ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOBL03

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

MACHINE LEARNING PLATFORM: DEPLOYING AND MANAGING MODELS IN THE CERN CONTROL SYSTEM

J.-B. de Martel, R. Gorbonosov, Dr. N. Madysa, CERN, Geneva, Switzerland

Abstract

Recent advances make machine learning (ML) a powerful tool to cope with the inherent complexity of accelerators, the large number of degrees of freedom and continuously drifting machine characteristics.

A diverse set of ML ecosystems, frameworks and tools are already being used at CERN for a variety of use cases such as optimization, anomaly detection and forecasting. We have adopted a unified approach to model storage, versioning and deployment which accommodates this diversity, and we apply software engineering best practices to achieve the reproducibility needed in the mission-critical context of particle accelerator controls.

This paper describes CERN Machine Learning Platform (MLP) - our central platform for storing, versioning and deploying ML models in the CERN Control Center. We present a unified solution which allows users to create, update and deploy models with minimal effort, without constraining their workflow or restricting their choice of tools. It also provides tooling to automate seamless model updates as the machine characteristics evolve. Moreover, the system allows model developers to focus on domain-specific development by abstracting infrastructural concerns.

MOTIVATION

Machine learning techniques and in particular neural networks are well suited to the unique challenges of particle accelerator controls [1]. Neural networks are already being used in CERN controls for a variety of use cases including anomaly detection [2], trajectory steering at LINAC4 and AWAKE [3], beam measurements [4] and collimator alignment [5] in the LHC.

In recent years, the rapid expansion of the ML ecosystem and the emergence of MLOps has created a multitude of tools and frameworks to assist data scientists with different aspects of the ML development workflow. These include tooling for experiment tracking and model management (e.g. Neptune [6], Comet [7]), feature storage (e.g. Feast [8]), pipeline and workflow automation (e.g. Pachyderm [9], Airflow [10]), hyper-parameter tuning (e.g. Katib [11], Sigopt [12]), deployment (e.g. Seldon [13]) and monitoring (e.g. Fiddler [14], Evidently [15]). Comprehensive tools which aim to address the whole ML lifecycle also exist, both open source (e.g. MLFlow [16], Kubeflow [17]) and proprietary (e.g. AWS Sagemaker [18], GCP Vertex AI [19]).

However, none of these comprehensive tools fit the use-cases required by CERN controls – they either constrain model developers' workflows or require in-depth knowledge of infrastructural tooling. Furthermore, these tools do not fully address requirements specific to accelerator controls such as high criticality, continuously drifting machine characteristics, variety of use-cases (online and offline, embedded and standalone) and the need to maintain different model configurations for each accelerator beam type.

For these reasons we present a machine learning platform (MLP) specific to CERN controls. It addresses the aforementioned issues by abstracting and simplifying model management, storage, and deployment concerns. In addition, it is open and extensible by design to cope with the rapidly evolving ML landscape and lack of generally accepted industry standard for MLOps. For the same reason, it is designed to be compatible with diverse ML model training environments (local, CERN infrastructure, and public cloud). Helping with rapid development of new models with tools such as experiment tracking or workflow automation are not goals of MLP – instead, it is designed to integrate with existing solutions.

CONCEPTS

We define models as the combination of a model type and model parameters. Model types contain the algorithm and logic of the model, e.g., the neural network architecture, the framework, and data pre- and post-processing. Model parameters are the data which configures the model type, e.g. trained weights of neural networks, and any other configuration variables. As the format of model parameters is highly dependent on the framework used[1], we decided to treat model parameters as opaque data, which we store but don't inspect within MLP.

A given model type can be associated with multiple model parameters. One use case for this is the use of different model parameters for each type of particle beam produced by the accelerators. The opposite is also true, given model parameters can be associated with different model types. For example, a given set of trained neural network weights can be used by a same model surrounded by different pre- and post- processing logic for different use cases.

Model types and model parameters evolve independently and are versioned separately, so we define model type versions (MTV) and model parameters versions (MPV). MTVs and MPVs compatibility follows a many-to-many relationship, as shown in Fig. 1.

