RESUMÉ

Formål

Formålet med dette speciale er at implementere og afprøve et system der kan forudsige CPU forbruget og kontrollere mængden af instanser en eller flere applicationer har, der er deployet i Kubernetes således at man kan være på forkant med at skalere mængden af instanser der skal til for at kunne besvare alle forespørgsler til applicationen. Vi har derfor foretaget en gennemgribende gennemgang af eksisterende løsninger og deres fremgangsmåde med at løse problemet, og i den forbindelse identificeret flere problematikker der kunne adresseres og forbedres. Vi har derudover undersøgt forskellige **Machine Learning** biblioteker med henblik på at finde det der både var nemmest at benytte således at vi kunne komme i mål inden for tidsfristen, samt det der var bedst egnet til at lave et sådant system. Vi brugte tiden på vores forspeciale på at lave forsøg der identificerede de modeller der bedst forudsagde CPU forbruget, samt lave en prototype af systemet. Vores løsning består af to komponenter; en **Autoscaler** samt en **Forecaster**. Autoscaleren er hjernen i systemet som ved hjælp af Forecasteren laver skaleringsbeslutninger på baggrund af den forudsigelse af CPU-forbrug Forecasteren laver. Autoscaleren består derudover af en frontend der giver brugeren mulighed for at monitorere forudsigelserne samt rette skalerings-parametre. Forecasteren står for alt hvad der har med **Machine Learning** at gøre, hvilket vil sige at den, ved forespørgsel fra Autoscaleren laver en forudsigelse, og fortsat træning af modellerne på det aktuelle CPU forbrug, for yderligere at forbedre modellernes præcision.

Eksperimentet

For at teste om hvorvidt vores autoskalerings-system er i stand til at forudsige CPU forbruget og skalere mængden af instanser på en fornuftig måde har vi opstillet et forsøg hvor Autoscaleren skal styre mængden af instanser for to forskellige deployments i 24 timer. De to deployments autoscaleren skal styre er den samme applikation bestående af en API med en række endpoints der har forskellig eksekveringstid før svar bliver sendt tilbage for at simulere en rigtig application. Forskellen på de to deployments ligger i mængden af forespørgsler der bliver sendt til dem hver i sær. Det ene deployment får sendt en mængde forespørgsler der svarer til det data modellerne er blevet trænet på, og det andet deployment får sendt en anden mængde forespørgsler, for at teste systemets evne til at tilpasse sig. Derudover foretager vi også kontroleksperimenter som vores system kan sammenlignes med. Vi foretager først en base case test, hvor vi slår alt autoskalering fra, for at se hvordan de to deployments klarer sig uden at mængden af instanser ændres fra en enkelt instans. Derudover foretager vi en ground truth test, hvor vi slår Kubernetes egen reaktive Horizontal Pod Autoscaler til, da dette er en veltestet og meget brugt løsning i industrien. Vores resultater viser at vores autoskalerings system forbedre den gennemsnitlige svar tid med mellem 18.3% og 20%. Derudover sænker den mængden af forespørgsler med svar tider på over et skund med mellem 92% og 94%.

Konklusion

Dette speciale præsenterer en omfattende implementering af et autoskalerings system med fokus på på funktionalitet Kubernetes der benytter sig af machine learning til at forudse CPU forbruget af deployments der kører i et Kubernetes cluster. Vores system demonstrere at ved at forudsige CPU forbruget, at der kan opnåes væsentlige forbedringer af den gennemsnitlige svar tid for applicationer på helt op til 20% imens mængden af forespørgsler der tager længere tid at besvare nedsættes med helt op til 99%. Derudover demonstrere det også evnen til at tilpasse sig andre workloads siden vores system klarer sig væsentligt bedre på også det sinusformede workload. Derudover er det udviklet til at kunne deployes direkte i Kubernetes i et production environment, med et minimum af afhængighed af andre applikationer/services. Der er dog nogle enkelte begrænsninger specielt i form af applikationer med uforudsigelige forbrugsmønstre hvor systemet med stor sandsynlighed ikke ville præstere ligeså godt. Det vil dog i langt de fleste tilfælde have mulighed for at tilpasse sig. I takt med at container baserede systemer bliver flere og flere og systemerne bliver mere og mere komplekse, vil forudsigende systemer som dette blive mere og mere brugbare, for at bibeholde den samme præstation af applikationerne. Dette speciale bigrager med et stort skridt i den rigtige retning for at udløse det fulde potientiale af forudsigende skalerings systemer.

