Master's thesis

Master's Programme in Computer Science

Designing a Machine Learning Pipeline with Continuous Training for Time Series Forecasting

Jan Koskinen

June 5, 2024

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Jan Koskinen

Työn nimi — Arbetets titel — Title

Designing a Machine Learning Pipeline with Continuous Training for Time Series Forecasting

Ohjaajat — Handledare — Supervisors

Dr. Mikko Raatikainen, Dr. Saku Suuriniemi

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	June 5, 2024	50 pages

Tiivistelmä — Referat — Abstract

Machine Learning Operations (MLOps) emerged as a practice for applying DevOps practices and culture for machine learning (ML) systems to increase the speed and reliability of deployments. These practices include advocating for automation and monitoring at all steps of the ML system construction, including integration, testing, deployment, and infrastructure management. In addition to continuous integration (CI) and continuous delivery (CD), MLOps introduces continuous training (CT), which is unique to ML systems and is concerned with automatically training and serving ML models.

Operating ML systems in production requires continuously adapting to the evolving input data. This is especially evident in time series data, which can experience frequent drifts. Moreover, implementing CT in practice is challenging and heavily dependent on the task and available data. Depending on the complexity of the model and the amount of data, the training process can be computationally costly. Using a scheduled interval for retraining is inefficient if the model still performs adequately.

We designed an ML pipeline capable of efficient continuous training using an error-based trigger for retraining the model. The ML pipeline is designed for a time series forecasting task, where the data is prone to frequent drifts. We applied the design science research methodology to identify the problem, design and develop a solution artifact, and evaluate its utility and efficacy.

The resulting solution utilizes an open-source MLOps platform that runs on Kubernetes. The solution includes a custom retrainer component to enable CT. We demonstrated the efficacy of the solution using real energy demand data from a university property in Finland. Our evaluation shows that the system is capable of efficient continuous training.

ACM Computing Classification System (CCS)
Software and its engineering → Software creation and management → Designing software
Computing Methodologies → Machine Learning

Avainsanat — Nyckelord — Keywords

MLOps, Continuous Training, Machine Learning Pipeline

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction

With the rise in popularity of *machine learning* (ML), more and more companies are interested in deploying machine learning-based systems to production. However, getting a model into production requires more than just building the model. Building a model is only one step towards building an integrated ML system and continuously operating it in production (Google Cloud, 2020).

DevOps is a popular practice in software engineering for developing and operating software systems (Leite et al., 2020). DevOps is an approach that emphasizes collaboration and automation in order to accelerate the delivery of software changes. DevOps introduces continuous integration (CI) and continuous delivery (CD) in the development process in order to automate the process of building, testing, and deploying software changes reliably.

Since ML systems are software systems (Mikkonen et al., 2021), DevOps practices also apply to developing ML-based systems. However, developing ML-based systems is fundamentally different from developing "traditional" software since the rules of the model are indirectly set by capturing patterns from data instead of being explicitly programmed. Furthermore, an ML-based system is affected by changes on three levels: data, model, and code. Changes in data are rarely independent, for example, a change in the distribution of one model input feature might affect the importance, weight, or use of the remaining features in the model. The addition or removal of model features can cause similar changes. This interdependence is known as the CACE principle: Changing Anything Changes Everything (Sculley et al., 2015), which introduces new challenges in developing and operating ML systems in production.

MLOps (Machine Learning Operations) is an extension of the DevOps methodology to address the unique challenges associated with machine learning and data science (Google Cloud, 2020; Kreuzberger et al., 2023). In addition to testing and validating code and components, continuous integration should also include testing and validation of data and models. Continuous delivery should involve not only the deployment of software artifacts or services but rather a *machine learning pipeline* that automatically deploys a model prediction service. MLOps also introduces *continuous training* (CT), which is concerned with automatically retraining and serving the models.

Operating ML systems in production requires continuously adapting to the evolving input

data. This is evident in time series data which can experience frequent drifts. In most real-world time series applications data arrives sequentially as a stream, which may flow at high speed and evolve over time, making static models ineffective and inappropriate (Oliveira et al., 2017).

Continuous training requires an ML pipeline to automate the process of delivering ML models to production (Kreuzberger et al., 2023). These pipelines encompass several components, from data processing to training and serving the ML model. Additionally, continuous training needs a trigger to run the pipeline and automatically retrain the model with new data. This trigger can be invoked manually on demand or automatically, for example, by a scheduled interval. However, retraining a model that is still performing well can be costly, especially for complex models that require large amounts of data (Wu et al., 2020; Derakhshan et al., 2019). Therefore, continuous training should be efficient and occur only when necessary. Moreover, implementing continuous training in practice can be challenging, requiring expertise in software engineering, data science, and domain-specific knowledge related to the task the model aims to solve.

This thesis addresses the problem of designing an ML pipeline for time series forecasting with efficient continuous training. We designed and implemented an ML pipeline that utilizes an open-source MLOps platform and employs a custom error-based retraining trigger. We demonstrate its ability to continuously train and serve a model for forecasting energy consumption using real energy consumption data. Our evaluation shows that the ML pipeline triggers retraining more efficiently than the scheduled interval-based approach.

The following is an outline of this thesis. In Chapter 2, we describe ML pipelines in more detail, the requirements for continuous training, and the role of model monitoring. Chapter 3 explains the research methodology used in the thesis. In Chapter 4, we describe and formalize the research problem, and define objectives for the design. In Chapter 5, we describe the design for the ML pipeline and the resulting implementation. In Chapter 6, we demonstrate the ML pipeline and evaluate it against our objectives. In Chapter 7, we revisit the research problem and discuss our contributions, limitations, and related work. Chapter 8 contains our conclusions and suggestions for future work.

2 Background

2.1 ML Pipeline Automation

An ML pipeline is a way to codify and automate the process of delivering an ML model to production. The process of delivering an ML model to production requires various steps from data extraction and preprocessing to model training and deployment. These can be completed manually or by an automated pipeline. These steps vary depending on the ML task, but commonly required steps include *data extraction, data analysis, data preparation, model training, model evaluation, model serving*, and *model monitoring* (Google Cloud, 2020).

Data extraction includes selecting and integrating the relevant data from various sources for the ML task. Data analysis refers to the exploratory data analysis required to understand the available data and to identify relevant characteristics needed for the model. Data preparation includes cleaning and validating the data, and splitting the data into training, validation, and test sets. In addition, it may include *feature engineering* which includes transforming the data into feature vectors that can be directly used as inputs for the model.

Model training includes using the prepared data to train various ML models and tuning hyperparameters to maximize their performance. Model evaluation includes calculating performance metrics using the test data set to assess the quality of the model. The test data set is produced during data preparation and never revealed for model training. Model validation is concerned with verifying that the model is adequate for deployment and meets a certain performance baseline. Model serving refers to deploying a model to a target environment to provide predictions. This deployment can take the form of a web server that delivers predictions via HTTP or be directly embedded in an edge device, such as a mobile phone. Model monitoring includes monitoring the predictive performance of the model to potentially invoke a new iteration of the process.

The level of automation for these steps represents the maturity of the MLOps practice in a team (Google Cloud, 2020). Developing an ML pipeline to automate as many steps as possible increases the reliability and velocity of delivering an ML model to production. A

manual process may be sufficient for teams that are beginning to apply ML for a use case or experimenting with a prototype. However, when a model requires frequent updates or multiple models are needed, a manual approach becomes insufficient. When moving from the manual process toward automation and MLOps practices, the focus of the deployed software artifact shifts from the model to the ML pipeline itself.

Automating the ML pipeline allows for continuous training, which incorporates retraining of models in production with new data samples. Retraining facilitates continuous delivery for model serving and keeps the model up to date. In addition to pipeline automation, a *retraining trigger* is required to invoke the pipeline. The retraining trigger may be invoked based on several conditions depending on the use case. Table 2.1 describes examples of retraining triggers utilized for continuous training. These conditions may also be combined to create smarter retraining triggers.

Table 2.1: Retraining triggers (Google Cloud, 2020).

Trigger	**Description**
Manual	Ad-hoc manual execution of ML pipeline
Scheduled interval	Retraining is triggered periodically on a daily, weekly, or monthly basis.
New data available	Retraining is triggered when new data is collected and made available.
Performance drop	Retraining is triggered when there is a noticeable performance degradation.
Data distribution shift	Retraining is triggered based on significant changes in the data distributions of the features used for predicting.

The following key technical components have been identified for implementing an ML pipeline that adheres to MLOps practices (Kreuzberger et al., 2023). A CI/CD component to take care of the build, test, and deploy steps to increase the speed and reliability of changes. A source code repository for storing, versioning, and collaboration. A workflow orchestration component for orchestrating tasks in the ML workflow. A feature store to provide a central storage of commonly used features. Model training infrastructure to provide computation resources (e.g., CPUs and GPUs) either in a distributed or non-distributed setting. A model registry to centrally store ML models along with their metadata. An ML metadata store for tracking various metadata during the execution of an ML pipeline. A model serving component to provide model predictions via e.g., a web

server. A monitoring component to continuously monitor the model performance and the underlying infrastructure.

The functionality of these components is provided through different software tools, which are integrated to form an ML pipeline. These components may also be provided through MLOps platforms such as SageMaker[1], which reduce the operational complexity of building the ML pipeline.

2.2 Model Monitoring and Retraining

As with most software systems, ML systems require monitoring to keep models running reliably in production (Breck et al., 2017). Monitoring gives visibility into a system, which is a core requirement for judging service health and diagnosing when things go wrong. Monitoring can provide early warnings for the myriad of things that can go wrong with systems running in production.

Effective monitoring includes finding and visualizing the important metrics to monitor and possibly triggering an alert for the developers of the model to react to. A well-balanced threshold for triggering an alert is important for finding actual problems in production without alerting for normal activity. Google's Site Reliability Engineering book (Beyer et al., 2016) provides some common reasons for monitoring: analyzing long-term trends, comparing over time or experiment groups e.g. A/B testing, alerting on conditions that require attention, displaying information about the system visually, and investigating and diagnosing issues.

Typical software systems that make requests over the network and communicate with different processes usually monitor for metrics such as latency, traffic, system load, and errors. ML systems require monitoring for additional model-specific areas. These areas include model performance, metrics related to incoming data, detecting outliers and drift, and explaining predictions (Klaise et al., 2020). ML models are complex pieces of software that are often required for cases where the desired behavior can not be effectively expressed in logic with a programming language. Maintaining ML systems over time can be difficult and costly (Sculley et al., 2015). Understanding ML models requires specific knowledge and the metrics for monitoring typical software systems are not sufficient.

Unlike programs with instructions manually written by a programmer using a program-

[1]https://aws.amazon.com/sagemaker/

ming language, a model-based program's source code can't be similarly inspected to determine its behavior. An ML model can be viewed as an approximation of a function f that can predict an output Y based on the input variables, also known as predictors or features $X = (X_1 \dots X_n)$. This can be written in a general form $Y = f(X) + \epsilon$, where ϵ refers to the error of the approximation due to noise or unmeasured variables (James et al., 2014). The objective of the model is to approximate the unknown function f as closely as possible.

A typical parametric machine learning model learns its parameters from data through *training*. Training typically involves using numerical optimization methods to find the parameters that minimize the value of a given cost function, which indicates the performance of the model on the training data set. The learned parameters are typically real numbers that are used in conjunction with the input features to approximate a model f. For example, in a simple linear regression model $Y = \beta_0 + \beta_1 X$, the parameters $\beta_0, \beta_1 \in \mathbb{R}$ are some real numbers that determine the slope and intercept of the regression line. In more complex models the number of parameters can be large, especially in the case of neural networks. These parameters are not human-interpretable in general, like instructions in a programming language, and it is difficult to determine the underlying behavior of a model. An ML model performs well when it can predict the correct output Y from a previously unseen set of features $X_1 \dots X_n$ with an acceptable margin of error for the given task.

An ML model's performance is dependent on the quality and integrity of the data that the model is trained on, and the data it receives as input (Breck et al., 2019). A model might be carefully validated and tested before deployment and still contain unexpected skews between the training and live data in a production environment. These data skews are sometimes referred to as *training-serving skew* and can happen for multiple reasons (Breck et al., 2017). For example, the distribution of features might differ between training data and live data due to incorrectly designing the training data or due to differences between data sources in the training and production environments.

Even though a model performs well after deployment, it's not guaranteed to continue performing well as time passes. Data might change due to changes in an external data source that is used for feature extraction, or other unforeseen events in the real world that could change things significantly e.g. COVID-19. This performance degradation over time is sometimes referred to as *model decay* or *model staleness*. Data is often dynamic in the real world and prone to data drifts that can happen gradually or abruptly (Gama et al., 2014). Models that are not adapting to data changes over time are static and become

stale over time.

One of the key features of ML systems is feedback loops where the outputs of a model together with the feedback from users can be used as examples for training data to the same model or another model entirely (Sculley et al., 2015). These direct feedback loops influence the behavior of the model over time based on the feedback from users. This can improve the performance of models over time, for example when predicting the click-through rate (CTR) of news headlines on a website where outcomes of users clicking a headline or not can be collected as labels for training data. Feedback loops can also negatively influence the behavior of models if the quality of the feedback is poor, biased, or the feedback is formed with malicious intent. Feedback loops can be difficult to debug since the performance of the model might improve in aggregate, but perform significantly worse for some subgroups of users. Feedback loops could also be hidden, where two systems influence each other indirectly through the world, for example, two independent stock-market prediction models where improvements or bugs in one system might influence the bidding and buying behavior of the other (Sculley et al., 2015).

Invalid data, data skews, model staleness, and problematic feedback loops are some of the model-specific problems that can cause issues after deploying models in production. In order to understand how a model is performing in a production environment, monitoring performance metrics such as error metrics, precision, and recall is essential (Klaise et al., 2020).

For regression tasks, performance can be evaluated by comparing the predicted values with observed values using some error-based metric, for example, the mean squared error (MSE). There are plenty of error metrics for evaluating performance in the literature and each metric is more or less suitable depending on the task at hand (Klaise et al., 2020). There is no silver bullet error metric for all cases since the error metric is a statistical measure that summarizes data into a single value that emphasizes certain aspects of the model performance.

For classification tasks precision, recall, and F_1 score are common metrics for evaluating performance. Precision is defined as the ratio of correct positive predictions to the overall number of positive predictions. Recall is the ratio of correct positive predictions to the overall positive examples. Depending on the classification task, one might prefer higher precision over recall and vice versa. F_1 score combines precision and recall and values both equally. F_1 score is defined as the harmonic mean H of precision and recall. The harmonic mean of precision and recall makes sure that significant differences between precision and

recall result in a lower value than the arithmetic mean would.

These metrics are also typically used for offline evaluation during the training process. However, these metrics for performance require some ground truth to be available to which the outputs of the model can be compared against. For some tasks, ground truth is available, or can be derived from feedback loops or manual labeling processes. These include, for example, short-term time series forecasting, and recommendation with the outcome from users as a feedback loop. Manually generating labels for predictions or some subset of predictions can work with small data sets, but is extremely hard to scale for larger data sets and requires a lot of manual work. Even though the availability of ground truth is important for performance monitoring, the set of available ground truth values can be biased or include invalid data which may negatively affect the model.

For many tasks, the ground truth might be unavailable or delayed such that it would take too long to know how a model is performing in the production environment (Klaise et al., 2020). Setting up a feedback loop or a labeling process can be too expensive or infeasible. For these cases, monitoring some *proxy metrics* for performance is critical. Proxy metrics can give some indicators of how the model is performing in the production environment, usually by correlating with poor model performance. These include, for example, monitoring the statistics of input and output data or defining some task-specific metrics that are correlated with poor performance. The latter could include, for example, monitoring the toxicity or repetitiveness of generated text from a language model.

Monitoring the input and output data by determining if the distributions have drifted is a signal that the model might not be performing well. However, when drifts occur between the inputs or outputs of training data and production data, it might not mean that the performance has degraded. In other words, the relationship between the model's inputs and outputs has not changed even though the distribution of the inputs $P(X)$ or outputs $P(Y)$ alone has drifted. If the relationship between the inputs and outputs $P(Y|X)$ has changed, then the model is outdated and will need to be retrained or changed entirely. This change in the conditional probabilities is known as *concept drift*. Concept drift can occur with or without a change in $P(X)$ and $P(Y)$ (Gama et al., 2014). By monitoring the drift of input and output distributions, some cases of concept drift can be caught and some drifts will lead to false alarms.

There are numerous metrics that can be useful for monitoring ML models, but it is crucial to prioritize those relevant to the specific task. The ML Test Score (Breck et al., 2017) rubric presents seven monitoring tests that are recommended for ML systems in produc-

tion. These tests are described in Table 2.2 and focus primarily on supervised ML systems that are trained continuously and perform low-latency inference on a server.

Table 2.2: Seven monitoring tests for ML systems (Breck et al., 2017).

Test	Description
Test 1	Dependency changes result in a notification.
Test 2	Data invariants hold for inputs.
Test 3	Training and serving are not skewed.
Test 4	Models are not too stale.
Test 5	Models are numerically stable
Test 6	Computing performance has not regressed.
Test 7	Prediction quality has not regressed.

In addition to understanding how a model is performing in a production environment, monitoring can be utilized to retrain the model. Model retraining should ideally occur once the performance of the model is no longer sufficient and the model would benefit from retraining using newer data samples. To evaluate model performance, effective monitoring is necessary.

Monitoring enables identifying when a model becomes stale, and retraining is a method for the model to adapt to the change. Retraining involves training a new model version using the same model configuration and hyperparameters but including newer data samples or excluding previous data samples. However, retraining may not always be sufficient for adapting to the change and might require using different model configurations or another model entirely.

Retraining can be triggered by utilizing monitoring to determine when the model should be retrained. It may also be triggered manually or periodically using a scheduled interval without monitoring. However, monitoring can help to determine when and how often the model should be retrained. Whether retraining is manual or automated, effective monitoring remains crucial.

3 Research Approach

This thesis follows the design science research methodology (DSRM) for research in information systems (Peffers et al., 2008). DSRM is used to create and evaluate IT artifacts to solve an identified problem in information systems. The process is iterative and typically includes seven activities described in Figure 3.1 for identifying the problem, designing and developing a solution artifact, and evaluating its utility and efficacy. We chose a design and development-focused approach in the DSRM process. The following describes the key activities of the DSRM process in this thesis.

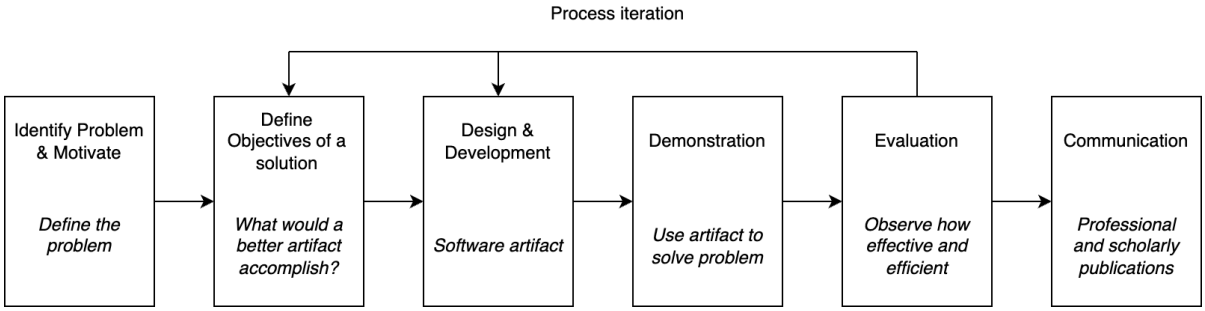


Figure 3.1: DSRM process (Peffers et al., 2008)

Problem identification and objectives. We present the motivation and context for the problem in Chapter 1, along with the background information in Chapter 2. We describe the research problem and formalize the research question. We derived the objectives for the solution from the identified problem and context. We list each objective that guided us in the design and development and describe their relevance to the problem. This activity is described in Chapter 4.

Design and development. We describe our design considerations and desired functionalities for the implementation. We present a conceptual solution architecture for the implementation. We then describe our implementation which was guided by the defined objectives and design. This activity is described in Chapter 5.

Demonstration and evaluation. We demonstrate the implementation through an experiment, detailing the experiment setup and presenting the results. We then evaluate the

implementation and demonstration results against the defined objectives. This process is described in Chapter 6. Finally, we reflect on the research problem and discuss the extent to which our contributions addressed it in Chapter 7.

For this research, we have been provided with a dataset containing real energy consumption data from several university properties in Finland. The dataset includes data from nine university campus buildings. It captures variations in energy consumption caused by factors such as seasonal changes in outdoor temperatures and fluctuations in building usage. A detailed description of the dataset is provided in Section 6.1. This dataset offers additional context for the activities in the DSMR process.

4 Problem and Objectives

4.1 Problem Identification and Motivation

Data in the real world is rarely static and often prone to noise and irregular drifts. The drifts may occur suddenly or over time, and the data can include invalid or missing values. This can be due to, for example, modeling non-stationary environments or the brittleness of sensors in the physical world. Models are typically trained on historical data representing a snapshot of the real world at a certain point in time. However, the world may change for various reasons and the snapshot becomes obsolete over time.

Online learning methods enable models to adapt to new data and stay current by incrementally updating the model with incoming data. However, online learning is sensitive to noise and outliers, requiring tuning for the specific use case to guarantee a high level of quality (Derakhshan et al., 2019).

A simple and robust way to address this problem is to retrain the model with historical data in addition to new data. Continuous training aims to automate the process of retraining the model in a reliable manner. Achieving continuous training in practice is challenging and heavily dependent on the task and available data. In some cases, the data might be prone to frequent drifts, the training data set might initially include only a few examples, or there might be a significant delay in the availability of ground truth. Furthermore, it may not be acceptable to simply implement continuous training with a scheduled interval for retraining. Depending on the complexity of the model and the amount of data, the training process can be computationally costly. Retraining the model can be quite wasteful if the model is still performing well.

Retraining only once the performance drops below an acceptable threshold is more efficient. We refer to this as the *error-based retraining trigger*. Additionally, it may be combined with other retraining triggers described in Table 2.1. However, it requires an acceptable delay for ground truth data and choosing the appropriate metrics and thresholds for the error.

There are plenty of options for choosing the appropriate signal that triggers retraining, metrics for evaluating the performance and feasibility of the model, and criteria for valid

data. Furthermore, abundant options exist for choosing appropriate tools for an ML task. These tools need to be integrated to form a cohesive system. Although platforms already exist for creating ML pipelines with seamless integration, they are mostly proprietary and pose a threat of vendor lock-in and reduced transparency. Therefore, it is the problem of utilizing the relevant methods and tools for the task to create an ML pipeline capable of efficient continuous training.

Frequent drifts are common in real-world time series data and the need for models to adapt to changes is crucial (Oliveira et al., 2017). This is evident in the energy consumption data where the energy consumption varies depending on the utilization of the buildings and the different seasonalities in Finland. Building a useful model that can forecast the energy consumption of properties over time requires continuously updating the model with new data. Furthermore, forecasting the energy consumption of a property with only a few initial data samples available exacerbates the need for the model to adapt to new data. Therefore, the research problem being addressed in this thesis is:

> *How to design and implement an ML Pipeline for time series forecasting with efficient continuous training using open-source tools?*

We applied the design science research framework to design and develop a software artifact. The resulting software artifact includes a model for forecasting energy consumption of university properties in Finland and an ML pipeline capable of continuously training and deploying the model.

4.2 Objectives of the Solution

The objective of the solution is to create an ML pipeline that is capable of continuous training. This includes automated retraining and deployment of the model in order to prevent model performance degradation and adapt to the evolving data over time. We prioritize the continuous training aspect of the solution over achieving optimal performance in forecasting energy consumption. However, our goal is for the model to remain useful and to maintain or enhance its performance through continuous training.

We list the following key objectives that guided us in the design and implementation of the solution in Table 4.1.

Each objective listed in the table aims to identify the necessary elements for designing and implementing a solution that addresses the research problem. The steps for delivering

Table 4.1: Solution objectives.

Objective	Description
Objective 1	The data preparation, model training, deployment, and retraining trigger should be automated.
Objective 2	Retraining should be efficient – we want to retrain only once the model's performance starts degrading and minimize the computational resources.
Objective 3	Model performance can be continuously inspected – we want to visually inspect how the model is performing for monitoring and troubleshooting purposes.
Objective 4	The model should be practical, interpretable, and reasonably accurate, and it should not perform worse than the existing model after retraining.
Objective 5	The solution should be portable, cloud platform-agnostic, and consist of open-source components.

a machine learning model to production should be automated through an ML pipeline (**Objective 1**). This pipeline must incorporate efficient continuous training, which should initiate retraining only when the model's performance drops below an acceptable level (**Objective 1, 2**). Retraining should be efficient to reduce the costs incurred from training the model (**Objective 2**). To enable retraining based on model performance, the system must monitor the model's performance using appropriate error metrics (**Objective 3**). Proper validation and evaluation steps are essential for continuous training to reliably retrain the model with new data (**Objective 3**). For accurate time series forecasts, it is crucial to understand the model and be able to configure and optimize it as needed (**Objective 4**). The ML pipeline should be built using open-source tools (**Objective 5**).

5 Design and Implementation

5.1 Conceptual Solution Description

We describe our design considerations and the desired functionalities of the solution based on the objectives defined in Table 4.1. We translated the solution objectives into functionalities that should be considered in the design and implementation of the solution. Table 5.1 describes the defined functionalities.

Table 5.1: Solution objectives translated to functionalities.

Functionality	**Description**
Functionality 1	A pipeline for training and deploying models. The pipeline should include tasks for data collection, preprocessing, training, evaluation, and deployment.
Functionality 2	An error-based retraining trigger that runs the pipeline to train and deploy a new model version.
Functionality 3	A monitoring service to inspect the model's performance and for troubleshooting purposes.
Functionality 4	An inference service that utilizes the trained model to provide forecasts.
Functionality 5	All implementation components are containerized and managed by a container orchestration service.

We ended up with four core components in the design. Training and deployment pipeline (**Functionality 1**), retraining trigger (**Functionality 2**), monitoring service (**Functionality 3**), and inference service (**Functionality 4**). The components are containerized applications that communicate within a network boundary and are managed by a container orchestration service (**Functionality 5**). The container orchestration service can consist of multiple nodes in a cluster or a single node and automates the deployment, scaling, and management of the containerized applications. The components are only able to communicate within the boundary through network isolation, however, requesting the forecast from outside of the boundary can be routed to the inference service.

We describe the conceptual solution description in Figure 5.1. The conceptual solution is

an ML pipeline capable of continuous training.

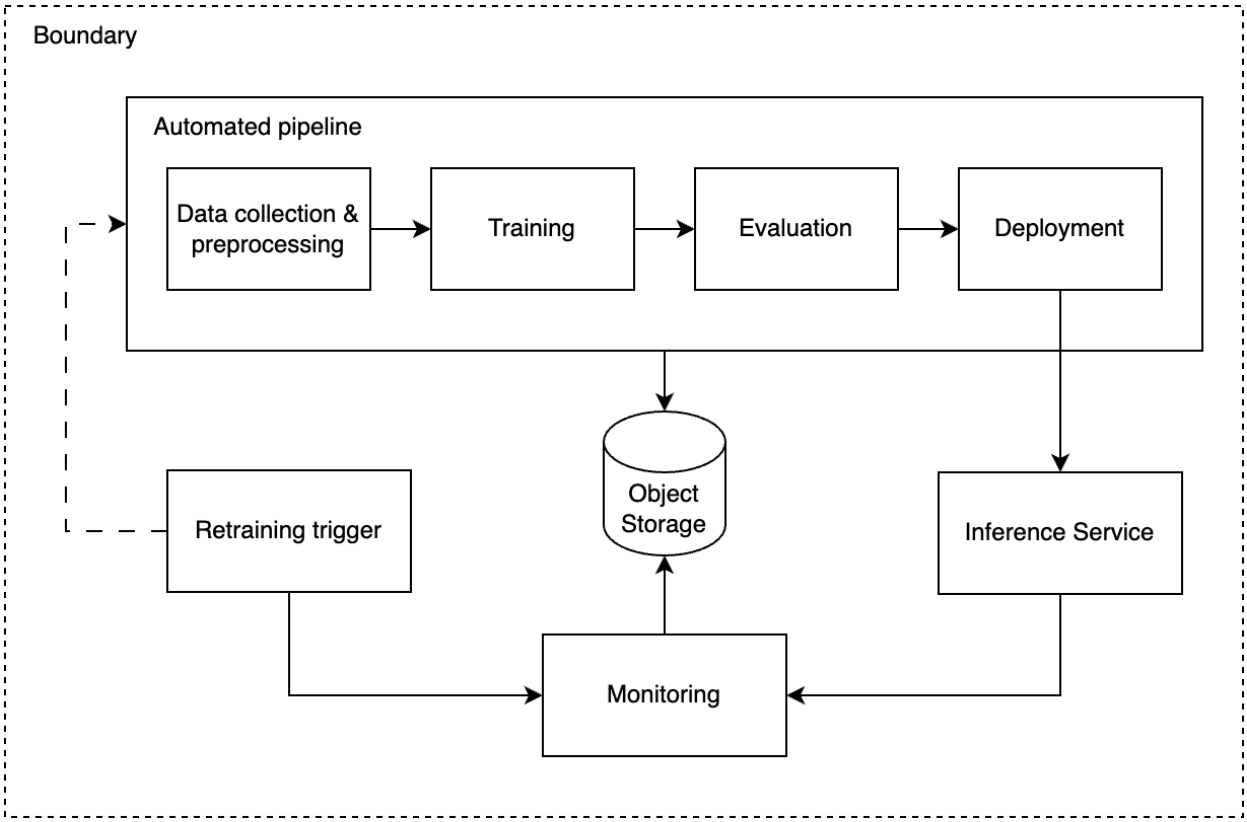


Figure 5.1: Conceptual solution description

The training and deployment pipeline includes four tasks that are independent components. Each task has a single responsibility, which makes it easier to reason about and verify its functionality. The pipeline runs the tasks sequentially and succeeds only if all of the tasks succeed.

The model is served via an *inference service* to ensure flexibility and ease of integration with other components. The inference service is capable of serving multiple models simultaneously. This can be useful when each property requires its own model for forecasting the energy demand.

The *monitoring service* receives and stores the prediction data produced by the inference service. It continuously checks for the availability of ground truth data for the predictions and calculates error metrics to estimate the model's performance. These error metrics are stored for each forecast and can be inspected visually. We assume that ground truth data from e.g. sensors is available in the object storage within a reasonable delay.

The retraining trigger includes a recurring job that fetches the error metrics from the

monitoring service to determine if the model should be retrained. If the error metrics exceed a certain threshold, then the pipeline will be executed, training a new model with the latest data and deploying it if it meets performance standards.

5.2 Selected Software Components

5.2.1 Kubernetes

Based on **Functionality 5** we used Kubernetes[1] (K8s) for the platform architecture and compute layer. Kubernetes is an industry-standard open-source platform for automating the deployment, scaling, and management of containerized applications. The Kubernetes compute layer can include CPU and GPU resources, and can run on local machines, clusters, and is supported by the majority of the cloud providers.

Kubernetes architecture is designed to manage containerized applications at scale. It consists of several key components that work together to ensure applications run reliably and efficiently. We outline the following Kubernetes concepts to enhance understanding of the platform.

Cluster. Kubernetes is deployed in a cluster which consists of a set of nodes that run containerized applications managed by Kubernetes. A cluster has at least one worker node and the control plane.

Control Plane. The control plane is responsible for managing the state of the cluster, including workload scheduling and responding to various cluster events, such as starting a new pod when one goes down. Key responsibilities of the control plane include handling API requests, storing cluster data, assigning workloads to nodes, and managing controllers to perform regular maintenance and operational tasks. Controllers are control loops that watch the state of the cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state.

Node. A node is a worker machine, either physical or virtual, in the Kubernetes cluster that runs containerized applications. Each node is managed by the control plane and includes services to ensure that containers are running correctly and to maintain network rules for pod communication.

Pod. A pod is the smallest deployable unit in Kubernetes. A pod represents a single

[1]https://kubernetes.io

instance of a running process in a cluster. Pods can contain one or more containers (usually one) that share storage, network, and the specification for how to run the containers. Containers are lightweight, executable software packages that include everything needed to run a piece of software, including the code, runtime, system tools, and libraries.

Deployment. A deployment provides declarative updates to applications. It describes the desired state of an application and the Kubernetes control plane makes sure that the actual state matches the desired state in the cluster. Deployments manage the creation and scaling of pods.

Service. A service is an abstraction that defines a logical set of pods and a policy for accessing them. Services enable load balancing and service discovery, allowing applications to communicate without needing to know the exact location of the pods.

A Custom Resource Definition (CRD). A CRD allows users to extend Kubernetes capabilities by defining their own resource types. CRDs enable the creation and management of custom resources that behave like built-in Kubernetes resources. By using CRDs, Kubernetes becomes a highly extensible platform, allowing developers to create applications that fit their specific needs.

5.2.2 OSS MLOps Platform

Based on **Functionalities 1, 3, 4, and 5** in Table 5.1 we decided to utilize an existing open-source MLOps platform. The OSS MLOps platform[1] (IML4E, 2023) consists of multiple open-source software components for model training, management, deployment, and orchestration and is run on top of Kubernetes. The components of the platform are described in Figure 5.2.

The platform does not include an error-based retraining trigger described in **Functionality 2**. We implemented a custom retraining component which is integrated into the platform. The component is described in Section 5.6.

The OSS platform contains the following core components.

Kubeflow Pipelines[2] is the orchestrator for the ML pipeline. Kubeflow pipelines enable the orchestration and automation of ML pipelines on top of Kubernetes. It enables managing dependencies between different execution steps in an ML pipeline and the execution

[1] https://github.com/OSS-MLOPS-PLATFORM/oss-mlops-platform
[2] https://www.kubeflow.org

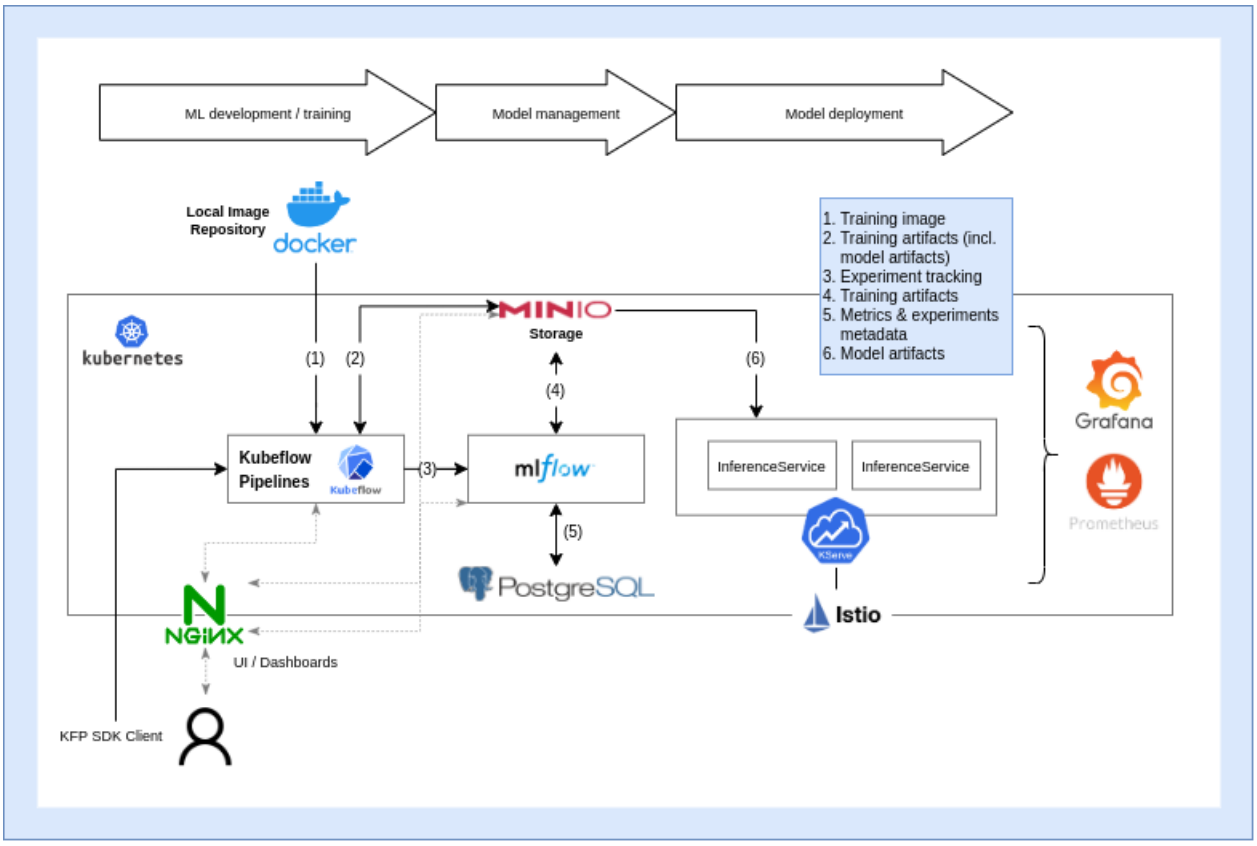


Figure 5.2: OSS MLOps Platform high-level architecture

of multiple ML pipelines in parallel.

MLflow[1] is used as an experiment tracker and model registry. Mflow provides both an API and dedicated UI for metadata tracking during the execution of an ML pipeline and enables the comparison of different ML experiments. In addition, it offers a model registry that supports the handling of different versions of models.

KServe[2] is the inference engine used for the deployment of ML models behind HTTP APIs. KServe provides a standardized inference protocol across ML frameworks, high scalability, and advanced deployments with canary rollout and experiments. KServe originated from the same project as Kubeflow pipelines and runs on top of Kubernetes.

In addition to the core components, the platform includes additional components for storage, monitoring, and routing.

Nginx[3] is used as a reverse proxy for the OSS platform. It exposes HTTP and HTTPS

[1]https://mlflow.org
[2]https://kserve.github.io/website
[3]https://www.nginx.com

routes from outside of the Kubernetes cluster to services within the cluster.

Docker registry[1] is used for hosting container images that are loaded into the Kubernetes cluster.

Grafana[2] **& Prometheus**[3] are used for monitoring the health of the OSS platform and for monitoring ML models. Prometheus records metrics in a time series database and supports flexible queries and real-time alerting. Grafana provides an interactive visualization of the Prometheus metrics in a dashboard web application.

MinIO[4] is used for providing object storage for the OSS platform. MinIO includes AWS S3 compatibility for storing data and is used for storing model binary files and data sets.

5.2.3 NeuralProphet

Based on **Functionality 4** in Table 5.1 and **Objective 4** in Table 4.1 we chose NeuralProphet (Triebe et al., 2021) as the model due to its practicality and user-friendly customization that enable it to fit intricate non-linear patterns.

NeuralProphet is a successor to the popular Prophet (Taylor and Letham, 2017) model, which set an industry standard for explainable, scalable, and user-friendly forecasting frameworks. NeuralProphet combines the classic time series components introduced by the Prophet package with ML-based modules enabling it to fit non-linear patterns.

The model is composed of multiple modules which each contribute an additive effect to the forecast. All model modules can be configured or switched off, and combined to compose the model. The modules must produce h outputs, where h denotes the number of steps to be forecasted in the future. The model components can be described as:

$T(t)$ = Trend at time t

$S(t)$ = Season at time t

$E(t)$ = Event and holiday effect at time t

$F(t)$ = Regression effects at time t for future-known covariates

$A(t)$ = Auto-regression effects at time t based on past observations

$L(t)$ = Regressions effects at time t for lagged observations of covariates

[1] https://docs.docker.com/registry
[2] https://grafana.com/
[3] https://prometheus.io
[4] https://min.io

Thus a one-step-ahead forecast where $h = 1$ can be formed as:

$$\hat{y}_t = T(t) + S(t) + E(t) + F(t) + A(t) + L(t)$$

The trend T is modeled using a classical approach as the combination of an offset m and a growth rate k. The trend effect at time t_1 is given by multiplying the growth rate by the difference in time since the starting point t_0 on top of the offset. The trend at t_1 can be described as: $T(t_1) = m + k \cdot (t_1 - t_0)$.

The seasonality S is modeled with the help of Fourier terms as was Originally done in Prophet. The amount of Fourier terms is defined for each seasonality depending on the periodicity of the season, for example, yearly seasonality with daily data. Fourier terms are defined as sine and cosine pairs and are a great tool for modeling seasonality as they produce smooth functions that can fit complex seasonal patterns.

Events and holidays E are modeled with a binary variable e that signals whether or not the event occurs at a particular time t. An event e at time t can be described as $E(t) = ze(t)$, where z denotes the model's coefficient corresponding to the event e.

Future regressors F include both past and future values of covariates. The future values of the covariates are known or may be predicted. For example, the next day's weather forecast may be used as a future value.

Auto-regression (AR) A refers to the process of regressing a variable's future value against its past values. A coefficient is fitted for each past value that controls the direction and magnitude of the effect of a particular past value on the forecast. The AR module is based on a modified version of the AR-Net (Triebe et al., 2019) model. AR-Net uses a feed-forward neural network approach for learning the coefficient values with a configurable number of hidden layers.

Lagged regressors L are used to correlate past observations to the target time series. For example, the temperature forecast of the previous days might be a good predictor for the temperature of the next day. Given a set of covariates, a separate lagged regressor is created for each covariate from the last n observations.

NeuralProphet is trained using the stochastic gradient descent (SGD) optimization strategy, enabling it to be scaled reliably to large datasets, and accommodate more complex model modules (Triebe et al., 2021). Users can specify various training hyperparameters to optimize the training process. However, all training-related hyperparameters come with default values automatically adjusted based on the dataset size, enhancing user-friendliness for ML practitioners.

NeuralProphet and Prophet perform nearly identically in their default configuration. However, NeuralProphet forecasts improve substantially when configuring it with any amount of auto-regression or forecasting multiple steps ahead (Triebe et al., 2021).

NeuralProphet provides a Python module that utilizes the popular PyTorch[1] machine learning library. The module provides a user-friendly API through the `NeuralProphet` class which enables configuring the aforementioned modules such as trend, seasonality, and auto-regression.

5.3 Implementation Architecture

The architectural overview of the implementation is illustrated in Figure 5.3. We utilized the selected software components to construct an ML pipeline with continuous training capabilities. The architecture outlines both the components provided by the MLOps platform and our own contributions. Our contributions include building the pipeline for training and deploying models using Kubeflow, developing a custom Retrainer component, implementing model serving functionality, and integrating all components to form a cohesive system. We provide a more detailed explanation of our contributions in the following sections.

Once the Kubernetes cluster for the MLOps platform is running, a developer can run a Python module to launch the pipeline with the appropriate configuration. Before executing the Python module, the container images for each pipeline step must be built and pushed to the container image repository provided by the MLOps platform. The implementation includes a script for building a container image, which is used in each pipeline step for simplicity and a faster building process.

The pipeline trains a NeuralProphet model with a data set available in object storage and deploys an inference service for serving the model. In addition, the Retrainer is deployed for automatically retraining the model based on its performance.

Applications that require model forecasts can integrate with the inference service and send HTTP requests to obtain the predictions. In addition to the Grafana dashboard for monitoring, Kubeflow, MLflow, MinIO, and Prometheus offer web-based UIs for administrative tasks.

We assume that the ground truth data for energy consumption will eventually be made

[1] https://pytorch.org/

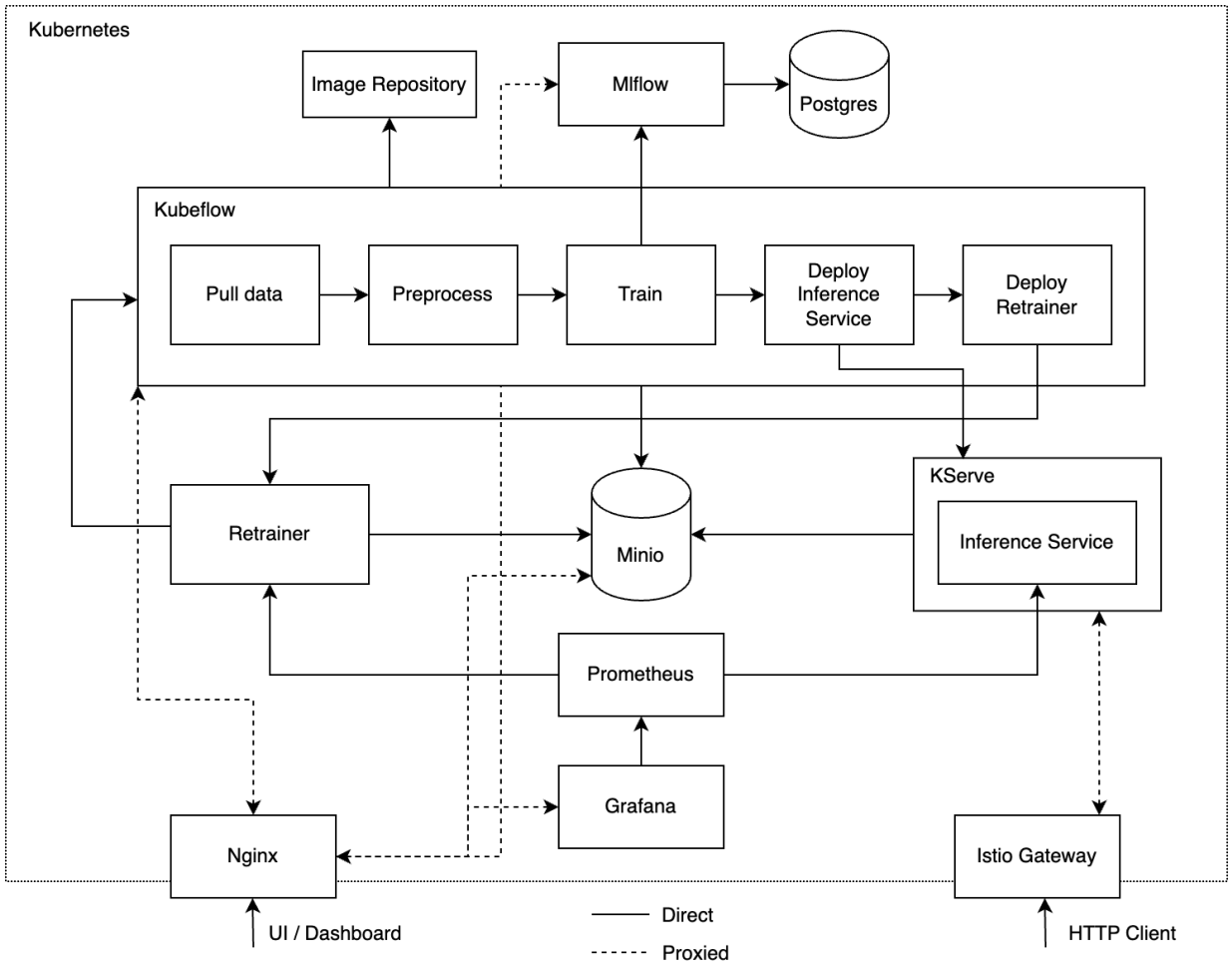


Figure 5.3: Solution Architecture

available in object storage in CSV format. The implementation does not include integration for publishing data collected from sources such as sensor systems of a property to the object storage.

The implementation differs from the design primarily by combining the functionality of the retraining trigger and the monitoring service into the Retrainer component. It also incorporates Prometheus for collecting metrics and visualizing them through the Grafana dashboard, enabling performance inspection of the model. Additionally, the evaluation step of the pipeline, as depicted in Figure 5.1, was left out of scope. The retraining process utilizes the same model configuration and hyperparameters but incorporates more recent data, which we found to not contain any significant outliers or erroneous values.

5.4 Training and Deployment Pipeline

The model training and deployment pipeline is implemented by utilizing the Kubeflow Pipelines (KFP) component of the OSS Platform described in Section 5.2.2. KFP composes multiple pipeline components together in a directed acyclic graph (DAG) and executes each component according to the graph.

KFP provides support for creating, using, and tracking ML artifacts during the execution of the pipeline. For example, an artifact could include a model, dataset, or evaluation metrics. A component may specify a number of artifacts as its input and as its output. This describes the relation between different components and is used to form the DAG that describes the execution order of the pipeline steps.

Each pipeline component is a self-contained program that performs one step in the ML pipeline. Each pipeline component is a Python module that runs in a Docker container and specifies its own dependencies. The Python module is specified as an entry point for the container which enables the container to run as an executable.

The training and deployment pipeline is comprised of six pipeline components: Pull data, Preprocess, Train, Deploy Inference Service, and Deploy Retrainer. The pipeline components are visualized in Figure 5.3.

5.4.1 Data Preparation and Preprocessing

The Pull data component fetches the raw data for training from the MinIO object storage component of the OSS Platform. The component fetches the raw data from a MinIO bucket, copies the data to its local disk, and outputs the data as a pipeline artifact.

The Preprocess component receives the raw training data as an input artifact and transforms the data into a Pandas[1] data frame. The component then renames the data frame column names, parses the proper datatypes for each column of the data frame, for example, dates are converted to Python's `datetime` objects and decimal numbers to floating points, drops duplicate rows, transforms zeros and negative values of energy consumption to a small positive minimum value ($1 \cdot 10^{-5}$) to prevent the model from forecasting negative energy consumption.

Finally, the Preprocess component splits the data frame into three data sets: train, val-

[1] https://pandas.pydata.org/

idation, and test. The train data set includes most of the data, the validation data set includes data for two forecast periods, and the test includes data for one forecast period. The following time series illustrates our selection of the data sets:

$$\overbrace{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10} \ldots t_k}^{\text{training set}} \overbrace{t_{k+1} \ldots t_{k+2h}}^{\text{validation set}} \overbrace{t_{k+2h+1} \ldots t_{k+2h+h}}^{\text{test set}}$$

We denote t_i as a time series entry at time step i, h as the forecast horizon, and k as the last time step included in the training set. The forecast horizon is configurable; we set it to one day ($h = 24$), which comprises 24 hourly time series entries. In our case, the test set includes the 24 most recent entries, the validation set includes the 48 entries preceding the test set, and the training set includes all remaining entries. These datasets are produced as output artifacts of the Preprocess component.

5.4.2 Training

The Train component receives the train, validation, and test data sets as input artifacts and starts an MLflow tracking run for the training process. A tracking run enables recording various metadata such as metrics and parameters of the model and storing the trained model in the MLflow model registry.

During the training process, the NeuralProphet model object is first instantiated and configured. The configuration and hyperparameters for the NeuralProphet model are described in Listing 5.1.

```
model = NeuralProphet(
    n_forecasts=N_FORECASTS,
    n_lags=N_LAGS,
    impute_missing=True,
    yearly_seasonality=True,
    weekly_seasonality=True,
    collect_metrics={
        'MAPE': mape.MeanAbsolutePercentageError(),
        'MAE': mae.MeanAbsoluteError()
    }
)
model.add_country_holidays(country_name='Finland')
model.add_future_regressor('Temp_outside')
```

Listing 5.1: NeuralProphet configuration

The parameter `n_forecasts` specifies the forecast horizon, `n_lags` specifies the number of past observations to use for the auto-regression and lagged regression modules, `weekly_seasonality` and `yearly_seasonality` enable seasonality and configure NeuralProphet to model multiple seasonalities with weekly and yearly periodicities, `impute_missing` imputes missing values by using a rolling average approach, and `collect_metrics` specifies metrics to compute during the training process.

The `add_country_holidays` method configures holiday events for the model to indicate whether a holiday occurs on a given day. The `add_future_regressor` method enables the future regressor module of NeuralProphet and utilizes a temperature forecast.

Once the model object is instantiated, the training and validation data are used to train the model. The model uses the validation data set to evaluate the performance during training. NeuralProphet provides a convenient `fit` method for fitting the model to the provided data. After the training is complete, the training metrics are logged in MLflow and the model is evaluated.

The model is used to forecast the next forecast period and the result is compared against the test data set. We used Mean Absolute Error (MAE) and Mean Absolute Percentage Error (MAPE) for assessing the model. These metrics are also used by the Retrainer and are discussed more in Section 5.6. The resulting MAE and MAPE metrics are then logged in MLflow along with other model artifacts. These include a plot of the model's performance on the test set and a plot of the NeuralProphet model components. Finally, the Train component saves the model in the MLflow model registry and outputs the MLflow tracking run ID and a model URI pointing to the serialized model.

5.4.3 Deployment

The deployment includes two components: Deploy Inference Service and Deploy Retrainer. Both are executed only after the Train component is executed successfully.

The Deploy Inference Service component receives the model URI and a container image URL as its inputs. The container image URL points to the latest shared container images in the Docker registry. The component specifies the KServe inference service to be deployed on the Kubernetes cluster. It uses the KServe client library and Kubernetes client library to define an inference service CRD and the model server container within it. The KServe Controller is responsible for creating and updating the service, deployment, pods, and other Kubernetes resources required for running the inference service. It also provides

functionality for fetching the model to a file system mount used by the model server container. The component uses the provided model URI in the inference service definition to automatically fetch the model and the image URL to define the server container image.

After the inference service is defined the component uses the KServie client library to either patch an existing inference service or create a new one if one does not exist. Patching is handled using a Kubernetes deployment that provides zero-downtime deployments. This is handled using a rolling update approach where Kubernetes waits for the new pod to be ready before removing the previous one from the Cluster.

The Deploy Retrainer component similarly receives the container image URL and a pipeline version of the currently running Kubeflow pipeline as its inputs. The component defines a pod and a deployment resource for the pod using the Kubernetes client library. The container definition for the pod uses the given container image and passes the pipeline version as an argument for the Retrainer Python module which is defined as the entry point for the container. Finally, it either patches an existing Retrainer deployment with a new one using a rolling update or creates a new deployment if one does not exist.

5.5 Inference Service

The Inference Service is a service that provides an API for requesting the forecast from the model through HTTP. It enables easy integration for multiple clients to fetch the current forecast through the API. The inference service is built using KServe and follows its protocol for defining the API endpoints. It provides endpoints for listing models, getting model information, and requesting predictions from the models.

KServe utilizes Knative Serving[1] to provide *serverless* deployments of the inference service. The serverless computing paradigm (Shafiei et al., 2022) hides the execution environment i.e. the computation node, virtual machine, or container from the customer, and includes automated scaling by demand.

After the deploy step in the pipeline creates an inference service CRD, which includes the container image of the Model Server, the KServe Controller reconciles the desired state of the service to be reflected in the cluster. It manages the creation of the Knative serverless deployment that enables autoscaling based on the incoming request workload including

[1] https://knative.dev/

scaling down to zero when no traffic is received. Under the hood, Knative utilizes Istio[1] for network traffic management and routes external traffic entering the KServe environment through the Ingress Gateway. Figure 5.4 gives an overview of KServe and its related Kubernetes resources.

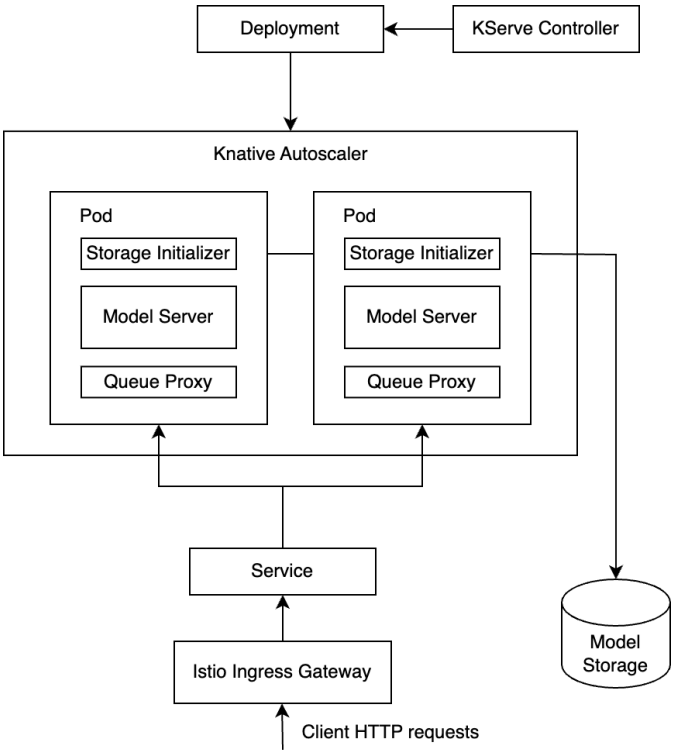


Figure 5.4: KServe Overview

The deployed pod includes a Model Server container and two sidecar containers. The Storage Initializer container is responsible for fetching the model from model storage and saving it in a file system mount used by the Model Server. The Queue Proxy sidecar container collects metrics and measures concurrent requests for autoscaling.

The Model Server container includes the actual Python module for running the server and producing forecasts. We used the `ModelServer` module from KServe Python SDK to build custom serving functionality. KServe includes out-of-the-box support for serving models built with popular frameworks such as PyTorch. However, our use case required more control over serving the model.

Once the Python model server process starts, the serialized model is loaded from the file system and deserialized back into the `NeuralProphet` class instance in memory. A MinIO client is also initialized for storing and retrieving data from the object storage.

[1] https://istio.io/

After the Inference Service receives a request for the energy demand forecast of a property, it routes the request to the model server. Once the model server receives the request it does the following steps.

1. Fetch the previous time series entries from the ground truth data set according to the number of lags configured for NeuralProphet.
2. Fetch the outdoor temperature forecast data for the next day.
3. Preprocess previous time series entries and temperature forecast data into data frames suitable for NeuralProphet
4. Use the trained model to forecast the energy demand for the next day by utilizing the previous time series entries and the temperate forecast data frames.
5. Store the produced forecast in the predictions data set in the object storage.
6. Respond to the HTTP request with the forecast data converted to JSON

We assume that the temperature forecast for the next day is available in the object storage. We did not include any integration with an actual weather data provider. For our demonstration purposes, we left it out of scope. We also assume that the ground truth data for the energy demand of a property will eventually be available in object storage. The Inference Service uses the last ground truth entry to determine the start time of the forecast.

5.6 Retrainer

The Retrainer component enables the CT aspect for the ML pipeline. It includes the trigger that launches the retraining and deployment of a new model. The Retrainer consists of a simple pod with one container that executes a Python module and a deployment that manages the pod within Kubernetes. The pod and deployment resources are defined in the Deploy Retrainer step of the Training and Deployment pipeline.

An internal busy loop is defined within the Python module that runs according to a configured time interval. It checks whether the specified error metrics exceed their threshold values. If the threshold values are exceeded the Training and Deployment pipeline is launched.

The error metrics include an absolute error and a relative error. We used Mean Absolute Error (MAE) and Mean Absolute Percentage Error (MAPE).

MAE is a simple and interpretable metric for measuring absolute error. It is defined as the arithmetic average of the absolute difference between predicted and actual values. The formula is defined as:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

where y_i is the actual value and $\hat{y}_i$ is the predicted value. The absolute error is particularly useful when the predicted and actual values are high since the relative error between the values might be small while the absolute error can be high. It is useful for noticing when absolute errors exceed a tolerable threshold when the energy demand is high.

MAPE is an interpretable and common metric for measuring relative error in forecasting. It measures the average absolute percentage error. The formula is defined as:

$$\text{MAPE} = \frac{1}{n}\sum_{t=1}^{n}\left|\frac{y_t - \hat{y}_t}{y_t}\right|$$

where y_t is the actual value and $\hat{y}_t$ is the predicted value at time step t. The relative error is useful when the actual and predicted values are low since the absolute error might be low but the relative error can be high. However, MAPE can not be used with zero or close-to-zero values since it might lead to a division by zero or extremely high values. We overcame this limitation by preprocessing the data and replacing negative and zero values with small positive values.

We use the combination of MAE and MAPE to determine if the retraining should be triggered. If both exceed their configured threshold value, then retraining should be triggered. We also check that the amount of new data — the actual and predicted values — since the last retraining is large enough to benefit from retraining a new model. We determined that the amount of new data should be over one-hundredth of the total data.

If we denote S_1 to equal the number of new time series entries since the last retraining, S_2 to denote the total amount of time series entries, p to denote the forecast period, T_{mae} to denote the threshold value for MAE, and T_{mape} to denote the threshold value for MAPE. Then the retraining will be triggered if all of the following conditions are true:

1. $S_1 > max(S_2 \cdot \frac{1}{100}, p)$
2. $\text{MAE} > T_{mae}$

3. MAPE $> T_{mape}$

Initially, when the total data set size S_2 is small, one-hundredth of the data set size could be less than the forecast period p. We used the max function to ensure that the size of S_1 should be greater than the forecast period p.

On each iteration of the busy loop, the Retrainer does the following steps.

1. Fetch the predictions data set from object storage and convert it to a data frame.
2. Fetch the ground truth data set from object storage and convert it to a data frame.
3. Get the last forecast from the predictions data frame and the corresponding ground truth data from the ground truth data frame.
4. Skip to step 7 if ground truth data is not available for the forecast.
5. Calculate MAE and MAPE using the predicted values and ground truth values
6. If the retraining conditions 1. - 3. defined above are all true, then run the Training and Deployment pipeline using the KFP Python SDK.
7. Sleep for a configured amount of time.

In step 5. the Retrainer calculates error metrics based only on the most recent forecast period. This approach highlights poor performance in individual forecasts without averaging it out across multiple forecasts. In step 6. the pipeline version argument is utilized to determine which Kubeflow pipeline is run. The argument is passed to the Python module in the Deploy Retrainer component of the pipeline.

The Retrainer also includes Prometheus integration and exposes an endpoint that the Prometheus server uses to scrape the metrics for monitoring. The calculated MAE and MAPE values and the number of retrains triggered are recorded in Prometheus metrics. These metrics can then be displayed over time in a Grafana dashboard.

6 Demonstration and Evaluation

6.1 Demonstration Setup

In our demonstration, we simulated a scenario aimed at forecasting the energy consumption of a university property, initially starting with a small number of data samples. We chose this scenario to showcase how the model must adapt to the evolving input data and how the system will trigger retraining.

We have been provided with data for energy consumption from several university properties in Finland as discussed in Chapter 3. The data is structured as a time series with an hourly frequency. It is formatted as a CSV file that includes columns for date, time, energy consumption, and outdoor temperature. The time series includes 25618 entries starting from January 2019 to April 2022. Table 6.1 describes a sample of the entries in the time series dataset that includes all of the relevant columns.

Time	Consumption	Unit	Temp_outside	Holidays	Property ID
2019-01-01 02:00:00	0.16	MWh	0.7	1	1
2019-01-01 03:00:00	0.16	MWh	0.8	1	1
2019-01-01 04:00:00	0.16	MWh	0.8	1	1

Table 6.1: Example of provided time series data.

In the scenario, the forecast period is one day (24 entries) and describes the property's energy consumption for the next day. The dataset included entries for nine properties of which we chose one randomly for the demonstration.

The demonstration utilized an initial dataset consisting of the first 10% of entries from the total dataset. The entries of the initial dataset started from timestamp `2019-01-01 02:00:00` to `2019-04-29 15:00:00`.

We used a Python script to simulate the scenario that utilizes the provided dataset. On each iteration of the script, a temperature forecast is fetched and stored in object storage, then the energy consumption forecast is requested from the inference service, and finally, the ground truth for the past day is published to object storage. The inference service utilizes the temperature forecast to forecast the energy consumption for the next period.

The script uses the actual temperature forecast values from the dataset without introducing any additional noise. Given the high accuracy of 24-hour temperature forecasts in Finland, we concluded that using the actual values is suitable for the demonstration.

We also configured the Retrainer component's scheduled interval to run at least once between every iteration of the simulation script to check if the model needs to be retrained after each day. The Retrainer calculates the error between the ground truth and the most recent forecast. If the absolute and relative error metrics described in 5.6 exceed the chosen threshold values, the Retrainer will trigger the pipeline to train and deploy a new model.

The threshold values for this demonstration were chosen based on the dataset. We examined the property's mean (0.14 MWh), maximum (0.65 MWh), and minimum (0 MWh) energy consumption, and determined that a threshold of 0.05 MWh for absolute error (MAE) and 20% relative error (MAPE) would be reasonable for the demonstration.

The main loop for the script is described in Listing 6.1. We use the variable `curr_date` to denote the current day in the scenario. On lines 6-13, we first check for an active retraining pipeline and wait until it has been completed. On lines 14-15, we update the current day temperature forecast and request the energy consumption forecast for the current day. Finally, on lines 17-19, we update the ground truth values for the current day and wait for a few seconds so that the Retrainer can run before the next iteration.

```
1  def run_experiment():
2      experiment_str = get_experient_id(kfp_client)
3      curr_date = FIRST_DATE
4      while curr_date < LAST_DATE:
5          logger.info(f'DAY {curr_date.isoformat()}')
6          active_run = get_active_run(experiment_str)
7          if active_run is not None:
8              logger.info('Waiting for retraining to complete...')
9              kfp_client.wait_for_run_completion(
10                 run_id=str(active_run.run_id),
11                 timeout=60 * 15,
12                 sleep_duration=10
13             )
14         update_temp_forecast(curr_date)
15         forecast = get_forecast(curr_date)
16         logger.info(f'Forecast\n{forecast.to_string()}')
17         update_ground_truth(curr_date)
18         curr_date = curr_date + timedelta(days=1)
```

```
19          time.sleep(10)
```

Listing 6.1: Scenario simulation loop

Before running the demonstration script, we started the MLOps platform Kubernetes cluster in a local setup. It initiates all of the services described in the platform in addition to the Retrainer. Once the services were initialized and running, we uploaded the initial data set described above to MinIO object storage and configured the Retrainer to run every 5 seconds.

6.2 Demonstration

After the demonstration setup, we launched the pipeline to train and deploy the model using the initial data set. Kubeflow Pipelines provides an admin UI to inspect the execution of pipelines. Figure 6.1 presents the execution graph of our pipeline in the KFP UI after successful execution. The graph displays each component of the pipeline described in Section 5.4 along with their dependency relationships.

During the training step, the model, evaluation metrics, and related artifacts were stored in MLflow. The artifacts include a figure illustrating the decomposition of the NeuralProphet model components. The figure visualizes all of the components used by the model such as trend and seasonality. After the training phase finished, the trend and yearly seasonality components of the initial model are depicted in Figure 6.3(a).

Once the pipeline was executed successfully and the inference service was ready to receive forecast requests, we started the demonstration script. The script started requesting the next-day energy consumption forecast from the inference service every 10 seconds. After each request, the script published the ground truth data of the forecast period to object storage. The Retrainer was configured to run every 5 seconds and checked whether the model should be retrained between each forecast.

During the execution of the demonstration script, the Retrainer successfully triggered the pipeline to train and deploy new models once the forecast errors exceeded their threshold values. During the demonstration, the error metrics and retraining occurrences could be inspected from the Grafana dashboard. Figure 6.2 shows the Grafana dashboard during a demonstration run. It shows the MAE and MAPE metrics over time and displays a red background for values that exceed the threshold and a green background for values below the threshold.

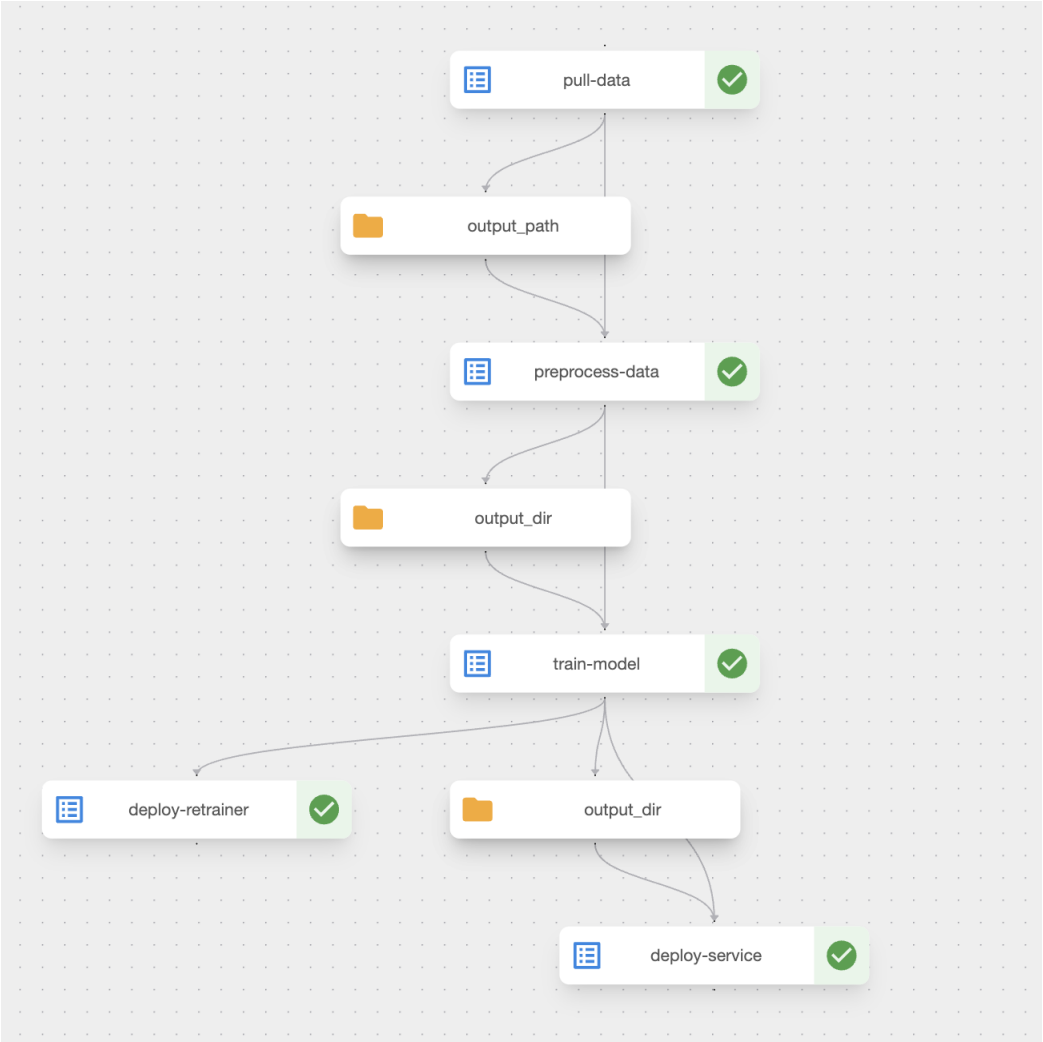


Figure 6.1: Kubeflow pipeline execution graph.

Once the model was retrained using a year's worth of hourly energy consumption data, the retraining frequency decreased. At that point, the retrained model could generate forecasts with acceptable performance over longer periods. The differences between the initially trained model and the model retrained with a full year of data are evident in the decomposition of the model components. Figure 6.3(b) illustrates the trend and yearly seasonality components of the model after it was retrained with over a year's worth of hourly energy consumption data. When comparing it to the components of the initially trained model in Figure 6.3(a), we can see why the initial model was not able to forecast the energy consumption accurately over time as designed in our demonstration scenario.

The demonstration script completed successfully after processing nearly three years of hourly energy consumption data. We collected the forecasts produced during the demon-

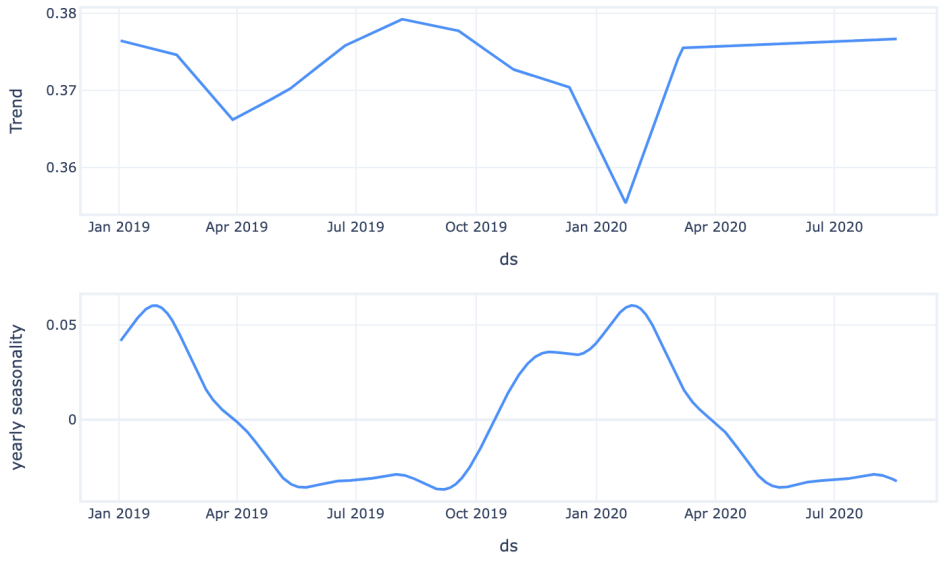
Figure 6.2: Grafana dashboard.

stration and compared them to the actual energy consumption data. Figure 6.4 shows the predicted and actual energy consumption values throughout the demonstration, including all retraining instances. We also calculated the MAE and MAPE for each forecast and plotted them along with their threshold values in Figure 6.5. The hourly time series entries are grouped by day and each year is split into its own subplot for better visualization.

The results show that the model was retrained 17 times during the demonstration. Out of 1067 forecasts, there were 17 instances where both MAE and MAPE values exceeded their respective threshold values of 0.05 and 0.2. As discussed previously, the retraining frequency decreased significantly after the first year. During the first year, the model was retrained 14 times out of the total 17 retraining instances. The results in Figure 6.5 show that after each retraining instance, the MAE fell below the threshold value for subsequent forecasts. However, during the first year, the performance improvements gained from retraining did not last long. The shortest interval between retraining instances was 7 days from the timestamps `2019-11-02` to `2019-11-09`. The longest interval was 338 days from timestamps `2021-01-16` to `2021-12-20`.

Additionally, Figure 6.5 illustrates the dual approach for calculating absolute and relative error, showing high relative error during periods of low energy consumption and lower relative error during periods of high energy consumption, even when the absolute error exceeds its threshold. For instance, between `2021-11` and `2022-02`, the absolute error frequently surpassed its threshold, while the relative error stayed below its threshold.

(a) Trend and yearly seasonality initally.



(b) Trend and yearly seasonality after multiple retrains.

Figure 6.3: NeuralProphet model trend and yearly seasonality components during the demonstration.

