IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 13, 2021, accepted August 31, 2021, date of publication September 3, 2021, date of current version September 16, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3110291

Managing and Deploying Distributed and Deep Neural Models Through Kafka-ML in the Cloud-to-Things Continuum

ALEJANDRO CARNERO, CRISTIAN MARTÍN, DANIEL R. TORRES, DANIEL GARRIDO, MANUEL DÍAZ, AND BARTOLOMÉ RUBIO
ITIS Software Institute, University of Málaga, 29016 Málaga, Spain

Corresponding author: Cristian Martín (cmf@lcc.uma.es)

This work was supported by the Spanish Projects ''rFOG: Improving Latency and Reliability of Offloaded Computation to the FOG for Critical Services'' under Grant RT2018-099777-B-100, ''IntegraDos: Providing Real-Time Services for the Internet of Things through Cloud Sensor Integration'' under Grant PY20_00788, and ''Advanced Monitoring System Based on Deep Learning Services in Fog'' under Grant UMA18FEDERJA-215.

ABSTRACT The Internet of Things (IoT) is constantly growing, generating an uninterrupted data stream pipeline to monitor physical world information. Hence, Artificial Intelligence (AI) continuously evolves, improving life quality and business and academic activities. Kafka-ML is an open-source framework that focuses on managing Machine Learning (ML) and AI pipelines through data streams in production scenarios. Consequently, it facilitates Deep Neural Network (DNN) deployments in real-world applications. However, this framework does not consider the distribution of DNN models on the Cloud-to-Things Continuum. Distributed DNN significantly reduces latency, allocating the computational and network load between different infrastructures. In this work, we have extended our Kafka-ML framework to support the management and deployment of Distributed DNN throughout the Cloud-to-Things Continuum. Moreover, we have considered the possibility of including early exits in the Cloud-to-Things layers to provide immediate responses upon predictions. We have evaluated these new features by adapting and deploying the DNN model AlexNet in three different Cloud-to-Things scenarios. Experiments demonstrate that Kafka-ML can significantly improve response time and throughput by distributing DNN models throughout the Cloud-to-Things Continuum, compared to a Cloud-only deployment.

INDEX TERMS Distributed deep neural networks, data streams, cloud computing, fog/edge computing, machine learning, artificial intelligence.

I. INTRODUCTION

Over the last years, a great number of sources and fields have persistently procured tons of data [1]. Machine Learning (ML) and Artificial Intelligence (AI) [2] algorithms have assumed an essential role in their processing and allowed us to rely on practical tools in this computerized era. This is beneficial since making useful predictions and suggestions have drastically improved business activities and people's lives. For instance, well-known social media companies analyze a huge amount of information to detect what they deem as inappropriate content [3], [4]. Also healthcare [5], [6], structural health monitoring [7], [8], and education systems [9], among others, have advanced thanks to this progress. Regardless, all this implies a continuous data processing that often does take a long time to give an accurate answer, which may not be sufficient in most real-time cases.

More recently, the Internet of Things (IoT) [10] is responsible for most of the production of data streams. In fact, the number of these data sources are increasing and 500 billion associated gadgets are estimated by 2030 [11]. This means that the coming years will challenge current ML/AI algorithms because of the immeasurable number of input sources and the required response time. Most systems that rely on these algorithms work with persistent and static information rather than volatile, increasing, and continuous data streams. In consequence, they could run into great difficulties for years to come due not only to uncontrollable

The associate editor coordinating the review of this manuscript and approving it for publication was Qing Yang.

data sources, but also adaptability problems. Therefore, data stream technologies should be considered in outstanding frameworks [12], such as Tensorflow [13] and PyTorch [14], which currently offer limited support for these tools.

To overcome previous challenges, we developed Kafka-ML [15], an open-source[1] framework to manage ML/AI pipelines through data streams. Kafka-ML is based on the popular and distributed message system Apache Kafka [16] for data stream dispatch and adopts a microservice and containerization architecture for its deployment in production environments. In the inference phase, ML/AI algorithms in Kafka-ML can be deployed as single processing units, i.e., the whole ML model, but they can also be distributed in clusters for fault tolerance and high availability. However, in situations where a low latency is required it is desirable to have faster responses as close as possible to where the information is produced (e.g., in the Edge or the Fog). It is unquestionable that the latency offered by paradigms such as cloud computing might not satisfy the requirements of low-latency applications, hence both Edge and Fog computing [17] are contributing to these use cases. Moreover, ML/AI algorithms, especially Deep Neural Networks (DNN), usually need a large number of computational resources and specialized hardware (e.g., GPUs), which may not satisfy the requirements of resource-constrained systems. For this reason, DNN are partitioned along the Cloud-to-Things Continuum in order to have smaller processing units (and sub-models) distributed. The Cloud-to-Things continuum (Figure 1) can be defined as a set of processing units, such as fog servers and edge devices, located between the IoT and the Cloud. Those processing units optimize bandwidth consumption and response times for time-sensitive applications. For instance, a possible deployment in this context could place IoT devices generating data streams connected to edge devices, then edge devices connected to fog servers for intermediary processing before sending the information generated to the Cloud. These processing units can allocate DNN, whose sub-models may present less accuracy at the lowest layers, but they do reduce the response time [18]. This paradigm is known as Distributed and Deep Neural Networks (DDNN) [19].

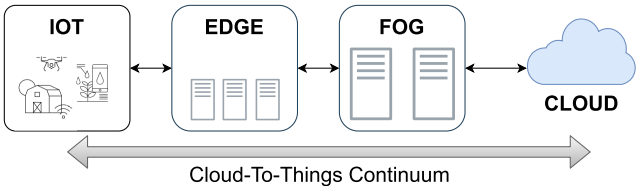

FIGURE 1. Cloud-to-things continuum.

As our framework Kafka-ML was only prepared to work with DNN and not with DDNN applications, in this paper, we introduce an extension which expands the functionality of Kafka-ML by enabling the use of DDNN in its life cycle. This new version offers the same features as Kafka-ML, such as the complete management pipeline of DDNN applications and the data stream management. In addition, it supports the training of DDNN models, the incorporation or early exits, and the deployment of their sub-models for latency optimization in distributed and heterogeneous clusters in the Cloud-to-Things Continuum. In [15], we presented the Kafka-ML framework and in [18] the architecture for connecting DDNN over the Cloud-to-Things Continuum. In this work, we focus on the extensions of Kafka-ML for enabling DDNN applications and managing its life cycle, in addition to further evaluations of a DDNN application in different layers over the Cloud-to-Things Continuum.

The rest of the article is organized as follows. Section II presents the related work. Section III explains the motivation to do this work. The design of distributed models is presented in Section IV. In Section V the ML/AI pipeline of distributed ML models in Kafka-ML is detailed. Then, in Section VI the Kafka-ML architecture for distributed ML models and its components are presented. Section VII shows the results of the validation carried out to evaluate the framework. And finally, Section VIII presents our conclusions and future possible works.

II. RELATED WORK

Kubeflow [20], an open-source machine learning toolkit for Kubernetes, allows configuring multiple steps of machine learning and artificial intelligence pipelines, such as hyper-parameters and pre-processing. Nonetheless, the flexible support for DDNN over the Cloud-to-Things Continuum and data streams is still out of this solution design. Edgelens [21] and HealthFog [22], as frameworks, deploy deep learning-based applications in Edge-Fog-Cloud environments and improve their Quality of Service (QoS). Both of them run non-distributed machine learning models, rather than adjust DDNNs to the continuum. Moreover, Edgelens scales down in resolution to reduce latency. IoTEF [23], a fault-tolerant architecture, manages and monitors cloud and edge clusters in a unified way. However, it still needs to automatize the distribution of DDNN in order to accomplish other QoS characteristics, such as low latency and optimization of inference processes beyond fault tolerance.

Kafka-ML follows a different approach than other distributed data stream frameworks such as Apache Storm, Apache Spark streaming, and Apache Flink [24]. Those provide frameworks to perform distributed computation over data streams at any scale. Apache SAMOA [25] is another project that aims to enable the development of ML algorithms through data streams without directly dealing with the complexity of underlying processing engines such as Apache Storm and Apache Samza. In general, these frameworks provide distributed engines for distributing computation in general with data streams, but they do not have a special focus or have limited support for facilitating popular ML/AI frameworks (e.g., TensorFlow) and ML/AI pipelines as Kafka-ML does.

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

Teerapittayanon *et al.* [26] introduced the concept of early exits with their BranchyNet approach. This concept allows inferences to end in middle layers, rather than go through the whole deep neural network model. The final accuracy could decrease when using early exits, but it is acceptable considering the significant reduction of response time. Partitioning deep neural networks based on these early exits reduce considerably the resource consumption over the Cloud-to-Things Continuum, and thus, the DDNN that follow this concept have a higher impact in reducing response time for time-sensitive applications as they also diminish the sending of messages to the highest levels of the continuum.

Additionally, there are studies that, by combining Cloud and Edge environments, show that the response time is accelerated while the network congestion is reduced [27]. Notwithstanding, along the continuum, Fog plays a role in addition to Edge and Cloud. This led to DINA [28], a fine-grained solution based on matching theory for dynamic DDNN partitioning in fog networks. Regardless, early exits, as BranchyNet [26] proposes, need to be considered since the inference early stops at middle layers reduce not only response time, but also network traffic [29] and computing capacity [30].

DIANNE [31] provides a flexible and modular framework for deep learning applications by splitting DNN into its elementary operations (e.g., softmax outputs and hidden layers) and implementing these as services that can be distributed across multiple and heterogeneous devices. Despite the great modularity of this approach, it is limited to self-developed components and not widely used frameworks, which may limit the adoption of the solution. Moreover, DIANNE does not consider Branchynet early exits for optimizing communication between layers.

An adaptive surgery scheme [32] dynamically splits DDNN to optimize both the latency and throughput under variable network conditions between the Edge and the Cloud. This approach does not consider the cases where the inference stops at the middle layers (early exits). In [33], an optimization algorithm to find an optimal partition that minimizes the inference time for BranchyNet networks is proposed. AAIoT [30] proposes a neural network segmentation method to optimize the computation allocation of inference tasks in multi-layer IoT systems. Edgent [34] presents a framework for dynamic DNN collaborative inference in Edge computing, combining DDNN dynamic portioning and Branchynet early exits to maximize the inference accuracy and optimizing the latency response.

The main drawbacks of previous approaches are: 1) they do not provide open-source implementations for the management of DDNN like Kafka-ML; 2) they do not consider current used ML frameworks like TensorFlow for a better extrapolation of the results; 3) some of them perform only simulations results; and, last but not least, 4) they do not consider fault tolerance or load balancing for infrastructures with variable conditions and data streams inputs like the IoT.

