Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Kafka-ML: Connecting the data stream with ML/AI frameworks

Cristian Martín [a,*], Peter Langendoerfer [b,c], Pouya Soltani Zarrin [b], Manuel Díaz [a], Bartolomé Rubio [a]

[a] *ITIS Software, University of Málaga, Málaga, Spain*
[b] *IHP - Leibniz-Institut für Innovative Mikroelektronik, Frankfurt (Oder), Germany*
[c] *Brandenburg University of Technology Cottbus-Senftenberg, Cottbus, Germany*

A R T I C L E   I N F O

Article history:
Received 29 October 2020
Received in revised form 27 July 2021
Accepted 28 July 2021
Available online 5 August 2021

Keywords:
Kafka-ML
Apache Kafka
Machine Learning
Artificial Intelligence
Data streams
Docker
Kubernetes
Distributed systems

A B S T R A C T

Machine Learning (ML) and Artificial Intelligence (AI) depend on data sources to train, improve, and make predictions through their algorithms. With the digital revolution and current paradigms like the Internet of Things, this information is turning from static data to continuous data streams. However, most of the ML/AI frameworks used nowadays are not fully prepared for this revolution. In this paper, we propose Kafka-ML, a novel and open-source framework that enables the management of ML/AI pipelines through data streams. Kafka-ML provides an accessible and user-friendly Web user interface where users can easily define ML models, to then train, evaluate, and deploy them for inferences. Kafka-ML itself and the components it deploys are fully managed through containerization technologies, which ensure their portability, easy distribution, and other features such as fault-tolerance and high availability. Finally, a novel approach has been introduced to manage and reuse data streams, which may eliminate the need for data storage or file systems.

1. Introduction

In this digital era, information is continuously acquired and processed everywhere, from many sources and for many purposes and sectors. In this sense, Machine Learning (ML) and Artificial Intelligence (AI) [1] are playing a decisive role in converting raw information into useful predictions and recommendations to improve fields such as business operations and the overall life of citizens. For instance, companies like Facebook [2] process millions of photos every day to detect inappropriate contents. This creates a continuous *data stream* for ML/AI algorithms and systems to face.

More recently, with the rise of the Internet of Things (IoT) [3], new sources of data have been enabled in the Internet era, with a forecast of 500 billion connected devices by 2030 [4]. Paradigms such as Industry 4.0, connected cars, and smart cities have become a possibility and, more importantly, they have contributed to the digitization of services in the physical world. As a result, the data stream has been increasing continuously, and forecasts predict a huge expansion for the upcoming years.

Traditionally, most of the ML/AI frameworks, which are behind the design and development of ML/AI algorithms, have been designed to work not with data streams, but with persistent datasets and static data. Even nowadays, popular Python frameworks [5] such as PyTorch, Theano, and TensorFlow provide, at the most, only partial support for data stream systems like Apache Kafka [6], the most popular data stream system. This does not merely include training of ML models, but also the rest of the steps that may be part of an ML/AI pipeline, such as ML model comparison and inference for production environments. The importance of ML model management in real-world and large-scale ML/AI pipelines has been addressed in the literature [7,8]. Current systems focus mainly on model definition and training, while other steps of ML/AI pipelines – such as model sharing, data management, and life cycle management – are largely ignored [9]. And these, especially the ML code, are only a tiny fraction of the real-world ML/AI applications [10]. Building an ML model for enterprise and real-world applications is an iterative process. ML developers and data scientists may evaluate hundreds of ML models before identifying one that meets acceptance criteria. Therefore, designing and building robust processes that facilitate this procedure would accelerate the release of ML/AI applications. In the era of Deep Learning [11], large ML models and complex patterns, this process is even more necessary since features are less human-based and more learned in an automatic manner. In the ML/AI pipeline, the management of the data life cycle [12] also plays an essential role to ensure the success of ML/applications, since the accuracy of an ML model is deeply tied to the data that is trained on. Although there are open-source solutions for

[*] Corresponding author.
E-mail address: cmf@lcc.uma.es (C. Martín).

https://doi.org/10.1016/j.future.2021.07.037

managing ML/AI pipelines such as Kubeflow [13] and NVIDIA DIGITS [14], and enterprise products such as Amazon SageMaker [15], Algorithmia [16], and Valohai [17], in general, they are not yet ready to work with data streams (or directly) and do not provide adequate management of data stream flows in ML/AI applications.

Furthermore, we aim at making ML and AI open and accessible for everyone. In this sense, AutoML initiatives such as Google Cloud AutoML [18] have contributed to high-quality models and solutions that developers with limited experience can adopt to meet multiple business needs. However, these initiatives have not yet been properly integrated with data stream pipelines. Another means to pave the way of users into ML and AI is to provide a missing ecosystem where they can share trained models and metrics (e.g., loss and accuracy) that can also be used to evaluate different models and configurations.

Finally, ML/AI solutions usually need requirements that are difficult to be accomplished in personal computers, and developers tend to adopt shared infrastructures to deploy their applications. Moreover, high availability, load balancing, and fault tolerance may be required in ML/AI mission-critical applications and should be provided in a transparent way to users. In this context, state-of-the-art containerization and container orchestration platforms can be exploited to distribute the load of a system and its components, in addition to providing fault tolerance and high availability in production systems.

Let us suppose that a company requires the continuous development of ML/AI models with the ingestion of heterogeneous and dynamic data stream sources for its business logic (e.g., from the IoT). In this case, the company could face the following challenges:

1. How can current ML/AI frameworks and their pipelines be integrated with continuous data streams?
2. How can an accessible and collaborative tool be achieved to evaluate and compare ML/AI model metrics and results?
3. How can data streams be reused and combined in ML/AI tasks?
4. How can a solution be applied to portable and high availability architectures for production deployments?

To cope with the aforementioned challenges, Kafka-ML,[1] a novel and open-source framework for the management of ML/AI pipelines through data streams, is presented here. The main innovation of this work is to reduce the gap between data streams and current ML/AI frameworks, providing an open-source framework to harmonize their full integration. Kafka-ML currently supports TensorFlow as ML framework to integrate data streams and ML/AI. However, our goal is to extend the support for other ML/AI frameworks in the near future. Kafka-ML offers an accessible and user-friendly Web user interface (following an approach similar to AutoML initiatives [19]) to manage ML/AI pipelines for both experts and non-experts on ML/AI. It enables users to train, compare and infer their ML algorithms by simply writing some lines of code. Moreover, this framework makes use of a novel approach to manage data streams, which can be reused as many times as they are configured, leading to eliminate the need for any data storage or file system for datasets in Kafka-ML.

Therefore, the main contributions of this paper are:

1. Presentation of Kafka-ML, a novel, open-source, accessible, and user-friendly framework to manage ML/AI pipelines with AutoML features through data streams.
2. Adoption of container technologies which enable portability, fault tolerance, and high availability in Kafka-ML.
3. A novel approach to manage data streams for ML/AI pipelines with no need for data storage or file systems.

[1] https://github.com/ertis-research/kafka-ml.

The rest of the paper is organized as follows: Section 2 presents a background study on Kafka-ML and related work is discussed in Section 3. In Section 4, the Kafka-ML architecture and its components are presented. Then, in Section 5, the ML/AI pipeline of Kafka-ML is introduced through an example. An evaluation of Kafka-ML is performed in Section 6. A discussion of how Kafka-ML has addressed the research challenges is presented in Section 7. Lastly, our conclusions and future work are presented in Section 8.

2. Background

2.1. Apache Kafka

Apache Kafka is a distributed messaging system (publish/subscribe) that can dispatch and consume large amounts of data at low latency. Traditional message queues can support high rates of message consumption by adding multiple consumers per topic, but only one consumer will receive each message at a time. Like message queues, publish/subscribe systems transmit information from producers to consumers. Nevertheless, in contrast to message queues, publish/subscribe systems allow multiple consumers to receive each message in a topic. Nowadays in the era of big data, data stream goes to multiple systems such as batch and stream processing, while necessitating a low latency. In order to satisfy both requirements, Apache Kafka provides the following features:

- Multi-customer distribution. As a publish/subscribe system, Apache Kafka enables the connection of multiple clients and customers to messages. Moreover, thanks to its integration with a wide range of solutions, such as Apache Hadoop, Apache Storm, TensorFlow, etc., this feature is highly practical.
- High rate of message dispatching. This is achieved by a conjunction of functionalities: (1) message set abstractions: messages are grouped together amortizing the overhead of the network round trip rather than sending a single message at a time; (2) binary message format: data chunks can be transferred without modifications; and (3) zero-copy optimizations to avoid many copies of the pagecache. However, one of its most notable features is the Kafka consumer group, which enables the distribution of messages in a cluster of customers managed by Apache Kafka, like message queues.

Topics are the stream of messages in Kafka, wherein producers can publish messages and consumers can subscribe to receive them. When a message is sent by a producer to Kafka, as opposed to many distributed queue frameworks, Kafka stores it in disk with a configurable retention policy, enabling later data retrieving by components. This is popularly known as the *distributed log* [20], which enables consumers to go through the log when required. In some cases, like ML training in Kafka-ML, this feature is suitable since all data are processed at once. Whether a failure occurs during this process, the customer can restart without losing any data or having to store them in a file system.

Load balancing and fault tolerance are also performed by partitions of the topics, where each topic can be divided into multiple partitions and each partition can have multiple replicas. Partitions allow the log to be divided into smaller units and to provide load balancing, while topic replicas enable fault tolerance. An Apache Kafka cluster consists in a peer-to-peer network of Brokers that share partitions and replicas. When having a consumer group,

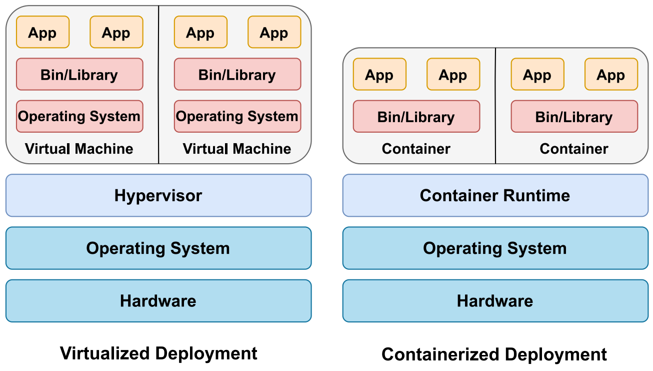


Fig. 1. Virtualization vs containerization.

partitions can be associated to customers enabling high dispatching data rates. Apache Kafka also incorporates different policies such as "at most one", "at least once", and "exactly one", which enable customized Quality of Service (QoS) policies for message dispatching.

Its popularity, its large number of implementations and integrations with many cloud computing systems, as well as its great acceptance in the community, have transformed Apache Kafka into the standard approach to interconnecting systems, ingesting data, and dispatching information.

In this work, Apache Kafka is used as a distributed messaging system to distribute received data streams (e.g., from the IoT) to the Kafka-ML framework for further processing in ML/AI pipelines (e.g., training and inference tasks). Its distributed features enable the Kafka-ML framework to scale up as required depending on the data stream consumption and production. Moreover, its guaranties regarding fault tolerance allow Kafka-ML to have a proper and reliable control of the data streams received.

2.2. Containerization and container orchestration platforms

Lightweight virtualization technologies such as containers have enabled a way of scaling and reallocating components, services, and applications. According to Pahl [21], containerization is "a technology to virtualize applications in a lightweight way that has resulted in a significant uptake in cloud applications management". Due to their lightweight nature, it is possible to install containers in a huge range of systems, including IoT devices, as demonstrated by our portable Fog infrastructure with Raspberry Pis [22]. Compared to traditional virtualization approaches, containers are similar to virtual machines, except they do not require a hypervisor to work with, as they only need a container runtime. Another important distinction is that containers are decoupled from the underlying infrastructure, so they are easily portable across clouds and Operating System (OS) distributions. Fig. 1 clarifies differences between virtualization and containerization.

When talking about containerization, it is always mandatory to mention Docker. Docker [23] is an open-source project that automatizes the application deployment inside containers. Its runtime, the Docker Engine, is the standard container runtime in the industry and enables its execution on multiple architectures (ARM, AMD64, x86). Moreover, Docker Engine allows containerized applications to specify what they need and how they will run, reducing dependency and installing problems for developers and deployment teams.

A cluster of machines running containers is usually managed through orchestration systems (like Docker Swarm or Kubernetes). These systems allow fault tolerance and high availability by scaling vertically (in federation mode) and horizontally. Kubernetes [24] is an open-source platform for managing containerized workloads and services in a cluster of nodes or servers, easing both automation and declarative configuration. Kubernetes enables continuous monitoring of containers (e.g., Docker) and their replicas to ensure that they continuously match the status defined for them, in addition to allowing other features for production environments such as high availability and load balancing.

In this work, all the components comprising the Kafka-ML framework and its dependencies, such as Apache Kafka, are containerized through Docker containers, offering a lightweight and portable solution to be distributed in container platforms. Kubernetes has been adopted to harmonize the deployment of Docker containers and enable the Kafka framework to be deployed with high availability and fault tolerance features in production environments.

3. Related work

To the best of our knowledge, Kafka-ML is the first open-source framework to provide an ML/AI pipeline solution to integrate ML/AI and data streams. Nevertheless, other approaches have similar goals or have provided some of the functionalities offered by Kafka-ML as described below.

Kubeflow [13] is a powerful ML toolkit for Kubernetes. In Kubeflow, users can configure multiple steps of an ML/AI pipeline such as hyper-parameters, pre-processing, training and inference. However, when running a Kubeflow pipeline, such as the official example for the Google Cloud Platform,[2] there may be some steps that are not required in the Kafka-ML pipeline, especially the ones that require building containers for training, and inference. In Kafka-ML, users merely need to interact with the Web UI (User Interface) for training and inference. In addition, data stream support has to be manually developed by Kubeflow ML developers and users. In Kafka-ML, the data stream management through Apache Kafka is supported in all of the pipeline. Kubeflow provides great support for Kubernetes and ML multi-frameworks, which are supported by a large ecosystem and community that are far beyond the scope and functionalities offered by Kafka-ML. Therefore, it may be worth studying the way of integrating both systems in the near future.

