EasyMLServe: Easy Deployment of REST Machine Learning Services

Oliver Neumann, Marcel Schilling, Markus Reischl, Ralf Mikut

Institute for Automation and Applied Informatics,
Karlsruhe Institute of Technology, Hermann-von-Helmholtz-Platz 1,
76344 Eggenstein-Leopoldshafen
E-Mail: oliver.neumann@kit.edu

Abstract

Various research domains use machine learning approaches because they can solve complex tasks by learning from data. Deploying machine learning models, however, is not trivial and developers have to implement complete solutions which are often installed locally and include Graphical User Interfaces (GUIs). Distributing software to various users on-site has several problems. Therefore, we propose a concept to deploy software in the cloud. There are several frameworks available based on Representational State Transfer (REST) which can be used to implement cloud-based machine learning services. However, machine learning services for scientific users have special requirements that state-of-the-art REST frameworks do not cover completely. We contribute an EasyMLServe software framework to deploy machine learning services in the cloud using REST interfaces and generic local or web-based GUIs. Furthermore, we apply our framework on two real-world applications, i. e., energy time-series forecasting and cell instance segmentation. The EasyMLServe framework and the use cases are available on GitHub.

1 Introduction

Machine Learning (ML) approaches are able to solve complex tasks in various domains by learning from data, for example, instance segmentation [1], translation [2], or text-to-image generation [3]. Nonetheless, users struggle to apply ML approaches to their data because adapting code and setting up software and hardware environments need expert knowledge [4]. Therefore, it is important to deploy ML models in a way such that non-expert users can apply these models to their data easily.

Typically, ML models are deployed by programming Graphical User Interfaces (GUIs) because users are familiar with it [4, 5, 6]. This deployment strategy, however, has some disadvantages. Each individual user needs powerful hardware because ML models are computationally expensive. The hardware device is often underutilized as users do not process data continuously. Additionally, the hardware does not scale with user requests. Experts have to perform the installation process on the user site because of software dependencies, for example, GPU drivers or ML frameworks. Code quality suffers because of mixing model and GUI code. Updating software is complicated since distributed code snippets along different users.

To solve these deployment problems, we contribute a software framework (EasyMLServe) to deploy ML approaches as cloud-based software services using Representational State Transfer (REST) [7]. Therefore, we introduce a concept of cloud-based software, the framework architecture, and apply the framework to two real-world applications. Additionally, the framework is available on GitHub (`https://github.com/KIT-IAI/EasyMLServe`), including two real-world applications.

In Section 2, we introduce existing frameworks for the cloud-based deployment of ML models. The requirements, concept, architecture, and implementation of the contributed framework is explained in Section 3. In Section 3.3, we evaluate the frameworks and apply EasyMLServe on two real-world use cases, i. e., energy time-series forecasting and instance segmentation of biological images. Afterwards, we discuss and conclude the results in Section 5.6 and Section 6.

2 Related Work

There are several REST frameworks available to deploy ML services. A lot of them focus on high-performance like TorchServe [8] or TFX Serving [9]. Other frameworks offer easy-to-use config-based deployment of REST ML frameworks like DEEP as a Service (DEEPaaS) [10].

TorchServe is a framework to deploy ML services in the cloud and it is part of the PyTorch ecosystem [11]. The framework is actively maintained and allows parallel requests as well as advanced features like model performance optimization. TorchServe offers a broad range of examples due to the large community. However, TorchServe is restricted to the PyTorch ecosystem and excludes other ML frameworks like TensorFlow [12] or Scikit-Learn [13].

TFX Serving is part of the TensorFlow Extended (TFX) framework for deploying productive ML pipelines and is also part of the TensorFlow ecosystem. TFX Serving is also actively maintained and supports parallel requests, advanced features, and offers a variety of examples. However, like TorchServe, it is not independent of the ML framework, has a complex interface and documentation, and no GUI support.

DEEPaaS is an independent framework to deploy ML services in the cloud. It is actively maintained and it offers a simple config-based interface description. DEEPaaS has support for multiple workers but it does not handle multiple GPU access. Therefore, DEEPaaS recommends running only one worker if there is one GPU. Additionally, DEEPaaS offers no examples and no GUI support.