III. MOTIVATION

Deep neural network models consist of a large number of hidden layers. The distributed deployment [21] of these tends to improve their response time [27], especially in high-complexity, critical problems which need real-time responses. Smaller neural networks significantly reduce the response delay and can fit in devices with resource-constrained limitations. However, they may also decrease the accuracy, and thus, the reliability of these critical systems, which may require greater confidence in the predictions. In this sense, a tradeoff between accuracy, latency, and device requirements can be present in the partitioning of deep neural networks. Therefore, instead of having IoT devices and applications that collect and send information from the environment to the Cloud, which processes it with all the latency and bandwidth consumption problems involved as previously envisaged in Kafka-ML; in this work we envisage distributed models that can be allocated in the continuum from the IoT to the Cloud, including layers and paradigms such as Edge and Fog computing. In these layers, DDNN will be dispatched to provide time-sensitive responses as close as possible to where the information is produced (IoT devices) when predictions are accurate enough.

Despite the fact that allocating edge and fog layers in the continuum improves latency and bandwidth consumption problems, as well as reducing response times due to workload distribution, these layers may be overloaded if they are used to run computationally hard inferences hundreds of times per minute. This overload can be handled by using partitioned models based on BranchyNet, which considers what are called early exits. Therefore, a large model could return an accurate answer in early layers rather than at the end of the model. Additionally, early exits allow us to partition and distribute these models over different architecture layers in the Cloud-to-Things Continuum. As a consequence, intermediate edge and fog layers could infer from a small part of a model and, if the prediction is accurate enough, return a response without the need to wait for a Cloud response.

On the other hand, the management and deployment of infrastructures in the Cloud-to-Things Continuum can pose several challenges due to the dynamic nature of its components. In these scenarios, an adequate management and fault-tolerant guaranties represent a must for the deployment of DDNN. Moreover, due to the mobility of its nodes, DDNN should be easily portable to avoid large delays. In this sense, containerization technologies represent a suitable approach for the deployment of DNN applications as demonstrated with Kafka-ML, an approach which should be addressed to manage the multi-layer infrastructures presented in the continuum and the deployment of DDNN.

To sum up, we aim to extend the functionality provided by Kafka-ML in order to allow it to work with DDNN and early exits in the Cloud-to-Things Continuum, and also to support the management, deployment and sharing of distributed ML

```
edge_input = keras.Input(shape=64,
    name='edge_input')
x = layers.Dense(64,
    activation=tf.nn.relu,
    name='relu')(edge_input)
output_to_cloud = layers.Dense(64,
    activation=tf.nn.relu,
    name='output_to_cloud')(x)
edge_output = layers.Dense(10,
    activation=tf.nn.softmax,
    name='edge_output')(output_to_cloud)
edge_model=keras.Model(inputs=[edge_input],
    outputs=[output_to_cloud, edge_output],
    name='edge_model')

cloud_input = keras.Input(shape=64,
    name='cloud_input')
x1 = layers.Dense(64,
    activation=tf.nn.relu,
    name='relu1')(cloud_input)
x2 = layers.Dense(128,
    activation=tf.nn.relu,
    name='relu2')(x1)
cloud_output = layers.Dense(10,
    activation=tf.nn.softmax,
    name='cloud_output')(x2)
cloud_model = keras.Model(inputs=
    cloud_input,
    outputs=[cloud_output],
    name='cloud_model')
```

Listing 1. Distributed ML models example for Kafka-ML.

```
input = keras.Input(shape=64, name='input')
edge = edge_model(input)
cloud = cloud_model(edge[0])
model = keras.Model(inputs=[input],
 outputs=[edge[1], cloud], name='model')
model.compile(optimizer='adam',
    loss={
'edge_model':
    'sparse_categorical_crossentropy',
'cloud_model':
    'sparse_categorical_crossentropy'
    },
    metrics=['accuracy'],
    loss_weights=[0.001, 0.001])
```

Listing 2. Entire network definition example.

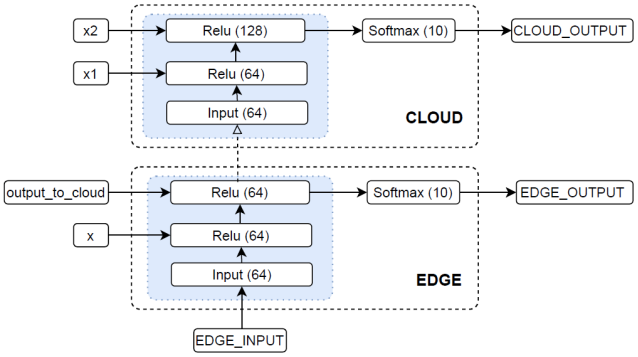

FIGURE 2. Example of a distributed neural network.

models, metrics, and results in high-performance and high-availability infrastructures.

IV. DEFINITION OF DISTRIBUTED MODELS IN KAFKA-ML

In order to have a better understanding of how distributed models are and can be designed in Kafka-ML, it will be explained how to define a distributed model step by step, as well as the necessary parts to connect it in the ML framework TensorFlow.

Listing 1 and 2 source codes may seem familiar. In fact, they are simple Python TensorFlow/Keras model definitions with certain hidden layers, outputs, and the compilation for training. At present, Kafka-ML integrations with different ML frameworks are still under development in order to expand its domain. Currently, TensorFlow/Keras is supported. Figure 2 shows an overview of the distributed deep neural model used for this example.

The first thing to consider when creating distributed models is the design of the structure of the global network that we want to implement, i.e., defining the distributed sub-models that will be part of it. In this case, an example is presented in Listing 1, in which the global model is made up of two different sub-models, which are two partitions to be deployed at Edge and the Cloud. Once the architecture has been decided, users, like in other ML frameworks, have to specify how the internal structure of each one of the distributed sub-models will be like. This structure consists of its input layer, its hidden layers, and its output layers. In this example, the sub-models have an equal input layer (*edge_input* and *cloud_input*) of shape 64 in order to accept images. Thus, the Edge model consists of only one hidden layer of 64 neurons (targeted for resource-constrained devices), followed by two output layers (*output_to_cloud* and *edge_output*). These represent the connecting layer to the Cloud and the early exit at the Edge respectively. These layers have to be specified as outputs in the configuration of the global Edge model. The Cloud model consists of two hidden layers with 64 and 128 neurons (*x1* and *x2* layers) followed by one output layer (*cloud_output*). In this case, there is no need of an early exit, since it is the last layer in this continuum. A notable aspect of these distributed models is that they allow obtaining partial outputs (predictions), as the information is processed throughout the global network. This is achieved by establishing different types of outputs within each layer of the continuum infrastructure (Edge and Cloud). For the example, the *softmax* output layer *edge_output* of the Edge model is the one which produces the intermediate prediction in the Edge, and *cloud_output* produces the final output in the Cloud. Another important output is the one that is responsible for

sending the information to the next layer of the distributed model so that it can continue to be processed (here is where the union of the distributed sub-models is performed). In our case, the ReLU output layer *output_to_cloud* of the Edge model carries out this task, which will be linked to the input layer (*cloud_input*) of the Cloud model. For this reason, *output_to_cloud* and *cloud_input* layers should have the same dimension.

Finally, for training purposes, a global neural network has to be created including each of the previously defined distributed sub-models as shown in Listing 2. As done so far for the sub-models, the inputs and outputs of the global model must be specified. However, here there is a special consideration, since the outputs and the inputs of the corresponding models have to be manually joined, following the same structure that was used to define each of the distributed sub-models (i.e., the order in which the outputs were defined for each of them). In this example, the outputs are the early exit (*edge[1]*: *edge_output*) at the Edge, and the Cloud output. There is only one input that corresponds to the Edge input. This process is performed automatically by Kafka-ML during training. In addition, we must ensure that the shape of the output tensors of the previous sub-models are the same as the shape of the input tensors of the subsequent sub-models. In this case, the shape of the output layer *output_to_cloud* of the Edge model must be the same as the shape of the input layer *cloud_input* of the Cloud model, which is 64 neurons. Finally, we will have to compile the global model by establishing a series of parameters that may vary, such as the optimizer, the types of the losses and the desired metrics. One important thing when defining a global neural network with different sub-models is that we have to specify one type of loss for each one of the distributed sub-models as well as the same number of *loss_weights* as the number of sub-models that we have. This example can be extrapolated to another number of distributed models and early exits.

V. PIPELINE OF A DISTRIBUTED ML MODEL IN KAFKA-ML

In this section, the pipeline of a distributed ML model in Kafka-ML is presented, including its life cycle. Before that, let us introduce an important concept in Kafka-ML: the 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 compare and evaluate metrics (e.g., accuracy and loss) 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. Since ML models in configurations will be trained with the same data, they should have the same similarities such as the same input layer. Note that a configuration can also include only one ML model.

The steps of the Kafka-ML pipeline are the following: 1) implementation and registration of distributed ML models; 2) creation of training configurations for a set of ML models to be trained; 3) deployments of a specific training configuration with certain parameters; 4) feeding the deployed configuration with training data streams; 5) obtaining and comparing the result metrics from the training phase; 6) deploying a selected ML model for inference and prediction making; and lastly, 7) inference phase to feed the deployed trained models with data streams to obtain their predicted results. Figure 3 shows the full sequence of the pipeline steps. Next, each of the steps is detailed.

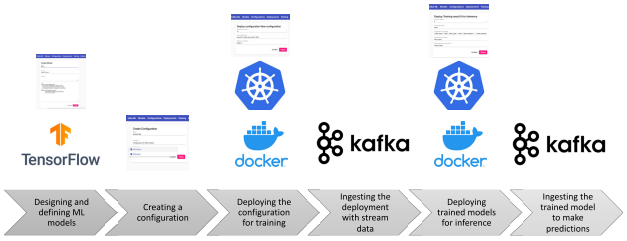

FIGURE 3. ML/AI pipeline in distributed Kafka-ML.

A. IMPLEMENTATION AND REGISTRATION OF THE DISTRIBUTED ML MODELS

From the beginning, we wanted to make Kafka-ML as simple as possible so we would let programmers to focus on what matters most to them, which is usually the creation of ML models. This is the reason why we developed a useful tool that enables easy testing and validation of ML models. Therefore, the only source code needed are the distributed ML models definitions as shown in Listing 1. On the other hand, we do not need to specify the entire network definition, as shown in Listing 2. This task will be performed automatically by Kafka-ML at the training phase.

Once the whole model is defined, we can split its sub-models and insert them into the Kafka-ML Web UI for model creation as shown in Figure 4. It is possible to define models directly in Kafka-ML, but it is advisable to use

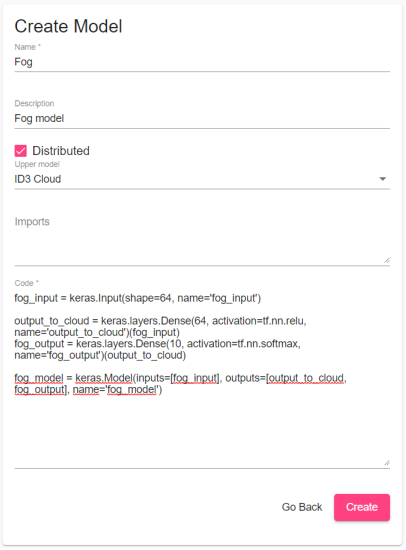

FIGURE 4. Definition of a distributed ML model in Kafka-ML.

an external and more powerful ML Integrated Development Environment (IDE) or editors such as Jupyter. If we need to add any other required function to the model, it can be inserted in the *imports* field. The form also has two fields related to the distribution: the first one indicates if we are creating a distributed model or not, and the second one specifies the next ML distributed model connected to this model. Once the model is submitted, its source code will be checked as a valid model and incorporated into Kafka-ML. If the model has been successfully created, the pipeline can be continued to the next step.