NVIDIA Deep Learning GPU Training System (DIGITS) [14] provides an interactive Web UI for training and inference of deep neural networks (DNNs) on multi-GPU systems. Unlike Kafka-ML, DIGITS is not a framework in itself, but a wrapper for NVCaffe, Torch, and TensorFlow, which provides a Web UI to these frameworks rather than dealing with them directly on the command-line. The main advantages of DIGITS are its native support for GPUs and three ML frameworks, the release of pre-trained models, and the functionality to see the accuracy and loss in real-time. Nevertheless, DIGITS does not support training and inference through data streams (datasets have to be imported instead) or the deployment of these tasks through containers for scaling, it has a dependency on GPUs and may require writing a source code on top of these frameworks.

Regarding enterprise solutions, Amazon SageMaker [15,25] is an ecosystem provided as part of Amazon Web Services (AWS) which offers a continuous integration and continuous delivery (CI/CD) service for ML/AI. Among its countless services, the following stand out: hyperparameter optimization, incremental training, and elastic (pausing and resuming) learning. SageMaker provides supports for ML frameworks such as Tensorflow, MXNet and Pytorch, and Amazon Kinesis helps with real-time data ingestion at scale. Although data streams can be seamlessly integrated

[2] https://www.kubeflow.org/docs/gke/gcp-e2e/.

for inference, for training they rely on data lakes such as Amazon S3, unlike Kafka-ML, an open-source project where both phases can be performed with data streams, without the need for data lakes. Despite the fact that Algoritmia [16] does not provide much information, it is apparently more focused on the delivery of models for inference and the automation of some pipeline processes and their monitoring. Valohai [17] enables the management of ML/AI pipelines in Kubernetes like Kubeflow. Valohai helps building complex ML/AI pipelines that can be automated including steps such as hyperparameter optimization. No information is provided regarding data streams. However, one of the most notable features of this platform is its automated and extensive version control for ML model traceability.

MOA [26] is a framework for online learning and data stream mining. MOA provides a graphical interface where users can execute and visualize ML tasks, including a collection of ML algorithms implementations for classification, regression, and clustering among others. Although Kafka-ML supports data streams, Kafka-ML and TensorFlow are not (yet) good at supporting online learning. On the other hand, Kafka-ML provides support for TensorFlow/Keras models and their large community, instead of creating a new framework with its own source code that could limit its adoption. Scikit-multiflow [27] is another framework for online learning, in this case for the popular framework scikit-learn, however it does not provide a Web interface or a full control of an ML/AI pipeline.

Ullah et al. [28] propose an ML-based system for real-time processing of data streams and action recognition. The data stream processing is performed in different ML steps for action recognition, including a pre-trained convolutional neural network for feature extraction, a deep autoencoder to learn temporal changes of actions, and a non-linear learning approach for classification. This solution also includes an online learning phase to continuously improve the model. This solution has been defined for action recognition and does not provide a generic framework for the deployment and management of ML/AI applications and data streams like Kafka-ML. Moreover, this solution lacks a distributed stream integration like Kafka-ML for better management and scalability of data streams.

Kafka-ML follows a different approach as compared to other distributed data stream frameworks [29,30] that are growing in the era of big data and data streams, such as Apache SAMOA [31], Apache Flink [32], Apache Spark and Spark Streaming [33], and the Lambda architecture [34,35]. Apache SAMOA is currently undergoing incubation at Apache and aims at enabling the development of ML algorithms through data streams without directly dealing with the complexity of underlying processing engines (e.g., Apache Storm [36] and Apache Samza [37]). Although without native support for data streams, SystemML [38] also follows a similar approach than SAMOA and provides a declarative ML language to abstract the development of ML/AI applications, which can be finally deployed on distributed systems such as Spark. Apache Flink provides a framework to perform computation over data streams at an in-memory speed and at any scale. Apache Spark is an engine for large-scale data processing, and Spark streaming is an extension of it for scalable and high-performance stream processing. In addition, the Lambda architecture allows the processing of large amounts of data in real time by having real-time and batch layers of processing. In general, these frameworks provide distributed engines for distributing any kind of computation with data streams, although they have limited support for, or do not have a special focus on, facilitating ML/AI pipelines and popular ML/AI frameworks such as TensorFlow and their large range of ML/AI solutions and community, as Kafka-ML does. Moreover, Kafka-ML can also enable the deployment of high availability and fault-tolerant ML/AI pipelines. MLlib [39], another extension of Apache Spark, supports ML/AI algorithms for common learning settings including regression, collaborative filtering and clustering in Spark deployments. Nevertheless, this support is limited for a number of common algorithms and does not support, for example, deep learning ML models such as Kafka-ML with TensorFlow.

Kafka Streams [40] is a Java library which allows building real-time processing applications with data streams allocated in Apache Kafka, i.e., the input and output of Kafka Streams applications are Kafka topics. Kafka Streams provides abstractions and operators (stateless and stateful) to work both with data streams and tables (data stream aggregations) for the development of streaming and microservice applications. Faust [41] is another open-source stream processing library which ports the ideas from Kafka Streams to Python. Like Kafka Stream, Faust provides support for data stream processing, sliding windows, and aggregate counts. Its interface is less verbose than Kafka Streams, and applications can be developed with very few lines of source code. While not directly supporting ML/AI applications, these streaming libraries rely on Apache Kafka as a distribution core as Kafka-ML does.

Despite its name, TensorFlow Serving [42] is an agnostic system to serve ML models in general and TensorFlow in particular. TensorFlow Serving provides a canonical Remote Procedure Call (RPC) server that serves ML models from a hosted service, where models can be uploaded and updated when required. Users have full control over the ML models submitted, and this allows to have multiple ML model versions simultaneously or rollback a version. This inference module was designed by Google TensorFlow for production environments. Regarding Kafka-ML, this module works with RPC requests instead of data streams, which can be fully controlled and reliably stored in the distributed log provided by Apache Kafka.

According to Assuncao et al. [43], we are facing the fourth generation of distributed data stream processing frameworks, where certain elements are located on the edge of the network to reduce latency. Calo et al. [44] propose a system for serving ML models on edge servers and optimizing IoT communications. Spanedge [45] is presented as a system to distribute stream processing across a central and some near-the-edge data centers. Going further, Pisani et al. [46] propose cross-platform code execution directly in IoT devices. The distribution of computation in all these layers, namely edge, fog, and cloud, what is known as the Cloud-to-Things continuum [47], will definitely optimize the execution of ML/AI applications, and Kafka-ML will deploy ML/AI applications in this continuum in future work.

MODELDB [7] was one of the first approaches in providing an open-source ML model management system, i.e., a system for ML model, metadata, and experiment versioning and management. All the information related to ML models (hyperparameters, type, author, and so on) can be uploaded to MODELDB to have full control of the experiments and optimizations carried out. ModelHub [9] provides a service for publishing, reusing and discovering Deep Learning models, such as a GitHub repository for software components. This is of special interest to have a better control of the developed ML models and to let users explore the ML model space through the network architecture and hyperparameter values. These projects are basically a metadata storage (or a git system) to track information about ML models and experiments, thus they do not manage or deploy any step of ML/AI pipelines like Kafka-ML. Although Kafka also allows obtaining information about the performance of ML models (automatically), it will take ideas from these projects to improve the information provenance and to have a better control of ML model versions.

Finally, Kafka-ML is related to some extent to AutoML projects such as OpenML [48], Bazaar [49], and Google Cloud AutoML [18].

Table 1
Overview of existing technical approaches.

Ref.	ML/AI pipeline management	Data streams	AutoML	Model metrics comparison	Popular ML framework integration	Open-source project
Kubeflow [13]	✓		✓	✓	✓	✓
DIGITS [14]	✓			✓	✓	✓
Amazon SageMaker [25]	✓	†	✓	✓	✓	
Algoritmia [16]	†	*		*	✓	
Valohai [17]	✓	*	✓	✓	✓	
MOA [26]		✓				✓
Scikit-multiflow [27]		✓		✓	✓	✓
[28]		✓				
Apache SAMOA [31]		✓				✓
Apache flink [32]		✓			✓	✓
Apache spark streaming [33]		✓			✓	✓
Lambda architecture [35]		✓				✓
Apache Samza [37]		✓			✓	✓
SystemML [38]						
MLlib [39]		✓	✓			✓
Kafka Streams [40]		✓				✓
Faust [41]		✓			✓	✓
TensorFlow serving [42]	†				✓	✓
[44]	✓					
Spanedge [45]		✓				
MODELDB [7]				✓		✓
ModelHub [9]				✓		
OpenML [48]				✓		✓
Bazaar [49]			✓	✓		✓
Google cloud AutoML [18]	✓		✓		✓	
Kafka-ML	✓	✓	✓	✓	✓	✓

✓Approach has this feature   Approach has not this feature   †Approach has partially this feature   * Information not available.

OpenML is a web platform where users can openly share, upload, and explore results, scientific tasks, data analysis flows, and datasets. Results and metrics of ML models can also be shared and compared in Kafka-ML. Moreover, data streams can also be managed and shared, as we will see in Section 4.7. Bazaar provides an AutoML system that – through ML primitives and a specification for data processing – automates and facilitates tasks such as the selection of the learning algorithm and hyper parameter tuning. This is an interesting approach to reduce the complexity of such required steps in ML/AI applications. However, it is tailored to a specific solution. In Kafka-ML we have adopted a state-of-the-art and well-known ML framework like TensorFlow for ML definition. We will consider how to optimize these steps in ML/AI pipelines and frameworks supported by Kafka-ML in the near future. Google Cloud AutoML provides high-quality ML models with little effort and no advanced knowledge of the subject. Reaching the quality of these models is beyond the scope of Kafka-ML. However, Kafka-ML provides an accessible platform, where only a few lines of ML model source code are required to start an ML/AI pipeline with data streams. Furthermore, Kafka-ML is an open-source project available for both experts and non-experts on ML/AI.

To conclude this section, Table 1 summarizes and compares previous approaches with Kafka-ML.

4. Kafka-ML architecture

Kafka-ML is an open-source framework that enables the management of the pipeline of ML/AI applications through data streams. Kafka-ML is a novel framework for integrating ML frameworks and data streams, which are continuously growing thanks to disruptive and massive data production paradigms such as the IoT. It presents a paradigm shift from traditional and static datasets used in ML/AI frameworks into continuous and dynamic data streams, offering a user-friendly, ready-to-use[3] and open platform to the community that allows managing ML/AI pipeline steps such as the inference deployment in productions environments. Kafka-ML works with configurations. A configuration is a logical set of ML models that can be grouped for training and evaluation. This can be useful when it is required to evaluate and compare metrics (e.g., loss and accuracy) of a set of ML models or just to define a group of them that can be trained with the *same* and *unique* data stream in parallel. Therefore, in case of having n ML models, all of which require a data stream for training, only one data stream has to be sent to Apache Kafka if a configuration has been defined with them. This can also apply to data scientists, which may train hundreds of ML models for an application and keep only one to be used in production [50]. And it is particularly true when they perform hyperparameter optimization [7]. Fig. 2 depicts the pipeline to manage these ML models and configurations through Kafka-ML: (1) designing and defining ML models with a few lines of ML model source code; (2) creating a training configuration for ML models, i.e., selecting a set of ML models to be trained and evaluated; (3) deploying the configuration for training and evaluation by selecting corresponding hyperparameters; (4) ingesting the deployed configuration with training and optionally evaluation data stream; (5) deploying trained ML models in production for inference; and, finally, (6) feeding deployed trained models for inference to make predictions with data streams. All the steps related to feeding the ML model (training and inference) can be carried out with data streams. Moreover, most of the previous steps use a RESTful API, so the pipeline can be automatized. Datastores might not be needed anymore with the management of data streams in Kafka-ML (Section 4.7).

Fig. 3 shows an overview of the Kafka-ML architecture. One of its main components is the back-end, which is responsible, among other things, for creating training jobs (Model training) and inference jobs for the trained models (Model inference) upon user request. As in many convectional web architectures, users do not interact directly with the back-end, but with the front-end, which communicates users with the back-end and provides a user-friendly UI for it. Data streams in Kafka-ML are received

[3] https://github.com/ertis-research/kafka-ml.

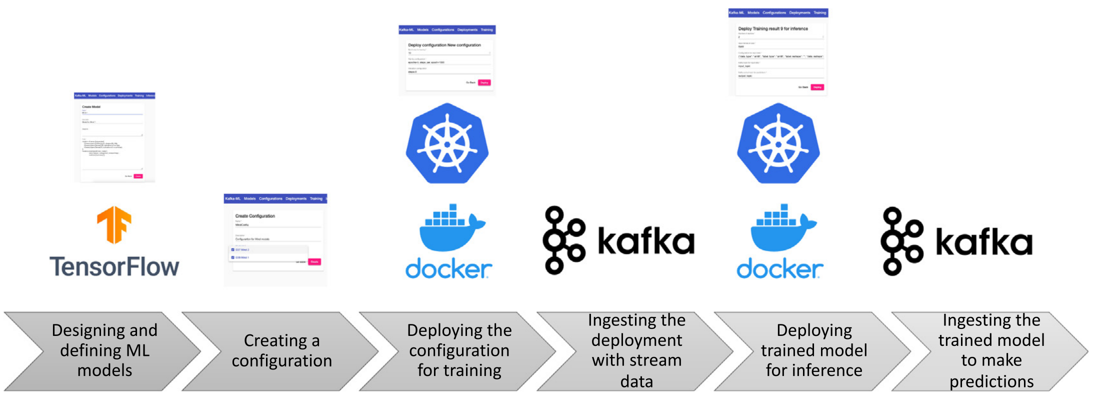

TensorFlow
docker
kafka
docker
kafka
Designing and defining ML models
Creating a configuration
Deploying the configuration for training
Ingesting the deployment with stream data
Deploying trained model for inference
Ingesting the trained model to make predictions

Fig. 2. ML/AI pipeline in Kafka-ML.

