average and standard deviation of each feature throughout the experimentation are visualized in Figure 6.3.2.

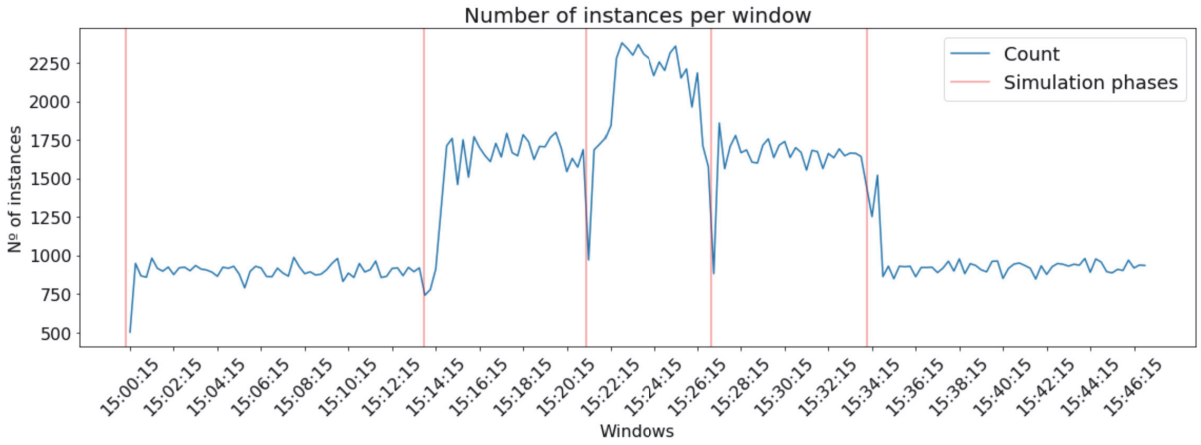

Figure 6.3.1: Number of instances per window along the experimentation. Vertical lines represent the start of a new stage.

In Section 5.4.1, the data streams for each feature were introduced, mentioning that they are deliberately divided into five distinguishable segments. The first and fifth segments contain normal values according with the baseline feature distributions used to train the model. The second and fourth segments are normal data with a 10% of outliers among the values, considered as values beyond three standard deviations from the baseline mean. Lastly, the features in the third segment have been drifted by changing its location and modifying their variance.

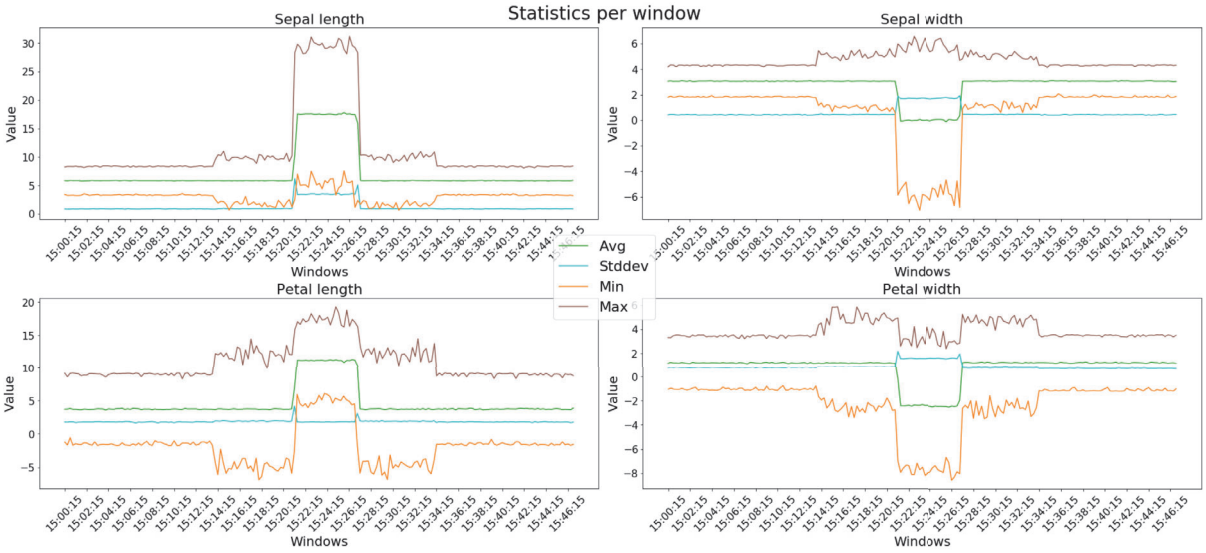

Figure 6.3.2: Statistics computed per window along the experimentation.

It can be observed how the statistics vary according with those segments. At the beginning and end of the plots, they remain stable since the statistical characteristics of the features do not change. Getting closer to the center of the plots, maximum and minimum values start to vary due to the outliers found in the data. However, the

mean and standard deviation stays mainly stable. Finally, in the center of the plots, all statistics are affected due to the data drift added in the third stage.

Each instance is assigned a request timestamp as soon as it reaches the Inference Service, before being processed by the model. Later in the Model Monitoring job, right after the computation of statistics another timestamp is recorded as the computation time. Statistics are computed on each window as soon as the watermark timestamp equals or exceeds the window end timestamp. Therefore, the computation delay can vary depending on where the request timestamp falls within the window, being the minimum delay possible the watermark delay. In other words, requests whose timestamps are close to the beginning of the window will be held as long as the window duration plus the watermark delay. By contrast, requests whose timestamps are close to the end of the window will still be held as long as the watermark delay.