Fremtidig udvikling

I forbindelse med udviklingen af vores autoskalerings-system, har vi identificeret en række ændringer der ville kunne forbedre løsningen yderligere, samt gøre at den ville være væsentligt mere relevant at bruge i industrien. Den første ting der kunne forbedres er at træne en model der er i stand til at udvælge den bedste model ud fra en række fejlmargen-udregninger således at vi kigger på flere ting end blot root mean squared error. Det også være relevant at benytte feature selection fra Optuna frameworket brugt til at optimere hyperparametre, samt at automatisere gen-tuning af modellerne ligesom med gen-træningen. Derudover kunne det være relevant at kigge på at have sæson-trænede modeller der er specialiseret i at forudsige eksempelvis weekender, eller andre højtider hvor forbruget er markant anderledes. Det kunne også være relevant at kigge på at træne modeller der kigger på at skalere vertikalt, altså mængden af resourcer en instans har tilgængeligt, da dette ofte er meget hurtigere end at starte nye instanser op. Her kunne det også være relevant at kigge på strømforbruget af hele Kubernetes clusteret for at optimere på dette samtidig med at QoS stadigvæk er overholdt. Man kunne også se om det kunne være relevant at kigge på at distribuere gen-træningen af modeller således at det skalerer væsentligt bedre end vores løsning. Dette kunne gøres ved

at implementere et kø-system hvor Autoscaleren blot forespørger gentræning af modellerne til en service hvor dette så bliver eksekveret når der er ledigt på GPU'erne. Det vil derudover også være nødvendigt at foretage nogle usability tests af frontenden således at de personer der rent faktisk skulle benytte sig af systemet også syntes at brugergrænsefladen er brugervenlig. Til sidst vil vil det også være relevant at foretage en række yderligere tests af systemet, hvor andre test applikationer og workloads ville blive benyttet således, at systemets soliditet, samt evne til at tilpasse sig enhver applikation ville blive understreget.

Implementing a Predictive Autoscaler in Kubernetes Using ML Time Series Forecasting Models

Jonathan Wisborg Fog, Jens Jacob Torvin Møller, and Thomas Møller Jensen,

***Abstract*—This thesis explores how a predictive autoscaling system for Kubernetes can be implemented using time series forecasting utilizing multiple different machine learning models, continuously trained on incoming data. The default reactive autoscaling solutions often leads to loss of QoS during high peak periods, since deployments begin to scale only when the peak is already apparent. We propose a predictive autoscaling solution able to predict the future load of multiple individual deployments and begin scaling each deployment accordingly before the load occurs. The solution is fairly easy-to-deploy, which only requires few dependencies being present in the cluster. The solution continuously monitors the cluster detecting new deployments for which the autoscaling can be applied. Experimental results demonstrate that the Autoscaling system is able to outperform the Kubernetes HPA on the average response time by between 14% and 20%, while lowering the amount of requests above one second by between 93% and 95%, while only using 3% more power, and between 2% and 5% more pods. The source code we used in this study is available as open-source on GitHub see *Section A* (p. 19).**

***Index Terms*—Kubernetes, autoscaling, machine learning, time series forecasting, cloud computing, microservices, predictive algorithms, resource optimization.**

