

Scalable AI for the continuous improvement of energy forecasts

Raphael Riege, Fraunhofer IEE, Kassel, Deutschland, raphael.riege@iee.fraunhofer.de

Lukas Koppenhagen, Fraunhofer IEE, Kassel, Deutschland, lukas.koppenhagen@iee.fraunhofer.de

Tim Nöbel, Fraunhofer IEE, Kassel, Deutschland, tim.noebel@iee.fraunhofer.de

Abstract

The weather-dependent volatility in the electrical energy supply system requires reliable forecasting models that take dynamic changes in power grids into account. With the increasing share of renewable energies, a fast and scalable process for energy forecasting becomes crucial. Therefore, in this article, we present a Machine Learning Operations (MLOps)-based concept for the continuous adaptation of energy forecasts and deploy and test it as a forecasting system in a Kubernetes environment. The forecasting system allows to regularly train and roll out individual models for each forecasting object. Experiments on scalable forecasting show that the system meets the time-critical requirements for renewable energy forecasts. We compare a python-based implementation with a java-based one with varying scaling levels. By scaling forecasting applications on demand, forecasts can be generated for 10,000 plants in less than 5 minutes. The shortest runtime was achieved by running a Java application at medium scale with up to 30 services running in parallel. The fact that higher scaling with up to 60 parallel running services had a longer runtime shows that a higher horizontal scaling does not necessarily lead to higher throughput in forecast generation. We therefore conclude that the runtime can be reduced by using a suitable implementation language and the optimal level of horizontal scaling.

1 Introduction

The increasing share of renewable energies (RE) in the energy mix, particularly as a result of the expansion of solar and wind energy generation, has led to a growing dependency on volatile energy generation within the electrical energy supply system [1]. Due to their dependence on the weather, wind and solar energy generation require precise energy production forecasts. These are essential for the planning and operation of electricity grids and for energy marketing. Reliable forecasts make it possible to reduce or efficiently implement balancing measures such as redispatch [2, 3].

A major challenge for accurate RE forecasts is that electricity production differs from plant to plant. Machine learning (ML) models generally show the highest forecast quality [4, 5]. However, renewable energy plants have location-dependent latent specifics, such as topological features in the immediate vicinity, the condition of the plant and other distorting factors. These specifics cannot be fully mapped in the master data of the plants or are not known. Therefore, the specific behavior of a plant must be learned by the models. An efficient solution is to combine plant-specific optimizations with the shared knowledge of learned behavior of similar plants using multitask learning [6, 7]. This approach has been demonstrated to be more resource-efficient than single-task learning, yet it still necessitates the training and deployment of a large number of model optimizations.

Other significant challenges are data and concept drifts, i.e. a change in the input data of a model or the relationship between input data and target variables, which can impair the model quality during operation [8]. Due to the increasing dynamics in the energy system, both on the generation and consumption side, changes occur regularly. Therefore, forecast errors increase, particularly in the case of added

assets with a short data history or changes in consumption behavior, such as the installation of heat pumps. Consequently, forecasts must be regularly adapted to new circumstances. This requires frequent updates of the forecasting models, as changed contexts can be learned by retraining the models [9].

An adoptable forecasting system must therefore be able to regularly retrain, deploy and provide a scalable number of models. These models must be integrated into services that generate forecasts in a short runtime, as processes such as redispatch and energy operation management are time critical. Likewise, it is important to make unused resources available to other processes in order to save costs and emissions [10]. For example, day-ahead forecasts are usually calculated 1 to 4 times a day and unused resources should be available in the meantime.

The aim of this paper is to develop and experimentally investigate a concept for a scalable energy forecasting system. Our focus is on the scalability of an inference service, which runs within a Kubernetes infrastructure and is responsible for applying the model and generating forecasts. We compare a python-based implementation with a java-based one across three scaling levels. We address a use case in which day-ahead forecasts are to be provided for 10,000 photovoltaic plants in the shortest possible time.

