PROCEEDINGS OF SCIENCE

Cloud native approach for Machine Learning as a Service for High Energy Physics

L. Giommi,[a,*] **D. Spiga,**[b] **V. Kuznetsov,**[c] **D. Bonacorsi**[a] **and M. Paladino**[a]

[a] *University of Bologna and INFN section of Bologna, Bologna, Italy*
[b] *INFN section of Perugia, Perugia, Italy*
[c] *Cornell University, Ithaca, USA*
E-mail: luca.giommi3@unibo.it, daniele.spiga@pg.infn.it, vkuznet@protonmail.com, daniele.bonacorsi@unibo.it, mattia.paladino@studio.unibo.it

Nowadays Machine Learning (ML) techniques are widely adopted in many areas of High-Energy Physics (HEP) and certainly will play a significant role also in the upcoming High-Luminosity LHC (HL-LHC) upgrade foreseen at CERN. A huge amount of data will be produced by LHC and collected by the experiments, facing challenges at the exascale.
Here, we present Machine Learning as a Service solution for HEP (MLaaS4HEP) to perform an entire ML pipeline (in terms of reading data, processing data, training ML models, serving predictions) in a completely model-agnostic fashion, directly using ROOT files of arbitrary size from local or distributed data sources.
With the new version of MLaaS4HEP code based on uproot4, we provide new features to improve users' experience with the framework and their workflows, e.g. users can provide some preprocessing operations to be applied to ROOT data before starting the ML pipeline. Then our approach is extended to use local and cloud resources via HTTP proxy which allows physicists to submit their workflows using the HTTP protocol. We discuss how this pipeline could be enabled in the INFN Cloud Provider and what could be the final architecture.

International Symposium on Grids & Clouds 2022 (ISGC 2022)
21 - 25 March, 2022
*Online, Academia Sinica Computing Centre (ASGC), Taipei, Taiwan****

*Speaker

PoS(ISGC2022)012

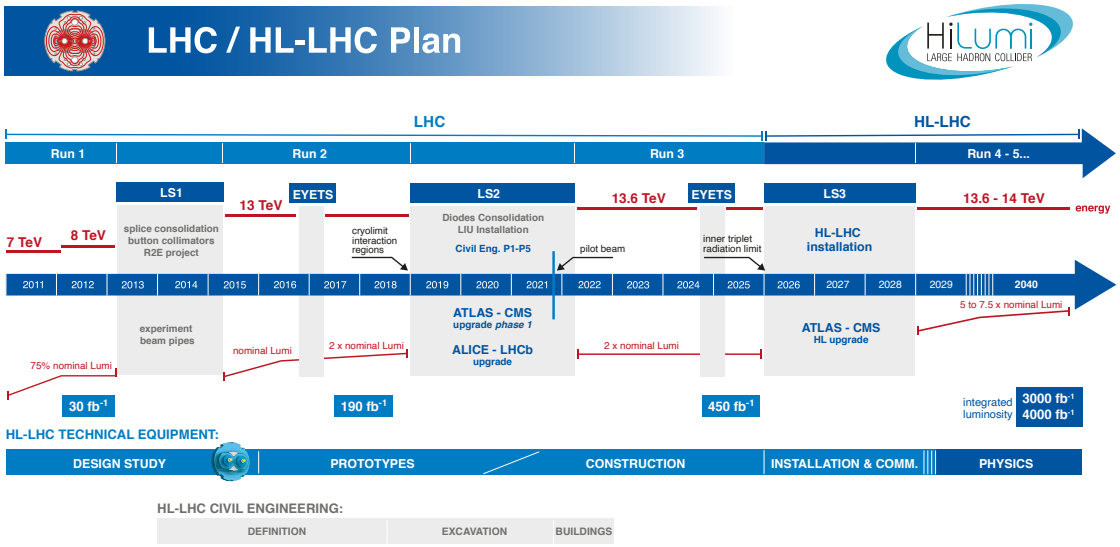

Figure 1: The current schedule for LHC and HL-LHC upgrade and run. Currently, the start of the HL-LHC run is foreseen at the beginning of 2029.

1. Introduction

The Large Hadron Collider (LHC) [1] at CERN allowed to make many discoveries in the High Energy Physics (HEP) domain, e.g. the Higgs boson was announced by the ATLAS and CMS experiments in 2012 [2, 3]. To improve the study of its properties and in order to search for new physics Beyond the Standard Model (BSM), a first upgrade of LHC has been implemented during the Long Shutdown 2, and now the experiments are preparing for a new data taking period. An additional upgrade will bring to the High Luminosity LHC (HL-LHC) phase [4], which is scheduled to start in 2029 when the integrated luminosity is expected to be 20 times larger than the current LHC one (see Figure 1), opening to new challenges due to the increase of data to be processed and their complexity.

Nowadays Machine Learning (ML) techniques are successfully used in many areas of HEP, e.g. in detector simulation, object reconstruction, identification, Monte Carlo generation, and they will play a significant role in the upcoming years when a huge amount of data will be produced by LHC, which will face challenges at the exascale. To favor the usage of ML in HEP analyses, it would be useful to have a service allowing to perform the entire ML pipeline (in terms of reading the data, training a ML model, and serving predictions) directly using ROOT files of arbitrary size from local or remote distributed data sources. We are working on a ML as a Service for HEP framework [5] (referred to as MLaaS4HEP in this paper) providing such kind of solution, as an R&D project in CMS. It was successfully validated using a CMS physics use case, and its performances were tested. We received important feedback from the analysts about their needs: in this paper, we will show that we introduced the possibility for users to provide preprocessing operations, such as defining new branches and applying cuts.

