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

Cristian Martín, Peter Langendoerfer, Manuel Díaz and Bartolomé Rubio

Abstract—Machine Learning (ML) and Artificial Intelligence (AI) have a dependency 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 into continuous data streams. However, most of the ML/AI frameworks used nowadays are not fully prepared for this revolution. In this paper, we proposed Kafka-ML, an open-source framework that enables the management of TensorFlow ML/AI pipelines through data streams (Apache Kafka). Kafka-ML provides an accessible and user-friendly Web UI where users can easily define ML models, to then train, evaluate and deploy them for inference. Kafka-ML itself and its deployed components are fully managed through containerization technologies, which ensure its portability and 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 lead to the (no) utilization of data storage and file systems.

Index Terms—Kafka-ML, Apache Kafka, Machine Learning, Artificial Intelligence, Data Streams, TensorFlow, Docker, Kubernetes

I. 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) are playing a decisive role in converting raw information into useful predictions and recommendations to improve both business operations and the life of citizens, among other. For instance, companies like Facebook process millions of photos every day to detect inappropriate content. This creates a continuous data stream of information that is facing ML/AI algorithms and systems.

More recently, with the proliferation of the Internet of Things (IoT), new sources of data have been enabled in the Internet era, with a forecast of 500 billion of connected devices by 2030. Paradigms such as Industry 4.0, Connected Cars and Smart Cities are being possible, and the most important, they contribute to the digitization of services and phenomena of the physical world. As a result, the data stream has continuously been increased and forecasts predict a huge expansion for coming years.

Traditionally, most of the ML/AI frameworks, which are behind the design and development of ML/AI algorithms, have not been designed to work with data streams, but with persistent datasets and static data. Even nowadays, popular Python frameworks like PyTorch, Theano and TensorFlow do not provide or only provide partial support for data stream systems like Apache Kafka, the most popular data stream system. This does not only include training of ML models, but also the rest of the steps that may be part of an ML/AI pipeline, such as inference, testing, and evaluation. To cope with this challenge, Kafka-ML, an open-source framework to manage ML/AI pipelines through data streams is presented.

Kafka-ML makes use of Apache Kafka and currently supports TensorFlow as ML framework to integrate data streams and ML/AI. However, the goal is to extend the support for ML/AI frameworks in the near future. Kafka-ML offers an accessible and user-friendly Web UI (following a similar to auto-ml initiatives) to manage the ML/AI pipeline for both experts and non-experts on ML/AI. Users just need to write a few lines of ML model code to train, compare, evaluate and do inference on their algorithms. Moreover, this framework makes use of a novel approach to manage data streams in Apache Kafka, that can be reused as many times as they are configured leading to the (no) need for any data storage or file system for datasets in Kafka-ML. Finally, Kafka-ML exploits containerization and container orchestration platforms to distribute the load of the system and facilitating the distribution of its components, in addition to providing fault-tolerance and high availability.

Therefore, the main contributions of this paper are:

1) The presentation of Kafka-ML, an open-source, accessible and user-friendly framework to manage ML/AI pipelines through data streams
2) A novel approach to manage the data streams of ML/AI pipelines with (no) need for data storage or file systems

The rest of the paper is organized as follows. Section II presents a background of Kafka-ML. In Section III the ML/AI pipeline of Kafka-ML is introduced. Then, in Section IV the Kafka-ML architecture and its components are presented. The approach for data stream management in Apache Kafka is presented in Section V. A validation of Kafka-ML is analyzed in Section VI and related work is discussed in Section VII. Lastly, our conclusions and future work are presented in Section VIII.

II. BACKGROUND

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

1https://github.com/ertis-research/kafka-ml
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 exchange 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, stream data goes to multiple systems like batch processing and stream processing, but also a low latency is required. Therefore, both features are required, and this is how Apache Kafka provides them:

- **Multi-customer distribution.** As a publish/subscribe system, Apache Kafka provides this functionality. However, thanks to its integration and support for a wide range of solutions like Apache Hadoop, Apache Storm, TensorFlow, etc., this feature is definitely more than possible.
- **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 the 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, on the contrary 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*, and enables consumers to go through the log as they have to. In some cases, like ML training in Kafka-ML, this feature is suitable since all data is processed all at once and whether a failure occurs during this process the customer can start again without losing any data and having to store it 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. Partition enables the log to be divided into smaller units and providing load balancing, and the topic replicas enable fault-tolerance. An Apache Kafka cluster is composed of a peer-to-peer network of Brokers that share partitions and replicas. When having a consumer group, partitions can be associated to customers enabling high dispatching rates. Apache Kafka also incorporates different policies such as ‘at most one’, ‘at least once’ and ‘exactly one’, which enables customized QoS policies for final applications.