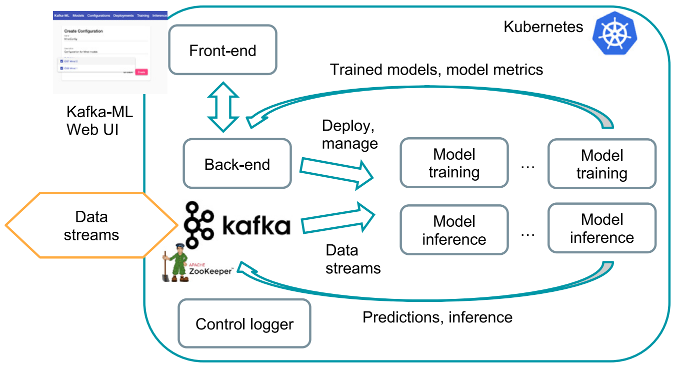


Fig. 3. Overview of Kafka-ML architecture.

by training and inference jobs, which mainly use them for model training and model prediction respectively. Therefore, no static datasets are required to work with Kafka-ML but data streams. Finally, the control logger is a logger for control messages, which will be explained in details in the following sections.

The Kafka-ML framework comprises a set of components based on the single-responsibility principle, comprising a microservice architecture [51]. All of the architecture components have been containerized so that they can run as Docker containers. This not only enables easy portability of the architecture, isolation between instances, and fast setup support for different platforms, but also their management and monitoring through Kubernetes. Kubernetes manages the life cycle of Kafka-ML and its components. Apache Kafka enables the data stream dispatching and management in the framework. Kafka-ML is an open-source project and its implementation, configurations, Kubernetes deployment files, and some examples can be found on GitHub.[4]

4.1. Front-end

The front-end provides a management Web UI where users can perform operations available in Kafka-ML, such as the creation of ML models, configurations, and their deployment for training and inference in a user-friendly and accessible way. This component enables the user interaction with Kafka-ML. The front-end makes use of the RESTful API offered by the back-end, and it is implemented using the popular TypeScript framework for Web development Angular. Since the front-end and back-end have been clearly differentiated, this architecture opens the door to the integration of third-party applications and the creation of new front-ends (e.g., a smartphone application).

4.2. Back-end

The back-end component serves a RESTful API to manage all the information contained in Kafka-ML, such as ML models, configurations, and deployments. This component is connected with the corresponding Kubernetes API to deploy and manage training as well as inference of configurations and ML models when ordered by users through the front-end. Moreover, the back-end also receives trained ML models and metrics after training a configuration. These trained models can be downloaded or deployed for later inference. This component has been implemented through the Python Web framework Django and the official Python client library for Kubernetes[5] for the deployment and management of Kubernetes components.

4.3. Model training

Once the back-end deploys a configuration, a job, a deployable unit containerizing a Docker container, will be executed per Kafka-ML model for training. Since the Kafka stream connector expects to have the data stream at the initiation, training cannot start until the data stream is available in the Kafka topic. We have used at least two Kafka topics to overcome this: (1) one or more data topics which only contain training and evaluation data streams required for training and evaluation; and, (2) a control topic, which informs deployed ML models through control messages *when* and *where* data streams are available for training and evaluation. Section 4.7 will discuss this in detail. A control message should contain at least the following information:

- **deployment_id**: ID of the deployed configuration that is going to ingest data stream.
- **topic**: Kafka topic where the training and evaluation data streams are.
- **input_format**: Format of the data stream (e.g, RAW, Avro).
- **input_config**: Parameters required by the data format used (e.g., the scheme to deserialize messages in Avro, and data type and reshape for label and data in RAW format).
- **validation_rate**: Percentage of data stream that will be used for evaluation. If *validation_rate* is equal to zero, only training will be performed.
- **total_msg**: Number of messages dispatched in the data stream. The number of messages sent are calculated by the Kafka-ML libraries.

[4] https://github.com/ertis-research/kafka-ml.

[5] https://github.com/kubernetes-client/python.

Once configurations and subsequently ML models receive control messages, they know exactly where data streams are in Apache Kafka, how to deserialize them and the amount of data streams destined for training and evaluation. Kafka-ML currently supports RAW format (suitable for single-input data streams that may request a reshape, like images) and Apache Avro [52] (suitable for complex and multi-input datasets where a scheme specifies how the data stream is decoded) to serialize/deserialize data streams. However, it is open to support new data formats. In each case, the information for deserialization is included in the control message (*input_config*), as, for example, the training and label data schemes for the Avro format. We have developed libraries for these two data formats, which make the data stream dispatching easier to implement for users since they deal with Kafka-ML aspects like sending the control message when the data stream has been sent. Without these libraries, users would have to send data streams manually to Apache Kafka, implement serialization/deserialization techniques and send control messages as detailed before.

Algorithm 1 describes the procedure of the training job once deployed by the Kafka-ML back-end and Fig. 4 depicts the sequence diagram of the training process. It is noteworthy that some steps such as management of exceptions, Kafka connections and data stream decoding have not been included for the sake of simplicity. Firstly, once deployed (1) by the back-end, the training job downloads the ML model from the back-end (2–3) and loads it as a TensorFlow model ready for training (4). Next, it creates a Kafka consumer that waits until receiving the control stream for this process (6), i.e., it matches the *deployment_id* received by the back-end in the job (1). The control message indicates how data stream sent by clients (5) can be deserialized (data format: msg.input_format; configuration parameters for the data format: msg.input_config), batch size used for training (msg.batch) and where data stream is available (Kafka topic: msg.topic). Then, the job obtains the data stream for training from the Kafka topic indicated (8). In case of any error during this process (e.g., the received data does not match the information indicated in the control message), the job will wait until a new control message is received (7). If the error is due to another reason and not to the data streams, the component will be restarted and will re-process control messages previously received. Once the data stream is obtained, the dataset is split for training and evaluation according to *validation_rate* received in the control message (9), i.e., this parameter indicates the amount of stream data for evaluation. If msg.validation_rate is set to 0, only training and not evaluation is performed, i.e., the entire data_stream goes to the training_stream variable. Therefore, if only training is going be performed, msg.validation_rate will be set to 0 when the control message is sent (6). Next, the model is trained and optionally evaluated (10). Note that Kafka-ML does not allow continual learning [53], i.e., training is carried out at once and not continuously accumulating knowledge from previous data streams. Therefore, in case of receiving a small data stream, ML models will be trained only with this data. The availability of data stream for training tasks is controlled by control messages, thus Kafka-ML users can have a proper control when the data stream accumulated is ready for training. The data stream for training must be at least larger than the batch size indicated in the control message sent. Finally, the job submits the trained model as well as the training and evaluation metrics to the back-end. This process is carried out in each job for every ML model in the configuration deployed.

```
Input: model_url, training_kwargs, evaluation_kwargs,
deployment_id
Result: Trained ML model and training and evaluation metrics
model_data ← downloadModelFromBackend(model_url);
model ← loadTensorFlowModel(model_data);
trained ← False;
while not trained do
    msg ← readControlStreams();
    if deployment_id == msg.deployment_id then
        decoder = getDecoder(msg.input_format,
        msg.input_config);
        data_stream ← readStream(msg.topic, decoder,
        msg.batch);
        if msg.validatition_rate > 0 then
            training_stream ← take(data_stream, (1 -
            msg.validation_rate));
            evaluation_stream ← split(data_stream,
            msg.validation_rate);
        else
            training_stream ← data_stream;
        end
        training_res ← trainModel(model, training_kwargs,
        training_stream);
        if msg.validatition_rate > 0 then
            evaluation_res ← evaluateModel(model,
            evaluation_kwargs, evaluation_stream);
        end
        uploadTrainedModelAndMetrics(model, training_res,
        evaluation_res);
        trained ← True;
    end
end
```

Algorithm 1: Training algorithm in Kafka-ML

The data stream flow starts when a client (e.g., an IoT device or gateway, a dataset, or any information source) sends a data stream (5) to an Apache Kafka topic (e.g., "iot" topic) and a cluster for training a configuration. A Kafka topic is the elementary way of connecting producers and customers (training job in this case) in Apache Kafka, and each topic is independent of the rest. Topics are managed by a cluster of Kafka Brokers, which comprise Kafka's architecture and are responsible for receiving a data stream from producers and distributing it to subscribing consumers. The data stream for training has to be serialized to be sent according to the data formats available in Kafka-ML (RAW and Avro). Once dispatched, a control message to the control topic configured in Kafka-ML (default "control" topic) has to be sent (6) indicating the location of the data stream sent (Kafka topic and position), the deserialization information, and the deployment to which the data stream is going, among other parameters. The available clients in Kafka-ML[6] can be used to send and serialize data streams as well as to send control messages. Once the corresponding jobs in the deployed configuration receive the control message (i.e., the *deployment_id* in the control message matches the one received by the back-end) (7), they all receive the data stream sent (8), e.g., *deployment_id* 1 in this example, in the "iot" topic in this case, and carry out the process described in Algorithm 1.

4.4. Model inference

After training an ML model and deploying it for inference through the front-end and back-end, a Replication Controller (a component that ensures that a specified number of replicas

[6] https://github.com/ertis-research/kafka-ml/tree/master/datasources.

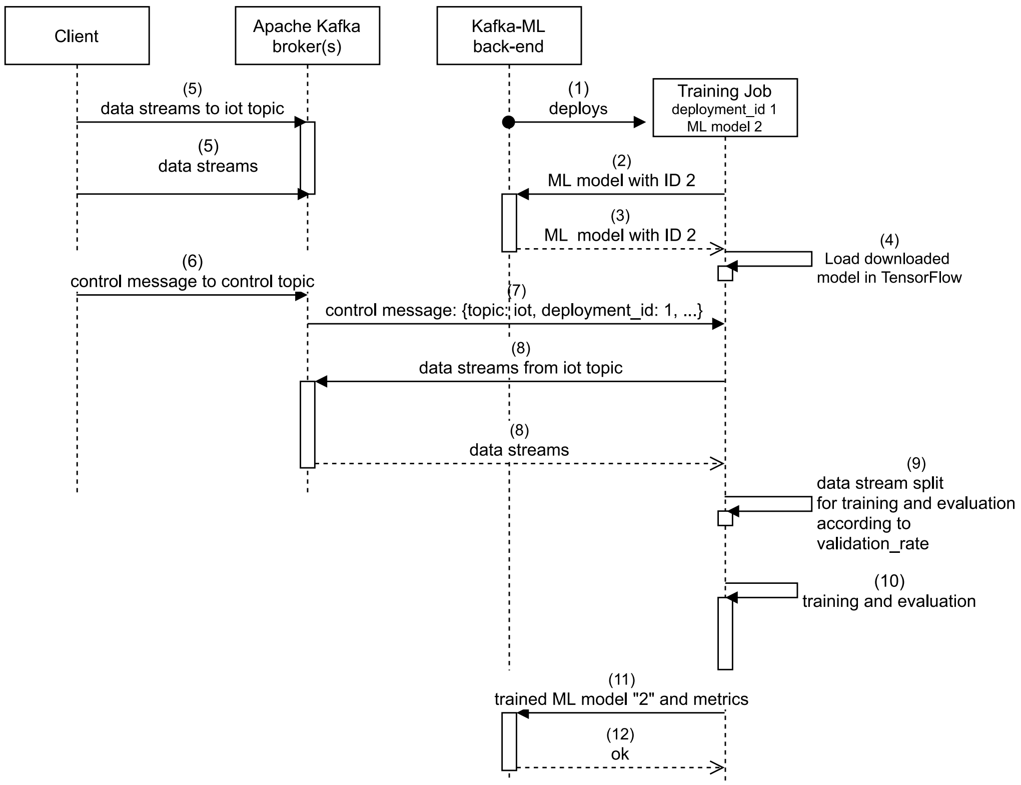

Client
Apache Kafka broker(s)
Kafka-ML back-end
Training Job deployment_id 1 ML model 2
(1) deploys
(2) ML model with ID 2
(3) ML model with ID 2
(4) Load downloaded model in TensorFlow
(5) data streams to iot topic
(5) data streams
(6) control message to control topic
(7) control message: {topic: iot, deployment_id: 1, ...}
(8) data streams from iot topic
(8) data streams
(9) data stream split for training and evaluation according to validation_rate
(10) training and evaluation
(11) trained ML model "2" and metrics
(12) ok

Fig. 4. Sequence diagram of the training process in Kafka-ML.

are running at all times) with established replicas will be executed in Kafka-ML. The condition to deploy an ML model is that it is trained in the Kafka-ML framework, assuming that an infrastructure for its deployment is available. In fact, the option to deploy and download the trained model in the front-end is enabled only once the ML model is trained. Replication of this component enables fault tolerance, high availability, and load balancing of the requests among inference deployments. Algorithm 2 describes the procedure of the inference in Kafka-ML and Fig. 5 the sequence diagram. Once the Replication Controller is deployed (1), the trained model from the back-end is downloaded (2–3) and it is loaded as a TensorFlow model ready for inference (4). Then, this component will start receiving data stream (6) and, subsequently, will make a prediction with them (7), which will be sent through the configured Kafka output topic (8). Each message is also deserialized with the information received by the back-end (1) and sent in JSON format to the output topic (8). Algorithm 2 is executed by each of the replicas of the inference deployed, and all of them are part of the same Kafka consumer group to distribute the data stream load thanks to the group_id parameter received and automatically generated by the back-end (1), which indicates the consumer group to join. The inference module and its replicas are independent of the Kafka brokers and topic partitions used. Inference replicas receive the Kafka cluster and topic to connect with (both for the input and output topics), which can contain either 1 broker or 1 partition or as many as defined in the Kafka-ML Web UI. This has different implications for the performance of Kafka-ML as we will study in Section 6, and the inference module adapts itself to all of them. If an unexpected data stream is received, such as an invalid format message, it will be ignored and the inference module will continue waiting for new messages. In case of any other failure, this component and its replicas will be restarted. When having multiple Kafka Brokers and partitions, the Replication Controller exploits the consumer group feature of Apache Kafka by matching replicas and partitions to provide load balancing and higher data ingestion.

```
Input: model_url, input_topic, output_topic,
       input_configuration, group_id
Result: Predictions to Apache Kafka
model_data ←
downloadTrainedModelFromBackend(model_url);
model ← loadTensorFlowModel(model_data);
deserializer ← getDeserializer(input_configuration);
while True do
    stream ← readStreams(input_topic);
    data ← decode(deserializer, stream);
    predictions ← predict(model, data);
    sendToKafka(predictions, output_topic);
end
```

Algorithm 2: Inference algorithm in Kafka-ML

In this case, the sequence (Fig. 5) requires less steps than the training since no control messages are required. Once clients send data streams to the Kafka input topic configured (5), the latter are directly received by the inference jobs and replicas (6) so they can obtain a prediction with the downloaded model (7) and dispatch it to the output topic (8). Finally, applications interested in receiving the predictions can subscribe to the output topic (9).

