Part II

Artificial Intelligence of Things (AIoT) and AI at the Edge

4

Machine Learning (ML) as a Service (MLaaS): Enhancing IoT with Intelligence, Adaptive Online Deep and Reinforcement Learning, Model Sharing, and Zero-knowledge Model Verification

Jorge Mira[1], Iván Moreno[1], Hervé Bardisbanian[2], and Jesús Gorroñogoitia[1]

[1]Atos Spain, Spain
[2]Capgemini, France
E-mail: jorge.mira@atos.net; ivan.moreno@atos.net;
herve.bardisbanian@capgemini.com; jesus.gorronogoitia@atos.net

Abstract

AI has changed our lives in many aspects, including the way we (as humans) interact with internet and computational devices, but also on way devices interact with us, and among them, in most of the processes of the industry and other socioeconomic domains, where machine learning (ML) based applications are getting increasing influence. Internet of Things (IoT) plays a key role in these process interactions, by providing contextual information that requires to be processed for extracting intelligence that would largely improve them. However, the delivery of ML-based applications for IoT domains faces the intrinsic complexity of ML operations, and the online interoperability with IoT devices. In this chapter, we present the IoT-NGIN ML as a service (MLaaS) platform, an MLOps platform devised for the delivery of intelligent

applications for IoT. Its services for online deep learning (DL) training and inference, ML model conversion and sharing, and zero-knowledge model verification based on blockchain technology are also presented.

Keywords: MLOps, deep learning, online learning, model translation, zero-knowledge model verification.

4.1 Introduction

Internet of Things (IoT) facilitates the extraction of information from systems, through devices and sensors connected to them. Companies owning those systems can infer knowledge about their behavior and performance, with the aim of improving their understanding of diverse aspects. As an example, metrics gathered from sensors can be immediately used to trigger an alert on the situation where a concrete metric value overpasses a predetermined threshold. However, the increasing number of devices and sensors is generating a huge volume of information that companies need to face, a challenge identified by the Big Data 5 Vs [1]. As a result, a simple system service could not be capable anymore to cope with the data intricateness. Therefore, new solutions are required to face this complexity and effectively and purposely infer valuable information from it. With the development of new AI information extraction and ML-based inference techniques and algorithms and the advent of increasing computation power, notably based on GPUs and TPUs, it is now achievable to extract value from huge volumes of data and even predict the future behavior of systems. These breakthroughs will enable systems' stakeholders to better comprehend their company activities and improve future planning, leading to increase business value.

Primary users of these ML-based techniques are data scientists and ML engineers, who require a ML platform that can provide all the necessary services to process data, train ML models, share, and deploy them. Implementing and maintaining such an ML platform is a complex, time-consuming, and costly endeavor, requiring expertise that most of the companies lack. Therefore, a leading industry trend is addressing the provisioning of this kind of ML platforms, by offering all the services required to build and execute ready-to-use ML models. In addition, these ML platforms support the development of custom-tailored ML systems for some specific use cases. Such ML platforms are commonly referred to as machine learning as a service (MLaaS).

Companies leverage MLaaS to reduce the time and cost of integrating their ML modeling and delivery procedures into their development and CI/CD environments. By using MLaaS, data scientists can procure and pre-process the data and train the model, by focusing on their core competency, that is, in the ML development, rather than on the burden of taking care of the underlying procedures and infrastructure, which are provided and managed by the MLaaS.

Several MLaaS platforms are commercially available, either provided by big Cloud service providers, such as AWS ML, Microsoft Azure ML Studio, and Google Cloud Platform (GCP) ML Studio, or by specialized companies (e.g., BigML, Domino, Arimo, etc.). On the contrary, there are few MLaaS frameworks built around open-source services that support ML development and delivery, such as Kubeflow, MLFlow, and AirFlow, although they do not constitute a complete MLaaS platform. Building such a platform is challenging because:

- Lots of different functions are required to build up a complete MLaaS.
- For the same function, there could be several open-source projects to choose from. Determining the right one could require a long and complex evaluation.
- Projects are envisioned, designed, and implemented for a particular purpose, but scarcely concerned with their requirements of integration with other external services.
- The complexity to install, configure, maintain, manage, and use integrated services could be high.
- Further customizations and adaptations may be needed on the integrated services to fit the functional and non-functional requirements of the MLaaS.

The IoT-NGIN project has envisaged a holistic view for a complete MLaaS platform, supporting ML development and delivery in the domain of IoT, addressing the functional and non-functional requirements expressed in the project, and its high-level architecture. This task has been realized by seeking open-source projects, by selecting suitable components for specific purposes, and by determining the procedures to integrate them together in order to constitute a comprehensive framework. Besides, IoT-NGIN has adopted GitOps technologies, such as IaC and ArgoCD to automate the platform building and delivery.

The IoT-NGIN has implemented and delivered a minimum viable product (MVP) of the MLaaS platform, aimed to validate the platform function itself,

and provide support for the use cases of the project Living Labs and the external projects that are adopting the IoT-NGIN technology.

The remainder of the chapter is organized as follows. Section 4.2 introduces the functional and technical specification of the IoT-NGIN MLaaS platform and its MVP implementation. The following sections describe additional ML services developed for MLaaS. In particular, Section 4.3 provides the functional and technical specification, implementation details, and validation results of the adaptive online deep learning service, while Section 4.4 does the same for model sharing, model translation, and zero-knowledge model verification services. Section 4.5 concludes the chapter.

4.2 MLaaS

4.2.1 MLaaS features

The functional view of the IoT-NGIN MLaaS platform is shown in Figure 4.1. In a high-level functional view, the platform is structured into i) the infrastructure hosting the platform and ii) the MLaaS services. This approach avoids binding MLaaS to a specific hosting environment, so permitting MLaaS to be delivered into diverse cloud infrastructures, including public, private, or even in bare-metal ones.

As the MLaaS platform aims to offer complete support for the ML development and delivery lifecycle, it includes the following functions:

- Data functions, including data acquisition, analysis, transformation, and storage;
- ML modeling, including ML model training, evaluation, and model transfer;
- ML deployment, including model sharing and translation;
- ML prediction, including model serving, batch, and real-time prediction.

Hosting infrastructure and monitoring/management tools are not part of the MLaaS platform. Nonetheless, the infrastructure must provide network access, computing resources, including CPUs/GPUs, and storage services. IoT MLaaS adopts a container-based microservice architecture compatible with Kubernetes clusters on bare-metal infrastructures.

As shown in Figure 4.1, the MLaaS platform consists of the following functional blocks:

- IoT gateway: Includes services to receive data from IoT devices, either through message queue brokers or HTTP/S REST API.