There are more REST frameworks for ML services available but all of them are very similar to the presented frameworks and have the same problems: They are focused on performance and, thus, are not independent of the ML framework. They offer many additional features, e.g., model management, which makes them complex, and they support no generic GUIs to support fast prototyping. With EasyMLServe, we want to offer an independent and easy-to-use REST framework for ML services that additionally deploys generative GUIs as a starting point for non-experts users.

3 EasyMLServe

In this section, we introduce the EasyMLServe framework to deploy ML services using REST APIs and generic GUIs. First, we define the requirements of REST frameworks for scientific ML services. Second, we describe the basic concept of REST APIs and how the data is exchanged between the ML service, the server, and the GUI. Third, we describe the most important EasyMLServe classes. Fourth, we explain how developers can deploy their ML approaches with the EasyMLServe framework.

3.1 Requirements

Developers in charge of implementing ML services for scientific users from different domains have requirements for REST frameworks to deploy their ML services.

- The REST framework should be actively maintained to ensure version compatibility, continuous improvement, and bug fixing.
- As developers of ML approaches for researchers, it is necessary to use different ML frameworks because not every novel ML approach is available in all ML frameworks. Therefore, a REST framework independent of the ML framework is required.
- In the research domain, prototypes of ML approaches need to be developed fast in the dynamic ML research community. Fast development can be achieved by offering easy-to-use and accessible frameworks.
- Non-expert users need a fast and easy way to use the ML approach. GUIs help to fulfill that objective.
- Real-world ML examples are needed to give developers a good starting point for their approaches. This makes the framework easier to use and reduces development time.

Other REST frameworks for ML services, in general, have additional requirements. However, those requirements are not needed in our use cases. Hence, we classify them as optional requirements:

- Industrial ML services need REST frameworks that handle thousands or more users in parallel which is challenging when thinking of large high-performance computing clusters on which the service should run. Examples of such applications can be found at Amazon, Spotify, or Netflix.
- Some REST frameworks offer more advanced features like model management that always come with additional code and configuration effort.

3.2 Concept

Based on the requirements, we propose a cloud-based service-oriented software architecture. Each ML approach is a software program (service) running on a remote computer (cloud). Data between services and users are exchanged using REST.

REST is a design principle for distributed systems and is based on the HTTP method stack [7]. Therefore, it can be used for every network, e. g., the internet or local private networks, and on any device, from smart meters to high-performance clusters. Software services that are based on REST often exchange data using JSON files which are equivalent to a list of key-value pairs.

If we apply the REST principle to our use case, we have a server offering a ML service by providing a REST interface. All communications between the GUI and server are based on HTTP messages. The REST interface, however, has no GUI. Therefore, our concept considers a GUI to communicate with the server via the REST interface. This GUI can be in form of a web page or a program running on a local computer. An overview of the data exchange between users and services is shown in Figure 1.

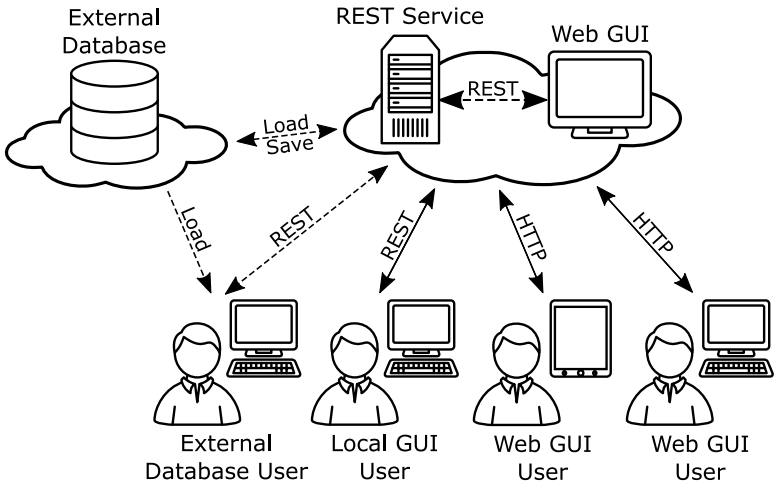


Figure 1: Data exchange between users and ML services. All data exchange between a user and a service is based on JSON objects. GUIs are deployed remotely as a web page or locally on a computer. Without GUIs, smart meters, or other low-end devices, can directly use the ML service using the REST interface. Additionally, data exchange between third-party actors is possible by using, for example, additional databases.