4.5. Control logger

The control logger component is just responsible for consuming control messages received in Kafka-ML and for sending them to the back-end. These control messages are used for two purposes in the back-end: (1) to allow sending messages again to other deployed configurations without the need to send the entire training and evaluation data stream, which is possible since Apache Kafka keeps data streams in the distributed log; and (2)

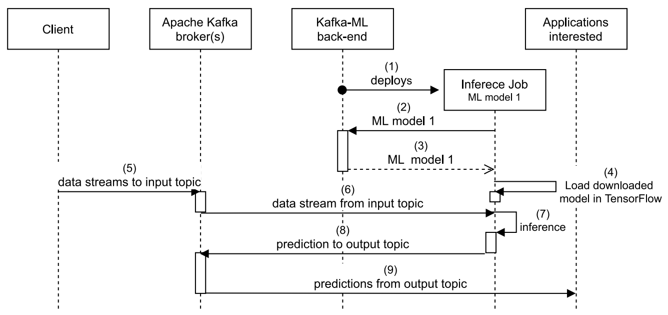

Client
Apache Kafka broker(s)
Kafka-ML back-end
Applications interested
(1) deploys
Inferece Job ML model 1
(2) ML model 1
(3) ML model 1
(4) Load downloaded model in TensorFlow
(5) data streams to input topic
(6) data stream from input topic
(7) inference
(8) prediction to output topic
(9) predictions from output topic

Fig. 5. Sequence diagram of the inference process in Kafka-ML.

to auto-configure the inference input format and configuration in the front-end with respect to the received information in control messages. The input format and configuration are not directly configured in the Kafka-ML Web UI, but they are defined in the control messages. Therefore, filling out these fields in the front-end automatically with this information facilitates the work for users in defining the data parameters when deploying an ML model for inference. During training, jobs receive the control data stream as well as this information.

4.6. Apache Kafka and ZooKeeper

To facilitate the deployment and management of Apache Kafka in our microservice infrastructure framework, we have deployed Apache Kafka[7] and Apache ZooKeeper[8] (required by Apache Kafka for synchronization) as jobs using Docker containers. We have enabled their exposure through a Kubernetes service both internally to the rest of the components and externally to enable other systems to send data streams.

4.7. Data stream management through Apache Kafka distributed log

As discussed in Section 2, the distributed log provided in Apache Kafka enables consumers to move along the log and read data streams as they wish. This is useful when a component/system that has to process all data at once (e.g., a training job) fails and needs to recover the whole data stream. In traditional message queue systems where each message may be deleted after consumption, a datastore may be needed to ensure there is no data loss in these situations.

On the other hand, since data streams can be configured to be kept in the log, these streams can be reused for training other deployed configurations and ML models without the need for sending the whole data stream again. The only requirement is to send the corresponding control message (tens of bytes) to the desired deployed configuration as long as the data stream is available in Apache Kafka with the established retention policy. An example of this functionality is shown in Fig. 6. Firstly, the first data stream (green data) was sent along with its control message (C1) to the deployed configuration D1. A control message C1 was sent again to allow configuration D2 to consume the same data stream. In the current distributed log state, this data stream is expiring and cannot be longer reused for another deployed configuration. The data stream associated with the control message C2 has been sent to the deployed configurations D3 and D5, which can still be reused for new configurations that want to use this data stream again. Finally, the data stream on the left in the training and evaluation stream is now entering the distributed log and the control message has not yet been sent since the data stream is not yet complete.

To allow training and evaluation tasks to move freely along the data stream, control messages specify not only where the data streams are, but also what their position in the distribution log is. This follows the following format provided by the KafkaDataset connector from TensorFlow/IO: [topic:partition:offset:length]. For instance, the example [kafka-ml:0:0:70000] specifies that in the topic "kafka-ml" and its partition 0, the data stream is available from the offset position 0 to 70 000. Through control messages, Kafka-ML informs deployed configurations where exactly data streams are. In the Kafka-ML Web UI, a form is available where users can see the list of the data stream that has been sent to Kafka-ML, which can be reused for other configurations.

As discussed, this behavior depends on the retention policy established in Apache Kafka. Current retention strategies within the Apache Kafka *delete* retention policy are:

1. Retention bytes: Control the maximum size a partition can grow up to before Kafka starts discarding old log segments to free up space. Default not applicable.
2. Retention ms: Control the maximum time a log will be retained before old log segments will be discarded to free up space. Default to 7 days.

It is noteworthy that Apache Kafka provides another retention policy known as the *compact* policy, which ensures that Kafka will retain at least the last known value for each message key for a single topic partition. Nevertheless, due to the necessity of neither loss nor compacting data, the delete retention policy would be preferred for Kafka-ML instead.

5. Pipeline of an ML model in Kafka-ML

In this section, we introduce the pipeline of an ML model in Kafka-ML using the example of the Exasens dataset. The open access Exasens dataset, available at the UCI machine learning repository,[9] is used for the classification of saliva samples of Chronic Obstructive Pulmonary Disease (COPD) patients and Healthy Controls (HC). This novel dataset contains information on saliva samples collected from groups of respiratory patients including COPD and HC [54]. The attributes of the dataset, used for the classification of subjects, include demographic information of patients (age, gender, and smoking status) and the dielectric properties of the characterized saliva samples for every class [55]. The source code of this example is included in the examples of Kafka-ML.[10] The metrics, models, and sections of the data used for analytics in this example are available at GitHub.[11] In the following subsections, each of such steps in the Kafka-ML pipeline (Fig. 2) is elaborated.

5.1. Designing and defining ML models

The novel initiative of this work was to develop a user-friendly framework which enables ML specialists to focus on developing ML models instead of learning a new library or using complex pipelines. A tool which enables easy testing and validation of ML models would considerably reduce the work load of ML developers and would allow them to focus on their expertise. As a result, the only source code needed for this approach is simply the ML model definition itself in a common ML framework, as shown in Listing 1.

[7] https://github.com/wurstmeister/kafka-docker.
[8] https://hub.docker.com/_/zookeeper.

[9] https://archive.ics.uci.edu/ml/datasets/Exasens.
[10] https://github.com/ertis-research/kafka-ml/tree/master/examples/HCOPD_Avro_format.
[11] https://github.com/Pouya-SZ/HCOPD.

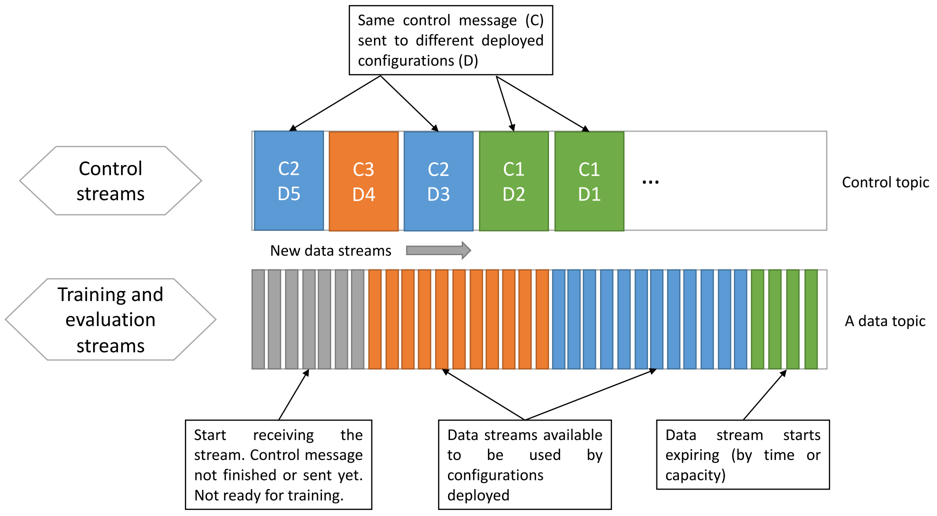

Same control message (C) sent to different deployed configurations (D)
Control streams
C2 D5
C3 D4
C2 D3
C1 D2
C1 D1
...
Control topic
New data streams
Training and evaluation streams
A data topic
Start receiving the stream. Control message not finished or sent yet. Not ready for training.
Data streams available to be used by configurations deployed
Data stream starts expiring (by time or capacity)

Fig. 6. Data stream management in Kafka-ML.

Listing 1: HCOPD TensorFlow/Keras ML code

```
model = tf.keras.Sequential([
tf.keras.layers.Dropout(0.2,
    input_shape=(4,)),
tf.keras.layers.Dense(4,
    activation='sigmoid'),
tf.keras.layers.Dense(2,
    activation='softmax')
])
model.compile(tf.keras.optimizers.Adam
(lr=.0001),
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

Listing 1 shows the source code of the model for the Exasens dataset. This code may seem familiar. In fact, it is a Python TensorFlow/Keras model with a dropout, a hidden layer, a single output and the compilation for training. This model is used for the classification of COPD and HC samples. An overview of this model, its inputs and outputs is shown in Fig. 7. Kafka-ML currently works with Python TensorFlow [56] and its support for Apache Kafka through TensorFlow/IO.[12] Therefore, Kafka-ML currently supports any ML/AI model with corresponding valid inputs and outputs that can be defined with Tensorflow/Keras, such as a deep learning model. Currently, Apache Kafka integrations are under development and its Kafka-ML domain is getting expanded by receiving further ML frameworks.

Once an ML model is defined using an ML editor (e.g, Jupyter [57]), the TensorFlow/Keras source code of the model can be inserted into the Kafka-ML Web UI for model creation, as shown in Fig. 8. It is noteworthy that the model can also be defined directly on Kafka-ML, though it is recommended to validate it beforehand using other and more powerful ML Integrated Development Environment (IDE) or editors. Other required functions for the model (if necessary) can be inserted in the *imports* field. Once the model is submitted, the source code will be checked as a valid TensorFlow model and will be incorporated into Kafka-ML, i.e., the semantics of the code will be checked to prevent malicious source codes, and an instance of the model in Tensorflow will be created and executed during the validation to corroborate that a valid TensorFlow model has been defined. If the model definition was successful, the pipeline will be continued to the next step.

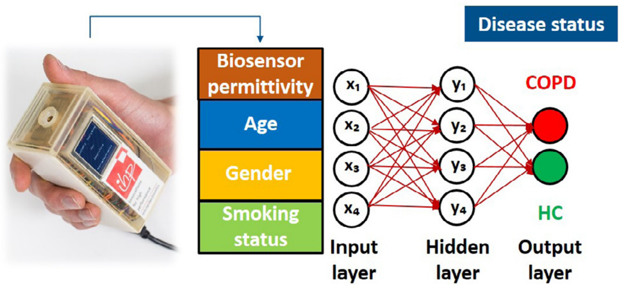
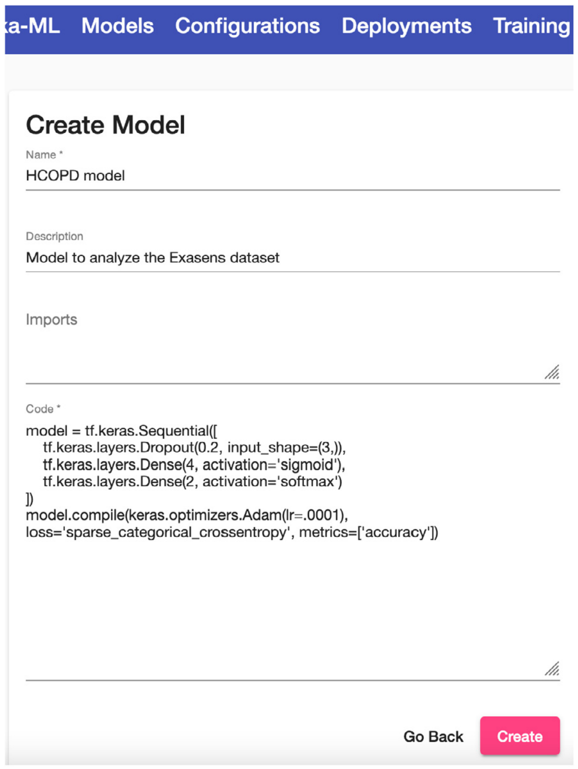


Fig. 7. Architecture of a neural network model used for the classification of COPD and HC samples available at the Exasens dataset, using IHP's permittivity biosensor.



Create Model

Name *
HCOPD model

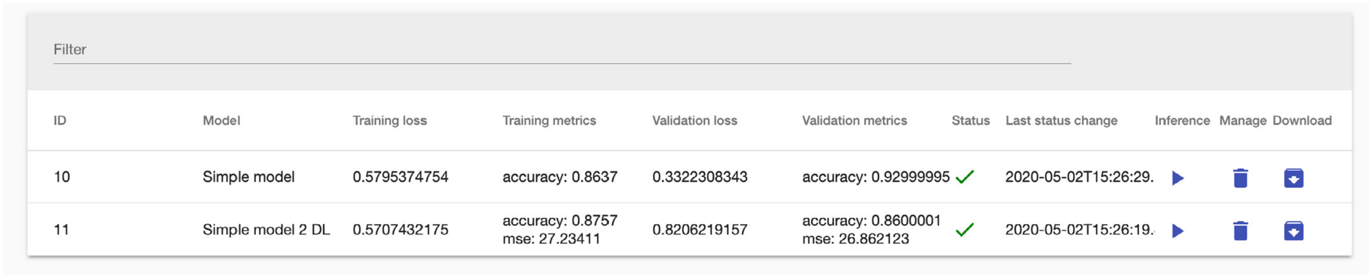
Description
Model to analyze the Exasens dataset

Imports

Code *
```
model = tf.keras.Sequential([
    tf.keras.layers.Dropout(0.2, input_shape=(3,)),
    tf.keras.layers.Dense(4, activation='sigmoid'),
    tf.keras.layers.Dense(2, activation='softmax')
])
model.compile(keras.optimizers.Adam(lr=.0001),
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Fig. 8. Definition of the HCOPD ML model in Kafka-ML.