2 Background and related work

Continuous training (CT) enables the dynamic re-training of models, allowing for seamless adaptation to new data. For instance, models can be updated with information from new power plants, or they can learn new behaviors from existing assets, while preserving the knowledge of previously learned behaviors [11, 12]. However, the potential of continuous training can only be realized if it is used alongside suitable information and communication technology

(ICT) infrastructures. For continuous deployment and commissioning without major delays, so-called Machine Learning Operations (MLOps) principles such as automation, workflow orchestration, reproducibility, versioning of data, models and code, continuous ML training and monitoring are described in the literature [13]. The aim is to seamlessly transfer ML models into production by standardizing the processes for model development and operation.

Some case studies have already shown how MLOps principles can be applied in the field of energy generation forecasting and energy price forecasting [14, 15]. These case studies address the challenge that ML pipelines are difficult to generalize for different use cases. As a limitation, the authors focus on single models and not the scalability of training and forecasting.

Subramanya et al [16] demonstrate how cloud design patterns can be used within a microservice infrastructure in Microsoft Azure to operate MLOps pipelines. Autoscaling mechanisms are also discussed here but not investigated further.

Burgueño-Romero et al [17] show that Kubernetes clusters are a suitable infrastructure for MLOps practices. In particular, it is shown that such an infrastructure is capable of processing a large amount of data.

Sigfridsson [18] describes a framework for computing many predictions, but it is on a much smaller scale than is required for the use case described in this paper.

The fact that autoscaling in Kubernetes represents an opportunity to significantly reduce energy consumption without major losses in the processing of requests is shown in [10].

Based on these findings, this paper describes a MLOps concept for continuous forecast provisioning and validates it within a microservice infrastructure on Kubernetes. The aim is to demonstrate how auto-scaling can enable the provision of scalable forecasts for a large number of assets. In summary, increased automation and scalability are necessary because of regular changes in the energy system. The

need for forecasts is increasing, particularly due to the dynamic expansion of renewable energies. Against this background, we developed a scalable and adoptable forecasting system.

3 Concept and architecture

The integration of microservices into a Kubernetes infrastructure provides a robust solution to meet continuous training (CT) and forecasting requirements. Kubernetes provides an abstraction of the hardware level, which enables scaling beyond the limits of individual machines. On such a Kubernetes cluster, containerized applications are executed in so-called pods. A pod is the smallest organizational unit in Kubernetes and often only houses one container. In addition to pods, there are other Kubernetes components that perform various administrative tasks. One of the biggest advantages of Kubernetes is that the cluster management system constantly monitors the status of a pod. If a pod crashes, it is automatically restarted. This enables the operation of fail-safe systems. [19]

Our proposed system, shown in **Figure 1**, consists of several central building blocks: Data Store, Model Store, Training Service, Inference Service and Quality Assurance Service.

The data store is responsible for storing all the data required for training and forecasting. This includes historical and real-time data on target variables, features and master data supplied by external sources. The data store also stores the generated forecasts, making them accessible from outside the cluster. The Model Store stores the trained models and makes them available to the forecast inference service. For each single asset, it stores an individual model together with a version tag and staging information. If a model is retrained, a new version is stored. Through technical validation and professional evaluation of the models, they move up to a productive stage. The model store is technically implemented with MLflow, which offers the functions mentioned [20].