The combination of an MTV and a compatible MPV forms a model. Models are fully configured neural networks or

[1] Common formats such as ONNX [20] exist but don't support certain operations such as custom layers or loss functions.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOBL03

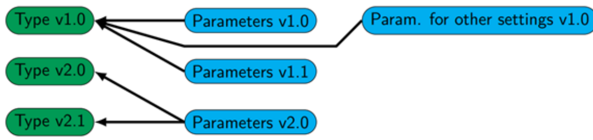

Figure 1: Model types, models parameters and compatibility.

other algorithms which can perform predictions. They expose a uniform prediction API. Models can be both embedded into a client process or deployed as a standalone process and accessed remotely.

MODEL DEVELOPMENT AND USAGE WORKFLOW

MLP has been designed in collaboration with ML model developers and follows their general workflow without restricting it. This section describes MLP features in the order in which they appear in a typical workflow.

Defining a Model

Developers define models exposing a common interface, the MLP Model API. This interface defines four operations:

1. a *fit* operation to train the model on the provided input data;
2. an *export parameters* operation to extract the current values of all model parameters;
3. an *import parameters* operation to configure the model using the provided parameters;
4. and a *predict* operation to return a prediction from the input data.

Registering Model Types

MLP aims to minimize the impact on model developers' workflow by automating management actions as much as possible. Nowadays the majority of model developers use Git as a version control system for their source code, and often use git tags to label particular versions of interest. MLP leverages git tags to automate model type registration: model types are registered automatically when developers push new tagged versions of their model code. A single project can contain multiple models (for example, this is crucial for composite models such as Auto-Encoders [21]). When a new version of the project code is pushed, one MTV is automatically registered within MLP for each model belonging to the project.

Publishing Model Parameters

Model parameters registration cannot re-use the same mechanism as model types since model parameters are created programmatically, practically never stored in git due to their large size and usually created at the end of a lengthy training process. Without using MLP, many model developers at CERN export the trained parameters to disk or an external service within the training script. This requires handling infrastructural concerns such as where to store these weights, how to version them, how to share them with other developers, monitoring the disk space available, etc. This often results in inconsistent naming patterns and file locations which creates additional complexity and a maintenance challenge.

MLP addresses these concerns by providing a client library to register trained parameters. Developers simply provide the model instance, a name for the trained parameters and a version number.

In continuous retraining use-cases, users can also leverage the version generation feature and let MLP decide which version number to assign to the new model parameters version to avoid implementing a complex versioning algorithm themselves.

Based on semantic versioning [22], the automated versioning logic is optimized for common use cases and looks at a limited amount of information to make an informed decision: the currently used MTV and the compatible MPVs registered in MLP. It will refuse to guess in the face of ambiguity. Table 1 illustrates the version guessing behavior for the most common cases.

Table 1: Model Parameters Version Number Generation

MTV	Highest MPV	$\Longrightarrow$	Generated MPV
1.0.0	none exist yet	$\Longrightarrow$	1.0
1.0.0	1.0	$\Longrightarrow$	1.1
1.6.0	1.1	$\Longrightarrow$	1.2
2.0.0	1.2	$\Longrightarrow$	2.0
3.3.0	4.0 (no 3.x)	$\Longrightarrow$	ambiguity
3.3.0	4.0 (3.3 exists)	$\Longrightarrow$	3.4

Managing Compatibility

The default behavior of the platform is optimized for the common use case while leaving the model developer in full control. When a MPV is published from a MTV instance, a compatibility link between them is created automatically.

Furthermore, MLP, based on semantic versioning, makes new MTVs inherit all the compatible MPVs from the previous MTV with the same major version number. The same applies to MPVs. When a breaking change is made to a model type or model parameters and compatibilities should not be copied from the previous version, model developers should bump the major version to indicate the backward-incompatibility to MLP and then the compatibilities will not be copied. For niche use cases, it is also possible to disable compatibility copying on creation. Compatibility links can also be added and removed manually using the client library.

Using Embedded Models

To embed an MLP model in an application written in the same language as the model, developers should use the client library to instantiate an embedded model. The client

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOBL03

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

application must specify the class of the locally available model type version and provide a parameters name and version. The client library will then take care of retrieving the appropriate MPV and load them into the model.