B. CREATION OF TRAINING CONFIGURATIONS

A configuration is simply a set of ML models to be deployed and trained together afterwards. To add a whole distributed model, only the sub-model which is placed at the top of its structure has to be selected, and Kafka-ML will automatically add the rest of the distributed chain to the configuration. A configuration can be created in the Kafka-ML Web UI as shown in Figure 5.

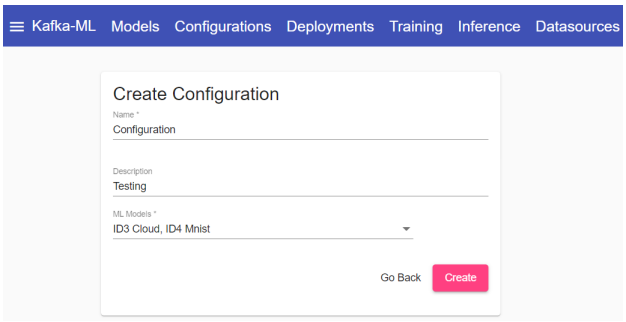

FIGURE 5. Creation of a configuration in Kafka-ML.

C. DEPLOYING THE CONFIGURATION FOR TRAINING

The first thing to deploy a configuration is to establish some parameters related to the training and evaluation phases, which are the batch size, epochs, and number of iterations. Then, if a user submits it, a task per Kafka-ML model (global) will be deployed. Note that all the sub-models will be trained in the same task. The first step carried out by each job deployed is to fetch its corresponding ML model (and sub-models) from the Kafka-ML existing models and load it to start training. Eventually, jobs wait until a data stream with training and optionally evaluation data is received through Apache Kafka. This allows us to have ready-to-train models and train them directly if there is already a data stream available in Kafka. A deployment can be created in the Kafka-ML Web UI as shown in Figure 6.

D. FEEDING THE DEPLOYMENT WITH DATA STREAM

The next step in the pipeline is to send the data streams to the deployments for training. The training phase will not start until the data stream is available in Kafka as the Kafka stream connector expects to have it at the beginning. Two

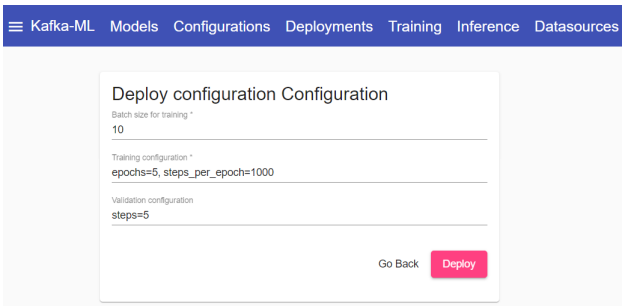

FIGURE 6. Deploying a configuration for training in Kafka-ML.

Kafka topics have been used for this purpose: the first one is the data topic itself, which only contains training and evaluation data streams for the training and evaluation phases; and the second one is the control topic, used to specify the deployed ML models through control messages and *when* and *where* (data topic) the data streams are available for training and evaluation. Control messages are further detailed in Kafka-ML [15].

After sending a data stream (e.g., from an IoT device) with the libraries provided in Kafka-ML (AVRO and RAW) to the corresponding configuration deployment, all the ML models included in the configuration will start the training.

E. OBTAINING THE METRICS RESULTING FROM THE TRAINING

Right after the training and evaluation are performed, Kafka-ML users will be able to visualize the defined metrics (e.g., loss and accuracy) for each trained model (and sub-model) in the Kafka-ML Web UI as shown in Figure 7. In that form, the results submitted by each training job to the Kafka-ML architecture are displayed. For each model, users can download the trained model, delete it, or deploy it for inference (next phase).

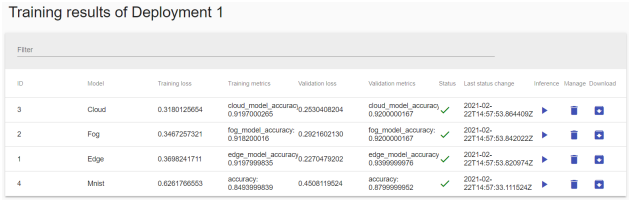

FIGURE 7. Training management and visualization in Kafka-ML.

F. DEPLOYING AN ML MODEL FOR INFERENCE

Once the ML models are trained, they can be deployed for inference using the Kafka-ML Web UI as shown in Figure 8. This form will ask for the number of inference replicas to be deployed. Replicas enable load balancing and fault tolerance among inference deployments. Users will have to specify some Kafka topics in the form: 1) input topic, for values to predict on; 2) output topic, for predictions; and, in case we are working with a distributed ML model, 3) upper topic, which is used to send the output prediction information to the upper model so it can continue the prediction flow. Topics are the

FIGURE 8. Deploying a distributed trained ML model for inference in Kafka-ML.

elementary way to allow producers (data sources) and consumers (training and inference jobs) to interact with Apache Kafka. Each topic is uniquely identified with a name. Kubernetes credentials can also be included for the deployment of the model (or sub-model) into another Kubernetes cluster in the continuum. Users will also have to define a prediction limit. This limit will be used to control the accuracy level in sub-models and, from that, elevate or not the prediction to the next layer. Therefore, if the accuracy is higher than the threshold, the output can be sent to the available early exit finishing the prediction flow.

G. FEEDING THE DEPLOYED TRAINED MODELS WITH DATA STREAM FOR INFERENCE

Finally, the ML/AI pipeline ends once the trained model is ready and deployed to make predictions through data streams. In this case, it is not necessary to send control messages, because the necessary information for inference (e.g., input and output topics) was already defined in the Kafka-ML Web UI (Fig. 8). Users and systems only have to send the data streams with the chosen data format to the input topic, and then the results of the inference will be sent immediately to the output topic once models make their predictions.

VI. KAFKA-ML ARCHITECTURE FOR DISTRIBUTED ML MODELS

Kafka-ML is a novel and open-source framework to manage ML/AI pipelines through data streams. In this work the Kafka-ML architecture has been extended to enable the management and deployment of distributed DNN models in the Cloud-to-Things Continuum. As seen in the previous section, Kafka-ML offers an open-source and easy-to-follow Web User Interface (UI) to deal with the ML/AI pipeline for both specialists and non-specialists on ML/AI. Users simply need to compose a couple of lines of ML model code to train, evaluate, assess, and perform inference on their algorithms. In addition, this structure utilizes a novel way to deal with data streams in Apache Kafka, that can be reused and so avoids the requirement for any information system or data storage for datasets in Kafka-ML (we will discuss this further in Section VI-E).

The Kafka-ML architecture encompasses a set of components based on the principle of single responsibility, that comprises a microservice architecture [35]. All of these components have been containerized so that they will run as Docker containers, enabling component isolation and portability. The management of a cluster of nodes and the deployment of Kafka-ML in distributed and production infrastructures are achieved through Kubernetes. Kubernetes enables continuous monitoring of the containers and their replicas to ensure that they continually match the state defined for them, as well as enabling other features for production environments such as high availability and load balancing. Kubernetes manages the lifecycle of Kafka-ML and its components. Kafka-ML for distributed ML models is an open-source project, and its implementation, configurations, Kubernetes deployment files and some examples can be found at our GitHub repository.[2] An overview of the Kafka-ML architecture, the new components, and the ones modified regarding the previous version of Kafka-ML is shown in Figure 9. In this overview, Kafka-ML is deployed in a Kubernetes cluster in the Cloud and sub-models are deployed in clusters along the Edge, the Fog, and the Cloud. The new components created in this architecture are shown in blue and the components modified in orange. The control logger is responsible for managing the control messages and sending them to the back-end. Apache Kafka and Zookeeper (required by Kafka) are also deployed as Docker containers and managed through Kubernetes.

A. BACKEND

The back-end component provides a RESTful API to manage all the information contained in Kafka-ML. This component works with the Kubernetes API to manage and deploy all the components needed in the distributed pipeline. This component was built using the Python web framework Django and the official Python client library for Kubernetes[3] for the deployment and management of Kubernetes components. Among the differences with the previous version of Kafka-ML are those changes made in the module itself that incorporate the new functionalities necessary to work with distributed neural networks (e.g., model and configuration definitions), as well as the new components responsible for carrying out the training and the inference phases of the distributed models.

B. FRONTEND

The front-end component offers a web user interface where users can manage all the information and functionality in

[2]https://github.com/ertis-research/kafka-ml
[3]https://github.com/kubernetes-client/python

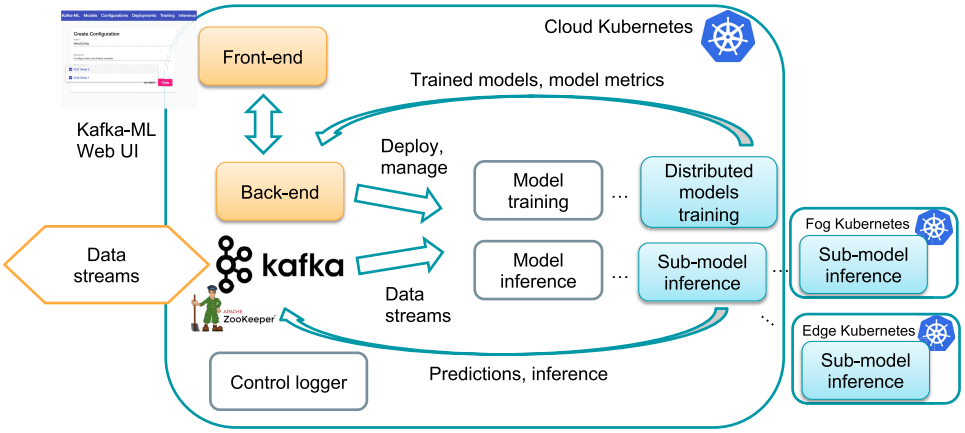

FIGURE 9. Overview of the new Kafka-ML architecture and its components deployed and managed in Kubernetes clusters. Orange component: modified; blue component: added.

Kafka-ML. This component uses the RESTful API defined in the back-end and has been implemented using the popular TypeScript framework for web development Angular. In this case, changes were included to provide users with all the new information and fields required in each of the different forms to complete the pipeline of the distributed models (e.g., Fig. 4 and 8).