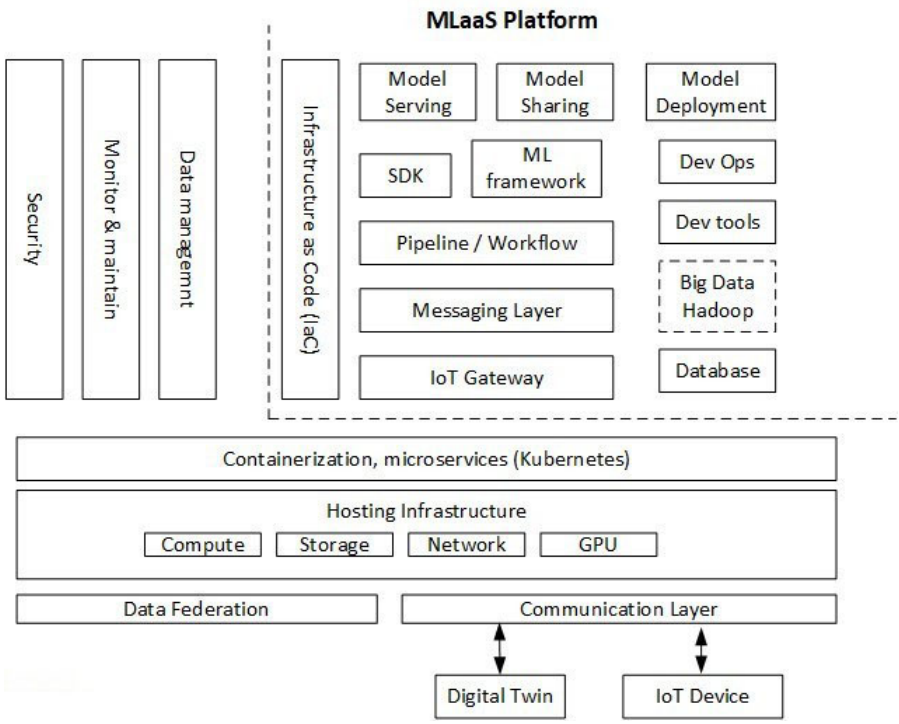

Figure 4.1 MLaaS framework functional architecture.

- Messaging layer: Includes components that interface the IoT gateway with other MLaaS upper services, by streaming the data events they consume.
- Database: Includes services providing data storage capabilities, including SQL and NoSQL databases, and time series services.
- Pipeline/workflow: Supports the building and deployment of portable, scalable ML workflows.
- ML framework: Provides ML frameworks and libraries required to build and train an ML model, including Tensorflow, Keras, PyTorch, scikit-learn, etc.
- SDK: Provides the development and testing environment to build and test ML models. A simple development environment supporting Python and Rust (as future work) is provided, but not a state-of-the-art IDE.
- Model serving: Offers services to deliver ML models through a REST API for prediction requests.

- Model sharing: Offers services to share ML models, to be used for model prediction, or for transfer learning. MLaaS also supports model translation across several popular ML frameworks. An external DLT system can be used to verify the model integrity by leveraging blockchain technology.
- Model deployment: Offers services to deliver ML models into the edge computing or into IoT devices; so they can infer predictions.
- Dev tools: Includes services aiming at assisting ML modeling, including notebook support and ML monitoring tools.
- DevOps: Includes CI/CD services to deploy new MLaaS services and ML models.
- Infrastructure as a Code (IaC): Contains the manifests required to configure the MLaaS platform.

4.2.2 MLaaS architecture, services, and delivery

The technical architecture of the reference implementation of the MLaaS platform is shown in Figure 4.2.

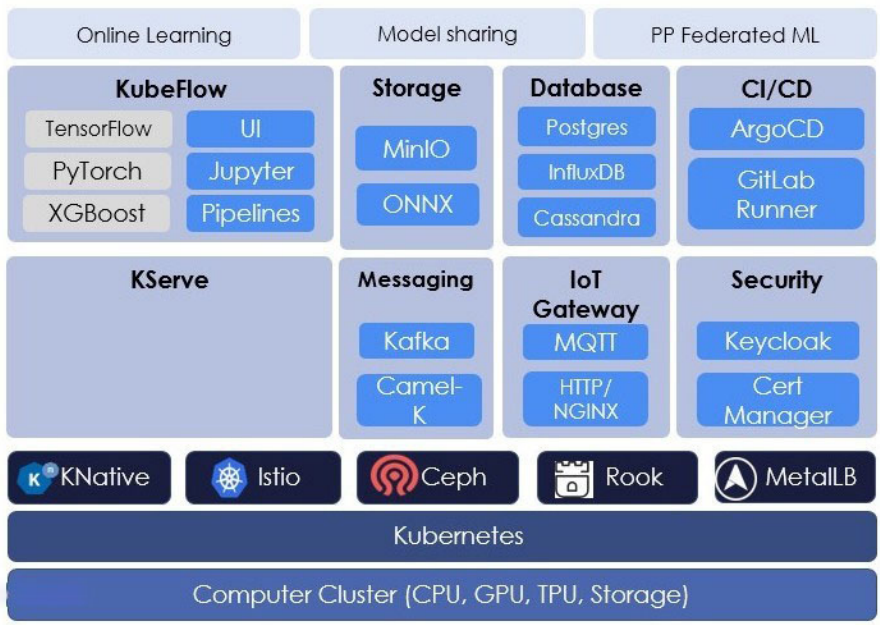

Figure 4.2 MLaaS framework reference technical architecture.

IoT-NGIN MLaaS uses Kubernetes as the main framework for the container-based instantiation of the MLaaS microservice architecture. Kubernetes is complemented with other services, including:

- Istio: A service mesh used by MLaaS components, such as Kubeflow and the Ingress gateway.
- Ceph: A unified storage service with an object block, being the default storage class.
- Rook: A cloud-native orchestrator for Kubernetes, used to manage Ceph storage.
- MetalLB: A load-balancer for metal Kubernetes clusters, used to allocate external load-balancer IP addresses to the Istio and the Nginx gateways.

MLaaS consists of several services hosted by the Kubernetes cluster. Current IoT-NGIN MLaaS platform does not include them all, although most of them, as some few services have not been required yet by the use cases; so its inclusion is left for future work. Kubeflow is the main component of the MLaaS platform. It offers ML frameworks (e.g., Tensorflow, Keras, PyTorch, MXNet, MPI, XGBoost, etc.) for model training, tools for pipelines/workflows implementation, and development tools such as Jupyter notebooks. Complementing Kubleflow, MLaaS includes KServe, a model inference service, for model serving. The IoT gateway is supported by i) Mosquitto MQTT, a message broker and ii) NGINX-based HTTP/S access to REST APIs. NGINX is a web server, also used as a reverse proxy and ingress gateway. These IoT gateway services are used to ingest data coming from IoT devices or digital twins. The messaging block, which exchange data messages between the IoT gateway and the Kubeflow/KServe services, is supported by i) Kafka, a distributed stream processing system with real-time data pipelines and integration and ii) Apache Camel-K, a lightweight integration framework for microservices. The storage block, which offers services for model sharing, is mainly supported by MinIO, a Kubernetes object storage, which can host ML models and other artifacts. The database block is supported by several SQL and non-SQL databases, including i) Postgres, an SQL object-relational database, ii) InfluxDB, a time-series platform with querying, monitoring, alerting, and visualization features, and ii) Casandra, a non-SQL distributed database. These services can be used for storing structured data and time series for ML model training. The CI/CD block is supported by i) ArgoCD, a declarative GitOps continuous delivery tool for Kubernetes and ii) GitLab Runner, a CI/CD GitLab pipeline runner. ArgoCD

is used to deploy and maintain the MLaaS platform from the IoT-NGIN Git repository. GitLab Runner is used to upload ML models into IoT devices. Secure access to MLaaS is supported by Keycloak, an AAI/IAM service with SSO authentication for external services. The access to the MLaaS platform is done either via an Istio Ingress gateway for some components such as the Kubeflow dashboard and the KServe prediction services, or via the Nginx ingress gateways for other components such as MinIO.

On top of the MLaaS platform, several services, developed by IoT-NGIN, offer IoT-oriented ML features, including adaptive online deep learning, model sharing and translation, and zero-knowledge model verification. These services are introduced in the following sections.

The IoT-NGIN MVP reference implementation of the MLaaS platform has been installed and configured by ArgoCD from service IaC manifests hosted in the IoT-NGIN GitLab repository [2] following a GitOps approach [3].