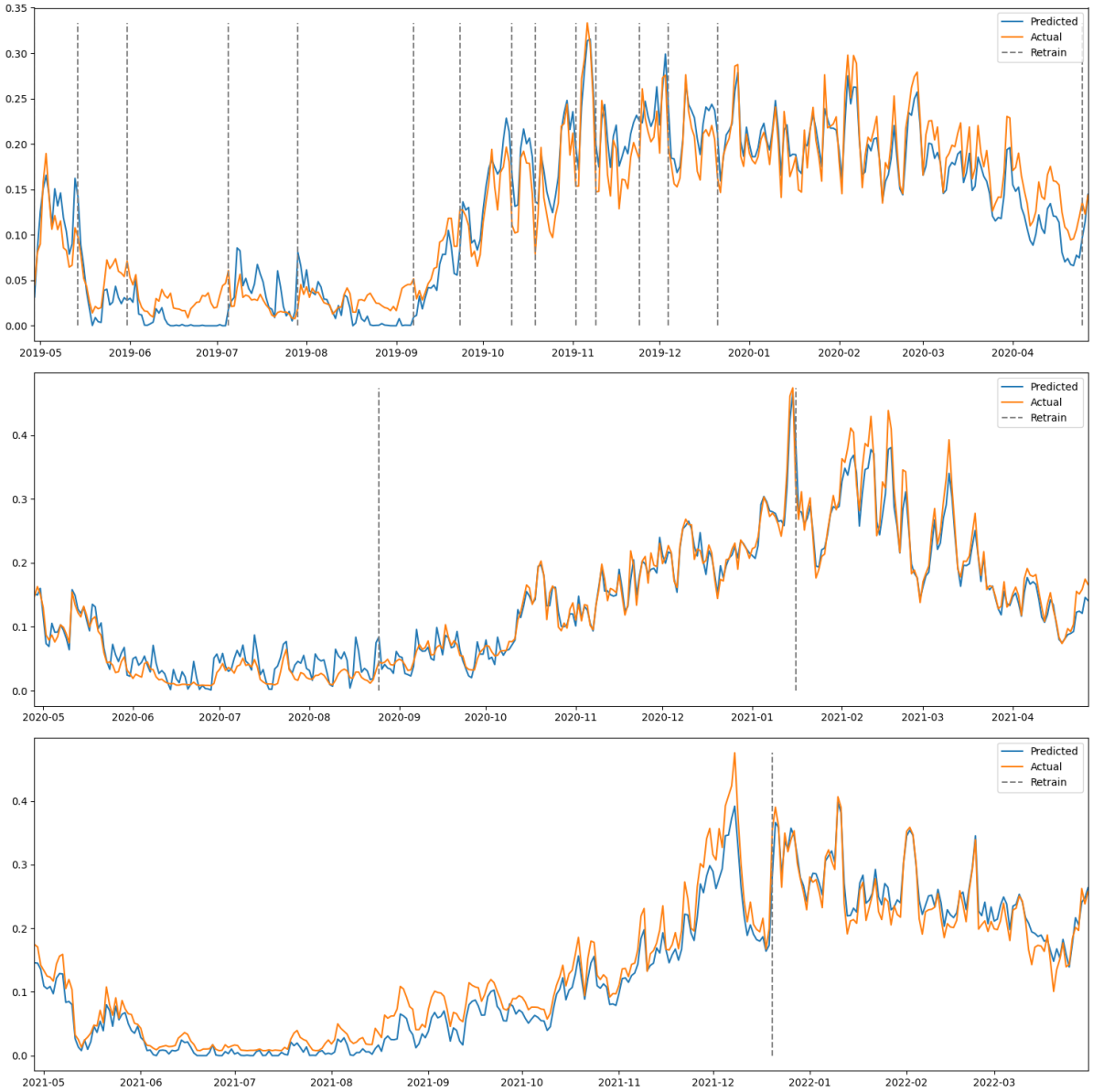


Figure 6.4: Demonstration results.

Figure 6.5: Demonstration result errors.

6.3 Evaluation

In this Section, we present the evaluation of the software artifact against the solution objectives defined in Section 4.2 Table 4.1. We analyzed the design, implementation, and demonstration results to determine the fulfillment of the objectives.

Objective 1. According to the objective, data preparation, model training, retraining, and deployment should be automated. This is reflected in the design through **Functionality 1** which states that the implementation should include an ML pipeline with the necessary steps for training and deploying models. In addition, **Functionality 2** states that the solution should include an error-based retraining trigger, which fulfills the retraining part.

The implementation reflects the objective through the ML pipeline for training and deploying models built on top of the Kubeflow Pipelines platform. The pipeline includes components for data preparation, training, and deployment. When the pipeline is triggered each of the pipeline components is automatically executed and a trained model is deployed. We consider this implementation to fulfill **Functionality 1** in the design.

Automated retraining is implemented in the Retrainer component, which automatically triggers the ML pipeline execution based on the model's performance and availability of new data. As long as prediction and ground truth data are available, the model will be automatically retrained. The new model is trained with the same model configuration and hyperparameters as the predecessor but includes more recent data, which we define as retraining. We consider the Retrainer to fulfill **Functionality 2** in the design.

Since both of the functionalities are fulfilled and the implementation reflects the necessary parts of the objective, we consider **Objective 1** to be fulfilled. However, if the model continues to perform well enough to not exceed the error thresholds, it might theoretically never retrain the model. This may not be a problem, but it can be remedied by configuring stricter threshold values.

Objective 2. According to the objective retraining should be efficient and happen only once the model's performance starts degrading. This is reflected in the design of **Functionality 1** and the implementation of the Retrainer component.

To assess the model's performance degradation, appropriate metrics for measuring performance are necessary. The Retrainer calculates an absolute error using MAE and a relative error using MAPE for each forecast. If the metrics exceed their configured threshold values

for a single forecast, we consider the model's performance to have been degraded. However, this assumes that the configured threshold values for MAE and MAPE reflect a limit of acceptable performance of the model. Determining the limit for acceptable performance of the model requires finding suitable threshold values for MAE and MAPE. Finding suitable threshold values might require careful tuning and domain expertise. We argue that performance degradation can be caught by comparing absolute and relative errors against suitable threshold values. Since the comparison is evaluated for each forecast individually, any single poor forecast can be identified.

Evaluating the efficiency of the Retrainer is challenging because it is relative. The model is retrained only once the absolute and relative error of a forecast exceeds their threshold values. When the model performs well, the need for frequent retraining is reduced, preventing unnecessary retraining that would have occurred with, for example, a scheduled interval-based approach. The results in Figure 6.5 show that during the simulated scenario, the model was retrained 17 times over a period of 1067 days. In contrast, a daily retraining schedule would have necessitated 1067 instances of retraining. Weekly and monthly schedules would have required retraining 152 and 36 times, respectively. Intervals of over one week could not have kept the model's performance below acceptable threshold values due to the frequent retraining instances required during the first year.

The Retrainer uses the full historical data for retraining which could make the training process more computationally costly over time. This property is not unique to the error-based retraining approach and is also present in scheduled interval-based retraining. This could be remedied by limiting the amount of historical data used for training the model.

We argue that with suitable threshold values our approach for retraining is efficient compared to scheduled interval-based retraining, provided that the capacity of the model is sufficient to improve after retraining. However, if the model shows no improvement after retraining, it will continue to undergo retraining whenever the defined amount of new data becomes available. Overcoming the issue might require using different model configurations or another model.

We consider **Objective 2** to be fulfilled. However, we consider the retraining approach to be efficient only in comparison to the scheduled interval-based approach.

Objective 3. According to the objective, the model performance can be continuously inspected for monitoring and troubleshooting purposes. This is reflected in the design of **Functionality 3** which states that the implementation should include a monitoring service for inspecting the performance of the model and for troubleshooting purposes.

The implementation includes Prometheus and Grafana which are provided by the MLOps platform. Prometheus is used for recording metrics in a time series database and Grafana is used for visualizing the metrics in a dashboard user interface. The Retrainer logs the MAE and MAPE values from the most recent forecast into Prometheus during each iteration of the busy loop. We created a custom Grafana dashboard to visualize the recorded error metrics over time. As illustrated in Figure 6.2, the dashboard displays the errors as a time series and uses two different background colors to indicate whether the values are above or below the specified threshold. This allows for visual inspection of the model's performance, assuming that the threshold values indicate the limits of acceptable performance.

For troubleshooting purposes, monitoring the error metrics might not be sufficient. The inference service and retrainer components each provide logs individually which can also be utilized for troubleshooting purposes. Therefore, we consider **Objective 3** to be fulfilled. However, the implementation does not support centralized logging for improved troubleshooting.

Objective 4. According to the objective, the model should be practical, interpretable, and reasonably accurate, and it should not perform worse than the existing model after retraining. This is reflected in our model selection for the implementation and design of **Functionality 4**.

The NeuralProphet model can be used through a Python package that can be conveniently downloaded from the Python Package Index (PyPI). The package exposes the `NeuralProphet` class for providing a user-friendly API for the model. The class accepts multiple parameters for configuring the underlying components of the model. Our configuration is described in Listing 5.1. Additionally, it supports saving and loading the model from disk for serving purposes. NeuralProphet provides a similar API as its predecessor Prophet, which set an industry standard for explainable, scalable, and user-friendly forecasting frameworks (Triebe et al., 2019), which we consider to be practical.

In the context of machine learning, interpretability is defined as *the ability to explain or to present in understandable terms to a human* (Doshi-Velez and Kim, 2017). NeuralProphet provides the ability to decompose the components used for producing the forecast. The components are described in Section 5.2.3. The `NeuralProphet` class includes a `plot_components` method for visualizing each component used for producing a forecast. Figure 6.3 describes a plot produced by the `plot_components` method during the demonstration. NeuralProphet produces interpretable forecast components of similar quality to Prophet on a set of generated time series (Triebe et al., 2019), which we consider to be

interpretable. However, the neural network-based AR module can be configured with a number of hidden layers which can significantly reduce the interpretability of the model.

The demonstration results in Figure 6.5 show that the MAE exceeded its threshold of 0.05 in 17 out of 1067 forecasts. Furthermore, after the model was trained with data samples from throughout the year, the frequency of exceeding the threshold value decreased significantly. The MAPE exceeded its threshold of 0.2 more frequently due to the presence of values close to zero. However, during months with higher energy consumption, the relative error remained below the threshold for a longer duration. We consider the model to be reasonably accurate for the use case.

The training and deployment pipeline lacks a proper evaluation step to verify if the newly trained model performs better than its predecessor. Since the new model is trained with the same NeuralProphet configuration and only adds in newer data samples, we assume its performance is at least equal to or better than the previous model. The results in Figure 6.5 show that the error decreases with each retraining of the model, indicating an improvement in the model's performance. However, if the newer data samples include invalid data or outliers the performance of the retrained model might be worse. The preprocessing step of the pipeline addresses zero and negative values and NeuralProphet imputes missing values using a rolling average approach. Implementing a proper data validation step may also be necessary to ensure that the retraining process does not include invalid data.

We consider **Objective 4** to be partially fulfilled. The absence of proper model evaluation and data validation steps prevents us from guaranteeing that the retrained model performs at least as well as its predecessor. Furthermore, the definition of interpretability is quite elusive and challenging to evaluate.

Objective 5. According to the objective, the implementation should be portable, cloud platform-agnostic, and consist of open-source components. This is reflected in the design of **Functionality 5**, which states that all implementation components should be containerized and managed by a container orchestration service. Additionally, it is reflected in our selection of the OSS MLOps Platform for the implementation.

The MLOps platform is built on top of Kubernetes, with all its components running in containers within Kubernetes. Kubernetes is open-sourced and a popular container orchestration platform. Kubernetes is cloud platform-agnostic and can be deployed in on-premises infrastructure. In addition, many cloud providers support managed Kubernetes services. All of the components of the platform are open-sourced, including the platform itself. Therefore, we consider **Objective 5** to be fulfilled.

7 Discussion

We followed the DSMR process from defining the problem and objectives to designing, developing, demonstrating, and evaluating the solution artifact. In this section, we reflect on our contributions and consider how the results were able to address the research problem. The research problem and context are described in Section 4.1. The research problem is formalized in the question: *How to design and implement an ML Pipeline for time series forecasting with efficient continuous training using open-source tools?*

Our contribution is an ML pipeline for training and serving a model to forecast energy consumption and is capable of efficient continuous training. We used the OSS MLOps platform to construct the ML pipeline and created a custom Retrainer component that uses an error-based retraining trigger to enable CT. We demonstrated its capability by forecasting the energy consumption of a university property, and compared the result with real energy consumption data. The pipeline is capable of training and serving a model to produce time series forecasts with reasonable accuracy based on our evaluation and results in Figure 6.4.

The results show that the pipeline successfully performed continuous training based on the forecast errors. It was able to adapt to the changing energy consumption patterns throughout the year, despite being initially trained on data from a short period. The Retrainer component calculated the absolute and relative error of each forecast and triggered retraining if the errors exceeded their threshold values. Based on our evaluation in Section 6.3 we consider the retraining trigger to be efficient compared to a scheduled interval-based retraining trigger.

Our implementation focused on the energy consumption forecasting task, but the approach is not specific to energy consumption forecasting and can be applied to other time series forecasting tasks. NeuralProphet is a general time series model and can be configured and optimized for various use cases. Moreover, the retraining strategy described in Section 5.6 is not limited to energy consumption forecasting and may be used for various time series tasks. However, the approach assumes that the ground truth data for the time series is available within a reasonable delay.

The implementation can be modified to support other time series forecasting tasks by modifying the data preparation and preprocessing phases to be suitable for the data set and

removing the dependency on data frame columns specific to the energy consumption data set. Additionally, the threshold values for errors and NeuralProphet need to be configured for the specific task. Our implementation is closely integrated with NeuralProphet and does not offer a straightforward way to use different models, however, our approach for the retraining trigger does not depend on NeuralProphet.

The implementation could be enhanced to better support other time series forecasting tasks by offering more comprehensive configuration options and using standardized data frame column names for the covariates utilized by NeuralProphet. In addition, the data preparation could be optimized by only using a configurable amount of historical data to enhance training efficiency. Generalizing the implementation for other time series forecasting tasks and optimized data processing were left out of scope.

According to the evaluation results in Section 6.3, our implementation successfully fulfilled the objectives we defined based on the research problem, however some with partial fulfillment. We identified some limitations and further improvements. The implementation lacks proper data validation and model evaluation components. Invalid data and outliers could cause the retrained model to perform worse than its predecessor. In our case, the dataset included some missing values and negative energy consumption readings, which were effectively handled by NeuralProphet and data preprocessing. However, in other scenarios, stricter data validation and model evaluation might be necessary.

Our demonstration focused on a single property from the energy consumption data set, which included data from nine university properties in total. The energy consumption patterns throughout the year were similar across the properties, but the consumption levels varied. Additionally, the data presented in the demonstration results was collected from a single run of the demonstration script. We executed the demonstration script on the entire dataset three times, with each run resulting in slightly different retraining frequencies due to the randomness introduced by the stochastic gradient descent method used in NeuralProphet. However, the overall trend of frequent retraining at the beginning and less frequent retraining towards the end was consistent.

Although the OSS MLOps platform was suitable for our objectives and design, it is quite complex and requires an understanding of the underlying components that make up the platform. Additional effort is needed to integrate all the components into a cohesive system. For some use cases, the platform's components may not be suitable or unnecessary, potentially adding unwanted complexity for the system maintainer. In our case, the benefits of the serverless layer provided by KServe did not justify the added additional

complexity. High scalability for model serving was not part of our objectives and the auto-scaling features were unnecessary for our purposes.

The platform primarily encapsulates the components within Kubernetes resources and includes the necessary configuration for their integration. This allows for the possibility of swapping components with different ones. However, additional configuration is required, and users need to incorporate the appropriate Kubernetes resources themselves.

The MLOps tooling landscape is abundant, with new tools and platforms emerging every year. Numerous alternative components could replace those provided by the OSS MLOps platform. For example, TorchServe[1] or Seldon Core[2] could be used for serving the model instead of KServe. TorchServe offers a simple way to serve PyTorch models, which NeuralProphet is built on, while Seldon Core is an inference platform similar to KServe that is deployed on Kubernetes. Some tools offer features that encompass the functionality of multiple components within the platform. One such tool is Ray[3], which provides a core distributed computing library for ML workloads. It includes higher-level libraries for data processing, training, and model serving, which utilize the core library. Ray can be deployed in Kubernetes, making it an alternative for handling various stages of the ML lifecycle.

In addition to online learning methods, there are various approaches in the literature designed to enhance the efficiency of the retraining process. The DeltaGrad (Wu et al., 2020) algorithm enables rapid retraining of ML models based on information cached during the training phase. Another method is to utilize sampling techniques to train the model incrementally with fewer data samples, known as incremental learning (Gepperth and Hammer, 2016). Additionally, some approaches utilize both efficient data preparation and sampling techniques for continuously training and deploying models (Derakhshan et al., 2019).

[1] https://pytorch.org/serve/
[2] https://www.seldon.io/solutions/seldon-core
[3] https://docs.ray.io/en/latest/

8 Conclusion

In this thesis, we examined the challenges of implementing continuous training efficiently. Implementing continuous training in practice is highly dependent on the task and available data. Time series forecasting is a common task in applied machine learning and frequent drifts are common in time series data exacerbating the need for the models to adapt. Therefore, we examined efficient continuous training in the context of time series forecasting and utilized real energy consumption data. We designed and developed an ML pipeline capable of efficient continuous training, following the design science research methodology. We used an open-source MLOps platform to construct the ML pipeline and created a custom error-based retraining trigger to continuously train and serve a model to forecast energy consumption. We demonstrated its ability to adapt to the changing energy consumption demands of a university campus property.

Our approach is limited to time series forecasting and assumes the availability of the ground truth values of the time series – the actual energy consumption readings in our case. Moreover, we consider the approach to be efficient only compared to the scheduled interval-based approach. Therefore, future research could focus on exploring methods to better measure the efficiency of continuous training and precisely define efficient continuous training so that it can be measured across various approaches. Additionally, further research is needed in implementing continuous training for tasks that incur a significant delay in the availability of ground truth or require significant effort in acquiring it.

Bibliography

Beyer, B., Jones, C., Petoff, J., and Murphy, N. R. (2016). *Site Reliability Engineering: How Google Runs Production Systems*. URL: http://landing.google.com/sre/book.html.

Breck, E., Cai, S., Nielsen, E., Salib, M., and Sculley, D. (2017). "The ML test score: A rubric for ML production readiness and technical debt reduction". In: *2017 IEEE International Conference on Big Data (IEEE BigData 2017), Boston, MA, USA, December 11-14, 2017*. Ed. by J. Nie, Z. Obradovic, T. Suzumura, R. Ghosh, R. Nambiar, C. Wang, H. Zang, R. Baeza-Yates, X. Hu, J. Kepner, A. Cuzzocrea, J. Tang, and M. Toyoda. IEEE Computer Society, pp. 1123–1132. DOI: 10.1109/BigData.2017.8258038. URL: https://doi.org/10.1109/BigData.2017.8258038.

Breck, E., Polyzotis, N., Roy, S., Whang, S., and Zinkevich, M. (2019). "Data Validation for Machine Learning". In: *Proceedings of the SysML Conference 2019 (SysML 2019), Stanford, CA, USA, March 31 - April 2, 2019*. Ed. by A. Talwalkar, V. Smith, and M. Zaharia. mlsys.org. URL: https://proceedings.mlsys.org/paper_files/paper/2019/hash/928f1160e52192e3e0017fb63ab65391-Abstract.html.

Derakhshan, B., Mahdiraji, A. R., Rabl, T., and Markl, V. (2019). "Continuous Deployment of Machine Learning Pipelines". In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. Ed. by M. Herschel, H. Galhardas, B. Reinwald, I. Fundulaki, C. Binnig, and Z. Kaoudi. OpenProceedings.org, pp. 397–408. DOI: 10.5441/002/EDBT.2019.35. URL: https://doi.org/10.5441/002/edbt.2019.35.

Doshi-Velez, F. and Kim, B. (2017). "A Roadmap for a Rigorous Science of Interpretability". In: *CoRR* abs/1702.08608. arXiv: 1702.08608. URL: http://arxiv.org/abs/1702.08608.

Gama, J., Zliobaite, I., Bifet, A., Pechenizkiy, M., and Bouchachia, A. (2014). "A survey on concept drift adaptation". In: *ACM Comput. Surv.* 46.4, 44:1–44:37. DOI: 10.1145/2523813. URL: https://doi.org/10.1145/2523813.

Gepperth, A. and Hammer, B. (2016). "Incremental learning algorithms and applications". In: *24th European Symposium on Artificial Neural Networks, ESANN 2016, Bruges, Belgium, April 27-29, 2016*. URL: https://www.esann.org/sites/default/files/proceedings/legacy/es2016-19.pdf.

Google Cloud, H. (2020). *MLOps: Continuous Delivery and Automation Pipelines in Machine Learning*.

IML4E (2023). *IML4E-D4.2-Initial MLOps methodology and the architecture of the IML4E framework*. Tech. rep. ITEA4.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2014). *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated. ISBN: 1461471370.

Klaise, J., Looveren, A. V., Cox, C., Vacanti, G., and Coca, A. (2020). "Monitoring and explainability of models in production". In: *CoRR* abs/2007.06299. arXiv: 2007.06299. URL: https://arxiv.org/abs/2007.06299.

Kreuzberger, D., Kühl, N., and Hirschl, S. (2023). "Machine Learning Operations (MLOps): Overview, Definition, and Architecture". In: *IEEE Access* 11, pp. 31866–31879. DOI: 10.1109/ACCESS.2023.3262138. URL: https://doi.org/10.1109/ACCESS.2023.3262138.

Leite, L. A. F., Rocha, C., Kon, F., Milojicic, D. S., and Meirelles, P. (2020). "A Survey of DevOps Concepts and Challenges". In: *ACM Comput. Surv.* 52.6, 127:1–127:35. DOI: 10.1145/3359981. URL: https://doi.org/10.1145/3359981.

Mikkonen, T., Nurminen, J. K., Raatikainen, M., Fronza, I., Mäkitalo, N., and Männistö, T. (2021). "Is Machine Learning Software Just Software: A Maintainability View". In: *Software Quality: Future Perspectives on Software Engineering Quality - 13th International Conference, SWQD 2021, Vienna, Austria, January 19-21, 2021, Proceedings*. Ed. by D. Winkler, S. Biffl, D. Méndez, M. Wimmer, and J. Bergsmann. Vol. 404. Lecture Notes in Business Information Processing. Springer, pp. 94–105. DOI: 10.1007/978-3-030-65854-0_8. URL: https://doi.org/10.1007/978-3-030-65854-0_8.

Oliveira, G. H. F. M., Cavalcante, R. C., Cabral, G. G., Minku, L. L., and Oliveira, A. L. I. (2017). "Time Series Forecasting in the Presence of Concept Drift: A PSO-based Approach". In: *29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA, November 6-8, 2017*. IEEE Computer Society, pp. 239–246. DOI: 10.1109/ICTAI.2017.00046. URL: https://doi.org/10.1109/ICTAI.2017.00046.

Peffers, K., Tuunanen, T., Rothenberger, M. A., and Chatterjee, S. (2008). "A Design Science Research Methodology for Information Systems Research". In: *J. Manag. Inf. Syst.* 24.3, pp. 45–77. URL: http://www.jmis-web.org/articles/765.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J., and Dennison, D. (2015). "Hidden Technical Debt in Machine

Learning Systems". In: *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*. Ed. by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, pp. 2503–2511. URL: https://proceedings.neurips.cc/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html.

Shafiei, H., Khonsari, A., and Mousavi, P. (2022). "Serverless Computing: A Survey of Opportunities, Challenges, and Applications". In: *ACM Comput. Surv.* 54.11s, 239:1–239:32. DOI: 10.1145/3510611. URL: https://doi.org/10.1145/3510611.

Taylor, S. J. and Letham, B. (2017). "Forecasting at Scale". In: *PeerJ Prepr.* 5, e3190. DOI: 10.7287/PEERJ.PREPRINTS.3190V1. URL: https://doi.org/10.7287/peerj.preprints.3190v1.

Triebe, O., Hewamalage, H., Pilyugina, P., Laptev, N., Bergmeir, C., and Rajagopal, R. (2021). "NeuralProphet: Explainable Forecasting at Scale". In: *CoRR* abs/2111.15397. arXiv: 2111.15397. URL: https://arxiv.org/abs/2111.15397.

Triebe, O., Laptev, N., and Rajagopal, R. (2019). "AR-Net: A simple Auto-Regressive Neural Network for time-series". In: *CoRR* abs/1911.12436. arXiv: 1911.12436. URL: http://arxiv.org/abs/1911.12436.

Wu, Y., Dobriban, E., and Davidson, S. B. (2020). "DeltaGrad: Rapid retraining of machine learning models". In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, pp. 10355–10366. URL: http://proceedings.mlr.press/v119/wu20b.html.