12 https://www.tensorflow.org/io.



Filter

ID	Model	Training loss	Training metrics	Validation loss	Validation metrics	Status	Last status change	Inference	Manage	Download
10	Simple model	0.5795374754	accuracy: 0.8637	0.3322308343	accuracy: 0.92999995	✓	2020-05-02T15:26:29.	▶	🗑	⬇
11	Simple model 2 DL	0.5707432175	accuracy: 0.8757 mse: 27.23411	0.8206219157	accuracy: 0.8600001 mse: 26.862123	✓	2020-05-02T15:26:19.	▶	🗑	⬇

Fig. 9. Training management and visualization in Kafka-ML.

5.2. Creating a configuration

Once ML models have been registered in the Kafka-ML Web UI, it is time to define configurations. For instance, in the Exasens dataset a target configuration could consist of having different models with different hidden layers to evaluate their performance. Since models in a configuration share the same hyper-parameters for training and evaluation during deployment (next phase), they must share some similarities to be grouped in a configuration, e.g., they should have the same input layer and may have the same learner, especially if they use specific parameters. It is noteworthy that a configuration can also be defined with only a model.

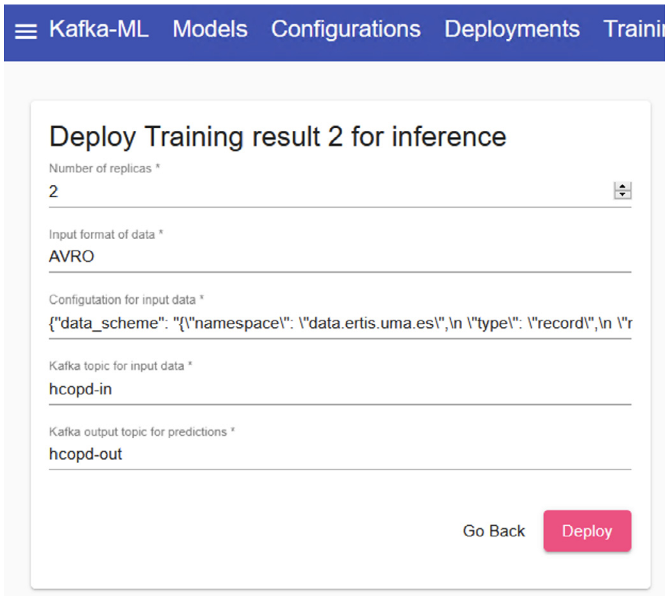
5.3. Deploying the configuration for training

After setting some training parameters such as the batch size, epochs, number of iterations, and optionally some parameters for evaluation in the Kafka-ML Web UI, the configuration will be ready to be deployed for training. In this example, a batch size of 10 is used, the following training parameters “epochs=5, steps_per_epoch=1000, shuffle=True”; and “steps=5” for evaluation. Validation rate was configured with 0.33, thereby 77% of data are used for training and 33% for evaluation. When users submit the configuration for training in the Web UI, a Kubernetes job will be deployed per Kafka-ML model (Fig. 4-(1)). One of the first steps that each deployed job carries out is fetching its corresponding ML model (e.g., Listing 1) from the Kafka-ML framework (Fig. 4-(2–3)) and loading it to start training (Fig. 4-(4)). Finally, jobs can resume until a data stream with training and optionally evaluation data is received through Apache Kafka (Fig. 4-(8)), right after the corresponding control message (Fig. 4-(6–7)). This allows both having ready-to-train ML models when a data stream is sent and direct training if the data stream is already in Kafka.

5.4. Ingesting the deployment with data stream

Once models have been deployed for training, data streams have to be sent for training and continuation of the pipeline (Fig. 4-(5)). Since the Exasens dataset requires a complex input type (gender, age, and smoking status), the Avro encoding was used both for data and labels. The Avro scheme for data is shown in Listing 2. Data streams are then serialized with the Python Avro library[13] and sent to the Apache Kafka topic “hcopd” using the Avro client available in Kafka-ML. This client automatically sends the control message right after dispatching the data stream for training. The control message requires some parameters like the validation rate and the deployment ID for which these data are intended (it can be obtained in the Web UI). All the models in the configuration will receive this control message (Fig. 4-(7)) and will start the training process with the data stream sent to the hcopd topic.

[13] https://avro.apache.org/docs/current/gettingstartedpython.html.



Deploy Training result 2 for inference

Number of replicas *
2

Input format of data *
AVRO

Configutation for input data *
{"data_scheme": "{\"namespace\": \"data.ertis.uma.es\",\n \"type\": \"record\",\n \"r

Kafka topic for input data *
hcopd-in

Kafka output topic for predictions *
hcopd-out

Fig. 10. Deploying a trained ML model for inference in Kafka-ML.

Listing 2: Avro scheme for the Exaxens dataset

```
{"namespace": "data.ertis.uma.es",
 "type": "record",
 "name": "HCOPD",
 "fields": [
     {"name": "gender", "type": "float"},
     {"name": "age", "type": "float"},
     {"name": "smoking", "type": "float"}
 ]
}
```

5.5. Deploying trained model for inference

Right after training and evaluation, both the trained model itself and its defined metrics (e.g., loss and accuracy) will be submitted by each training job to the Kafka-ML back-end (Fig. 4-(11)). Results can be visualized in the Kafka-ML Web UI as shown in Fig. 9 for a configuration with two ML models for the Exasens dataset. Through this, users can evaluate and compare ML models and metrics defined in a collaborative way (challenge 2). For each training result, users can edit or download the trained model, or deploy it for inference (play button).

In the inference deployment (Fig. 10), users can select the number of inference replicas to be deployed. This exploits the consumer group feature of Apache Kafka, thereby enabling load balancing and fault-tolerance for inference. Since all interactions are done through Apache Kafka, users have to configure the input topic (for values to predict on) and output topic (for predictions). The information for deserialization (the Avro scheme in this example) is obtained from the control message dispatched and submitted by the Control logger component to the back-end, thereby the "Configuration for input data" field in this form is automatically filled out from this information. This field contains the information for data and label serialization during inference, the Avro schemes in this example. The input and output topics used in this example are "hcopd-in" and "hcopd-out", respectively.

5.6. Ingesting deployed trained models with data stream for inference

Finally, the ML/AI pipeline concludes once the trained models are deployed to make predictions and recommendations through data streams (Fig. 5-(1)). In this case, no control messages have to be sent since the input and output topics, as well as the information for deserialization, have been previously defined in the Web-UI (Fig. 10) and received by inference replicas. Users and systems just need to send encoded data streams with the Avro scheme presented in this example to the "hcopd-in" topic (Fig. 5-(5)), and inference results about the diseased and HC will immediately be sent after processing to the "hcopd-out" topic (Fig. 5-(8)), which can be subscribed by applications interested (Fig. 5-(9)).

6. Evaluation

We have conducted several experiments to evaluate the performance regarding features of Kafka-ML such as fault tolerance, load balancing, and high availability. In particular, we have focused the performance evaluation on the inference since this is the step where we can exploit most of these features through replication. Training is also another important step. However, as distributed training (planned for future work) is not yet supported in Kafka-ML, it is performed individually for each ML model, and there is no capacity for further distribution and replication of data streams. This does not mean that the training is not fault-tolerant, since if for whatever reason it fails or does not receive the corresponding data streams, it will restart until the model is trained satisfactorily. The system is fault-tolerant to hardware or node failures during training. Moreover, the data stream is reliably stored in Apache Kafka. Regarding training, we have analyzed how the number of the data stream sent affects the ML model metrics and the training time.

We have picked Faust and TensorFlow Serving for comparison because they both can run and serve TensorFlow models like Kafka-ML. Faust is a stream processing framework written in Python so we could easily port the inference implementation to it (Kafka-ML is mainly written in Python). In addition, as Faust also uses Apache Kafka, we can evaluate inference with data streams in two different streaming systems. TensorFlow Serving is perfectly integrated with Docker and allows easy loading of ML models through its hosted service. Therefore, we only had to download the trained model from Kafka-ML and create a configuration file to run it in Kubernetes. TensorFlow Serving is an inference production module developed by Google. Although it does not support data streams (it supports RPC instead), it is still a good candidate for comparison in production environments.

As discussed in Section 5, Kafka-ML can deploy simple TensorFlow ML models for ML/AI applications. However, Kafka-ML can deploy any other compatible ML model in Tensorflow, as well as larger and more feature-intensive ones, such as Deep Learning models. We have used the well-known VGG16 Deep Learning model [58] for the performance evaluation in Kafka-ML with an ML model that requires greater needs. As seen in

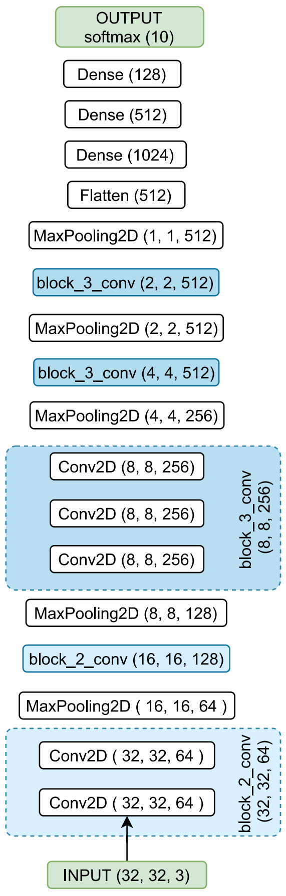


Fig. 11. VGG16 model used for performance evaluation.

Fig. 11, this ML model contains a large number of deep layers, both dense and convolutional. VGG16 gets a 92.7% top-5 test accuracy on ImageNet [59], and thus, we can successfully train the VGG16 model to classify images on CIFAR10,[14] which is the dataset used for training and evaluation processes. The CIFAR-10 dataset consists of 60,000 32 × 32 color images in 10 classes (such as animals, cars, and airplanes), with 6000 images per class. Specifically, we got an accuracy of 87,33% for the VGG16 model and CIFAR10 dataset during training with 100 epochs. Because this dataset contains images, we used RAW serialization. To make a comparison with the model for the Exasens dataset seen above, the VGG16 model occupies 61MB versus the few kilobytes of the Exasens ML model, and the training time lasted several hours versus tens of seconds for the Exasens model. For the sake of reproducibility, this example is also included in the examples of Kafka-ML.[15]

All the Kafka-ML components were replicated and deployed in a Kubernetes cluster. Each experiment was performed ten times, starting by a client sending a CIFAR10 image to an input topic until a response from an output topic is received. Experiments with different numbers of clients sending information concurrently reported the average running time and throughput. The experimental setup to evaluate Kafka-ML in the distributed Kubernetes cluster is described in Section 6.1. The performance evaluation was carried out in the following scenarios:

14 https://www.cs.toronto.edu/~kriz/cifar.html.
15 https://github.com/ertis-research/kafka-ml/tree/master/examples/VGG16_CIFAR10_RAW_format.

- Replication performance of Kafka-ML with 1 Kafka Broker (Section 6.2).
- Deployment in a higher availability scenario with 3 Kafka Brokers. (Section 6.3).
- Evaluation of the fault tolerance and high availability features of Kafka-ML (Section 6.4).
- Comparison with other proposals (Section 6.5).

Finally, Section 6.6) analyzes the performance of Kafka-ML with respect to training.

6.1. Experimental setup

Hardware configuration. All the experiments were performed on a five-node Kubernetes cluster at our private cloud infrastructure in VMware vCloud. Each node has 4 virtual CPUs in 2 sockets, and 16 GB of RAM. The client that sent the information and where the results were measured from was a PC with 8 GB of RAM, 1 CPU and 4 cores.

Software configuration. Each one of the five nodes runs Kubernetes v1.19.3 and Docker 19.03.13 on top of Ubuntu 16.04.7 LTS. A Kubernetes master was deployed in one node, while the remaining four are Kubernetes workers. The PC with the client runs Windows 10 Professional.

6.2. Scenario 1: 1 Kafka Broker

In this setup the inference module was configured to use 1 Kafka Broker and Kafka topics were configured with 1 partition and 1 replica each. Kafka topics and partitions can be created in multiple ways, for example, using an open-source tool such as Kafka manager.[16] Then, the topics defined are configured during the deployment of the inference (Fig. 10) in Kafka-ML. This scenario covers the minimum needs for Kafka-ML: the use of a single broker and one partition per topic. Fig. 12 shows the average latency response with different number of clients and different inference replicas. It can be observed that by increasing the replication of the inference deployment, response time is reduced up to 80%, comparing results with 1 replica to 32 replicas. The results of 16 and 32 replicas are very similar so they are negligible in the graph.