I. Introduction

CLOUD native and orchestration platforms such as Kubernetes has gained more popularity in recent years [1], due to its ability to streamline the process of deploying multi-container solutions and ease the maintainability of these deployments. Furthermore these orchestration platforms have the ability to automatically scale these deployments, if the load, i.e. traffic, CPU, memory, becomes too large according to a user-defined threshold. The scaling can be applied in multiple ways, that is, scaling the amount of nodes, pods or allocated resources [2]. Scaling the amount of pods for a deployment is known as horizontal pod scaling where scaling the amount of resources is known as vertical scaling [3]. Traditional horizontal pod autoscaling occurs reactively, that is, as soon as the considered metric becomes larger than a given threshold, additional pods automatically start deploying. Most major cloud service providers implement some form of predictive autoscaling using ML models to generate a forecast, consisting of timestamps and their respective amount of a given metric, based on former data, which makes it safe to assume that the field has gained popularity in recent years. This allows orchestration platforms to start creating pods before the load increases, maintaining quality of service (QoS), by preparing the deployment to handle the increased load beforehand, thus eliminating the necessity to wait for the pod to be created. Conversely, when the load is decreasing, it allows the orchestration platforms to scale down deployments, thus eliminating over-provisioning. Futhermore, when demand, and thus also load, fluctuates the necessity to scale down a deployment which needs to be scaled up again immediately can also be eliminated, by utilizing predictive autoscaling.

J. W. Fog, J. J. T. Møller, and T. M. Jensen are with the Department of Computer Science, Aalborg University.

Thesis submitted June 13, 2025.

A. Problem Statement

The challenges addressed in this thesis include:

- How to develop a system utilizing ML to allow for multi-deployment and deployment-specific predictive horizontal pod autoscaling in a Kubernetes cluster.
- Is such a system able to outperform the traditional reactive autoscaling process based on a comparative analysis of response time and other metrics?
- Is the system able to retrain and adapt to different loads, to overcome the obstacle of using a general-purpose model?

B. Significance and Contribution

This research contributes to the field of cloud-native ML scheduling by:

- Exposing and developing an MLOps pipeline with an extensive pre-trained model zoo, data preprocessing pipeline and recurrent model training and deployment.
- Developing a deployment-ready Kubernetes workload which is able to proactively scale deployments using deployment-specific models and forecasts.
- Providing a comparative analysis between the traditional autoscaling process and the developed system.

Paper structure: Section II presents background information on autoscaling in Kubernetes as well as various forecasting methods. Section III presents the design of the autoscaling system as well as how it will be tested and measured. Section IV describes how the different parts described in section III have been implemented. Section V presents the results of the system tests and VI discusses said results and various design and implementation choices. Section VII concludes the work, while VII-A provides various ways of extending the work in the future.

II. Background and Related Work

A. ML Automation

Automated Machine Learning (AutoML), a part of MLOps, is a process where the end-to-end development of machine

learning models is automated [4] [5] . This includes automating the reoccurring aspects of the ML lifecycle typically consisting of, but not limited to [6]:

- Data collection / Data preparation
- Data preprocessing
- Model tuning
- Model training
- Model evaluation
- Model selection
- Model deployment
- Model retraining / retuning

Automating such repetitive tasks can help engineers save time, abstract away some of the complexity which ML tend to exhibit and continuously select the best-performing model. The AutoML/MLOps approach does though come with some limitations such as making the model development more opaque [7].

B. Autoscaling in Kubernetes

Kubernetes (K8s) is an open source platform for managing containerized applications. It manages workloads, i.e. applications, and provides features such as scaling, both vertical and horizontal, load balancing and self-healing, in the event of failures during deployment of containers [8]. A common way of scaling deployments in Kubernetes is using the Horizontal Pod Autoscaler (HPA). Kubernetes HPA automatically starts and stops pods depending on metrics such as CPU, memory or custom metrics. This ensures that should the demand for a deployment increase, the amount of pods increases to match the incoming load. The scaling happens in a control loop which runs intermittently with a custom-defined interval (sync period) and the scaling is based on configured thresholds, resulting in a reactive scaling strategy.

C. Limitations of Reactive Autoscaling

The reactive scaling strategy introduces several limitations including:

- Over-provisioning — When demand for workloads increases the HPA can as a result start up several pods to handle the load, which afterwards may still be running, even though the load has decreased, if for example the sync period is set too large [9].
- Fluctuating loads — When loads fluctuate a lot during a short period of time, and the sync period is not set properly, the HPA eventually scales up and down, even though it could have potentially just kept the same amount of pods running throughout this period [10].
- Cloud provider costs — If the cluster is running in a cloud provider environment, where the billing method is pay-as-you-go, having an unnecessary amount of pods running may grow expensive [9].
- Scaling delay — As mentioned, pod creation may take some time, depending on the size of the containers, which can lead to pods needing to terminate immediately after starting up, if the demand has decreased since, wasting resources unnecessarily in the creation process. Another even greater problem introduced by this delay is that the workload is simply not ready to handle the increasing demand, since the pod creation only happens after the demand increase [10].