Users can also choose to always use the latest compatible parameters version. This is the right choice for most use cases. Together with automatic versioning, this provides a way to update running models automatically.

Using Standalone Models

MLP models can also be used as standalone models and accessed remotely using the standalone model API. Users connect to the model by specifying the model type name and model parameters name; they can then use the model as if it was available locally.

This not only allows models to be called from any language that supports HTTP requests, but also enables client applications to always use the latest models with no effort required from application developers. Standalone models serve prediction requests over the network, allowing them to be called from any language, not just the language in which they were written.

IMPLEMENTATION

The main components of the implementation are the model API, the client library, the model registry server, and the standalone serving cluster. A simplified overview of the system architecture is presented in Fig. 2.

Model API

Today, the machine learning landscape is heavily dominated by Python and virtually all models at CERN are implemented in the Python language. For this reason, MLP models are currently restricted to Python and the model API is implemented using Python abstract base classes [23]. A model class must implement the *MlpModel* interface (see Fig. 3) to be considered a model type by the Machine Learning Platform. This is usually accomplished through direct inheritance, although other methods for niche use cases exist. Default extensible implementations of parameter saving and loading logic are provided for commonly used frameworks (tensorflow [24], pytorch [25], scikit-learn [26]) to facilitate the implementation.

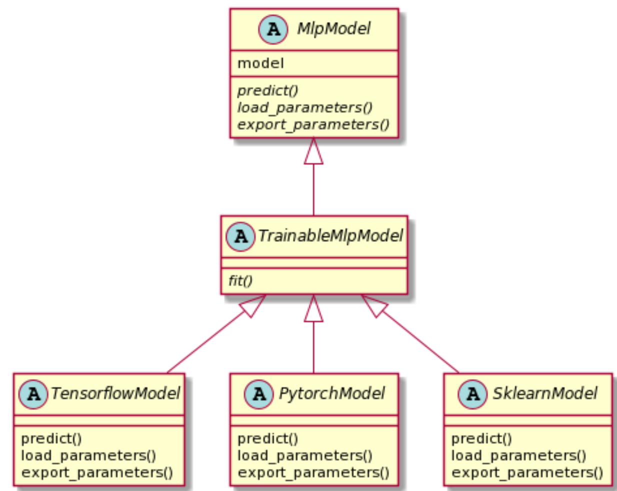

Figure 3: Class Diagram of the MLP Model API.

Model Registry Server

The model registry server is responsible for storing and managing MTVs, MPVs and their compatibilities. It also provides the version generation logic described above. It is implemented as a standard Java/Spring Boot application exposing REST endpoints and documented with Swagger. It uses a relational database to store MTV, MPV and compatibility metadata and an object storage service to store MPV binary objects (trained weights). The relational database is an Oracle database versioned with Liquibase [27]. The object storage service currently uses a CERN NFS service but will change to the Openstack Object Store (Swift) in the near future.

Client Library

MLP provides a single client library for model developers and users alike. Distributed as a Python package, it allows