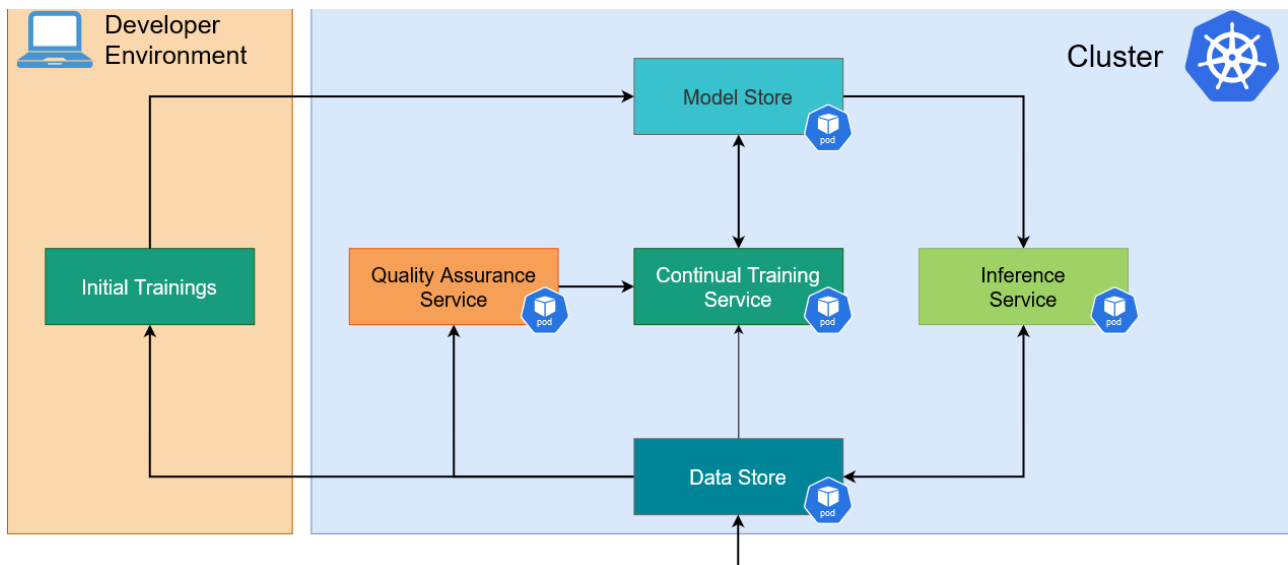


Figure 1 Forecasting system consisting of interrelated services

The initial ML training on a large history of data is carried out outside the Kubernetes infrastructure. The continuous training (CT) runs in a training pipeline and is automated by a training service. This service loads training data from the data store and the current version of the model from the model store and then retrains the model. Various CT variants are conceivable here, which differ in whether only new data or a combination of historical and new data is used for retraining [12]. All trained model versions are stored as artifacts in the model store.

The inference service creates forecasts by applying the current models in the productive stage to new data. This is done on a time-controlled basis in regular batches. Forecasts are available to users via the data store.

The quality assurance service monitors forecast quality and initiates retraining if a significant decline in forecast quality is detected. For this purpose, past forecasts are compared with the respective real-time measured values.

This architecture enables efficient and dynamic adaptation to changing requirements and continuously ensures the quality of the forecasts.

4 Experiment for the use case: Inference service for scalable PV forecasts

In the experiment, we focus on forecasting for 10,000 virtual photovoltaic (PV) plants. Based on Vogt et al [21], we used a Bayesian Embedding Multi-Task Learning (BEMTL) model, which was trained on different PV parks. In the experiment, we investigated the scalability and duration of the forecast generation. Since the forecast quality for individual assets was outside the scope of research, we used the same model for prediction for each asset. However, we simulated the case of 10,000 different models by repeatedly reloading the model from the model store. The models for this use case differ only slightly in their weights so that their difference in computing time is negligible. Therefore, representative results can also be achieved with this simplification.

We trained the model using Python and the PyTorch framework [22], converted it into an Open Neural Network Exchange (ONNX) format and stored it in the model store. ONNX is an open model standard and makes it possible to run the model in different environments [23]. As input, we use numerical weather forecasts for the following day for temperature, global radiation and wind speed from the weather model ICON [24] as well as the azimuth and solar altitude. We generated a deterministic forecast time series for the day-ahead. The aim of the experiment was to calculate and provide 10,000 forecasts in the shortest possible time.

4.1 Implementation of a forecast pipeline

The experiments of a dynamically scalable inference service are carried out on Fraunhofer IEE's internal Kubernetes cluster. The cluster has a total of 2132 CPU cores and 4.5 TB of RAM. The experimental setup shares this resource pool with other applications that use this infrastructure. A maximum of 10 CPU cores per pod are available with a pod limit of 60 pods. This means that for the experiments up to 600 cores can be used, but in practice this is not a limitation.

The inference service is deployed on the cluster and comprises multiple Kubernetes components, as illustrated in **Figure 2**.

The first component is a Deployment wherein a pod containing the forecasting application is defined. The application is a stateless application so that each replica runs independently and can process incoming requests at any time. We implemented two different variants of the application and compared them the experiments. Both variants have the same structure and perform the same steps, but one variant is a Python FastAPI application [25], while the other is a Java Spring Boot application [26]. The Java application uses the Deep Java Library (DJL) to execute ONNX models. DJL offers various engines to support the typical ML frameworks, including ONNX [27]. In Python, ONNX models can be executed via an ONNX Runtime Inference Session [23]. The application provides a REST endpoint that receives the HTTP-Request containing the parameters for identifying the model and the input. This triggers the inference process: First, the model artifacts are loaded from MLflow. In the second step, the input data is retrieved from the data store via a REST interface. Then the application calculates the prediction by applying the model on the input data. In the third step, the prediction is stored in the data store for further use.