D. Time Series Forecasting

Time series forecasting involves predicting future values based on previous values [11]. Time series consist of entries of equally spaced timestamps and discrete values on the form (T, v) for uni-variate time series and $(T, (v_1, ..v_n))$ for multivariate time series. A time series is ordered by time, that is, $T_0 < T_1$.

E. Forecasting Methods

Time series forecasting can be carried out in several ways. According to [12] time series forecasting methods can be divided into *traditional time series forecasting models*, *machine learning models* and *hybrid models*, where traditional forecasting models can be further divided into *linear models* and *non-linear models*. Traditional methods are based on mathematical and statistical models [12]. Linear models are easier to understand and implement, utilizing linear assumptions, but non-linear models sometimes prove better, since they can overcome these linear assumptions, and capture more complex relationships [12].

Machine learning models, which include neural networks, prove more efficient as datasets become larger, non-linear and more complex, due to their self-organizing behavior [12].

Hybrid models is a combination of traditional models and machine learning models, used to address the accidental and complex nature which time series exhibit. The hybrid method essentially combines aspects of multiple methods to produce more accurate predictions [12].

Ensemble models is a method that combines multiple learners or models to improve the predictive performance [13]. [14] considers the ensemble method as a subclass of hybrid models whereas [15] considers ensemble models as a method of combining weak, homogeneous models and hybrid models as a method of combining completely different and heterogeneous models.

Multiple ensembling methods exist including, but not limited to:

- Stacking — Multiple base learners are trained on the same dataset, with different training algorithms, each providing a prediction which are compiled and used as training data for the final model (meta-learner), which could be any model [13].
- Bagging — Multiple base learners are trained using the same training algorithm, but with a modified dataset for each learner (bootstrap resampling) [13]. The predictions are compiled and used as a final prediction.
- Boosting — Trains a learner on an initial dataset, resamples the data by prioritizing the misclassified instances and trains a new learner on the resampled dataset, and resamples again yielding a new dataset produced from the two former learners. This process repeats, lastly combining and weighting all the learners to produce a final prediction [13].

F. Related Work & Existing Solutions

Several research efforts and existing solutions have focused on proactively scaling deployments in cloud environments:

- PredictKube by Dysnix is a solution which uses a single AI model which can observe metrics such as the requests-per-second or CPU load and show the trend for up to six hours. PredictKube utilizes the KEDA framework [16] and can be used to proactively scale deployments. PredictKube is a proprietary service [17].
- KubeFlow by Google is a collection of Kubernetes based components used in different stages of the ML lifecycle. It introduces pipelines (KFP), Kubeflow Trainer for model training, Katib for AutoML and KServe for serving models [18].
- Yuan, H. & Liao, S. propose a Kubernetes Operator predictive autoscaling mechanism, which utilizes two forecasting models, Holt-Winter and GRU, and a data-collecting REST API to obtain metrics to dynamically scale the number of instances [19].
- Naayini, P. propose a Kubernetes-based MLOps architecture which builds upon multiple open-source tools such as Kubeflow, MLflow & ONNX to cover the entire ML lifecycle [20].
- Koskinen, Jan. proposes, in his master thesis, an ML pipeline designed for time series forecasting and continuous training, build upon the OSS MLOps platform [21], an ML platform consisting of components such as Kubernetes, MLflow, Kubeflow, KServe, Prometheus and Grafana. The proposition furthermore scientifically examines the challenges which arise when implementing continuous training [22].
- In our preliminary research [23] we provide a foundation for solving these problem statements, by exploring and comparing different time series forecasting models and implementing a proof-of-concept application which should be able to fetch data from Prometheus, utilize a single embedded forecasting model to predict future timestamps and lastly display the forecast in a Graphical User Interface (GUI).
- In [10] a Proactive Pod Autoscaler (PPA) has been developed which is able to forecast workloads based on custom user-defined metrics. The solution does outperform the default HPA. The paper does mention that their solution lacks automation with regards to model tuning and optimization. It mentions an approach which involves running the solution with multiple models being fed the same data and then automatically selecting the best model.