Its popularity, its large number of implementations and integrations with many cloud computing systems, and its great acceptance in the community have converted Apache Kafka into the de facto solution for interconnecting systems, ingesting data and dispatching information.

### III. PIPELINE OF AN ML MODEL IN KAFKA-ML

In this section, we introduce the pipeline of an ML model in Kafka-ML from it is designed to it is ready, and trained, to make predictions from data streams. Fig. 1 depicts the pipeline and steps to be carried out: A) designing and defining the ML model; B) creating a training configuration of ML models, i.e., selecting a set of ML model(s) to be trained; C) deploying the configuration for training; D) ingesting the deployment with training and optionally evaluation stream data through Apache Kafka; E) deploying the trained model for inference; F) and finally, feeding the deployed trained model for inference to make predictions with data streams. As you can see, the pipeline can be automatized, and all the steps related to feeding the ML model (training and inference) are carried out with data streams. Datastores might be not needed anymore (Section V). In the following, each of such steps is detailed.

#### A. Designing and defining ML models

From the first moment, we wanted to make this step as simple as possible to let ML developers to focus on ML models instead of learning a new library or using complex pipelines. A tool that can enable easy testing and validation of ML models to ML developers would considerable facilitate their work and would let them focus on what they are expert on. For this reason, the only source code needed is the ML model definition itself in a popular ML framework as shown in Listing 1.

```
Listing 1. Example ML code for Kafka-ML.
model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, input_dim=100, activation=tf.nn.relu),
    tf.keras.layers.Dense(1, activation=tf.nn.sigmoid)])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Listing 1 source code may seem familiar. In fact, it is a simple Python TensorFlow/Keras model with a hidden layer, a single output and the compilation for training. Kafka-ML currently supports Python TensorFlow [5] due its support for Apache Kafka through TensorFlow/IO[6]. We hope the support for more ML frameworks to Apache Kafka to expand the support of Kafka-ML.

---

2https://www.tensorflow.org/io
Once the model is defined with your ML editor (e.g., Jupyter), the TensorFlow/Keras source code of the model can be inserted in the Kafka-ML Web UI (User Interface) for model creation as shown in Figure 2. Note that the model can also be defined directly on Kafka-ML, but it might be worth it to validate it beforehand in other and more powerful ML IDEs or editors. Other required functions for the model (if any) can be inserted in the imports field. Once submitted the model, the source code will be checked as a valid TensorFlow model and incorporated into Kafka-ML. If the model has been successfully defined, the pipeline can be continued to the next step.

C. Deploying the configuration for training

After setting some training parameters like the batch size, epochs and number of iterations, and optionally some parameters for evaluation in the Kafka-ML Web UI (Fig. 4), the configuration will be ready to be deployed for training. If so, a task will be deployed per Kafka-ML model. Then, one of the first steps that each deployed job carries out is fetching its corresponding ML model from the Kafka-ML model and loading it to start training. Finally, jobs can resume until a data stream with training and optionally evaluation data is received through Apache Kafka. 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.

D. Ingesting the deployment with stream data

Once the models are deployed and to continue the pipeline, the data stream for training has to be sent for the deployment. It can also be submitted before though. Since the Kafka stream connector expects to have the stream data when it initiates, the training cannot start until the data stream is available in Kafka. We have used at least two Kafka topics to overcome this: 1) data topic(s) which only contain training and evaluation data streams required for training and evaluation; 2) and a control topic, which informs deployed ML models through control messages when and where the data streams are available for training and evaluation. Section V will discuss this in detail. A control message should contain at least the following information:

- **deployment_id**: ID of the deployed configuration where the data stream goes.
- **topic**: Kafka topic where the training and evaluation data streams are.
- **input_format**: Format of the data stream (e.g., RAW, AVRO).
- **input_config**: Configuration required by the data format chosen (e.g., the scheme in Avro).
- **validation_rate**: Percentage of stream data 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.

Kafka-ML currently supports RAW format (suitable for single-input data streams that may request a reshape, like...