C. TRAINING DISTRIBUTED MODELS

Once a specific configuration which contains partitioned ML models is deployed, a job (a deployable unit in Kubernetes) will be executed according to each Kafka-ML model for training and containerizing in a Docker container. If the configuration contains other distributed or non-distributed models that are going to be trained with the same data stream, a job per ML model will also be created. Algorithm 1 describes the procedure of the distributed training job in Kafka-ML, and Figure 10 depicts the sequence diagram of the training process. In the sequence diagram, the training job is executed for a model which is composed of three sub-models. Note that some steps, such as management of exceptions and data stream decoding, have not been included for simplicity. At first, each job downloads every ML model contained in the distributed chain from the back-end (steps 1-7) and then builds the whole distributed ML model (8) to be trained. Next, the job starts receiving control stream messages until it receives the one it expects (12), i.e., it matches the deployment_id received. The control message also indicates where the data streams are allocated (13) and how the data stream are designated for training and evaluation. The training and, optionally, the evaluation will be performed using the data stream information received in the control stream message. At the end and after training, the job sends each trained model in the distributed chain and their training and evaluation metrics to the back-end (17-22). This process, that can require large computation capabilities, can be performed in an instance of Kafka-ML in a powerful infrastructure like the Cloud.

Algorithm 1: Distributed Training Algorithm in Kafka-ML

```
Input: models_urls, training_kwargs,
       evaluation_kwargs, deployment_id, stream data
Result: Distributed trained ML models and training and
        evaluation metrics
models[] ←
 downloadModelFromBackend(models_urls);
trained ← False;
while not trained do
    msg ← readControlStreams();
    if deployment_id == msg.deployment_id then
        training_stream ← readStream(msg.topic);
        if msg.validatition_rate > 0 then
            training_stream ← take(data_stream,
              msg.validation_rate);
            evaluation_stream ← split(data_stream,
              msg.validation_rate)
        end
        model ← buildCompleteModel(models_urls);
        training_res ← trainModel(model,
          training_kwargs, training_stream);
        if msg.validatition_rate > 0 then
            evaluation_res ← evaluateModel(model,
              evaluation_kwargs, evaluation_stream);
        end
        uploadTrainedModelAndMetrics(models_urls,
          model, training_res, evaluation_res);
        trained ← True;
    end
end
```

D. INFERENCE OF DISTRIBUTED MODELS

Once a distributed ML model is trained and deployed for inference through the Kafka-ML web UI, a replication controller (a Kubernetes component that ensures that a specified

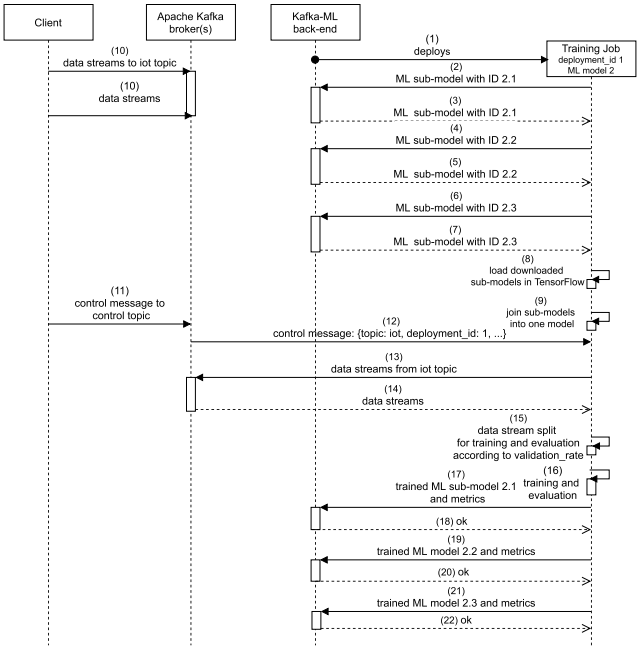

FIGURE 10. Sequence diagram of the training process in Kafka-ML.

number of replicas is running at all times) will be executed with the replicas established along with its corresponding Docker containers. Replicas ensure load balancing and high availability of this component. Algorithm 2 describes the procedure of the distributed inference in Kafka-ML and Figure 11 the sequence diagram. In the sequence diagram, a model composed by three sub-models are deployed in three layers of the continuum: Edge, Fog and Cloud. In this case, a replication controller will be executed per ML sub-model in the distributed chain, and the Kubernetes cluster where this model will be deployed (e.g., Edge, Fog, Cloud) can be configured in Kafka-ML during the deployment. When this component finishes to download the requested trained sub-model (4-5, 7-8, 10-11), the models are loaded (6, 9, 12), and the component starts receiving data streams to then make predictions on them (15). Next, predictions are sent through the Kafka output topic configured if the accuracy obtained is higher than the threshold configured during deployment; otherwise, the partial results are sent to the next layer in the distributed chain to continue the distributed flow (17 and 21). In this example, data streams are firstly received by the edge layer, which does not provide enough accuracy. Then, data streams are sent to the Fog layer to finally receive a response at the Cloud by interested applications. Replication controller is useful when having multiple Kafka brokers and partitions because it exploits the consumer group feature of Apache Kafka by matching replicas and partitions to provide load balancing and higher data ingestion.

E. DATA STREAMING MANAGEMENT IN KAFKA-ML

Data stream management in Kafka-ML is provided regarding its two main tasks: training and inference. A Kafka

Algorithm 2: Distributed Inference Algorithm in Kafka-ML

```
Input: model_url, input_topic, output_topic,
       upper_topic, input_configuration, stream data,
       limit
Result: Predictions to Apache Kafka
model ←
 downloadTrainedModelFromBackend(model_url);
deserializer ← getDeserializer(input_configuration);
while True do
    stream ← readStreams(input_topic);
    data ← decode(deserializer, stream);
    predictions ← predict(model, data);
    if max(predictions) < limit then
        sendToKafka(predictions, upper_topic);
    else
        sendToKafka(predictions, output_topic);
    end
end
```

topic is the elementary way of connecting data sources and Kafka-ML tasks in Kafka-ML, allocating the data stream sent by Kafka-ML end users. Topic are independent of each other and are uniquely identified with a defined name. Fault tolerance and load balancing to ensure the availability of the information sent are provided 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 to provide load balancing, while topic replicas enable fault tolerance through replication. A topic can be automatically created when sending a data stream in Apache Kafka, or defined using an open-source tool such as Kafka manager.[4] Finally, topics are managed by a cluster of Kafka brokers, which comprise Kafka's architecture and are responsible for receiving a data stream from data sources and distributing it to subscribing consumers (e.g., training and inference tasks). Data streams sent through Kafka must be encoded using the AVRO and RAW data format supported in Kafka-ML. For this purpose, several scripts have been provided in the project repository so that users can code and send data streams and control messages in a simple way.

Data streams sent for training use the distributed log provided in Apache Kafka, where training tasks move along the log and read data streams as they are indicated by control messages. Right after sending a data stream, a control message is dispatched and it indicates where exactly (topic and position in the distributed log) the data stream sent are available. Note that, thanks to this, the same data stream sent can be reused by different training tasks only by sending some control messages, but also for a training task that fails and needs to recover the whole data stream. In traditional message queue systems where each message can be deleted

[4]https://github.com/yahoo/CMAK

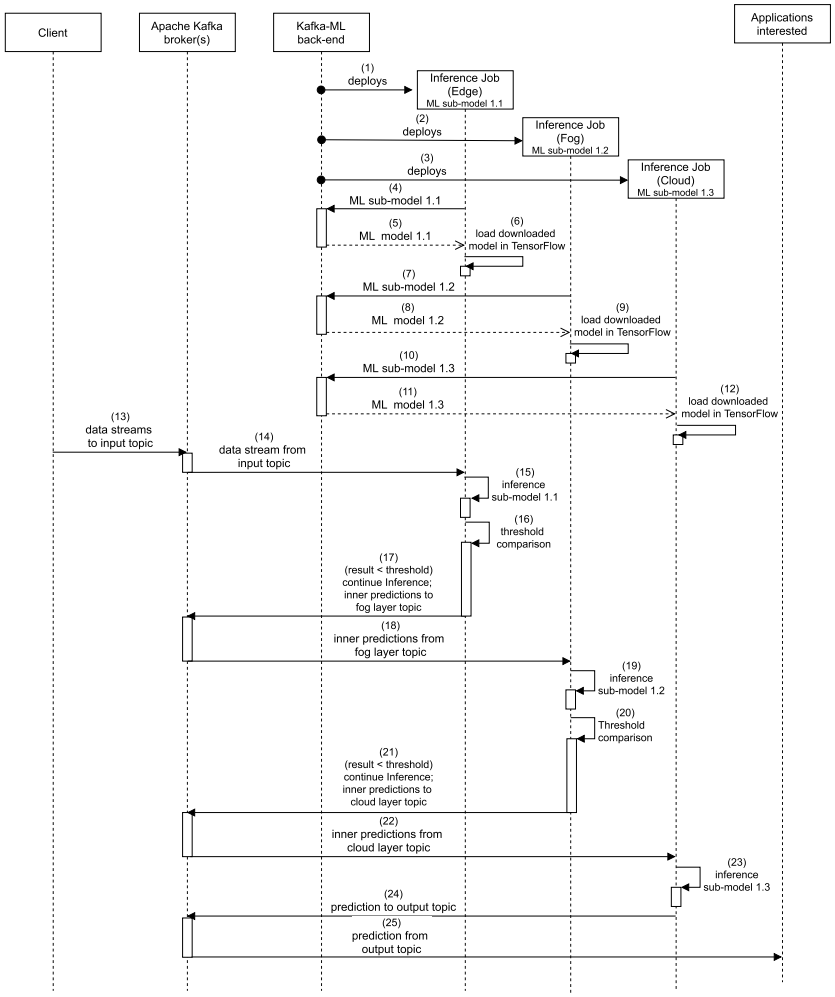

FIGURE 11. Sequence diagram of the inference process in Kafka-ML.

after consumption, a datastore may be needed to ensure there is no data loss in these situations. An example of the data stream management for training is illustrated in Fig. 12.

Firstly, the green data stream was sent along with the 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 state, the green data stream is expiring and cannot be longer reused for another training task. Then, the blue data stream was sent for configurations D3 and D5, whereas the orange data stream was for configuration D4. Orange and blue data streams can still be reused for new configurations that want to use this data stream. Finally, the gray data stream is now entering the distributed log, and a control message has not yet been sent since the data stream is not complete.

Regarding inference, the process requires fewer steps since it is not necessary to send control messages (data streams can be processed one by one). Data streams are sent one by one by data sources (e.g., the IoT) to an input topic according to the chosen encoding, and the inference result will be dispatched to an early exit output topic (if the accuracy obtained is higher than a defined threshold) or to the upper layer in the Cloud-to-Things Continuum to continue the inference process. The same data stream can be processed by multiple inference deployments, just by configuring the same input topic in Kafka-ML (Figure 8). Note that, in order to do this, the inference ML models must accept the same input. Finally, to support a higher load of data streams and data sources, the inference module can be deployed with replication forming an inference group. In this case, each member of the group will receive a single data stream at a time and the load is distributed across the group thanks to the consumer group feature of Apache Kafka.