For the EasyMLServe framework, we decided that all requests and responses of the REST interface are encoded as JSON objects. It would be possible to exchange data directly as files but that increases the framework complexity. Therefore, all data from and to the ML services are encoded as JSON objects which makes it easier for developers and reduces framework complexity.

Depending on the GUI type, there are two ways of data exchange between users and the GUI. First, for web-based GUIs, we propose to communicate with the GUI via HTTP but not using a REST interface. Instead, the GUI is deployed using a web server that displays a web page to use the ML service. Second, for local GUIs, our concept suggests interacting directly with the GUI without using HTTP. In both cases, however, the GUI has to communicate with the ML service using the REST interface.

Data can also be exchanged with additional actors using any kind of communication protocol. A simple example would be to load and store data

from a database instead of uploading data via HTTP which can accelerate processing.

Regarding hardware constraints, the server needs to be deployed on powerful hardware depending on the ML task. For instance segmentation tasks, a Graphical Processing Unit (GPU) is recommended but for time-series forecasting using Linear Regression, a CPU is sufficient. The GUI, however, can be deployed on low-end hardware. Additionally, if a GUI is not needed, low-end devices can directly communicate with the REST interface. Smart meters, for example, can forecast energy time-series data or microscopes can process images by using high-performance computing clusters.

3.3 Architecture

The EasyMLServe framework consists of three major classes, i. e., *EasyMLService*, *EasyMLServer*, and *EasyMLUI*. The actual ML service is represented by the *EasyMLService* class. The *EasyMLServer* deploys the REST interface. *EasyMLUI* is the base class for all other UI classes. The relation between these three classes is shown in Figure 2.

EasyMLService is responsible for loading the model and processing JSON request of the REST interface. It returns JSON objects as a result of the REST interface request.

EasyMLServer provide the actual REST interface and passes all REST requests to the *EasyMLService*. Therefore, the *EasyMLServer* has to deploy a web server that handles HTTP messages which is done by the Uvicorn framework [15].

EasyMLUI is the base class for all available generic GUIs of the EasyMLServe framework. It handles the exchange of data between the user and the actual REST ML service. Therefore, *EasyMLUI* takes user input, prepares a REST request, and gets a REST response by passing the REST request to the REST ML service via HTTP. Currently, we support two frameworks for GUIs, i. e., PyQt [16] and Gradio [17]. Both GUIs are available using the child classes *QtEasyMLUI* and *GradioEasyMLUI*.

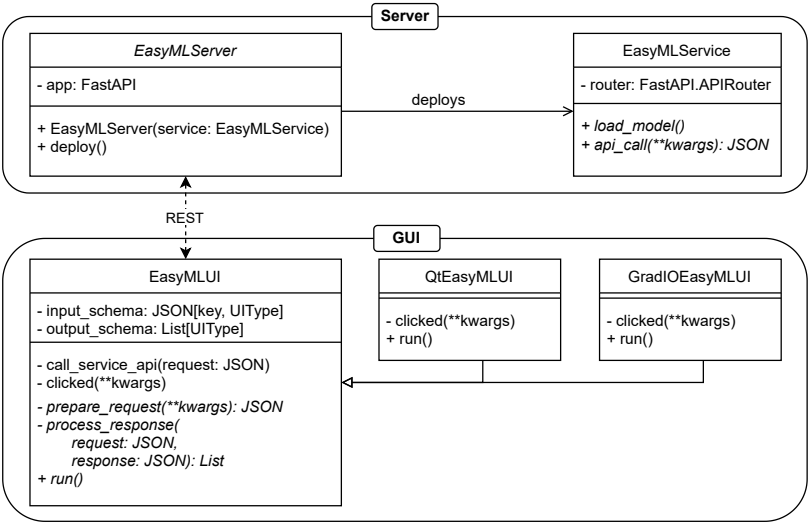


Figure 2: The basic architecture of EasyMLServe with the most important attributes and methods. The top block (Server) shows the relation between *EasyMLServer* and *EasyMLService*. The bottom part (GUI) shows the GUI classes and their relation. Both parts are separated such that an *EasyMLService* can be deployed by the server without using an *EasyMLUI* GUI.