images) and Apache Avro (suitable for complex and multi-input datasets where a scheme specifies how the data stream is decoded), however, it is opened for the support of new data formats. In each case, the information for decoding 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 since they deal with Kafka-ML aspects like sending the control message when the data stream has been sent.

After dispatching the data stream with the libraries provided from an IoT device or gateway, a dataset or any information source to the corresponding deployment_id, all the ML models grouped in the configuration will start the training, and evaluation (if validation_rate > 0).

E. Deploying trained models for inference

Right after training and evaluation, both the trained model itself and the metrics defined (e.g., loss and accuracy) will be submitted by each training Job to the Kafka-ML architecture. Results can be visualized in the Kafka-ML Web UI as shown in Figure 6. For each result, users can edit it, download the trained model, or deploy it for inference.

In the inference deployment (Fig. 6), 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. Moreover, all the interactions are done through Apache Kafka, and users have to configure the input topic (for values to predict) and output topic (for predictions).

F. Ingesting the deployed trained models with stream data for inference

Finally, the ML/AI pipeline concludes once the trained model is ready and deployed to make predictions and recommendations through data streams. In this case, no control messages have to be sent since the input and output topics, and the input format and configuration have been previously defined in the Web-UI (Fig. 5). Users and systems just need to send encoded data streams with the data format defined to the input topic, and inference results will be immediately sent after model prediction to the output topic configured.

IV. KAFKA-ML ARCHITECTURE

The Kafka-ML architecture comprises a set of components based on the single-responsibility principle, comprising a microservice architecture. All of these components have been containerized so that they can run as Docker containers. This does not only enable easy portability of the architecture, isolation between instances and fast setup support for different platforms but also their management and monitoring through a container orchestration platform like Kubernetes [6]. Kubernetes enables continuous monitoring of containers 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. Kubernetes manages the life of cycle of Kafka-ML and its components. Kafka-ML is an open-source project and its implementation, configurations, Kubernetes deployment files and some examples can be found in our GitHub repository.

An overview of the Kafka-ML architecture is shown in Fig. 7 and below each component is detailed.

A. Front-end

The front-end provides a management Web UI where users can perform the operations available in Kafka-ML such as the creation of ML models and configurations and the deployment

3https://github.com/ertis-research/kafka-ml
of them for training and inference in a user-friendly and accessible way. The front-end makes use of the RESTful API offered by the back-end, and it has been 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-end (e.g., a smartphone application).

B. 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 in contact with the corresponding Kubernetes API to deploy and manage training and inference of configurations and ML models when ordered by users. Moreover, the back-end also receives the 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 along with the official Python client library for Kubernetes for the deployment and management of Kubernetes components.

C. Model Training

Once the back-end deploys a configuration, a Job, a deployable unit in Kubernetes, will be executed per Kafka-ML model for training and containerized as a Docker container. Algorithm 1 describes the procedure of the training Job. Note that some steps such as management of exceptions and data stream decoding have not been included for simplicity. Firstly, it downloads the ML model from the back-end. Next, it starts receiving the control stream until it receives the control stream expected, i.e., it matches the deployment_id received. Training and optionally evaluation are performed through the data stream received in the control stream message. Finally, the Job submits the trained model and the training and evaluation metrics to the back-end.

D. Model Inference

After training an ML model and deploying it for inference through the back-end, a Replication Controller (a Kubernetes component that ensures that a specified number of replicas are running at all times) with the replicas established will be executed in Kubernetes along with its corresponding Docker containers. Algorithm 2 describes the procedure of the inference in Kafka-ML. Once downloaded the trained model from the back-end, this component starts receiving stream data to then make a prediction with the stream received and sending it through the Kafka output topic configured. 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.

### Algorithm 1: Training algorithm in Kafka-ML

**Data:** model_url, training_kwargs, evaluation_kwargs, deployment_id, stream data

**Result:** Trained ML model and training and evaluation metrics