The average throughput of this scenario is shown in Fig. 13. In this case, the higher throughput is obtained with 8–16 replicas and 8 clients. This may be due to the overload of clients that Kafka-ML can handle with a single Kafka Broker, so this is the optimum step between a higher data rate and a higher number of clients. For this reason, after 8 replicas, there is no considerable improvement.

With this scenario we have shown how Kafka-ML can successfully handle different client loads through the replication of the inference module. This is of special interest when working with data streams since paradigms such as the IoT produce many of these and sometimes in variable conditions, for which Kafka-ML could be perfectly adapted.

6.3. Scenario 2: 3 Kafka Brokers

In this scenario the inference module was configured with 3 Kafka Brokers. Thanks to this, the topics could be configured to support a larger number of partitions (one per Broker) and a replication of 2. The inference deployment can now exploit the Kafka consumer group with the 3 partitions configured to better distribute the load, and the replication level of 2 allows 1 Kafka Broker to fail without losing information.

16 https://github.com/yahoo/CMAK.

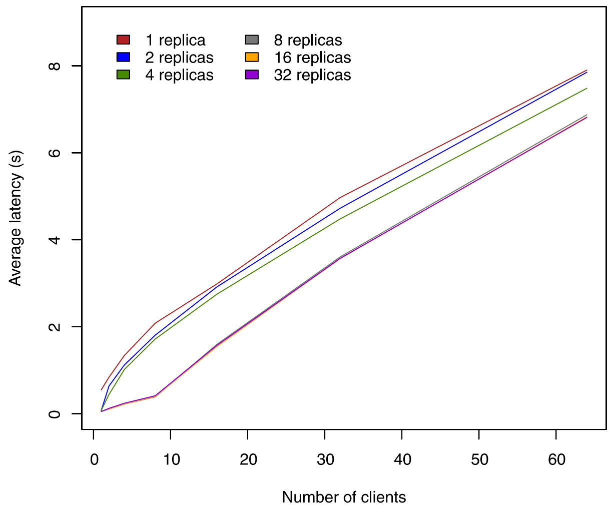


Fig. 12. Average inference latency response with different number of clients (1 Kafka Broker).

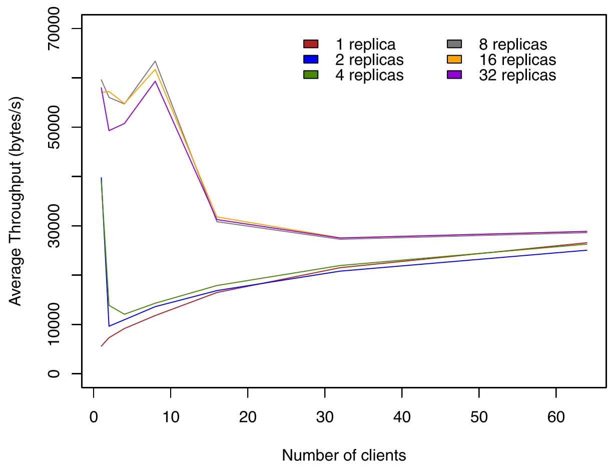


Fig. 13. Average inference throughput with different number of clients (1 Kafka Broker).

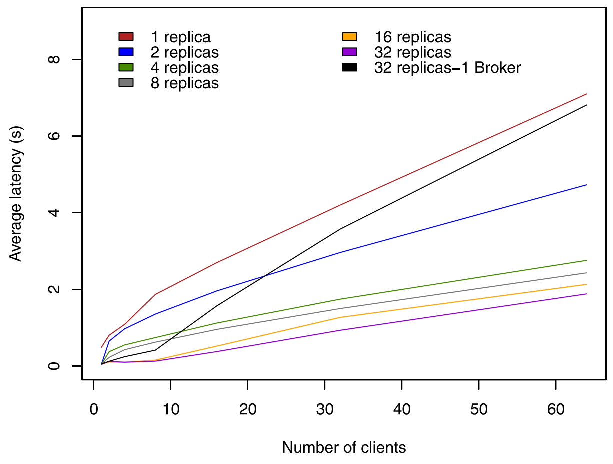


Fig. 14. Average latency response of inference with different number of clients (3 Kafka Brokers).

Comparing the latency response of this scenario with the 32-replica result obtained in the previous one (Fig. 14), it can be

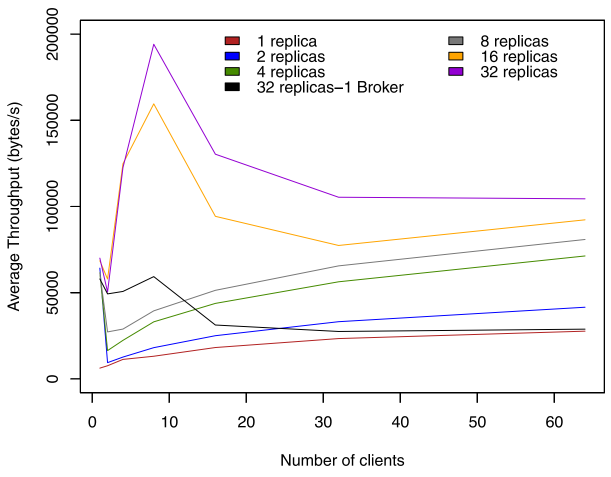


Fig. 15. Average throughput of inference with different number of clients (3 Kafka Brokers).

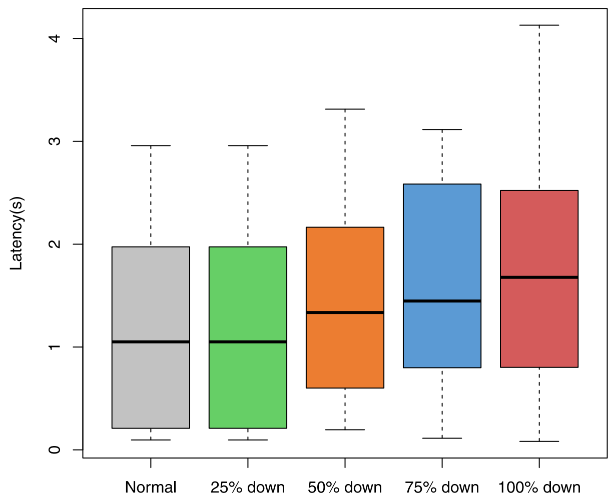


Fig. 16. Inference latency response with 32 clients and different replicas down.

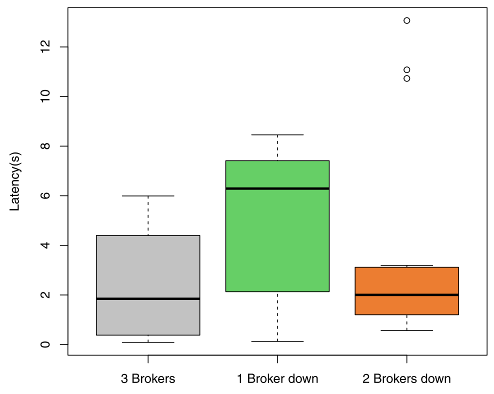


Fig. 17. Inference latency response with 32 clients and different Apache Kafka Brokers down.

seen that the inference module can exploit the larger number of Brokers and partitions to better distribute the load among its replicas. Obtaining a latency reduction percentage of more than 70% by comparing the lowest latency response (32 replicas) in the two scenarios.

Throughput (Fig. 15) also improves the results of the previous scenario with a higher number of Brokers and partitions. As in the previous scenario, the best compromise is with 8 clients, for which an improvement of more than 69% is obtained, in this case with 32 replicas that better exploit the distribution.

In this scenario we have seen how Kafka-ML can exploit Kafka's consumer groups and provide load balancing between inference replicas with different data stream ingestion, improving the performance of Scenario 1. With this setup, Kafka-ML can be adapted to conditions that require a higher load of clients and data streams with a lower latency response than Scenario 1.

6.4. Scenario 3: Fault tolerance and high availability

To evaluate the fault tolerance and high availability features of Kafka-ML, we deployed the inference module as in Scenario 2 and manually stopped some of its replicas to investigate how this would possibly affect the response time. Fig. 16 shows the median response (black line) with 32 clients and different number of replicas down in each case. The main boxes indicate the 25th and 75th percentile of the plotted data. The average percentage increases with respect to a normal situation with 25%, 50%, 75%, and 100% of replicas down is 19.06%, 28.49%, 40.52%, and 53.7%, respectively. This shows that Kafka-ML can tolerate up to 100% replication downtime, thanks to the continuous monitoring and restoration of inference replicas.

We also evaluated how the crash of Kafka Brokers (1 and 2) affects Kafka-ML (Fig. 17). For this case, we configured the inference module with Kafka topics with 3 partitions and 3 replicas. As the Brokers were configured with 3 replicas, there was no data loss in the crash of either 1 or 2 Brokers. The average response time increases threefold with one Broker down, and barely with 2 Brokers down. However, since there was one partition for each Broker, some communications failed, namely 36% for the one-Broker crash and 50% for the two-Broker crash. Therefore, even though the two-Broker crash offered lower response times, there were further lost communications with the Kafka library used. We will look at how to improve this in the future. It is important to note that these results are only at the moment when the Brokers are down, since our framework deploys them in very few seconds after crashing.

To sum up, in this scenario it has been shown how Kafka-ML can tolerate component failures and provide high availability. Specifically, Kafka-ML can tolerate both failures of its inference replicas deployed and of the message distribution brokers used without loss of information and providing high availability to end users.

6.5. Scenario 4: Comparison with other proposals

Finally, in this last scenario we evaluated Kafka-ML with respect to Faust and TensorFlow Serving. One Kafka Broker was used for this scenario since TensorFlow Serving is not using Apache Kafka, and we also had problems setting up Faust with more Brokers. The deployment of 1 and 32 replicas was performed for each of the platforms. Fig. 18 shows the average response time. Regarding the streaming systems, Kafka-ML offers slightly better results than Faust with both 1 and 32 replicas. In the case of TensorFlow Serving, the response time is much lower than Kafka-ML and Faust. It is true that TensorFlow Serving is not a streaming system like Kafka-ML but an RPC server, so this comparison might not be entirely fair. Moreover, TensorFlow Serving is a synchronous server, while Kafka-ML is asynchronous and abstracts both the information producers and the possible consumers and interested applications. With this we could have data sources sending information (e.g., IoT devices) and applications

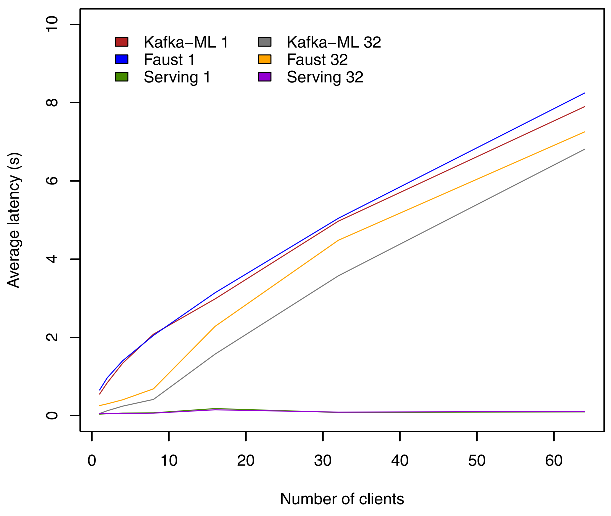


Fig. 18. Average throughput of inference with different number of clients (3 Kafka Brokers).

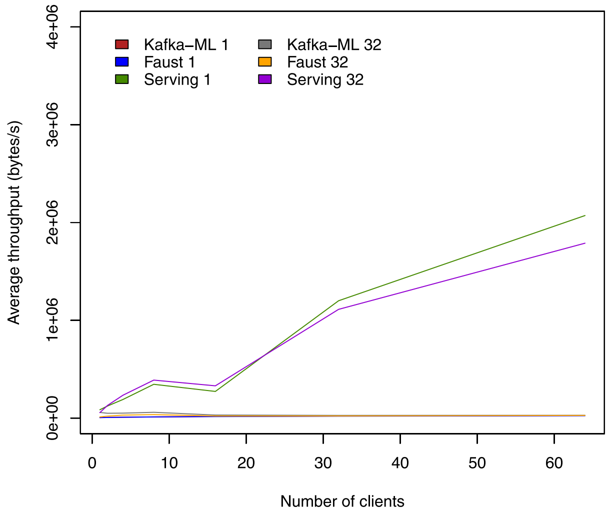


Fig. 19. Average throughput of inference with different number of clients (3 Kafka Brokers).

interested in prediction results working independently, unlike the client–server model of TensorFlow Serving. The management of data streams is also better done with Apache Kafka (and Kafka-ML) than working without any control (and any fault tolerance guarantee) with an RPC server. In this case, if TensorFlow Serving fails for whatever reason, there is no way to backup the data sent for later processing like Kafka-ML with the distributed log. Moreover, Kafka-ML provides other suitable features for real-world ML/AI applications such as the management of the training and inference steps. However, with this we show that TensorFlow Serving is optimized for production systems (but without data streams), and we will study in the future how we can integrate data streams and Kafka-ML with it. The same scenario happens with throughput (Fig. 19) as described for latency.

6.6. Kafka-ML training performance

In this last section, we will analyze how the number of the data stream sent affects the metrics of the ML models and the training time. For the evaluation, we used the ML model VGG16 and the CIFAR10 dataset also used for the inference performance tests. In this case, we have used only 25 epochs to compare the results since the training time is high as we will see later. We are aware of this, and that is why we want to extend Kafka-ML to allow distributed training and GPU support. The VGG16 model used is also a deep learning model with many layers that requires a lot of processing for training. Ninety percent of the data streams were intended for training and 10% for evaluation, a batch size of 10, and a learning rate scheduler with epoch decay have also been used in this case. Each test has been performed only once due to the training time.

Table 2
Validation metrics of the VGG16 model with a different number of data streams sent.