4.3 Adaptive Online Deep Learning

4.3.1 Introduction

IoT ecosystems consist of a large number of devices (sensors, processors, and communications hardware) that are capable of collecting information about a specific environment, processing that information and sending it without any kind of human interaction. Hence, IoT devices generate dynamic data flows resulting in a non-feasible way to train an ML or deep learning (DL) algorithm in the traditional way (i.e., with a fixed dataset). Online learning (OL) technique allows to train ML models with datasets obtained from dynamic data flows. Thus, models can be retrained every time new data gets available; so the model knowledge is extended continuously. Another advantage of this technique is that models trained with OL can be adapted in real time to changes in the data distribution, minimizing the impact of the data drift problem. Therefore, OL technique can enable the adoption of AI in scenarios where it was not feasible before.

4.3.2 Features

An OL service must offer at least two features based on the characteristics of this AI approach: i) the dynamic training of ML models, as data become available, and ii) the inference provision when requested, by using the latest trained ML model. The dynamic training feature trains the model associated,

by configuration, with the OL service, with datasets continuously fed into the OL service through streaming. The inference provision feature offers predictions generated by the current trained associated model. Model snapshots can be eventually stored in the MLaaS storage when significant performance gains are achieved, regulated by some configurable policies.

To offer the abovementioned main features, OL service supports real-time communication in order to receive the incoming data (see Figure 4.3). Among the communication protocols that are most used in the IoT domain, the Pub/Sub [9] pattern stands out, which allows different services to communicate asynchronously with very low latency. Pub/Sub is made up of producers and consumers. Producers communicate with consumers by broadcasting events. A consumer must be subscribed to a specific topic where the publisher is broadcasting on. The OL service supports real-time communication by integrating Kafka and MQTT. Kafka is an event-based platform that supports Pub/Sub as well as allows event streams to be stored and processed. In addition to real-time data, the OL service also processes data that comes through REST APIs.

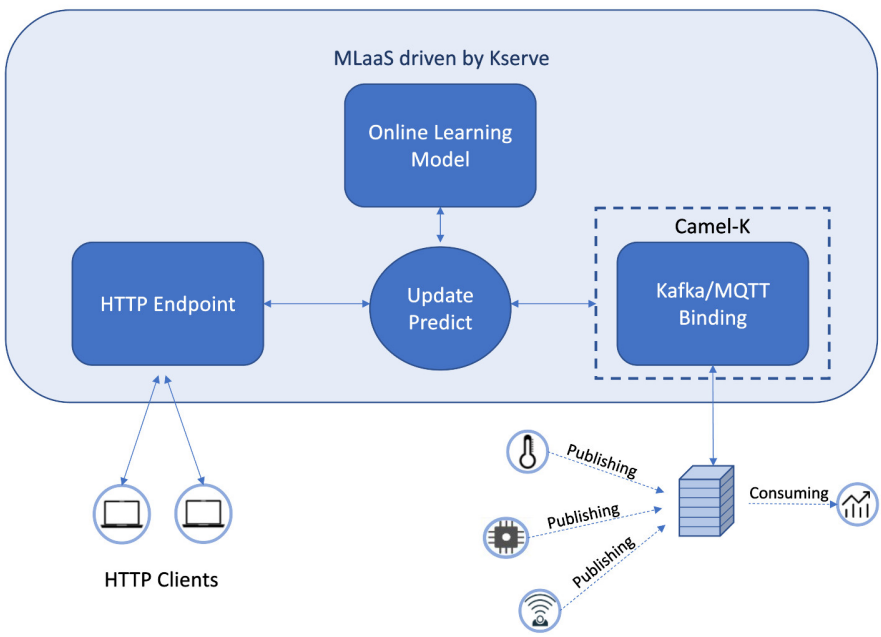

Figure 4.3 Online learning service concept.

Once the OL service receives the data, it uses an ML model to either i) perform a new training or ii) provide inference based on the input data. For this, the OL service supports some of the most popular frameworks for the development of ML models, such as TensorFlow, Keras, PyTorch, Sklearn, and Vowpal Wabbit. The OL service is deployed through the MLaaS KServe framework, which enables serverless model inference, through an HTTP-based REST API. However, as mentioned above, in IoT-NGIN applications, the data is commonly transmitted using protocols other than HTTP. For this reason, the deployment of the OL service requires an MQTT/Kafka-HTTP binding in order to receive the data. The Camel-K framework offers some integrators that perfectly fit this need.

4.3.3 Technical solution

This section describes the technical architecture details of the OL service. Figure 4.4 depicts all components present in the OL service.

As commented in a previous section, data is often sent through streaming flows in IoT scenarios, and the OL service instance only offers an HTTP endpoint; so it does not support, by default, PUB/SUB protocols such as MQTT or Kafka. Thus, a binding acting as a mediator between PUB/SUB and HTTP is needed. The binding is implemented using Camel-K. Whenever new data is published in the broker, the Camel-K binding receives it and redirects it to the HTTP REST API endpoint of the OL service. The binding can be seen as a Kafka/MQTT consumer, which is subscribed to a specific topic and when it receives new data, it redirects it to the OL service.

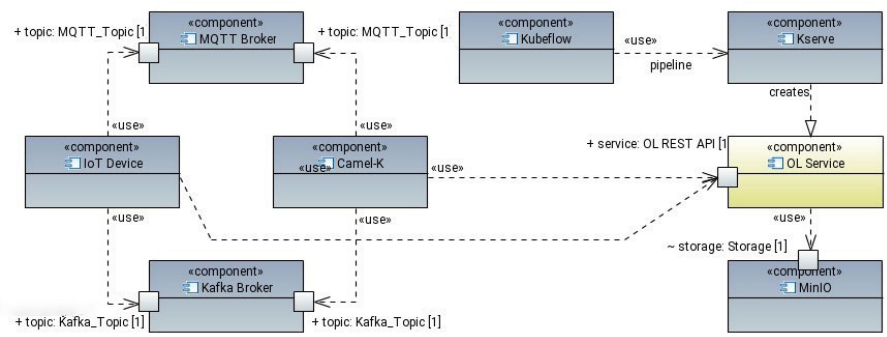

Figure 4.4 Online learning architecture.

When new data is published to retrain the model, the OL service processes this data and performs a training step. It analyzes the model losses and validation metrics and only in case the overall model performance is improved this model version is updated in the MLaaS storage (in this case, in MinIO), replacing the previous one.

Online model training is triggered on demand, by IoT devices, applications, or users, either by directly accessing the HTTP endpoint or by streaming through PUB/SUB on concrete topics. In the former case, a dataset is provided in batches, while in the latter case, dataset is provided in streaming, so that the dataset batch is created by the OL service once enough data is received. Next, data is pre-processed. The pre-processing procedure depends on the ML model architecture and the data structure, and it is use-case specific. KServe allows injecting into the OL workflow a module specialized in the dataset pre-processing stage, known as Transformer. Thus, the Predictor module for training and performing predictions remains independent of the use case and can be reused in any scenario. The only module that needs to be customized to each use case is the Transformer.

In the model inference scenario, predictions are requested by the IoT device, application, or user. The request can include an array with either i) some input data or ii) an empty array. After processing the request, OL returns the prediction when input data is provided, or it returns last available prediction when it is empty. This is useful when working for use cases requiring the forecasting of time series forecasting.

Another module optionally included in the OL service is the Explainer, powered by KServe. This module incorporates, to the OL workflow, an XAI layer that provides an explanation for the prediction performed by the ML model. It consists of a REST API endpoint that is waiting for the input data of the inference request. This module is optional and must be implemented by the ML model developer. If included, the OL offers an explanation to the prediction.

The OL service is deployed using KServe framework through Kubeflow. Kubeflow is utilized to deploy and execute ML workflows and KServe allows to serve ML models on arbitrary AI frameworks. The ML workflow contains the KServe implementation for deploying the OL service, and it is declared within a Kubeflow pipeline. The execution of this pipeline creates an OL service instance in MLaaS, exposed through an HTTP API REST endpoint. This instance encloses an ML model, waiting for incoming data in JSON format, either to be updated (i.e., retrained) or to perform an inference (i.e., prediction).