```python
model ← downloadModelFromBackend(model_url);
trained ← False;
while not trained do
    msg ← readControlStreams();
    if deployment_id == msg.deployment_id then
        training_stream ← readStream(msg.topic);
        if msg.validation_rate > 0 then
            training_stream ← take(data_stream,
                                  msg.validation_rate);
            evaluation_stream ← split(data_stream,
                                   msg.validation_rate);
        end
        training_res ← trainModel(model,
                                 training_kwars, training_stream);
        if msg.validation_rate > 0 then
            evaluation_res ← evaluateModel(model,
                                             evaluation_kwars, evaluation_stream);
        end
        uploadTrainedModelAndMetrics(model_url, model, training_res, evaluation_res);
        trained ← True;
    end
end
```

### Algorithm 2: Inference algorithm in Kafka-ML

**Data:** model_url, input_topic, output_topic, input_configuration, stream data

**Result:** Predictions to Apache Kafka

```python
model ← downloadTrainedModelFromBackend(model_url);
deserializer ← getDeserializer(input_configuration);
while True do
    stream ← readStreams(input_topic);
    data ← decode(deserializer, stream);
    predictions ← predict(model, data);
    sendDataToKafka(predictions, output_topic);
end
```
E. Control logger

The control logger component is just responsible to consume the control messages received in Kafka-ML, and send them to the back-end. These control messages are used for two purposes in the back-end: 1) allowing to send them again to other deployed configurations without the need to send the entire training and evaluation data stream, which is possible since Apache Kafka keeps the data streams in the distributed log; and 2) auto-configuring the inference input format and configuration with the information received in the 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, so this 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 and thereby this information.

F. Apache Kafka and ZooKeeper

To facilitate the deployment and management of Apache Kafka, and also to leverage the possibilities offered by Kubernetes, we have deployed Apache Kafka\(^1\) and Apache ZooKeeper\(^2\), as Jobs using Docker containers in Kubernetes. 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 the data streams.

V. DATA STREAM MANAGEMENT THROUGH APACHE KAFKA DISTRIBUTED LOG

As discussed in Section \[\] 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 has to recover all data stream. In traditional message queue systems where each message may be deleted after consumption, a datastore may be needed to enable other systems to send the data streams.

On the other hand, since data streams can be configured to be kept in the log, these streams can be reused for training into other deployed configurations and ML models without the need to send 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 retention policy established. An example of this functionality is shown in Fig. \[\]. 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 the consume the same data stream. In the current distributed log state, this data stream is expiring and cannot be longer reused to another deployed configuration. The data stream associated with the control message C2 has been sent to the deployed configurations D3 and D5 and can be still reused for new configurations that want to use this stream.

To allow previous management of the stream data, control messages do not only specify the topic where the data stream are but also what is their position in the distribution log. This follows the following list 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 70000. Through this control message, Kafka-ML informs deployed configurations where exactly are the data streams they are waiting for even using the same topic for all of them. In the Kafka-ML Web UI, a form is available where users can see the list of the data stream sent to Kafka-ML and send again the data stream to 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 to before Kafka will discard 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.

Note 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.

VI. VALIDATION

To validate Kafka-ML we have used the novel dataset for the classification of Chronic Obstructive Pulmonary Disease (COPD) patients and Healthy Controls \[\], available in this GitHub repo\[\]. This dataset uses multiple inputs such as age, smoking status and gender to predict the diagnosis (COPD-HC-Asthma-Infected) of patients. Therefore, an Avro encoding format was designed for both the training and label data. The source code of this validation is included in the examples of Kafka-ML\[\]. The Keras model source code was inserted as defined in COPD (Listing 2).

Listing 2. COPD Keras ML code used in Kafka-ML

\[
\texttt{model} = \texttt{tf.keras.Sequential}([\texttt{tf.keras.layers.Dropout}(0.2, \texttt{input_shape}=(3,))], \texttt{tf.keras.layers.Dense}(4, \texttt{activation}='sigmoid'), \texttt{tf.keras.layers.Dense}(2, \texttt{activation}='softmax'))
\]

\[\]

5\https://github.com/wurstmeister/kafka-docker
6\https://hub.docker.com/_/zookeeper
7\https://github.com/Pouya-SZ/HCOPD
8\https://github.com/eritis-research/kafka-ml/tree/master/examples/HCOPD_Avro_format
model.compile(tf.keras.optimizers.Adam(lr=0.0001), loss='sparse_categorical_crossentropy', metrics=['accuracy'])

Through this example, we have measured and evaluated the response time of Kafka-ML regarding data stream and containerization. Latency response has been measured to study the impact of training and inference in the following cases: 1) without the Kafka integration (no data streams); 2) with the data stream integration; 3) and with both the data stream integration and containerization. Note that the training response includes the data stream ingestion and the inference response includes the latency response between a data is sent until the prediction is received. Training has been performed with a batch size of 10 and this configuration introduced into the Kafka-ML Web-UI: “epochs=1000, steps_per_epoch=22, shuffle=True, verbose=0”.

The validation was performed on a single Kubernetes cluster running on a MacBook Pro laptop with 16GB. Latency response of the COPD dataset regarding training and inference are shown in Tables I and II respectively. In the case of data streams, the latency response can be admissible taking into account the advantages seen for the ML/AI pipeline. In the case of containerization, the latency is a little higher than data streams, especially for training. For inference is lower since Kafka is deployed in Kubernetes and thereby the network delay is smaller. We will study how to improve containerization training through distribution and GPU support.

| TABLE I | TRAINING LATENCY RESPONSE (S) |
| --- | --- | --- |
| Normal | Data streams | Data streams & containerization |
| 27,37 | 29,61 | 31,44 |

As performed with the COPD dataset and ML model, Kafka-ML can be used to manage the ML/AI pipeline of other ML works, facilitating its evaluation and data stream management, and its source code is openly available in GitHub for it.

**VII. RELATED WORK**

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

NVIDIA Deep Learning GPU Training System (DIGITS) [8] 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 itself, but it is a wrapper for NVCaffe, Torch and TensorFlow, which provides a Web interface to those 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) and the deployment of these tasks through containers for scaling, it has a dependency on GPUs and may require to write own source code on top of these frameworks.

Kubeflow [9] 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 (Fig. 9), there may be some steps that are not required in the Kafka-ML pipeline (Fig. 1), especially these ones that require to build...
containers for training and inference. In Kafka-ML, users just need to interact with the Web UI for training and inference. In addition, data stream support would have to be manually developed by Kubeflow ML developers and users. In Kafka-ML, the data stream management through Apache Kafka is supported in all the pipeline. In any case, Kubeflow provides great support for Kubernetes and ML multi-frameworks, and it is supported by a large ecosystem and community that is 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.

MOA [10] 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 well supporting (yet) online learning. On the other hand, Kafka-ML provides support for TensorFlow/Keras models, and its large community, instead of creating a new framework with own source code that could limit its adoption. Scikit-multiflow [11] is another framework for online learning, in this case for the popular framework scikit-learn, however it does not provide a Web interface nor a full control of an ML/AI pipeline.

Kafka-ML follows a different approach than other distributed data stream frameworks such as Apache SAMOA [12], Apache Flink [13] and the Lambda architecture [14], [15]. Apache SAMOA is currently undergoing incubation at Apache and aims to enable the development of ML algorithms through data streams without directly dealing with the complexity of underlying processing engines (e.g., Apache Storm and Apache Samza). Apache Flink provides a framework to perform computation over data streams at in-memory speed and at any scale. And 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, but they have limited support 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-tolerance ML/AI pipelines.