The need of providing a service that could use transparently cloud resources pushed us towards the "cloudification" of the MLaaS4HEP framework. In a previous work [6] we used Dynamic On Demand Analysis Service (DODAS) [7] as a Platform as Service (PaaS) [8] in order to obtain an infrastructure that worked with cloud resources and where MLaaS4HEP was used without any effort from user side, providing a Software as a Service solution (SaaS).

In this work, we present a first solution of an HTTP service for MLaaS4HEP, that allows to submit workflows using HTTP calls. Since the aim is to integrate it into the INFN Cloud [9] portfolio of services, it was necessary to include a proxy server that could manage users' authentication and authorization [10]. We will describe the steps performed to enable such MLaaS4HEP service.

2. Machine Learning as a Service for HEP

The MLaaS4HEP solution we proposed [5] allows to:

- natively read HEP data, that means being able to read ROOT files of arbitrary size from local or remote distributed data-sources via XrootD [11];
- use heterogeneous resources both for training and inference, like local CPU, GPUs, cloud resources, etc.;
- use different ML libraries and frameworks of user choice, e.g. Keras [12], TF [13], PyTorch [14], etc.;
- serve pre-trained HEP ML models, like a models repository, and access it easily from any place, any code, and any framework.

MLaaS4HEP is structured in three different layers: Data Streaming, Data Training, and Data Inference layers (see Figure 2).

The first two layers [15], written in the Python programming language, use the uproot [16] library to read (local and remote) ROOT data and access them as NumPy [17] arrays. Then the data are prepared to be delivered to the ML model for training. We extended the uproot library providing a Python Generator that reads, handles, and delivers data in chunks. Such implementation provides efficient access to large datasets since it does not require loading the entire dataset into the RAM of the training node. In this phase the data are handled, which means they are preprocessed, normalized and the dimension of jagged/awkward [18] arrays are fixed [1].

The Data Streaming and Data Training layers contribute together to the MLaaS4HEP training workflow, consisting in two different steps schematized in Figure 3.

In the first step (denoted by ① in Figure 3) the input ROOT files are read in chunks (which size is fixed by the user) and a specs file is computed. This file contains useful information about the ROOT files, e.g. the maximum dimension of the attributes in jagged branches, and the minimum and maximum values for each branch. In the second step, shown as ②, the ML training phase is performed. It consists of a loop where each time the right proportion of events (the same presented

[1] ROOT files stores flat branches and non-flat branches: the former store simple values (e.g. float or integer numbers), while the latter (also called jagged) contain vectors which dimension can vary along the branch. For more information, see [5].