Size	Loss	Accuracy	Time (s)
6000	1.515	0.34	39
4000	1.704	0.360	26
2000	1.037	0.6	13
1000	1.163	0.5800	6

Fig. 20 shows the training loss of the VGG16 model regarding the training epochs with the 4 different data stream sizes used (60,000, 40,000, 20,000 and 10,000 respectively). As can be seen, as the number of data streams used for training decreases, the loss, in general, starts with a higher value and takes more epochs to converge. This may be due to the fact that having a smaller dataset makes it more difficult for the training to find patterns for the large number of layers used. Regarding accuracy (Fig. 21), the behavior is also similar. As the dataset is smaller, the accuracy takes more epochs to raise, and in this case, the learning rate update is noticeable in terms of accuracy after 10 epochs. The overall accuracy of this deep learning model is low because not many epochs have been used due to the time required. As described in Algorithm 1, validation is performed right after training. In this case, we do not obtain the results by epochs as during the training, and that is why no learning curves of the validation have been shown. Only the resulting metrics are obtained through the algorithm, which are shown in Table 2. The size used for validation shown in the table is 10% of the data stream sizes indicated previously. In general, as the size of the data stream used for validation increases, the accuracy decreases, which may indicate that the model needs further training. The time is also proportional to the number of data streams sent and significantly less than during training.

Finally, regarding the training time, it can be seen in Fig. 22 that the training time is proportional to the number of data streams sent. With the size of 60,000 data streams, the training process took 19.34 h (around 2780 s per epoch) while with 10,000 data streams it took 3.26 h (around 470 s per epoch).

7. Discussion

In this section, the main innovations of Kafka-ML and how this open-source framework addresses the main challenges presented in the introduction are discussed.

- *How can current ML/AI frameworks and their pipelines be integrated with continuous data streams?*

Kafka-ML provides an innovative and open-source framework to integrate data streams and ML/AI frameworks. To the best of our knowledge, Kafka-ML is the first open-source framework to do so. This framework enables the management of the pipeline of ML/AI applications, since an ML model is defined until it is trained and deployed to perform inference (Fig. 2). Kafka-ML manages the steps involved in the management of the pipeline of ML/AI applications such as container creation and data stream ingestion

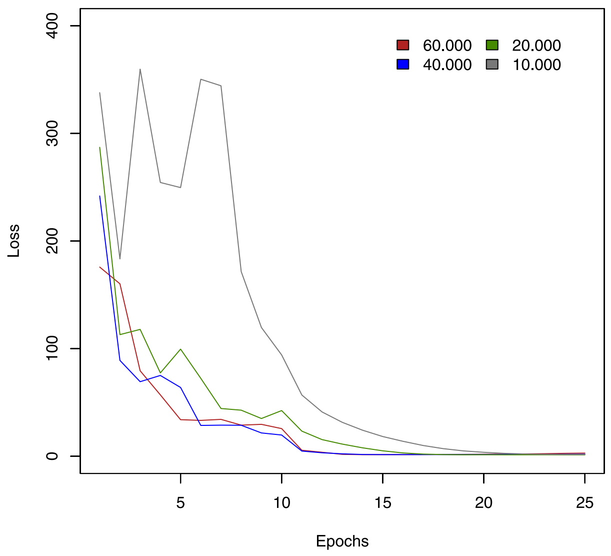


Fig. 20. Training loss of the VGG16 model with a different number of data streams sent.

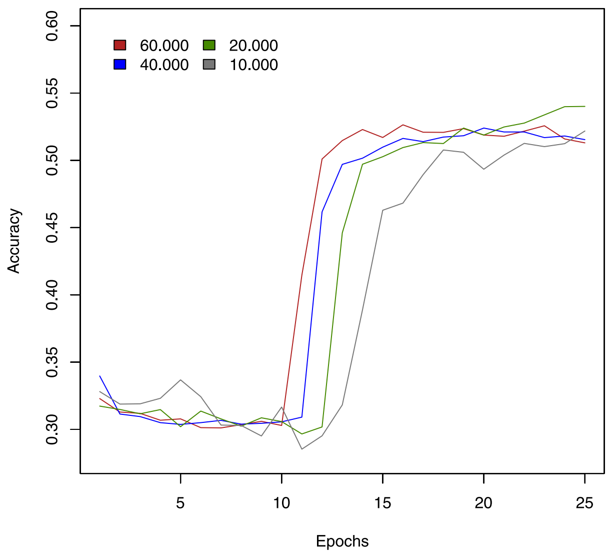


Fig. 21. Training accuracy of the VGG16 model with a different number of data streams sent.

transparently to end users. As future work, learning curves such as those seen in Section 6.6 will be made available in the Kafka-ML interface so that users can better assess when models are ready for inference through data streams.

- *How can an accessible and collaborative tool be achieved to evaluate and compare ML/AI model metrics and results?*

As shown in Fig. 9, once configurations are trained, users can visualize the metrics or even download the trained ML models in the Kafka-ML Web UI. Metrics are automatically inferred based on the ML model definition and are available in Kafka-ML right after training and evaluation. Moreover, users can easily group a set of ML models through configurations to evaluate their performance using a single data stream. The data stream adoption also opens the door to new synergies and data source integrations such as the IoT. The pipeline to manage the life cycle of ML models is intuitive and can be deployed in a shared and collaborative infrastructure. Therefore, this open-source and accessible framework

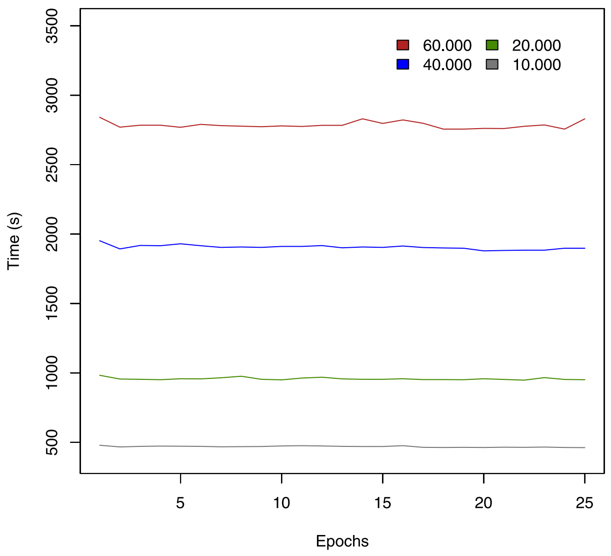


Fig. 22. Training time of the VGG16 model with a different number of data streams sent.

can be definitely used as a collaborative tool to evaluate the performance of ML models using data streams. It is noteworthy that this is just the initial phase of the work and Kafka-ML intends to improve its ML model and hyperparameter versioning system in the near future.

- *How can data streams be reused and combined in ML/AI tasks?*

The management of data streams in Kafka-ML, described in Section 4.7, enables effectively controlling of the received data streams, thus allowing their reusability. This exploits the distributed log of Apache Kafka, where consumers can freely move through to consult both current and historical data. A Kafka control topic is used to indicate Kafka-ML tasks like training in which topic and where exactly in the log data streams are available. Therefore, once data streams are sent for training, they can be reused by sending only a control message (a few bytes). Regarding inference, by specifying only the input topic, data streams can be reused among different deployed models. Furthermore, the data stream load among inference tasks can be distributed thanks to the group functionality of Apache Kafka and the replication possibility provided in the inference module as seen in the evaluation.

- *How can a solution be applied to portable and high availability architectures for production deployments?*

The Kafka-ML architecture and its components have been designed as microservices, which can be easily made portable and installed through Dockers containers. This reduces dependency problems installing software since each container specifies how and which software is required for building and running applications. Their lightweight also facilitates the portability of applications. However, Kafka-ML is not only intended to run with Docker containers but also in high availability and production environments. For this purpose, Kafka-ML is managed by a container orchestration solution, which is responsible for managing Kafka-ML components and their tasks, such as training and inference, ensuring the scalability of system when required and offering fault tolerance and high availability. It is noteworthy that the data stream management in Kafka-ML also provides support for fault tolerance regarding data streams through topic replicas and high availability through topic partitions and consumer groups, ensuring that there is no loss of information.

8. Conclusions and future work

In this paper, Kafka-ML, a novel and open-source framework for managing the pipeline of ML/AI applications through data streams, has been presented. Kafka-ML is characterized by its accessibility and ease of use since users need only a few lines of source code to create an ML model in its Web UI to control the ML/AI pipeline, creating configurations to evaluate different ML models, training, validating, and deploying trained models for inference. Therefore, Kafka-ML offers an innovative and open-source solution to manage the daily tasks performed by many ML/AI researchers and developers worldwide. All of this with a new approach based on data streams, which can be properly managed and losslessly stored in Kafka-ML. Moreover, a novel approach based on the distributed log of Apache Kafka has been adopted to have full control over received data streams in Kafka-ML, enabling its ML/AI applications to reuse these data streams and eliminating their dependency on data storage or file systems. Kafka-ML architecture and deployed jobs (training and inference) are fully containerized, enabling fault-tolerance and high availability for production environments. It is important to note that the approach applied with Apache Kafka in Kafka-ML can also be extrapolated to other systems and solutions. Kafka-ML is openly available in GitHub to be used and improved by both experts and non-experts on ML/AI adopting data streams.

As future work, we have pointed out the following challenges and improvements to Kafka-ML:

- Distributed inference. Deep neural network layers can be partitioned into multiple and independent ML models as well as through intermediary exits [60]. Their execution can be optimized in the fog, edge, and cloud computing paradigms. The objective is to enable the training and partitioning of ML models in Kafka-ML, so as to deploy them in the IoT-Cloud continuum [61]. New architectures to support the whole data flow between layers are also required.
- Distributed training. Currently, training is performed in a single container that may not be enough for large neural networks. Other approaches for distributed training in Kubernetes, such as Kubeflow and GPU support [62], should be explored in this regard.
- Support for other ML frameworks. This will depend on the developments of other ML frameworks to enable Apache Kafka, as TensorFlow did with TensorFlow/IO. In any case, new data stream connectors to other ML frameworks can be explored.
- IoT and ML/AI. IoT is taking place into the ML/AI pipeline as demonstrated by initiatives such as uTensor[17] and TensorFlow Lite[18] for on-device inference. The generation of ML models for IoT devices and even theirs installation from Kafka-ML could expand the ML/AI pipeline to the IoT.
- Integrate other processing tasks. Finally, many applications such as structural health monitoring may use ML/AI but also other statistical and processing tasks that may require the same data stream. Therefore, Kafka-ML could also manage these non-ML/AI tasks to integrate them with the data stream utilized.
- Improve the provenance of information. Linking ML models with datasets utilized, choice of hyperparameters, and any related information that could be useful for users to keep track of the applied transformations and model governance [63].
- Obtaining more results on the end-user's experience of using Kafka-ML. Four members of our research team have tested the Kafka-ML platform and have confirmed its usability and ease of use. As a future work, our idea is to collect the feedback of users outside of our group.

17 https://github.com/uTensor/uTensor.

18 https://www.tensorflow.org/lite.

CRediT authorship contribution statement

Cristian Martín: Software development, Conceptualization, First manuscript draft. **Peter Langendoerfer:** Supervised the research, Conceptualization, Manuscript review, Funding. **Pouya Soltani Zarrin:** Helped in integrating with the Exasens dataset, Use case redaction, Manuscript review. **Manuel Díaz:** Supervised the research, Conceptualization, Manuscript review, Funding. **Bartolomé Rubio:** Supervised the research, Conceptualization, Manuscript review, Funding.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is funded by the Spanish projects RT2018-099777-B-100 (“rFOG: Improving Latency and Reliability of Offloaded Computation to the FOG for Critical Services”), PY20_00788 (“IntegraDos: Providing Real-Time Services for the Internet of Things through Cloud Sensor Integration”) and UMA18FEDERJA-215 (“Advanced Monitoring System Based on Deep Learning Services in Fog”). Cristian Martín was with a postdoc grant from the Spanish project TIC-1572 (“MIsTIca: Critical Infrastructures Monitoring based on Wireless Technologies”) and his research stay at IHP has been funded through a mobility grant from the University of Malaga and IHP funding. Funding for open access charge: Universidad de Malaga/CBUA. We are grateful for the work of all the reviewers who have greatly contributed to improving the quality of this article. We would like to express our gratitude to Kai Wähner for his inspiration and ideas through numerous articles and GitHub repositories on Kafka and its combination with TensorFlow.

References

[1] Y. Lu, Artificial intelligence: a survey on evolution, models, applications and future trends, J. Manage. Anal. 6 (1) (2019) 1–29.

[2] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, S. Muthukrishnan, One trillion edges: Graph processing at facebook-scale, Proc. VLDB Endow. 8 (12) (2015) 1804–1815.

[3] M. Díaz, C. Martín, B. Rubio, State-of-the-art, challenges, and open issues in the integration of Internet of things and cloud computing, J. Netw. Comput. Appl. 67 (2016) 99–117.

[4] Internet of Things at a Glance, 2021, Available online: https://emarsonindia.com/wp-content/uploads/2020/02/Internet-of-Things.pdf, (accessed on 15 May 2021).

[5] G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, Á.L. García, I. Heredia, P. Malík, L. Hluchỳ, Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey, Artif. Intell. Rev. 52 (1) (2019) 77–124.

[6] Apache Kafka, 2021, Available online: http://kafka.apache.org/, (accessed on 13 May 2021).

[7] M. Vartak, S. Madden, MODELDB: Opportunities and challenges in managing machine learning models, IEEE Data Eng. Bull. 41 (4) (2018) 16–25.

[8] C. Weber, P. Reimann, MMP-a platform to manage machine learning models in industry 4.0 environments, in: 2020 IEEE 24th International Enterprise Distributed Object Computing Workshop (EDOCW), Oct. 5, Eindhoven, Netherlands, IEEE, 2020, pp. 91–94.