Finally, Kafka-ML is related somehow to Auto-ML projects such OpenML [16] and Google Cloud AutoML [17]. 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 (using configurations) in Kafka-ML. Moreover, data streams can also be managed and shared as seen in Section V 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 and user-friendly platform, where only a few lines of ML model source code are required to start an ML/AI pipeline. Furthermore, Kafka-ML is an open-source project available for both experts and non-experts on ML/AI.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, Kafka-ML, an open-source framework to manage the pipeline of ML/AI applications through data streams has been presented. Kafka-ML exploits the popular data stream system Apache Kafka and the Python ML framework TensorFlow to integrate both ML/AI and data streams. Kafka-ML is characterized by its accessibility and easy-use since with only a few lines source code, users can create an ML model in its Web UI to then control the ML/AI pipeline, creating configurations to evaluate different ML models, training, validating and deploying trained models for inference. Moreover, a novel approach based on the distributed log of Apache Kafka has been adopted to have full control of the data streams received in Kafka-ML, enabling its ML/AI applications to reuse these data streams and maybe removing their dependency on data storage or file systems. Kafka-ML is fully containerized, and its deployed components (training and inference) as well. Docker and Kubernetes are in charge of containerization and orchestrating the Kafka-ML architecture for fault-tolerance and high availability respectively. 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, and through intermediary exits [18], their execution can be optimized in the Fog, Edge and Cloud computing paradigms. The objective is to enable the training and partition of ML models in Kafka-ML, to later deploy them in the IoT-Cloud continuum. 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 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. The IoT is taking place into the ML/AI pipeline as demonstrated by initiatives such as uTensor[9] and TensorFlow lid[10] for on-device inference. The generation of ML models for IoT devices and even its installation from Kafka-ML could expand the ML/AI pipeline until 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 used.

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”) and UMA18FEDERIA-215 (“Advanced Monitoring System based on Deep Learning Services in Fog”). Cristian Martín is 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. We would like to express our anonymous gratitude to Kai Whner 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,” Journal of Management Analytics, vol. 6, no. 1, pp. 1–29, 2019.

[2] M. Díaz, C. Martín, and B. Rubio, “State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing,” Journal of Network and Computer Applications, vol. 67, pp. 99–117, 2016.

[3] “Internet of things at a glance,” Available online: https://emarsonindia.com/wp-content/uploads/2020/02/Internet-of-Things.pdf (accessed on 15 May 2020).

[4] “Apache kafka,” Available online: http://kafka.apache.org/ (accessed on 13 May 2020).

[5] 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, 2016, pp. 265–283.

[6] “Kubernetes,” Available online: https://kubernetes.io/ (accessed on 15 May 2020).

[7] P. Soltani Zarrin, F. Ihne Jamal, N. Roeckendorf, and 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, vol. 7, no. 1, p. 11, 2019.

[8] L. Yeager, J. Bernauer, A. Gray, and M. Houston, “Digits: the deep learning gpu training system,” in ICML 2015 AutoML Workshop, July 11, Lille, France, 2015.

[9] “Kubeflow,” Available online: https://www.kubeflow.org/ (accessed on 13 May 2020).

[10] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, “MOA: massive online analysis,” J. Mach. Learn. Res., vol. 11, pp. 1601–1604, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1859903

[11] J. Montiel, J. Read, A. Bifet, and T. Abdessalem, “Scikit-multiflow: A multi-output streaming framework,” Journal of Machine Learning Research, vol. 19, no. 72, pp. 1–5, 2018. [Online]. Available: http://jmlr.org/papers/v19/18-251.html

[12] “Apache samoa,” Available online: https://samoanterp.coa apache.org/ (accessed on 13 May 2020).

[13] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, “Apache flink: Stream and batch processing in a single engine,” Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, vol. 36, no. 4, 2015.

[14] M. Díaz, C. Martín, and B. Rubio, “\lambda-coap: An internet of things and cloud computing integration based on the lambda architecture and coap,” in Collaborative Computing: Networking, Applications, and Worksharing: 11th International Conference, CollaborateCom 2015, Wuhan, November 10-11, 2015. China. Proceedings, S. Guo, X. Liao, F. Liu, and Y. Zhu, Eds. Cham: Springer International Publishing, 2016, pp. 195–206.

[15] S. Ge, H. Isah, F. Zulkernine, and 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.

[16] “Openml: Machine learning together,” Available online: https://www.openml.org/ (accessed on 13 May 2020).

[17] “Google cloud automl,” Available online: https://cloud.google.com/automl (accessed on 13 May 2020).

[18] S. Teerapittayanon, B. McDanel, and 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 58, Atlanta, GA, USA. IEEE, 2017, pp. 328–339.