3.4 Implementation

Developers that use EasyMLServe need to implement two classes: A service class with *EasyMLService* as parent class and a GUI class with *QtEasyMLUI* or *GradioEasyMLUI* as parent class. All relevant classes and their methods are shown in Figure 2.

EasyMLService consists of two methods developers need to implement, i. e., `load_model` and `api_call`. The `load_model` method is called after the *EasyMLService* is initialized and allows the user to load the model. The `api_call` method is called when a REST request is received with the actual JSON object. In code, developers have to implement a *EasyMLService* class like:

```
1 class MyMLService(EasyMLService):
2
3     def load_model(self):
4         # load and prepare model
5         pass
6
7     def process(self, request):
8         response ← {...} # prepare REST response
9         return response
```

EasyMLUI is the base class for all other GUI classes. Every GUI class has to implement two methods, i. e., prepare_request and `process_response`. The `prepare_request` method gets all user inputs defined in the input scheme and returns the REST request JSON encoded. The `process_response` method prepares and returns results that should be displayed to the user. In code, developers have to implement a *EasyMLUI* class like:

```
1 class MyMLServiceUI(EasyMLUI):
2
3     def prepare_request(self, some_ui_input):
4         request ← {...} # prepare REST request
5         return request
6
7     def process_response(self, request, response):
8         results ← ... # prepare results (e.g. plots)
9         return results
```

To initialize *EasyMLUI* classes users have to define an input and output scheme. Input schemes describe the data users pass to the GUI, e. g., a CSV file. Output schemes determine what the GUI is presenting to the user, e. g., segmentation results or evaluation files. We need to define an input scheme as data can be passed in several ways, for example, text for translation tasks can be passed via text files or by typing text into text boxes. The output scheme is needed because displaying the response is often not sufficient, for example, when displaying segmentation results users may also be interested in the number of cells or mean cell size.

To define the input and output scheme, we implemented a set of *UITypes* which produce suitable GUI elements. These *UITypes* can be: *Text*, *TextLong*, *Number*, *Range*, *SingleChoice*, *MultipleChoice*, *File*, *ImageFile*, *CSVFile*, *TimeSeriesCSVFile*, or *Plot*.

After implementing an *EasyMLService* and *EasyMLUI* the resulting ML service and GUI can be deployed by calling the `run` method. In code, starting the server and service looks like:

```
# server.py
service ← MyMLService()
server ← EasyMLServer(service)
server.run()
```

```
# ui.py
input_schema ← {'some_ui_input': UIType()}
output_schema ← [Plot()]
app ← MyMLServiceUI(name← 'Example␣Service',
                    input_schema← input_schema,
                    output_schema← output_schema)
app.run()
```

4 Results

In the following, we evaluate the presented REST frameworks TorchServe [8], TFX Serve [9], DEEPaaS [10], and EasyMLServe based on the previously defined requirements for REST frameworks in scientific domains. Afterwards, we apply the EasyMLServe framework on two real-world applications, i. e.,

Table 1: REST frameworks for ML services evaluated on the necessary and optional requirements. Regarding necessary requirements, REST frameworks need to be actively maintained, independent of the ML framework, easy accessible (not complex), supports generative GUIs, and real-world examples. Regarding optional requirements, REST frameworks need to handle parallel requests and advanced features (e.g. model management).

Requirements	**REST Frameworks for ML**			
	TorchServe	TFX Serving	DEEPaaS	EasyMLServe
Maintained	✓	✓	✓	✓
Independent	✗	✗	✓	✓
Accessible	✗	✗	✓	✓
GUI Support	✗	✗	✗	✓
Examples	✓	✓	✗	✓
Parallel	✓	✓	(✓)	(✓)
Advanced Features	✓	✓	✗	✗

energy time-series forecasting and cell instance segmentation, to demonstrate the benefits of the proposed framework.