[9] H. Miao, A. Li, L.S. Davis, A. Deshpande, Towards unified data and lifecycle management for deep learning, in: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), April 19-22, San Diego, CA, USA, IEEE, 2017, pp. 571–582.
[10] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, D. Dennison, Hidden technical debt in machine learning systems, Adv. Neural Inf. Process. Syst. 28 (2015) 2503–2511.
[11] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436–444.
[12] N. Polyzotis, S. Roy, S.E. Whang, M. Zinkevich, Data lifecycle challenges in production machine learning: a survey, ACM SIGMOD Rec. 47 (2) (2018) 17–28.
[13] Kubeflow, 2021, Available online: https://www.kubeflow.org/, (accessed on 13 May 2021).
[14] L. Yeager, J. Bernauer, A. Gray, M. Houston, Digits: the deep learning gpu training system, in: ICML 2015 AutoML Workshop, July 11, Lille, France, 2015.
[15] E. Liberty, Z. Karnin, B. Xiang, L. Rouesnel, B. Coskun, R. Nallapati, J. Delgado, A. Sadoughi, Y. Astashonok, P. Das, et al., Elastic machine learning algorithms in amazon sagemaker, in: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, June 14-19, Portland, OR, USA, 2020, pp. 731–737.
[16] Algorithmia - The enterprise MLOps platform, 2021, Available online: https://algorithmia.com/, (accessed on 13 April 2021).
[17] Valohai - Machine learning management Platform, 2021, Available online: https://valohai.com/, (accessed on 13 April 2021).
[18] Google cloud AutoML, 2021, Available online: https://cloud.google.com/automl, (accessed on 13 May 2021).
[19] X. He, K. Zhao, X. Chu, AutoML: A survey of the State-of-the-Art, 2019, arXiv preprint arXiv:1908.00709.
[20] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, J. Stein, Building a replicated logging system with apache kafka, Proc. VLDB Endow. 8 (12) (2015) 1654–1655.
[21] C. Pahl, A. Brogi, J. Soldani, P. Jamshidi, Cloud container technologies: a state-of-the-art review, IEEE Trans. Cloud Comput. PP (2017) 1, http://dx.doi.org/10.1109/TCC.2017.2702586.
[22] C. Martín, D.R. Torres, M. Díaz, B. Rubio, Fogpi: A portable fog infrastructure through raspberry pis, in: 2020 9th Mediterranean Conference on Embedded Computing (MECO), 8-11 June, Budva, Montenegro, IEEE, 2020, pp. 1–3.
[23] Docker, 2021, Available online: https://www.docker.com, (accessed on 20 October 2021).
[24] Kubernetes, 2021, Available online: https://kubernetes.io/, (accessed on 15 May 2021).
[25] Amazon SageMaker - Machine learning for every data scientist and developer, 2021, Available online: https://aws.amazon.com/sagemaker/, (accessed on 13 April 2021).
[26] A. Bifet, G. Holmes, R. Kirkby, B. Pfahringer, MOA: Massive online analysis, J. Mach. Learn. Res. 11 (2010) 1601–1604, http://portal.acm.org/citation.cfm?id=1859903.
[27] J. Montiel, J. Read, A. Bifet, T. Abdessalem, Scikit-multiflow: A multi-output streaming framework, J. Mach. Learn. Res. 19 (72) (2018) 1–5, http://jmlr.org/papers/v19/18-251.html.
[28] A. Ullah, K. Muhammad, I.U. Haq, S.W. Baik, Action recognition using optimized deep autoencoder and CNN for surveillance data streams of non-stationary environments, Future Gener. Comput. Syst. 96 (2019) 386–397.
[29] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, S. Khan, A survey of distributed data stream processing frameworks, IEEE Access 7 (2019) 154300–154316.
[30] G. Van Dongen, D. Van den Poel, Evaluation of stream processing frameworks, IEEE Trans. Parallel Distrib. Syst. 31 (8) (2020) 1845–1858.
[31] Apache Samoa, 2021, Available online: https://samoa.incubator.apache.org/, (accessed on 13 May 2021).
[32] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink: Stream and batch processing in a single engine, Bull. IEEE Comput. Soc. Tech. Comm. Data Eng. 36 (4) (2015).
[33] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M.J. Franklin, et al., Apache spark: a unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65.
[34] M. Díaz, C. Martín, B. Rubio, λ-CoAP: AN internet of things and cloud computing integration based on the lambda architecture and coap, in: S. Guo, X. Liao, F. Liu, Y. Zhu (Eds.), Collaborative Computing: Networking, Applications, and Worksharing: 11th International Conference, CollaborateCom 2015, Wuhan, November 10-11, 2015, China. Proceedings, Springer International Publishing, Cham, 2016, pp. 195–206.
[35] S. Ge, H. Isah, F. Zulkernine, S. Khan, A scalable framework for multilevel streaming data analytics using deep learning, in: 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), July 15-19, Milwaukee, Wisconsin, USA, Vol. 2, IEEE, 2019, pp. 189–194.
[36] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J.M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al., Storm@ twitter, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, June 22-27, Snowbird, Utah, USA, 2014, pp. 147–156.
[37] S.A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, R.H. Campbell, Samza: stateful scalable stream processing at LinkedIn, Proc. VLDB Endow. 10 (12) (2017) 1634–1645, August 28 - September 1, Munich, Germany.
[38] M. Boehm, M.W. Dusenberry, D. Eriksson, A.V. Evfimievski, F.M. Manshadi, N. Pansare, B. Reinwald, F.R. Reiss, P. Sen, A.C. Surve, et al., Systemml: Declarative machine learning on spark, Proc. VLDB Endow. 9 (13) (2016) 1425–1436.
[39] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al., Mllib: Machine learning in apache spark, J. Mach. Learn. Res. 17 (1) (2016) 1235–1241.
[40] M.J. Sax, G. Wang, M. Weidlich, J.-C. Freytag, Streams and tables: Two sides of the same coin, in: Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics, August 27, Janeiro, Brazil, in: BIRTE '18, Association for Computing Machinery, New York, NY, USA, 2018, http://dx.doi.org/10.1145/3242153.3242155.
[41] Faust - Python Stream Processing, 2021, Available online: https://faust.readthedocs.io/, (accessed on 12 April 2021).
[42] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, J. Soyke, Tensorflow-serving: Flexible, high-performance ml serving, 2017, arXiv:1712.06139.
[43] M.D. de Assuncao, A. da Silva Veith, R. Buyya, Distributed data stream processing and edge computing: A survey on resource elasticity and future directions, J. Netw. Comput. Appl. 103 (2018) 1–17.
[44] S.B. Calo, M. Touna, D.C. Verma, A. Cullen, Edge computing architecture for applying AI to IoT, in: 2017 IEEE International Conference on Big Data (Big Data), Dec 11-14, Boston, MA, USA, 2017, pp. 3012–3016, http://dx.doi.org/10.1109/BigData.2017.8258272.
[45] H.P. Sajjad, K. Danniswara, A. Al-Shishtawy, V. Vlassov, Spanedge: Towards unifying stream processing over central and near-the-edge data centers, in: 2016 IEEE/ACM Symposium on Edge Computing (SEC), Oct. 27-28, Washington DC, USA, IEEE, 2016, pp. 168–178.
[46] F. Pisani, J.R. Brunetta, V.M. Do Rosario, E. Borin, Beyond the fog: Bringing cross-platform code execution to constrained iot devices, in: 2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Oct. 17-20, Campinas, Brazil, IEEE, 2017, pp. 17–24.
[47] M. Chiang, T. Zhang, Fog and IoT: An overview of research opportunities, IEEE Internet Things J. 3 (6) (2016) 854–864.
[48] OpenML: Machine learning together, 2021, Available online: https://www.openml.org/, (accessed on 13 May 2021).
[49] M.J. Smith, C. Sala, J.M. Kanter, K. Veeramachaneni, The machine learning bazaar: Harnessing the ml ecosystem for effective system development, in: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, June 14-19, Portland, OR, USA, 2020, pp. 785–800.
[50] A. Fard, A. Le, G. Larionov, W. Dhillon, C. Bear, Vertica-ML: Distributed machine learning in vertica database, in: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, June 14-19, Portland, OR, USA, 2020, pp. 755–768.
[51] N. Alshuqayran, N. Ali, R. Evans, A systematic mapping study in microservice architecture, in: 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), 4-6 Nov., Macau, China, IEEE, 2016, pp. 44–51.
[52] D. Vohra, Apache avro, in: Practical Hadoop Ecosystem, Springer, 2016, pp. 303–323.
[53] M. Delange, R. Aljundi, M. Masana, S. Parisot, X. Jia, A. Leonardis, G. Slabaugh, T. Tuytelaars, A continual learning survey: Defying forgetting in classification tasks, IEEE Trans. Pattern Anal. Mach. Intell. (2021) 1–1.
[54] P.S. Zarrin, N. Roeckendorf, C. Wenger, In-vitro classification of saliva samples of COPD patients and healthy controls using machine learning tools, IEEE Access 8 (2020) 168053–168060.
[55] P. Soltani Zarrin, F. Ibne Jamal, N. Roeckendorf, C. Wenger, Development of a portable dielectric biosensor for rapid detection of viscosity variations and its in vitro evaluations using saliva samples of COPD patients and healthy control, Healthcare 7 (1) (2019) 11.
[56] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: A system for large-scale machine learning, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), Nov. 2-4, Savannah, GA, USA, 2016, pp. 265–283.
[57] J.M. Perkel, Why Jupyter is data scientists' computational notebook of choice, Nature 563 (7732) (2018) 145–147.
[58] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, 2015, arXiv:1409.1556.

[59] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, Imagenet: A large-scale hierarchical image database, in: 2009 IEEE Conference on Computer Vision and Pattern Recognition, June 20-25, Miami, FL, USA, Ieee, 2009, pp. 248–255.

[60] S. Teerapittayanon, B. McDanel, H.-T. Kung, Distributed deep neural networks over the cloud, the edge and end devices, in: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), June 5–8, Atlanta, GA, USA, IEEE, 2017, pp. 328–339.

[61] V.B. Souza, X. Masip-Bruin, E. Marín-Tordera, S. Sànchez-López, J. Garcia, G.-J. Ren, A. Jukan, A.J. Ferrer, Towards a proper service placement in combined fog-to-cloud (F2C) architectures, Future Gener. Comput. Syst. 87 (2018) 1–15.

[62] T. Parnell, C. Dünner, K. Atasu, M. Sifalakis, H. Pozidis, Tera-scale coordinate descent on GPUs, Future Gener. Comput. Syst. 108 (2020) 1173–1191.

[63] V. Sridhar, S. Subramanian, D. Arteaga, S. Sundararaman, D. Roselli, N. Talagala, Model governance: Reducing the anarchy of production {ML}, in: 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18), July 11–13, Boston, MA, USA, 2018, pp. 351–358.

Cristian Martín received an M.Sc. in Computer Engineering, an M.Sc. in Software Engineering and Artificial Intelligence and a Ph.D. in Computer Science from the University of Málaga (Spain) in 2014, 2015 and 2018 respectively. Currently, he is a Postdoctoral Researcher at the University of Málaga. Previously, he has worked as a software engineer in various tech companies with RFID technology and software development. He is also a member of the ITIS software Institute of the University of Málaga. His research interests focus on the integration of the Internet of Things with Cloud/-Fog/Edge Computing, Machine Learning, Structural Health Monitoring and IoT Reliability.

Prof. Dr. Peter Langendörfer holds a diploma and a doctorate degree in computer science. Since 2000 he is with the IHP in Frankfurt (Oder). There, he is leading the wireless systems department. From 2012 till 2020 he was leading the chair for security in pervasive systems at the Technical University of Cottbus-Senftenberg. Since 2020 he owns the chair wireless systems at the Technical University of Cottbus-Senftenberg. He has published more than 150 refereed technical articles, filed 17 patents of which 10 have been granted already. He worked as guest editor for many renowned journals e.g. Wireless Communications and Mobile Computing (Wiley) and ACM Transactions on Internet Technology. Peter is highly interested in security for resource constraint devices, low power protocols, efficient implementations of A means and resilience.

Pouya Soltani Zarrin received the master's degree in Biomedical Engineering from Western University, Canada, and his Doctor of Engineering degree from the Brandenburg University of Technology, Germany. Currently, he is a Clinical Scientist at Profound Medical Inc., working on image-guided surgical interventions for cancer treatment. Previously, he worked as a Research Scientist at IHP Microelectronics on medical device development and AI integration for healthcare solutions. He has expertise in design, development, and testing of medical mechatronic systems and implementation of machine learning techniques for precision diagnostics. His research interests include medical device design, bio-sensing systems, AI, machine learning for healthcare, Image-guided robotic surgery, and medical mechatronic systems.

Manuel Díaz is Full Professor in the Computer Science Department at the University of Málaga and Head of the ERTIS research group and member of the ITIS software Institute. His research interests are in distributed and real time systems, Internet of Things and P2P, especially in the context of middleware platforms and critical systems. In the last years his main area of work has been focused on WSN and monitoring systems, especially in energy monitoring (FP7 e-balance project), water infrastructure monitoring (FP7 SAID project) and Energy Efficient Buildings (FP7 SEEDS). He has collaborated in many technology transfer projects with different companies such as Tecnatom, Telefónica, Indra or Abengoa. He was the coordinator of the FP6 SMEPP project and main researcher for UMA in the WSAN4CIP (ICT FP7). He is also co-founder of the spinoff Softcrits and responsible of its R&D department.

Bartolomé Rubio received his MS and Ph.D. degree in Computer Engineering from the University of Málaga in 1990 and 1998, respectively. From 1991 to 2000 he was an Assistant Professor at the Department of Languages and Computer Science of the University of Málaga. Since 2001 he has been an Associate Professor in the same department. He has been working in the areas of distributed and parallel programming and coordination models and languages. Currently, he is specially involved in the research fields of Wireless Sensor and Actor Networks and the Integration of Internet of Things and Cloud Computing. He has been a member of the Software Engineering group of the University of Málaga (GISUM) since its foundation and recently is a member of the ITIS software Institute of the University of Málaga.