The proposed solution is developed as a Kubernetes oriented, deployment-ready application, which can be implemented in any Kubernetes cluster, with minimal dependencies, while still covering aspects of the ML life cycle from data collection and preprocessing to training, evaluation, comparative model selection, deployment and retraining. The solution differs by providing the ability to adjust the forecast horizon, e.g. how much time into the future the system predicts, by having an extensive pre-trained model zoo, which can all be deployment-specifically retrained and used to create a forecast, thus always providing the best possible forecast available at the moment for a specific deployment. The proposed solution further builds upon the aforementioned preliminary work in [23], since the models used were tuned and trained as a part of the comparative analysis. Furthermore, we use the modules from the **Autoscaler** component in the preliminary work [23] as a strong foundation for the proposed Autoscaling system. The ML part of our proposed solution also utilizes the *Darts* library. The hyperparameter tuning functionality is strongly based upon the previous hyperparameter tuning functionality used to train and tune the models used in [23]. The objective function is also almost identical. Utilizing this work, we propose a deployment-ready autoscaling solution deployed in a real-world Kubernetes cluster and a comparative test, verifying whether the proposed solution is competitive to the default Kubernetes HPA.

III. Methodology

Section III-A describes the overall system architecture we will use to build our autoscaling system. Section III-B describes how we collect data for training. Section III-C describes how we have designed the ML part of our system. Section III-D describes the preprocessing pipeline used to get the data ready for the training. Section III-E presents the design of the autoscaling part of the system and section III-F describes the database schema design we use for collected and produced data. Section III-G presents the visual design of the frontend as well as the various functions available. Lastly Sections III-H, III-I and III-J describe the cloud setup used for training and tests, as well as the metrics we will use for evaluation respectively.

A. System Architecture

Our predictive autoscaling system for Kubernetes consists of the following components:

- **Forecaster**: Uses the ML models from the Darts library to predict the future CPU utilization for the different deployments, as well as continued training on historical data.
- **Autoscaler**: Uses the forecasts to make scaling decisions based on configurable policies.
- Frontend: A GUI where the forecast can be monitored and edited, as well as different settings.
- Data Collection (Prometheus): A service which is capable of monitoring the CPU utilization of all the pods which can be used for further training.
- Database: A database where various information about the deployments as well as historical data, trained models and forecasts can be stored.

We have chosen this overall architecture since we learned from our previous work that packing everything into the same container would result in performance and reliability difficulties since we are running the Autoscaling system in a cloud environment with limited resources, hence distributing the computation into more containers allows for better performance.

Figure 1 (p. 5) illustrates how our **Autoscaler** interacts with Kubernetes, through the Kubernetes API, the frontend, **Forecaster**, database as well as Prometheus which is used for data collection.

B. Data Collection (Prometheus)

For data collection we chose to use Prometheus since it comes out of the box with the functionality we need which is historical sampling of data and is made to run in Kubernetes which makes it ideal for our scenario. We use Prometheus to collect the CPU utilization for the deployment to continuosly re-train the models. The reason that we only collect the CPU utilization is that the amount of requests is roughly equivalent to that of the CPU utilization [10] and furthermore that some of the models only allow for uni-variate time series.

C. Forecaster

The **Forecaster** consists of an **API module** which is an interface that allows communication with the **Autoscaler**, a **database module** to insert forecasts as well as retrieve models and historical data, and a **ML module** which is in charge of retraining the models and providing forecasts for the **Autoscaler**. Furthermore it consists of a preprocessing pipeline, which all incoming historical data must pass through, to ensure its validity in regards to the training process. Figure *Figure 2a* (p. 6) displays a detailed view of the **Forecaster**'s architecture and how it communicates with the system.

For the ML module we chose to use the Darts library as we did in our preliminary research since it provides a large array of different time series forecasting models.

The ML models used from the Darts library are listed in *Table I* (p. 4):

Models	
ARIMA	BATS
VARIMA	Croston
AutoARIMA	Prophet
StatsForecastAutoARIMA	RNNModel
ExponentialSmoothing	BlockRNNModel
TBATS	N-BEATSModel
StatsForecastAutoETS	N-HiTSModel
Theta	TCNModel
FourTheta	DLinearModel
StatsForecastAutoCES	NLinearModel
StatsForecastAutoTheta	TransformerModel
KalmanForecaster	TiDEModel
TSMixerModel	TFTModel

TABLE I: Models used from the Darts library.

A more detailed description of the models can be seen in our preliminary research [23]. The StatsForecastAutoARIMA and VARIMA models are still not used because of training issues and single dimensional data respectively. Furthermore, TBATS, BATS, RNNModel, KalmanForecaster, N-BEATSModel, and StatsForecastAutoCES are not used due to long training times, which are not concomitant with the requirements of a time-critical system.

The **Trainer module** continuously trains all deployment-specific models at predefined intervals and saves them to the database. When all models have been retrained a forecast is generated for each of the models and compared against the historical data. This essentially makes all the models compete continuously in a *winner-takes-all* manner, similar to ensembling. This ensures that all models are always trained on the most recent data from Prometheus and that the most accurate models forecast is used and displayed in the UI.

D. Data Preprocessing

The **Forecaster** module implements a preprocessing pipeline through which all historical data from the **Prometheus** module must pass. The pipeline is responsible for preparing the data for the **Trainer** module, by removing outliers, assuring that it contains no negative values, removing noise, filling missing values and normalizing the data. These measures are implemented to help the models make more sense of the raw data and increase the training efficiency and forecast accuracy [24]. A flowchart describing the pipeline can be seen in *Figure 3* (p. 6).

A comparison of the raw data versus the transformed data can be seen in *Figure 4* (p. 7). The seasonality of the data appear more distinguished and the seasonal patterns (peaks) are more clearly defined when the data processing has been applied. The plot on the right with the transformed data shows that some outliers still appear in the data. This can be accounted for by changing the threshold, which is a user-defined argument passed to the preprocessing pipeline, used to determine when values should be considered as outliers.

E. Autoscaler

The **Autoscaler** consists of an **API module** such that the frontend can retrieve and/or update forecasts and settings from the database, a **database module** in charge of communication with the database, a **Prometheus module** to to retrieve CPU metrics from the Kubernetes cluster, a **Kubernetes module** to find the deployments running in the cluster as well as changing their pod count, and lastly the **runner module** which is in charge of making scaling decisions based on the forecast generated by the **Forecaster** for each of the deployments in the cluster. *Figure 2b* (p. 6) displays a detailed view of the **Autoscaler**'s architecture and how it communicates with the system.

F. Database

For storing historical data, forecasts, deployment settings and ML models we have chosen to use a PostgreSQL database. The database consists of the tables listed in *Table II* (p. 5).

G. Frontend

Since the **Autoscaler** is developed using the ASP.NET Core framework, it also serves the frontend in our system. The frontend consists of a main page where all the relevant deployments are viewable and there is a clear distinction between which deployments have autoscaling enabled. The frontend also consists of a page for each of the deployments where their individual forecasts as well as deployment-specific