4.1 Evaluation

EasyMLServe is a REST framework for ML services that is focused on deployment of ML approaches for the research community. Therefore, we define specific requirements which need to be solved.

Evaluating existing REST frameworks with these requirements, we see that REST frameworks, which are focused on performance, are restricted to one ML framework, e. g., TorchServe or TFX Serve. Other ML frameworks like DEEPaaS, however, are independent of the ML framework but offer no generative GUI support.

Our EasyMLServe framework fulfills all requirements and closes the gap of actively maintained, ML framework independent, easy-to-use, and generative GUI supported REST frameworks. A comparison of the REST frameworks based on all requirements, including the optional ones, is shown in Table 1.

Additionally, to demonstrate the capabilities of EasyMLServe, we implemented two real-world use cases. First, electrical load forecasting for

Germany which is representative for several time-series ML problems. Second, biological cell instance segmentation which is a common ML task for image processing.

4.2 Time-Series Forecasting

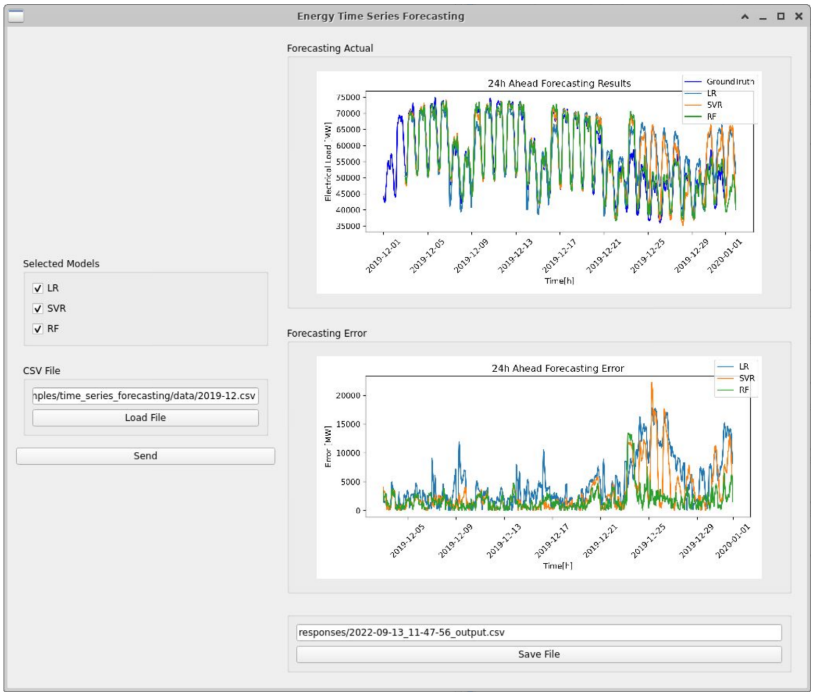
In the first use case, we forecast the electrical load for Germany one day ahead. Hourly electrical load data is used from the Open Power System Data (OPSD) dataset [14]. We used the years 2015 to 2018 for training. Regarding the models, we consider a Linear Regression, Support Vector Machine, and Random Forest model. All models are implemented using Scikit-Learn [13] with default parameter settings.

Our ML service expects a list of model names to use for the forecast, a list of time steps, and the corresponding energy values also as a list. After applying the selected models on the energy time-series, the ML service returns a forecast for each selected model in a list. Each forecast contains the corresponding model name, a list of time steps, and the corresponding energy values.