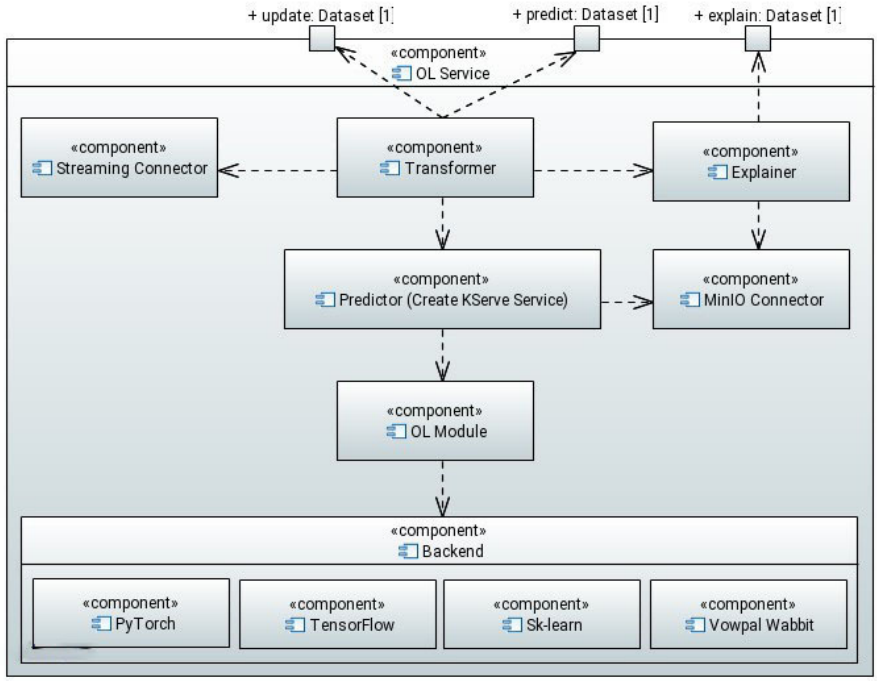

Figure 4.5 OL service internal architecture.

OL service modules have been implemented in Python since it offers a large ecosystem of libraries for AI. OL service is composed of the following modules (see Figure 4.5).

- Predictor (Create KServe service). It is responsible for deploying the REST API service within the OL service. It creates an HTTP endpoint that exposes the OL API for model update or prediction. The main library used in this module is KServe.
- Transformer. It receives a raw dataset and performs the data pre-processing stage. Therefore, it contains all functions needed to prepare the data for the ML model. This module is use case dependent; so it changes for each scenario.
- Explainer. It receives the pre-processed inference input data and returns the significance of each feature in the prediction. It is powered by KServe and must be implemented by the ML model developer.

- Online learning module. This API links the previous module to the backend module. It is responsible for choosing the correct backend and transmitting the model update or prediction requests.
- Streaming connector. This module provides tools to support real-time protocols. This module is not being used because KServe only supports HTTP connections, but it is implemented in case future versions of KServe start working with streaming data. The main libraries that have been used for its implementation and testing are Kafka and Paho-MQTT.
- MinIO connector. It provides the required tools to download and upload the ML models. This version stores the trained ML model in MinIO storage each time the model performance gain overpasses a given threshold. This module is based on the MinIO library.
- Backend. Modules are responsible for including required functions to perform ML model updates or predictions in each framework. The first version includes the following frameworks: Sklearn, Vowpal Wabbit, TensorFlow, and Pytorch.

Apart from the main OL service implementation, additional developments are also required for having the service deployed. They are listed below along with a brief description.

- OL service adaptation: This is the initial step and consists of configuring the OL service to set different parameters such as the MinIO host and the buckets where the ML models are stored, the backend (framework that was used to implement the model) to use in order to perform the model update or the prediction.
- Create the Docker image: Once the OL service is configured, it is required to wrap it within a Docker image that will be uploaded in a Docker registry so that Kubeflow can include it into the pipeline.
- Define KServe YAML manifest: This manifest defines the configuration of the OL service when deployed. It defines the name of the inference service, the number of replicas, the CPU limits, or the Docker image to use, among others.
- Create Kubeflow pipeline: At this point, we have the Docker image ready to use and the KServe YAML manifest that defines the OL service. The next step is to create a Kubeflow pipeline to incorporate the KServe YAML manifest and thus be able to run it.
- Run Kubeflow pipeline: This step deploys the OL service as an HTTP inference service.

- Define Camel-K binding: Camel-K binding consists of a YAML file that defines the broker and topics in which data is being dumped and the prediction service deployed in the previous step in order to resend the data.

4.3.4 Evaluation

This section describes a customized implementation of the MLaaS OL service and its evaluation on a smart energy forecasting scenario, which is depicted in Figure 4.6.

This smart energy scenario consists of a power-voltage (PV) electric grid (EG), whose status metrics are monitored by attached IoT devices. These metrics are published into a MQTT broker in specific topics and consumed by the OL service. The OL service hosts a specific ML model that is continuously trained as soon as new data is available.

The objective of the OL service is to forecast the EG power generation within the next 24 hours, giving a training dataset representing generation in the last 24/36 hours, published in the MQTT topic for power generation. This OL service faces the problem of time series forecasting, where the data becomes available as time goes by, which is a common use case for OL.

Once new data arrives at the OL service, it proceeds with the preprocessing step so that the data is prepared to be processed by the ML model

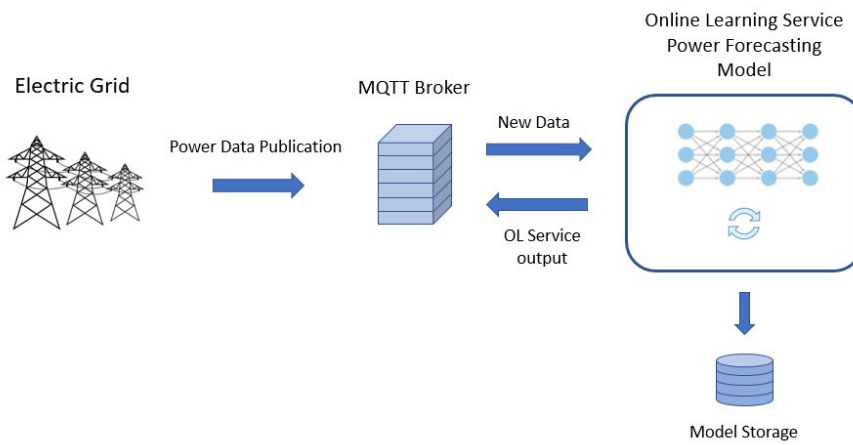

Figure 4.6 Smart energy forecasting scenario.

either for training the model or for inferencing the model in order to obtain forecasting. The following steps summarize the pre-processing stage (see Figure 4.7).

- Extraction of power value: The IoT devices transmit three-phase electric voltage and current data; hence, it is required to extract power value from these data.
- Resampling of the power data: Since the sampling rate in the smart energy scenario is too low, around 1 second, it is needed to resample the power data by aggregating all the values received within 1 hour and computing its average power. For this purpose, the Pandas library is used.
- Data scalation: ML presents higher performance and stability when all values are scaled between 0 and 1. Therefore, the power data is scaled by using the max−min scale strategy with pre-processing functions from the scikit-learn library.
- Time series windowing: The univariate time series forecasting algorithms take vectors as input. This step creates an input vector that contains the scaled averaged power per hour that is used to update the model or perform a prediction. The vectors are created by using different tools from Pandas and Numpy.

After the pre-processing stage, data is ready to train the ML model. However, so far, we have not provided any information about the architecture of the ML model hosted by OL service. The selected model architecture is based in recurrent neural networks (RNNs) [6] since we are facing time series forecasting problems. RRNs have demonstrated to work well when facing time series data, although they present some disadvantages such as the vanishing gradient problem [8]. After some evaluation, the selected layer is the gated recurrent unit (GRU) [7] because it solves the vanishing gradient problem suffered by the original RNN and presents faster convergence rate than other types of RNN such as long short-term memory (LSTM) variant. After the recurrent layers, we add two fully connected layers to apply a

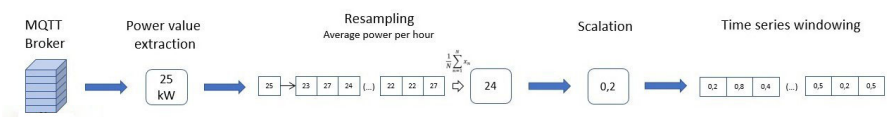

Figure 4.7 OL pre-processing stage.

linear transformation to the outputs of the GRU layers. Figure 4.8 depicts the architecture scheme.

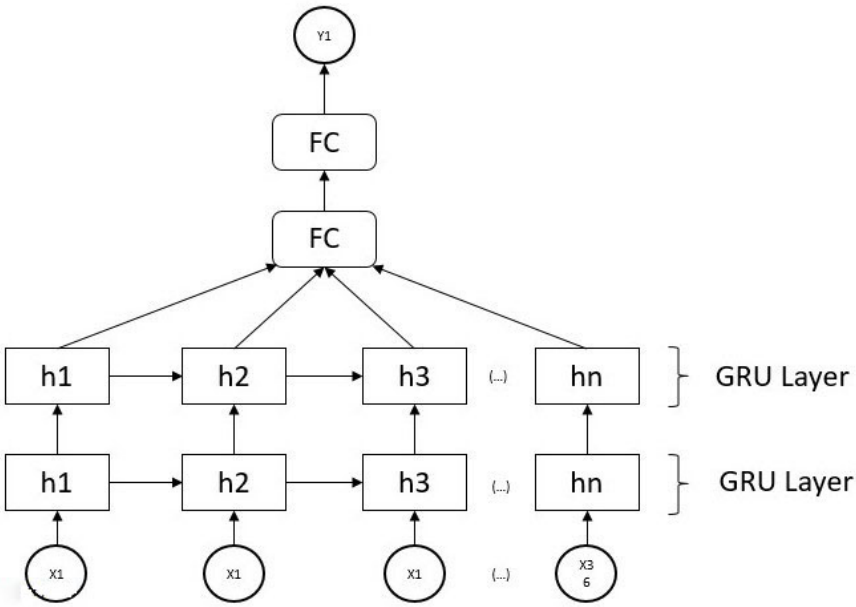

Figure 4.8 DL architecture for smart energy forecasting OL service.

Most of the time series forecasting problems faced with DL use the mean square error (MSE) between real values and model outputs as the loss function since it is one of the simplest to use. The neural network and the training procedure are implemented using the Pytorch library. This library is one of the most extended DL libraries, thanks to its easy-to-use framework with a large number of tools for DL.

To validate that the selected DL model architecture is a valid solution for this power forecasting scenario and deliver a ready-to-use OL forecasting service based on the trained DL model, we started by collecting power dataset for a time frame of 20 days. A data analysis was carried out to find out trends, seasonality, and correlation between power samples. After the analysis, we found out a daily seasonal component; so we could assume a period of 24 hours. After performing a slight experimentation, the OL service uses the training hyper-parameters shown in Table 4.1.

Table 4.1 Hyper-parameters for the OL service.

Hyper-parameter	Value
Epochs	50
Learning rate	0.005
$\beta 1$	0.9
$\beta 2$	0.99
Optimizer	Adam
Loss function	Mean squared error
Batch size	128

We train the DL architecture during 50 epochs with a batch size of 128 samples. We also use Adam optimizer [14] with a learning rate of 0.005 and $\beta 1$ and $\beta 2$ coefficients present values of 0.9 and 0.99, respectively.

Figure 4.9 shows the actual power data (orange line), inferences performed by the DL model (blue points) and the forecasting intervals with a 90% of confidence interval (blue area). It is important to note that the forecasting intervals can be computed since the errors between the actual data and the model predictions present a distribution that can be considered as Gaussian.

To assume errors that come from the Gaussian distribution, they have been subjected to normality tests: Shapiro−Wilk [10], Anderson−Darling [11], and D'Agostino−Pearson [12]. These tests consist of statistical hypothesis tests and allow checking whether the data contains certain property. Thus, two hypotheses are defined: the null hypothesis and the alternative hypothesis. The null hypothesis supports that the data probably comes from a normal distribution while the alternative hypothesis defends that the data present a different distribution. The statistical test returns a probability known as *p*-value. If this result presents a value lower than the defined significance level (0.05 in this case), the null hypothesis must be rejected; so the data distribution cannot be assumed as normal. Table 4.2 shows the *p*-values obtained. These normality tests have been implemented by using the Statsmodels and Scipy libraries.

Table 4.2 Normality test *p*-values.

Normality test	Power generation forecasting
Shapiro−Wilk	0.47
Anderson−Darling	0.76
Agostino−Pearson	0.10

The model can learn the seasonal variations that the generated power seems to have. Moreover, the inferences performed using the validation

subset (data not included during training) offer significant good performance since the MSE obtained is 0.009 (see Figure 4.9).

OL solution includes an optional component to add an XAI dedicated REST API endpoint to provide explanations to the model output and obtain model predictions transparency. For this purpose, we have used the Captum library, which is an open-source Python library specialized in model interpretation methods built on Pytorch.

Captum allows to use different XAI methods to compute the importance of each input feature in the model prediction. Among several methods tested, DeepLIFT (deep learning important features) [13] provided best results; so it is the selected XAI method. This method belongs to the XAI backpropagation-based approach. This approach tries to highlight the input features that are easily predictable from the output.

DeepLIFT consists of on decomposing the output prediction of a neural network on a specific input by backpropagating the contribution of all neurons in the network to every feature of the input. It compares the activation of each neuron to its reference activation (a default or neutral input) and assigns contribution scores according to the difference. DeepLIFT also can separate positive from negative contributions; therefore, the features that have a positive impact on the prediction can be discriminated from the ones with a negative impact.

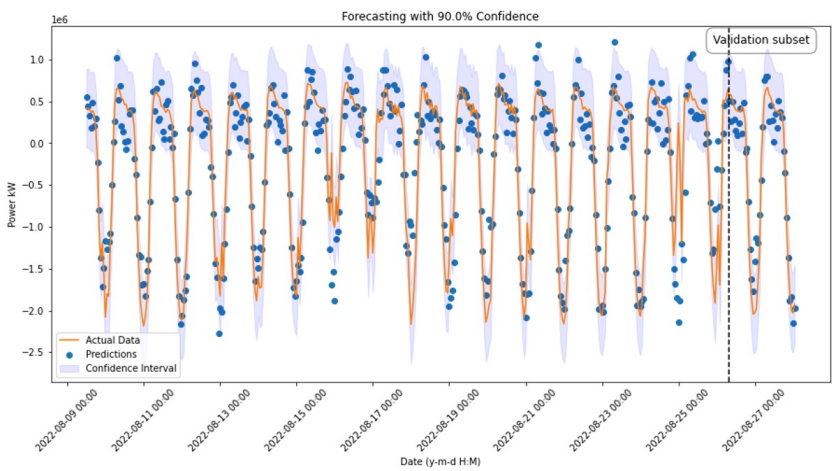

Figure 4.9 Power generation forecasting.

To verify whether DeepLIFT provides reasonable explanations, we have carried out a small evaluation of the power generation forecasting model. For this purpose, we have selected 1 input vector with 36 power measurements. We use DeepLIFT to obtain information about the features that have shown the highest relevance to return the prediction and we represent the contribution scores of each of the samples, as shown in Figure 4.10. Those features with high positive contribution are represented with green points, those that do not present an impact on the prediction are in yellow, and features that present negative contribution are in red. Therefore, DeepLIFT conclusion is the more recent the power sample, the more relevant it is.

At this point, both DL model and DeepLIFT methods have been validated and the OL service deployment can be carried out. The DL model is stored in MinIO so that the OL service can update it or can use it to perform predictions.

The deployment works in the same way described in the previous section. OL service implementation is configured so that the service loads and saves the model in the specific MinIO bucket and uses the Pytorch backend to train or predict, since the model has been defined by using this library. Furthermore, the XAI module script is added to the OL implementation. Then the OL service is wrapped in a Docker image, which is uploaded to a Docker registry. Later, the KServe YAML file is created for the service. The next step consists of creating the Kubeflow pipeline and executing it; so the OL service

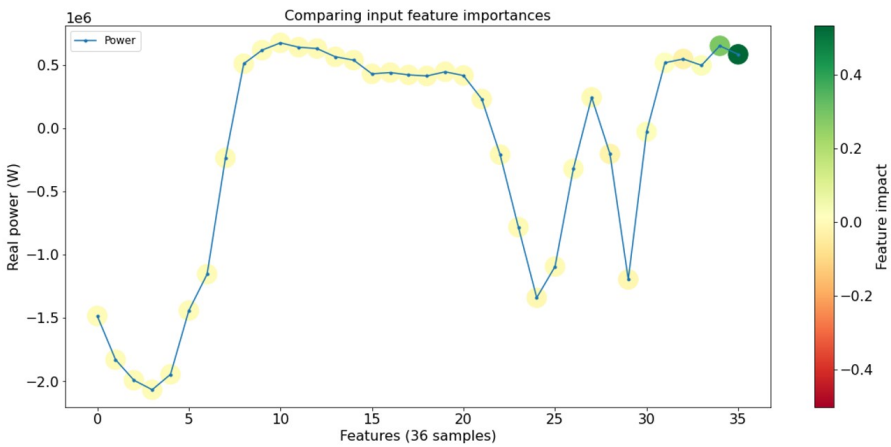

Figure 4.10 XAI analysis of power generation forecasting.

is deployed with an HTTP endpoint. Finally, the Camel-K binding is created indicating the MQTT broker address with the specific topic.

MLaaS online learning service code is available at the IoT-NGIN GitLab repository [4].

4.4 Model Sharing, Model Translation, and Zero-knowledge Model Verification

4.4.1 Introduction

The main motivation for the zero-knowledge verification framework is to provide a mechanism for ensuring the following:

1. The training phase of a machine learning model exclusively involves the inputs declared by the model owners.
2. Replicability of the training phase.
3. Immutability of the results of the training phase (i.e., trained model).
4. Ability to create an intermediate representation of the trained model in a common machine learning framework.

Since the training phase of an ML model is a deterministic process (provided the required initial conditions, including the seeds for any operation involving PRNG, e.g., batch normalization, and excluding non-deterministic models, i.e., VAE), by having full control of the datasets involved, the model architecture and hyper-parameter values, we can ensure that the model weights resulting from the training are the direct result from the inputs provided [18]. For this reason, to ensure that the resulting weights of a trained model exclusively involve the provided datasets, the training phase must be carried out in the premises of the system. To provide the ability for verifying and tracing the lineage of all inputs of the models, a platform for storing all relevant metadata and datasets involved is required. Additionally, the possibility for creating an intermediate representation of the model maximizes the compatibility of the registered models across a wide range of execution platforms, avoiding lock-in of the models in their original machine learning frameworks.

4.4.2 Features

In this section, we will introduce the services that make up the architecture of the system, and their responsibilities.

Model sharing service – features overview:

The model sharing service oversees the model training phase and storage of the results, given a model implemented in one of the supported ML backends, a set of hyper-parameter values, and the datasets involved. In addition to this, it is also in charge of providing access to, and enforcing access rules for the registered datasets and models.

The model registration process depends on the following inputs:

- model architecture implemented in one of the supported ML backends;
- an existing dataset in the system, for which the company and model developer are granted access to;
- hyper-parameter values for the model.

For each registered model, and all its associated inputs, a smart contract containing all the relevant metadata to ensure the reproducibility of the results of the training phase is deployed in the blockchain via the zero-knowledge verification service (see next subsection). The static files for the model architectures, resulting model weights and training metadata, as well as the datasets involved are stored in an object storage repository, to enforce control over the full storage lifecycle. To ensure the isolation of company resources in the shared object storage instance, each company resources are stored in independent buckets, with access credentials scoped to the company's resources. These credentials are obscured away from the end users and are generated and used exclusively in a programmatic manner by the system

Model training service – features overview:

The model training service oversees the training phase for each model registered via the model sharing service. To perform the training phase for a registered model, it first validates all the necessary inputs (as described in the introduction of the model sharing service).

Zero-knowledge model verification service – features overview:

The zero-knowledge model verification service provides a framework for end-to-end verification of stored models, and dataset identification. The blockchain is based on the Quorum blockchain service [16], which is an open source private blockchain platform with a fully capable implementation of the Ethereum virtual machine. For each model and dataset stored, there is a

corresponding smart contract deployed in the blockchain. When deploying an Ethereum smart contract, the Ethereum virtual machine (EVM) stores internal bits of information, which are accessible when transacting with the contract. In this manner, we use smart contracts as sources of truth for all the relevant metadata for datasets and models. The metadata stored in the blockchain differs for models and datasets. As mentioned before, the objective is to store all the necessary metadata to verify and ensure the traceability and replicability of trained models. For this reason, one of the key pieces of stored metadata for trained models is a hash of the model weights, which allows the model sharing service to ensure the integrity of the artifact in the object storage repository. For datasets, as computing the hash of large files is a computationally intensive task, we store relevant statistics (unidimensional, matrix), as well as sample sizes, and total samples. By storing the hash of the model representation in the deployed smart contract, we can ensure the integrity of the stored models. We also store other model metadata, such as input and output vector sizes, and other relevant information regarding the datasets involved in the training phase.

Model translation service – features overview:

The model translation service provides a framework for generating an intermediate representation for machine learning models implemented in several frameworks. It leverages ONNX [17], a machine learning framework used as an intermediate compatibility layer between other popular machine learning frameworks, by providing an open format for representing machine learning models. One of the main use cases for needing an intermediate representation is due to hardware optimization concerns. Depending on the framework in which a machine learning model has been developed, the framework's backend implementation may apply different optimizations to different hardware. The goal of using ONNX is to be able to access the implemented hardware-specific optimizations avoiding the lock-in of the implementation in a particular framework, allowing the development of machine learning models in an open format that can be used to leverage the hardware optimizations implemented by other frameworks regardless of the original implementation's backend. There exist implementations for providing compatibility layers with ONNX for the most popular machine learning frameworks, e.g., PyTorch, Tensorflow, and scikit-learn. Some of these implementations are community efforts (i.e., open-source implementations), and in other cases such as in PyTorch, support is built in the framework.