The second component is a Horizontal Pod Autoscaler (HPA), which handles the scaling of the application. Kubernetes assigns resources to the pods, which the HPA uses to measure the load of the individual pods. If the load in the pods exceeds a set limit, the HPA creates new replicas of the service as additional pods. Conversely, if the load decreases, the HPA scales back down by shutting down pods. The advantage of using the HPA is that resources are only used when they are needed.

The third component is a Load Balancer, which distributes the incoming requests to the pods, i.e. replicated inference services. At the same time, it provides a common address for all pods for requests from outside the cluster. One feature of the Load Balancer is that it does not queue incoming requests and waits for a service to become available but passes them through immediately. In addition, once distributed, requests are no longer redistributed to new pods that are started by the HPA. This must be considered when running the experiments.

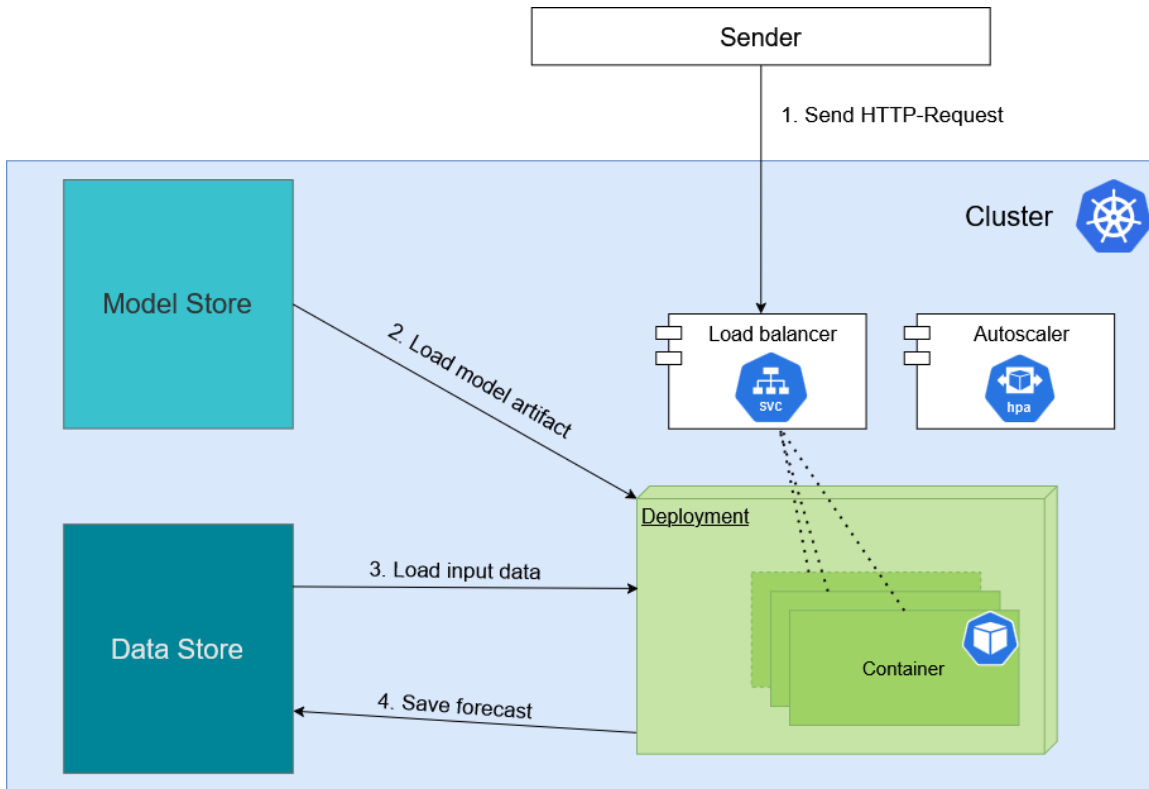


Figure 2 Process steps and components of the inference service

4.2 Experimental procedure

To examine the system, we calculated 10,000 predictions with both implementations and used three different scaling configurations and compared the runtimes.