VII. EVALUATION

As discussed previously, the Kafka-ML extension presented in this work allows users to manage and deploy Distributed DNNs. For the evaluation of this new feature on Kafka-ML, we have defined different scenarios. These scenarios consider three possible cases according to the Cloud-to-Things Continuum defined in Figure 1. The first scenario consists of the exclusive use of the Cloud to make predictions, sending

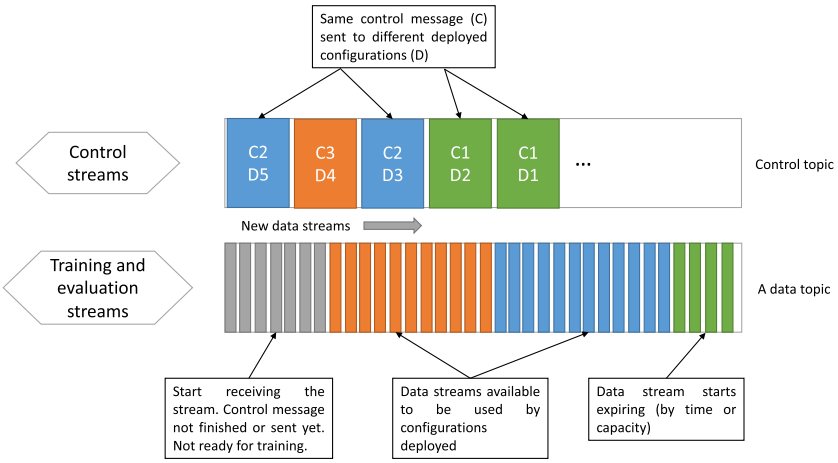

FIGURE 12. Data stream management in Kafka-ML [15].

all the computational load to this layer. The other two scenarios consider an Edge layer and Edge-Fog layers respectively. We have defined the primary ML model based on BranchyNet [26] and AlexNet [36]. AlexNet is a well-known deep neural network that achieved state-of-the-art results for the ImageNet LSVRC-2010 competition. We have employed the CIFAR10 [37] dataset for both the training and evaluation processes of different model versions using Kafka-ML. CIFAR10 consists of 6000 32 × 32 color images in 10 well balanced classes. We have used 80% during the training step, and the remaining 20% percent during the test phase. The ML model versions are slightly different for each case since early exits have been considered to adapt the model to the different scenarios:

1) **Scenario 1**. This scenario considers IoT devices sending requests to the Cloud directly. In other terms, the Cloud fully processes all the requests (i.e., the model inferences), and there are no Edge and Fog devices. Hence, we have not included any early exit, and we have not partitioned the model into different sub-models in this case. After introducing the ML model in Kafka-ML, the framework will train and deploy it into the Kubernetes cluster configured in the Cloud. Figure 13 shows the ML model deployed for the prediction process in the Cloud.
2) **Scenario 2**. Figure 14 depicts the model shown in Figure 13 divided in two sub-models, one for the Edge layer and a second one for the Cloud layer. After configuring these sub-models in Kafka-ML, users can perform their training in Kafka-ML and then deploy them into the different clusters of the Cloud-to-Things layers. Therefore, the IoT devices will send requests to the Edge cluster to start the predictions for this scenario. The Edge cluster will handle part of the prediction process and provide a result by the added early exit. If the prediction performed by the Edge early exit is not precise enough (i.e., the returned probability of the prediction is smaller than a specified threshold), the Edge will send a request to the Cloud to continue the inference process. To decide whether or not we send information to superior levels of the Cloud-to-Things Continuum, we have set a threshold value of 0.8. We have considered this value since most of the early exit responses higher than this value match the class returned in the Cloud for CIFAR10. We can increase the threshold value to provide higher precision. Otherwise, decreasing this value will result in faster responses because the Edge layer will send fewer requests to the Cloud layer. Therefore, this threshold is a trade-off value between precision and response time.
3) **Scenario 3**. Finally, we have included an additional early exit in the last scenario and divided the

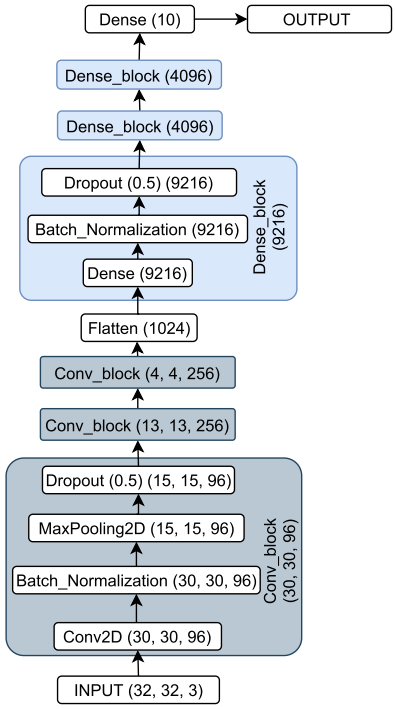

FIGURE 13. Scenario 1: AlexNet Branchynet-based DDNN deployed at the Cloud.

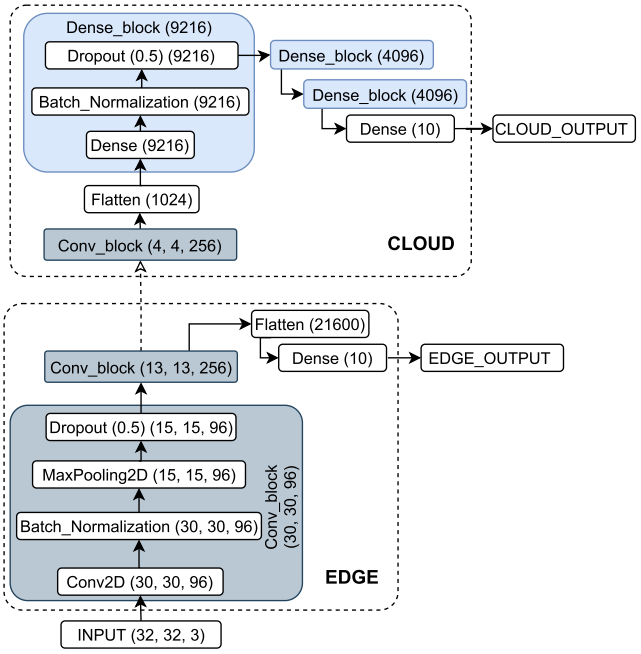

FIGURE 14. Scenario 2: AlexNet Branchynet-based DDNN deployed at the Edge, and the Cloud.

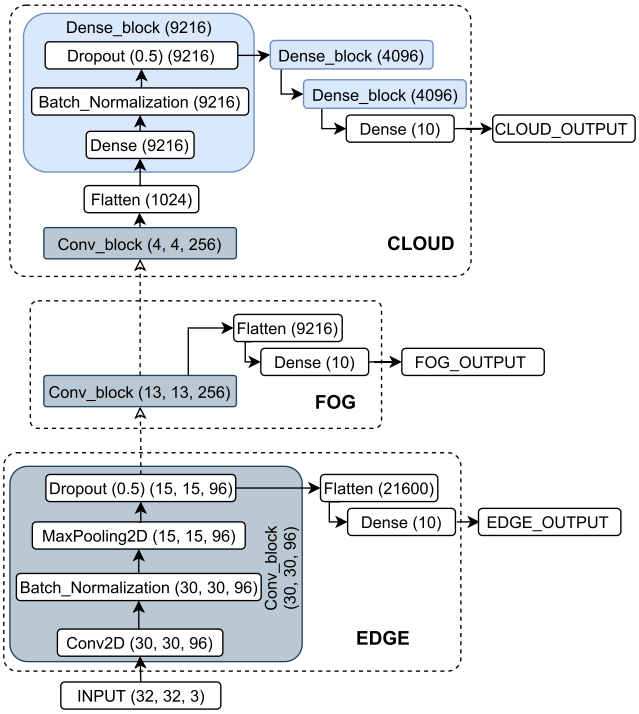

FIGURE 15. Scenario 3: AlexNet Branchynet-based DDNN deployed at the Edge, the Fog, and the Cloud.

Edge sub-model into two sub-models. One of these sub-models includes the layers of the neural network for the Fog layer. The remaining neural network layers comprise the final Edge sub-model. Consequently, this scenario has an additional layer in the continuum in addition to the Edge and Cloud considered in Figure 14. For this case, Kafka-ML will deploy the models over the Edge, Fog, and Cloud layers connecting them when required. As in the previous case, the IoT device will send requests to the Edge device, and the threshold value is 0.8. However, in this case, the Edge layer will communicate with the Fog layer instead of the Cloud layer continuing the prediction process. Then, the Fog layer will ask the Cloud layer to end the prediction if the probability given by the Fog early exit is lower than the threshold in the Fog layer (same than in the in the Edge layer). Figure 15 represents the sub-models for this scenario and their respective early exits.

In these three scenarios, we identify four architecture layers. All cases have the same resources on each layer. Nevertheless, we do not use some of these layers in all scenarios. The four layers have the following configurations:

- **IoT or Devices**: We have configured a single PC as a client to send information and measure the final results. This computer has Windows 10 with 16 GB of RAM and an i5-7400 3.00 GHz processor. The prediction requests of an increasing number of clients have been simulated using this PC.
- **Edge**: The Edge layer is built on a single computer with 62 GB of RAM and an i9-10900K CPU at 3.70 GHz. This computer has Kubernetes v1.21.2 and Docker 20.10.7 running on Ubuntu 21.04.
- **Fog**: A five-node Kubernetes cluster at our private VMWare vCloud infrastructure comprises the Fog layer. The cluster consists of a master node and four workers. Each node of this cluster has a total of 4 vCPUs and 16 GB RAM available. Each of them runs Ubuntu 16.04.7 LTS with Kubernetes v1.19.3 and Docker 19.03.13.
- **Cloud**: The configuration for the Cloud layer has been set in Google Cloud Platform. We have configured a Kubernetes cluster with 6 Nodes, 12 vCPUs and 24 GB memory to evaluate the new feature of our framework.

Considering these scenarios and the configurations mentioned above, various tests were carried out. The tests aim to evaluate the performance of Kafka-ML by increasing the data ingestion (controlling the number of clients requesting predictions) and increasing replications and partitions of Kafka topics. First, a simple scenario with a single Kafka broker was deployed, and later three Kafka brokers were used to evaluate high availability features. The latest experiments are to assess the fault tolerance and high availability characteristics of the new Kafka-ML feature. For the sake of clarity, we have listed the evaluations performed for each scenario in the following points:

- Data ingestion performance of the Kafka-ML inference instances with one Kafka broker.
- Deployment in a higher availability scenario with three Kafka brokers.
- Evaluation of the fault tolerance and high availability characteristics of Distributed DNNs in Kafka-ML.