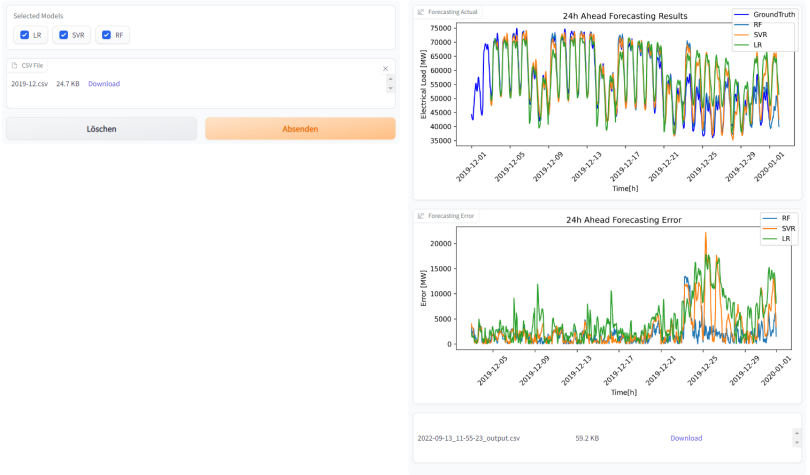
Regarding the GUI input scheme, we choose the MultipleChoice GUI element to define the model names. For the time steps and energy values, we use the CSVFile GUI element. This CSV file needs to be parsed to finally return the model names, time steps, and energy values as one JSON request. The parsing is done in the `prepare_request` method.

For the output scheme, we return two plots and a CSV file. For the two plots, we use the Plot GUI element to visualize the forecast and forecast error. Regarding the CSV file, a File GUI element which contains the actual one-day ahead forecasts is used.

Both supported GUIs can be deployed with EasyMLServe. Developers are able to switch easily between the GUIs by changing the parent class. The resulting GUIs are shown in Figure 3.



(a) Qt UI



(b) Gradio UI

Figure 3: Qt (a) and Gradio (b) based GUIs for the energy time-series forecasting use case. Qt is a locally deployed GUI. Gradio is a web-based deployed GUI.

4.3 Image Segmentation

The second real-world use case focuses on machine learning for images. We use microscopic images from the LIVECell dataset [18] and train a UNet [19] utilizing the KaIDA framework [20].

The ML service expects a Base64 encoded image containing the image encoding, data type, and shape. After segmenting the cells, our service returns an image of detected instances Base64 encoded. Note, the resulting ML service response has the same structure as the input request.

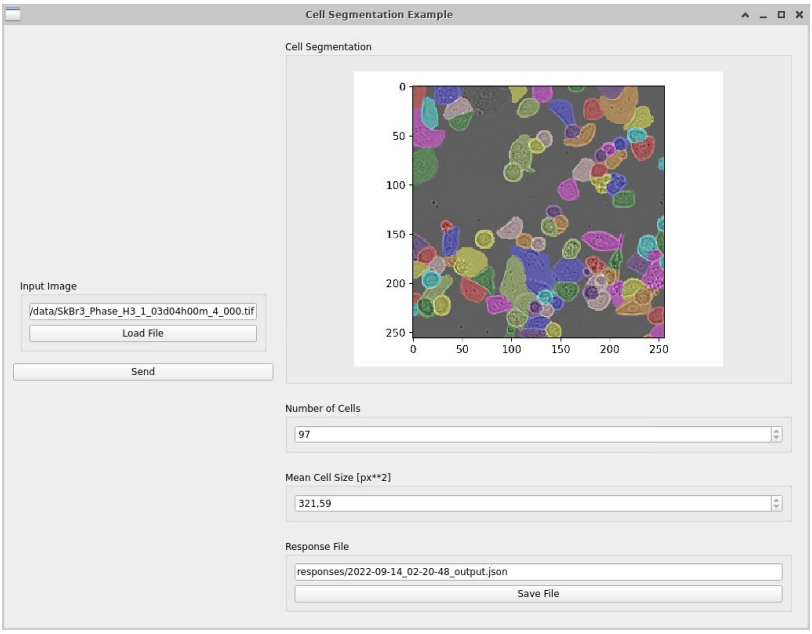
Regarding the GUI input scheme, we only use a File GUI element to select the image file. After loading the image, the image is encoded as Base64 and the REST request is created as JSON.

For the output scheme, we want to display the instances image, number of cells, mean cell size, and the ML service response as a JSON file. The instance overlay image is displayed by using the Plot GUI element of EasyMLServe. The number of cells and mean cell size are visualized using the Number GUI element. The response is displayed as a File GUI element where a user can download it.

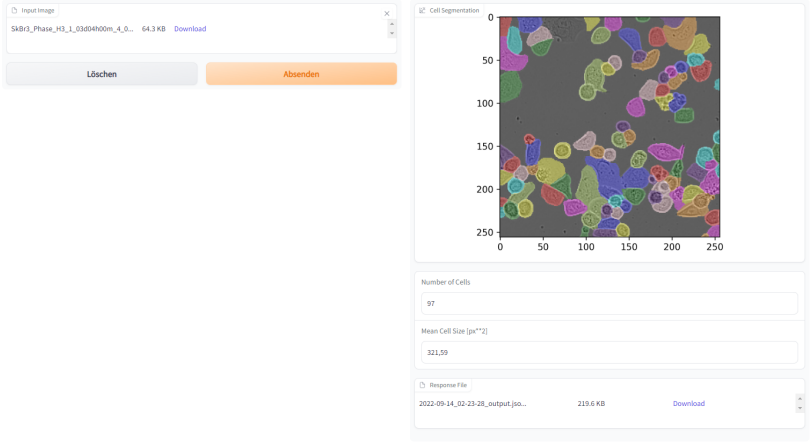
The GUI generation framework of EasyMLServe supports Qt, as a local GUI, and Gradio, as a web-based GUI. Developers can switch between the two GUI frameworks by changing the parent class of the implemented *EasyMLUI* class. Both GUIs can be seen in Figure 4.

5 Discussion

EasyMLServe is a framework to easily deploy ML services using REST. To reduce complexity and make the framework as slim as possible we focus on the deployment part and leave out the training which is done by the developers in any environment. This has the disadvantage that users need help of experts in case they want to change the running ML service. Some frameworks offer training routines that allow users to retrain the models. However, we think also



(a) Qt UI



(b) Gradio UI

Figure 4: Qt (a) and Gradio (b) based GUIs for the cell segmentation use case. Qt is a locally deployed GUI. Gradio is a web-based deployed GUI.

retraining needs the supervision of experts. Especially in the research context where it is important that results are reliable.

The easy deployment of EasyMLServe includes generic GUIs. These GUIs support prototyping and fast deployment. The complexity of such GUIs, however, is limited. It can not cover all possible GUIs without losing its accessibility. Therefore, developers still have to develop GUIs on their own if the limited complexity of the generic GUIs is not enough.

In EasyMLServe, ML services exchange data using JSON objects which is a common way for REST services. It is also possible to directly upload files using multipart/form-data. This would avoid encoding and decoding files and thus reduces processing time and package size. We restricted the EasyMLServe REST interface to JSON objects because it made the framework and communication for the GUI easier. Furthermore, processing times of ML services are mostly restricted to the ML approach itself which is usually the most computational expensive part, e. g., instance segmentation using Deep Neural Networks running on a GPU.

For the implementation of EasyMLServe, we use Python as the programming language. Currently, the most common ML frameworks are written in Python. Therefore, all recent ML approaches are available in Python. However, ML approaches that are not written in Python can not easily be integrated into the presented framework.

Finally, EasyMLServe is a novel framework and we just started with a first version. There are bugs that need to be found and fixed as well as features which are currently missing. Nonetheless, bugs will appear and feature requirements will occur when developers apply this framework which belongs to a normal life cycle of software frameworks.

6 Conclusion

Scientific users have special requirements on the deployment of ML approaches. Deploying software solutions on-site has several disadvantages. Therefore, we propose a cloud-based solution that is based on REST and define

requirements of REST frameworks for scientific usage. These requirements are evaluated on the presented REST frameworks. Existing frameworks do not cover all necessary requirements completely and thus we contribute EasyMLServe, a REST framework for easy deployment of ML services in the cloud. Additionally, our presented framework offers generic GUIs for fast and easy prototyping.

EasyMLServe is a fast solution for ML developers to implement ML services in the cloud. It is actively maintained, independent of the ML framework, easy-to-use, supports generic local or web-based GUIs, and offers real-world applications as a starting point for developers.

To further improve EasyMLServe, we propose to deploy existing solutions with the EasyMLServe framework, for example, pyWATTS pipelines [21]. EasyMLServe is a novel framework which is under development. In future work, we aim to improve the EasyMLServe framework by fixing bugs and add additional features to enhance the user experience. This includes more GUI elements which need to be supported by the GUI generator as well as more complex GUIs.

Acknowledgments

This project is funded by the Helmholtz Association's Initiative and Networking Fund through Helmholtz AI, the Helmholtz Association under the Programs "Energy System Design"(ESD) and „Natural, Artificial and Cognitive Information Processing“ (NACIP), and the German Research Foundation (DFG) under Germany's Excellence Strategy – EXC number 2064/1 – Project number 390727645.

References

[1] C. Stringer, T. Wang, M. Michaelos, and M. Pachitariu, "Cellpose: a generalist algorithm for cellular segmentation," Nature Methods, vol. 18, pp. 100–106, 2021.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, et al., "Attention is All you Need," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[3] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, et al., "Zero-Shot Text-to-Image Generation," in *International Conference on Machine Learning*, vol. 38 pp. 8821–8831, 2021.

[4] E. Gomez-de-Mariscal, C. Garcia-Lopez-de-Haro, W. Ouyang, L. Donati, E. Lundberg, et al., "DeepImageJ: A user-friendly environment to run deep learning models in ImageJ," Nature Methods, vol. 18, pp. 1192–1195, 2021.

[5] S. Belda, L. Pipia, P. Morcillo-Pallarés, J. P. Rivera-Caicedo, E. Amin, et al., "DATimeS: A machine learning time series GUI toolbox for gap-filling and vegetation phenology trends detection," Environmental Modelling & Software, vol. 127, p. 104666, 2020.

[6] C. Doty, S. Gallagher, W. Cui, W. Chen, S. Bhushan, et al., "Design of a graphical user interface for few-shot machine learning classification of electron microscopy data," Computational Materials Science, vol. 203, p. 111121, Feb. 2022.

[7] R.T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," *Doctoral dissertation*, University of California, Irvine, 2000.

[8] H. Bafna, et al., "TorchServe a flexible and easy to use tool for serving and scaling PyTorch models in production," on *GitHub*, `https://github.com/pytorch/serve`, accessed: Sep. 2022.

[9] K. Gorovoy, et al., "TensorFlow Serving a flexible, high-performance serving system for machine learning models, designed for production environments," on *GitHub*, `https://github.com/tensorflow/serving`, accessed: Sep. 2022.

[10] A. Lopez Garcia, "DEEPaaS API: a REST API for Machine Learning and Deep Learning models," Journal of Open Source Software, pp. 1517, 2019.

[11] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," Software available at tensorflow.org, 2015.

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, et al., "Scikit-learn: Machine Learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.

[14] F. Wiese, I. Schlecht, W.D. Bunke, C. Gerbaulet, L. Hirth, et al., "Open Power System Data – Frictionless data for electricity system modelling," Applied Energy, vol. 236, pp. 401–409, 2019.

[15] T. Christie, et al., "Uvicorn an ASGI web server implementation for Python," on *GitHub*, `https://github.com/encode/uvicorn`, accessed: Sep. 2022.

[16] Riverbank Computing Limited, The QT Company, "PyQt6 Reference Guide," on web, `https://www.riverbankcomputing.com/static/Docs/PyQt6/`, accessed: Sep. 2022.

[17] A. Abid, A. Abdalla, A. Abid, D. Khan, A. Alfozan, et al., "Gradio: Hassle-Free Sharing and Testing of ML Models in the Wild," arXiv, 2019.

[18] C. Edlund, T. R. Jackson, N. Khalid, N. Bevan, T. Dale, et al., "LIVECell – A large-scale dataset for label-free live cell segmentation," Nature Methods, vol. 18, pp. 1038–1045, 2021.

[19] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *Medical Image Computing and Computer-Assisted Intervention* pp. 234–241, Springer International Publishing, 2015.

[20] M. P. Schilling, S. Schmelzer, L. Klinger, and M. Reischl, "KaIDA: a modular tool for assisting image annotation in deep learning," Journal of Integrative Bioinformatics, pp. 20220018, 2022.

[21] B. Heidrich, A. Bartschat, M. Turowski, O. Neumann, K. Phipps, et al., “pyWATTS: Python Workflow Automation Tool for Time Series,” arXiv, 2021.