4.4.3 Technical implementation

In this section, we will provide further details about the technical implementation of the services, and further insight into their interactions, as shown in Figure 4.11. Before further introducing the services individually, we will mention some guiding architectural and engineering practices followed in the development phase. We have followed a microservices approach for the design and development of the services. All the introduced services are implemented in Python, using the FastAPI framework. The OpenAPI specification for the services is generated dynamically by FastAPI. Documentation for the services is offered via Swagger, provided by FastAPI. Clients for the services APIs are programmatically generated with the OpenAPI generator library by using the OpenAPI specifications provided by FastAPI. For deployment, we followed a container-based approach, using Kubernetes as the container orchestration framework of choice.

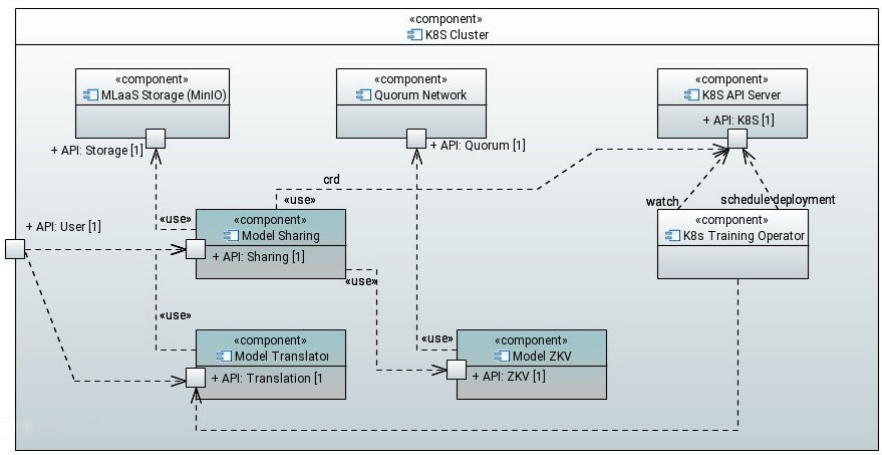

Figure 4.11 Internal architecture.

Model sharing service – technical details of implementation:

The access to the MLaaS block storage (powered by a MinIO instance) layer is handled exclusively by this service. Abstractions are provided for storage and retrieval of any size datasets and model architecture artifacts, as well as the results of the training processes. When storing datasets, and depending on the nature of each dataset, relevant statistics are collected and stored

in the blockchain, in addition to storing the dataset in the block storage. Equivalently for model registration petitions, relevant metadata is extracted from the user's uploaded file and stored in the blockchain service in the form of an Ethereum smart contract. Each dataset and model is provided with a unique ID that is exposed to the user, and stored in an internal MLaaS relational database. For authorization purposes, each entry of this database registers the developer and its company in order to restrict the access of these assets. Authentication and authorization for storing and accessing models is implemented with MLaaS Keycloak. The JWT tokens provided by Keycloak provide with all necessary information for the model sharing service to enforce access rules for the tenants.

Upon model registration requests, the model sharing service will initiate a model training job, by means of the model training's HTTP API using its Python client implementation. Equivalently, communication with the blockchain service is achieved by means of the blockchain service's API Python client. For models for which their training jobs have finished successfully, the model sharing API allows for the download of the training results, after verifying the integrity of its associated artifacts by means of comparison of the hash stored in the blockchain contract, and the hash of the artifact upon download from the object storage layer. This service relies on a relational database in which, for each registered model, information about the company, developer, and smart contract address is stored.

The following operations are implemented in the HTTP REST API:

1. dataset and model registration (contract and storage in block storage layer);
2. dataset and model download.

Zero-knowledge model verification service – technical details of implementation:

The blockchain is powered by an instance of Quorum (MLaaS DLT), a private distributed ledger technology (DLT) implemented as a fork of the Ethereum blockchain. In our use case, a single member runs all the nodes that make up the network. Quorum supports private transactions (supported by Tessera, a component for private transaction management), in which encrypted data can be transferred between network participants and stored in a way such that only the involved participants can see the data. It is fully compatible

with the Ethereum APIs and implements full support for smart contracts. To interact with the blockchain, we use the Web3.py Python library, and use the Solidity language specification for developing the smart contract code. The service uses separate addresses for each company to interact with the blockchain. These addresses' private keys are never shared with the end users. Using separate addresses for each company reduces the surface attack, limiting the exposure of any leaked private key to the scope of the company. Smart contracts are therefore deployed to the accounts issued for each of the companies.

The Ethereum virtual machine (EVM) runs contract code to completion or up to transaction gas (currency for covering transaction costs) exhaustion. However, Quorum allows for free-gas networks, and since it is a single member network, there exist no incentives for requiring gas for transacting in the network. In addition to the deployment of smart contracts for models and datasets, the service also implements the ability to fetch all stored metadata for already deployed contracts.

The following operations are implemented in the HTTP REST API:

1. deploying of smart contracts for datasets in the blockchain;
2. deployment of smart contracts for models in the blockchain;
3. fetching all metadata stored in a contract address.

Model translation service – technical details of implementation:

The model translation service provides a compatibility layer between three of the most widely spread ML frameworks (Pytorch, Tensorflow, and scikit-learn) by providing a managed service on top of ONNX [17]. Requests for model translation are allowed for all models already registered and for which their training jobs have finished successfully.

The following operations are implemented in the HTTP REST API:

1. generate intermediate representation in ONNX for an existing model.

Model training service – technical details of implementation:

The model training service is implemented as a Kubernetes custom operator, using the framework Kopf. Model training jobs are modeled as custom Kubernetes resource definitions (CRD), which include all the relevant information for the operator to process the training job.

On model registry, model developers must also specify a Docker image to be used as the environment for the model training. This requirement is set

in order to provide model developers with maximum flexibility as well as allowing model developers to share the same environment across their local model development and training, and the training procedure performed in the model training cluster, essentially abstracting end users from any possible overhead introduced by the zero-knowledge verification process.

The results of the training are treated as output artifacts and must be stored in a specific output directory. This output directory is a mount of a Kubernetes persistent volume claim (PVC), and the contents of the PVC will be stored as a result of the model training job in the block storage upon an exit code that is received from the training container.

The custom resources are processed by the implemented Kubernetes operator in a queue-like manner and can be processed in either a sequential or parallel manner, depending on the availability and scalability of the hardware.

The model sharing service, upon the reception of a model registration request, will create an instance of the custom resource definition in the training Kubernetes cluster, with all the relevant input for the cluster to schedule the training job. This input contains:

- pointer to the UID of target model architecture;
- pointer to the UID of the target datasets;
- image registry link for the Docker training image;
- all relevant model starting conditions (e.g., PRNG seeds), exposed as environment variables to the containers running the specified Docker image.

The following operations are implemented in the HTTP REST API:

1. register model training jobs (creates the CRD in the target Kubernetes cluster);
2. check processing status for model training jobs.

The implementation code of the MLaaS services model sharing, model translator, and zero-knowledge verification service is available at the IoT-NGIN GitLab repository [15].

4.4.4 Evaluation

To exemplify the usage of the zero-knowledge verification framework, we will introduce the following test case, in which we register the Pix2Pix generative model [19] and the CMP facade dataset [20] and, after its training,

generate the intermediate representation of the trained model in the ONNX framework (see Figure 4.12).

The Pix2Pix model implements a cGAN (conditional generative adversarial network), which is able to map input images to output images of a learned distribution from the training samples. For our use case, we will use building facades as the training input, and we expect the trained model to produce artificial facades, based on the learned latent distribution from the CMP facade dataset.