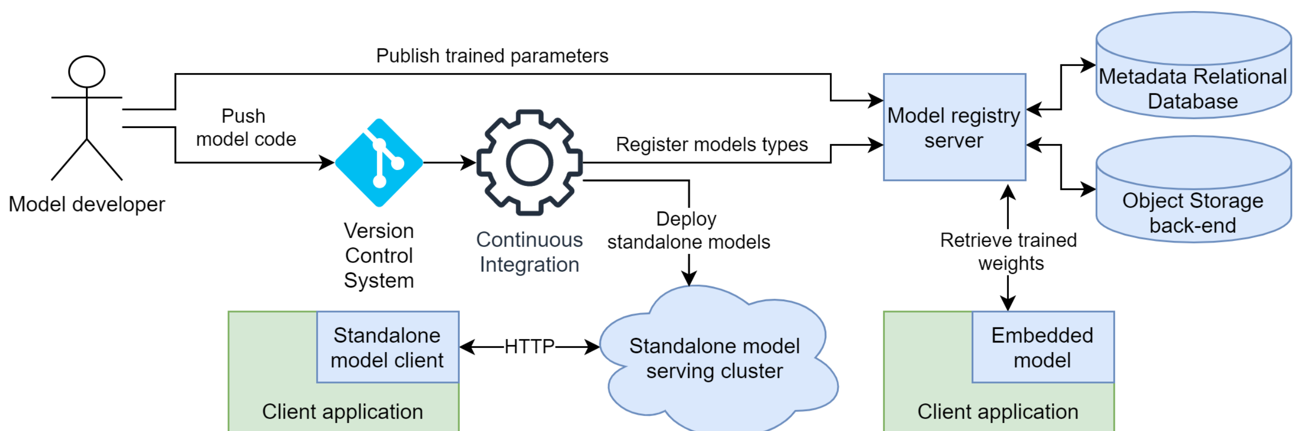

Figure 2: Simplified architecture diagram. Highlighted in blue are the components provided by MLP.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOBL03

model users to publish model parameters, perform search queries against the model registry, and instantiate and use both embedded (see Listing 2) and standalone (see Listing 5) models.

To publish the model parameters of a trained model to MLP, model developers instantiate the MLP client and call a publishing method, providing the trained model, a name for the parameters, and an optional version number, as shown in Listing 1.

```
from mlp_client import Client, Profile, AUTO
from my_model import MyModel

model = BeamLineModel()
model.fit(...)

client = Client(Profile.PRO)
client.publish_model_parameters_version(
    model,
    name="proton_beam_config",
    version=AUTO, # generated by server
)
```

Listing 1: Publishing model parameters versions

When re-training a model continuously, the combination of automatic version generation in the training process (Listing 1) and automatic version selection in the consumer process (Listing 2) makes it easy to set up continuous retraining processes without implementing complex version management logic on both the training and consumer sides.

Registration Automation

Automatic model type registration is accomplished using Continuous Integration (CI). Model developers only need to include a Gitlab CI template provided by MLP. As a single project can contain multiple models, model developers must also register their models as Python entry points under a specific key, as shown in Listing 3.

When model developers push a git tag, the CI jobs defined by the template will publish the package containing the model code to a central repository. It will then iterate over all the models declared in the package and publish them to the

```
from mlp_client import Client, Profile, AUTO
from my_model import MyModel

client = Client(Profile.PRO)
model = client.create_model(
    model_class=BeamLineModel,
    params_name="ion_beam_config",
    params_version=AUTO
)

result = model.predict(data)
```

Listing 2: Using embedded models

```
[options.entry_points]
mlp_models =
    model_1 = awake:VaeEncoder
    model_2 = awake:VaeDecoder
```

Listing 3: Model type declaration in setup.cfg

model registry server. The version of the new model types is determined from the name of the tag. The provided CI template can be extended with custom steps, such as linting and testing jobs, as shown in Listing 4, leaving model developers complete control of their CI pipelines and allowing them to add additional model validation steps if appropriate.

```
include:
- project: machine-learning-platform/mlp-ci
  file:mlp-ci-template.yml

variables:
 project_name: simple_ann

my_custom_test_job:
 script: ./custom-test-script.py
```

Listing 4: Including and extending the Gitlab CI template

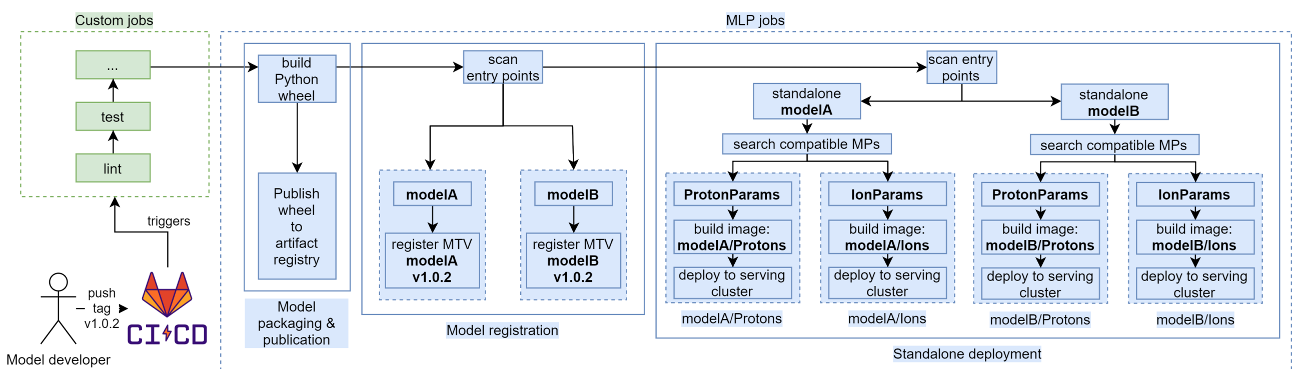

Figure 4: Simplified CI diagram.