To prevent all requests from being distributed to the initial pods and the newly created ones not being used, we introduced Batching on the client side. This ensures that the requests are regularly redistributed, and all pods are utilized. We used a batch size of 100x100 for the runs with active HPA. This means that 100 requests are made for the calculation of 100 forecasts each. These HTTP GET requests transfer all parameters via the query string. The parameters are then used to load the model configuration from the model store and the corresponding data from the data store. This allows the application to perform the inference. The application then saved the resulting output data in the data store.

As a baseline, we used one pod to calculate the predictions sequentially. We expected that this approach would be the most time-consuming. In the next configuration, we configured the HPA so that at least 3 pods are active and up to 30 pods can be started if required. We expected that this approach would be much faster than with just one pod, but would not achieve 30 times the throughput. In the last configuration, the run starts again with 3 pods, but this time

the HPA can launch up to 60 pods. We assumed that this approach does not achieve 60 times the throughput of the baseline, but can achieve approximately twice the throughput of the approach with the pod limit of 30.

5 Results

In the baseline version without horizontal scaling with a pod limit of 1, the Java implementation took around 27 minutes and the Python implementation took around 1 hour and 7 minutes to calculate 10,000 forecasts. Horizontal scaling could significantly reduce these runtimes.

Figure 3 shows that a target runtime of less than 300 seconds was achieved for 10,000 requests in all higher scaling configurations. The shortest runtime, at 3 minutes and 47 seconds, was achieved using the Java application with 30 pods. With this configuration, it was possible to create more than 600 forecasts every 10 seconds, approximately 140 seconds after the start of the process. This high throughput was occasionally achieved with the Python variant and 30 pods, but it could not be maintained over a longer period. This resulted in a total duration of 4 minutes and 37 seconds.