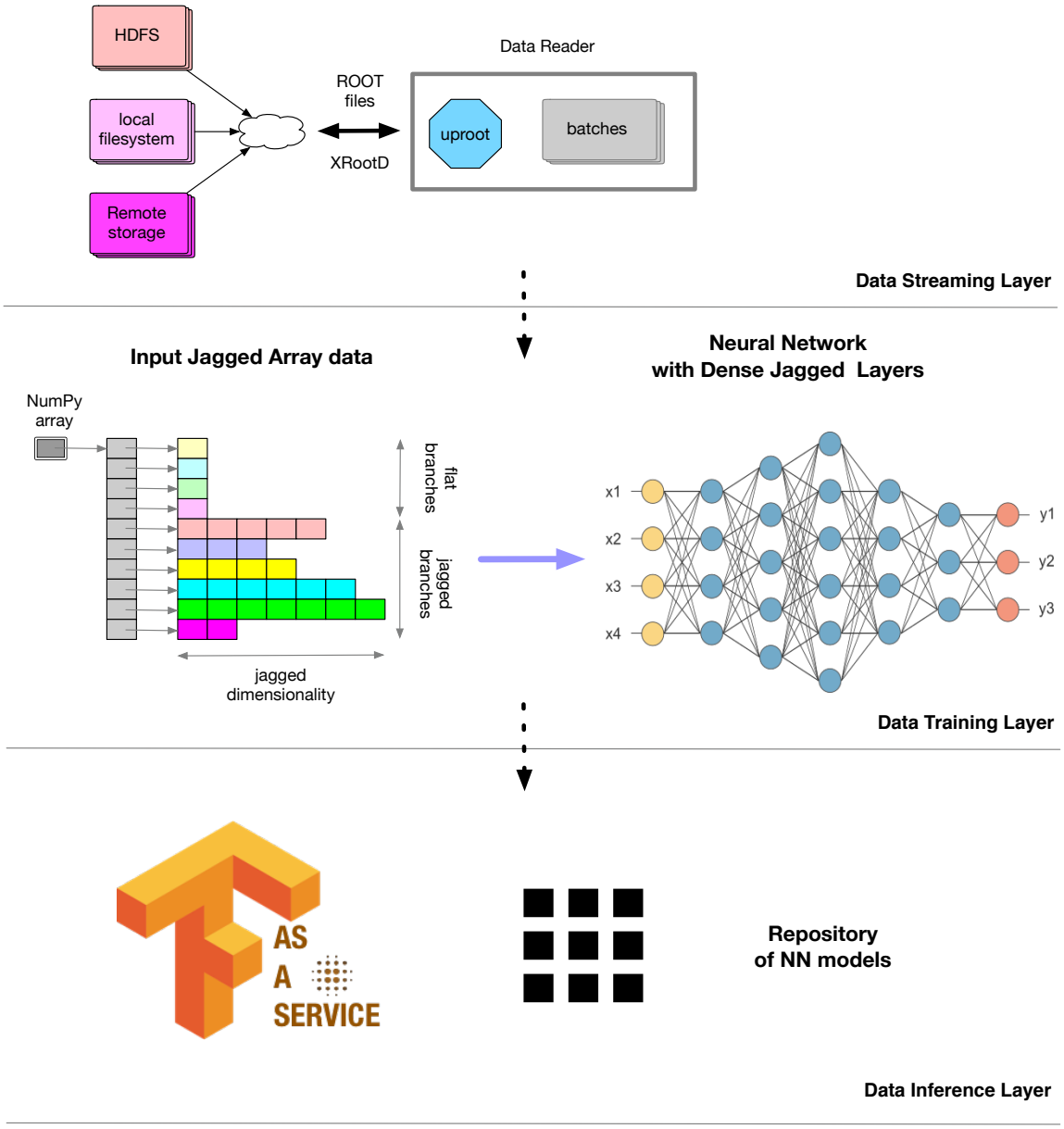

Figure 2: MLaaS4HEP architecture diagram representing its three layers: Data Streaming Layer (top), Data Training Layer (middle), and Data Inference Layer (bottom).

in the ROOT files) is read from each file. Then the events are handled, which means they are converted into NumPy arrays, preprocessing operations provided by the user are applied (this is an addition with respect to previous works and will be covered in more detail in Section 2.1), the dimension of the jagged arrays is fixed and the values are normalized. This part is performed using the information contained in the specs file computed previously. Finally, the chunk is used to train the ML model chosen by the user, for a number of epochs and a batch size fixed a priory by the user itself. The loop continues, creating new chunks of events to use for the training until all the ROOT files are completely read. At the end of the cycle, the training process of the model is completed

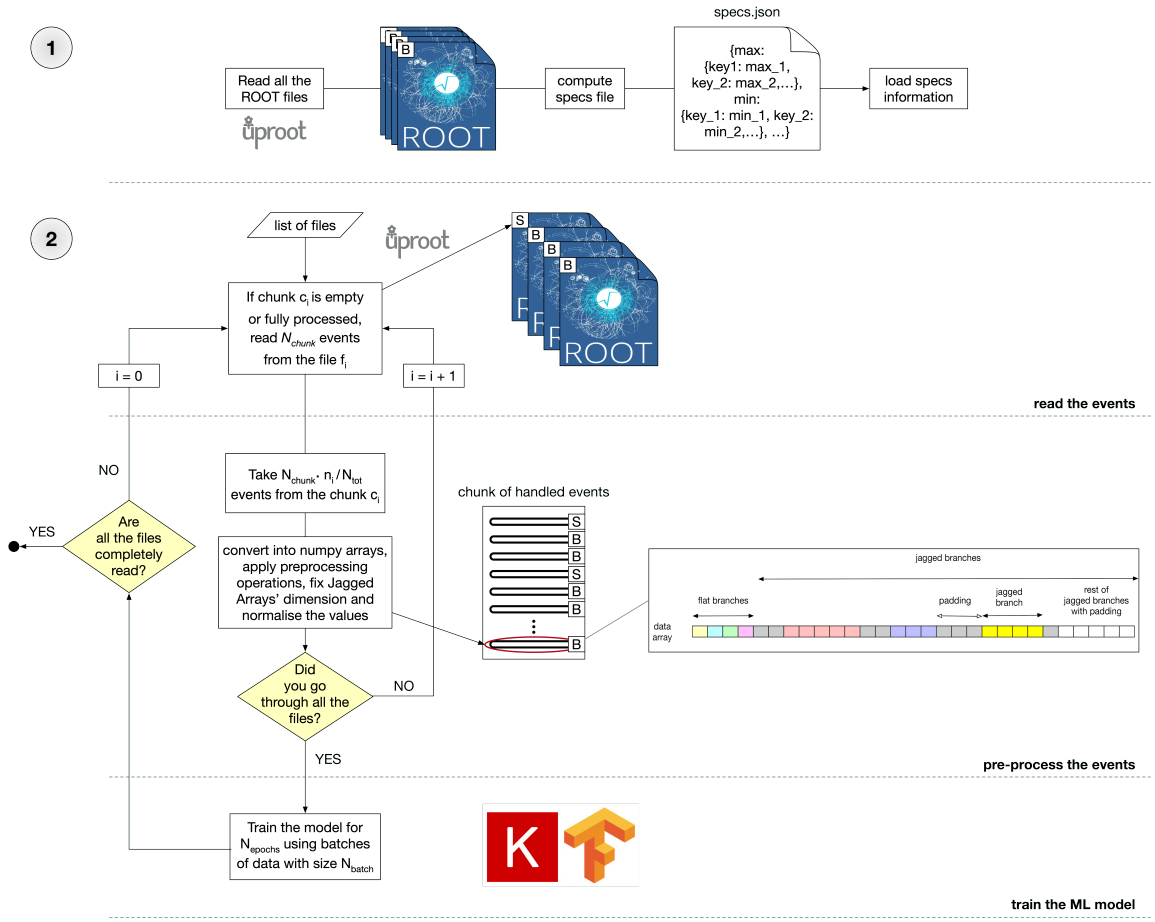

Figure 3: Schematic representation of the steps performed in the MLaaS4HEP pipeline, in particular those inside the Data Streaming and Data Training layers (see text for details). In the picture, N_{chunk} stands for the chunk size, n_i for the number of events in the file f_i, N_{tot} for the whole amount of events stored in the ROOT files.

and the model produced can be used to make inference in physics analysis.