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOBL03

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

Standalone Deployment

If the model is declared as standalone, a standalone deployment job is also run. For each model type declared in the model developer's project, the standalone deployment job will identify all compatible model type / model parameters combinations and deploy the latest version of each pair to the standalone serving cluster. To deploy the models to the serving cluster, the standalone deployment job first builds container images which include the MTV, the MPV and a light Python web server. The produced images are deployed to the standalone serving cluster in a subsequent CI job, replacing outdated models if applicable. A simplified diagram of the pipeline is shown in Fig. 4. When the container is started, the web server listens for prediction requests, forwards them to the MTV/MPV combination, and returns a serialized result to the remote caller.

Standalone Serving

The standalone serving component, still in the prototype phase [28], is responsible for serving model predictions to remote clients over HTTP. It is currently implemented as a Kubernetes cluster with an Nginx ingress which listens for prediction requests and forwards them to the appropriate model container. Users can access standalone models using the Python client and use them in a similar way to embedded models, as shown in Listing 5. At the moment, a simple json-based serialization mechanism is used and will be improved in the future.

```
from mlp_client import Client, Profile, AUTO
from my_model import MyModel

client = Client(Profile.PRO)
model = client.create_standalone_model(
    model_class="BeamLineModel",
    params_name="proton_beam_config",
    params_version=AUTO
)
result = model.predict(data)
```

Listing 5: Using standalone models

Technical solutions to autoscale model containers based on usage are currently under investigation. Candidates include the Kubernetes Horizontal Pod Autoscaler [29], KEDA [30] and KFServing [31].

ACHIEVEMENTS AND FUTURE PLANS

Despite its early prototype status, MLP is already undergoing user testing for an application in the AWAKE experiment [32] using embedded models. It is also being evaluated for prediction tasks at the Super Proton Synchrotron, CERN's second largest accelerator, using convolutional neural networks.

The future plans of MLP are primarily focused on integration with other systems, including ML-specific tools and CERN internal services. ML-specific tools include experiment management tools such as Neptune [6] or Weights & Biases [33] and workflow and pipeline automation tools such as Apache Airflow [10]. CERN internal tools include a generic GUI application for numeric optimization, the accelerator time-series data logging service (NXCALS [34]), and settings management tools (INCA/LSA [35]).

Other plans include opening MLP to public cloud services to simplify model training on external infrastructure and adding user-defined metadata to models to enhance search capabilities.

REFERENCES

[1] A. L. Edelen, S. G. Biedron, B. E. Chase, D. Edstrom, S. V. Milton, and P. Stabile, "Neural networks for modeling and control of particle accelerators," *IEEE Transactions on Nuclear Science*, vol. 63, no. 2, pp. 878–897, 2016.

[2] F. Tilaro, B. Bradu, M. Gonzalez-Berges, M. Roshchin, and F. Varela, "Model learning algorithms for anomaly detection in CERN control systems," in *16th International Conference on Accelerator and Large Experimental Physics Control Systems*, 2018, p. TUCPA04.

[3] V. Kain, S. Hirlander, B. Goddard, F. M. Velotti, G. Z. Della Porta, N. Bruchon, and G. Valentino, "Sample-efficient reinforcement learning for cern accelerator control," *Phys. Rev. Accel. Beams*, vol. 23, p. 124801, Dec 2020. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevAccelBeams.23.124801

[4] P. Arpaia, G. Azzopardi, F. Blanc, X. Buffat, L. Coyle, E. Fol, F. Giordano, M. Giovannozzi, T. Pieloni, R. Prevete, S. Redaelli, B. Salvachua, B. Salvant, M. Schenk, M. S. Camillocci, R. Tomás, G. Valentino, F. F. V. der Veken, and J. Wenninger, "Beam measurements and machine learning at the cern large hadron collider," 2021.