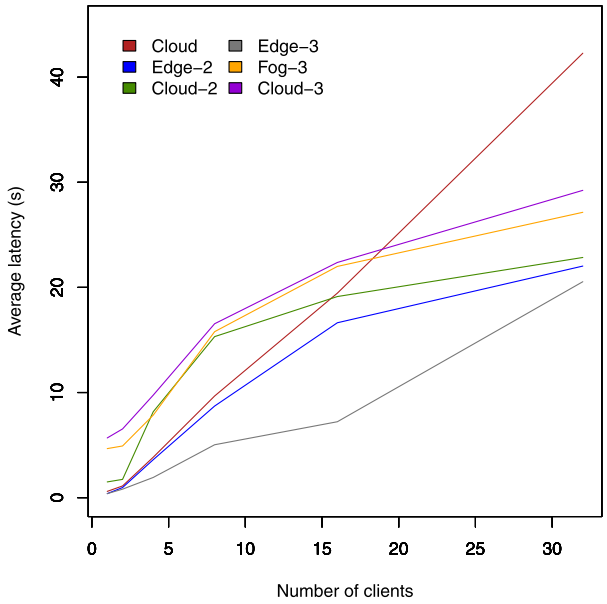

FIGURE 16. Average inference latency response with different numbers of clients (1 replica and 1 Kafka broker).

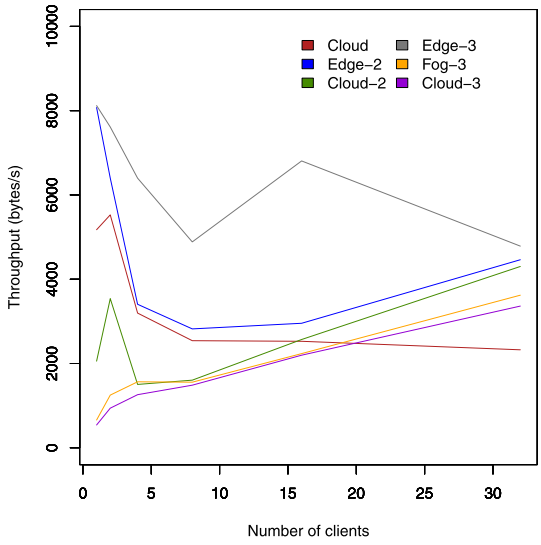

FIGURE 17. Average throughput with different numbers of clients (1 replica and 1 Kafka broker).

All analyses were conducted using a single random image per client, all of them requesting a prediction at the same time. To provide statistical confidence in the results shown throughout this section, we obtained the average results after repeating this process 25 independent times for each case.

A. TEST 1. DATA INGESTION WITH ONE KAFKA BROKER

The first evaluation performed for each scenario considers a single Kafka broker on each layer of the Cloud-to-Things Continuum. Mainly, we want to analyze how the increasing number of clients affects the inference performance. Therefore, the average latency and network communication throughput on each architectural layer of each scenario were measured. The average latency gives us the mean response time since a sender device (e.g., IoT device) requests a prediction. The network communication throughput represents the number of bytes sent and processed per second on each layer of the Cloud-to-Things Continuum.

Using a single replica of the model inference service, we can evaluate how this feature behaves with the most basic deployment. Figure 16 shows the mean values for the inference latency response for an increasing number of clients using one Kafka broker and a single replica. Scenario 1 corresponds to the Cloud output, Scenario 2 to the Edge-2 and Cloud-2 outputs, and finally Scenario 3, with three layers, corresponds to Edge-3, Fog-3, and Cloud-3 respectively. This will also apply to the following tests. For all the scenarios, as the number of clients increases, the response time also augments. For Scenario 1 (Cloud), representing the non-partitioned model in the Cloud, we can observe that the times obtained on average with few clients are better than the response time in Fog (Fog-3) and Cloud (Cloud-2 and Cloud-3) layers for the other scenarios. Nevertheless, the response time offered by the Edge layers in the remaining scenarios (Edge-2 and Edge-3) is significantly lower than in Scenario 1. These results mean that many results will be evaluated and returned by the Edge layer instead of always sending requests to the Cloud layer, leading to a faster response. The results from layers above the Edge in Scenario 2 and Scenario 3 show that having intermediate layers, the response time of the inferences that have to reach these layers increases compared to the Edge layer. However, these scenarios significantly make up for the slow responses of Scenario 1 when we have a higher number of requests and clients. As shown in Figure 16, as we increase the number of clients, latency starts to increase significantly in Scenario 1, with either of the other two scenarios being much more efficient. Comparing the results obtained for Scenario 2 and Scenario 3, we can see that the reduction of computational load (fewer layers in Scenario 3 Edge sub-model) in the Edge helps to reduce the response time and load in this layer. Similar to the increase of latency in Scenario 1 with the increment of clients, latency response also increases in Scenario 2 as the requests increase. Consequently, a third layer in the continuum (Scenario 3) is more beneficial when there is a massive increase in the total number of requests. In this case, the Edge layer (Edge-3) provides lower response times, although the upper layers (Fog-3 and Cloud-3) provide higher response times than Cloud-2. In this case, there is a trade-off between faster response times offered by the Edge in Scenario 3 (which are responded in most cases) and increased latency in the upper layers.

To support these results, we have looked at the average throughput using different numbers of clients. Figure 17 shows that Scenario 1, with a higher number of clients, processes fewer bytes per second. In contrast, Scenario i2 and Scenario 3 can process more information per second.

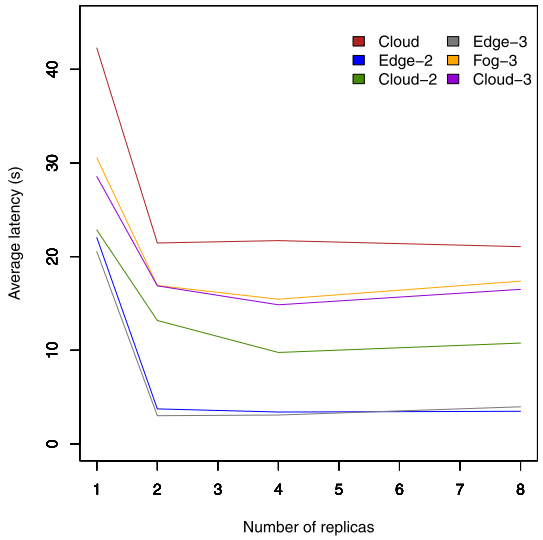

FIGURE 18. Average inference latency response with different numbers of replicas and 1 Kafka broker.

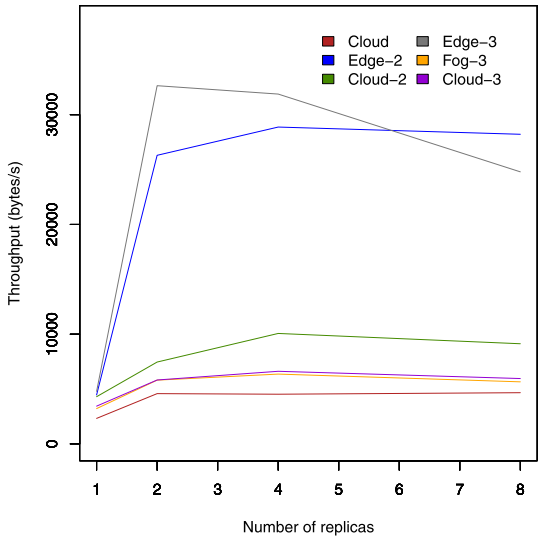

FIGURE 19. Average throughput with different numbers of replicas and 1 Kafka broker.

Figure 17 also shows that the throughput of the first layers is the highest. In other words, the Edge layers must be able to process as much information as possible. In this way, the upper layers of the continuum are less overloaded and can perform much more expensive operations avoiding cases that do not need such dedication.

Figure 18 shows the behavior with 32 clients and increasing the number of replicas of the inference module in all the layers. We can see that the number of replicas benefits the response time significantly. However, when using more than two replicas, we see how it tends to stabilize or even offers worse results. This behavior is due to the fact that having only one broker and one partition increasing the number of replicas does not take advantage of these capabilities in a satisfactory way and there is an overload in the system due to the replication. We can corroborate this statement if we look at Figure 19. This Figure represents the throughput for a different number of replicas and 32 clients. It can be seen that the throughput when using more than two replicas is not significantly affected, as happens with latency.

B. TEST 2. HIGHER AVAILABILITY ENVIRONMENT WITH THREE KAFKA BROKERS

Using a single Kafka broker to manage the responses can suppose a bottleneck in the communications among layers. In other words, a unique broker on each layer has to receive and send all of the prediction messages. Therefore, it results in an overload of work for the mediator between the layers. For this reason, three Kafka brokers were used during this test in order to assess the behavior of Kafka-ML in a higher performance scenario. By increasing the number of brokers, we can evaluate the influence of this higher performance scenario in Kafka-ML and how this reduces the bottleneck caused by a higher number of requests.

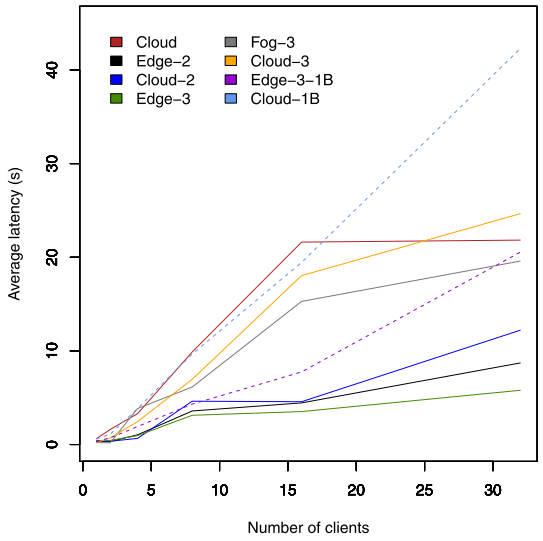

FIGURE 20. Average inference latency response with different numbers of clients (1 replica and 3 Kafka brokers).

In Figure 20, the average response time obtained when using three Kafka brokers with a different number of clients is shown. In addition, several discontinuous lines have been included to represent the Cloud-to-Things Continuum layer with the worst and best results from Test 1 with a single Kafka broker (Cloud and Edge-3 respectively). In Scenario 1, the Cloud is still worse in most cases, while other Scenarios improve significantly. Scenario 2 and Scenario 3 considerably reduce the response time in all their layers, resulting in better times than Scenario 1 and the best results obtained in Test 1 (Figure 16). We demonstrate herewith the improved performance of Kafka-ML in this deployment with three Kafka brokers.

These results are further supported by Figure 21. This Figure contemplates the throughput for a different number of

clients when using three Kafka brokers. As in the previous test, we can see that the Edge layers process much more information than the Cloud from the first scenario proposed. Moreover, with three Kafka brokers, the processed information is significantly higher, as we can verify by looking at the dashed lines in Figure 21 from the previous test. By giving a much faster response at the Edge level, the Fog and Cloud are not overloaded and can focus on requests that require their computing capacity.