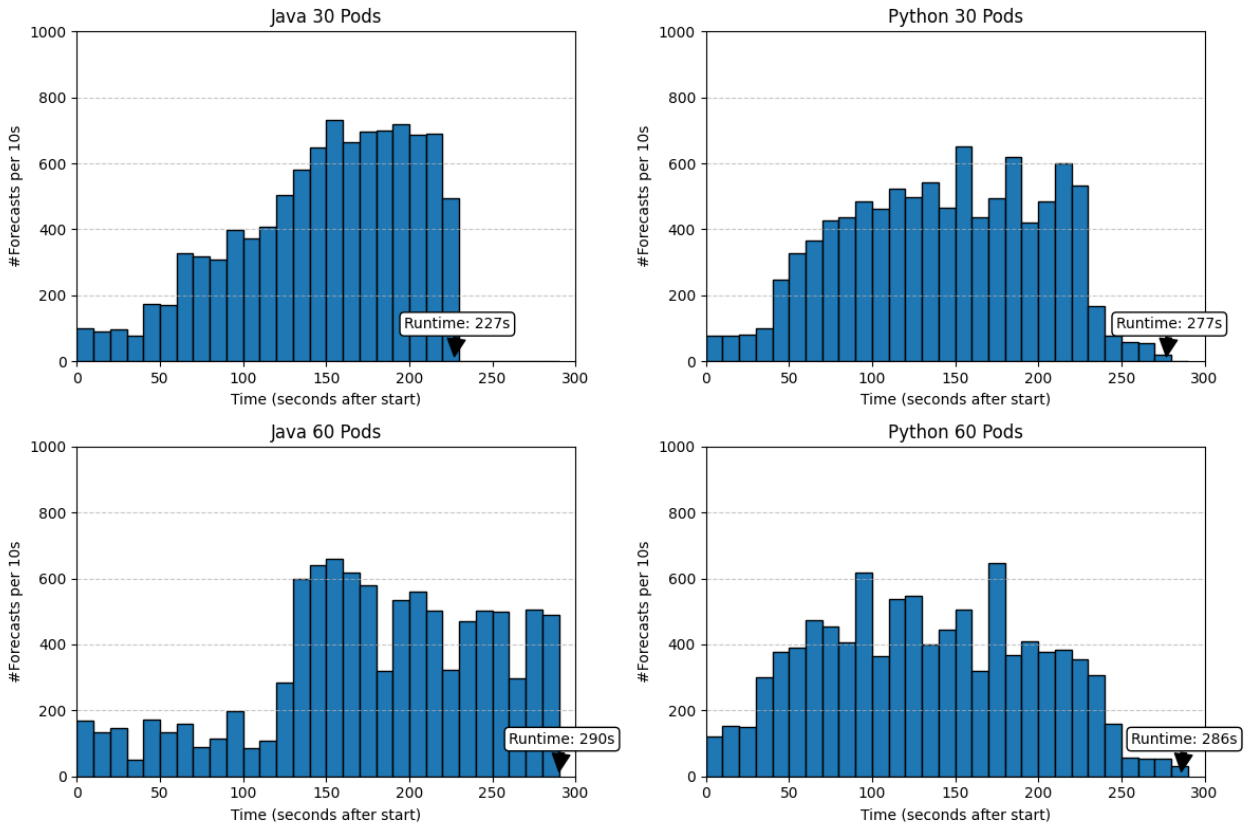


Figure 3 Throughputs of forecast generation in different implementations and scaling variants

It is noticeable that both the Java and Python implementations were faster with a maximum of 30 pods rather than 60 pods. This shows additional overhead caused by using a larger number of pods. One possible explanation for this is that requests are distributed by the Load Balancer to pods that are still starting. This means that some requests get stuck in a queue and are not instantly processed.

The Java variant with 60 pods in particular shows that the Java implementation has a longer start time. As a result, the Python variants earlier achieve a higher throughput. However, both Python variants show an inefficient behavior towards the end of the runtime, recognizable by a significant decrease in throughput. This could indicate an uneven distribution of requests across the pods, with pods that start earlier receiving a higher number of requests than those that become active later.

One advantage of the baseline variant is its 0% failure rate, meaning all 10,000 requests were processed without error. However, the failure rates in the scaling variants are also low. In the Java variant, they are 0.39% (30 pods) and 0.41% (60 pods), and in the Python variant, they are 2.58% (30 pods) and 2.2% (60 pods). These rates are acceptable, as missing forecasts can easily be recalculated.

Both implementations support dynamic loading of data and models, which enables automated model updates and scalable forecasting. As the models were trained in Python, it was necessary to put additional effort into the Java implementation with the Deep Java Library (DJL). In comparison, the implementation effort in Python was minimal. In addition, MLflow, which was used for model loading, offers a direct Python API that enables seamless integration.

For the Java environment, on the other hand, a custom solution had to be developed to load the models directly from the model database. These differences illustrate the additional complexity associated with the Java implementation.

6 Conclusions and future work

Based on MLOps principles, a concept for the continuous adaptation of forecast generation was developed in this article. This concept makes it possible to regularly retrain and commission individual models for a large number of RE power plants and can be expanded to include additional forecast objects.

The aim was to evaluate whether the proposed system design meets the time-critical requirements of a renewable energy forecasting system and can respond automatically to increasing load by scaling applications horizontally.

The inference applications in Java and Python achieved calculation times of less than 5 minutes for 10,000 forecasts. The failure rate during provision was less than 3%. The calculation times include several data retrievals from different data sources of the distributed system. Due to the short runtime and low failure rates, it can be stated that the proposed system design meets the requirements and represents a promising approach for provisioning a large number of highly specialized ML models for the energy industry.

The Java application scaling up to 30 pods had the shortest runtime. The fact that scaling up to 60 pods resulted in a

longer runtime shows that a higher pod limit does not necessarily lead to a higher throughput. However, it should be noted that the optimal number of pods depends on the number of forecasts. For example, we assume that for a higher number of forecast objects than used in our experiments the optimal number of pods is also higher. Furthermore, this optimum depends on the throughput of the application and the distribution of requests. Future work should investigate a procedure for optimizing scaling parameters. In addition to the maximum number of pods, future work could optimize further parameters in the HPA configuration like the requested CPU and RAM per pod.