[5] G. Azzopardi, A. Muscat, S. Redaelli, B. Salvachua, and G. Valentino, "Operational results of LHC collimator alignment using machine learning," in *10th International Particle Accelerator Conference*, 2019, p. TUZZPLM1.

[6] "Neptune: experiment management and collaboration tool," https://neptune.ai.

[7] "Comet," https://www.comet.ml/.

[8] "Feast.dev," https://feast.dev/.

[9] "Pachyderm," https://www.pachyderm.com/.

[10] Apache Software Foundation, "Apache Airflow," https://airflow.apache.org/.

[11] J. George, C. Gao, R. Liu, H. G. Liu, Y. Tang, R. Pydipaty, and A. K. Saha, "A scalable and cloud-native hyperparameter tuning system," 2020.

[12] S. Clark and P. Hayes, "SigOpt," https://sigopt.com, 2019.

[13] C. Cox, G. Sunner, A. Saucedo, R. Dawson, A. Gonzalez, and R. Skolasinski, "Seldon Core: A framework to deploy, manage and scale your production machine learning to thousands of models," https://github.com/SeldonIO/seldon-core, 2018.

[14] "Fiddler," https://www.fiddler.ai.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOBL03

[15] "Evidently AI," https://evidentlyai.com/.

[16] A. Chen, A. Chow, A. Davidson, A. DCunha, A. Ghodsi, S. Hong, A. Konwinski, C. Mewald, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, A. Singh, F. Xie, M. Zaharia, R. Zang, J. Zheng, and C. Zumar, "Developments in mlflow: A system to accelerate the machine learning lifecycle," *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, 2020.

[17] "Kubeflow," https://www.kubeflow.org/.

[18] "AWS Sagemaker," https://aws.amazon.com/sagemaker/.

[19] "GCP Vertex AI," https://cloud.google.com/vertex-ai.

[20] J. Bai, F. Lu, K. Zhang *et al.*, "ONNX: Open Neural Network Exchange," https://github.com/onnx/onnx, 2019.

[21] D. P. Kingma and M. Welling, "An Introduction to Variational Autoencoders," *arXiv e-prints*, p. arXiv:1906.02691, Jun. 2019.

[22] "Semantic versioning 2.0.0," https://semver.org.

[23] "Abstract Base Classes," https://docs.python.org/3/library/abc.html.

[24] "Tensorflow," https://www.tensorflow.org.

[25] "Pytorch," https://pytorch.org.

[26] "Scikit-learn," https://scikit-learn.org/stable.

[27] "Liquibase," https://www.liquibase.org.

[28] R. Voirin, T. Oulevey, and M. Vanden Eynden, "The State of Containerization in CERN Accelerator Controls," in *18th Int. Conf. on Accelerator and Large Experimental Control Systems (ICALEPCS'21), Shanghai, China*, 2021.

[29] "Kubernetes Horizontal Pod Autoscaler," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale.

[30] "Kubernetes Event-Driven Autoscaling," https://keda.sh.

[31] "KFServing," https://github.com/kubeflow/kfserving.

[32] E. Adli, A. Ahuja, O. Apsimon, R. Apsimon, A.-M. Bachmann, D. Barrientos, F. Batsch, J. Bauche, V. B. Olsen, M. Bernardini *et al.*, "Acceleration of electrons in the plasma wakefield of a proton bunch," *Nature*, vol. 561, no. 7723, pp. 363–367, 2018.

[33] "Weights and Biases," https://wandb.ai.

[34] J. Wozniak, C. Roderick, and S. R. WEPHA163, "Nxcals-architecture and challenges of the next cern accelerator logging service," in *17th Int. Conf. on Accelerator and Large Experimental Control Systems (ICALEPCS'19), New York, NY, USA*, 2019.

[35] D. Jacquet, R. Gorbonosov, and G. Kruk, "LSA - the High Level Application Software of the LHC - and Its Performance During the First Three Years of Operation," in *14th Int. Conf. on Accelerator and Large Experimental Control Systems (ICALEPCS'13), San Francisco, CA, USA*, 2013.

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI