

DRAFT

## D3.4

# ML models sharing and transfer learning implementation

**WORKPACKAGE** WP3

**DOCUMENT** D3.4

**REVISION** V1.0

**DELIVERY DATE** 31/03/2023

**PROGRAMME IDENTIFIER** H2020-ICT-2020-1

**GRANT AGREEMENT ID** 957246

**START DATE OF THE PROJECT** 01/10/2020

**DURATION** 3 YEARS

© Copyright by the IoT-NGIN Consortium

This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No 957246



## DISCLAIMER

This document does not represent the opinion of the European Commission, and the European Commission is not responsible for any use that might be made of its content.

This document may contain material, which is the copyright of certain IoT-NGIN consortium parties, and may not be reproduced or copied without permission. All IoT-NGIN consortium parties have agreed to full publication of this document. The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the IoT-NGIN consortium as a whole, nor a certain party of the IoT-NGIN consortium warrant that the information contained in this document is capable of use, nor that use of the information is free from risk, and does not accept any liability for loss or damage suffered using this information.

## ACKNOWLEDGEMENT

This document is a deliverable of IoT-NGIN project. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 957246.

The opinions expressed in this document reflect only the author's view and in no way reflect the European Commission's opinions. The European Commission is not responsible for any use that may be made of the information it contains.

<b>PROJECT ACRONYM</b>	IoT-NGIN
<b>PROJECT TITLE</b>	Next Generation IoT as part of Next Generation Internet
<b>CALL ID</b>	H2020-ICT-2020-1
<b>CALL NAME</b>	Information and Communication Technologies
<b>TOPIC</b>	ICT-56-2020 - Next Generation Internet of Things
<b>TYPE OF ACTION</b>	Research and Innovation Action
<b>COORDINATOR</b>	Capgemini Technology Services (CAP)
<b>PRINCIPAL CONTRACTORS</b>	Atos Spain S.A. (ATOS), ERICSSON GmbH (EDD), ABB Oy (ABB), NETCOMPANY-INTRASOFT SA (INTRA), Engineering-Ingegneria Informatica SPA (ENG), Robert Bosch Espana Fabrica Aranjuez SA (BOSCHN), ASM Terni SpA (ASM), Forum Virium Helsinki (FVH), ENTERSOFT SA (OPT), eBOS Technologies Ltd (EBOS), Privanova SAS (PRI), Synelxis Solutions S.A. (SYN), CUMUCORE Oy (CMC), Emotion s.r.l. (EMOT), AALTO-Korkeakoulusaatio (AALTO), i2CAT Foundation (I2CAT), Rheinisch-Westfälische Hochschule Aachen (RWTH), Sorbonne Université (SU)
<b>WORKPACKAGE</b>	WP3
<b>DELIVERABLE TYPE</b>	REPORT
<b>DISSEMINATION LEVEL</b>	PUBLIC
<b>DELIVERABLE STATE</b>	FINAL
<b>CONTRACTUAL DATE OF DELIVERY</b>	31/03/2023
<b>ACTUAL DATE OF DELIVERY</b>	31/03/2023
<b>DOCUMENT TITLE</b>	ML models sharing and transfer learning implementation
<b>AUTHOR(S)</b>	J. Mira (ATOS), I. Moreno(ATOS), J. Gorroñoigoitia Cruz (ATOS), T. Velivassaki (SYN), Ch. Betzelos (SYN), D. Skias (INTRA)
<b>REVIEWER(S)</b>	A. Voulkidis (SYN), Dimitrios Skias (INTRA)
<b>ABSTRACT</b>	SEE EXECUTIVE SUMMARY
<b>HISTORY</b>	SEE DOCUMENT HISTORY
<b>KEYWORDS</b>	Deep Learning, Reinforcement Learning, Online Learning, Federated Learning, Machine Learning, Privacy Preservation, AI, ML

## Document History

Version	Date	Contributor(s)	Description
V0.1	27/01/2023	ATOS	Table of Contents
V0.2	17/03/2023	ATOS, SYN	Sections 1-5
V0.3	21/03/2023	ATOS	Peer-review version
V0.4	24/03/2023	ATOS, SYN, INTRA	Peer-review
V0.5	28/03/2023	ATOS, SYN	Post-Peer-Review version
V0.6	29/03/2023	ATOS	Camera-ready version
V0.7	30/03/2023	CAP	Final quality check
V1.0	31/03/2023	ATOS	Final version

DRAFT - PENDING EC APPROVAL

# Table of Contents

Document History .....	4
Table of Contents .....	5
List of Figures.....	7
List of Tables.....	9
List of Acronyms and Abbreviations.....	10
Executive Summary .....	13
1 Introduction.....	14
1.1 Intended Audience.....	15
1.2 Relations to other activities .....	16
1.3 Document overview .....	18
2 IoT-NGIN Machine Learning.....	19
2.1 Online Learning.....	19
2.1.1 Description.....	19
2.1.2 Technical design.....	20
2.1.3 Implementation for IoT-NGIN LLs .....	23
2.2 Reinforcement Learning .....	30
2.2.1 Description.....	31
2.2.2 Technical design.....	31
2.2.3 Implementation for IoT-NGIN LLs .....	33
3 IoT-NGIN Privacy Preserving Federated Learning.....	36
3.1 Description.....	37
3.2 Technical design.....	38
3.2.1 Description of subcomponents .....	39
3.2.2 Interfaces.....	40
4 Polyglot Model Sharing.....	43
4.1 Description.....	43
4.2 Technical Design.....	43
4.2.1 Architecture.....	43
4.2.2 Implementation .....	47
4.3 Implementation for IoT-NGIN LLs .....	49
5 Installation and User Guide.....	55
5.1 Online Learning Service .....	55
5.2 Reinforcement Learning Service .....	58

D3.4 – ML models sharing and transfer learning implementation

5.3	Privacy-preserving Federated Learning Framework .....	58
5.3.1	Prerequisites.....	58
5.3.2	Installation Guide.....	58
5.3.3	User Guide .....	63
5.4	Polyglot Model Sharing Framework .....	72
5.4.1	External dependencies.....	73
5.4.2	Running Services Locally.....	74
5.4.3	Deployment on Kubernetes Clusters .....	75
6	Conclusions .....	78
7	Annex Components and Interfaces for Polyglot Model Sharing .....	79
7.1	Model Sharing service.....	79
7.1.1	HTTP REST API Operations.....	79
7.1.2	Dataset Registration.....	80
7.1.3	Model Registration, Training and Retrieval .....	80
7.1.4	Model Metadata Retrieval.....	81
7.2	Blockchain service.....	82
7.2.1	Smart Contract Interfaces.....	82
7.2.2	HTTP REST API Operations.....	85
7.2.3	Dataset and Model Contract Deployment.....	85
7.2.4	Contract metadata request.....	87
7.2.5	Model verification.....	87
7.3	Model Training service.....	88
7.3.1	HTTP REST API Operations.....	88
7.3.2	Model training request.....	89
7.4	Model Translation service .....	90
7.4.1	HTTP REST API Operations.....	90
7.4.2	Model translation request.....	91
8	References .....	92

DRAFT - PENDING EC APPROVAL

## List of Figures

Figure 1 - Work packages structure .....	16
Figure 2 - Overall architecture of the OL service .....	21
Figure 3 - Technical architecture of the OL service .....	22
Figure 4 - DeepLIFT results for power generation forecasting in UC9 .....	25
Figure 5 - DeepLIFT results for power generation forecasting in UC10 .....	25
Figure 6 - Model performance monitoring .....	26
Figure 7 - Data drift monitoring .....	27
Figure 8 - DL architecture .....	28
Figure 9 - Training results for power consumption forecasting of UC10 .....	29
Figure 10 - Preprocessing stage object detection model .....	30
Figure 11 - RL service high level scheme .....	32
Figure 12 - Sequence diagram of interaction between Tensorforce and environment .....	32
Figure 13 - Energy demand for Domestic clusters .....	34
Figure 14 - Energy demand for Industrial clusters .....	34
Figure 15 - The PPFL API deployment approach through Docker containers, K8S and Argo Workflows .....	38
Figure 16 - The technical design of the PPFL API .....	39
Figure 17 - Architecture diagram of the Polyglot Model Sharing framework .....	44
Figure 18 - Polyglot Model Sharing framework implementation repository directory structure .....	47
Figure 19 - Sequence diagram of the Polyglot Model Sharing Framework demo use case ..	49
Figure 20 - Demo use case's implementation directory structure .....	50
Figure 21 - Dataset artifact stored in the object storage .....	52
Figure 22 - Trained model in MLaaS storage .....	54
Figure 23 - Workflow to deploy OL service .....	55
Figure 24 - Workflow to deploy OL monitoring service .....	57
Figure 25 - REST call to retrieve a user access token. ....	63
Figure 26 - Parameters setting in the POST request body for running the FedPATE framework. ....	64
Figure 27 - Deployed pods for FedPATE training .....	65
Figure 28 - Training through FedPATE ran and completed .....	65
Figure 29 - Final model trained through FedPATE has been stored in MLaaS' Minio storage	66
Figure 30 - Starting the NVIDIA FLARE server through the PPFL API .....	67
Figure 31 - NVIDIA FLARE server and observer pods .....	67

Figure 32 - Running NVIDIA FLARE client software for client-1 .....68

Figure 33 - Indicative parameter setting for starting a training process though NVIDIA FLARE .....69

Figure 34 - Logs of the training process at the client side.....69

Figure 35 - Final model of NVIDIA FLARE training stored in Minio model storage component of the MLaaS platform.....70

Figure 36 - Instantiating TFF training through the PPFL API .....71

Figure 37 - The training process through TFF has been completed.....71

Figure 38 - Final model trained through TFF stored in MLaaS' Minio storage .....72

Figure 39 - Project's GitLab container registry interface .....73

Figure 40 - Model Sharing service's Kubernetes manifests.....76

Figure 41 - Sequence diagram of the Model Sharing service's dataset registration operation .....80

Figure 42 - Sequence diagram of the Model Sharing service's model registration and retrieval .....81

Figure 43 - Sequence diagram of the Model Sharing service's model metadata retrieval operation .....82

Figure 44 - Sequence diagram of a dataset's contract deployment operation .....86

Figure 45 - Sequence diagram of a model's contract deployment operation .....86

Figure 46 - Sequence diagram of the Blockchain service's contract metadata request operation .....87

Figure 47 - Sequence diagram of the Blockchain service's model verification operation.....88

Figure 48 - Sequence diagram of the Model Training service's training request operation...89

Figure 49 - Sequence diagram of the Model Training service's training job workflow implementation .....90

Figure 50 - Sequence diagram of the Model Translation service's model translation operation .....91

DRAFT - PENDING APPROVAL



## List of Tables

Table 1 - Relation of WP3 activities with other WPs and tasks .....	17
Table 2 - Smart Energy LL MQTT topics for forecasting services .....	23
Table 3 - Transfer Learning hyper-parameters.....	28
Table 4 - Normality test results (p-values) .....	29
Table 5 - Details of the ML and FL frameworks analyzed in D3.3.....	37
Table 6 - PPFL API interfaces.....	40
Table 7 - Environmental variables' configuration for local deployment .....	58
Table 8 - Environmental variables' configuration for Docker deployment.....	60
Table 9 - Environmental variables' configuration for K8S deployment.....	61
Table 10 - Model Sharing service HTTP REST API operations.....	79
Table 11 - Dataset smart contract metadata specification .....	83
Table 12 - Dataset smart contract operations .....	83
Table 13 - Model smart contract metadata specification.....	84
Table 14 - Model smart contract operations .....	84
Table 15 - Blockchain service HTTP REST API operations.....	85
Table 16 - Model Training service HTTP REST API operations .....	88
Table 17 - Model Translation service HTTP REST API operations .....	90

DRAFT - PENDING EC APPROVAL

## List of Acronyms and Abbreviations

AAA	Authentication, Authorization and Accounting
ABI	Application Binary Interface
AI	Artificial Intelligence
AP	Average Precision
API	Application Programming Interface
BCE	Binary Cross-Entropy
BDVA	Big Data Value Association
CA	Certificate Authority
CI/CD	Continuous Integration/Continuous Delivery
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DL	Deep Learning
DP	Differential Privacy
DQN	Deep Q-Network
D<X>	Deliverable
ECG	Electric Charging Station
EG	Electric Grid
EIP	Enterprise Integration Patterns
EV	Electric Vehicle
EVM	Ethereum Virtual Machine
FC	Federated Core
FL	Federated Learning
FQDN	Fully-Qualified Domain Name
GAN	Generative Adversarial Network
GitOps	Git Operations
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
HE	Homomorphic encryption

HTTP	HyperText Transfer Protocol
IaC	Infrastructure as Code
ID	Identifier
IDS	Intrusion Detection System
IoT	Internet of Things
IoU	Intersection over Union
JSON	JavaScript Object Notation
LL	Living Labs
mAP	mean Average Precision
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
ME	Mean Error
ML	Machine Learning
MLaaS	Machine Learning as a Service
MLOps	ML Operations
MQTT	MQ Telemetry Transport
MSE	Mean Squared Error
NNI	Neural Network Intelligence
NV FLARE	NVIDIA Federated Learning Application Runtime Environment
OL	Online Learning
ONNX	Open Neural Network Exchange
OS	Operating System
PATE	Private Aggregation of Teacher Ensembles
POC	Proof-Of-Concept
PPFL	Privacy-Preserving Federated Learning
PPFLaaS	Privacy-Preserving Federated Learning as a Service
REST	Representational State Transfer
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SCR	Self-Consumption Ratio

SDK	Software Development Toolkit
SGD	Stochastic Gradient Descent
SRIA	Strategic Research and Innovation Agenda
SSL	Secure Sockets Layer
SSR	Self-Sufficiency Ratio
SVT	Sparse Vector Technique
TFF	TensorFlow Federated
TPU	Tensor Processing Unit
UC	Use Case
UI	User Interface
UUID	Universal Unique Identifier
VOC	Visual Object Class
WP	Work Package
w.r.t.	With regards to
XAI	eXplainable Artificial Intelligence
YAML	Yet Another Markup Language
YOLO	You Only Look Once

DRAFT - PENDING EC APPROVAL

## Executive Summary

Endowing IoT-based applications and devices with intelligence capabilities to take informed decisions based on the device environment is of paramount importance in nowadays application scenarios. IoT-NGIN offers a set of frameworks and services aimed for enhancing the intelligence of these IoT applications and devices, namely: I) The Machine Learning as a Service (MLaaS), as the the main IoT-NGIN MLOps platform for IoT, and II) the Privacy Preserving Federated Learning (PPFL) platform, specialized for federated Learning, and iii) complementary services and frameworks, including the online learning service and the model sharing framework, that are delivered with the MLaaS.

This document describes the progress over D3.3, towards the fulfilment of these IoT-NGIN goals to endow IoT applications with intelligence. Specifically, the main outcomes towards this goal reported in this deliverable include:

- Extended and new features for online learning, including: i) a new internal pipeline implementation based on KServe that provides greater flexibility of reuse in multiple inference scenarios, ii) the support for eXplainable AI (XAI) to explain how models learn from features, iii) a learn monitoring system for detecting data drift. ML applications developed for the IoT-NGIN Living Lab (LL) use cases are also reported.
- A Reinforcement Learning (RL) based implementation for system optimization, deployed within MLaaS, and tailored to optimize electric grids for the Smart Energy LL.
- A common entry point API that harmonizes the access of third parties to the PPFL platform, common for all the FL frameworks supported.
- A Model Sharing framework implementation, that enables an integrity-guarantee batch training of ML models, and their registrawithin the MLaaS model storage for futher sharing and reused, as well as their conversion into ONNX intermeditate model for inference in any environment compatible with ONNX runtime.

These MLOps platforms, frameworks and services are released as open source in the IoT-NGIN project's public GitLab repository <https://gitlab.com/h2020-iot-ngin>, contributing to the IoT community.

Planned future work includes extending the application of these MLOps plaforms and services in additional IoT-NGIN LL use cases, which will be reported in D6.3 [1] and D7.4 [2].

# 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 them in diverse aspects. As an example, metrics gathered from sensors can be immediately used to trigger an alert on a detected malfunctioning situation. However, the overpopulation of devices and sensors is generating an increasing volume of information that companies need to face, a challenge identified by the Big Data 5 V's [3]. As a result, a simple system service could not be capable anymore of coping 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 Graphics Processing Units (GPUs) and Tensor Processing Units (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 an ML platform that can provide all the necessary services to process data, train, share and deploy ML models. Implementing and maintaining such an ML platform is a complex, time-consuming and costly endeavor, requiring expertise most of companies lack of. 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 modelling and delivery procedures into their development and Continuous Integration/Continuous Delivery (CI/CD) environments. By using MLaaS, data scientists can procure and preprocess 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.

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 Infrastructure as Code (IaC) [4] and ArgoCD [5] to automate the platform building and delivery.

The design and implementation of the IoT-NGIN MLaaS platform was described in a series of previous reports, namely D3.1 [6], D3.2 [7], D3.3 [8]. In those reports, a number of additional services and frameworks, extending the MLaaS platform with additional ML capabilities, were also reported (both functional and technically), including the **online learning service**, the **reinforcement learning (RL) - based optimization service**, and the **privacy-preserving federated learning framework**. Another service framework, the **polyglot model sharing**

**service framework** was also reported in the specification (D3.1) but its implementation was not described in detail until this document.

The present document is a technical report, entitled “ML models sharing and Transfer learning implementation”, the fourth and last deliverable of the WP3 series (D3.4) and reports the results of the activities of Task 3.2 “*Deep learning/reinforcement learning techniques to enhance training processes*”, Task 3.3 “*Confidentiality-preserving federated ML models*” and T3.4 “*Machine Learning Model Sharing*”. The results of the activities of Task 3.1 “*Big Data and ML framework architecture*” are not reported in this document as it ended by the time D3.3 was released, where its results were reported in detail.

Moreover, the activities reported in D3.4 align with the objectives of WP3:

- Define, design and develop a big data management and privacy preserving federated ML layer, based on Big Data Value (BDV) Strategic Research and Innovation Agenda (SRIA)4.0 [9], to train and share ML models.
- Implement innovative deep learning techniques to enhance training processes with inline adaptive self-learning that will improve the resulting machine learning models automatically.

This document complements D3.3 with updates and new features implemented for the *online learning service*, the *reinforcement learning (RL) - based optimization service*, and the *privacy-preserving federated learning framework*. For these services, it focuses on describing these last updates and new features, and references D3.3 for other functional and technical aspects already reported, avoiding duplicating D3.3 content, unless needed for the sake of better understanding.

## 1.1 Intended Audience

The intended audience includes data scientists and ML engineers which may find this report inspiring for their research and development efforts as well as ML service providers who aim to adopt the IoT-NGIN MLOps paradigm and the enhancements towards reinforcement, online and federated learning and integrity-guarantee model sharing. The document provides technical specifications, as well as practical guidance allowing interested audience to test and adopt the IoT-NGIN developments. The document describes the design and implementation of the online learning and reinforcement learning techniques, the privacy-preserving federated learning API and the polyglot model sharing framework.

Moreover, the document might be of interest to business users who wish to adopt different technical approaches for machine learning, privacy-preserving federated learning and sharing into their processes. The document provides insights for exploiting the IoT-NGIN tools in the context of the Living Lab (LL) use cases, which could be indicative for other use cases as well.

Finally, this report is useful internally, for the project partners which develop ML related solutions, perform integration and validation activities, as well as for the Living Labs. Useful feedback could be also received from the Advisory Board, including both technical and impact creation comments.

## 1.2 Relations to other activities

The activities of WP3 are strongly linked to other IoT-NGIN activities, as indicated its work package structure rendered in Figure 1.

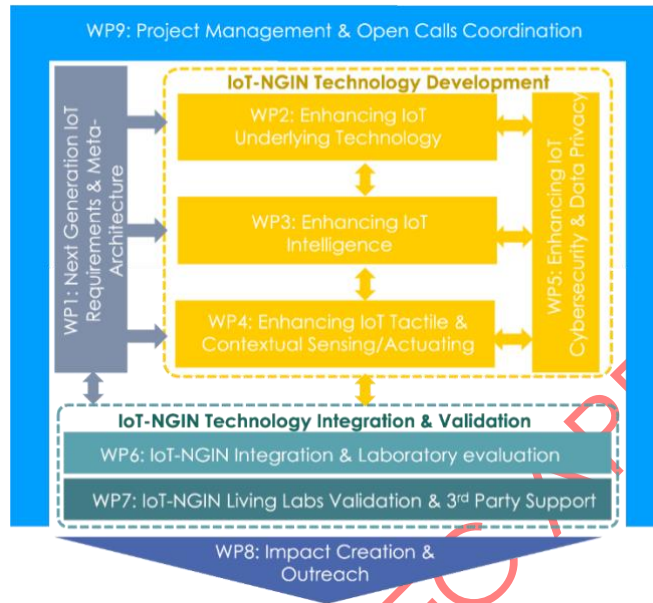


Figure 1 - Work packages structure

Table 1 describes the relation between WP3 tasks and the other IoT-NGIN activities.

Table 1 - Relation of WP3 activities with other WPs and tasks

WP	Relation to WP3 and D3.3
WP1	The definition of the Living Labs' use cases (UCs) inspires D3.4 for defining the design and implementation of the online and reinforcement learning services, the polyglot model sharing framework, and the federated learning framework, and for providing tailored implementations in the context of these use cases.
WP2	Models trained with the online learning service have been integrated into the Secure Edge Cloud framework for IoT micro-services for secure execution of ML models (see D2.3 [10]).
WP4	The object detection models of the IoT Device Discovery module have been served within the MLaaS platform as instances of the online learning service.
WP5	WP5 develops cybersecurity tools for securing FL operation. These tools can work together with the FL modules of WP3 to ensure protection



D3.4 – ML models sharing and transfer learning implementation

	<p>against data and model poisoning attacks. Moreover, the WP5 tools are based on ML functions, which may be trained and served via the MLaaS platform.</p>
WP6	<p>WP3 components have been integrated with the rest of project's technologies and frameworks in WP6, while the MLaaS platform and tools can be useful in the development of application logic for the use cases or the Open Calls.</p>
WP7	<p>WP3 components have been implemented and used in several living labs and use cases.</p> <p>WP3 supports 3<sup>rd</sup> parties by offering ML models via MLaaS model training and sharing.</p>
WP8	<p>WP3 provides notable outcomes and results for supporting impact creation activities. Moreover, it considers feedback (e.g., from the market analysis and business modelling tasks) which could be relevant for updating or enhancing the WP3 design and development.</p>

DRAFT - PENDING EC APPROVAL

## 1.3 Document overview

The remainder of the document is organized as follows.

Section 2 introduces the updates and new features implemented during this reporting period for the IoT-NGIN online and reinforcement learning services, including their functional specification, technical design and implementation, as well as the specific implementation for the LL use cases where these techniques have been adopted.

Section 3 introduces the updates and new features implemented for the Privacy-Preserving Federated Learning framework, focusing on the common access API for deploying FL tasks across the PPFL framework.

Section 4 describes the functional specification, technical design and implementation of the Polyglot Model Sharing framework, and its in-lab validation with an example use case.

Section 5 provides installation and user guidance, of above services and frameworks, which are published as open source.

Section 6 draws conclusions and summarizes next steps for future work.

Annex in Section 7 describes the main components of the Polyglot Model Sharing Framework, their REST APIs and the main processes they implement.

DRAFT - PENDING EC APPROVAL

## 2 IoT-NGIN Machine Learning

### 2.1 Online Learning

This section describes the updated and the new features implemented for the Online Learning (OL) Service, as part of the MLaaS platform, introduced in D3.3. For the sake of facilitating the reader understand these features and their implementation, this section may borrow some content from D3.3 when needed, but in most of situations, it will refer to it, to avoid extending the document unnecessarily.

#### 2.1.1 Description

D3.3 described the first version of the Online Learning serv. The first version focused on providing a service capable of dynamically training ML models for IoT applications and also performing inferences on demand.

Dynamic training, also known as Online Learning, concerns the training of ML models when new data is available. Thus, the model is trained continuously. This paradigm acquires great importance in the context of IoT due to the large number of sensors or devices that can be present on common scenarios, and the large amount of information captured by them dynamically.

In addition, the OL service supports both API REST requests and streaming data since most IoT devices generate communication flows in real time.

The OL service version reported in this document maintains these features and offers an improved implementation that offers i) greater flexibility and ii) new features.

Greater flexibility is achieved with the incorporation of a pre/post processing module that mediates between the service clients and the model. This modification allows to deploy the pre/post processor and the OL service as two separate microservices, thus letting them to scale independently.

Regarding the new features, a module has been implemented for XAI (eXplainable Artificial Intelligence), that attempts to explain the predictions made to answer the question: Why has the model made this prediction? XAI is a set of methods and processes that help to comprehend and trust the prediction driven by the ML model. Moreover, it helps to characterize the model performance by providing the impact of the input data for a given prediction, adding transparency to the prediction and capacity for model bias detection. This module is optional, that is, the OL service does not come with an explainer deployed by default since it is a use case dependent module.

Finally, a complementary service to OL, called OL monitoring, has been implemented. This service monitors, in real time, the performance of the model and the input data sent to the OL service, both for training and inference, so that it can detect a degradation on the model performance and if data drift phenomenon has occurred.

Data drift is defined as the variation between the data in the training phase and in the deployment phase. This phenomenon can be caused by different factors, the most common are that the training data did not include the entire population of the data or the distribution of the data varies over time.

## 2.1.2 Technical design

The overall architecture design of the OL framework is shown in Figure 2. The internal architecture of the OL service in Figure 3.

The OL service is deployed into MLaaS using Kserve [11] through Kubeflow. [12]. Kserve allows to deploy 3 types of components: Predictor, Transformer and Explainer. Each of these components expose a REST API as a HTTP service<sup>1</sup>.

The Predictor<sup>2</sup> is the ML model hosting service. It is responsible of performing dynamic training and providing inferences on demand. This component is designed to be reusable across specialize OL services for different use cases. It loads the model from MLaaS Model Storage (MinIO [13]) to make a prediction or updates it back when the model has been improved after training. A new strategy has been implemented so the model is only updated and stored back to MinIO when the training losses have been lowered.

The Transformer is a service, located between the client and the Predictor, that is responsible for preparing the data, received in JSON format, so that the Predictor can train the model or make a prediction. In addition, the Transformer also performs the post-processing of the prediction returned by the Predictor. In this way, the inference obtained is processed so that the client gets the prediction in a more appropriate format. Thus, the Transformer that processes the input data is use-case specific. That is, each OL service will encompass a different Transformer.

As the Transformer is the module that receives the input data, it must also support receiving data coming from MQTT [14] and Kafka services. To cover this requirement, Camel-K [15] is used (as explained in D3.3).

The Explainer service aims to provide justification to the predictions. This service is only accessible through HTTP and extracts the most significant features of the input data. In this way, it is possible to understand which part of the input data has had more influence in the final prediction and also the way by which the model has obtained the prediction. This service is also use-case specific since the XAI the algorithm depends on the input data type and the framework used to implement the ML model.

The monitoring service consists of a HTTP endpoint deployed using FastAPI framework [16], a Prometheus engine [17] and a Grafana web tool [18]. This service monitors the input data and the model performance in real time by using the Evidently [19] package. To do this, the Predictor sends the input data and the inferences made to this service, and then, by using Evidently, statistical hypothesis tests to detect data drift are carried out, and evaluation metrics of the model in production are computed.

To detect data drift, Evidently requires a reference dataset. This dataset is collected by the model developer following use-case specific methods. For instance, it could be the dataset the ML model developer uses to perform some tests during initial model development to validate the model architecture.

The results obtained by Evidently are collected in Prometheus, mediated by an exporter that Evidently registers in Prometheus, which scrapes the data from the Evidently monitoring service. In Grafana a data source pointing to Prometheus is configured. Thus, Grafana can

---

<sup>1</sup> The REST API exposed by the Predictor is hidden so only the Transformer gets access

<sup>2</sup> Despite its name, the Predictor is used to either i) train the model online, ii) process an inference, through different APIs

access these results for visualization, in dashboards, where users can analyze the performance graphs. The dashboard provides different evaluation metrics: mean error<sup>3</sup>, mean absolute error <sup>4</sup>and mean absolute percentage error<sup>5</sup>. It also shows the history of these metrics as a time series. Lastly, it provides the bias of the predictions both in the training and in the production phase. This bias indicates whether the model tends to overestimate or underestimate the value with its predictions.

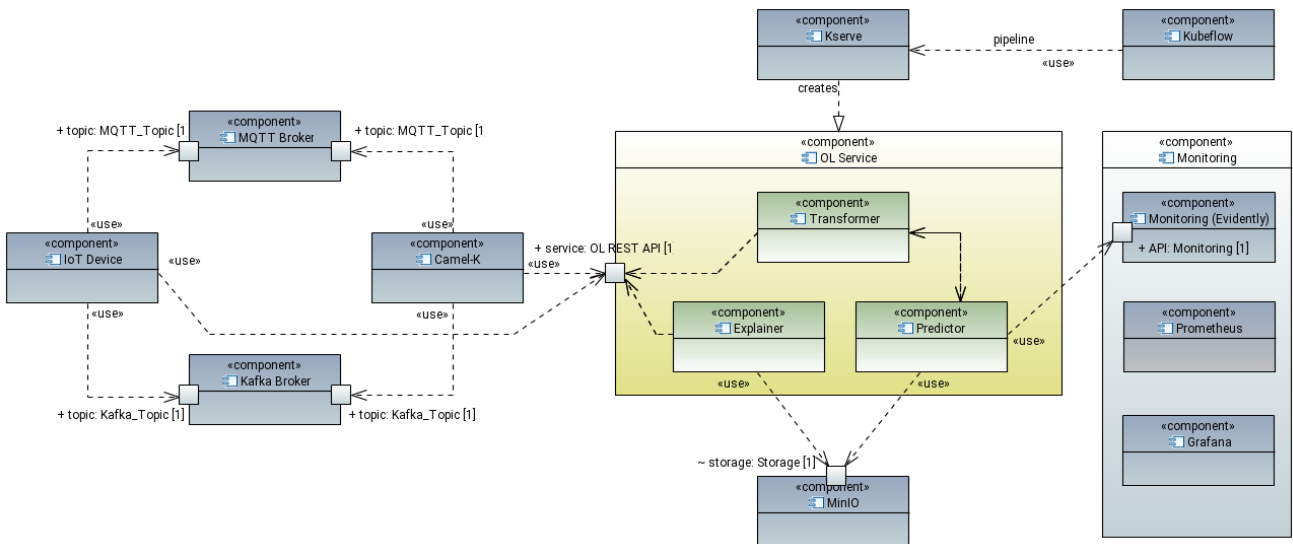


Figure 2 - Overall architecture of the OL service

New OL modules have been implemented or updated from D3.3, using Python since it offers a large ecosystem of specialized libraries for AI. These modules are listed below:

- **Create Kserve Service (Predictor).** 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 the functions needed to prepare the data for the ML model. It also performs the post-processing stage, so the prediction is processed to be provided in more convenient format for the requester. The implementation of this module is use-case specific.
- **Explainer.** It receives the preprocessed 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 Predictor to the backend module. It is responsible for choosing the correct backend and transmitting the model update or the prediction requests. It also implements the model saving strategy.

<sup>3</sup> ME:  $ME = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)$ , where  $N$ ,  $y_i$  and  $\hat{y}_i$  are the number of predictions, actual and forecast value respectively.

<sup>4</sup> MAE:  $MAE = \frac{1}{N} \sum_{i=1}^N |y_i - \hat{y}_i|$ , where  $N$ ,  $y_i$  and  $\hat{y}_i$  are the number of predictions, actual and forecast value respectively.

<sup>5</sup> MAPE:  $MAPE = \frac{1}{N} \sum_{i=1}^N \left| \frac{y_i - \hat{y}_i}{y_i} \right|$ , where  $N$ ,  $y_i$  and  $\hat{y}_i$  are the number of predictions, actual and forecast value respectively.

D3.4 – ML models sharing and transfer learning implementation

- **Streaming connector.** This module provides tools to support real-time protocols. This module is not being used in current deployment because Kserve only supports HTTP connections, but it is implemented in case future versions of Kserve supports streaming data. The main libraries that have been used for its implementation and its testing are *Kafka* [20] and *Paho-MQTT* [21].
- **MiniIO Connector.** Provides the required tools to download and upload the ML models. This version stores the trained ML model in MinIO storage in case the model performance gain overpass a given threshold during the training. This module is based on the *MinIO* library.
- **Backend.** Modules responsible for including required functions to perform ML model updates or predictions for each ML framework. Current version supports the following frameworks: *Sklearn*, *Vowpal Wabbit*, *TensorFlow* and *Pytorch*.
- **OL Monitoring.** It receives input data and model predictions to compute different model performance metrics and to carry out statistical hypothesis tests to detect data drift using Evidently. These results are scraped by the Prometheus agent periodically, to be displayed in the Grafana dashboards.

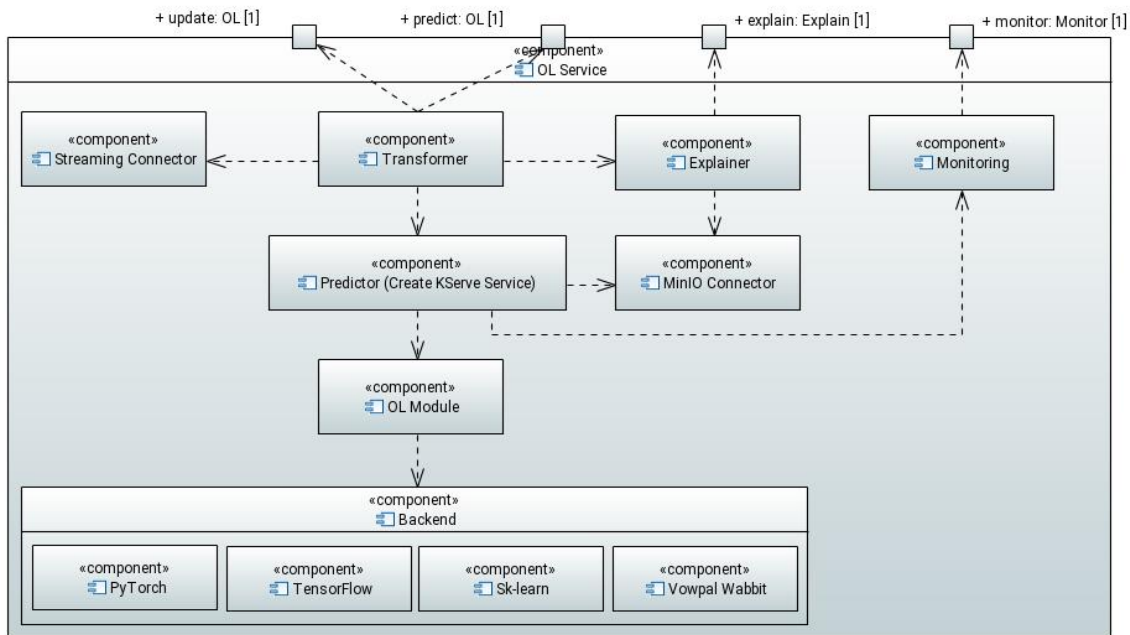


Figure 3 - Technical architecture of the OL service

As described in previous deliverable (D3.3), additional steps are required for deploying the recent releases of the OL services. The whole process is described in section 5.1. The main steps are:

- 1- **OL Service Adaptation:** it is the initial step to configure the OL service. Three different modules are set and implemented:
  - a. **Predictor:** it sets different parameters such as the MinIO host and the buckets where the ML models are stored, the backend (framework which was used to implement the model) to use in order to perform the model update or the prediction.
  - b. **Transformer:** it Implements the use-case specific pre/post processing methods for the input data.

- c. Explainer (Optional): it implements the XAI algorithm to provide explanations about the inferences made.
- 2- Create the OL Docker images:** once all Kserve components are configured, they are enclosed in Docker images, and registered in the registry from where Kubeflow retrieves them to build the deployment pipeline. There is a docker image per module.
- 3- Define the Kserve YAML manifest:** This manifest defines the configuration of the OL service during its deployment. It defines the name of the inference service, the number of replicas, the CPU limits and the Docker images to use, among other configurations.
- 4- Create the Kubeflow pipeline:** This step creates a Kubeflow pipeline that instantiates the Kserve YAML manifest when executed.
- 5- Run the Kubeflow pipeline:** This step deploys the OL service as an HTTP inference service.
- 6- Define the Camel-k binding:** Camel-K binding consists of a YAML file that defines the broker and topics where data is being dumped and the prediction service to where the data is transferred to.
- 7- OL monitoring adaptation:** This step configures the type of data to be monitored and the type of task (regression or classification) to be performed by the OL service.
- 8- Create the OL monitoring Docker image:** Once the OL monitoring is configured, it must be encapsulated in a Docker image and uploaded into the Docker registry.
- 9- Create the monitoring YAML manifest:** it configures some aspects of the monitoring service, before deploying, such as service name.

The source code of the implementation of the Online Learning service is available at the [IoT-NGIN GitLab repository](#) [22].

### 2.1.3 Implementation for IoT-NGIN LLs

This section describes the application of the Online Learning services for the training and inference in some of the IoT-NGIN LL use cases, as a way to evaluate these services. The following describes a joint work between WP3 and WP6, included in this document for the sake of completeness. Incoming results of the application of these services to the LL use cases will be also reported in D6.3 [1].

#### 2.1.3.1 Smart Energy

D3.3 described the Smart Energy scenario and the procedure to deploy two power generation forecasting services for UC9 and UC10. These services have been improved with the inclusion of the explainer and also with the adoption of the monitoring service. Implemented services for UC9 and UC10 are shown at

Table 2.

Table 2 - Smart Energy LL MQTT topics for forecasting services

Service	Description	MQTT Meter/Topic	
		UC9	UC10



D3.4 – ML models sharing and transfer learning implementation

Power Consumption Forecasting	Use power data over the time to predict the consumption in next 24-36 hours.	Smart Meter/ BBB60XX	Power Quality Analyzer/ W4
Power Generation Forecasting	Use power data over the time to predict the generation in next 24-36 hours.	PMU/ 3640f24ba43a 423188979372 bae6277a	Power Quality Analyzer/ W6

To include the explainer component in both services, a XAI method that provides the explanations to the predictions is required. To implement this method, the Captum library [23] is used. It is an open-source Python library specialized in XAI for models implemented with Pytorch [24].

Captum allows to use different XAI methods to compute the importance of each input feature in the model prediction. Different methods have been tested, being DeepLIFT (Deep Learning Important FeaTures) [25] the one that provided the best explanations. This method belongs to the XAI backpropagation-based approach. These approach highlights the input features that are easily predictable from the output.

DeepLIFT attempts to decompose the output prediction of a neural network on a specific input by backpropagating the contribution of each neuron to each input feature. It compares the activation of the neurons with its reference activation (a default or neutral input) and assigns contribution scores according to the difference. DeepLIFT has the capacity of detecting both positive and negative contributions, so the explanations distinguish features with positive impact on the prediction from those with a negative impact.

To verify that DeepLIFT provides acceptable explanations, a small evaluation has been performed for both Smart Energy forecasting services. An input vector with 36 past power measurements (the forecasting model requires an input dataset with 36 features, see D3.3) is used with the DeepLIFT method to derive which features have the highest impact on the prediction. The impact of each feature is represented in Figure 4 and Figure 5 for UC9 and UC10, respectively. Features with a positive contribution are render in green, features with no major impact in yellow and the ones with negative contribution in red. The conclusion that can be extracted from DeepLIFT is that the features with highest impact are the last ones in the input tensor (represented in the x-axis in figures), that is, the most recent ones, as expected.



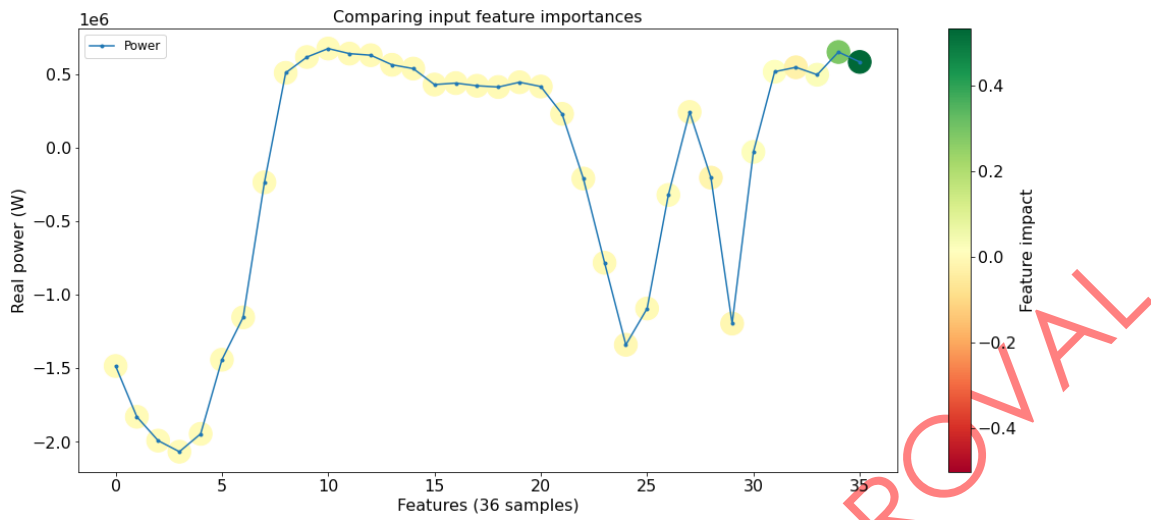


Figure 4 - DeepLIFT results for power generation forecasting in UC9

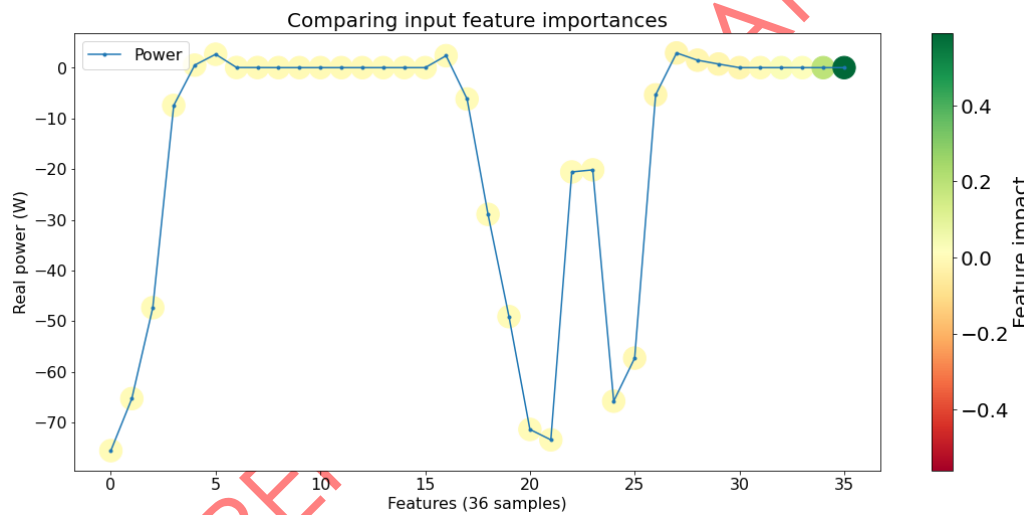


Figure 5 - DeepLIFT results for power generation forecasting in UC10

Once DeepLIFT explanations have been verified, the forecasting services can be updated to include the explainer component. To do so, some steps described in the previous section are followed. In particular, the creation of the docker image for the explainer and the addition of the explainer section in the manifest YAML. Then, the Kubeflow should be defined and executed.

The procedure to deploy the monitoring service for the UC9 power generation forecasting service is detailed below.

1. The first step is to configure the monitoring service: the type of data to be monitored and the type of ML task (Classification or Regression) being performed. In addition, a reference dataset must be added as a baseline to compare the new data coming in with. Once this service and the reference dataset have been configured, its Docker image is created and uploaded into the Docker registry. Afterwards, the YAML manifest is applied, configuring the name of the service.

2. Next, the OL service is configured, in particular the predictor component, to enable monitoring. A boolean environment configuration flag enables/disables monitoring, and another one defines the endpoint of the monitoring REST API.

Grafana dashboards are shown in Figure 6 and Figure 7. Figure 6 shows the performance monitoring of the model. As explained in section 2.1.2, this monitoring provides ME, MAE and MAPE evaluation metrics at the present time and over time. It also provides information about the bias of the model's predictions. In this way, it is possible to know if the model tends to overestimate or underestimate. Figure 7 shows the monitoring of the data drift. This dashboard provides information about the data drift in different ways. The *Dataset Drift* panel shows whether or not the data drift has been detected over time. The *share of drifted features* panel shows the percentage of features that suffer of data drift. Since this implementation only has a single variable (the generated power), when data drift is detected, 100% of the dataset presents data drift.

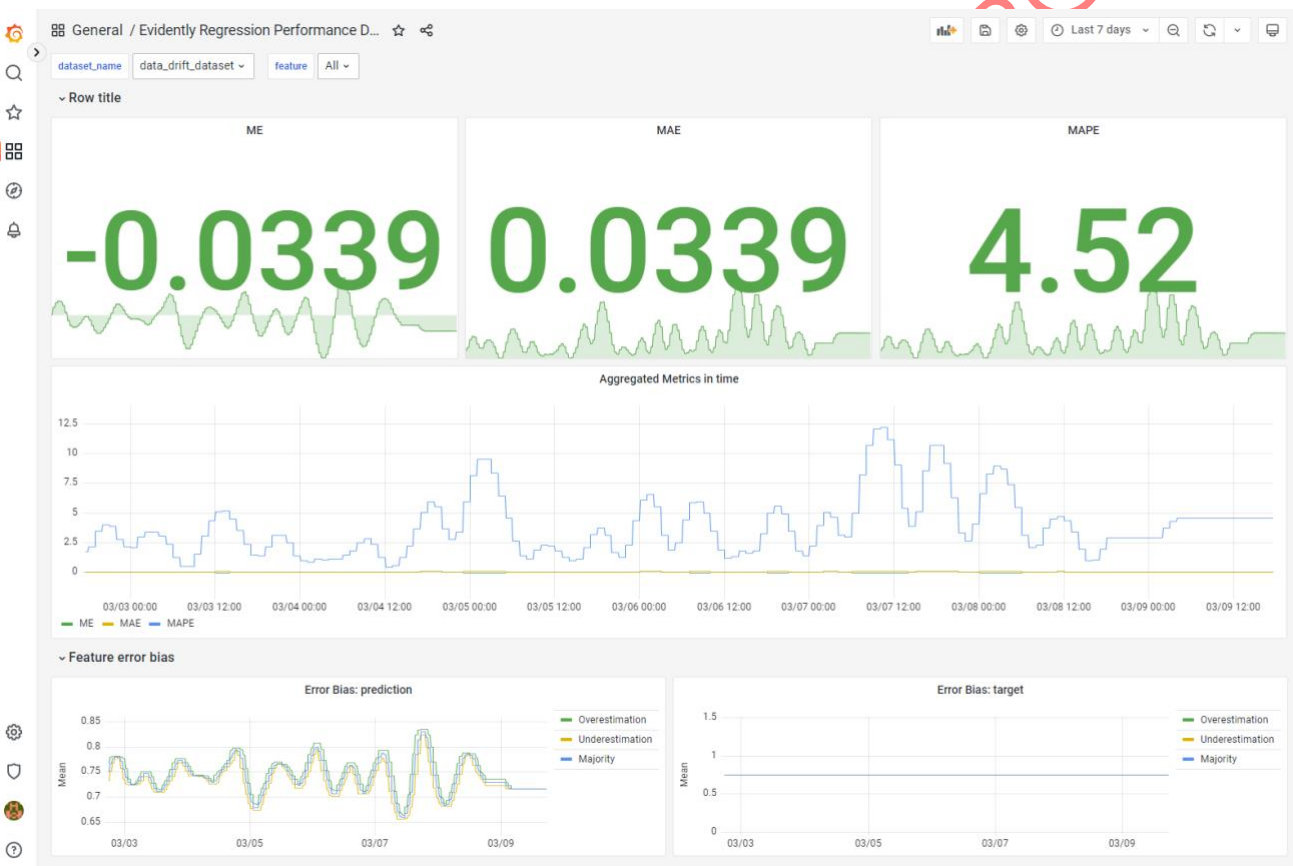


Figure 6 - Model performance monitoring

DRY

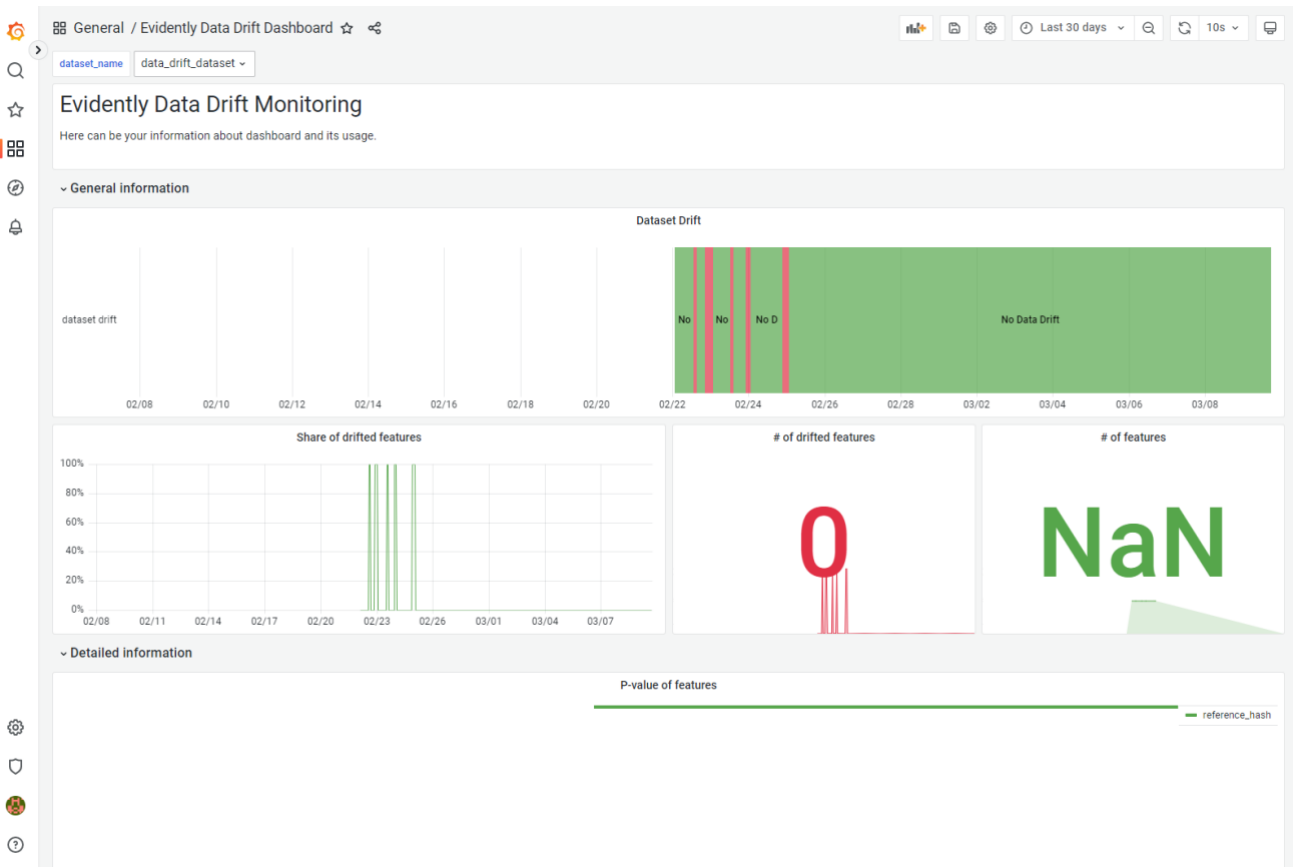


Figure 7 - Data drift monitoring

The last service that has been implemented for the Smart Energy LL UC10 is the forecasting of the consumed power. The scenario remains the same as described in D3.3. The electrical network is publishing messages with the consumed power in an MQTT broker, and the OL service has to infer what the power consumption will be like 24 hours later by using the previous 36 power samples. The pre-processing performed in the same way.

The implementation of the DL model leverages transfer learning technique. Transfer learning is an ML technique that retrains a model, originally designed and trained for a specific purpose, to be reused for another inference objective. The rationale behind this technique is to take advantage of the "knowledge" acquired by the model in the first learning objective to learn more quickly on the second one. From the implementation point of view, Transfer Learning consists of freezing the input layers of the model that has already been trained. That is, the weights of these layers are not adjusted in the training phase with the dataset for the second learning purpose.

Figure 8 shows the architecture of the model. The description of this architecture is described in section 3.1.3 of D3.3 [8]. The layer to be frozen is the first Gated Recurrent Unit (GRU) layer. Therefore, the weights of the rest of the layers will be retrained.

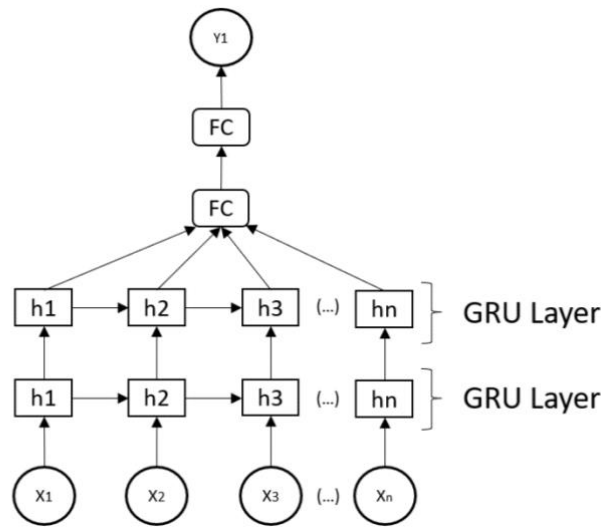


Figure 8 - DL architecture

Before deploying the OL service with this architecture, a little experimentation has been conducted to verify that the technique is promising. For this, a dataset of consumed power has been collected for approximately 15 days and a small pre-analysis of this dataset has been carried out. Alike the UC10 power generation forecasting service, this dataset presents a 24-hour seasonality. Table 3 shows the chosen hyperparameters for training the predictive model.

Table 3 - Transfer Learning hyper-parameters

Hyper-parameter	Value
Epochs	50
Learning rate	0.001
Optimizer	Adam [26]
Loss function	Mean Squared Error
Batch size	128

Training results are shown hereafter in the same way as in D3.3. Figure 9 shows the actual power data (orange line), inferences performed by the ML model (blue points) and the forecasting intervals with a 90% of confidence (blue area). 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. Normality hypothesis tests have been carried out to assume that errors come from a gaussian distribution. The tests are Shapiro-Wilk, Anderson-Darling and D'Agostino-Pearson [27]. 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 tests return 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. Therefore, it is correct to assume that errors come from gaussian distribution, as shown at Table 4.

Table 4 - Normality test results (p-values)

Normality Test	Power consumption forecasting model
Shapiro-Wilk	0.56
Anderson-Darling	0.61
Agostino-Pearson	0.15

The model can learn seasonality in data. The MSE obtained using the validation subset is 0.012.

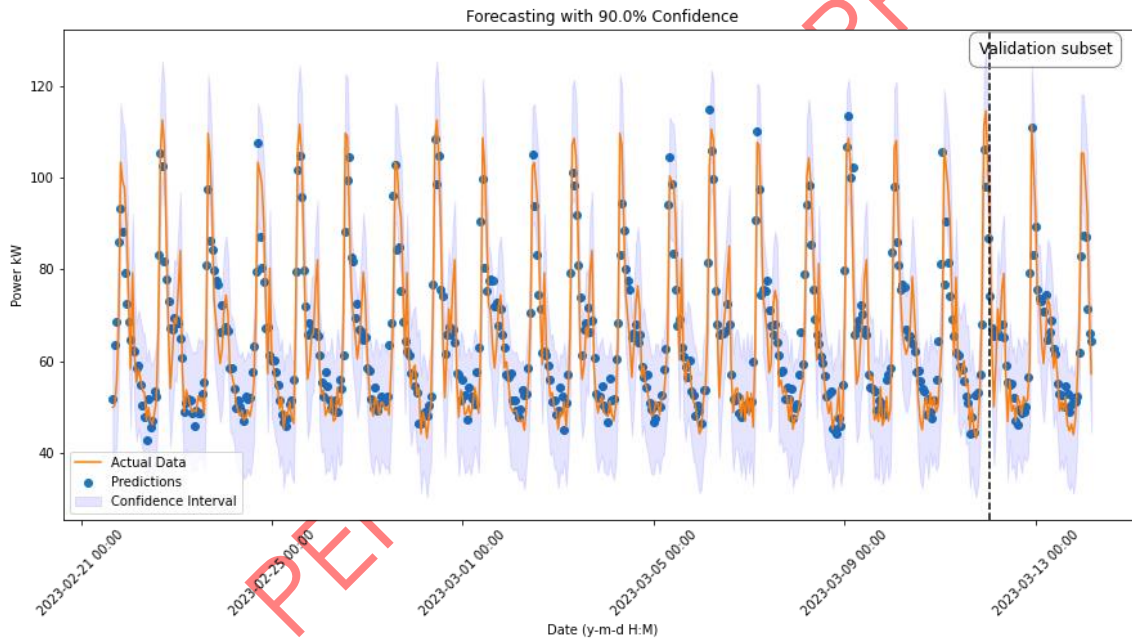


Figure 9 - Training results for power consumption forecasting of UC10

The model is stored in MinIO so that the corresponding OL service can load it and perform updates and inferences.

The deployment procedure for the OL service corresponding to this energy demand forecasting UC is the same as described in section 2.1.2.

### 2.1.3.2 Smart Agriculture

The Smart Agriculture LL offers different UCs that require ML modeling for different purposes, as described in D3.1. In the previous section, we have addressed some Smart Energy LL UCs that require dynamic training for ML models due to the nature of the source training datasets.

On the contrary, other LL UCs require different approaches for ML training and inference in MLaaS.

This section describes the deployment process of a service that hosts an AI-based object detection model, capable of detecting Synelixis' SynField IoT devices in images taken by mobile phones. This model is the first step towards indexing of the SynField devices on the overall network and eventually the Smart Agriculture LL, as explained in D4.3 [28].

The service is deployed using the Kserve library, by adopting the same procedure described in section 2.1.2. This library only exposes a REST API endpoint that attends requests enclosing input data in JSON format, but it does not support images in its payload. As the object detection service requires images as input data, it is necessary to encode the images in base64 before they are sent to the service. Moreover, the model needs to be registered in the MLaaS Model Storage (MinIO). The object detection service retrieves the model from MinIO and then processes inferences on demand. The service receives a base64 encoded image. At the pre-processing stage, the Transformer component prepares the data for the model. The preprocessing consists of decoding the image and converting it into a 3-dimensional tensor. This component makes use of the *Base64* *CV2* and *Numpy* libraries. Figure 10 depicts the process.



Figure 10 - Preprocessing stage object detection model

Once the 3D tensor is ready, the Transformer component sends it to the Predictor component. As explained above, this component loads the model from MinIO, receives the tensor as input, and processes the prediction. It sends the prediction to the Transformer, which postprocesses it to include the detected object frame into the original image, base64-encodes it, and sends it back in JSON format.

The procedure described in section 2.1.2 has been applied to deploy the service in the MLaaS platform, including the creation of a Docker image that encapsulates the Transformer and its registration into the Docker registry, the creation of the YAML manifests for configuring the Transformer and Predictor components, and finally the definition and execution of the Kubeflow pipeline.

## 2.2 Reinforcement Learning

This section describes the functional specification, technical design and implementation and its application to the IoT-NGIN LL UCs of a Reinforcement Learning (RL) based optimization engine. The conceptual description of this RL-based optimizer for the Smart Energy LL UCs was introduced in D3.3. RL-based optimization engines are very use-case specific, although different standard RL-optimization techniques reported in the scientific literature can be applied. The following subsections provide an updated description of the RL-based



implementation for the UC9 Electric Grid optimization. A detailed evaluation of the performance of this optimizer will be reported in the deliverable D6.3 [1].

## 2.2.1 Description

Reinforcement learning is an ML paradigm that consists of an agent that learns how to optimize a given system behavior from its interaction with the system environment. In each interaction with the environment, the agent takes an action, based on its own state in order to maximize a reward. This paradigm was described in D3.3. The following describes the RL-based optimization model design and technical implementation, which is use-case specific, for the UC9 Grid Energy Optimization use case described in D3.3. The technical design and implementation of this optimization model is based on standard RL algorithms and implementation, and therefore reusable for application to other cases, such as the UC10 described in D3.3. Therefore, section 2.2.2 provides a reusable technical design and implementation, and section 2.2.3 provides the details to concretize the implementation to the UC9 optimizer, by providing details on the action set, state set and rewards.

## 2.2.2 Technical design

In D3.3, the high-level architecture of the RL-based optimization service was presented. Figure 11 shows the description of the architecture at a high level.

Alike OL, the RL service is deployed in MLaaS with Kserve, following the same procedure described in section 2.1.2. An interaction with the environment, through publish/subscribe protocols, may be required, so the possibility of creating a camel-k binding is conceived to support this functionality.

The RL module is responsible for training the AI model with RL algorithms. For this purpose, the Tensorforce framework [29] has been selected. This open-source deep reinforcement learning library, implemented on *Tensorflow* [30], has been selected since it adopts a set of high-level design choices such as the modular component-based design, the separation between RL algorithm and application that make it suitable for our technical requirements.

Tensorforce requires to define two objects, namely i) the agent and ii) the environment. The agent is the entity responsible for taking an action when the environment is in a certain state, aiming to maximize the reward. The environment is the entity that returns the reward and new state after applying the action. Then, the agent learns the actions that provide the highest reward.

Tensorforce offers model free algorithms, which do not learn a model of the transition functions of the environment to make predictions of future states and rewards, from both families: Q-Learning and Policy Optimization. Q-Learning algorithms aim to learn optimal policy based on state-action value pair while Policy Optimization ones learn the optimal policy by optimizing the policy distribution.

Within the Q-learning group, Tensorforce provides the Deep Q-Network (DQN), Double DQN, and Dueling DQN algorithms. While from the Policy Optimization group, it offers the Policy Gradient, Proximal Policy Optimization and Actor-Critic algorithms.

D3.4 – ML models sharing and transfer learning implementation

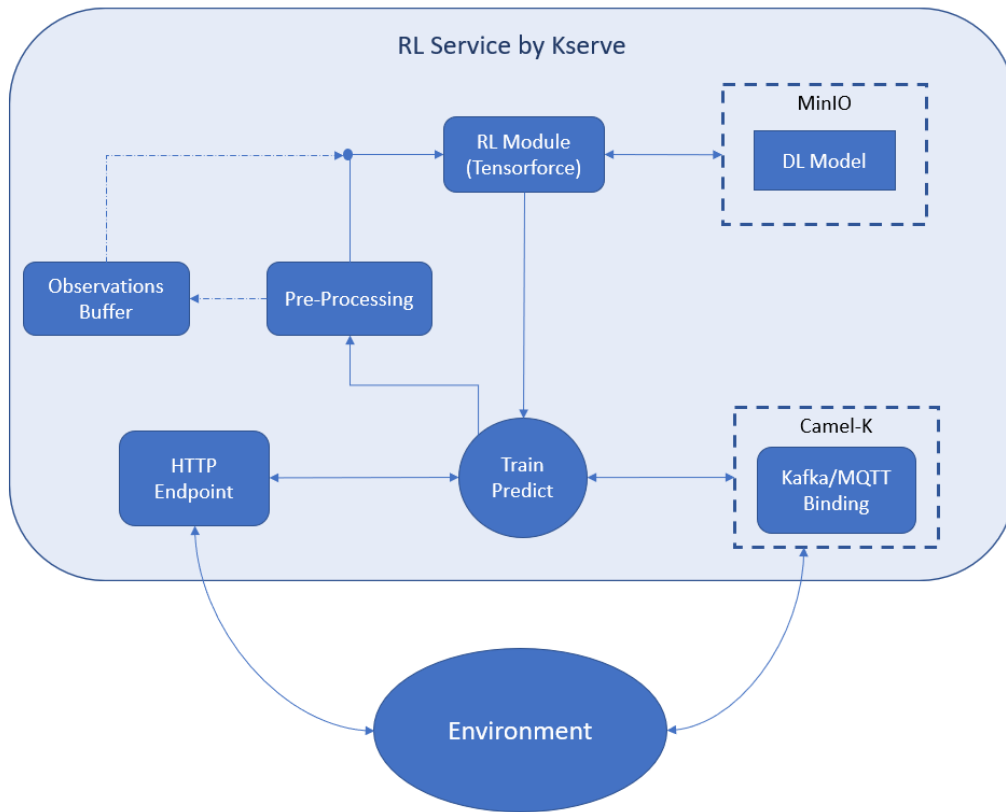


Figure 11 - RL service high level scheme

The environment the agent interacts with can be a real one or simulated. In any case, the Tensorforce entity transmits the action taken by the agent to the environment and waits for it to return a new state and reward. This process is shown in Figure 12.

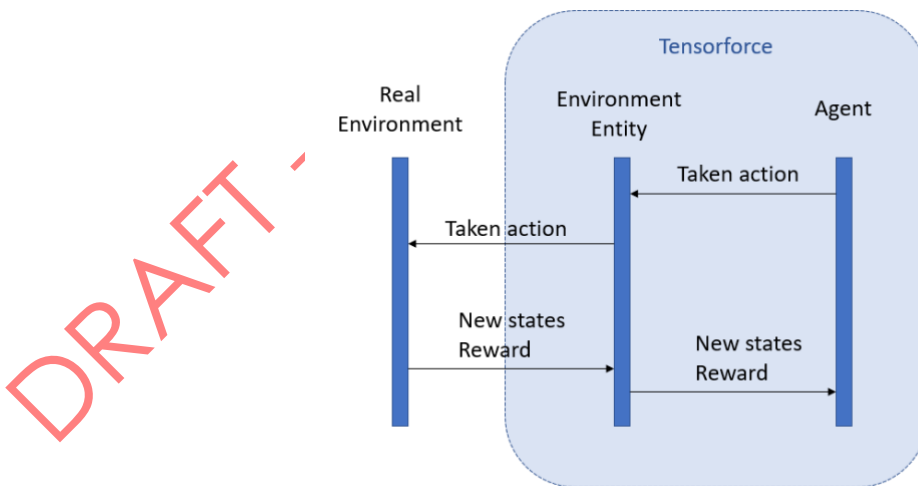


Figure 12 - Sequence diagram of interaction between Tensorforce and environment



## 2.2.3 Implementation for IoT-NGIN LLS

In D3.3, the need to implement RL-based optimization model was presented to cover the use cases UC9 and UC10, namely “Move from Reacting to Acting in Smart Grid Monitoring” and “Control, and Driver-friendly dispatchable EV charging”, respectively, with the purpose of optimizing and controlling the electrical network.

This section describes an implementation of a RL-based optimization for the UC9, based on the Tensorflow framework.

### Environment

As we cannot get access to the real UC9 electric grid (EG) to interact with, a EG simulator has been implemented by the EG owners, using the pandapower framework [32]. This framework uses the Flow PYPOWER power solver to create a calculation network program with the aim of automating and optimizing power systems. As a result, the simulator describes the same environment state as the real EG network when connected with the same sources of power generation and consumption.

The operation of the simulator is as follows: once the electrical network is specified in pandapower, the simulator reads the power loads demanded by all the consumer groups connected to the network and the power generated along a day. The data has a resolution of 15 minutes, so there are 96 values of domestic and industrial consumers' demand and generated power.

Then, the simulator introduces the data into the corresponding loads on the electric grid and performs a simulation. The simulation results consist of the different parameters of the grid state. those that are used as a reward are referenced in the following reward section.

### States

The optimizer acts on the customers' energy demand, that is, it modulates the distribution of energy demand throughout the day, so it is necessary to know what the energy demand state is they are in or equivalently, which is the distribution of current energy demand.

Two types of customers can be distinguished: domestic and industrial. There are 13 loads (client groups) of each. To try to better understand what the data looks like, a pre-analysis of dataset collected over a year has been carried out.

Figure 13 shows the average distribution of each domestic load for one day. It can be seen that all distributions are very similar. In addition, the load "Load\_D\_486" presents a much higher energy demand than the rest and accounts for 38.5% of the total energy demand. Also considering the loads "Load\_D\_491" and "Load\_D\_493", there is around 70% of the power demanded.

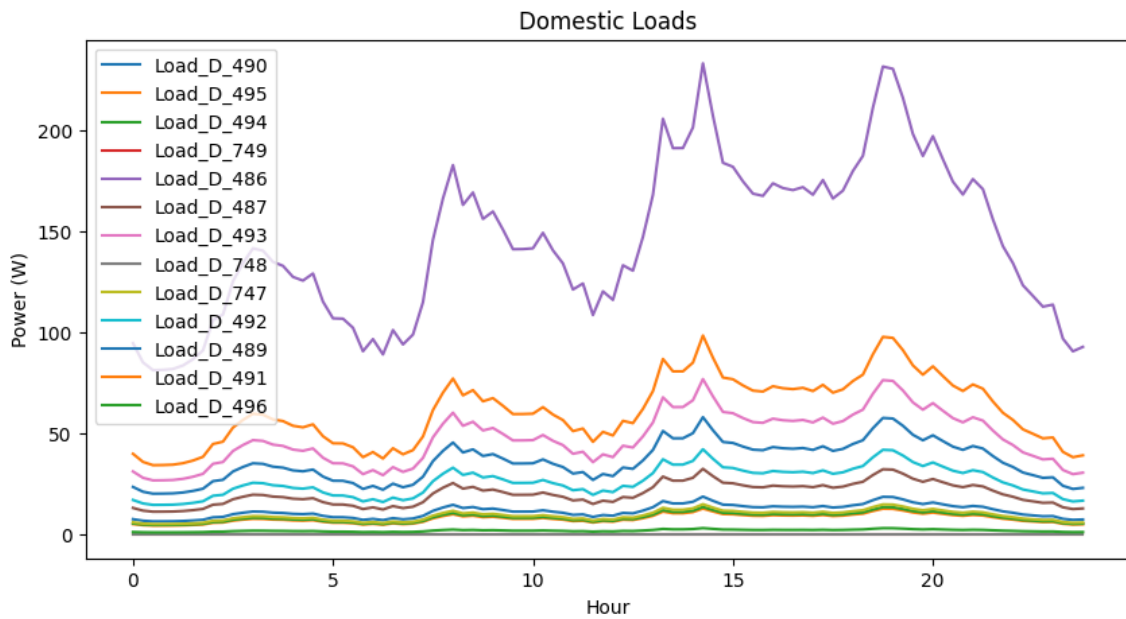


Figure 13 - Energy demand for Domestic clusters

As for industrial loads shown in Figure 14, something similar occurs. All the groups present a very similar distribution, and the total energy demand is practically concentrated in 4 loads: Load\_I\_486, Load\_I\_491, Load\_I\_487 and Load\_I\_749. The energy demand of these loads accounts for 77% of the total.

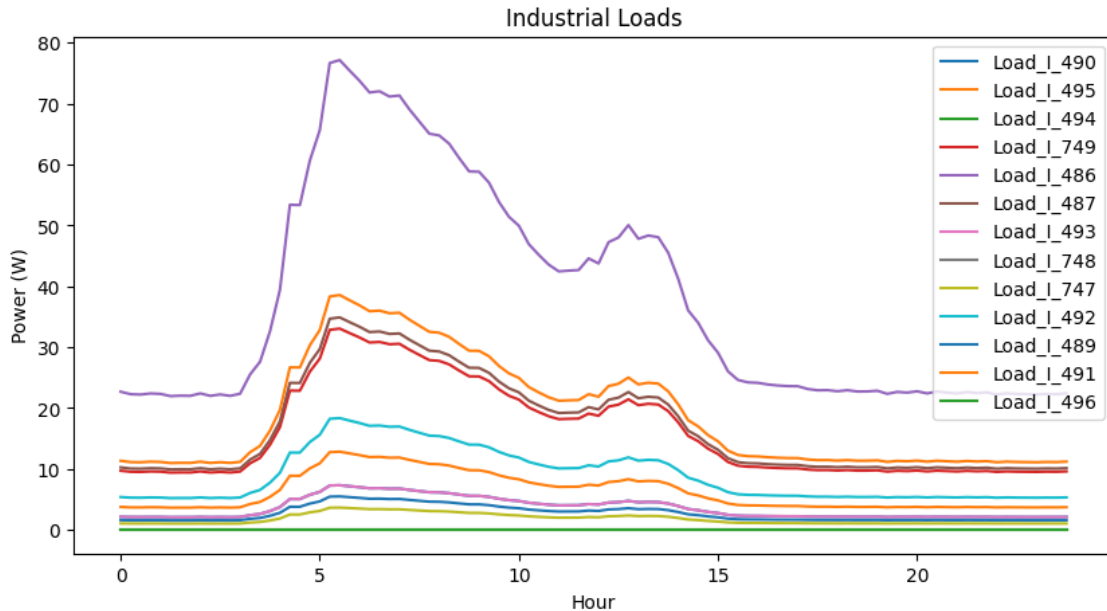


Figure 14 - Energy demand for Industrial clusters

Therefore, this first version tries to simplify the problem by acting only on these 7 loads. In this way, the state set is much smaller, speeding up development thus.

**Actions**

The RL-based optimizer seeks to optimize the energy demand of the electrical network. To do this, the distribution of energy demand of each load is modified, which, is turned into a shift in energy demand throughout the day.

This version of the optimizer uses four sets of discrete actions. The first set is the load selection. As explained above, seven loads over which to act are considered. Therefore, the action set encompasses 7 possible actions.

The second action set of actions determines the start time of the energy displacement. Therefore, this action set includes 24 different actions.

The third action set determines the percentage of energy that is shifted at different times. 3 different energy shifts have been established 1%, 5% and 10%. For example, if from 8:00 a.m. to 9:00 a.m. there is an energy demand of 10W and you want to make a 10% shift to the time slot between 9:00 a.m. and 10:00 a.m., this second time slot will increase your power demand by 1W, and the energy demand from 8:00 a.m. to 9:00 a.m. will decrease if by the same amount.

The energy demand shift presents a number of restrictions. The first one is that the total energy demanded by a load must remain constant throughout the day. The second one is that the shift can only occur within 2 contiguous time slots, the displacement cannot be applied to any arbitrary distant time slots.

These restrictions are considered in the model implementation phase.

### Rewards

The objective of the system is to optimize the operation of the electrical grid. This version tries to maximize two grid performance parameters: SSR (self-sufficiency ratio) and SCR (self-consumption ratio). SSR can be defined as the relation between the energy produced by the network and the energy consumed by the network. SCR is defined as the ratio between the power consumed by the network and the power produced.

The employee reward function is mean of both ratios:  $R = \frac{1}{2}(SCR + SSR)$ .

### Experiments

Results obtained from initial experiments with this first version of the RL-based optimizer for the Electric Grid optimization UC9 will be reported in D7.3 [33]. Following experiments will be conducted over more advanced releases of the optimizer in the context of WP6, and their results will be reported in D6.3 and D7.4.

The source code of this first implementation of the Electric Grid optimizer is available at the [IoT-NGIN GitLab repository](#) [31].

### 3 IoT-NGIN Privacy Preserving Federated Learning

The parallel evolution of both edge computing and Federated Learning has brought valuable opportunities in the deeper penetration of intelligence into modern services and applications through the exploitation of available computational and energy resources at different levels of the devices' hierarchy.

Edge computing brings computation closer to the edge, i.e., basically closer to the end user. This implies significant benefits e.g., related to reduced latency and increased energy efficiency, since considerable communication cost is saved, as data and workloads remain local, rather than being transmitted to a remote cloud server. Moreover, data are kept at the data owner's/controller's side, rather than being transferred to (untrusted) cloud third parties.

The edge computing paradigm is manifested for Machine Learning through Federated Learning (FL), which allows distributed ML model training among a set of collaborating (federated) edge nodes. In FL, data remain local, as local are the ML model training computations on that data, happening at edge devices. Then, local models are communicated to a central aggregation server, which combines the individual model updates received from individual edge nodes. A number of FL frameworks in literature support setting up and running training tasks in a federated way, which have diverse capabilities in real or simulated setups, number of nodes, ML techniques supported, etc. A comprehensive and comparative analysis of the state-of-the-art FL frameworks has been provided in D3.1 [6].

Yet, the "traditional" FL approach relies on the existence of trusted entities, both for the aggregation server and the FL participants, as well as on trusted communications among them. However, in realistic scenarios, this assumption is often optimistic.

To address the lack of trust in FL systems, privacy preservation techniques have been suggested in literature, which span the local updates' masking (e.g., through Differential Privacy - DP, Multi-party-computation), and identity protection (e.g., through homomorphic encryption, other cryptographic techniques, etc.). Such techniques have been analyzed in D3.1.

However, the practical application of those approaches and their efficiency/cost in the ML model development of different domains is not common ground across FL frameworks, ML algorithms and application domains. In IoT-NGIN, we have extensively investigated the application of privacy preservation mechanisms for three state-of-the-art FL frameworks, namely NVIDIA FLARE [34], Tensorflow Federated (TFF) [35] and FedPATE, an adaptation of Flower [36] to the Private Aggregation of Teacher Ensembles (PATE) approach [37], as tabulated in Table 5. This work is reported in D3.3.

Table 5 - Details of the ML and FL frameworks analyzed in D3.3

FL framework	ML algorithm	ML framework	Privacy preservation technique
NVIDIA FLARE	Single-Stage Object Detector (Yolov5)	Pytorch	DP, HE
FedPATE	Convolutional Neural Networks (CNNs) for image classification (Image Classifier)	PyTorch	DP
TFF	Multi-Layer Perceptron-Classifier	Tensorflow	DP

Based on this work, analyzing the efficiency and impact of privacy preservation in FL training of indicative ML models through the selected three FL frameworks, the next logical step of IoT-NGIN contributions has been towards the facilitation of the execution of FL training tasks with the preferred FL framework. To this end, we introduce the **Privacy-Preserving Federated Learning API (PPFL API)**, as a single-entry point for instantiating FL training tasks.

### 3.1 Description

#### Motivation

Usually, the execution of FL training requires significant effort related to the manual configuration for both the setup of the selected FL framework, the porting of ML models into the framework and the instantiation of the training activities. This also assumes familiarity with the internal operation of each FL framework, which might not always be the case. The great amount of manual work burdens the automation of the training tasks, required for accelerating operations in both real and simulation scenarios. To this end, IoT-NGIN introduces the PPFL API, as an indicative entry point for initiating FL training tasks through the analyzed frameworks and for the indicative ML models investigated, which represent indicative ML applications.

#### Approach

The introduction of a common entry point for the instantiation of FL training tasks would facilitate the work of the AI developer. Indeed, the PPFL API is implemented to hide the complexity and specificities of each FL framework and privacy preservation technique from the (API) user and to facilitate the automation of the FL training processes through simple API calls. Moreover, the integration with the MLaaS platform for model storage enables the use of the model through model sharing or model serving functionalities of the MLaaS platform via the relevant endpoints.

Considering the computational overhead that FL training may require, the deployment of FL training tasks on the cloud has been addressed through the PPFL API, benefiting from the resilience and scaling features inherent in a cloud environment. As such, the PPFL API is designed as a cloud native application, which is also able to automate the deployment of

FL training tasks in a cloud native architecture. Based on these, the source code of each FL framework is containerized as Docker images, which are published in a Container repository, Dockerhub [38] in our case. The Docker images are instantiated as deployments described in K8S manifests, indicating the components and configuration for each FL framework as Kubernetes app. Then, a training task with the selected FL framework can be deployed, submitting it as an Argo Workflow [39] in the underlying K8S deployment. The deployment approach adopted for the PPFL API is depicted in Figure 15.

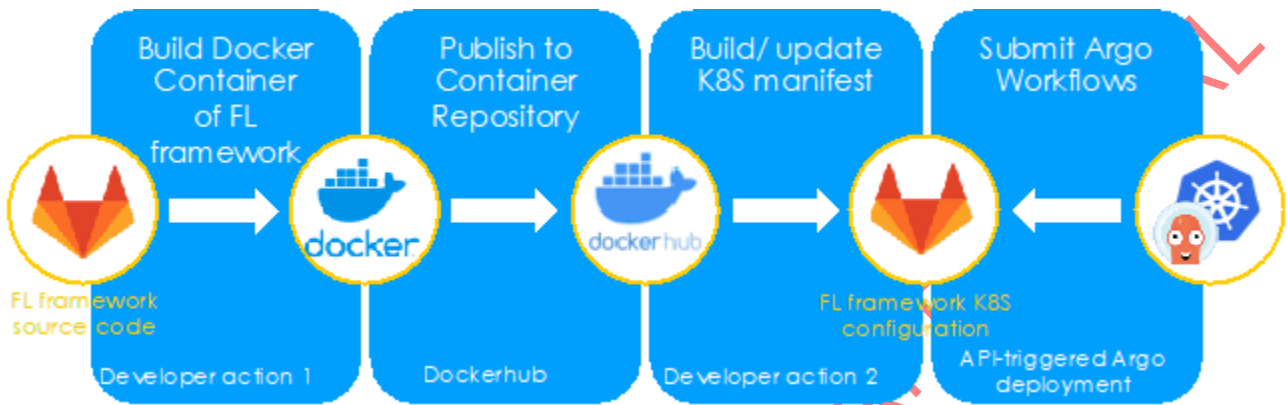


Figure 15 - The PPFL API deployment approach through Docker containers, K8S and Argo Workflows

The first step towards the realization of the PPFL API has been the homogenized presentation of the three FL frameworks. The parameterization of each of the three FL frameworks has been analyzed, so that the definition of the parameters' values can be provided by the PPFL API user. Such parameters may include FL related ones, such as the number of clients, ML related ones, such as the number of rounds and epochs, or even privacy preservation techniques related, such as DP parameter, etc.

Code updates have been applied to allow containerization and all three frameworks have been containerized as Docker images and described as K8S manifests. Specifically, the FL framework containers currently support operation in simulation mode for TFF and FedPATE, as well as operation on real nodes through NVIDIA FLARE. In addition, having the maximum level of automation possible in mind, no hardcoded information or configuration has been left within the containers. On the contrary, as the configuration parameters differ among the FL frameworks, they are provided by the user and are properly set for each FL framework by the PPFL API, e.g., as environmental variables or configuration files.

### 3.2 Technical design

The aim of the PPFL API is to enable the AI developer to easily deploy FL training tasks through NVIDIA FLARE, FedPATE or TFF, following a CI/CD paradigm for the integration and deployment of FL frameworks.

In order to achieve this, the PPFL API incorporates the following functionalities:

- User interface allowing interaction of the AI developer with the PPFL API
- Management/execution of the received requests
- Execution of training tasks as defined in the FL request
- Access control-based API protection

- Integration with the model storage component of the MLaaS platform, in order to store the trained global model after each FL training execution and make it available to third parties via the MLaaS.

The development view of the PPFL API is depicted in Figure 16.

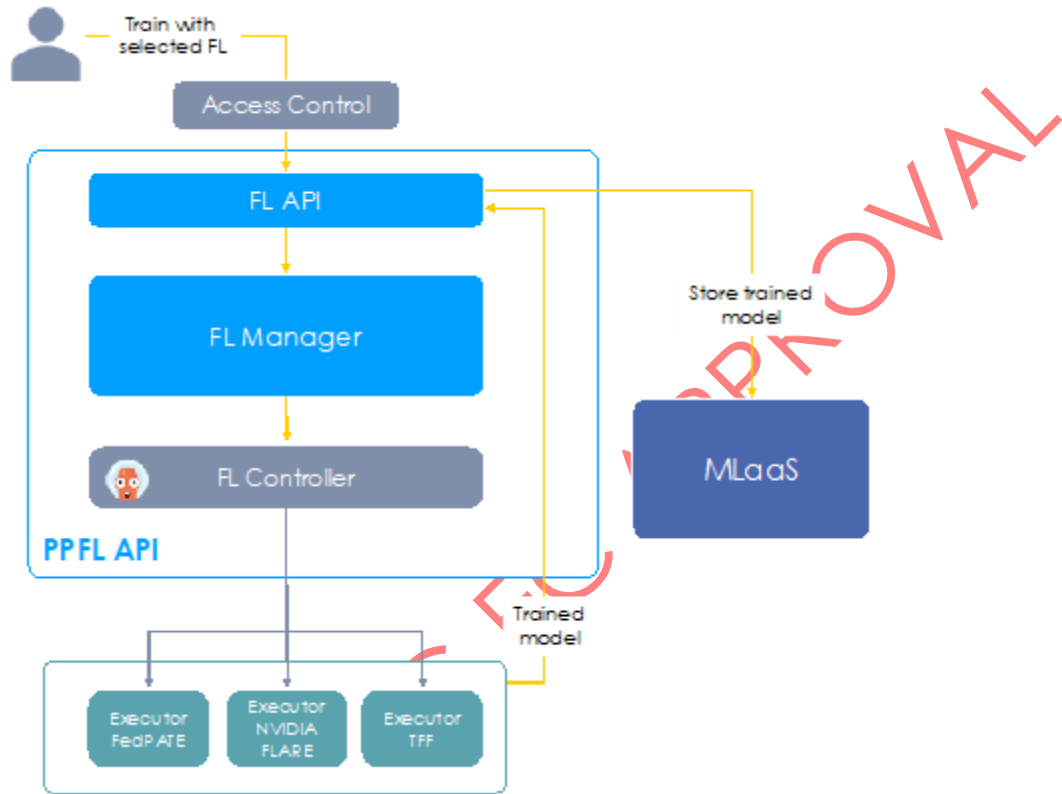


Figure 16 - The technical design of the PPFL API

### 3.2.1 Description of subcomponents

#### FL API

The FL API is the interaction point of external users or services to the PPFL API. It provides a set of RESTful endpoints which allow for submitting requests for deployment of FL training activities. The API is Authentication, Authorization and Accounting (AAA)-protected through OAuth 2.0 [40] and OpenID Connect (OIDC) [41] based processes for access and identity management. Moreover, the validation of requests is within the functionalities of this subcomponent.

These requests are forwarded to the FL Controller for being further managed and processed.

#### FL Manager

The PPFL API is designed on the premise of supporting automated, API-driven operation, without tight coupling with specific FL frameworks. The FL Manager acts as a link between an FL framework and external users, separating them properly, so that users do not need to interact with specific FL frameworks. Through this decoupling, adding or removing supported



FL frameworks is made easy, while additional feature development in each of these frameworks can vary from one to another with minimal impact on the external user. The FL Manager receives the deployment requests, registers relevant information and communicates with the relevant controller in order to execute the request. Moreover, this subcomponent is responsible for managing the outcome of the execution and forwarding it to the API for the external users' information.

### FL Controller

The FL Controller is responsible for instantiating FL training tasks in the relevant FL framework. The Controller obtains the container images of the FL framework and deploys it within the Kubernetes cluster. Depending on the FL framework, the controller may trigger the training in simulation or in real mode. In the simulation mode, the Controller instantiates both the FL server and the user-specified number of FL clients, initiates the training session, receives the status of the execution result and the storage of the final model in the MLaaS platform. In the real mode, the Controller instantiates the FL server, while the FL clients need to be instantiated individually within the cluster, under the clients' responsibility. The Controller will instantiate the training process and will collect any information about the status of the execution, as well as the storage of the training model in MLaaS.

### 3.2.2 Interfaces

The interfaces of the PPFL API through which the external user, e.g., an AI developer, may interact with the API are tabulated in Table 6. The API is also online documented via Swagger, which is accessible at <https://ppfl-api.iot-ngin.onelab/swagger/>, while examples of its usage are provided in section 5.3.

Table 6 - PPFL API interfaces

PPFL API		
Provided Interfaces	FedPATE Interface	
	Description	The interface enables requests for training tasks through the FedPATE FL framework.
	End-point URL	<a href="http://{BASE_URL}/fp/">http://{BASE_URL}/fp/</a> <a href="http://{BASE_URL}/fp/{FP_ID}">http://{BASE_URL}/fp/{FP_ID}</a>
	Protocol used	HTTP
	Methods	GET/POST/DELETE
	Message	Request Body (POST): { "net_name": "name", "num_rounds": 1, "num_teachers": 1, "teacher_epochs": 1,



D3.4 – ML models sharing and transfer learning implementation

	<pre> "student_epochs": 1, "batch_size": 1, "learning_rate": 1, "noise": "noise", "epsilon": 0, "sigma": 0, "num_queries": 1, "noise_data": 1, "bucket_name": "name", "domain_name": "name", "access_key": "key", "secret_key": "key" }                 </pre> <p><i>*values are indicative only</i></p>
TensorFlow Federated	
Description	The interface enables requests for training tasks through the TFF FL framework.
End-point URL	<a href="http://{BASE_URL}/tff/">http://{BASE_URL}/tff/</a> <a href="http://{BASE_URL}/tff/{tff_ID}">http://{BASE_URL}/tff/{tff_ID}</a>
Protocol used	HTTP
Methods	GET/POST/DELETE
Message	Request Body (POST): <pre> {   "imb_parameter": 1,   "dp_parameter": 1,   "n_clients": 1,   "n_samples": 1,   "target_label": "label",   "minio_repo": "repo_url",   "access_key": "key",   "secret_key": "key" }                 </pre> <p><i>*values are indicative only</i></p>
NVIDIA FLARE - Server	
Description	The interface enables requests for starting or shutting down the server and the admin client of the NVIDIA FLARE FL framework.
End-point URL	<a href="http://{BASE_URL}/nvoverser/">http://{BASE_URL}/nvoverser/</a> <a href="http://{BASE_URL}/nvoverser/{nvoverser_ID}">http://{BASE_URL}/nvoverser/{nvoverser_ID}</a>

D3.4 – ML models sharing and transfer learning implementation

	Protocol used	HTTP
	Methods	GET/POST/DELETE
	Message	Request Body (POST): <pre>{   "start": true,   "shutdown": true }</pre>
	NVIDIA FLARE - Training	
	Description	The interface enables requests for triggering training tasks through the NVIDIA FLARE FL framework.
	End-point URL	<a href="http://{BASE_URL}/nv/">http://{BASE_URL}/nv/</a> <a href="http://{BASE_URL}/nv/{nv_ID}">http://{BASE_URL}/nv/{nv_ID}</a>
	Protocol used	HTTP
	Methods	GET/POST/DELETE
	Message	Request Body (POST): <pre>{   "epochs": 1,   "batch_size": 1,   "num_clients": 1,   "num_rounds": 1,   "bucket_name": "name",   "domain_name": "name",   "secret_key": "key",   "access_key": "key" }</pre> <p><i>*values are indicative only</i></p>
	Required Interfaces	No interfaces are required for the PPFL API.

DRAFT - PENDING EC APPROVAL

## 4 Polyglot Model Sharing

### 4.1 Description

The Machine Learning Models Sharing Platform offers a secure environment to manage datasets and the machine learning models' lifecycle, offering a user-friendly interface that abstracts end users from the complexities involved, such as, to cite a few of them: the metadata management for enabling assets' traceability, guaranteeing their immutability, as well as ensuring the reproducibility for all trained machine learning models.

The platform offers a transparent data integrity mechanism, which guarantees the immutability and complete traceability of all artifacts stored (e.g., datasets, machine learning models, training results); this is achieved by using a Blockchain-based approach for storing relevant metadata for all its assets.

The feature set that the framework offers is very useful for any applications that involve the management of machine learning models that involve sensitive information. Additionally, due to the implementation of the Polyglot Model service, the platform is especially suited for IoT applications, providing a simple solution for handling sensitive data, machine learning model training, and offering a platform-independent runtime machine learning model representation that enables a wider array of compatible hardware for inference purposes.

### 4.2 Technical Design

#### 4.2.1 Architecture

The Polyglot Model Sharing framework has been designed following a microservices-based approach, with distinct, single purpose services, namely the Model Sharing, the Blockchain, the Model Training and the Model translation services. This decision was taken in concordance to other key architectural aspects, mainly offering a Cloud native solution, with a strong focus on DevOps & Continuous Integration / Development and containerization.

We will now introduce the main components of the platform (see its architecture in Figure 17) in the following sub-sections.

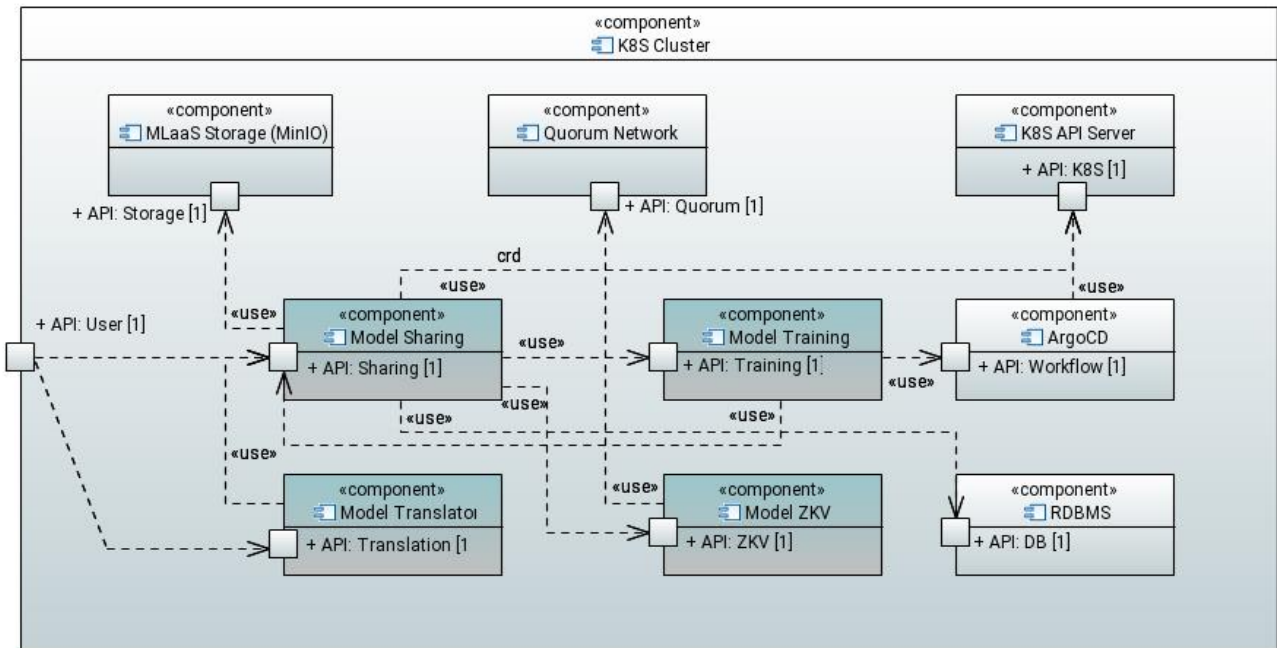


Figure 17 - Architecture diagram of the Polyglot Model Sharing framework

## 4.2.1.1 Components

### 4.2.1.1.1 Model Sharing service

This service provides an interface for registering (i.e., storing its metadata and artifacts) and retrieving ML models and datasets in the platform.

It implements operations for registering and downloading datasets, as well as registering models, scheduling their training, and downloading the resulting trained models.

For all the assets under its governance, it stores relevant metadata in the blockchain, with the help of the platform's Blockchain service (see next subsection).

By storing all relevant assets' metadata in the platform's blockchain instance, and overseeing the training phase of the models in a controlled environment, we can guarantee the replicability of all the trained models, and ensure that the training results were achieved exclusively with the specified inputs. Additionally, this metadata allows for guaranteeing the integrity of all stored assets.

The operations implemented and offered by the service are further detailed in the implementation section.

### 4.2.1.1.2 Blockchain service

This service provides an interface for deploying and interacting with smart contracts in a EVM (Ethereum Virtual Machine) [42] blockchain (e.g., ConsenSys Quorum [43]).

It implements operations for creating, retrieving, and interacting with smart contracts in the platform's blockchain instance.

The operations implemented and offered by the service are further detailed in the implementation section.

### 4.2.1.1.3 Model Training service

This service is in charge of scheduling model training jobs to be executed in the training cluster. As mentioned in the platform's introductory section, the main motivation for this service is offering a controlled environment for the training of machine learning models, to be able to guarantee the reproducibility and immutability of the resulting trained machine learning models. This service trains models in a batch manner, given complete train and test datasets, unlike the Online Learning service introduced in section 2.1, which trains models with data injected dynamically. Moreover, this service is only required to guarantee the integrity of a model stored in the MLaaS storage for a given dataset.

Model training jobs are scheduled executions of containers that enclose the definition and instructions for the training of a registered machine learning model.

Registered models must provide a training container image, stored in a container registry of choice, which will be provided with the training dataset indicated upon the model's registration, and its resulting model will be automatically registered in the platform, making it available to the end user, and the rest of the platform's features.

The model training jobs are executed in the platform's MLaaS Kubernetes Cluster, with the help of Argo Workflows [39].

The operations implemented and offered by the service are further detailed in the implementation section.

### 4.2.1.1.4 Model Translation service

This service offers the ability for the platform to transform registered machine learning models into the Open Neural Network Exchange (ONNX) [44] format upon training, offering a compatibility layer for all stored models in the platform.

By providing this compatibility layer, model developers can implement their machine learning models in their backend of choice (i.e., TensorFlow, PyTorch), without compromising on the limited hardware support provided by their backend of choice.

The operations implemented and offered by the service are further detailed in the implementation section.

## 4.2.1.2 External Dependencies

The platform relies on several external components for the implementation of its core features.

We will proceed to introduce the platform's external dependencies in the following sub-sections.

### 4.2.1.2.1 MLaaS PostgreSQL instance

The MLaaS PostgreSQL [45] instance is required in order to host the relational database that the Model Sharing service depends on for storing additional information about the models registered in the platform.

#### 4.2.1.2.2 IoT-NGIN ConsenSys Quorum instance

The IoT-NGIN ConsenSys Quorum Blockchain instance is required by the Blockchain service, in order to store the metadata of the datasets and machine learning models registered in the platform.

The metadata is stored in smart contracts, which are immutable programs stored in the blockchain, which control the access and actions of their implementation, based on the contract's definition.

For our use case, contracts are used as means of storage of the relevant metadata of datasets and models registered in the platform.

#### 4.2.1.2.3 MLaaS Model Storage instance

The MLaaS Model Storage instance, based on MinIO, is required by the Model Sharing service, as the platform of choice for object storage solution.

The choice of object storage as the platform's artifact storage solution is justified by the flexibility provided in its organizational structure, and its content-agnostic approach to storage.

The MLaaS MinIO instance for the project stores all the platform's persistent assets, i.e., datasets and machine learning models.

#### 4.2.1.2.4 Argo Workflows instance

The Argo Workflows instance is required by the Model Training service, in order to schedule the model training jobs.

Argo Workflows allows for the execution of sequential jobs in Kubernetes Clusters. In our use case, each model registered on the platform triggers a model training job, which executes a user-defined container image, providing it with the associated model dataset, and registers the resulting model in the platform. The registration step, in this context, involves storing the training results in the Model Sharing service, which, in turn, coordinates the update of the model metadata in the blockchain instance by means of interfacing with the Blockchain service. A more in-depth explanation of this process can be found in the section 4.3.

## 4.2.2 Implementation

The repository containing the implementation of the platform is hosted in the [IoT-NGIN GitLab project \[46\]](#).

### 4.2.2.1 Project Repository Structure

We will now further detail the project's repository structure (see Figure 18).

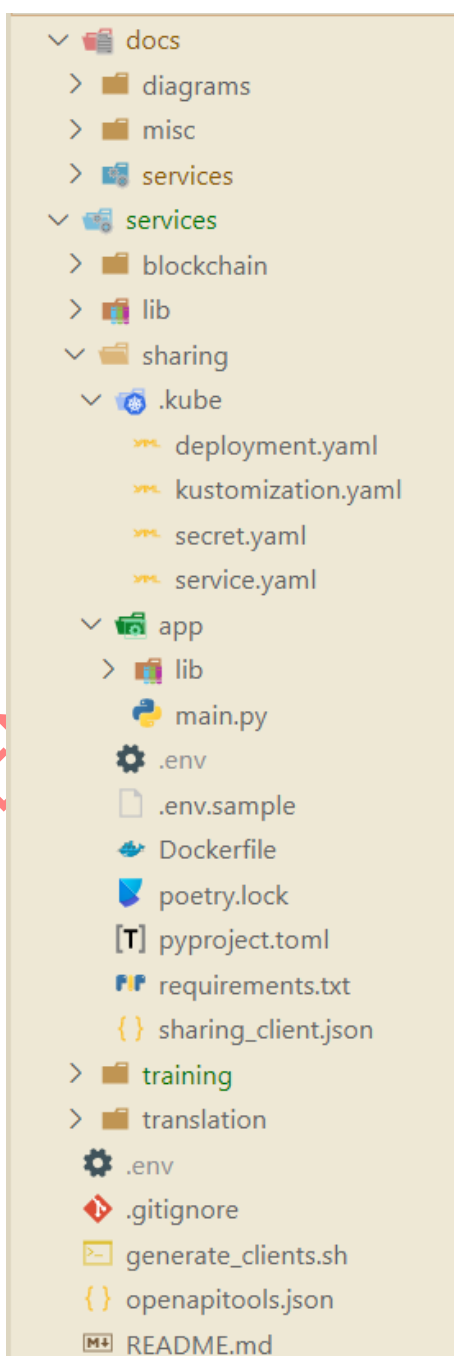


Figure 18 - Polyglot Model Sharing framework implementation repository directory structure

## D3.4 – ML models sharing and transfer learning implementation

- **/docs:** directory containing platform documentation and technical diagrams
- **/services:** directory containing the platform's services implementations
  - o **/services/{service}:** directory containing the service's implementation
    - **.kube/:** contains the service's Kubernetes resource definitions
    - **app/:** contains the Python project implementation
    - **.env.sample:** reference file for the service's required environment variables
    - **Dockerfile:** contains the instructions for building the service's container image
    - **poetry.lock:** service's Poetry dependencies lockfile
    - **pyproject.toml:** configuration file for the Python project's tooling
    - **requirements.txt:** compiled form of the Python project's dependencies
    - **{service}\_client.json:** OpenAPI schema for the service's HTTP REST API
- **generate\_clients.sh:** script for programmatically generating SDK for interacting with the platform service's HTTP REST APIs, based off their OpenAPI specification definitions
- **openapitools.json:** OpenAPI Generator configuration file
- **README.md:** project's repository introductory documentation

The platform components (services) have been implemented following a microservices-based approach, with distinct, single purpose services.

They have been developed in Python, using the [FastApi](#) framework. The service's dependency management is handled using [Poetry](#) [47].

Each service exposes a REST API, along with its specification and the documentation available on the OpenAPI format; a Swagger UI instance is always accessible at the `/docs` HTTP route. The [OpenAPI](#) schema [48] for the REST API is dynamically generated by FastAPI when running the server, and can be accessed via HTTP at `/openapi.json`.

In order to interact with the service, a Python client for each service is provided, providing a SDK for interacting with the service's REST API. This client is generated programmatically using the [OpenAPITools' OpenAPI Generator](#) [49], using the aforementioned OpenAPI schema.

The Python clients are published in the project's [GitLab Package Registry](#) [50].

A [Dockerfile](#) [51] is provided for each service, in order to build a container image that runs the service in a containerized way.

In a similar manner, the container image, built from the included Dockerfile instructions, is published in the project's [registry](#) [52].

The set of Kubernetes resources required for deploying the service in a Kubernetes cluster can be found at the `.kube` directory in the service's code repository.

## 4.2.2.2 Components

In section 7 Annex, the main components of the Polyglot Model Sharing Framework are described in terms of their exposed interfaces and their main supported processes.



### 4.3 Implementation for IoT-NGIN LLs

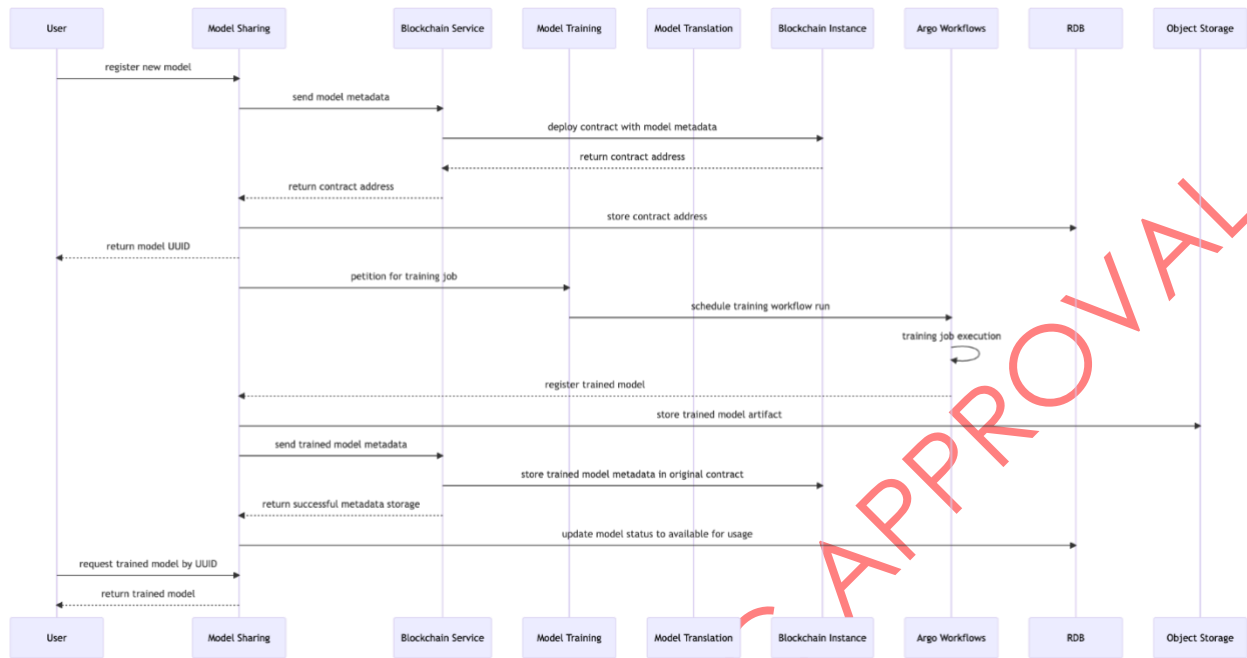


Figure 19 - Sequence diagram of the Polyglot Model Sharing Framework demo use case

At the time of writing, the Polyglot Model Sharing Framework has not been used by the IoT-NGIN Living Labs' use cases, but it is expected some of them (e.g., Smart Agriculture and Smart Energy LLs) will use it during the final validation, reported by WP6. In this section, we provide our own use case to showcase (Figure 19) the overall end-to-end usage of the platform.

In the sequence diagram shown in Figure 19, we can see in further detail the interactions between the platform's services, as well as the platform's external dependencies for our own use case.

1. We register the dataset to be used in the training phase in the platform
2. We register the machine learning Training model in the platform, referencing the registered dataset
3. Wait for the model training step to complete.
4. Retrieve (download) the model
5. Request model in ONNX format (polyglot service)

For our example use case, we document, in the following, the workflow for registering and training an image classification model on the platform.

The implementation for this demo Service case can be found in the project's GitLab repository, in the /demo directory (Figure 20).

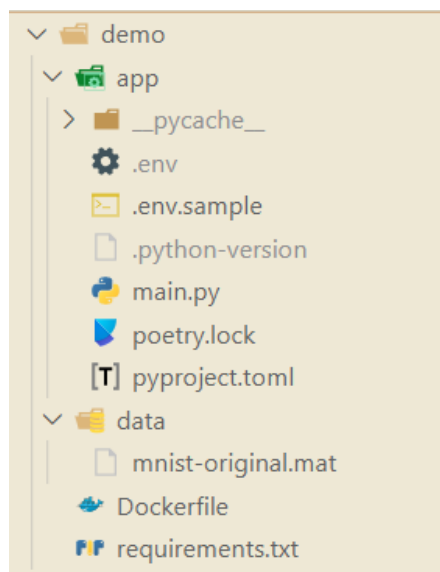


Figure 20 - Demo use case's implementation directory structure

The machine learning task is a multinomial image classification [53] for handwritten digits. We will use the MNIST [54] dataset for training this model.

We implement the image classification model in Pytorch [24]. This image classification model will be based on a CNN architecture [55] (Listing 1).

```

MNISTNetwork(
  (features): Sequential(
    (0): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=800, out_features=1024, bias=True)
    (2): ReLU()
    (3): Linear(in_features=1024, out_features=512, bias=True)
    (4): ReLU()
    (5): Linear(in_features=512, out_features=10, bias=True)
    (6): Softmax(dim=1)
  )
)

```

Listing 1 - Demo use case classification machine learning model definition

For us to be able to register and train our machine learning model in the platform, we need to provide our model definition and training process as a container image (Listing 2). This container image must conform to the guidelines specified in the implementation section 7.3 of the Model Training service.

```
FROM python:3.8.16-slim

RUN apt-get update

WORKDIR /code

COPY requirements.txt /code/requirements.txt
RUN pip install -r requirements.txt

COPY ./app /code

CMD ["python", "main.py"]
```

Listing 2 - Dockerfile instructions for the demo use case's model training container image

In our use case, we are using this platform's GitLab project container registry for hosting our training image.

Once we have all the required pre-requisites, we will further proceed with the model registration process.

Before registering the model in the platform, we first must register the dataset that we want to use for our model training job.

We will do this by executing the following HTTP request:

`bash`

```
curl --request POST \
  --url http://localhost:9005/dataset \
  --header 'content-type: multipart/form-data' \
  --form dataset=@mnist-original.mat \
  --form 'metadata={
    "organization_id": "5d066d58-06de-47ab-a2f6-c8f413e21947",
    "samples_dimension": "(1,32,32)"
  }'
```

Upon the successful registration of the dataset, the service will respond to our request with a unique identifier (UUID) for the dataset. We will use the received UUID for referring to our dataset when registering our model in the platform.

`json`

```
{
  "dataset_id": "b2a30898-b72c-4b08-802b-7786425d01f9"
}
```

We will now check the platform's MinIO instance to check that the dataset artifact has been registered in the object storage (Figure 21):

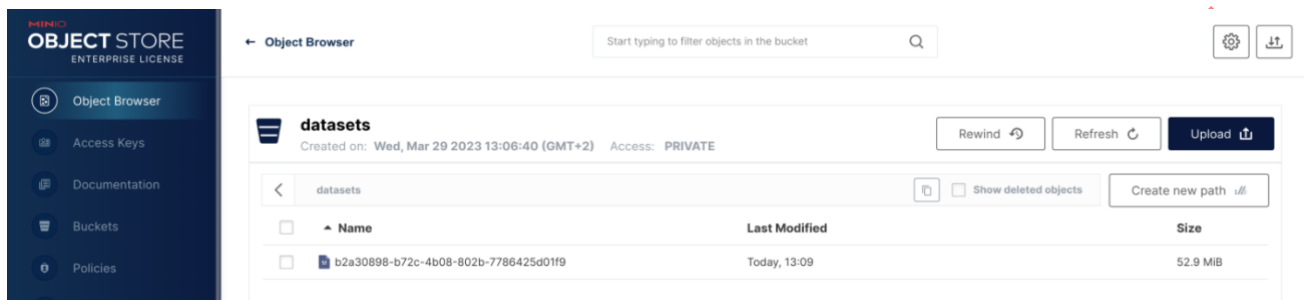


Figure 21 - Dataset artifact stored in the object storage

As we can see, the dataset has been stored in the datasets bucket, named with the UUID that we received upon registration.

We can additionally check that the dataset's metadata has been successfully registered in the platform's blockchain instance by verifying that a smart contract for this dataset has been deployed.

This is possible by executing the following request to the Blockchain service, indicating the address of the deployed contract (which can be found in the project's internal PostgreSQL database).

**bash**

```
curl --request GET \
  --url
http://127.0.0.1:9007/contract/0x9ab7CA8a88F8e351f9b0eEEA5777929210199295
```

The service responds with the following JSON-encoded text, which represents the metadata stored in the smart contract.

**json**

```
{
  "organization_id": "5d066d58-06de-47ab-a2f6-c8f413e21947",
  "samples_dimension": "(1,32,32)",
  "size_bytes": 55426379,
  "hash": "adbc812a1f0ab4c881a41fc872cb643e"
}
```

We will now register our model in the platform, referring to our dataset with the UUID received in the previous step. We will do so by executing the following HTTP request.

**bash**

```
curl --request POST \  
  --url http://localhost:9005/model/ \  
  --header 'content-type: multipart/form-data' \  
  --form 'metadata={  
    "model_params": {  
      "developer_id": "dd8fca81-a999-40f5-81f0-68627e303a27",  
      "organization_id": "5d066d58-06de-47ab-a2f6-c8f413e21947",  
    },  
    "data_params": {  
      "dataset_id": "b2a30898-b72c-4b08-802b-7786425d01f9"  
    }  
  }' \  
  --form train_image=registry.gitlab.com/h2020-iot-  
  nginx/enhancing_iot_intelligence/t3_4/ml-model-sharing/demo_model_image
```

Upon the successful registration of the model in the platform, the service will respond to our request with a unique identifier (UUID) for the model. We will use the received UUID for referring to our model in the platform.

json

```
{  
  "model_id": "62e8efbc-47e5-4e60-abe5-8fb8a7b676ba"  
}
```

We will now wait for the model training job to finish. We can check the status of the training job by invoking the model retrieval procedure in the Model Sharing service.

json

```
curl --request GET \  
  --url http://localhost:9005/model/62e8efbc-47e5-4e60-abe5-8fb8a7b676ba
```

If the training job is still not finished, the service will respond in this way:

text

```
"Model 62e8efbc-47e5-4e60-abe5-8fb8a7b676ba has not been trained yet. Please,  
try again later."
```

Otherwise, we will receive the resulting artifact from the training job.

We will now, as in the case of the dataset registration, check that the trained model has been stored in the platform's MinIO instance (Figure 22).

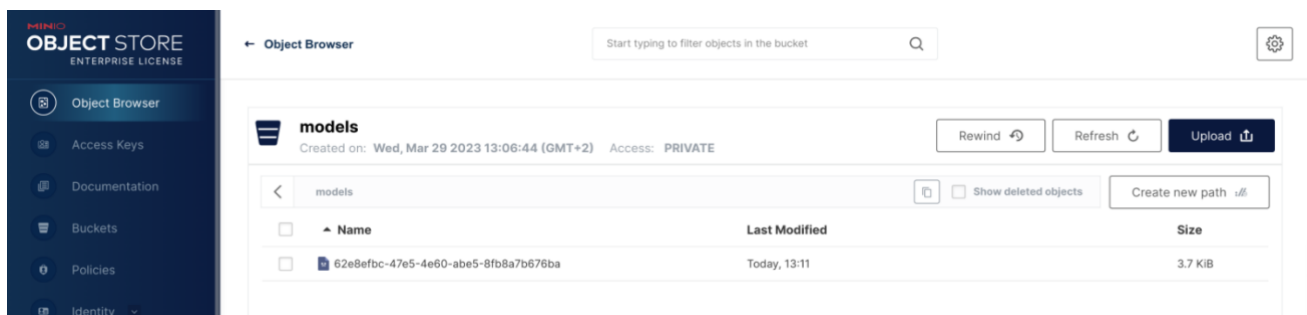


Figure 22 - Trained model in MLaaS storage

As in the previous case, we can find our model stored in the models' bucket, with the UUID we received when retrieving the model from the Model Sharing service.

We will also check that the model's metadata stored in the blockchain includes the training results (i.e., it has been verified):

```
bash
```

```
curl --request GET \
  --url
http://127.0.0.1:9007/contract/0x43D1F9096674B5722D359B6402381816d5B22F28
```

Inspecting the response received from the service, we can identify that the *res\_hash* parameter is present, and that it matches the one we can calculate in the trained model previously retrieved.

```
json
```

```
{
  "model_params": {
    "developer_id": "dd8fca81-a999-40f5-81f0-68627e303a27",
    "organization_id": "5d066d58-06de-47ab-a2f6-c8f413e21947",
    "train_image_hash": "5b4d62ed98ebaaf764a63ae31eaaf8d6",
    "res_hash": "f73b9eb23dcf464a1916d8e991dfaff4"
  },
  "data_params": {
    "dataset_id": "b2a30898-b72c-4b08-802b-7786425d01f9"
  }
}
```

For the last step of our demo, we will invoke the model translation service in order to translate our trained model into the ONNX format. We will do so by executing the following request.

```
json
```

```
curl --request POST \
  --url http://localhost:9008/translate/62e8efbc-47e5-4e60-abe5-8fb8a7b676ba
```

The service will then respond to our request with the trained model in the ONNX format.

## 5 Installation and User Guide

### 5.1 Online Learning Service

This section focuses on the updates required to create, deploy and use an OL service within the MLaaS, w.r.t. the procedure described in D3.3. Figure 23 shows the procedure for creating and deploying an OL service.

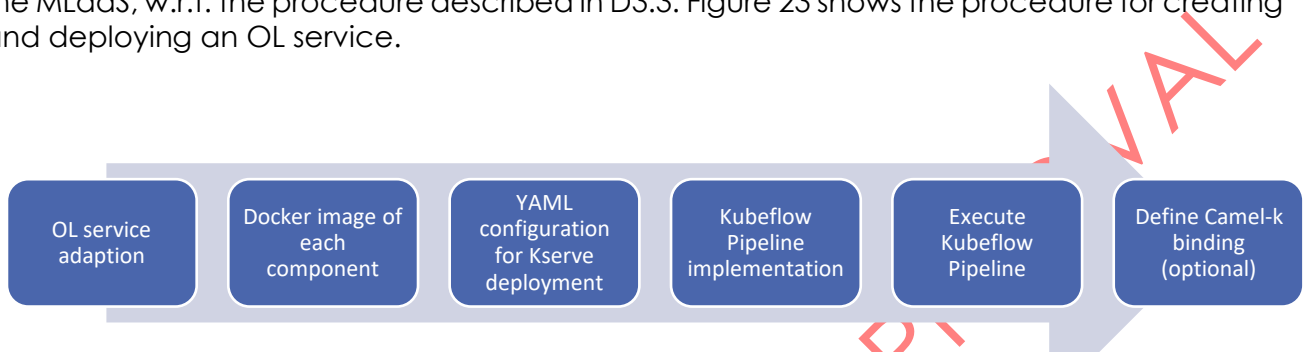


Figure 23 - Workflow to deploy OL service

Alike D3.3, it is required to have a ML/DL model implemented before creating the OL service. The model creation stage is out of the scope of OL framework and must be provided by ML engineers. In addition, the model also must be registered in the Model Sharing.

The first step is the OL service adaptation. This stage involves the provisioning of three different components:

- i. *Predictor*: Configuration of different parameters that are specific for each OL service such as the MinIO location, the framework used for implementing the ML/DL model, the service name, etc.
- ii. *Transformer*: Implementation of the data pre and post processing pipeline. Each OL service requires its own *Transformer* since the input data is use-case specific.
- iii. *Explainer (Optional)*: Implementation to provide explanations to the inferences made. This component is optional and the XAI algorithm must be implemented by the ML engineers and stored in MinIO so the Explainer can load it.

Once the components are implemented and configured, each of them needs to be encapsulated in a Docker image. This Docker image must be uploaded to a Docker registry in order for Kubeflow to include it to the pipeline. An example of Dockerfile is provided in D3.3.

At this point, each OL component is containerized, and Docker images are available in Docker registry.

Before deploying the OL service by executing a Kubeflow pipeline, Kserve YAML manifest is defined (Listing 3). This file specifies the Docker image of each component, name of service and configuration of environment variables.

```
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: # <Inference Service Name>
  namespace: # <Namespace>
spec:
  predictor:
    containers:
      - name: # Conatiner name
        image: # <OL_Predictor_Image>
        imagePullPolicy: Always
        envFrom:
          - configMapRef:
              name: # <Env_Config_Map>
            - secretRef:
                name: # <Env_Secret>
  transformer:
    containers:
      - image: # <OL_Transformer_Image>
        name: # Conatiner name
        imagePullPolicy: Always
        envFrom:
          - configMapRef:
              name: # <Env_Config_Map>
            - secretRef:
                name: # <Env_Secret>
  explainer:
    containers:
      - image: # <OL_Explainer_Image>
        name: # Conatiner name
        imagePullPolicy: Always
        envFrom:
          - configMapRef:
              name: # <Env_Config_Map>
            - secretRef:
                name: # <Env_Secret>
```

Listing 3 - Kserve YAML manifest

Once Kserve YAML manifest is created, the next steps are the same as described in D3.3. At Kubeflow, a Jupyter Notebook is created and the Kubeflow pipeline is implemented. Once the execution is finished, the OL API REST service will be deployed.

Optionally, Camel-K binding can be created to support pub/sub communications. The binding creation is also available in D3.3.



Section 2.1.2 described OL monitoring service. This is an additional service, so its implementation and deployment are outside of the Kubeflow Pipeline. This service needs Grafana and Prometheus to be deployed in the cluster so its metrics can be visualized.

Both Grafana and Prometheus have been deployed using *Helm* [56]. Helm allows to configurate applications by applying a configuration YAML manifest. Therefore, Prometheus is configured so it can scrap new OL monitoring service. The configuration file can be find in the [GitLab repository](#) [57].

Figure 24 depicts the deployment steps.

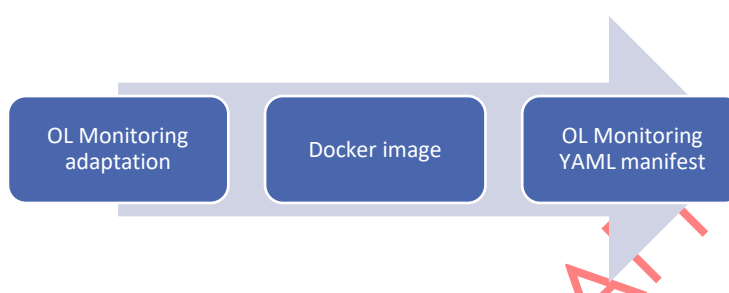


Figure 24 - Workflow to deploy OL monitoring service.

The first step is the OL monitoring service adaptation. This step aims to configure some aspects such as the data type to be monitored, the reference data and kind of metrics to be registered. The service implementation can be founded in the [Monitoring for MLaaS GitLab repository](#) [57]. Once the service is ready, it must be containerized in a docker image and upload to a Docker registry. The Dockerfile can be found in the [GitLab repository for MLaaS Prometheus](#) [58]. Finally, the YAML manifest is defined ([Manifest for MLaaS Prometheus](#) [59]) and applied so the service is deployed.

After following the steps described above, there are two API REST services i) the OL service and ii) the OL monitoring service. OL Service presents two endpoints, one corresponds to the Transformer and the other to the Explainer. The Transformer is waiting to receive data in JSON format either to update the model or to perform an inference. The Explainer is expecting data in JSON format, however in this case it is to provide an explanation of the inference. The way to invoke the OL services is described in D3.3.

Regarding the OL monitoring service, this service does not present any endpoint accessible from outside of MLaaS. However, the Grafana GUI can be accessed to observe the OL monitoring service metrics. Grafana allows you to create dashboards or import them in JSON format. Any of the 2 options is valid and works. Some of the dashboards that have been shown in section 2.1.3.1 can be located in the [GitLab project for MLaaS Grafana](#) [60].

## 5.2 Reinforcement Learning Service

Instructions to install the RL-based Optimizer as an MLaaS service and to use it will be given in D6.3 [1]. At the time of writing, a first version of the RL-based Optimizer has been provided, which needs preliminary evaluation (planned to be reported in D7.3 [33]), further development and improvement, and therefore, it is not yet ready to be delivered for wide usage within the IoT-NGIN pilot.

## 5.3 Privacy-preserving Federated Learning Framework

The following instructions refer to the installation of PPFL API.

### 5.3.1 Prerequisites

The prerequisite technologies to successful installation of the PPFL API include:

- Docker [51]
- Kubernetes [61]
- Argo Workflows [39]
- Keycloak [62]
- Helm [56]
- Linux OS, ideally Ubuntu 20.04 LTS

### 5.3.2 Installation Guide

#### 5.3.2.1 Local deployment

First, the repository for the PPFL API must be cloned in the desired local directory.

```
bash
```

```
git clone https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/privacy-preserving-federated-learning/privacy-preserving-federated-learning-api.git
```

Next, the required packages must be installed.

```
bash
```

```
pipenv install
```

As the PPFL API relies on Keycloak for AAA services, the environmental variables must be defined in the host OS as follows.

Table 7 - Environmental variables' configuration for local deployment

Variable	Description	Example
KEYCLOAK_SERVER_URL	The URL of the Keycloak server	https://keycloak.iot-nginx.onelab.eu/
KEYCLOAK_REALM	The name of the Keycloak realm	iot-nginx
KEYCLOAK_CLIENT_ID	The ID of the Keycloak client	ppfl
KEYCLOAK_CLIENT_SECRET_KEY	The client secret key used to authenticate with Keycloak	<set to the key ' ' in secret '>

The values for KEYCLOAK\_SERVER\_URL, KEYCLOAK\_REALM, KEYCLOAK\_CLIENT\_ID, KEYCLOAK\_CLIENT\_SECRET\_KEY must be replaced with the appropriate values for the Keycloak instance that will be used.

Finally, in order to run the API server, the following command should be hit.

```
bash
```

```
python manage.py runserver
```

The API server is now running at <http://localhost:8000/>.

Finally, the user may call the PPFL API at [http://localhost:8000/{fl\\_framework}/](http://localhost:8000/{fl_framework}/), providing a valid JWT token in the authorization header, where *fl\_framework* could be *tff*, *fp*, *nv* for TensorFlow Federated, FedPATE or NVIDIA Flare, respectively.

### 5.3.2.2 Docker deployment

To deploy the API using Docker, the steps provided below must be followed.

First, build the Docker image using the following command.

```
bash
```

```
docker build -t your-image-name
```

Next, push the Docker image to a registry, replacing *docker-registry* and *image-name* with the appropriate values for your environment.

```
bash
```

```
docker login
docker tag image-name docker-registry/image-name
docker push docker-registry/image-name
```

Copy the *docker-compose.yml* located under the *fl-api* directory to a server supporting docker and docker-compose. Then, create a *.env* file under the same directory with the configuration presented in Table 8.

Table 8 - Environmental variables' configuration for Docker deployment

Env variable name	Description
POSTGRES_HOST	The hostname of the PostgreSQL server
POSTGRES_DB	The name of the PostgreSQL database
POSTGRES_USER	The username for the PostgreSQL user
POSTGRES_PASSWORD	The password for the PostgreSQL user
DJANGO_SETTINGS_MODULE	The settings module used by Django
SECRET_KEY	The secret key of the PPFL API app
ARGO_HOST	The URL of the Argo Workflows server
NAMESPACE	The K8S namespace in which the application is deployed.
KEYCLOAK_SERVER_URL	The URL of the Keycloak server
KEYCLOAK_REALM	The name of the Keycloak realm
KEYCLOAK_CLIENT_ID	The ID of the Keycloak client
KEYCLOAK_CLIENT_SECRET_KEY	The client secret key used to authenticate with Keycloak

Deploy the Docker image to a Docker host, issuing the command.

```
bash
```

```
sudo docker-compose up --build
```

Finally, the user may call the PPFL API Docker container at [http://localhost:8000/{fl\\_framework}/](http://localhost:8000/{fl_framework}/), providing their token as authorization header (acquired as explained in section 5.3.2.1), where *fl\_framework* could be *tff*, *fp*, *nv* for TensorFlow Federated, FedPATE or NVIDIA Flare, respectively.

### 5.3.2.3 Kubernetes Deployment

For the installation of the API in Kubernetes, the PPFL API repository must first be cloned with the following command.

```
bash
```

```
git clone https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/privacy-preserving-federated-learning/privacy-preserving-federated-learning-api.git
```

Then, the environmental variables for the PPFL-API Kubernetes deployment must be configured.

```
bash
```

```
cd ppfl-api/fl-api/Kubernetes/ppfl
```

The PPFL-API relies on a Postgres instance, on ArgoWorkflows and a Keycloak instance, as e.g., the one integrated with IoT-NGIN Access Control component. Before deploying the PPFL API, the secrets located in `kubernetes/postgres/secrets.yaml` and `kubernetes/ppfl/secrets.yaml` have to be configured. In addition, the environmental variables in the PPFL API deployment, located in `kubernetes/ppfl/deployment.yaml`, must be filled in with the appropriate values, as presented in Table 9.

Table 9 - Environmental variables' configuration for K8S deployment

Env variable name	Description	Value*
POSTGRES_HOST	The hostname of the PostgreSQL server	iot-ngin-ppfl-api-postgres
POSTGRES_DB	The name of the PostgreSQL database	database
POSTGRES_USER	The username for the PostgreSQL user	user
POSTGRES_PASSWORD	The password for the PostgreSQL user	<set to the key 'postgres-password' in secret 'iot-ngin-ppfl-api-postgres'>
DJANGO_SETTINGS_MODULE	The settings module used by Django	flapi.settings.prod
SECRET_KEY	The secret key of the PPFL API app	<Set the key 'SECRET_KEY' in secret 'iot-ppfl-api'>
ARGO_HOST	The URL of the Argo Workflows server	http://argocd-argo-workflows-server
NAMESPACE	The K8S namespace in which the application is deployed.	iot-ngin
KEYCLOAK_SERVER_URL	The URL of the Keycloak server	https://keycloak.iot-ngin.onelab.eu/
KEYCLOAK_REALM	The name of the Keycloak realm	iot-ngin
KEYCLOAK_CLIENT_ID	The ID of the Keycloak client	ppfl
KEYCLOAK_CLIENT_SECRET_KEY	The client secret key used to authenticate with Keycloak	<set to the key 'KEYCLOAK_CLIENT_SECRET_KEY' in secret 'iot-ppfl-api'>

*\*Indicative values from a test deployment used for exemplary purposes in this guide*

The installation of the auxiliary components is described in the following.

### Argo Workflows

The PPFL-API relies on the Argo Workflows component to deploy the dockerized FL frameworks. The installation of the component is performed via Helm, particularly using the bitnami chart. To install, the following step must be performed.

**bash**

```
cd ppfl-api/fl-api/Kubernetes/argoworkflows
helm repo add bitnami https://charts.bitnami.com/bitnami
helm install argocd -f values.yaml bitnami/argo-workflows
kubectl apply -f role.yaml
```

### POSTGRES

To install the Postgres instance, the secrets.yaml file found in (ppfl-api/fl-api/kubernetes/postgres/secrets.yaml) needs to be configured with the login credentials of the Database and deployed, with the following command.

**bash**

```
kubectl apply -f kubernetes/postgres/secrets.yaml
```

Also, the name of the global.postgresql.auth.database key must be changed in the values.yaml file of the Postgres (ppfl-api/fl-api/kubernetes/postgres/values.yaml) to match the environmental variables POSTGRES\_DB in the Kubernetes deployment manifest of PPFL-API. Then, it can be deployed with the following command.

**bash**

```
helm install iot-ppfl-api-postgres bitnami/postgresql -n iot-ngin --version 12.1.3 -f kubernetes/postgres/values.yaml
```

### PPFL-API deployment

Finally, to deploy the PPFL-API, the following command must be executed.

**bash**

```
kubectl apply -f kubernetes/ppfl
```



## D3.4 – ML models sharing and transfer learning implementation

- --num\_teachers: int, the number of teachers/clients.
- --teacher\_epochs: int, number of epochs to train the teachers.
- --student\_epochs: int, number of epochs to train the student.
- --batch\_size: int, batch size (32 or 16).
- --learning\_rate: float, learning rate (default=0.001).
- --noise: str, type of noise for the aggregation mechanism ('laplacian' or 'gaussian', default='laplacian').
- --epsilon: float, epsilon for laplacian distribution (if --noise='laplacian').
- --sigma: float, standard deviation for gaussian-normal distribution (if --noise='gaussian').
- --num\_queries: int, number of queries to the student.
- --noise\_data: int, the number of data to add Laplacian noise (must be less than num\_queries).
- --bucket\_name: str, the bucket in MinIO where the best student model will be stored.
- --domain\_name: str, the domain where the model will be stored.
- --access\_key: str, access key for accessing MinIO.
- --secret\_key: str, secret key for accessing MinIO.

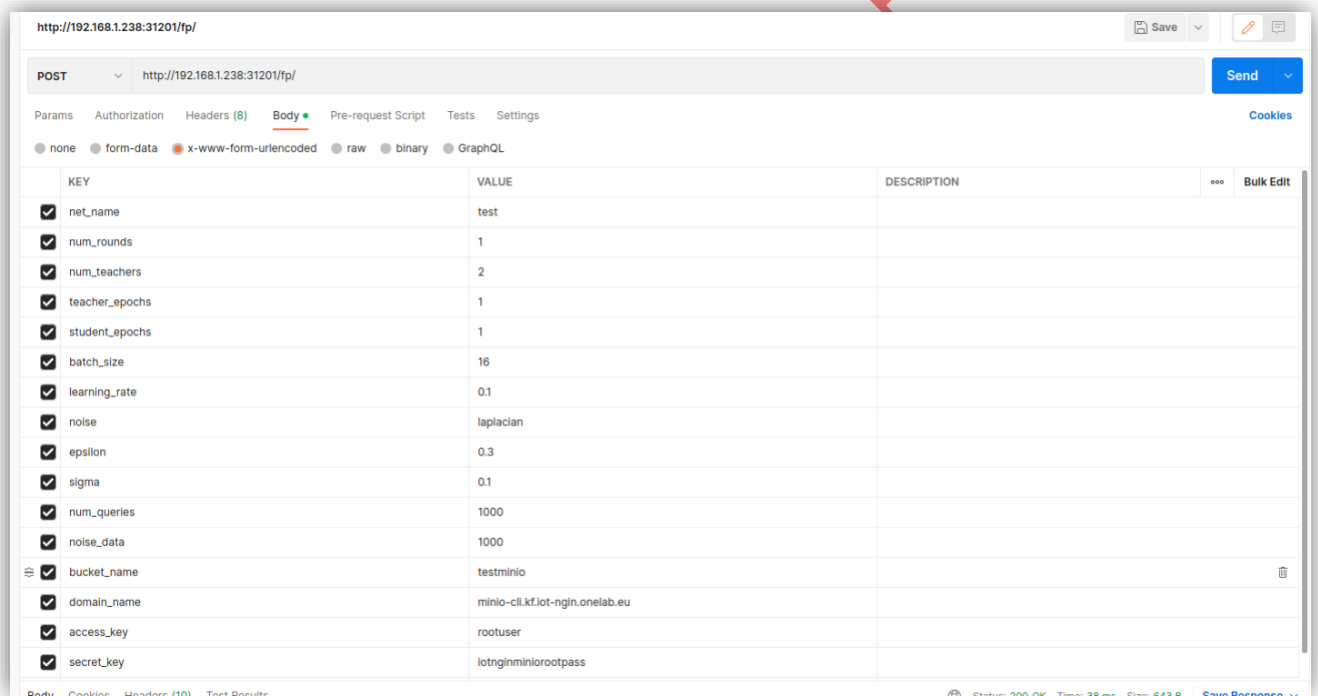


Figure 26 - Parameters setting in the POST request body for running the FedPATE framework.

After a successful request, a container is deployed and the training of the model starts in FedPATE in simulation mode.

Figure 27 shows that the pod for FedPATE (Flower PATE) has been created successfully. This view is available to the administrator of the MLaaS platform.



### D3.4 – ML models sharing and transfer learning implementation

```

PROBLEMS  DEBUG CONSOLE  TERMINAL
ARGO_INCLUDE_SCRIPT_OUTPUT: false
ARGO_DEADLINE: 0001-01-01T00:00:00Z
Mounts:
  /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-ngjfc (ro)
Conditions:
  Type             Status
  Initialized       True
  Ready             False
  ContainersReady  False
  PodScheduled     True
Volumes:
  kube-api-access-ngjfc:
    Type:              Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds: 3607
    ConfigMapName:      kube-root-ca.crt
    ConfigMapOptional:  <nil>
    DownwardAPI:       true
  OCS Class:          BestEffort
Node-Selectors:      <none>
Tolerations:         node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                    node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason      Age   From          Message
  ----    -
  Normal  Scheduled  43m   default-scheduler  Successfully assigned playground/threading-fxmkn-1424549102 to compute-r540-0
  Normal  Pulling    43m   kubelet        Pulling image "docker.io/bitnami/argo-workflow-exec:3.2.6-debian-10-r35"
  Normal  Pulled     38m   kubelet        Successfully pulled image "docker.io/bitnami/argo-workflow-exec:3.2.6-debian-10-r35" in 5m48.454037627s
  Normal  Created    38m   kubelet        Created container wait
  Normal  Started    38m   kubelet        Started container wait
  Normal  Pulled     38m   kubelet        Container image "synelixis/flower-pate:api-v23" already present on machine
  Normal  Created    38m   kubelet        Created container main
  Normal  Started    38m   kubelet        Started container main

```

Figure 27 - Deployed pods for FedPATE training

Then, the logs in Figure 28 show the execution of the training process in FedPATE, as well as its completion.

```

DEBUG flower 2023-03-14 09:44:25,896 | server.py:232 | Training Teachers..
My device is cpu.
Files already downloaded and verified
Files already downloaded and verified
My id is: 0
Round: 1
My device is cpu.
Files already downloaded and verified
Files already downloaded and verified
My id is: 1
Round: 1
100% |██████████| 1563/1563 [14:15<00:00, 1.83it/s]
100% |██████████| 1563/1563 [14:16<00:00, 1.82it/s]
INFO flower 2023-03-14 09:58:48,949 | server.py:241 | Found existing trained teacher models
DEBUG flower 2023-03-14 09:58:48,949 | server.py:242 | Teachers Training finished
DEBUG flower 2023-03-14 09:58:48,949 | server.py:244 | fit round received 2 results and 0 failures
DEBUG flower 2023-03-14 09:58:48,949 | server.py:251 | Evaluate Teachers
DEBUG flower 2023-03-14 09:58:54,471 | server.py:258 | Take predictions from Teachers
DEBUG flower 2023-03-14 09:59:50,403 | server.py:265 | Predictions Taken
DEBUG flower 2023-03-14 09:59:50,611 | server.py:272 | Training Student..
My device is cpu.
Files already downloaded and verified
Files already downloaded and verified
100% |██████████| 563/563 [01:31<00:00, 6.15it/s]
DEBUG flower 2023-03-14 10:02:09,292 | server.py:276 | Student's Training finished
WARNING flower 2023-03-14 10:02:09,391 | pate_2.py:365 | No fit metrics aggregation fn provided
INFO flower 2023-03-14 10:02:09,392 | server.py:159 | FL finished in 1065.2363123251125
INFO flower 2023-03-14 10:02:09,398 | app.py:178 | app fit: losses_distributed {}
INFO flower 2023-03-14 10:02:09,398 | app.py:179 | app fit: metrics_distributed {}
INFO flower 2023-03-14 10:02:09,398 | app.py:180 | app fit: losses_centralized {}
INFO flower 2023-03-14 10:02:09,398 | app.py:181 | app fit: metrics_centralized {}
DEBUG flower 2023-03-14 10:02:09,422 | connection.py:121 | gRPC channel closed
INFO flower 2023-03-14 10:02:09,423 | app.py:101 | Disconnect and shut down
Epoch 1: teacher's train loss 3.4722366333067812, teacher's accuracy 0.09644, test accuracy 0.113
Saving best model for epoch: 1
Updating saved model..
Test loss: 0.14526207756996154, test accuracy: 0.113
DEBUG flower 2023-03-14 10:02:09,426 | connection.py:121 | gRPC channel closed
INFO flower 2023-03-14 10:02:09,426 | app.py:101 | Disconnect and shut down
Epoch 1: student's train loss 1.988318324089593, student's accuracy 0.106, test accuracy 0.109
Saving best model for epoch: 1
Saving 'best_student_model_test.pth' model to testminio MinIO bucket..
Model Saved to testminio MinIO bucket!!!
DEBUG flower 2023-03-14 10:02:09,428 | connection.py:121 | gRPC channel closed
INFO flower 2023-03-14 10:02:09,428 | app.py:101 | Disconnect and shut down
Epoch 1: teacher's train loss 0.4721096456050873, teacher's accuracy 0.10188, test accuracy 0.106
Saving best model for epoch: 1
Directory 'fed-pate/models/1_rounds_2_teachers_1_teacher_epochs_CIFAR10' created!!!
Updating saved model..
Test loss: 11145.585934532643, test accuracy: 0.106
Extracting ./dataset/cifar-10-python.tar.gz to ./dataset
Files already downloaded and verified

```

Figure 28 - Training through FedPATE ran and completed

At the end of the training, the trained model is stored in an external storage. In the context of the IoT-NGIN project, the Minio storage integrated in MLaaS platform is utilized. Figure 29 shows that the final trained student model (i.e., best\_student\_model\_test.pth) is available at MLaaS' MinIO object storage, after the training procedure is finished.

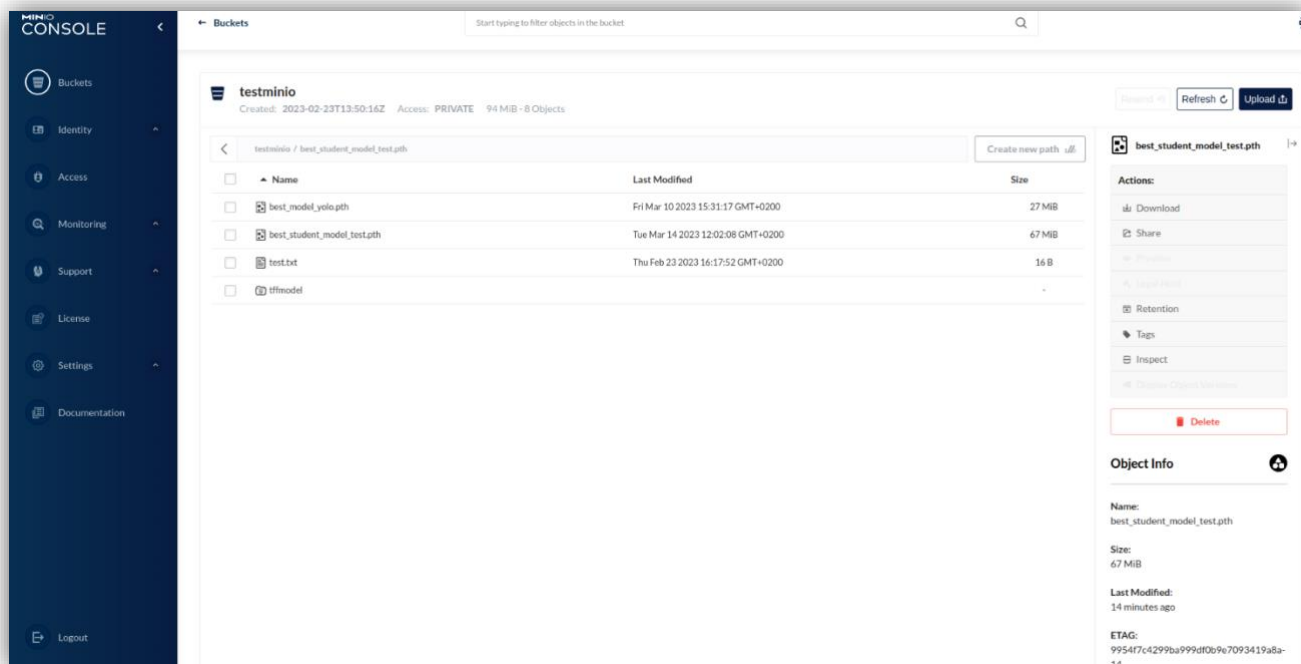


Figure 29 - Final model trained through FedPATE has been stored in MLaaS' Minio storage

### NVIDIA FLARE

Here, an indicative execution of FL training process under real nodes is demonstrated through NVIDIA FLARE. As a first step, the NVIDIA FLARE server (federated server or aggregator) and the observer services have to be started, in order for the clients later to be able to connect to the server and start their training.

#### Server and Observer

Figure 30 shows the request to start the server and the observer of the NVIDIA FLARE framework. The request parameters include:

- --start: Boolean; When set to true, the request asks for the server to start.
- --shutdown: Boolean; When set to true, the request asks for the server to shut down.

D3.4 – ML models sharing and transfer learning implementation

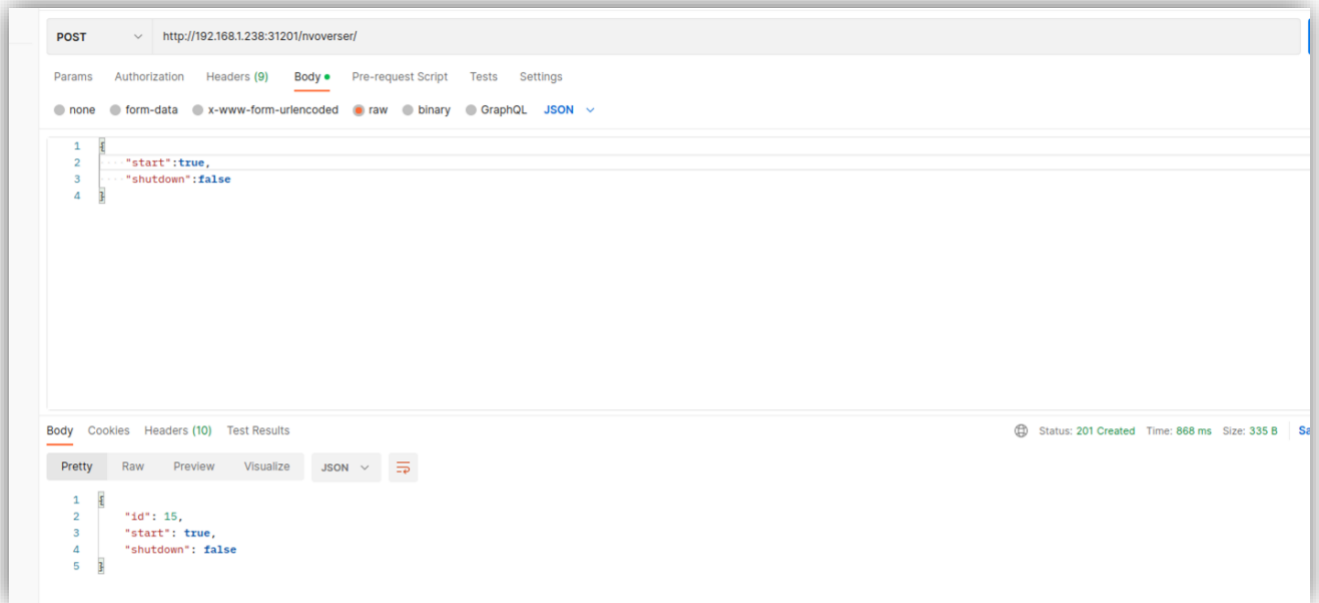


Figure 30 - Starting the NVIDIA FLARE server through the PPFL API

The previous POST request successfully creates the observer and server pods, as shown in Figure 31.

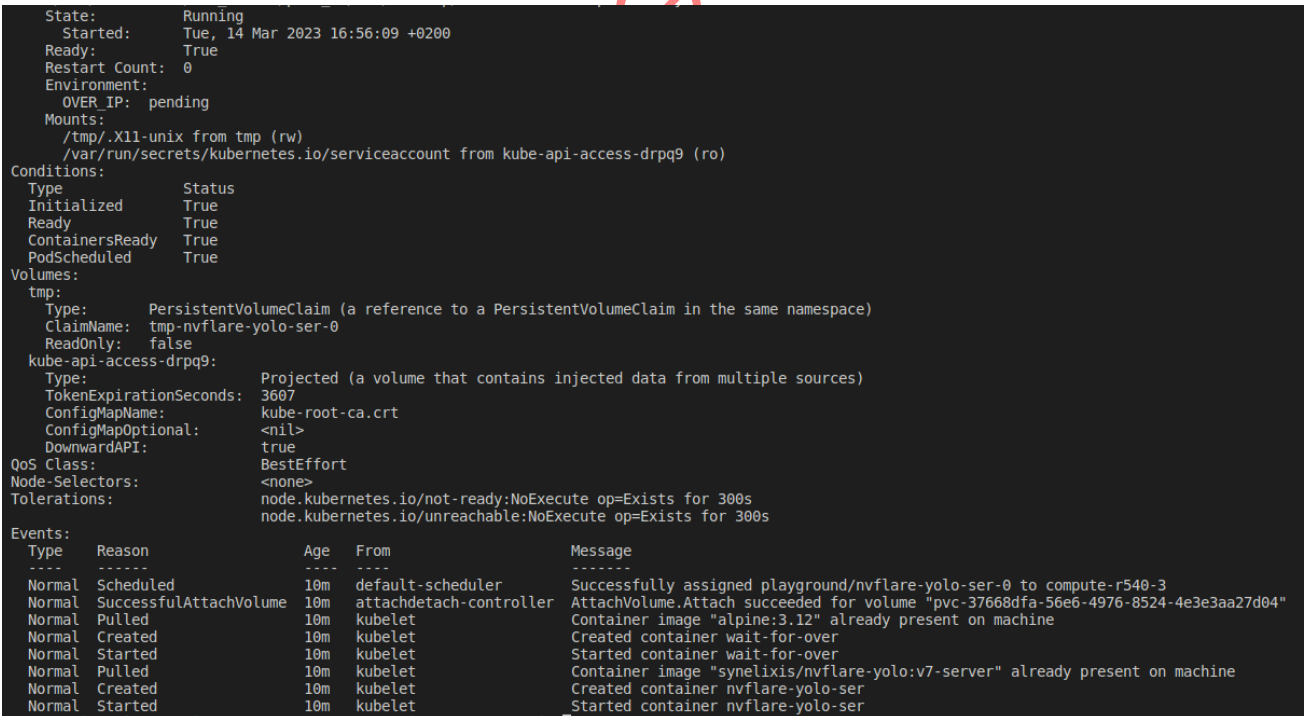


Figure 31 - NVIDIA FLARE server and observer pods

## Clients

The next step for initiating a training session is the instantiation of the clients and their connection to the Federated server already running. As this is a real deployment, the clients are instantiated individually, running the relevant client software. Particularly, each client should download and pull a clients' specific docker image from Synelxis' docker registry called "synelxis/nvflare-yolo:v7-client". Afterwards, in order for the ML model to be trained on each client's private data, a docker volume has to be created in the above docker image as the ML model must have access to these data. This can be done with the following command: "docker run -it --gpus all -v /path/to/client's-private-data:/nvflare-yolo/voc\_pascal\_small synelxis/nvflare-yolo:v7-client /bin/bash", where the flag "--gpus all" is used for gpu acceleration, which also opens an interactive terminal. Finally, in order to connect to the Federated Learning system and thus with the FL server, each client should run the script "./clients/client-*client-index*/startup/start.sh", where *client-index* identifies the client. Figure 32 shows the described procedure for the first client out of the 3 of our FL system. As we can see, the client has made a docker volume in order to give access to its private data and then it successfully connects to the FL system by running the script "./clients/client-1/startup/start.sh", as it is the first client to our FL system. The remaining clients follow similar procedure with indices "2" and "3" set for the *client-index* in the script for the second and the third one, respectively.

```

ubuntu@synal:~$ docker run -it --gpus all -v /home/ubuntu/nvflare_api/client-0:/nvflare-yolo/voc_pascal_small synelxis/nvflare-yolo:v7-client /bin/bash
root@95e553a6128f:/nvflare-yolo# ./clients/client-1/startup/start.sh
root@95e553a6128f:/nvflare-yolo# WORKSPACE set to /nvflare-yolo/clients/client-1/startup/..
PYTHONPATH is /local/custom:
start fl because of no pid.fl
new pid 16
Waiting for SP...
2023-03-22 08:53:27,538 - FederatedClient - INFO - Got the new primary SP: ser:8002
2023-03-22 08:53:28,558 - FederatedClient - INFO - Successfully registered client:client-1 for project AGV_Yolov5. Token:55e5d247-0016-4693-b592-f170fa45eff
6 SSID:961b5623-b70b-442f-a4d7-2b9e5ee825c3

```

Figure 32 - Running NVIDIA FLARE client software for client-1

## Training

After the server has been created and the clients have connected to it successfully, a training process may be triggered through a POST request with the parameters shown in Figure 33.

D3.4 – ML models sharing and transfer learning implementation

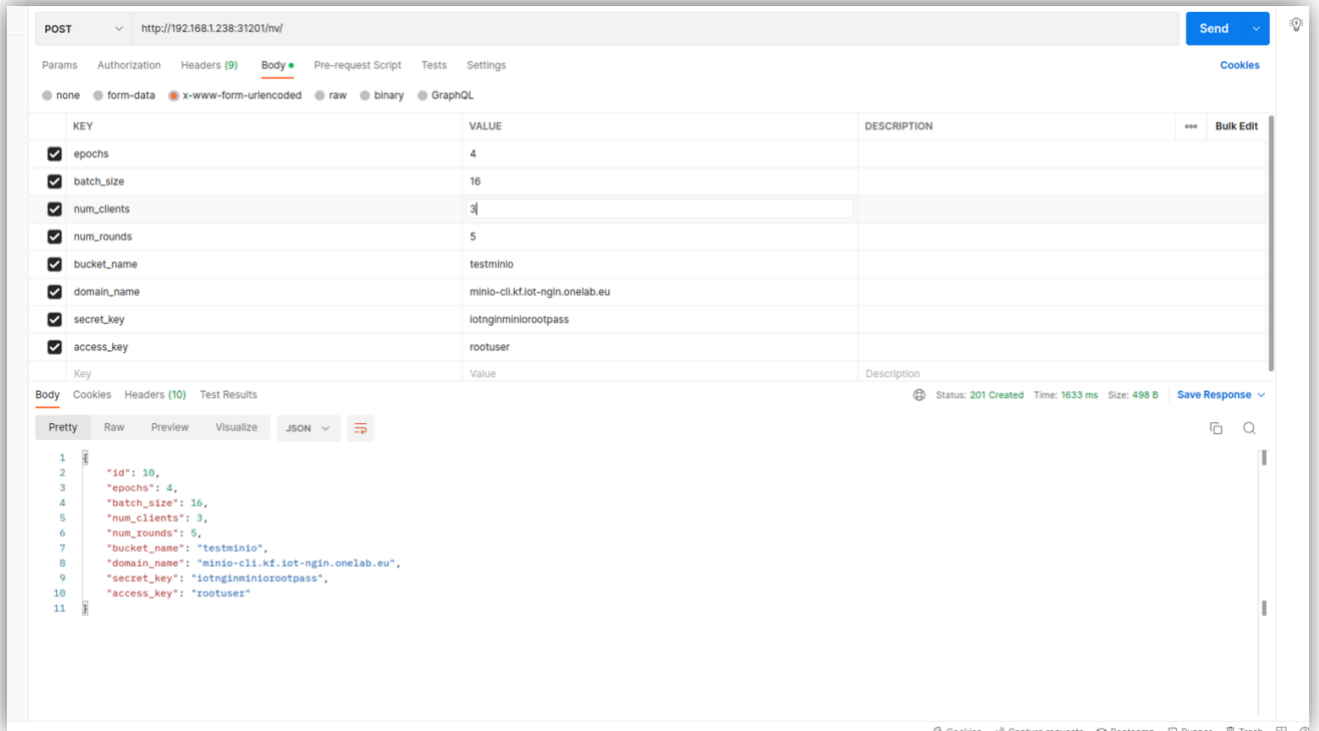


Figure 33 - Indicative parameter setting for starting a training process through NVIDIA FLARE

Assuming valid request parameter setting, the FL controller for NVIDIA FLARE starts the model training.

Indicatively, Figure 34 depicts the training process in one of the three clients.

```

2023-03-16 11:54:20,474 - ClientRunner - INFO - [run=46db9a03-931f-4131-ba79-f488e369c56c, peer=AGV_Yolov5, peer_run=46db9a03-931f-4131-ba79-f488e369c56c, task_name=validate, task_id=93ae3046-ce3c-49aa-8d7f-1e4aad776771]: invoking task executor <class 'yolov validator.YoloValidator'>
[run=46db9a03-931f-4131-ba79-f488e369c56c, peer=AGV_Yolov5, peer_run=46db9a03-931f-4131-ba79-f488e369c56c, task_name=validate, task_id=93ae3046-ce3c-49aa-8d7f-1e4aad776771]: invoking task executor <class 'yolov validator.YoloValidator'>
Saving trained model to MinIO..
Model Saved to MinIO bucket!!
      Class  Images  Labels    P      R    mAP@0.5  mAP@0.5:0.95: 100% |██████████| 5/5 [00:02<00:00, 2.49it/s]
2023-03-16 11:55:02,181 - yolov5 - INFO - all      133      183      0.72  0.852  0.826  0.562
      all      133      183      0.72  0.852  0.826  0.562
2023-03-16 11:55:02,501 - YoloValidator - INFO - [run=46db9a03-931f-4131-ba79-f488e369c56c, peer=AGV_Yolov5, peer_run=46db9a03-931f-4131-ba79-f488e369c56c, task_name=validate, task_id=93ae3046-ce3c-49aa-8d7f-1e4aad776771]: Accuracy when validating client-1's model on client-1s data: (0.7198668262252432, 0.8524590163934426, 0.8256225006309411, 0.5619670252927114, 0.02410305291414261, 0.000977161107584834, 0.0)
[run=46db9a03-931f-4131-ba79-f488e369c56c, peer=AGV_Yolov5, peer_run=46db9a03-931f-4131-ba79-f488e369c56c, task_name=validate, task_id=93ae3046-ce3c-49aa-8d7f-1e4aad776771]: Accuracy when validating client-1's model on client-1s data: (0.7198668202252432, 0.8524590163934426, 0.8256225006309411, 0.5619670252927114, 0.02410305291414261, 0.000977161107584834, 0.0)
2023-03-16 11:55:02,502 - ClientRunner - INFO - [run=46db9a03-931f-4131-ba79-f488e369c56c, peer=AGV_Yolov5, peer_run=46db9a03-931f-4131-ba79-f488e369c56c, task_name=validate, task_id=93ae3046-ce3c-49aa-8d7f-1e4aad776771]: finished processing task
[run=46db9a03-931f-4131-ba79-f488e369c56c, peer=AGV_Yolov5, peer_run=46db9a03-931f-4131-ba79-f488e369c56c, task_name=validate, task_id=93ae3046-ce3c-49aa-8d7f-1e4aad776771]: finished processing task
2023-03-16 11:55:02,503 - FederatedClient - INFO - Starting to push execute result.
Starting to push execute result.
2023-03-16 11:55:02,503 - Communicator - INFO - Send submitUpdate to AGV_Yolov5 server
Send submitUpdate to AGV_Yolov5 server

```

Figure 34 - Logs of the training process at the client side.

After the training procedure has finished, the final model is stored in the Minio instance of the MLaaS platform, as shown in Figure 35.

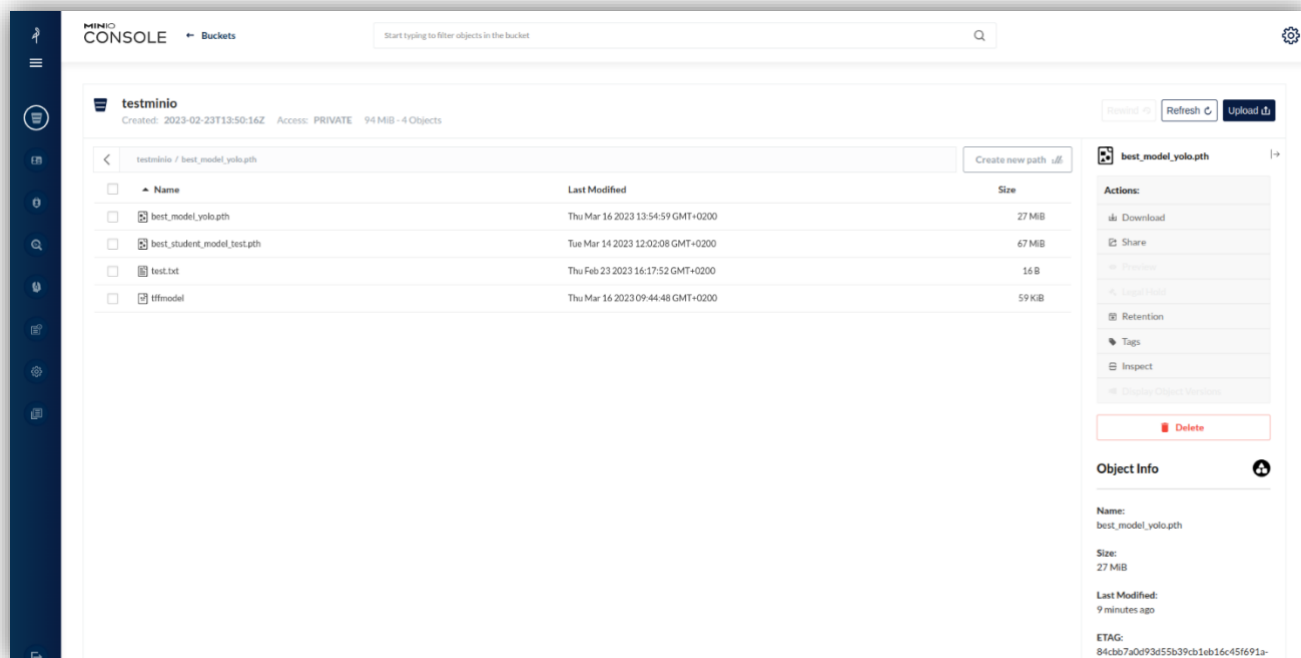


Figure 35 - Final model of NVIDIA FLARE training stored in Minio model storage component of the MLaaS platform

## TensorFlow Federated

Last, but not least, the instantiation of the FL training process (simulation mode) through TensorFlow Federated is presented. First, the training parameters are provided in the POST request for triggering the training, under the /tff endpoint, as shown in Figure 36. Specifically, the following parameters must be set:

- --epochs: int, number of epochs to train.
- --batch\_size: int, batch size (32 or 16).
- --num\_clients: int, the number of clients.
- --num\_rounds: int, number of rounds for FL.
- --bucket\_name: str, the bucket in MinIO where model will be stored.
- --domain\_name: str, the domain where the model will be stored.
- --secret\_key: str, secret key for accessing MinIO.
- --access\_key: str, access key for accessing MinIO.

Assuming valid parameter setting, this request will result in the relevant pod being deployed in the cluster via the FL Controller.

D3.4 – ML models sharing and transfer learning implementation

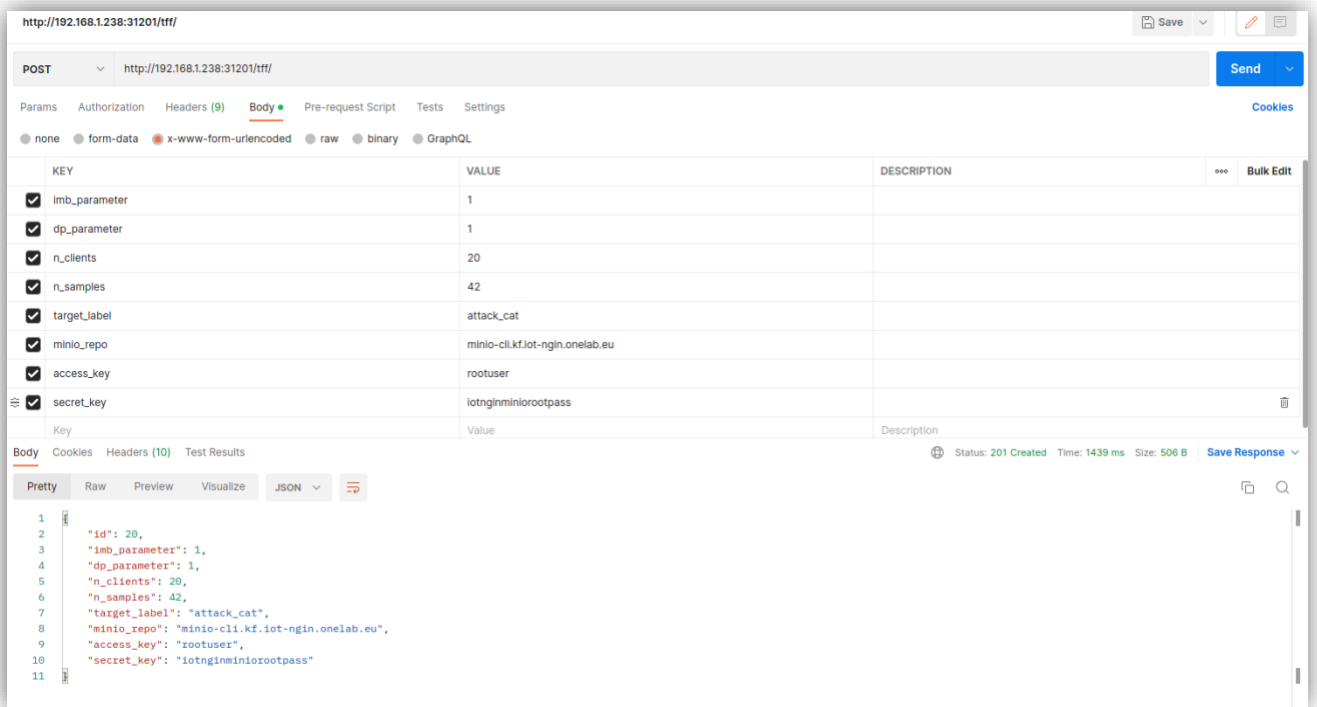


Figure 36 - Instantiating TFF training through the PPFL API

The training process gets started and completed, as indicated in the logs of Figure 37.

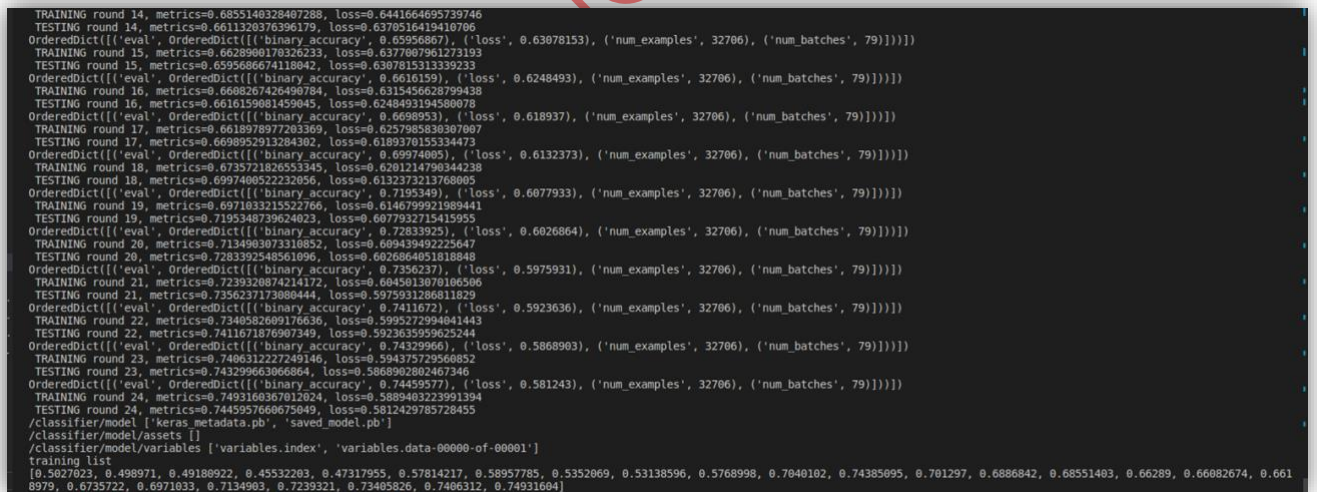


Figure 37 - The training process through TFF has been completed

As with the other two FL frameworks, the final model gets stored in the Minio component of the MLaaS platform, as depicted in Figure 38.



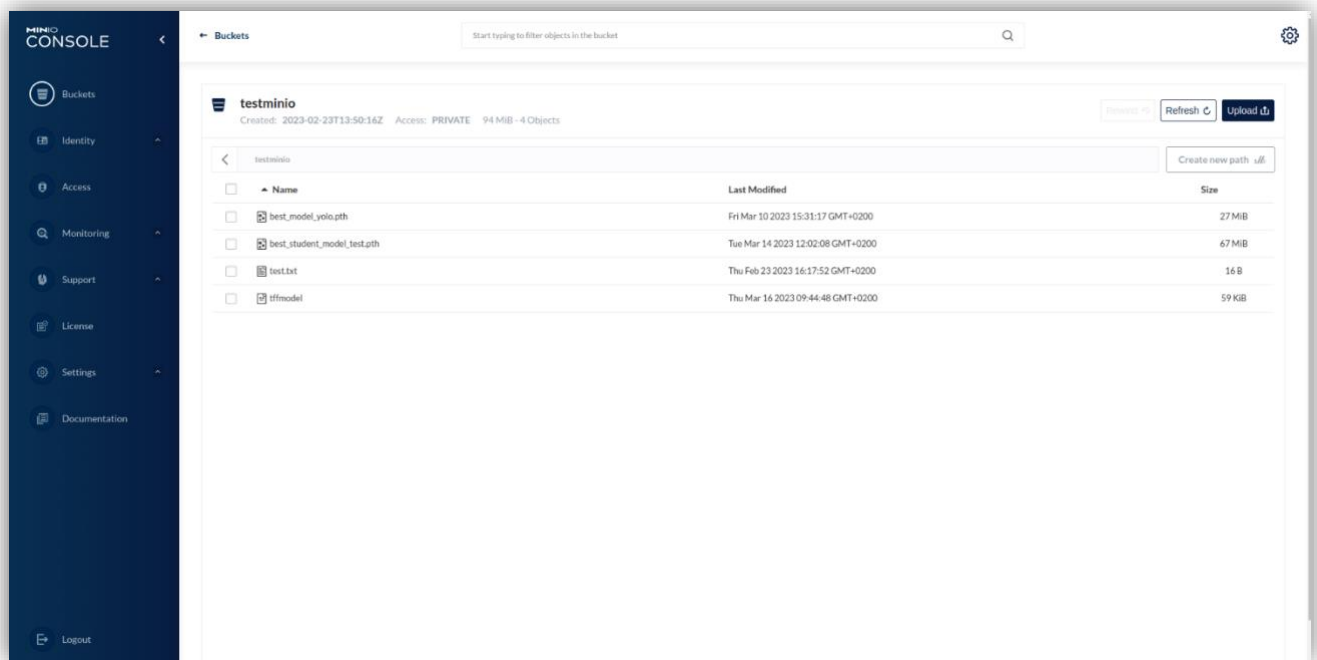


Figure 38 - Final model trained through TFF stored in MLaaS' Minio storage

## 5.4 Polyglot Model Sharing Framework

This section describes the way to launch the Model Sharing framework in a computational environment. A similar procedure has been followed to install the framework in MLaaS, by using their K8s manifests.

There are two main ways of running the Model Sharing platform in a local environment:

1. Running each service's project in the host OS
2. Running each service as a container

Each services provides a container image, which can be run using [Podman \[64\]](#). The source Dockerfiles including the instructions of the container images for each service are located in their respective containing directories, as previously described in the implementation section. The project's container image registry offers a visual interface (Figure 39) for inspecting the hosted artifacts, which can be accessed from the project's GitLab repository [65].



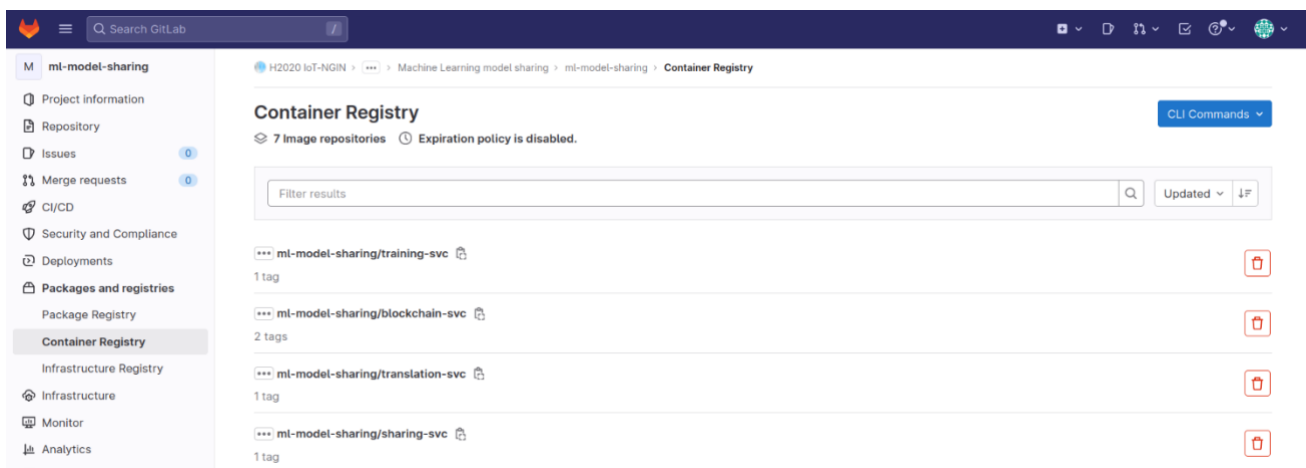


Figure 39 - Project's GitLab container registry interface

The environment variables required by each service are documented in their respective `.env.sample` files (Listing 4).

`plain`

```
DB_HOST=localhost
DB_USER=test
DB_PASS=test
DB_NAME=test
DB_PORT=5432
MINIO_ENDPOINT=127.0.0.1:9000
MINIO_ACCESS=w88MSWdgwM51PBN1
MINIO_SECRET=RB1cEFACZzzMx59JgGapIJ7iVnkA3B5S
MINIO_DATASETS_BUCKET=datasets
MINIO_MODELS_RES_BUCKET=models_res
MINIO_SECURE=false
BLOCKCHAIN_ENDPOINT=http://localhost:9005
DOWNLOAD_DIR=/tmp/model-sharing/download
```

Listing 4 - Model Sharing service's `.env.sample` file

In this section, we will detail how to setup a local environment.

## 5.4.1 External dependencies

### 5.4.1.1 PostgreSQL instance

You can run a PostgreSQL instance locally with Podman using the following command:

`bash`

```
podman run --name postgres-model-sharing -e POSTGRES_PASSWORD=<password> -e
POSTGRES_USER=<user> -e POSTGRES_PASSWORD=<pass> -e POSTGRES_DB=<db_name> -d -p
8080:8080 postgres
```

### 5.4.1.2 ConsenSys Quorum Instance

Please, refer to the [following official guide \[66\]](#) for running a Quorum instance locally, using a local Kubernetes cluster (e.g. [minikube \[67\]](#)).

bash

```
podman run --name postgres-model-sharing -e POSTGRES_PASSWORD=<password> -e POSTGRES_USER=<user> -e POSTGRES_PASSWORD=<pass> -e POSTGRES_DB=<db_name> -d -p 8080:8080 postgres
```

### 5.4.1.3 MinIO Instance

You can run a MinIO instance locally with Podman using the following command:

bash

```
podman run -d -p 9000:9000 -p 9001:9001 quay.io/minio/minio server /data --console-address ":9001"
```

### 5.4.1.4 Argo Workflows Instance

Please, refer to the official guide [68] for running an Argo Workflows instance locally, using a local Kubernetes cluster (e.g., minikube).

## 5.4.2 Running Services Locally

As the project has been designed following a microservices-based approach, it is possible to launch the platform's components individually; however, please note that in most scenarios the platform's features are result of the collaboration with multiple services, and integration tests will usually require multiple services to be running concurrently.

### 5.4.2.1 Running Services in Host OS

For each service, it's recommended to setup a Python virtual environment in order to avoid conflicts in the project's dependencies. The dependencies required by each service are compiled in their respective requirements.txt, and can be installed using the pip CLI tool.

The recommended way to setup the virtual environment and install the dependencies of the services is by using Poetry. For this, execute the following command in the service's root directory:

bash

```
poetry install
```

By default, by running this command, Poetry will create a virtual environment, installing on it all of the required dependencies. Additionally, upon completion of the command, Poetry will activate the virtual environment.

When running the services in the host OS (not containerized), they will look, by default, for a `.env` file in their root directory. This `.env` file must conform to the structure defined in the service's `.env`, `.sample` file.

You can launch any of the services using the following command:

`bash`

```
python -m uvicorn services.<service>.app.main:app --port <port> --reload --log-level trace
```

### 5.4.2.2 Running Services in Containers

You can run any of the services containers locally with Podman using the following command:

`bash`

```
podman run -d -p <port>:<port> -env-file .env registry.gitlab.com/h2020-iot-nginx/enhancing_iot_intelligence/t3_4/ml-model-sharing/<service>  
<service_name_podman>
```

Note that we are providing the service with the required environment variables by pointing to a `.env` file; this `.env` file must conform to the structure defined in the service's `.env.sample` file.

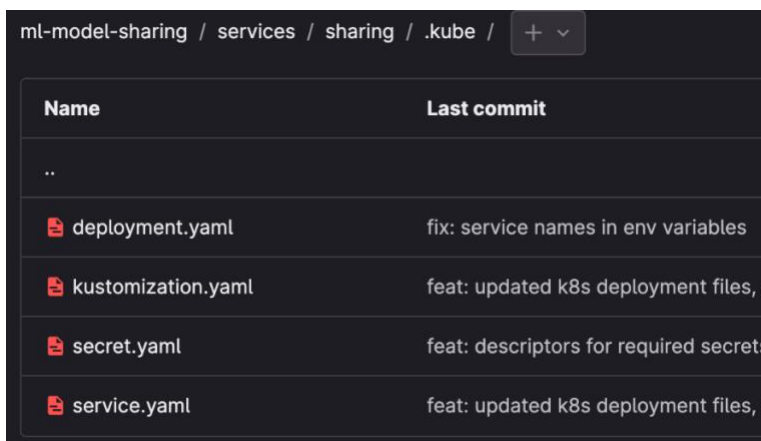
### 5.4.2.3 Invoking the HTTP REST APIs

Please, refer to section 4.3 (Implementation for IoT-NGIN LLs) for an in-depth, guided demo use case in which the process for invoking the services' REST APIs is further detailed.

### 5.4.3 Deployment on Kubernetes Clusters

In order to deploy any of the framework's services in a Kubernetes cluster, it is recommended to use the provided Kubernetes manifests as a starting point.

Each service's Kubernetes manifest files are stored in their containing implementation directory in the project's GitLab repository (Figure 40), on the `.kube` directory.



Name	Last commit
..	
deployment.yaml	fix: service names in env variables
kustomization.yaml	feat: updated k8s deployment files, c
secret.yaml	feat: descriptors for required secrets
service.yaml	feat: updated k8s deployment files, c

Figure 40 - Model Sharing service's Kubernetes manifests

Please, note that it is required to specify the correct values for the environment variables required by each service for their deployment. This environment variables are located in their respective deployment.yaml Kubernetes manifest template files (Listing 5).

### yaml

```
spec:
  containers:
    - image: registry.gitlab.com/h2020-iot-
      nginx/enhancing_iot_intelligence/t3_4/ml-model-sharing/sharing:latest
      name: sharing-service
      ports:
        - containerPort: 80
      resources: {}
      env:
        # MLAAS POSTGRESQL
        - name: DB_HOST:
          value: localhost
        - name: DB_USER:
          value: test
        - name: DB_PASS:
          valueFrom:
            secretKeyRef:
              name: sharing
              key: DB_PASS
        - name: DB_NAME:
          value: test
        - name: DB_PORT:
          value: 5432
```

Listing 5 - Excerpt from Model Sharing service's deployment.yaml Kubernetes manifest template file

Additionally, some of the environmental variables are specified using Kubernetes secrets, following Kubernetes's security best practices [69]. It is therefore required to create these secrets in the target Kubernetes cluster. A template for these secrets is provided for all the required secrets (Listing 6).

yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: sharing
  namespace: mlaas #Same namespace services/pods are deployed
type: Opaque
stringData:
  # MLAAS POSTGRESQL
  DB_PASS: test
  # MLAAS MINIO
  MINIO_SECRET: RB1cEFACZzzMx59JgGapIJ7iVnkA3B5S
```

Listing 6 - Model Sharing service's secret.yaml Kubernetes manifest template file

DRAFT - PENDING ECU

## 6 Conclusions

This document is the last one of a series that have been reporting the development of the IoT-NGIN MLOps platforms and services, including the MLaaS platform, the PPFL platform, and a number of frameworks and services that have been built and delivered with MLaaS, namely the Online Learning service, and the Polyglot Model Sharing service. While the MLaaS architecture and technical specification of its reference implementation was described in D3.3, this document has reported the main progress beyond D3.3 on the development of the final versions of the Online Learning service, the first version of the RL-based optimization, the PPFL platform and the Polyglot Model Sharing framework.

The Online Learning service has been reimplemented to take advantage of the further flexibility of KServe pipeline, to foster its reusability across multiple inference scenarios for other models. Additionally, XAI explainers have been injected, aiming to provide insights to explain how the trained models learn from features. A new monitoring platform has been integrated, featuring the detection of data drift and accounting for model learning error over the time. Model transfer techniques have been adopted to re-train models to infer similar knowledge, as evaluated for energy demand forecasting in the Smart Energy LL. Another example of exposing a model inference service with MLaaS for WP4 object detection has been described as well.

The PPFL framework has been extended with a common entry-point API that gives a harmonized operational interface to the different integrated FL frameworks. Moreover, a CI/CD Cloud based approach for FL task deployment has been followed for the API implementation.

The model sharing framework has been specified both functionally and technically, and details of its micro-service implementation have been given. Evaluation of the framework has been conducted in a dedicated use case that has been conceived for such purpose.

The software implementations of the components presented in this deliverable are offered as open source on the project's page on the public Gitlab repository, at [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_intelligence/](https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/). Details about installation and usage are also provided in this report for convenience of interested audience.

As future work, all the components described in this report will be further adopted and evaluated in the IoT-NGIN LL use cases, as well as with the open call projects, during the last evaluation period of the project and will be reported in forthcoming D6.3 and D7.4 deliverables.

## 7 Annex Components and Interfaces for Polyglot Model Sharing

### 7.1 Model Sharing service

The main purpose of the Model Sharing service is coordinating the storage and retrieval of all the assets in the platform (i.e., datasets and machine learning models). To achieve this, it will exclusively manage access to the platform's object storage instance, and coordinates with the Blockchain service for the deployment, verification and retrieval of smart contracts in the platform's blockchain instance. Additionally, it coordinates the scheduling of model training jobs for newly registered machine learning models by coordinating with the Model Training service.

#### 7.1.1 HTTP REST API Operations

The Model Sharing service exposes the following operations (see Table 10) through its HTTP REST API:

Table 10 - Model Sharing service HTTP REST API operations

Method	Route	Description
POST	/dataset	Register and store a new dataset in the platform
POST	/model	Register and schedule the training of a machine learning model
GET	/model/{model_id}	Retrieve a registered model upon successful training job
GET	/dataset/{dataset_id}	Retrieve a registered dataset
PUT	/model/{model_id}	Stores the resulting model from the model training job, and makes it available for retrieval
GET	/model/{model_id}/metadata	Retrieves the metadata stored in the blockchain for a registered model

### 7.1.2 Dataset Registration

In Figure 41, we describe the implementation of the dataset registration operation, which involves the storage of the dataset artifact on the platform's object storage instance, as well as deployment of its associated smart contract in the platform's blockchain instance through the Blockchain service.

The dataset contents are stored in the object storage instance, on the dataset bucket, identified by the UUID assigned by the Model Sharing service. Notice that all the original file's metadata, including filename and content type are not preserved upon its storage on the object storage instance. This original metadata is stored externally, on the MLaaS Relational Database instance.

The additional metadata required for ensuring the immutability of the registered dataset is stored in the blockchain instance. This metadata can be retrieved at any time by the other services through the Blockchain Service.

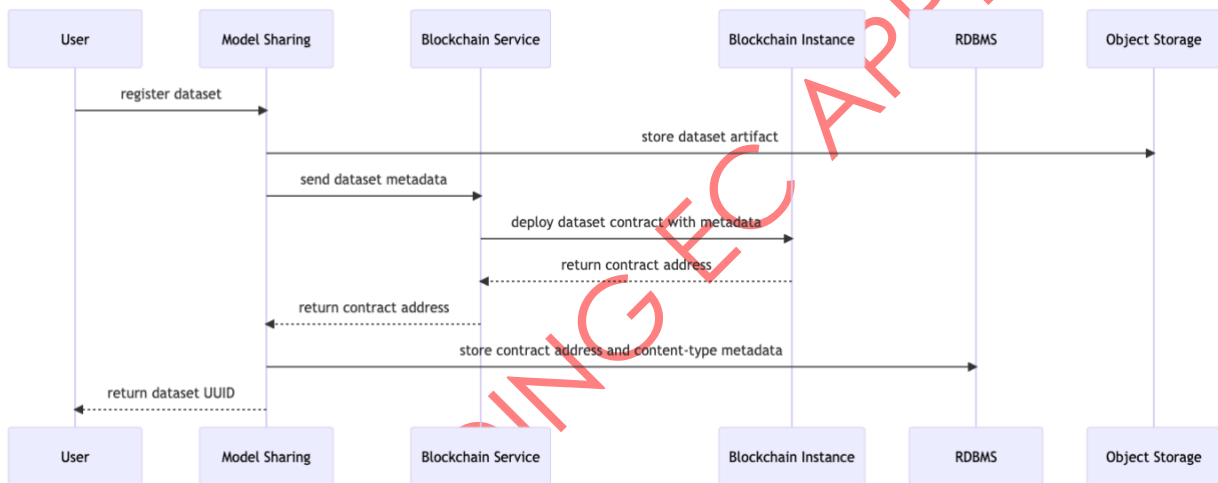


Figure 41 - Sequence diagram of the Model Sharing service's dataset registration operation

### 7.1.3 Model Registration, Training and Retrieval

In Figure 42, we describe the implementation of the model registration operation, which involves the storage of its metadata in the Blockchain instance, and model training job scheduling via the Model Training service. Upon the successful execution of the training job, the resulting artifacts are registered, and the model's smart contract is verified, updating its stored metadata. From this moment, the model can be retrieved by the end users.

Upon successful training of the registered model, the resulting artifacts are stored in the object storage instance, on the model bucket, identified by the UUID assigned by the Model Sharing service. Notice that all the original artifact's metadata, including filename and content type are not preserved upon its storage on the object storage instance. This original metadata is stored externally, on the MLaaS Relational Database instance.

The additional metadata required for ensuring the reproducibility of the training results, as well as the immutability of the resulting artifacts, is stored in the blockchain instance. This metadata can be retrieved at any time by the other services through the Blockchain Service.



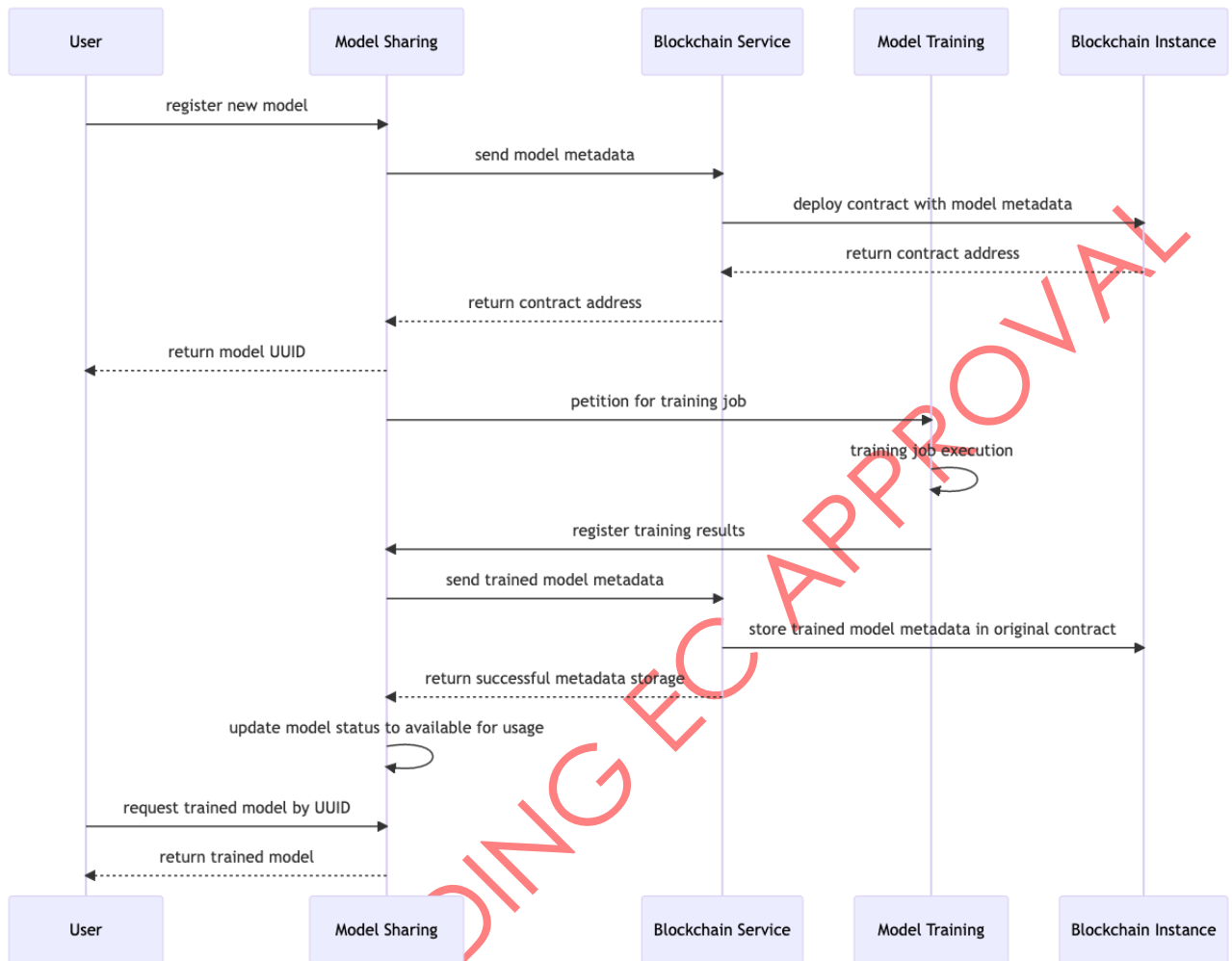


Figure 42 - Sequence diagram of the Model Sharing service's model registration and retrieval

### 7.1.4 Model Metadata Retrieval

In Figure 43, we can visualize the metadata retrieval operation implementation, in which we retrieve the metadata of a registered machine learning model, which is stored in a smart contract deployed in the project's blockchain instance – which we access through the Blockchain service.

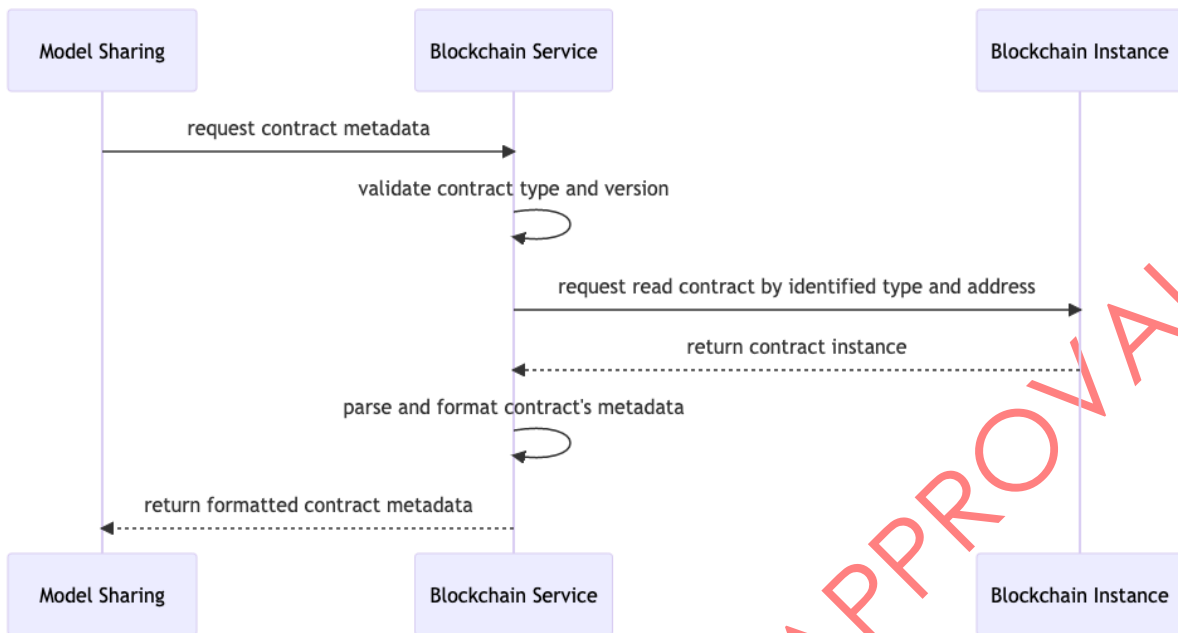


Figure 43 - Sequence diagram of the Model Sharing service's model metadata retrieval operation

## 7.2 Blockchain service

The Blockchain service provides the interface for interacting with the blockchain instance, for all the other platform components.

This service is responsible for managing all the information registered in the blockchain instance, in the form of smart contracts.

These smart contracts are deployed into the blockchain for all registered assets in the platform (datasets and models).

For each asset type, there exists a smart contract definition that contains the machine code and rules for interacting with the deployed contracts. This smart contract definition is implemented in Solidity [70], a high-level language used for smart contract definitions, which compiles to EVM executable bytecode (ABI – Application Binary Interface).

### 7.2.1 Smart Contract Interfaces

The metadata stored depends on the contract interface; in our platform, we have two contract interfaces:

- Dataset contract [16]
- Machine learning model contract

The dataset contract (Table 11) stores the following pieces of metadata:

Table 11 - Dataset smart contract metadata specification

Field	Data Type	Description	Example
organization_id	string	Unique identifier of the dataset's owner organization	5d066d58-06de-47ab-a2f6-c8f413e21947
samples_dimension	string	Dataset sample dimension	(128,128,3)
size_bytes	int256	Stores the metadata of a resulting model training job	1840000
hash	string	MD5 hash of the dataset contents	0cc175b9c0f1b6a831c399e269772661

The dataset contract (Table 12) implements the following operations:

Table 12 - Dataset smart contract operations

Method	Returns	Description	Example
get_dataset_params	string memory	Returns the UUID of the associated dataset	5d066d58-06de-47ab-a2f6-c8f413e21947

The machine learning model contract (Table 13) stores the following pieces of metadata:

Table 13 - Model smart contract metadata specification

Field	Data Type	Description	Example
organization_id	string	Unique identifier of the machine learning model developer's company	5d066d58-06de-47ab-a2f6-c8f413e21947
developer_id	string	Unique identifier of the machine learning model developer	5d066d58-06de-47ab-a2f6-c8f413e21947
dataset_id	string	Unique identifier of the dataset specified for the model training job	5d066d58-06de-47ab-a2f6-c8f413e21947
train_image_hash	string	MD5 hash of the model's training container image	0cc175b9c0f1b6a831c399e269772661
res_hash	string	MD5 hash of the resulting artifacts from the model training job	0cc175b9c0f1b6a831c399e269772661

The machine learning model contract (Table 14) implements the following operations:

Table 14 - Model smart contract operations

Method	Returns	Description
get_model_params	(string memory, string memory, string memory, string memory)	Returns all model-related metadata stored in the contract
get_data_params	string memory	Returns the UUID of the associated dataset
verify	Void	Stores the MD5 hash of the resulting artifacts from the model training job

## 7.2.2 HTTP REST API Operations

The Blockchain service exposes the following operations through its HTTP REST API (Table 15):

Table 15 - Blockchain service HTTP REST API operations

Method	Route	Description
POST	/contract	Deploy a smart contract into the blockchain
GET	/contract/{address}	Fetch a deployed contract's metadata
POST	/contract/{address}/verify	Stores the metadata of a resulting model training job

*Note: the REST API exposed by this service is not intended for the end users of the platform, but only to be accessed directly by the rest of the services of the system.*

We will now introduce a more in-depth explanation for the implementation of this service's operations.

## 7.2.3 Dataset and Model Contract Deployment

In these diagrams (Figure 44, Figure 45), we detail the deployment of the contract of a dataset or model registered in the platform, in the context of the registration of a new dataset or model in the platform. Note that both processes differ, especially due to the extra step involved in the model registration process (further detailed in section 4.2.2.2.5), the model verification, which requires an update of the stored metadata in its smart contract.

D3.4 – ML models sharing and transfer learning implementation

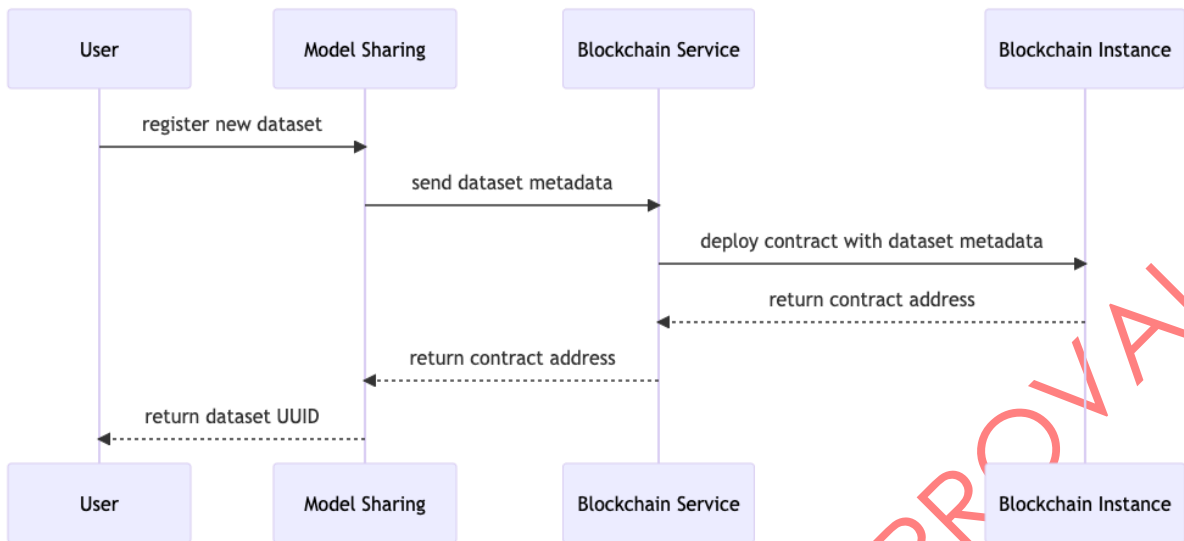


Figure 44 - Sequence diagram of a dataset's contract deployment operation

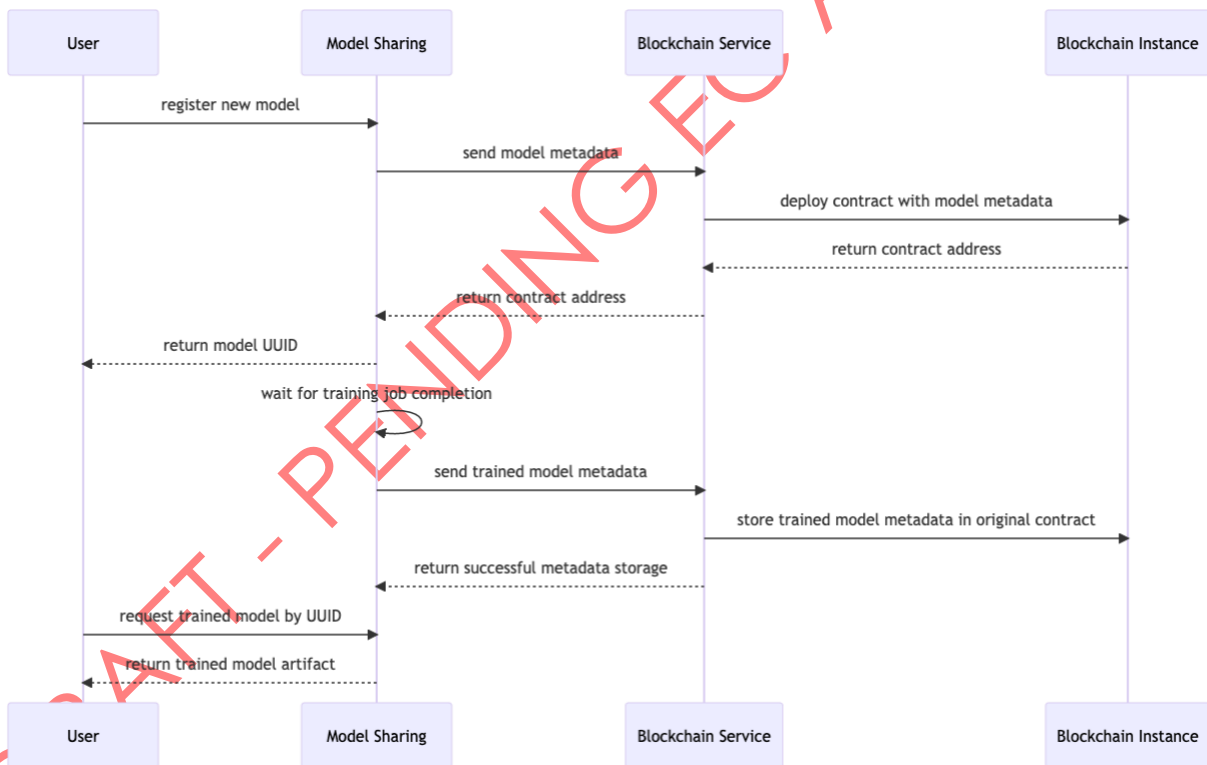


Figure 45 - Sequence diagram of a model's contract deployment operation

## 7.2.4 Contract metadata request

In Figure 46, we detail the metadata request operation, performed for contracts deployed in the blockchain.

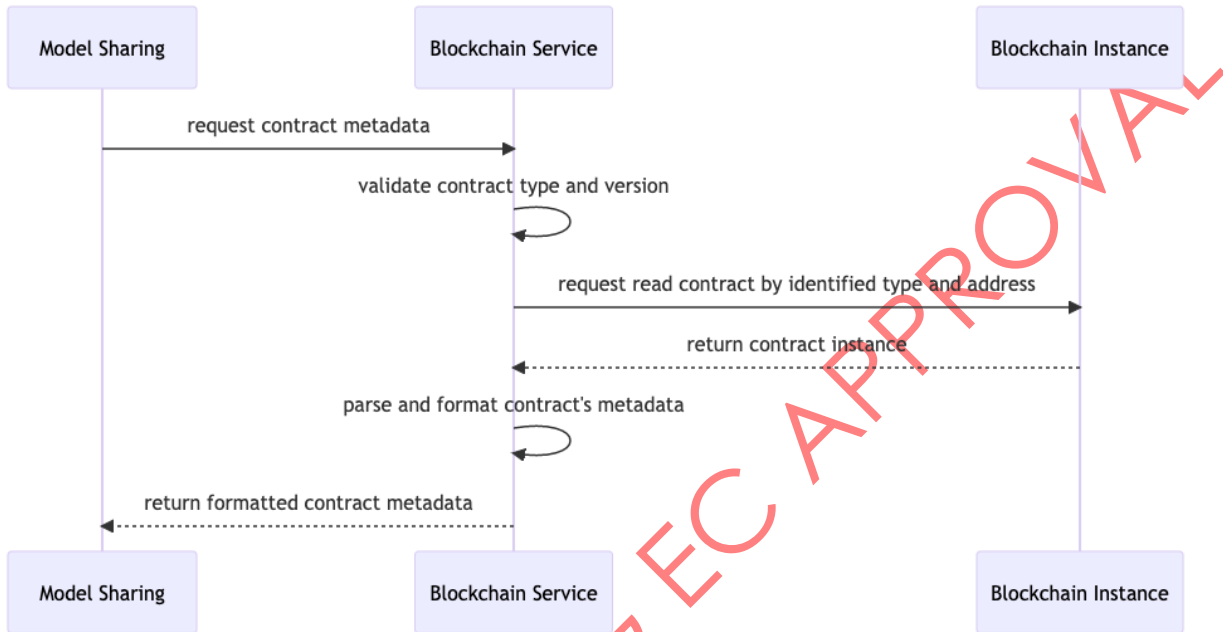


Figure 46 - Sequence diagram of the Blockchain Service's contract metadata request operation

## 7.2.5 Model verification

In Figure 47, we can observe the verification step of a model training job, in which metadata of the resulting artifacts from the model training job is stored in the original smart contract of the machine learning model.

The verification step is handled by the verification operation of the Blockchain Service. This operation invokes a function in the deployed smart contract of a registered model, which stores the MD5 digest (i.e., hash) of the resulting artifact from the model training job.

This function is implemented on the smart contract's definition and ensures that the digest of the resulting model training artifact is only stored once (i.e., the field is immutable).

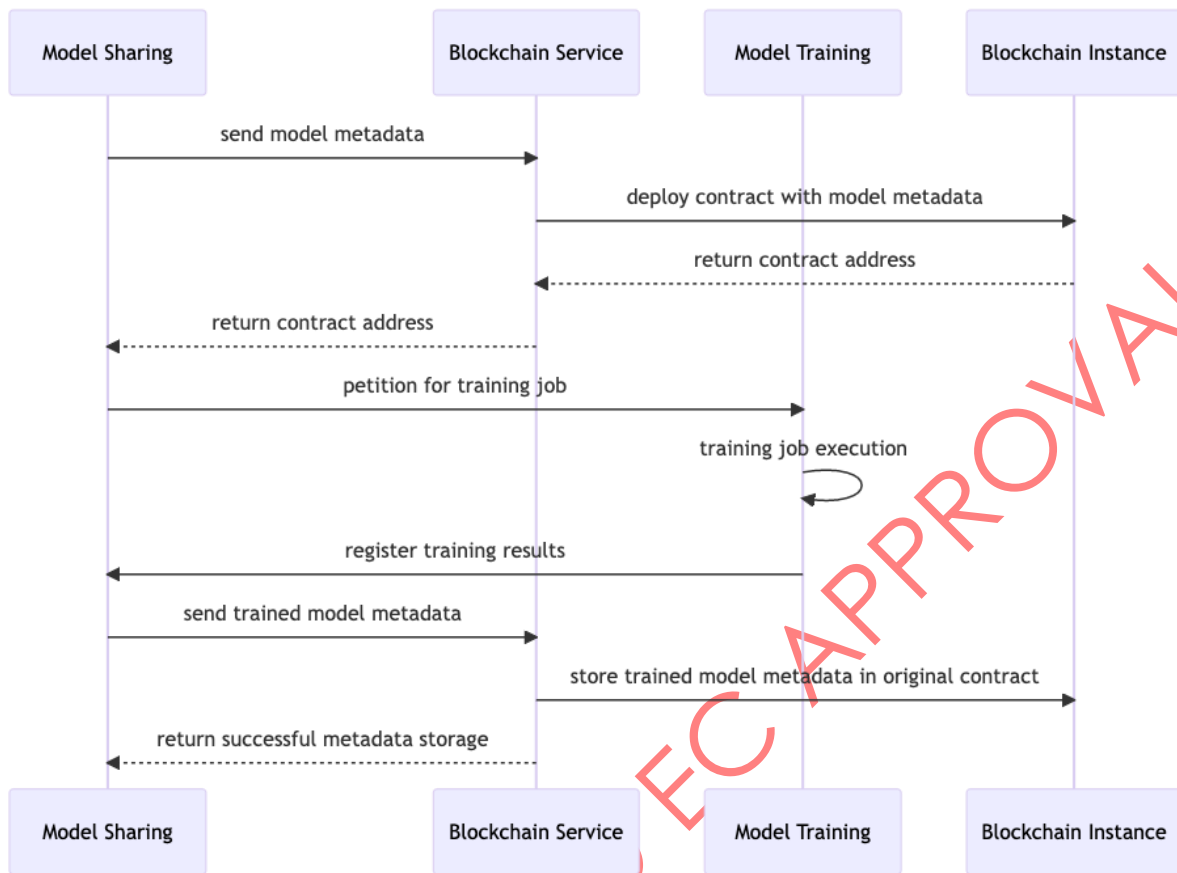


Figure 47 - Sequence diagram of the Blockchain service's model verification operation

## 7.3 Model Training service

The purpose of the Model Training service is to schedule the execution of training jobs for new models registered in the system. This training jobs are sequential executions of a workflow consisting of three container images, a pre-training container that provides the training container with the appropriate datasets, the training container provided by the users, and a post-training container that registers the training results in the platform.

### 7.3.1 HTTP REST API Operations

The Model Training service exposes the following operations through its HTTP REST API (Table 16).

Table 16 - Model Training service HTTP REST API operations

Method	Route	Description
POST	/train/{model_id}	Schedule a model training job in the platform

We will now introduce a more in-depth explanation for the implementation of this service's operations.



### 7.3.2 Model training request

In Figure 48, we can observe the interactions between the platform's services, from the moment that an end user registers a new machine learning model in the platform, up until the model training process is completed successfully, and the end user can retrieve the resulting machine learning model.

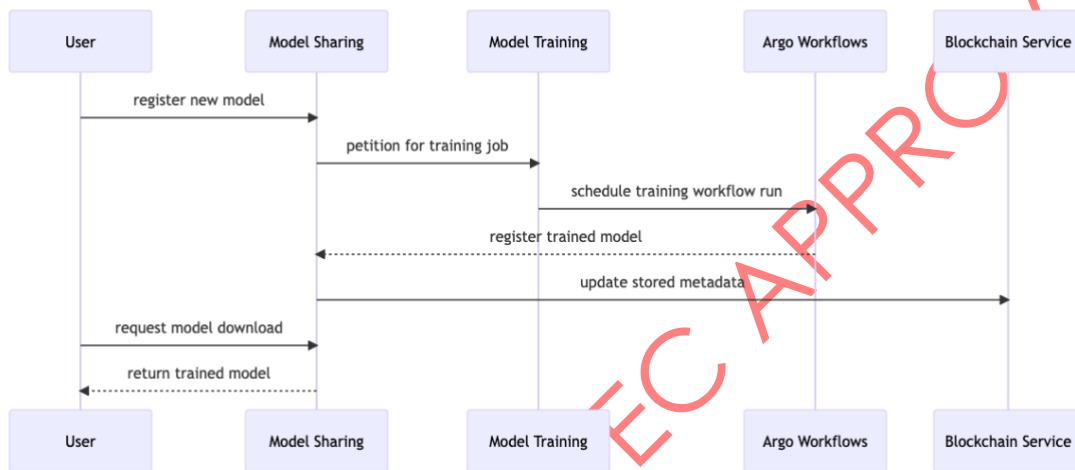


Figure 48 - Sequence diagram of the Model Training service's training request operation

In Figure 49, we further detail the internal workflow of a model training job, coordinated with Argo Workflows, and the steps taken in a model training job.

DRAFT - PENDING EC APPROVAL

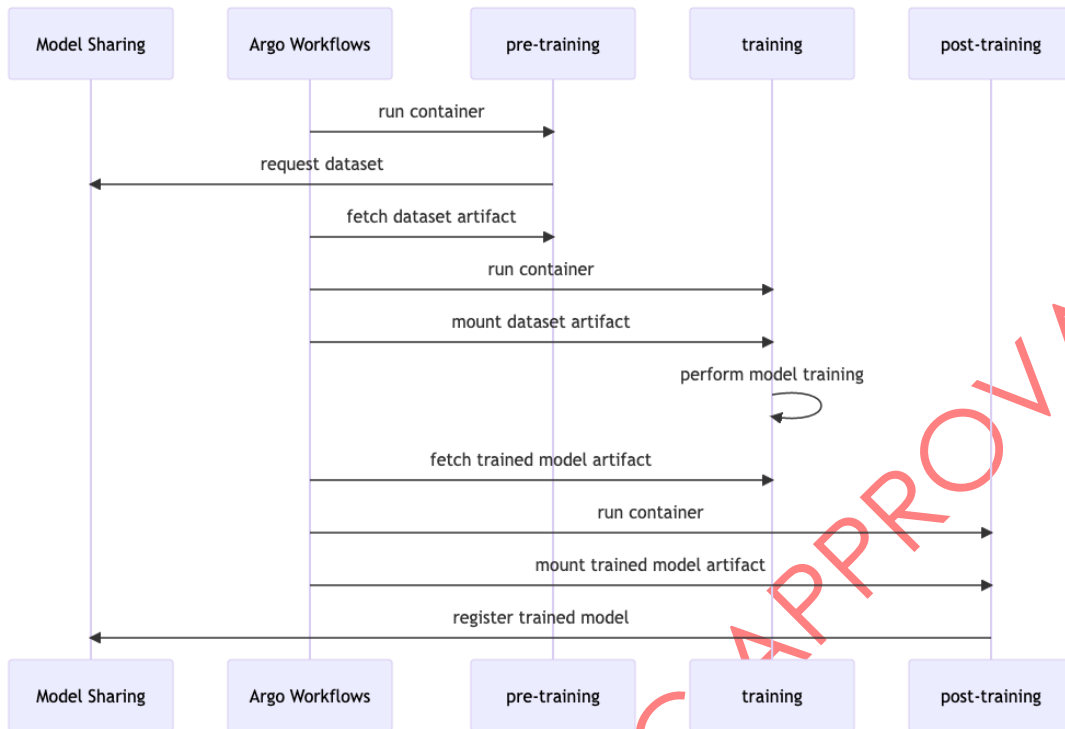


Figure 49 - Sequence diagram of the Model Training service's training job workflow implementation

## 7.4 Model Translation service

The main purpose of the model translation service is to provide ONNX representations for the machine learning models registered in the platform, upon their successful training process.

### 7.4.1 HTTP REST API Operations

The Model Training service exposes the following operations through its HTTP REST API (Table 17).

Table 17 - Model Translation service HTTP REST API operations

Method	Route	Description
POST	/translate/{model_id}	Request intermediate representation for a registered machine learning model

We will now introduce a more in-depth explanation for the implementation of this service's operations.

### 7.4.2 Model translation request

The model translation request operation allows to translate a registered machine learning model into the ONNX format (upon completion of its training job). The following sequence diagram (Figure 50) illustrates the operation's implementation.

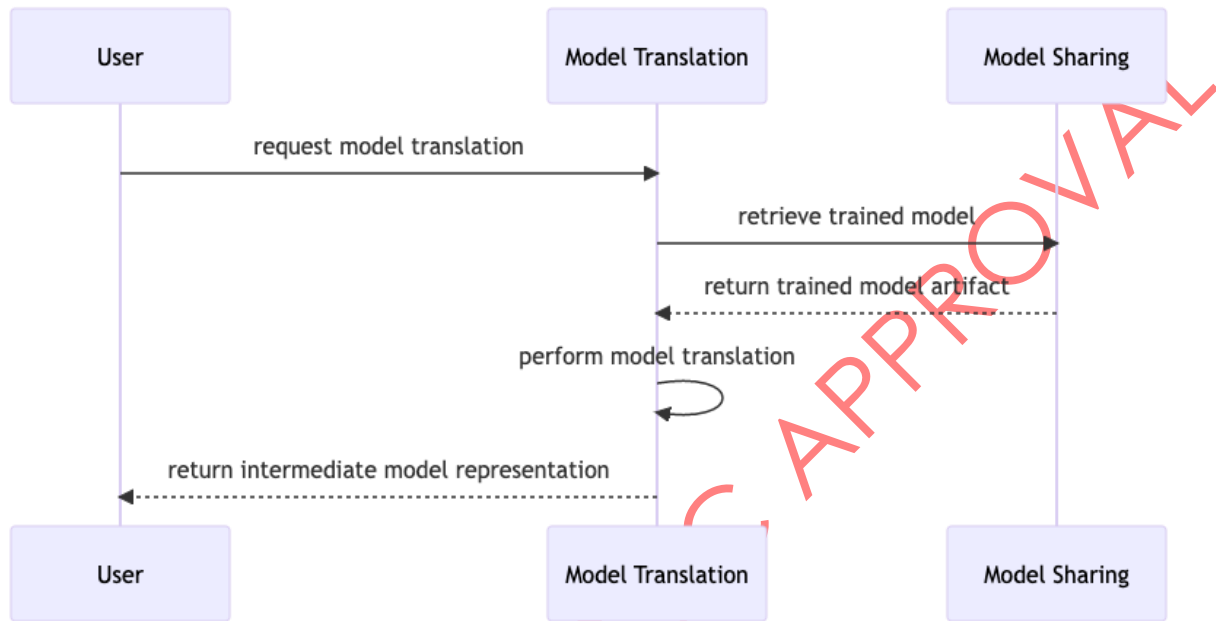


Figure 50 - Sequence diagram of the Model Translation service's model translation operation

DRAFT - PENDING

## 8 References

- [1] IoT-NGIN, D6.3 Interoperable IoT-NGIN meta-architecture & laboratory evaluation, H2020 - 957246 - IoT-NGIN Deliverable Report, 2023.
- [2] IoT-NGIN, D7.4 IoT-NGIN Living Labs use cases assessment and replication guidelines, H2020 - 957246 - IoT-NGIN Deliverable Report, 2023.
- [3] ""The 5 V's of big data".," Watson Health Perspectives, 17 September 2016. [Online]. Available: <https://www.ibm.com/watson-health/merative-divestiture>.
- [4] K. Morris, Infrastructure as code: managing servers in the cloud., O'Reilly Media, Inc, 2016.
- [5] "Argo-CD," [Online]. Available: <https://argo-cd.readthedocs.io/en/stable/>.
- [6] IoT-NGIN, D3.1 - Enhancing deep learning and reinforcement learning, H2020-957246 IoT-NGIN Deliverable Report, 2021.
- [7] IoT-NGIN, "D3.2 - Enhancing Confidentiality preserving federated ML," H2020 - 957246 - IoT-NGIN Deliverable Report, 2021.
- [8] IoT-NGIN, D3.3 Enhanced IoT federated deep learning/ reinforcement ML, H2020-957246 IoT-NGIN Deliverable Report, 2022.
- [9] BDVA, "Big Data Value (BDV) Strategic Research and Innovation Agenda (SRIA)," [Online]. Available: [https://bdva.eu/sites/default/files/BDVA\\_SRIA\\_v4\\_Ed1.1.pdf](https://bdva.eu/sites/default/files/BDVA_SRIA_v4_Ed1.1.pdf).
- [10] IoT-NGIN, D2.3 Enhanced IoT Underlying Technology, final version, H2020-957246 IoT-NGIN Deliverable Report, 2023.
- [11] "Kserve," [Online]. Available: <https://kserve.github.io/website/0.10/>.
- [12] "Kubeflow," [Online]. Available: <https://www.kubeflow.org/>.
- [13] "MinIO," [Online]. Available: <https://min.io/>.
- [14] "MQTT," [Online]. Available: <https://mqtt.org/>.
- [15] "Apache Camel-K," [Online]. Available: <https://camel.apache.org/camel-k/1.12.x/index.html>.
- [16] "FastAPI," [Online]. Available: <https://fastapi.tiangolo.com>.
- [17] "Prometheus," [Online]. Available: <https://prometheus.io/>.
- [18] "Grafana," [Online]. Available: <https://grafana.com/>.
- [19] "Evidently AI," [Online]. Available: <https://www.evidentlyai.com/>.

- [20] "Apache Kafka," [Online]. Available: <https://kafka.apache.org/>.
- [21] "Paho-MQTT," [Online]. Available: <https://pypi.org/project/paho-mqtt/>.
- [22] "Online Learning GitLab Repository," [Online]. Available: [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_intelligence/t3\\_2/online\\_learning/-/tree/main?ref\\_type=heads](https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/t3_2/online_learning/-/tree/main?ref_type=heads).
- [23] "Captum," [Online]. Available: <https://captum.ai/>.
- [24] "PyTorch," [Online]. Available: <https://pytorch.org/>.
- [25] A. G. P. & K. A. Shrikumar, "Learning important features through propagating activation differences," *International conference on machine learning*, pp. 3145-3153.
- [26] D. P. Kingma and J. Lei Ba, "ADAM: A method for stochastic optimization," 2014.
- [27] B. W. & S. C. H. Yap, "Comparisons of various types of normality tests," *Journal of Statistical Computation and Simulation*, vol. 81, no. 12, pp. 2141-2155, 2011.
- [28] IoT-NGIN, "D4.3 - Enhancing IoT Tactile & Contextual Sensing/Actuating," H2020 - 957246 - IoT-NGIN Deliverable Report, 2022.
- [29] "Tensorforce," [Online]. Available: <https://github.com/tensorforce/tensorforce>.
- [30] "Tensorflow," [Online]. Available: <https://www.tensorflow.org/>.
- [31] "RL-Based Optimization GitLab Repository," [Online]. Available: [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_intelligence/t3\\_2/rl-optimization](https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/t3_2/rl-optimization).
- [32] "Pandapower," [Online]. Available: <http://www.pandapower.org/>.
- [33] IoT-NGIN, D7.3 IoT-NGIN Living Labs use cases intermediate results, H2020 - 957246 - IoT-NGIN Deliverable Report, 2023.
- [34] "NVIDIA FLARE," [Online]. Available: <https://developer.nvidia.com/flare>.
- [35] "TensorFlow Federated: Machine Learning on Decentralized Data," [Online]. Available: <https://www.tensorflow.org/federated>.
- [36] "Flower," [Online]. Available: <https://flower.dev>.
- [37] N. Papernot, M. Abadi, U. Erlingsson, I. Goodfellow and K. Talwar, "Semi-supervised knowledge transfer for deep learning from private training data," in *ICLR*, 2017.
- [38] Docker Inc., "Docker hub," [Online]. Available: <https://hub.docker.com/>. [Accessed 2023].
- [39] Argo, "Argo Workflows," GitHub, [Online]. Available: <https://argoproj.github.io/argo-workflows/>. [Accessed 2023].
- [40] IETF OAuth Working Group, "OAuth 2.0," [Online]. Available: <https://oauth.net/2/>.

- [41] "OpenID Connect," [Online]. Available: <https://openid.net/connect/>.
- [42] "Ethereum Virtual Machine," [Online]. Available: <https://ethereum.org/en/developers/docs/evm/>.
- [43] "ConsenSys Quorum," [Online]. Available: <https://consensys.net/quorum/abs/>.
- [44] "Open Neural Network Exchange," [Online]. Available: <https://onnx.ai/>.
- [45] "PostgreSQL," [Online]. Available: <https://www.postgresql.org/>.
- [46] "Model Sharing GitLab project," [Online]. Available: [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_intelligence/t3\\_4/ml-model-sharing](https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/t3_4/ml-model-sharing).
- [47] "Poetry," [Online]. Available: <https://python-poetry.org>.
- [48] "OpenAPI Schema," [Online]. Available: <https://spec.openapis.org/oas/v3.1.0>.
- [49] "OpenAPITools' OpenAPI Generator," [Online]. Available: <https://github.com/OpenAPITools/openapi-generator>.
- [50] "GitLab Package Registry," [Online]. Available: [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_intelligence/t3\\_4/ml-model-sharing/-/packages](https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/t3_4/ml-model-sharing/-/packages).
- [51] Docker, "Dockerfile Builder," [Online]. Available: <https://docs.docker.com/engine/reference/builder/>.
- [52] "Model Sharing Gitlab container registry," [Online]. Available: [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_intelligence/t3\\_4/ml-model-sharing/container\\_registry](https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/t3_4/ml-model-sharing/container_registry).
- [53] M. Aly, "Survey on multiclass classification methods," *Neural Netw*, vol. 19, no. 1-9, p. 2, 2005.
- [54] C. I. N. C. C. G. L. N. Y. C. J. B. Yann LeCun, "THE MNIST DATABASE of handwritten digits," Microsoft Research, [Online]. Available: <http://yann.lecun.com/exdb/mnist/>.
- [55] K. & N. R. O'Shea, "An introduction to convolutional neural networks," arXiv preprint arXiv:1511.0845, 2015.
- [56] HELM, "Helm - The package manager for Kubernetes," [Online]. Available: <https://helm.sh>. [Accessed 2022].
- [57] "Prometheus manifests for IoT-NGIN MLaaS," [Online]. Available: [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_intelligence/t3\\_2/online\\_learning/-/tree/main/monitoring/prometheus](https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/t3_2/online_learning/-/tree/main/monitoring/prometheus).
- [58] "Dockerfile for MLaaS Prometheus," [Online]. Available: [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_intelligence/t3\\_2/online\\_learning/-/blob/main/monitoring/Dockerfile](https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/t3_2/online_learning/-/blob/main/monitoring/Dockerfile).

- [59] "Kubernetes manifest for MLaaS Prometheus," [Online]. Available: [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_intelligence/t3\\_2/online\\_learning/-/tree/main/monitoring/deployment](https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/t3_2/online_learning/-/tree/main/monitoring/deployment).
- [60] "GitLab project for MLaaS Grafana," [Online]. Available: [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_intelligence/t3\\_2/online\\_learning/-/tree/main/monitoring/grafana/dashboards](https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/t3_2/online_learning/-/tree/main/monitoring/grafana/dashboards).
- [61] Kubernetes, "Kubernetes," [Online]. Available: <https://kubernetes.io/>.
- [62] Keycloak, "https://www.keycloak.org," [Online]. Available: <https://www.keycloak.org>.
- [63] Postman, Inc., "Postman REST Client," [Online]. Available: <https://www.postman.com/product/rest-client/>. [Accessed 2023].
- [64] "Podman," [Online]. Available: <https://podman.io>.
- [65] "Model Sharing GitLab Repository," [Online]. Available: [https://gitlab.com/h2020-iot-ngin/enhancing\\_iot\\_intelligence/t3\\_4/ml-model-sharing/container\\_registry/](https://gitlab.com/h2020-iot-ngin/enhancing_iot_intelligence/t3_4/ml-model-sharing/container_registry/).
- [66] "Consensys Quorum official guide," [Online]. Available: <https://consensys.net/quorum/products/guides/getting-started-with-consensys-quorum/>.
- [67] "Minikube," [Online]. Available: <https://minikube.sigs.k8s.io/docs/>.
- [68] "ArgoCD Workflow user guide," [Online]. Available: <https://argoproj.github.io/argo-workflows/workflow-concepts/>.
- [69] "Kubernetes security best practices," [Online]. Available: <https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/>.
- [70] "Solidity," [Online]. Available: <https://docs.soliditylang.org/>.
- [71] IoT-NGIN, "D9.1 - Project Handbook," H2020-957246 IoT-NGIN Deliverable Report, 2020.