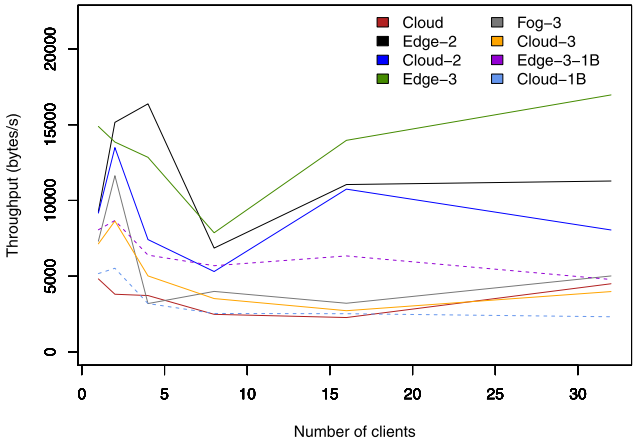

FIGURE 21. Average inference throughput with different numbers of clients (1 replica and 3 Kafka brokers).

As in the previous test, the increase in the number of replicas does not seem to improve significantly. However, as shown in Figure 22, the response times with 32 clients and a different number of replicas reach a minimum in each layer. This behavior ensures that the increase in replicas benefits the response time. Therefore, a more significant decrease could be seen with a greater number of requests and clients. Figure 22 shows that the number of replicas does not imply an overload for the system and improves the results obtained in Figure 20 with a single replica. The deployment of Kafka-ML with a larger number of brokers contributes to favoring replication, since replicas can share brokers and thus reduce latency.

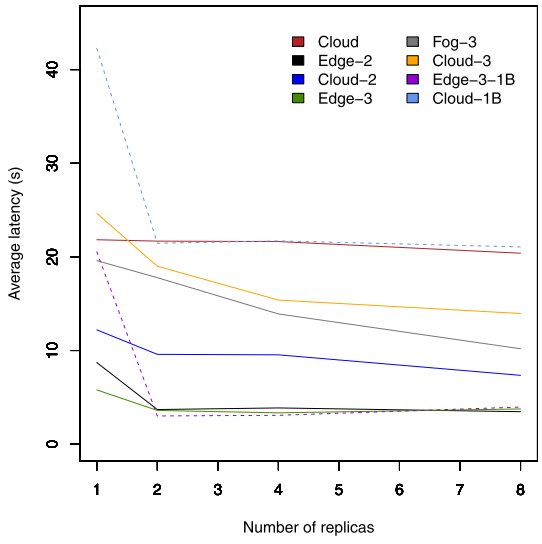

FIGURE 22. Average inference latency response with different numbers of replicas and 3 Kafka brokers.

In Figure 23, we can see that the throughput improves considerably in most layers when using two replicas. Moreover, the increment of replicas above represents a slight improvement, unlike the previous test, where replication decreased performance.

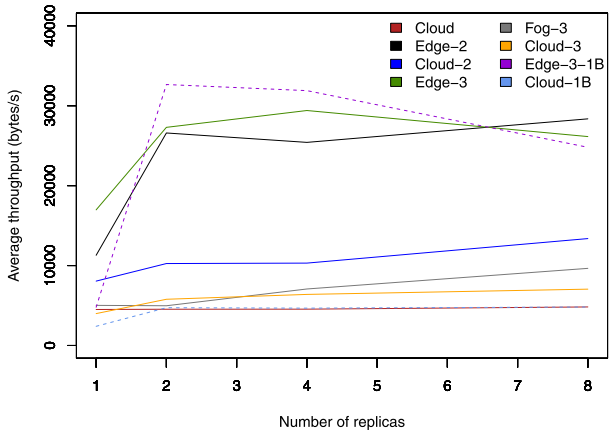

FIGURE 23. Average inference throughput with different numbers of replicas and 3 Kafka brokers.

The results of this second test demonstrate that the new distributed model feature of Kafka-ML results in significant time improvements. In addition, it is prepared to accept a large number of requests and give a response in the shortest possible time. Using three Kafka brokers, we have verified that Kafka-ML performs adequately in a high-performance scenario. Additionally, using DDNNs along the Cloud-to-Things Continuum has proven to be much more efficient than relying solely on Cloud platforms.

C. TEST 3. FAULT TOLERANCE AND HIGH AVAILABILITY EVALUATION

In this last test, how the performance of Kafka-ML is affected in the presence of failure situations was evaluated. In particular, the fault tolerance and high availability characteristics were evaluated in the presence of failures in the deployed Kafka-ML inference replicas and the Kafka brokers. For the tests, the previous high performance deployment of Kafka-ML with three brokers was adopted, considering eight replicas, Scenario 3 to cover the whole Cloud-to-Things Continuum (Edge-Fog-Cloud), and the largest data ingestion (32 clients). During the execution of the tests, the inference components and the Kafka brokers were manually stopped to simulate failures in these components.

Figures 24, 25, and 26 show the inference response time when different amounts of replicas stop working at the Edge, Fog, and Cloud respectively. The results show that, as the number of dropped replicas increases, the response time of

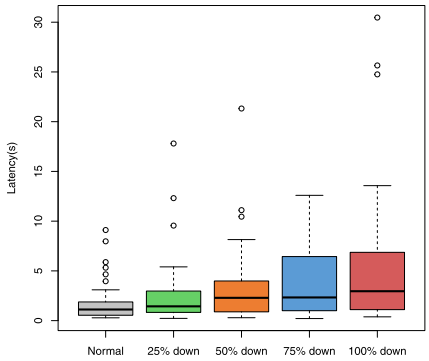

FIGURE 24. Inference latency response at the Edge with 32 clients and different numbers of replicas down.

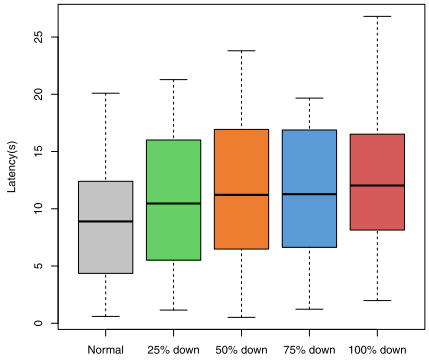

FIGURE 25. Inference latency response at the Fog with 32 clients and different numbers of replicas down.

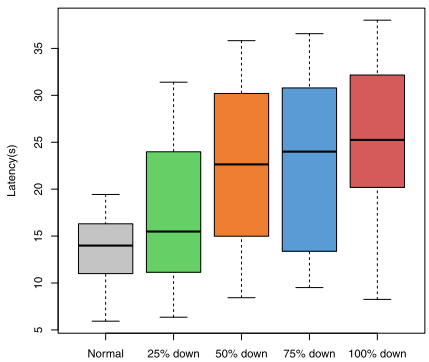

FIGURE 26. Inference latency response at the Cloud with 32 clients and different numbers of replicas down.

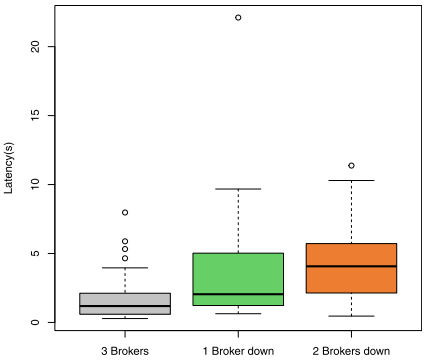

FIGURE 27. Inference latency response at the Edge with 32 clients and different Apache Kafka brokers down.

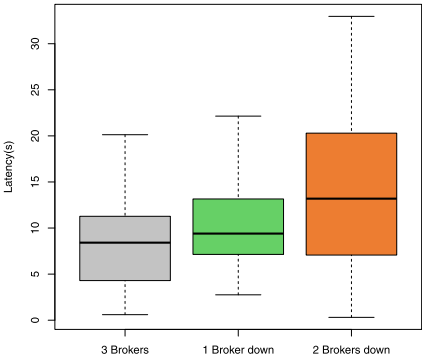

FIGURE 28. Inference latency response at the fog with 32 clients and different Apache Kafka brokers down.

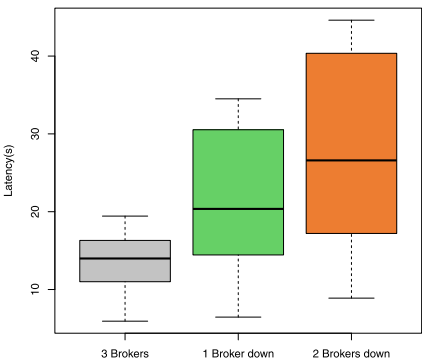

FIGURE 29. Inference latency response at the Cloud with 32 clients and different Apache Kafka brokers down.

Kafka-ML inference also increases in the three layers studied. This has a greater impact on the highest layers in the continuum, such as the Fog and Cloud, where latency is also higher. However, the results show how Kafka-ML can handle the failure of up to 100% of the deployed inference modules. This is achieved thanks to the continuous monitoring of Kafka-ML components offered by Kubernetes, which detects when an inference component or server fails, and brings the inference component back upon an available node in a very short time.

Figures 27, 28, and 29 show, on the other hand, the response time of the Edge, Fog, and Cloud respectively when one and two Kafka nodes fail. As happens with the inference module, the failures of the Kafka brokers are also handled in Kafka-ML. Likewise, the response time is increased in all cases, especially in the upper layers. However, the failures are also detected by the continuous monitoring performed and returned to normal operation in a short time.

The results of this test demonstrate how Kafka-ML offers fault tolerance and high availability features during its execution. In particular, we have verified how Kafka-ML successfully handles the failures of the inference module and the Kafka brokers, in charge of inter-layer communication,

with a slight increase in latency. Therefore, this demonstrates the viability of Kafka-ML for applications that require low latency with distributed models along the continuum and high availability and fault tolerance.

VIII. CONCLUSION

The allocation and management of distributed and deep neural networks in the Cloud-to-Things Continuum is of great interest for applications with time-sensitive requirements. Moreover, their harmony with data stream technologies would facilitate their integration with current and disrupting data stream sources like the IoT. For this reason, we have extended our open-source framework Kafka-ML, which harmonizes data streams with ML/AI frameworks, to support the deployment and management of distributed DNN pipelines over the Cloud-to-Things Continuum. This includes the pipeline of ML/AI applications: the design of distributed ML models; their training with data streams; and their deployment in the continuum with state-of-the-art containerization technologies–providing fault tolerance and high-availability–and data stream ingestion. Moreover, thanks to Kafka-ML, model sharing, metrics evaluation, downloading of models, and data stream management are also possible in an open-source solution. Early exits based on BranchyNet have been adopted to provide as fast a response as possible when the accuracy is acceptable. As a result, the validation of this framework in an Edge-Cloud and Edge-Fog-Cloud deployment demonstrates the improvement of our proposed approach regarding only a Cloud deployment.

Dynamic DDNN partitioning will be explored as future work to adapt DDNN to the current status of the deployment scenarios (networking + hardware) by allocating DDNN layers at the right time at the right place in the Cloud-to-Things Continuum. This work was also among the future work envisaged in Kafka-ML. Therefore, we will continue improving the Kafka-ML framework by supporting distributed training and accelerating the training phase; furtherance for more ML/AL frameworks beyond TensorFlow; and on-device inference management and deployment for critical decisions.

REFERENCES

[1] S. Sagiroglu and D. Sinanc, "Big data: A review," in *Proc. Int. Conf. Collaboration Technol. Syst. (CTS)*, San Diego, CA, USA, May 2013, pp. 42–47.

[2] Y. Lu, "Artificial intelligence: A survey on evolution, models, applications and future trends," *J. Manage. Anal.*, vol. 6, no. 1, pp. 1–29, Jan. 2019.

[3] G. Stringhini, C. Kruegel, and G. Vigna, "Detecting spammers on social networks," in *Proc. 26th Annu. Comput. Secur. Appl. Conf.*, New York, NY, USA, 2010, pp. 1–9, doi: 10.1145/1920261.1920263.

[4] E. Mariconti, G. Suarez-Tangil, J. Blackburn, E. De Cristofaro, N. Kourtellis, I. Leontiadis, J. L. Serrano, and G. Stringhini, "'You know what to do' proactive detection of YouTube videos targeted by coordinated hate attacks," *Proc. ACM Hum.-Comput. Interact.*, vol. 3, Nov. 2019, pp. 1–15, doi: 10.1145/3359309.

[5] A. Abdelaziz, M. Elhoseny, A. S. Salama, and A. M. Riad, "A machine learning model for improving healthcare services on cloud computing environment," *Meas. J. Int. Meas.*, vol. 119, pp. 117–128, Apr. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0263224118300228

[6] M. Chen, Y. Hao, K. Hwang, L. Wang, and L. Wang, "Disease prediction by machine learning over big data from healthcare communities," *IEEE Access*, vol. 5, pp. 8869–8879, 2017.

[7] L. Piyathilaka, D. Preethichandra, U. Izhar, and G. Kahandawa, "Real-time concrete crack detection and instance segmentation using deep transfer learning," *Eng. Proc.*, vol. 2, no. 1, p. 91, 2020. [Online]. Available: https://www.mdpi.com/2673-4591/2/1/91

[8] Y. Bao, Z. Tang, H. Li, and Y. Zhang, "Computer vision and deep learning–based data anomaly detection method for structural health monitoring," *Struct. Health Monit.*, vol. 18, no. 2, pp. 401–421, 2019.

[9] L. Hui and T. Chin-Chung, "A review of using machine learning approaches for precision education," *J. Educ. Technol. Soc.*, vol. 24, no. 1, pp. 250–266, 2021.

[10] 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," *J. Netw. Comput. Appl.*, vol. 67, pp. 99–117, May 2016.

[11] *Internet of Things at a Glance*. Accessed: Feb. 20, 2021. [Online]. Available: https://emarsonindia.com/wp-content/uploads/2020/02/Internet-of-Things.pdf

[12] G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, A. L. García, I. Heredia, P. Malík, and L. Hluchý, "Machine learning and deep learning frameworks and libraries for large-scale data mining: A survey," *Artif. Intell. Rev.*, vol. 52, no. 1, pp. 77–124, 2019

[13] M. Abadi. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. [Online]. Available: https://www.tensorflow.org/

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, and Z. Lin, "Pytorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[15] C. Martín, P. Langendoerfer, P. S. Zarrin, M. Díaz, and B. Rubio, "Kafka-ML: Connecting the data stream with ML/AI frameworks," *Future Gener. Comput. Syst.*, vol. 126, pp. 15–33, Jan. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X21002995

[16] *Apache Kafka*. Accessed: May 13, 2020. [Online]. Available: http://kafka.apache.org/

[17] C. M. Fernandez, M. D. Rodriguez, and B. R. Munoz, "An edge computing architecture in the Internet of Things," in *Proc. IEEE 21st Int. Symp. Real-Time Distrib. Comput. (ISORC)*, Singapore, May 2018, pp. 99–102.

[18] D. R. Torres, C. Martín, B. Rubio, and M. Díaz, "An open source framework based on Kafka-ML for DDNN inference over the Cloud-to-Things continuum," *J. Syst. Archit.*, vol. 118, Sep. 2021, Art. no. 102214.

[19] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Atlanta, GA, USA, Jun. 2017, pp. 328–339.

[20] *Kubernetes*. Accessed: Jan. 9, 2021. [Online]. Available: https://www.kubeflow.org/

[21] S. Tuli, N. Basumatary, and R. Buyya, "EdgeLens: Deep learning based object detection in integrated IoT, fog and cloud computing environments," in *Proc. 4th Int. Conf. Inf. Syst. Comput. Netw. (ISCON)*, Mathura, India, Nov. 2019, pp. 496–502.

[22] S. Tuli, N. Basumatary, S. S. Gill, M. Kahani, R. C. Arya, G. S. Wander, and R. Buyya, "HealthFog: An ensemble deep learning based smart healthcare system for automatic diagnosis of heart diseases in integrated IoT and fog computing environments," *Future Gener. Comput. Syst.*, vol. 104, pp. 187–200, Mar. 2020.

[23] A. Javed, J. Robert, K. Heljanko, and K. Främling, "IoTEF: A federated edge-cloud architecture for fault-tolerant IoT applications," *J. Grid Comput.*, vol. 10, pp. 1–24, Jan. 2020.

[24] M. D. da Assunção, A. da S. Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *J. Netw. Comput. Appl.*, vol. 103, pp. 1–17, Feb. 2018.

[25] N. Kourtellis, G. D. F. Morales, and A. Bifet, "Large-scale learning from data streams with apache samoa," in *Learning From Data Streams in Evolving Environments*. Cham, Switzerland: Springer, 2019, pp. 177–207.

[26] S. Teerapittayanon, B. McDanel, and H. T. Kung, "BranchyNet: Fast inference via early exiting from deep neural networks," in *Proc. 23rd Int. Conf. Pattern Recognit. (ICPR)*, Cancun, Mexico, Dec. 2016, pp. 2464–2469.

[27] L. A. Steffenel, M. Kirsch Pinheiro, and C. Souveyet, "Assessing the impact of unbalanced resources and communications in edge computing," *Pervas. Mobile Comput.*, vol. 71, Feb. 2021, Art. no. 101321. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574119220301450

[28] T. Mohammed, C. Joe-Wong, R. Babbar, and M. D. Francesco, "Distributed inference acceleration with adaptive DNN partitioning and offloading," in *Proc. IEEE INFOCOM - IEEE Conf. Comput. Commun.*, Beijing, China, Jul. 2020, pp. 854–863.
[29] R. G. Pacheco and R. S. Couto, "Inference time optimization using BranchyNet partitioning," 2020, *arXiv:2005.04099*. [Online]. Available: http://arxiv.org/abs/2005.04099
[30] J. Zhou, Y. Wang, K. Ota, and M. Dong, "AAIoT: Accelerating artificial intelligence in IoT systems," *IEEE Wireless Commun. Lett.*, vol. 8, no. 3, pp. 825–828, Jun. 2019.
[31] E. D. Coninck, S. Bohez, S. Leroux, T. Verbelen, B. Vankeirsbilck, P. Simoens, and B. Dhoedt, "DIANNE: A modular framework for designing, training and deploying deep neural networks on heterogeneous distributed infrastructure," *J. Syst. Softw.*, vol. 141, pp. 52–65, Jul. 2018.
[32] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive DNN surgery for inference acceleration on the edge," in *Proc. IEEE Conf. Comput. Commun.*, Paris, France, Apr. 2019, pp. 1423–1431.
[33] R. G. Pacheco and R. S. Couto, "Inference time optimization using BranchyNet partitioning," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Rennes, France, Jul. 2020, pp. 1–6.
[34] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge AI: On-demand accelerating deep neural network inference via edge computing," *IEEE Trans. Wireless Commun.*, vol. 19, no. 1, pp. 447–457, Jan. 2020.
[35] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *J. Syst. Softw.*, vol. 146, pp. 215–232, Dec. 2018.
[36] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.
[37] A. Krizhevsky, "Learning multiple layers of features from tiny images," M.S. thesis, Univ. Toronto, Toronto, ON, Canada, 2009.

ALEJANDRO CARNERO received the bachelor's degree in software engineering from the University of Málaga, Spain, in 2020, where he is currently pursuing the M.S. degree in software engineering and artificial intelligence. Since 2020, he has been a Research Assistant with the ERTIS Research Group, University of Málaga. His research interest includes distributed deep machine learning along with the IoT field.

CRISTIAN MARTÍN received the M.S. degree in computer engineering, the M.S. degree in software engineering and artificial intelligence, and the Ph.D. degree in computer science from the University of Málaga, Spain, in 2014, 2015, and 2018, respectively. He worked as a Software Engineer in various tech companies with RFID technology and software development. He is currently a Postdoctoral Researcher with the University of Málaga. He is also a member of the ITIS Software Institute, University of Málaga. His research interests include integration of the Internet of Things with cloud/fog/edge computing, machine learning, structural health monitoring, and the IoT reliability.

DANIEL R. TORRES received the Diploma and bachelor's degrees in computer science engineering, in 2017 and 2018, respectively. He is currently pursuing the master's degree in computer science with the ERTIS Research Laboratory, University of Málaga. He worked as a Software Engineer in various tech companies, where his principal functions were developing and implementing software apps, such as scripting, web, and iOS, and data analysis. He is currently working with the ERTIS Research Laboratory, University of Málaga. His main research interests include artificial intelligence applications, physics, and neuroscience. He received the Certificate in a Higher Educational and Vocational Training in Computer and Network System Administration.

DANIEL GARRIDO received the M.S. and Ph.D. degrees in computer science from the University of Málaga, Spain, in 1999 and 2006, respectively. Since 2006, he has been working in different public and private projects related to distributed programming, simulation, and real-time systems, especially in the areas of software engineering. From 2007 to 2009, he was an Assistant Professor with the Department of Languages and Computer Science, University of Málaga, where he has been an Associate Professor, since 2009. He was one of the Founders of the spin-off company Softcrits, Software for Critical Systems, Málaga. He has published several articles in international refereed journals and outstanding congresses in the field.

MANUEL DÍAZ is currently a Full Professor with the Computer Science Department, University of Málaga, where he is also the Head of the ERTIS Research Group and a member of the ITIS software Institute. In the last years, his main area of work has been focused on WSN and monitoring systems, especially on 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 a Main Researcher of UMA in the WSAN4CIP (ICT FP7). He is also the Co-Founder of the spin-off Softcrits, where he is the Head of the Research and Development Department. His research interests include distributed and real-time systems, Internet of Things and P2P, especially in the context of middleware platforms and critical systems.

BARTOLOMÉ RUBIO received the M.S. and Ph.D. degrees in computer engineering from the University of Málaga, in 1990 and 1998, respectively. From 1991 to 2000, he was an Assistant Professor with the Department of Languages and Computer Science, University of Málaga, where he has been an Associate Professor, since 2001. He has been working in the areas of distributed and parallel programming and coordination models and languages. He is specially involved in the research fields of wireless sensor and actor networks and the integration of the Internet of Things and cloud computing. He has been a member of the Software Engineering Group, University of Málaga, since its foundation. He is a member of the ITIS Software Institute, University of Málaga.