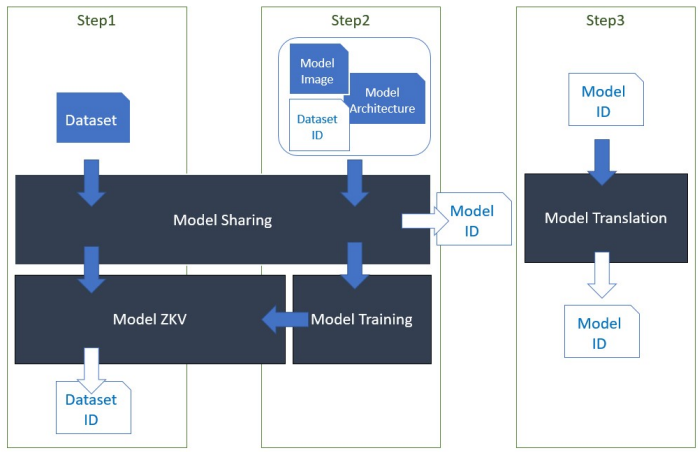

Figure 4.12 Evaluation process.

Before being able to register the model in the system, it is necessary to register all its associated datasets in the model sharing service (in our case, the CMP facade dataset). To do that, we make an HTTP POST request to the datasets endpoint in the model sharing service. The HTTP POST request is a multipart request, including the model architecture file and the initial metadata for the model (e.g., sample number, sample dimension, etc.).

```
curl --request POST\

--url http://<mlaas_model_sharing> /dataset/\

--header ‘Content-Type: multipart/form-data’\

--form model=@model\
```

--form ‘metadata={ ‘‘sample_dimension’’: ‘‘(500,500)’’, ‘‘sample_num’’ : 606}}’

Once the dataset is successfully registered in the system, we proceed with the registration of the model. In a similar way to the dataset registration process, to register the model in the system, we make an HTTP POST request to the model registration endpoint in the model sharing service. This POST request is also a multipart request, with the file containing the model architecture developed in the original machine learning framework, and the required model metadata (e.g., ID of the training dataset, input sample dimensions, total number of parameters, and initial conditions for all the hyper-parameters of the model, and link to the registry hosting the training Docker image).

curl --request POST\

--url http://<mlaas_model_sharing>/model/\

--header ‘Content-Type: multipart/form-data’\

--form model=@model\

--form ‘metadata={ ‘‘model_params’’: { ‘‘sample_dimension’’: ‘‘(500, 500)’’, ‘‘params’’: ‘‘<json_params_initial_cond>’’}, ‘‘data_params’’: { ‘‘dataset_id’’: ‘‘<dataset_id>’’}}’

In the model registration step, the model sharing service, upon verifying all necessary inputs, will trigger a new job in the model training service to schedule the training of the model and its associated datasets using the specified Docker image.

The model training service will then execute the training job, scheduling the Kubernetes deployment, and executing until an exit code is received from the container. Upon receiving a graceful exit code, it will then copy all the contents of the persistent volume attached to the container into the block storage layer, via the model sharing service. In this step, the model sharing service will also update the smart contract associated with the model with the metadata of the final trained model (e.g., model weights hash). We can verify the updated metadata by making a GET request to the model sharing service metadata endpoint:

curl --request GET\

--url http://<mlaas_model_sharing> /metadata/<model_id>

From this step onwards, the model is ready to be used for inference, being able to download it securely using the model sharing service, or to create an intermediate representation of the trained model in ONNX via the model translation service.

curl --request GET\

--url http://<mlaas_model_sharing>/model/<model_id>

To create an intermediate representation of the model, the developer must perform an HTTP POST request to the model translation service, specifying the ID of the trained model:

curl --request POST\

--url http://<mlaas_model_translation> /translate /<model_id>

Note that the service will reject any petitions for registered models for which its training jobs are not completed. Upon receiving the request, the model translation service will then perform the conversion of the model from the initial framework to ONNX and will store the results of this operation under a new ID (i.e., a new model). This new model will have all the invariant metadata of the original model, except for the changing metadata, e.g., backend (i.e., framework) of the model, model hash. Note that there was no training step involved in the model translation step. This is since ONNX allows for model conversion without the need of re-training the model.

Therefore, the model is now available in the original backend under the ID associated upon its registration in the system, and the ONNX version of the same model is also available under a newly assigned ID (as received in the response of the model translation service API call). It is necessary to assign different IDs as the zero-knowledge verification framework treats models as individual, independent units, due to the uniqueness of the metadata involved in the verification process.

4.5 Conclusion

This chapter has introduced the IoT-NGIN concept of MLaaS, its main features, and the implementation details of the MVP instantiated in

the IoT-NGIN cluster. It has also provided the functional and technical specification of additional MLOps services incorporated into MLaaS: services required for MLOps in the IoT domain, such as the adaptive online learning service, intended for the ML model training and inference from streaming datasets, or other services that extend the MLOps functionality, such as the model sharing, model translator, and zero-knowledge model verification, which are not part of the MLOps frameworks available in the open-source community.

MLaaS is being used by IoT-NGIN use cases for the MLOps management of ML-based applications in the IoT domain. In particular, the usage of MLaaS for online model training and inference for smart energy forecasting has been used in the evaluation of the online learning services. The adoption of MLaaS in the other IoT-NGIN use cases will be the focus of development in the rest of the IoT-NGIN project.

Acknowledgements

The work presented in this chapter has been funded by the IoT-NGIN project, contract no. 957246, within the H2020 Framework Program of the European Commission.

References

[1] "The 5 V's of big data". Watson Health Perspectives. 17 September 2016.

[2] MLaaS platform. IoT-NGIN Gitlab repository: https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/ml-as-a-service

[3] Beetz, F., & Harrer, S. (2021). GitOps: The Evolution of DevOps?. IEEE Software.

[4] MLaaS Online Learning repository at IoT-NGIN Gitlab: https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/t3_2/online_learning

[5] Sayan Putatunda (2021), Practical Machine Learning for Streaming Data with Python: Design, Develop, and Validate Online Learning Models. ISBN: 9781484268674. Appress.

[6] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning internal representations by error propagation," San Diego, California: Institute for Cognitive Science, University of California., 1985.

[7] Rahul Dey, Fathi M. Salem, "Gate-Variants of Gated Recurrent Unit (GRU) Neural Networks", Department of Electrical and Computer Engineering Michigan State University, 2017

[8] Razvan Pascanu, Tomas Mikolov, Yoshua Bangio, “On the difficulty of training Recurrent Neural Networks,” 2012.
[9] Jonathan Matsson, “The Publish-Subscribe Pattern”, 2018
[10] S. S. Shapiro, M. B. Wilk, “An analysis of Variance Test for Normality (Complete Samples)”, 1965
[11] T. W. Anderson. D. A. Darling, “Asymptotic Theory of Certain ‘Goodness of Fit’ Criteria Based on Sthocastic Processes”, 1952
[12] Ralph D’Agostino and E. S. Pearson, “Tests for Departure from Normality. Empirical Results for the Distributions of b2 and $\sqrt{}$b1”, 1973, p.613-622.
[13] Avanti Shrikumar, Peyton Greeside, Anshul Kubdaje, “Learning Important Features Through Propagating Activation Differences”, 2019
[14] Diederik P. Kingman, Jimmy Lei Ba, “ADAM: A method for stochastic optimization,” 2014
[15] MLaaS Model Sharing, Model Translator and Zero Knowledge Verification repository at IoT-NGIN Gitlab: https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/t3_4/ml-model-sharing
[16] Quorum Blockchain Service: https://consensys.net/quorum/qbs/
[17] ONNX: https://onnx.ai/
[18] Chen, Boyuan, et al. “Towards Training Reproducible Deep Learning Models.” Proceedings of the 44th International Conference on Software Engineering, 2022, https://doi.org/10.1145/3510003.3510163.
[19] Pix2Pix model: examples/tensorflow_examples/models/pix2pix at master ů tensorflow/examples ů GitHub
[20] CMP Facade Dataset: CMP Facade Database (cvut.cz)