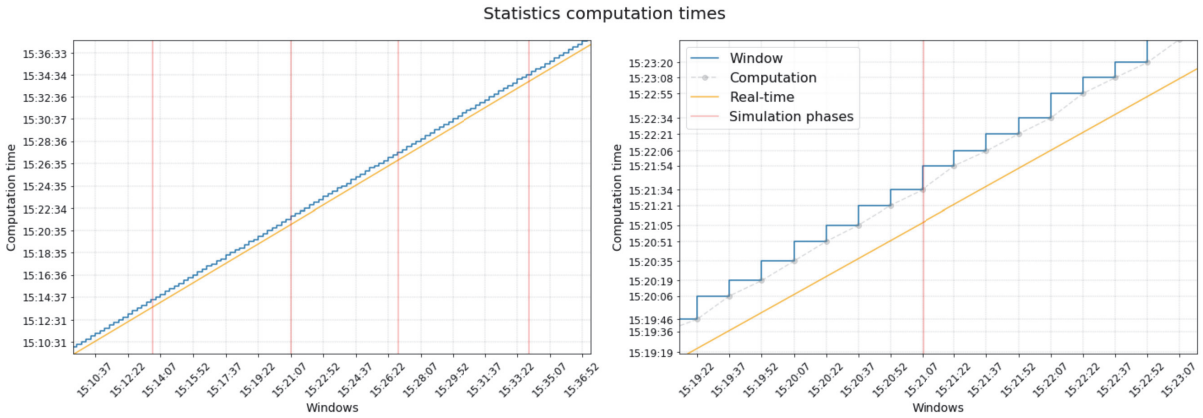

Figure 6.3.3: Statistics computation times compared with their corresponding window times (window mid-points). Vertical lines indicate the start of a experimentation stage. The gap between window and computation times remains almost the same along the different experimentation phases.

Additionally, two extra delays should be considered. If the computations over a window take longer than the window length, the next window will not be computed until the previous execution finishes. Also, instances consumed by the Model Monitoring job are first used for making predictions in the Inference Service, then sent to the Inference Logger where they are transformed and forwarded to Kafka, and finally consumed by the Model Monitoring job. All these hops involve certain latencies.

Figure 6.3.3 visualizes the computation times over their corresponding windows, and compares them with real time. The mid-point of the window is taken as reference for the comparisons, hence real delays can differ in +/- 7.5 seconds. These delays are quite disperse with minimum and maximum values of 23.48 and 34.34 seconds, as

shown in Figure 6.3.4. The bulk of computations are conducted within 30 seconds after request time. More concretely, percentiles 25_{th}, 50_{th} and 75_{th} correspond to 26.64, 28.69 and 29.92 seconds, respectively. These delays might be reduced by decreasing the window length or watermark delay. However, at this point additional criteria need to be considered depending on the real scenario such as acceptable computation delays, relevance of late events or minimum number of instances required per window.

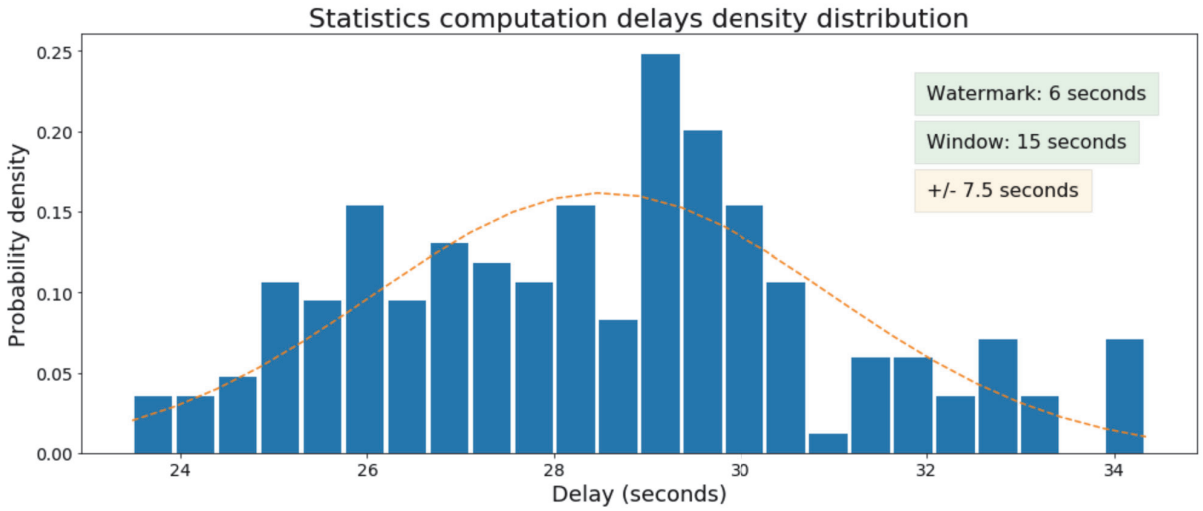

Figure 6.3.4: Delays between window times and statistics computation over the windows. Since window mid-points are taken as the references for the windows, real delays can differ in +/- 7.5 seconds.

6.4 Outliers detection

As already mentioned, the outliers detection approach included in this work is distance-based over baseline descriptive statistics, where feature values that fall beyond three standard deviations from the baseline mean are considered anomalous. One alert is generated per feature value. In this case, since feature values are analyzed individually, outlier detection is computed directly over the data stream without windowing. Right after an outlier is detected a timestamp is recorded as detection time. Figure 6.4.1 shows the detection times over their corresponding request times as well as the density distribution of detection delays. As it can be observed, detection times stay considerably constant over time, within the range between 2 and 8 seconds. The bulk of the detection times are less than 6 seconds, being the 25_{th}, 50_{th} and 75_{th} percentiles 4.11, 4.98 and 5.9 seconds, respectively.

As shown in the Figure 6.4.2, the longest detection delays are located in the third

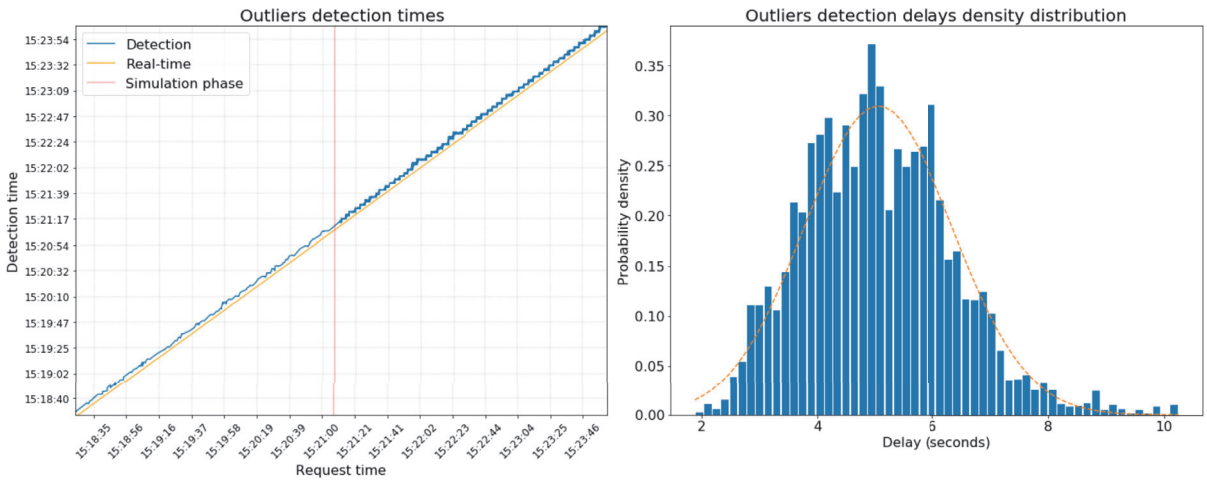

Figure 6.4.1: Outliers detection times along the experimentation. Figure on the left represents computation times over their corresponding requests times, zoomed in at the beginning of the third stage. The right-hand figure shows the density distribution of computation delays.

stage where the traffic load is heavier and more outliers are detected due to data drift presented in this stage. It can also be appreciated that outlier detection occurrences follow a similar pattern between the features. This is due to the way outlier detection is computed. All the feature values in the same instance are analysed in the same iteration. Therefore, in case that more than one feature value is anomalous within the same instance, their detection times are almost identical.

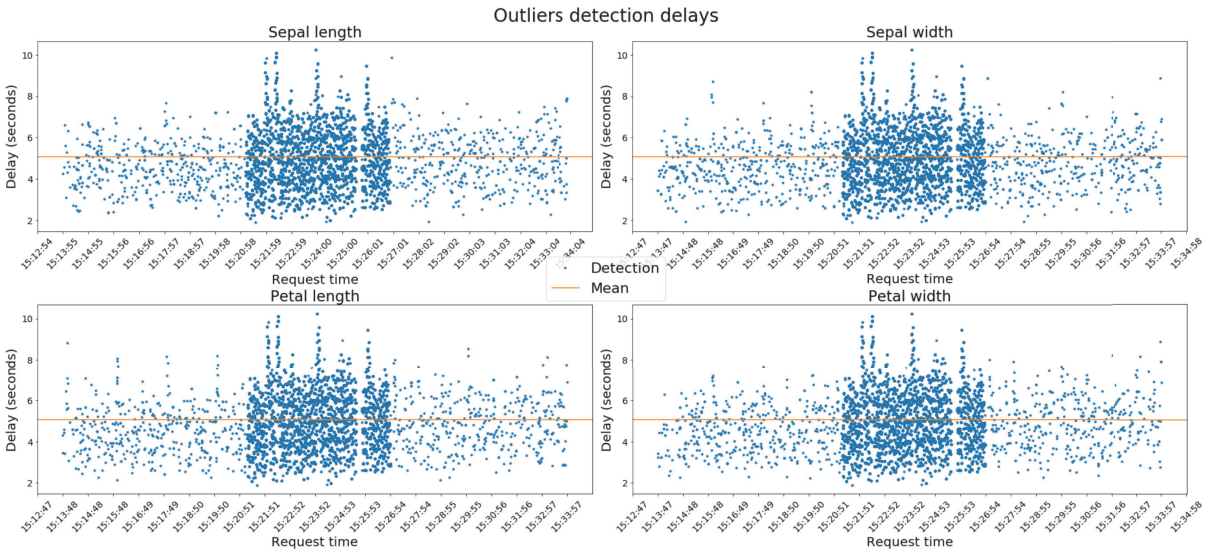

Figure 6.4.2: Delays between request times and the detection of outliers per feature.

6.5 Drift detection

As introduced in Section 4.2.3, the Model Monitoring framework includes the computation of three distribution-based data drift metrics: Wasserstein distance (or Earth's Mover Distance), Kullback-Leibler divergence and Jensen-Shannon divergence. A threshold-value can be provided per each algorithm to generate alerts when the computed coefficient exceed the corresponding thresholds. However, for a better visualization of the concept drift evolution in this experimentation no threshold has been defined, storing all the coefficients into Kafka.

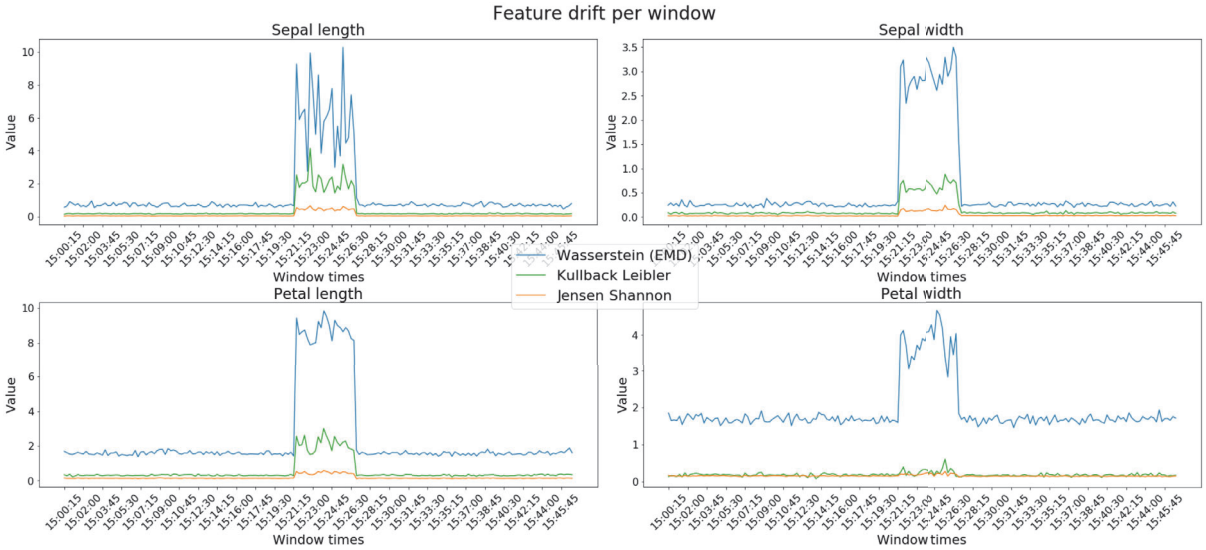

Figure 6.5.1: Data drift evolution per feature along the experimentation

Figure 6.5.1 shows the evolution of data drift for each feature. As it can be observed, all coefficients remain stable across stages except in the third one where data drift was added in all feature instances.

It is noticeable that Wasserstein distance is the most sensitive measure among the three. This is because it focuses on the distance between observed and baseline distributions, while the other metrics focus on the similarity between them. Therefore, features where the range of observed values is wider (i.e sepal length and petal length) show considerably higher distances. Also, features where observed values are more concentrated (i.e dense) and overlapping more the baseline distribution (i.e sepal and petal width) show lower divergence values. Comparing these results with the inference streams, a higher drift is noticeable in *sepal length* and *petal length* features.

As with the statistics, right after the computation of drift metrics a timestamp is recorded as computation time. Since the algorithms implemented are distribution-

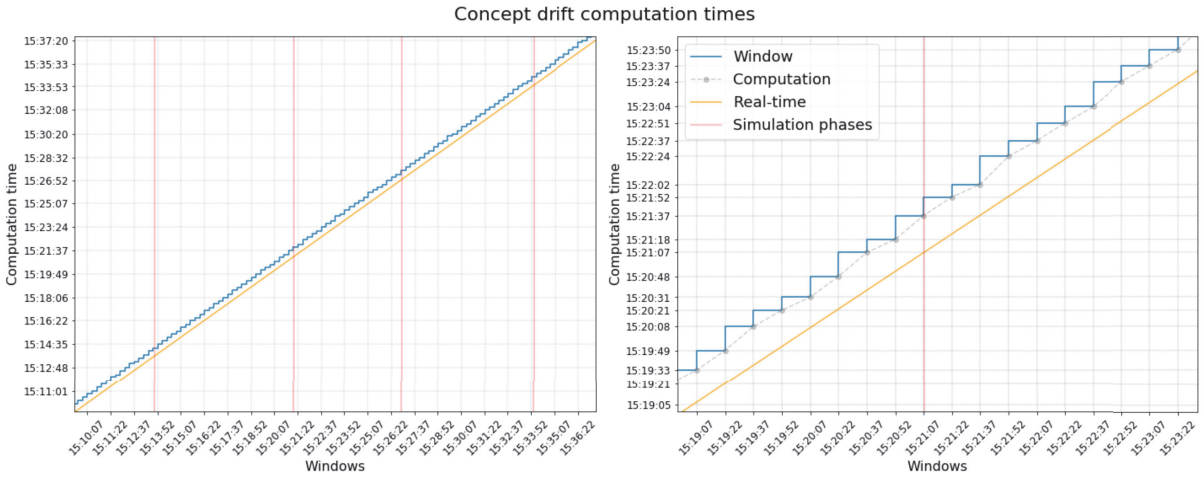

Figure 6.5.2: Data drift detection computation times.

based, they are computed over the windowed statistics. Therefore, drift detection latencies are expected to be at least as high as those of the statistics computation. Similarly, these latencies can be reduced by decreasing the window length or watermark delay when computing the statistics, as explained in the previous section.

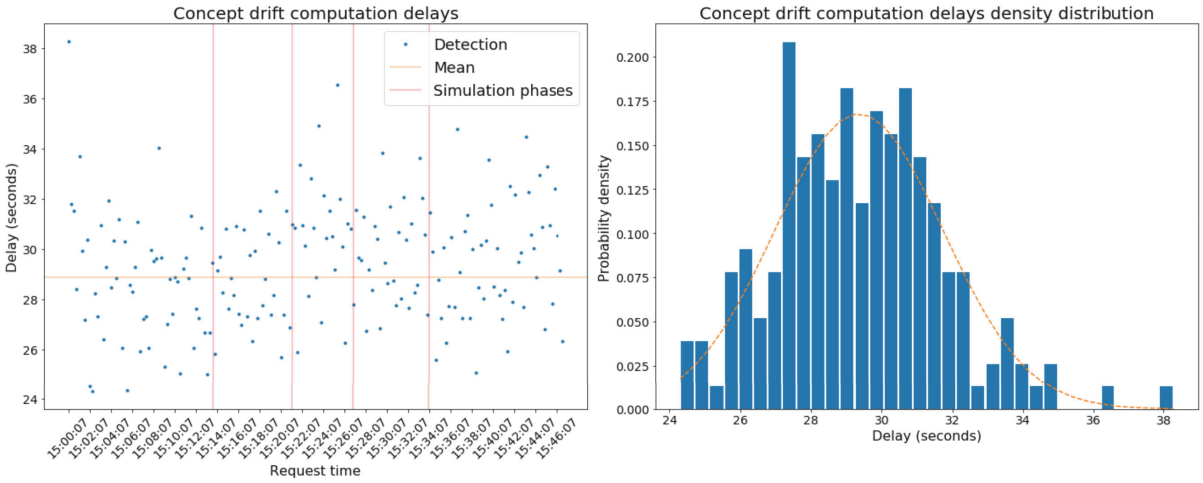

Figure 6.5.3: Delays between window times and concept drift detection.

Figure 6.5.2 shows the computation times over their corresponding windows, and compares them with real time. Again, the mid-point of the window is taken as reference for the comparisons, hence real delays can differ in +/- 7.5 seconds. In this case, delays closely follow a normal distribution with minimum and maximum values of 24.30 and 38.27 seconds, as shown in Figure 6.5.3. Also, the bulk of computations are conducted within 31 seconds after request time. More concretely, percentiles 25_{th}, 50_{th} and 75_{th} correspond to 27.63, 29.28 and 30.86 seconds, respectively.

Chapter 7

Conclusions

This section concludes the work with a general summary of the architecture performance and the resulting inference analysis, followed by limitations worth highlighting, a retrospective on the research question and future work.

7.1 Overview of experimentation results

In terms of scalability, either the Inference Service and Inference Logger are able to auto-scale on demand. While the former reached a throughput and general latencies of 178 request-per-second (RPS) and 1.9 seconds (excluding momentary peaks), respectively, the latter adapted similarly, staying close with near 177 RPS and 64 milliseconds (90_{th} percentile). These momentary peaks occurred each time the system needed to up-scale, causing a 0.06% of failed requests and therefore forwarding less requests to the Inference Logger.

Regarding the Kafka cluster, it managed to ingest and deliver both inference logs and inference analysis with maximum incoming and outgoing byte rates of 236 and 293 KiB per second, respectively. The resulted total log size was 74.24 MiB.

As for inference analysis, regarding distance-based outliers detection, the 75% were performed within 5.9 seconds after request time, reaching up to 10 seconds mainly during previously mentioned latency peaks. On the other hand, concerning window-based statistics and distribution-based data drift detection over those statistics, the 75% were computed within 29.92 and 30.86 seconds after request time (taking window mid-points as reference), respectively. These latencies reached maximum values of

34.34 and 38.27 seconds.

Contrasting the inference analysis results with the feature streams generated for the experimentation, it is noticeable that these results effectively represent the statistical characteristics of the features along the different stages.

Given these results, it is possible to verify the scalability of the architecture under different demands as they were configured in the experimentation.

7.2 Limitations

Regarding scalability, both the Inference Service and Inference Logger can auto-scale on demand due to the capabilities provided by Knative serving for the deployment of server-less services. Conversely, although both Spark and Kafka operators are scalable solutions, they still do not provide auto-scaling capabilities at the time of this work. For instance, in Spark the number of executors has to be defined during its deployment.

In addition, there is a limit on the number of cluster nodes in which stability is officially guaranteed by Kubernetes. This limit refers to 5000 nodes at the time of this thesis.

Concerning cluster performance monitoring, tools used for this purpose tend to be resource consuming specially in terms of memory. In container-based solutions available resources are shared between containers. When containers exceed pre-defined memory limits or there are no more available resources in the cluster node, they are automatically terminated by Kubernetes (i.e pod mortality). For this reason, it might be worth configuring high memory-consuming tools such as Prometheus to use an external storage if possible. Another alternative is having dedicated memory-rich nodes and configure Kubernetes to deploy mentioned tools always in those nodes.

Finally, as mentioned in previous sections the scalability configuration of each component should be further analysed and adopted to real needs. Additional criteria in a real scenario could be acceptable delays in the detection of outliers or data drift, expected demand, desired number of instances per window or the balance between resource exploitation and acceptable latencies.

7.3 Retrospective on the research question

As for the goals established in the first chapter and pursued along the thesis, they can be considered achieved. First, the proposed architecture is scalable and cloud-native. The Model Monitoring framework has been implemented for inference statistics computation, and outliers and drift detection on top of Spark Structured Streaming. Therefore, model monitoring is performed in a streaming fashion. A Kubernetes operator named Model Monitoring Operator has been developed to simplify and automate the configuration and deployment of the different components. Lastly, the scalability and performance of the solution as well as the inference analysis has been detailed and evaluated.

Reflecting on the research question raised at the beginning, the architecture proposed in this work can be considered a satisfactory answer to the research question. By these means, it is an open-source scalable automated cloud-native solution for ML model serving and monitoring in a streaming fashion.

7.4 Future Work

Possibilities of extending this work are multiple and mainly relate to the Model Monitoring framework, Model Monitoring Operator and Spark Structured Streaming.

Concerning the framework, support for other types of variables than continuous was left for future work. Also, new outlier and drift detection algorithms can be added to the framework by extending the interfaces provided.

Additionally, the arrival of Spark 3.0 brings better integration with Kubernetes as well as maturity in already existent features that can be leveraged to improve the performance of the system and reduce latencies. Among these features are auto-scaling of executors or continuous triggers (i.e low-latency executions without micro-batch processing).

Finally, there are numerous potential improvements such as additional utilities for model deployment or active learning among other techniques. For instance, support for comparing the performance of two models on the fly can be used in model deployment techniques such as blue-green or A/B testing. In addition, a richer analysis

of the model behavior as well as detailed statistical properties of drifted data can be leveraged by active learning implementations to decide when to perform the next training step or which data use for this training.

Bibliography

[1] Adadi, A. and Berrada, M. “Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI)”. In: *IEEE Access* 6 (2018), pp. 52138–52160.

[2] Agache, Alexandru, Brooker, Marc, Iordache, Alexandra, Liguori, Anthony, Neugebauer, Rolf, Piwonka, Phil, and Popa, Diana-Maria. “Firecracker: Lightweight Virtualization for Serverless Applications”. In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[3] Agrawal, Shikha and Agrawal, Jitendra. “Survey on Anomaly Detection using Data Mining Techniques”. In: *Procedia Computer Science* 60 (Dec. 2015), pp. 708–713. DOI: 10.1016/j.procs.2015.08.220.

[4] Amazon. *Amazon Sagemaker*. URL: https://aws.amazon.com/es/sagemaker/ (visited on 05/01/2020).

[5] Amazon. *AWS EKS*. URL: https://aws.amazon.com/es/eks/ (visited on 05/01/2020).

[6] Amazon. *AWS Lambda*. URL: https://aws.amazon.com/es/lambda/ (visited on 05/01/2020).

[7] Bach, Sebastian, Binder, Alexander, Montavon, Grégoire, Klauschen, Frederick, Müller, Klaus-Robert, and Samek, Wojciech. “On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation”. In: *PLoS ONE* 10 (2015).

[8] Baena-García, Manuel, Campo-Ávila, José, Fidalgo-Merino, Raúl, Bifet, Albert, Gavald, Ricard, and Morales-Bueno, Rafael. “Early Drift Detection Method”. In: (Jan. 2006).

[9] Barredo Arrieta, Alejandro, Diaz Rodriguez, Natalia, Del Ser, Javier, Bennetot, Adrien, Tabik, Siham, Barbado González, Alberto, Garcia, Salvador, Gil-Lopez, Sergio, Molina, Daniel, Benjamins, V. Richard, Chatila, Raja, and Herrera, Francisco. “Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges toward Responsible AI”. In: *Information Fusion* (Dec. 2019). DOI: 10.1016/j.inffus.2019.12.012.

[10] *BentoML - Model serving made easy*. URL: https://github.com/bentoml/bentoml (visited on 05/01/2020).

[11] Bernstein, D. “Containers and Cloud: From LXC to Docker to Kubernetes”. In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.

[12] Bhattacharjee, A., Chhokra, A. D., Kang, Z., Sun, H., Gokhale, A., and Karsai, G. “BARISTA: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services”. In: *2019 IEEE International Conference on Cloud Engineering (IC2E)*. 2019, pp. 23–33.

[13] Bifet, Albert and Gavaldà, Ricard. “Learning from Time-Changing Data with Adaptive Windowing”. In: vol. 7. Apr. 2007. DOI: 10.1137/1.9781611972771.42.

[14] Carlini, N. and Wagner, D. “Towards Evaluating the Robustness of Neural Networks”. In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 39–57.

[15] Carreira, Joao. “A Case for Serverless Machine Learning”. In: 2018.

[16] Chae, MinSu, Lee, HwaMin, and Lee, Kiyeol. “A performance comparison of linux containers and virtual machines using Docker and KVM”. In: *Cluster Computing* 22 (Jan. 2019). DOI: 10.1007/s10586-017-1511-2.

[17] Chakraborty, Anirban, Alam, Manaar, Dey, Vishal, Chattopadhyay, Anupam, and Mukhopadhyay, Debdeep. “Adversarial Attacks and Defences: A Survey”. In: *ArXiv* abs/1810.00069 (2018).

[18] Chandola, Varun and Kumar, Vipin. “Outlier Detection : A Survey”. In: *ACM Computing Surveys* 41 (Jan. 2009).

[19] community, Etcd. *Etcd Docs*. URL: https://etcd.io/ (visited on 05/01/2020).

[20] community, Istio. *Istio Docs*. URL: https://istio.io/latest/docs/ (visited on 05/01/2020).

[21] Docker. *Docker Swarm*. URL: https://docs.docker.com/engine/swarm/ (visited on 05/01/2020).

[22] Elastic. *ElasticSearch*. URL: https://www.elastic.co (visited on 05/01/2020).

[23] Eyk, Erwin van, Iosup, Alexandru, Abad, Cristina L., Grohmann, Johannes, and Eismann, Simon. "A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ICPE '18. Berlin, Germany: Association for Computing Machinery, 2018, pp. 21–24. ISBN: 9781450356299. DOI: 10.1145/3185768.3186308. URL: https://doi.org/10.1145/3185768.3186308.

[24] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. "An updated performance comparison of virtual machines and Linux containers". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015, pp. 171–172.

[25] Foundation, Apache Software. *Apache Kafka*. URL: https://kafka.apache.org/ (visited on 05/01/2020).

[26] Foundation, Apache Software. *Apache Spark*. URL: https://spark.apache.org/docs/2.4.5/structured-streaming-programming-guide.html (visited on 05/01/2020).

[27] Foundation, The Linux. *CloudEvents*. URL: https://cloudevents.io/ (visited on 05/01/2020).

[28] Foundation, The Linux. *Prometheus*. URL: https://prometheus.io/ (visited on 05/01/2020).

[29] Frías-Blanco, I., Campo-Ávila, J. d., Ramos-Jiménez, G., Morales-Bueno, R., Ortiz-Díaz, A., and Caballero-Mota, Y. "Online and Non-Parametric Drift Detection Methods Based on Hoeffding's Bounds". In: *IEEE Transactions on Knowledge and Data Engineering* 27.3 (2015), pp. 810–823.

[30] Gama, João, Medas, Pedro, Castillo, Gladys, and Rodrigues, Pedro. “Learning with Drift Detection”. In: vol. 8. Sept. 2004, pp. 286–295. DOI: 10.1007/978-3-540-28645-5_29.

[31] Gama, João, Žliobaitundefined, Indrundefined, Bifet, Albert, Pechenizkiy, Mykola, and Bouchachia, Abdelhamid. “A Survey on Concept Drift Adaptation”. In: *ACM Comput. Surv.* 46.4 (Mar. 2014). ISSN: 0360-0300. DOI: 10.1145/2523813. URL: https://doi.org/10.1145/2523813.

[32] Ghanta, Sindhu, Subramanian, Sriram, Khermosh, Lior, Shah, Harshil, Goldberg, Yakov, Sundararaman, Swaminathan, Roselli, Drew S., and Talagala, Nisha. “MPP: Model Performance Predictor”. In: *OpML*. 2019.

[33] Ghanta, Sindhu, Subramanian, Sriram, Khermosh, Lior, Sundararaman, Swaminathan, Shah, Harshil, Goldberg, Yakov, Roselli, Drew S., and Talagala, Nisha. “ML Health: Fitness Tracking for Production Models”. In: *ArXiv* abs/1902.02808 (2019).

[34] Google. *Google AI Platform*. URL: https://cloud.google.com/ai-platform (visited on 05/01/2020).

[35] Google. *Google Functions Documentation*. URL: https://cloud.google.com/functions/docs (visited on 05/01/2020).

[36] Google. *Google Kubernetes Engine*. URL: https://cloud.google.com/kubernetes-engine (visited on 05/01/2020).

[37] Google. *KFServing docs*. URL: https://github.com/kubeflow/kfserving/tree/master/docs (visited on 05/01/2020).

[38] Google. *KFServing github repository*. URL: https://github.com/kubeflow/kfserving (visited on 05/01/2020).

[39] Google. *Kubeflow - A Machine Learning toolkit for Kubernetes*. URL: https://www.kubeflow.org/ (visited on 05/01/2020).

[40] Google. *Kubernetes*. URL: https://kubernetes.io/es/ (visited on 05/01/2020).

[41] Gözüaçundefinedk, Ömer, Büyükçakundefinedr, Alican, Bonab, Hamed, and Can, Fazli. "Unsupervised Concept Drift Detection with a Discriminative Classifier". In: *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. CIKM '19. Beijing, China: Association for Computing Machinery, 2019, pp. 2365–2368. ISBN: 9781450369763. DOI: 10.1145/3357384.3358144. URL: https://doi.org/10.1145/3357384.3358144.

[42] Gupta, M., Gao, J., Aggarwal, C. C., and Han, J. "Outlier Detection for Temporal Data: A Survey". In: *IEEE Transactions on Knowledge and Data Engineering* 26.9 (2014), pp. 2250–2267.

[43] Hodge, Victoria. "A Survey of Outlier Detection Methodologies". In: *Artificial Intelligence Review* 22 (Oct. 2004), pp. 85–126. DOI: 10.1023/B:AIRE.0000045502.10941.a9.

[44] Hoens, T., Polikar, Robi, and Chawla, Nitesh. "Learning from streaming data with concept drift and imbalance: An overview". In: *Progress in Artificial Intelligence* 1 (Apr. 2012). DOI: 10.1007/s13748-011-0008-0.

[45] IBM. *IBM Cloud Functions Documentation*. URL: https://cloud.ibm.com/docs/openwhisk (visited on 05/01/2020).

[46] Inoubli, Wissem, Aridhi, Sabeur, Mezni, Haithem, Maddouri, Mondher, and Mephu Nguifo, Engelbert. "A Comparative Study on Streaming Frameworks for Big Data". In: Aug. 2018.

[47] Ishakian, V., Muthusamy, V., and Slominski, A. "Serving Deep Learning Models in a Serverless Platform". In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 2018, pp. 257–262.

[48] JetStack. *Cert-Manager*. URL: https://cert-manager.io/ (visited on 05/01/2020).

[49] Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., and Markl, V. "Benchmarking Distributed Stream Data Processing Systems". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 1507–1518.

[50] Khamassi, Imen, Sayed Mouchaweh, M., Hammami, Moez, and Ghe´dira, Khaled. “Self-Adaptive Windowing Approach for Handling Complex Concept Drift”. In: *Cognitive Computation* (June 2015). DOI: 10.1007/s12559-015-9341-0.

[51] Kolhe, S. and Dhage, S. “Comparative study on Virtual Machine Monitors for cloud”. In: *2012 World Congress on Information and Communication Technologies*. 2012, pp. 425–430.

[52] Koychev, Ivan and Lothian, Robert. “Tracking Drifting Concepts by Time Window Optimisation”. In: Jan. 2006, pp. 46–59. DOI: 10.1007/978-1-84628-226-3_5.

[53] Lazarescu, Mihai, Venkatesh, Svetha, and Bui, Hung Hai. “Using multiple windows to track concept drift”. In: *Intell. Data Anal.* 8 (2004), pp. 29–59.

[54] Logicalclocks. *Hopsworks - A Data-Intensive AI platoform with a Feature Store*. URL: https://github.com/logicalclocks/hopsworks (visited on 05/01/2020).

[55] Lopez Garcia, Alvaro, Tran, Viet, Alic, Andy, Caballer, Miguel, Campos Plasencia, Isabel, Costantini, Alessandro, Dlugolinsky, Stefan, Duma, Doina, Donvito, Giacinto, Gomes, Jorge, Cacha, Ignacio, Marco de Lucas, Jesus, Ito, Keiichi, Kozlov, Valentin, Nguyen, Giang, Orviz Fernandez, Pablo, Sustr, Zdenek, Wolniewicz, Pawel, Antonacci, Marica, and Plociennik, Marcin. “A cloud-based framework for machine learning workloads and applications”. In: *IEEE Access* PP (Jan. 2020), pp. 1–1. DOI: 10.1109/ACCESS.2020.2964386.

[56] Lu, J., Liu, A., Dong, F., Gu, F., Gama, J., and Zhang, G. “Learning under Concept Drift: A Review”. In: *IEEE Transactions on Knowledge and Data Engineering* 31.12 (2019), pp. 2346–2363.

[57] Madhavapeddy, Anil, Mortier, Richard, Rotsos, Charalampos, Scott, David, Singh, Balraj, Gazagnaire, Thomas, Smith, Steven, Hand, Steven, and Crowcroft, Jon. “Unikernels: Library Operating Systems for the Cloud”. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 461–472. ISBN: 9781450318709. DOI: 10.1145/2451116.2451167. URL: https://doi.org/10.1145/2451116.2451167.

[58] Marcu, Ovidiu, Costan, Alexandru, Antoniu, Gabriel, and Pérez, María. “Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks”. In: Sept. 2016, pp. 433–442. DOI: 10.1109/CLUSTER.2016.22.

[59] Microsoft. *Azure Functions Documentation*. URL: https://docs.microsoft.com/en-us/azure/azure-functions/ (visited on 05/01/2020).

[60] Microsoft. *Azure Kubernetes Service*. URL: https://azure.microsoft.com/es-es/services/kubernetes-service/ (visited on 05/01/2020).

[61] Microsoft. *Azure Machine Learning*. URL: https://azure.microsoft.com/es-es/services/machine-learning/ (visited on 05/01/2020).

[62] OpenStack. *Welcome to OpenStack Documentation*. URL: https://docs.openstack.org/ussuri/?_ga=2.150817654.1928316480.1592218952-1618055762.1592218952 (visited on 05/01/2020).

[63] Ozdag, Mesut. “Adversarial Attacks and Defenses Against Deep Neural Networks: A Survey”. In: *Procedia Computer Science* 140 (2018), pp. 152–161.

[64] Pahl, C. “Containerization and the PaaS Cloud”. In: *IEEE Cloud Computing* 2.3 (2015), pp. 24–31.

[65] Pinto, Fábio, Sampaio, Marco O. P., and Bizarro, Pedro. *Automatic Model Monitoring for Data Streams*. 2019. arXiv: 1908.04240 [cs.LG].

[66] Quiñonero-Candela, J., Sugiyama, M., Schwaighofer, A., and Lawrence, N. D. “When Training and Test Sets Are Different: Characterizing Learning Transfer”. In: *Dataset Shift in Machine Learning*. 2009, pp. 3–28.

[67] Ribeiro, Marco Tulio, Singh, Sameer, and Guestrin, Carlos. ““Why Should I Trust You?”: Explaining the Predictions of Any Classifier”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1135–1144. ISBN: 9781450342322. DOI: 10.1145/2939672.2939778. URL: https://doi.org/10.1145/2939672.2939778.

[68] Ribeiro, Marco Tulio, Singh, Sameer, and Guestrin, Carlos. *Anchors: High-Precision Model-Agnostic Explanations*. 2018. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16982.

[69] Rubner, Yossi, Tomasi, Carlo, and Guibas, Leonidas. “The Earth Mover’s Distance as a metric for image retrieval”. In: *International Journal of Computer Vision* 40 (Jan. 2000), pp. 99–121.

[70] Schelter, Sebastian, Grafberger, Stefan, Schmidt, Philipp Martin, Rukat, Tammo, Kiessling, Mario, Taptunov, Andrey, Biessmann, Felix, and Lange, Dustin. “Deequ-Data Quality Validation for Machine Learning Pipelines”. In: 2018.

[71] Sculley, D., Holt, Gary, Golovin, Daniel, Davydov, Eugene, Phillips, Todd, Ebner, Dietmar, Chaudhary, Vinay, Young, Michael, Crespo, Jean-François, and Dennison, Dan. “Hidden Technical Debt in Machine Learning Systems”. In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett. Curran Associates, Inc., 2015, pp. 2503–2511. URL: http://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf.

[72] Seldon. *Alibi-detect*. URL: https://github.com/SeldonIO/alibi-detect (visited on 05/01/2020).

[73] Seldon. *StreamDM*. URL: https://github.com/huawei-noah/streamDM (visited on 05/01/2020).

[74] Shahrad, Mohammad, Balkind, Jonathan, and Wentzlaff, David. “Architectural Implications of Function-as-a-Service Computing”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 1063–1075. ISBN: 9781450369381. DOI: 10.1145/3352460.3358296. URL: https://doi.org/10.1145/3352460.3358296.

[75] Shahrad, Mohammad, Fonseca, Rodrigo, Goiri, Iñigo, Chaudhry, Gohar, Batum, Paul, Cooke, Jason, Laureano, Eduardo, Tresness, Colby, Russinovich, Mark, and Bianchini, Ricardo. “Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider”. In: *ArXiv* abs/2003.03423 (2020).

[76] Shannon, C. E. “A mathematical theory of communication”. In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423.

[77] Shopify. *Sarama, a Go library for Apache Kafka*. URL: https://github.com/Shopify/sarama (visited on 05/01/2020).

[78] Silva, Vitor, Kirikova, Marite, and Alksnis, Gundars. “Containers for Virtualization: An Overview”. In: *Applied Computer Systems* 23 (May 2018), pp. 21–27. DOI: 10.2478/acss-2018-0003.

[79] Talbot, Joshua, Pikula, Przemek, Sweetmore, Craig, Rowe, Samuel, Hindy, Hanan, Tachtatzis, Christos, Atkinson, Robert, and Bellekens, Xavier. *A Security Perspective on Unikernels*. Nov. 2019.

[80] Wang, Liang, Li, Mengyuan, Zhang, Yinqian, Ristenpart, Thomas, and Swift, Michael. “Peeking behind the Curtains of Serverless Platforms”. In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC ’18. Boston, MA, USA: USENIX Association, 2018, pp. 133–145. ISBN: 9781931971447.

[81] Webb, Geoffrey, Hyde, Roy, Cao, Hong, Nguyen, Hai-Long, and Petitjean, François. “Characterizing Concept Drift”. In: *Data Mining and Knowledge Discovery* 30 (Nov. 2015). DOI: 10.1007/s10618-015-0448-4.

[82] Xie, Haochen. “Principles, patterns, and techniques for designing and implementing practical fluent interfaces in Java”. In: Oct. 2017, pp. 45–47. DOI: 10.1145/3135932.3135948.

[83] Zhang, Chengliang, Yu, Minchen, Wang, Wei, and Yan, Feng. “MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving”. In: *USENIX Annual Technical Conference*. 2019.

Appendix - Contents

Appendix A

Model Monitoring Job configuration

Table A.0.1: Model information. MODEL_INFO environment variable fields

Field	Description	Type	Required
name	Model name	String	X
id	Model id	String	
version	Model version	Int	

Table A.0.2: Trigger configuration. A field of MONITORING_CONFIG environment variable.

Field	Description	Type	Required
trigger.window	Window configuration	Object	X
trigger.window.duration	Window duration	Int	X
trigger.window.slide	Window slide	Int	X
trigger.window.watermarkDelay	Window watermark	Int	X

Table A.0.3: Stats definitions. A field of MONITORING_CONFIG environment variable.

Field	Description	Type	Required
max	Max statistic	Empty Object	
min	Min statistic	Empty Object	
count	Count statistic	Empty Object	
distr	Feature distribution	Empty Object	
avg	Average statistic	Empty Object	
mean	Mean statistic	Empty Object	
stddev	Standard deviation statistic	Object	
stddev.type	Population or sample	String	
perc	Percentiles definition	Object	
perc.percentiles	Percentile values	Array[String]	
perc.iqr	Interquartile range	Boolean	
cov	Covariance statistic	Object	
cov.type	Population or sample	String	
corr	Correlation statistic	Object	
corr.type	Population or sample	String	

Table A.0.4: Outliers detectors. A field of MONITORING_CONFIG environment variable.

Field	Description	Type	Required
outliers.descriptive	Outliers based on descriptive stats. List of stats names.	Array[String]	

Table A.0.5: Concept drift detectors. A field of MONITORING_CONFIG environment variable.

Field	Description	Type	Required
drift.wasserstein	Wasserstein distance	Object	
drift.wasserstein.threshold	Threshold value	Double	
drift.wasserstein.showAll	Obviate the threshold and include all distances.	Boolean	
drift.kullbackLeibler	Kullback-Leibler divergence	Object	
drift.kullbackLeibler.threshold	Threshold value	Double	
drift.kullbackLeibler.showAll	Obviate the threshold and include all distances.	Boolean	
drift.jensenShannon	Jensen-Shannon divergence	Object	
drift.jensenShannon.threshold	Threshold value	Double	
drift.jensenShannon.showAll	Obviate the threshold and include all distances.	Boolean	

Table A.0.6: Baseline statistics. A field of MONITORING_CONFIG environment variable.

Field	Description	Type	Required
baseline.descriptive	Descriptive stats	Map[Map[String, Double]]	
baseline.distributions	Features distributions	Map[Map[String, Double]]	

Table A.0.7: Storage configuration. STORAGE_CONFIG environment variable fields.

Field	Description	Type	Required
inference	Data source specification	Object	X
inference.kafka	Kafka configuration	Object	X
analysis	Sinks specification	Object	X
analysis.stats	Sink for statistics	Object	X
analysis.stats.kafka	Kafka configuration	Object	X
analysis.outliers	Sink for outliers	Object	
analysis.outliers.kafka	Kafka configuration	Object	
analysis.drift	Sink for drift detection	Object	
analysis.drift.kafka	Kafka configuration	Object	

Table A.0.8: Kafka configuration. A field include multiple times in STORAGE_CONFIG environment variable.

Field	Description	Type	Required
brokers	Comma-separated bootstrap servers endpoints	String	X
topic	Kafka topic configuration	Object	X
topic.name	Topic name	String	X
topic.partitions	Topic partitions	Int	
topic.replicationFactor	Topic replication factor	Int	

Table A.0.9: Job configuration. JOB_CONFIG environment variable fields.

Field	Description	Type	Required
timeout	Job timeout in seconds	Int	