An important aspect of MLaaS4HEP is its being ML model and framework agnostic, that means the user can choose the desired framework and provide directly the definition of the ML model to use in the training phase.

The MLaaS4HEP training workflow is performed running the *workflow.py* script, e.g. in the following way:

```
./workflow.py --files=files.txt --labels=labels.txt --model=model.py
--params=params.json --preproc=preproc.json
```

The *workflow.py* script takes several files as arguments:

- *files.txt* stores the path of the input ROOT files;
- *labels.txt* stores the labels of the input ROOT files in case of classification problems;

- *model.py* stores the definition of the custom ML model to use in the training phase, in the user's favorite ML framework;
- *params.json* stores the parameters on which MLaaS4HEP is based, e.g. number of events to use, chunk size, batch size, redirector path for files located in remote storage;
- *preproc.json* stores the definition of preprocessing operations to be applied to data.

The Data Inference Layer is implemented as TensorFlow as a Service (TFaaS) [19], written in the Go programming language [20], and based on HTTP protocol. We chose the Go language because it natively supports concurrency and it is very well integrated with the TF library. A TFaaS server (e.g. [21] hosted by CERN) can be used as a global repository of pre-trained TF models (version 1 and 2) and serve predictions using the loaded TF models. It is experiment agnostic and can work with any HTTP-based client, in particular both Python and C++ clients have been developed on top of REST APIs (end-points). With TFaaS the clients can test multiple TF models at the same time allowing a rapid development or continuous training of TF models, and their validation.

2.1 Preprocessing operations

We updated the MLaaS4HEP code to support version 4 of uproot and this enabled to introduce preprocessing operations to data. The user can provide a proper *preproc.json* file, like the one shown in Figure 4, that allows to create new branches and apply cuts both on new and on existing branches.

The file is structured in three main categories: *new_branch*, *flat_cut*, and *jagged_cut*. The *new_branch* component allows the user to define new branches and gives the possibility to apply cuts on them, while *flat_cut* and *jagged_cut* allow to apply cuts on existing flat and jagged branches respectively. If the user does not want to create new branches or apply cuts on flat or jagged branches, he/she just should remove the corresponding category from the *preproc.json* file.

2.1.1 *new_branch*

A new branch can be defined through mathematical operations involving existing branches. To create new branches the user should provide under the *new_branch* category a series of information:

- the name of the new branch,
- the mathematical definition of the new branch involving the existing branches, which supports both basic operations (+, -, *, /, **) and NumPy functions [22],
- the type of the new branch (flat or jagged),
- the cut to apply,
- remove or not the new branch after the cut,
- the list of branches to be removed after the creation of the new branch.

```
{
 "new_branch": {

    "log_partonE": {
       "def": "log(partonE)",
       "type": "jagged",
       "cut_1": ["log_partonE<6.31", "any"],
       "cut_2": ["log_partonE>5.85", "all"],
       "remove": "False",
       "keys_to_remove": ["partonE"]},

    "nJets_square": {
       "def": "nJets**2",
       "type": "flat",
       "cut": "1<=nJets_square<=16",
       "remove": "False",
       "keys_to_remove": ["nJets"]}},

 "flat_cut": {
    "nLeptons": {
    "cut": "0<=nLeptons<=2",
    "remove": "False"}},

 "jagged_cut": {
    "partonPt": {
    "cut": ["partonPt>200", "all"],
    "remove": "False"}}
 }
```

Figure 4: Example of a *preproc.json* file provided by the user to apply preprocessing operations on data. See the text for more details.

In case of cuts on jagged branches, the user should specify the type of the cut, choosing between *all* and *any*. The *all* type is used when the cut condition must be satisfied by all values of a given jagged branch, while the *any* type when at least one element in a given jagged branch satisfies the cut condition. In case of multiple cuts, they must be ordered using *cut_i* as key, where *i* indicates the number of the cut. If the user does not want to apply cuts on the new branch or does not want to remove any branch, then the *cut* and *keys_to_remove* keys must be removed respectively from the *preproc.json* file.

2.1.2 *flat_cut* and *jagged_cut*

To apply cuts on flat and jagged branches the structure to be used is similar to the aforementioned one. The following information must be provided:

- the name of the branch to cut on,
- the cut to apply,
- remove or not the branch after the cut.

To apply more than one cut within the same branch just number the cut key using *cut_i*, as seen in Section 2.1.1.