As the inference applications used were in a technical proof-of-concept state, there is still room for optimization here too, which could further reduce calculation times. The results suggest that implementations in Java enable shorter processing times, particularly if the initial start time can be reduced. However, the long loading times of the DJL hinder the performance of the Java implementation. By switching to ONNX runtime instead of DJL, the application startup time can probably be reduced as only one ML framework is supported and its overhead is potentially lower. It is also possible to optimize the environment parameters of the applications, such as the maximum number of threads and the frequency of incoming requests.

Java and Spring Boot in particular are industry standards for the development of stable applications. There are other, specialized solutions for lightweight cloud-native applications. To further improve performance, a framework such as Quarkus, which is optimized for use with Kubernetes, could be utilized. Improvements could also be achieved using a Native Image which can be created utilizing GraalVM. GraalVM offers several advantages, including startup times in less than a second, as opposed to several seconds [28].

The use of multitask learning models means that the model artifacts consist of two components. One component is the generalized model, which forms the basis for all inferences and is applied to all power plant forecasts according to a hard parameter sharing approach. The second component is a specialized embedding that represents the specifics of each power plant. In the current experiment setup, both components were always loaded from the model store for inference. As further optimization, the generalized model can be cached so that only the embedding, which is many times smaller, needs to be loaded for inferences.

Finally, future work could make a comparison with KServe, a model-serving platform from the KubeFlow ecosystem. It is based on Kubernetes and supports many modern ML frameworks, including ONNX. It has many parallels to the solution presented in this article, but the KServe team describes a problem at higher scaling [29]. The Model-Mesh feature, which is intended to provide a solution, is still in the alpha stage [30].

7 Acknowledgements

This work is based on the results of the projects „KI-Servicezentrum für sensible und kritische Infrastrukturen“

(KISSKI) and „Kontinuierlich selbstlernende Vorhersagemethoden und Services in smarten Energiemärkten und -netzen“ (KonSEnz). KISSKI is funded by the Federal Ministry of Education and Research and KonSEnz is funded by the Federal Ministry of Economics and Climate Protection.

8 Literature

- [1] Agora Energiewende: Die Energiewende in Deutschland: Stand der Dinge 2024. Rückblick auf die wesentlichen Entwicklungen sowie Ausblick auf 2025, 2025
- [2] Sehne, F., Anton Kaifel, Marc Deissenroth, Elke Lorenz, Jan Dobschinski and Uwe Klann: Bedeutung von Prognosen für die Energiewende. Beiträge zur FVEE-Jahrestagung 2016 (2016)
- [3] Kloubert, M.-L., Schwippe, J., Muller, S. C. and Rehtanz, C.: Analyzing the impact of forecasting errors on redispatch and control reserve activation in congested transmission networks. in: IEEE Eindhoven PowerTech 2015. 2015, pp. 1–6
- [4] Ying, C., Wang, W., Yu, J., Li, Q., Yu, D. and Liu, J.: Deep learning for renewable energy forecasting: A taxonomy, and systematic literature review. *Journal of Cleaner Production* 384 (2023), 135414
- [5] Ahmed, R., Sreeram, V., Mishra, Y. and Arif, M. D.: A review and evaluation of the state-of-the-art in PV solar power forecasting: Techniques and optimization. *Renewable and Sustainable Energy Reviews* 124 (2020) 109792
- [6] Schreiber, J. and Sick, B.: Model selection, adaptation, and combination for transfer learning in wind and photovoltaic power forecasts. *Energy and AI* 14 (2023) 100249
- [7] Nivarthi, C. P., Vogt, S. and Sick, B.: Multi-Task Representation Learning for Renewable-Power Forecasting: A Comparative Analysis of Unified Autoencoder Variants and Task-Embedding Dimensions. *Machine Learning and Knowledge Extraction* 5 (2023) 3, pp. 1214–1233
- [8] Agrahari, S. and Singh, A. K.: Concept Drift Detection in Data Stream Mining: A literature review. *Journal of King Saud University - Computer and Information Sciences* 34 (2022) 10, pp. 9523–9540
- [9] He, Y., Huang, Z., Vogt, S. and Sick, B.: PrOuD: Probabilistic Outlier Detection Solution for Time-Series Analysis of Real-World Photovoltaic Inverters. *Energies* 17 (2024) 1, pp. 64
- [10] Dimolitsas, I., Spatharakis, D., Dechouniotis, D. and Papavassiliou, S.: AHP4HPA: An AHP-based Autoscaling Framework for Kubernetes Clusters at the Network Edge. *GLOBECOM 2022 - 2022 IEEE Global Communications Conference. IEEE 2022*, pp. 2566–2571
- [11] He, Y. and Sick, B.: CLear: An Adaptive Continual Learning Framework for Regression Tasks. *AI Perspectives* 3 (2021) 1, 109
- [12] He, Y., Henze, J. and Sick, B.: Continuous Learning of Deep Neural Networks to Improve Forecasts for

- Regional Energy Markets. IFAC-PapersOnLine 53 (2020) 2, pp. 12175–12182
- [13] Machine Learning Operations (MLOps): Overview, Definition, and Architecture, Kreuzberger, D., Kühl, N. and Hirschl, S., 2022
- [14] Subramanya, R., Sierla, S. and Vyatkin, V.: From DevOps to MLOps: Overview and Application to Electricity Market Forecasting. Applied Sciences 12 (2022) 19, 9851
- [15] Gürses-Tran, G. and Monti, A.: Advances in Time Series Forecasting Development for Power Systems' Operation with MLOps. Forecasting 4 (2022) 2, pp. 501–524
- [16] Subramanya, R., Räisänen, P., Sierla, S. and Vyatkin, V.: Cloud Computing Design Patterns for MLOps: Applications to Virtual Power Plants. IECON 2023-49th Annual Conference of the IEEE Industrial Electronics Society. IEEE 2023, pp. 1–7
- [17] Burgueño-Romero, A. M., Barba-González, C. and Aldana-Montes, J. F.: Big Data-driven MLOps workflow for annual high-resolution land cover classification models. Future Generation Computer Systems 163 (2025), 107499
- [18] Sigfridsson, P.: An Open-Source Framework for Large-Scale ML Model Serving. 1401-5749. UPTec IT 22033. 2022
- [19] The Kubernetes Authors: Kubernetes Documentation, 2024. <https://kubernetes.io/docs/home/>, accessed on: 14th January 2025
- [20] Alla, S. and Adari, S. K.: Introduction to MLFlow. in: Alla, S. and Adari, S. K. (Hrsg.): Beginning MLOps with MLFlow. Berkeley, CA: Apress 2021, pp. 125–227
- [21] Vogt, S., Braun, A., Dobschinski, J. and Sick, B.: Wind power forecasting based on deep neural networks and transfer learning. Proceedings 18th Wind Integration Workshop. 2019
- [22] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S.: PyTorch: An Imperative Style, High-Performance Deep Learning Library. in: Advances in Neural Information Processing Systems 32. Curran Associates, Inc 2019, pp. 8024–8035
- [23] ONNX Runtime developers: ONNX Runtime, 2021. <https://onnxruntime.ai/>, accessed on: 14th January 2025
- [24] Zängl, G., Reinert, D., Rípodas, P. and Baldauf, M.: The ICON (ICOsahedral Non-hydrostatic) modelling framework of DWD and MPI-M: Description of the non-hydrostatic dynamical core. Quarterly Journal of the Royal Meteorological Society 141 (2015) 687, pp. 563–579
- [25] Ramírez, S.: FastAPI, 2025. <https://fastapi.tiangolo.com>, accessed on: 14th January 2025
- [26] Broadcom: Spring Boot, 2025. <https://docs.spring.io/spring-boot/index.html>, accessed on: 14th January 2025
- [27] The DJL Community: Deep Java Library, 2025. <https://docs.djl.ai/master/index.html>, accessed on: 14th January 2025
- [28] Sipek, M., Muharemagic, D., Mihaljevic, B. and Radovan, A.: Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus. 2020 43rd International Convention on Information, Communication and Electronic Technology (MI-PRO). IEEE 2020, S. 1746–1751
- [29] The KServe Authors: The model deployment scalability problem, 2021. <https://kservice.github.io/website/latest/modelserving/mms/multi-model-serving/#benefit-of-using-modelmesh-for-multi-model-serving>, accessed on: 14th January 2025
- [30] The KServe Authors: ModelMesh Serving, 2021. <https://kservice.github.io/website/latest/modelserving/mms/modelmesh/overview/>, accessed on: 14th January 2025