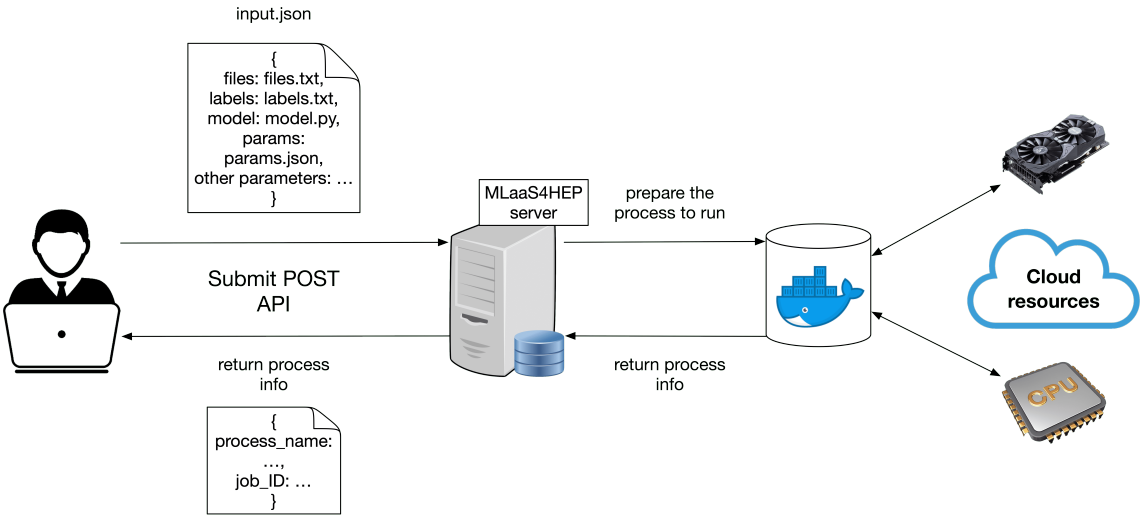

Figure 5: Example of a *submit* API. The user supplies a JSON file with basic information, e.g. the location of the input ROOT files, and the definition of the ML to be used. The server runs a Docker [26] container performing the MLaaS4HEP workflow and returns information about the process to the user.

3. Cloud native approach for MLaaS4HEP

In order to provide a service that can be easily exploited by HEP analysts in their workflows, we started working on an HTTP service for MLaaS4HEP [23] using the Flask [24] framework. We chose Flask because it allows to develop easily web applications in Python. The goal is to create a cloud service that could use cloud resources and could be integrated into the INFN Cloud portfolio of services. The basic idea is that a user can submit a MLaaS4HEP workflow request to a MLaaS4HEP server through an HTTP call with curl [25], as shown in Figure 5, passing an *input.json* file containing the necessary information to run the process. In this solution [23] the process is a Docker [26] container.

In the *input.json* file, the user specifies the mandatory information to run the MLaaS4HEP workflow, e.g. the location of the input ROOT files and the definition of the ML to be used. Then the user can also specify the memory size and number of CPUs to run the container, and the folder of the host file system to be connected with the container.

The call to the *submit* API, when the MLaaS4HEP server runs on localhost and listens on port 8080, is the following:

```
curl -H "Content-Type: application/json" -d @input.json
http://localhost:8080/submit
```

Other simple APIs we implemented allow to check the status of the container and write the logs of the container in a file.

One of the requirements for the MLaaS4HEP service to be integrated into the INFN Cloud is to have an authentication system to manage the users' access.

3.1 Token-based authentication

Modern applications are often designed around APIs. APIs provide access to data or services, and they typically need to restrict API access to authorized parties. Therefore applications need the authorization to call APIs, which means if an application wants to call an API on a user's behalf to access resources owned by the user, it needs the user's consent [27]. In the past, a user often had to share their credentials with the application to enable such an API call on their behalf. Nowadays token-based authentication is widely adopted. In this scenario, a server that authenticates the user and verifies his/her request is needed. When verification is complete, the server issues a token and responds to the request. The user may still have one password to remember, but the token offers a form of access much harder to steal. During the life of the token, the user can access the website or app that the token has been issued for, rather than having to re-enter credentials each time he/she goes back to the same webpage, app, or any other resource [28].

The OAuth 2.0 Authorization Framework [29], published in 2012, was designed to enable an application to obtain authorization to call third-party APIs. With OAuth 2.0, an application can obtain a user's consent, to call an API on their behalf, and not need their credentials for the API site. The application simply gets an Access Token to access a resource on behalf of the user. The Access Token is only intended for API access and not to convey information about the authentication event of the user. OAuth 2.0 defines four roles involved in an authorization request.

- **Resource Server**. A service (with APIs) storing protected resources to be accessed by an application.
- **Resource Owner**. A user or other entity that owns protected resources at the resource server.
- **Client**. An application that needs to access resources at the resource server, on the resource owner's behalf, or on its own behalf.
- **Authorization Server**. A service trusted by the resource server to authorize the client to call the resource server. It authenticates the client or resource owner and requests consent from the resource owner if the client will make requests on the resource owner's behalf.

The OpenID Connect (OIDC) [30] protocol provides an identity service layer on top of OAuth 2.0, designed to allow authorization servers to authenticate users for applications and return the results in a standard way. When a user accesses an application, it redirects the user's browser to an authorization server that implements OIDC. OIDC calls such an authorization server an OpenID Provider. It interacts with the user to authenticate him/her, and then the user's browser is redirected back to the application. The application can request that claims about the authenticated user be returned in a security token called ID Token. Alternatively, it can request an OAuth 2.0 Access Token and use it to call the OpenID Provider's UserInfo endpoint to obtain the claims. Because OIDC is a layer on top of OAuth 2.0, an application can use an OpenID Provider for both user authentication and authorization to call the OpenID Provider's API.

Moreover, OIDC enables Single Sign-On (SSO) [31]. SSO occurs when a user logs in to one application and is then signed in to other applications automatically, regardless of the platform, technology, or domain the user is using. Thus the user signs in only one time.

Access tokens and ID tokens are different. Access tokens are defined in OAuth, ID tokens are defined in OIDC [32]. Access tokens are what the OAuth client uses to make requests to an API of the resource server. They are issued by the authorization server after successfully authenticating the user and obtaining his/her consent. ID tokens contain information about what happened when a user authenticated and are intended to be read by the OAuth client. The ID tokens may also contain information about the users, such as their name or email address, although that is not a requirement of an ID Token. ID tokens are JWTs [33], while Access tokens usually are JWTs but may also be a random string. JWT is a way to encode claims in a JSON document that is then signed. Typically an ID Token holds identification information in the payload, while an Access Token holds authorization information (specified in the "scope" claim).

3.2 Integration of a proxy server for user authentication

We decided to use a reverse proxy server to manage users' authentication for access to the MLaaS4HEP server. We successfully integrated two solutions: auth-proxy-server [34] which is an R&D product of CMS, and OAuth2-Proxy [35] which is a generic tool. To be as much generic as possible, the second option is adopted in the continuation of the work. The steps performed to properly integrate and use OAuth2-Proxy locally with the MLaaS4HEP server [23] are the following.

- Register a client using oidc-agent [36], choosing *https://cms-auth.web.cern.ch/* as authorization server. In this phase, the user is authenticated and provides authorization to the client to talk with the authorization server.
- Obtain an Access Token for the registered client, setting the audience to the *CLIENT_ID*.
- Prepare a proper configuration file for the OAuth2-Proxy server choosing the *CLIENT_ID* and *CLIENT_SECRET* of the registered client (a similar one is prepared for the case of TLS connections).
- Run the OAuth2-Proxy server using the pre-built binary or the pre-built docker image.
- Make a curl call to a MLaaS4HEP API (e.g. *submit*) passing the token obtained previously and using the port where the proxy is running, like the following one (for TLS connections the endpoint will be *https://localhost:4433/submit*).

```
curl -L -k -H 'Content-Type: application/json' -H "Authorization: Bearer ${TOKEN}"
-d @input.json http://localhost:4180/submit
```

This solution has been successfully tested locally on a macOS laptop and on a CentOS 7 VM of INFN Cloud.

At this point of work the real authorization phase is not implemented, which means each user with a valid Access Token is authorized to access the Resource Server, i.e. the MLaaS4HEP server, and the scopes of the token are ignored. A proper authorization phase will be integrated in the continuation of this work.

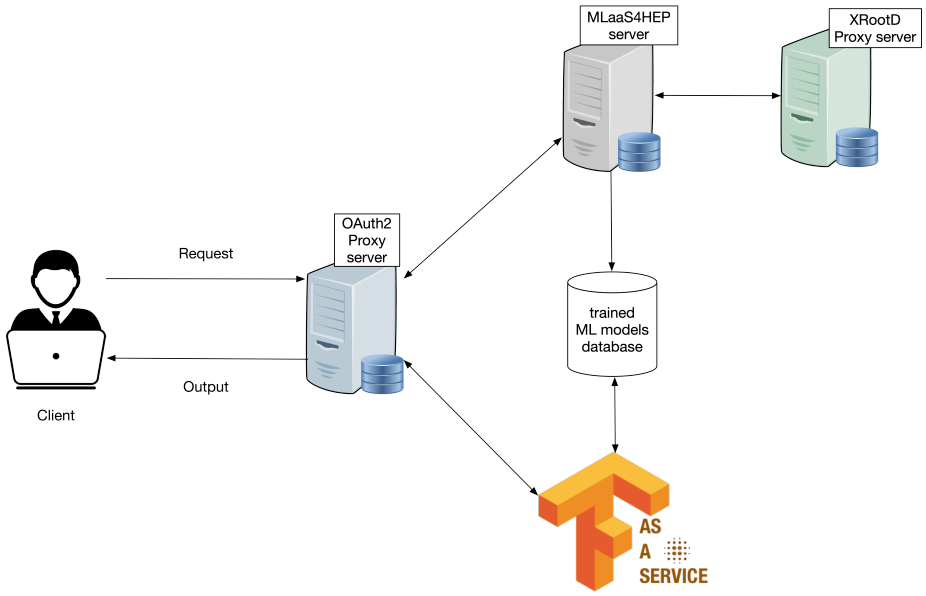

Figure 6: Solution connecting OAuth2-Proxy server, MLaaS4HEP server, and TFaaS. The XRootD Proxy server will allow to store X.509 certificates necessary to read remotely data located on Grid sites.

3.3 Future directions

The next step of this work will be creating a first working prototype, where the OAuth2-Proxy server, MLaaS4HEP server, and TFaaS are connected. The ML models trained by the MLaaS4HEP server will be stored in a database to which TFaaS can have access to make inference. Then we plan to add an XRootD Proxy server [37] to store X.509 certificates necessary to read remotely data located on Grid [38] sites. For a schematic representation of this architecture, see Figure 6.

In our final view of the whole architecture, a possible solution is to split the MLaaS4HEP workflow into two different processes, one for the phase of reading plus handling data and one for training, and they will need shared storage for keeping handled data. These two processes are Docker containers and for proper management of the requests and scheduling of resources, we plan to use Kubernetes [39]. The best option for shared storage is to use SSDs which guarantee fast IO and therefore good throughput in writing/reading data. For a schematic representation of the preliminary final architecture, see Figure 7.

4. Conclusions

ML techniques are successfully used in HEP and they will be crucial in the next years, in particular during the HL-LHC phase when a significant increase in data production and complexity is expected. We proposed a Machine Learning as a Service solution for HEP that allows performing an entire ML pipeline (in terms of reading data, processing data, training ML models, serving predictions) in a completely model-agnostic fashion, directly using ROOT files of arbitrary size from local or distributed data sources.

In this paper, we showed how the user can now apply preprocessing operations on data, like defining new branches and applying cuts. Then we presented our work on a cloud native approach

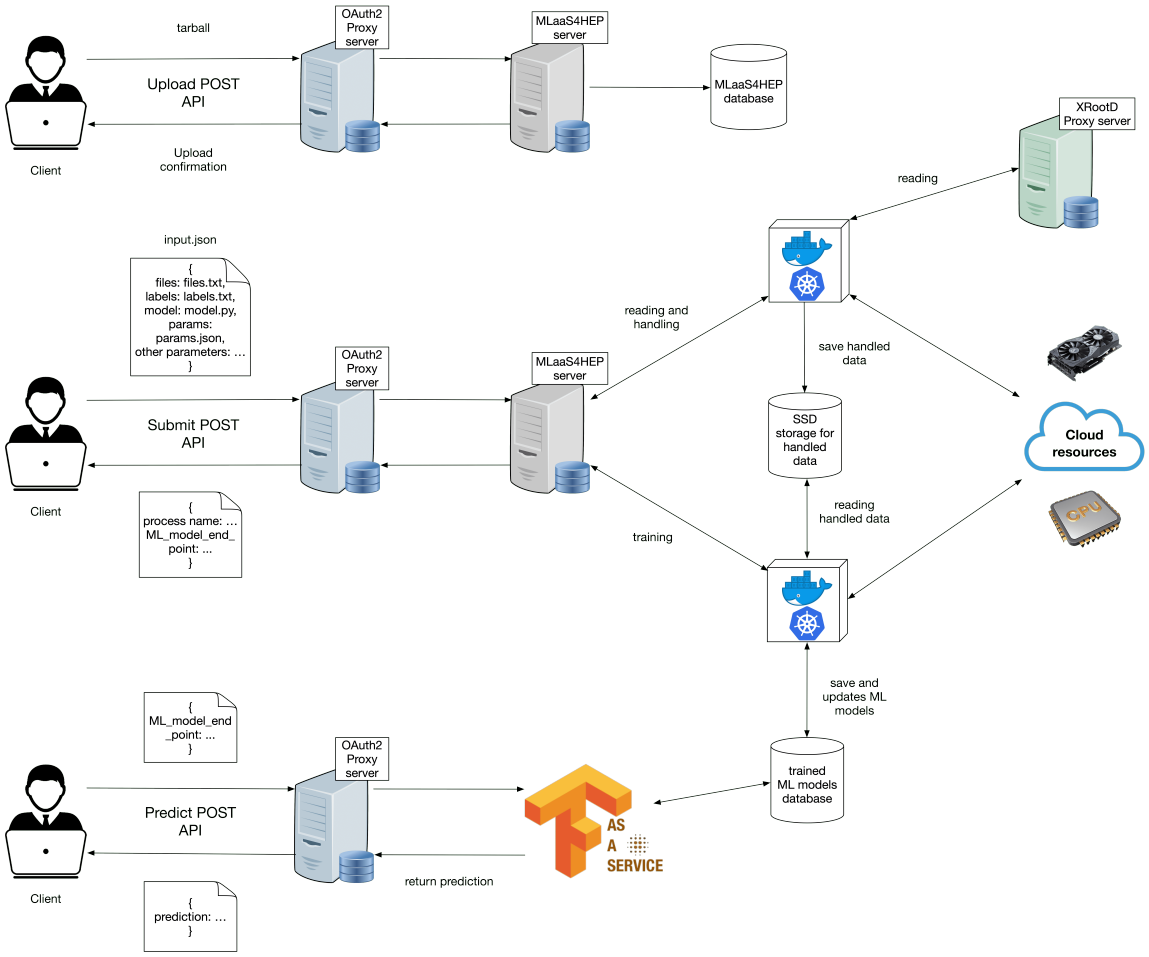

Figure 7: Preliminary final architecture. The user uploads a tarball containing the needed files, then sends a *submit* request that executes the MLaaS4HEP workflow. The workflow is split into two different processes, one for the phase of reading plus handling data and one for training, which will need a shared SSD storage for keeping handled data. Then the trained ML model can be used to make predictions using the *predict* API of TFaaS. The part of authentication/authorization will be managed by the OAuth2-Proxy server and the part of reading remote ROOT files will be possible thanks to the XRootD Proxy server. For proper management of requests and scheduling of resources, we plan to use Kubernetes.

for MLaaS4HEP, i.e. a cloud service which could be integrated into the INFN Cloud portfolio of services. A MLaaS4HEP server was written using Flask, which allows to submit MLaaS4HEP workflows simply using HTTP calls. To manage users' access to this service an authentication system was integrated using an OAuth2 Proxy server. We tested the whole deployment on INFN Cloud using local data of a specific CMS physics use case, i.e. a signal vs background discrimination problem in a $t\bar{t}H$ analysis [5].

As a continuation of this work, we planned in short term to integrate an XRootD Proxy server to enable the reading of remote ROOT files to the MLaaS4HEP server, and to integrate TFaaS in the whole work providing a first working prototype.

References

[1] O. Bruning, H. Burkhardt and S. Myers, *Large Hadron Collider*, Prog. Part. Nucl. Phys. **67** (2012) 705-734

[2] ATLAS Collaboration, *Observation of a New Particle in the Search for the Standard Model Higgs Boson with the ATLAS Detector at the LHC*, Phys. Lett. B **716** (2012) 1-29 [hep-ex/1207.7214]

[3] CMS Collaboration, *Observation of a New Boson at a Mass of 125 GeV with the CMS Experiment at the LHC*, Phys. Lett. B **716** (2012) 30-61 [hep-ex/1207.7235]

[4] *The HL-LHC project*. https://hilumilhc.web.cern.ch/content/hl-lhc-project

[5] V. Kuznetsov, L. Giommi, D. Bonacorsi, *MLaaS4HEP: Machine Learning as a Service for HEP*, *Comput Softw Big Sci* **5**, 17 (2021).

[6] L. Giommi, V. Kuznetsov, D. Bonacorsi, D. Spiga, *Machine Learning as a Service for High Energy Physics on heterogeneous computing resources*, in proceedings of *International Symposium on Grids and Clouds*, Proceedings of Science SISSA, 2021.

[7] D. Spiga et al., *DODAS: How to effectively exploit heterogeneous clouds for scientific computations*, in proceedings of *International Symposium on Grids and Clouds in conjunction with Frontiers in Computational Drug Discovery*, Proceedings of Science SISSA, 2018.

[8] P. Mell and T. Grance, *The NIST Definition of Cloud Computing*, National Institute of Standards & Technology (2011)

[9] *INFN-Cloud*. https://www.cloud.infn.it

[10] *Authentication vs. Authorization*. https://auth0.com/docs/get-started/identity-fundamentals/authentication-and-authorization

[11] *A high performance, scalable fault tolerant access to data repositories of many kinds*. http://xrootd.org

[12] *Keras AI library*. https://keras.io

[13] *Tensor Flow AI library*. http://www.tensorflow.org

[14] *PyTorch AI library*. https://www.pytorch.org

[15] *Machine Learning as a Service for HEP*. https://github.com/vkuznet/MLaaS4HEP

[16] *DIANA-HEP Scikit-hep uproot library. Minimalist ROOT I/O in pure Python and NumPy*. https://github.com/scikit-hep/uproot4

[17] *Scientific package to represent data as multi-dimensional arrays*. http://www.numpy.org

[18] *Awkward Array*. https://awkward-array.readthedocs.io/en/latest/

[19] V. Kuznetsov, *TensorFlow as a Service*. http://github.com/vkuznet/TFaaS

[20] *Go programming language*. http://www.golang.org

[21] *A TFaaS demo server hosted by CERN*. https://cms-tfaas.cern.ch/

[22] *Numpy functions usable with uproot*. https://github.com/scikit-hep/uproot4/blob/main/src/uproot/language/python.py#L237

[23] *MLaaS4HEP server*. https://github.com/lgiommi/MLaaS4HEP_server

[24] *Flask: web development one drop at a time*. https://flask.palletsprojects.com/en/2.1.x/

[25] *curl: command line tool and library for transferring data with URLs*. https://curl.se

[26] *Docker*. https://www.docker.com

[27] Y. Wilson and A. Hingnikar, *Solving Identity Management in Modern Applications: Demystifying OAuth 2.0, OpenID Connect, and SAML 2.0*, Apress 2019

[28] *What Is Token-Based Authentication?* https://www.okta.com/identity-101/what-is-token-based-authentication/

[29] *The OAuth 2.0 Authorization Framework*. https://datatracker.ietf.org/doc/html/rfc6749

[30] *OpenID Connect*. https://openid.net/connect/

[31] *Single Sign-On*. https://auth0.com/docs/authenticate/single-sign-on

[32] *ID Tokens vs Access Tokens*. https://oauth.net/id-tokens-vs-access-tokens/

[33] *JSON Web Token (JWT)*. https://datatracker.ietf.org/doc/html/rfc7519

[34] *auth-proxy-server*. https://github.com/dmwm/auth-proxy-server

[35] *OAuth2 Proxy*. https://datatracker.ietf.org/doc/html/rfc7519

[36] *oidc-agent*. https://indigo-dc.gitbook.io/oidc-agent/

[37] *XRootD. Proxy Storage Services (Caching, Non-Caching, & Server-less Caching) Configuration Reference*. https://xrootd.slac.stanford.edu/doc/dev50/pss_config.htm

[38] *The Grid: A system of tiers*. https://home.cern/science/computing/grid-system-tiers

[39] *Kubernetes: an open-source system for automating deployment, scaling, and management of containerized applications*. https://kubernetes.io