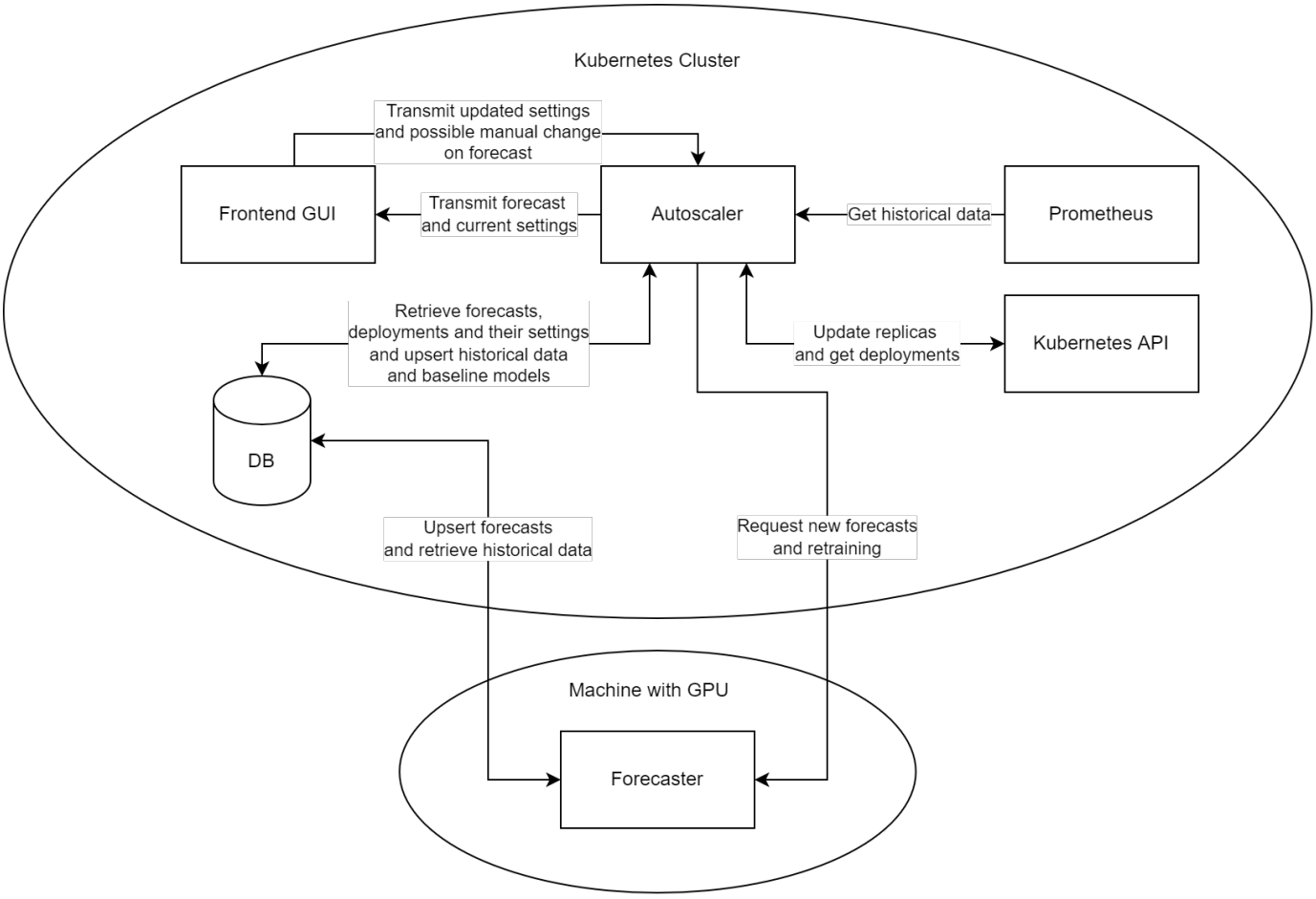


Fig. 1: System interaction overview.

Tables	Description
Model Table (Id, ServiceId, Name, Bin, Ckpt, TrainedAt)	Storing the model binaries for the individual services
Forecast Table (Id, ServiceId, CreatedAt, ModelId, Forecast, HasManualChange)	Storing the forecasts for the individual services
Historical Table (Id, ServiceId, CreatedAt, HistoricData)	Storing the historical data for the individual services
Settings Table (Id, ServiceId, ScaleUp, ScaleDown, MinReplicas, MaxReplicas, ScalePeriod, TrainInterval)	Storing the settings for the individual services
Services Table (Id, Name, AutoscalingEnabled)	Storing information about the service
BaselineModel Table (Id, Name, Bin, Ckpt, TrainedAt)	Storing pre-trained model binaries

TABLE II: Database tables and their attributes.

settings can be viewed and edited. Figure *Figure 5* (p. 7) and *Figure 6* (p. 7) shows the visual design of the two pages which make up the GUI.

H. Training Setup

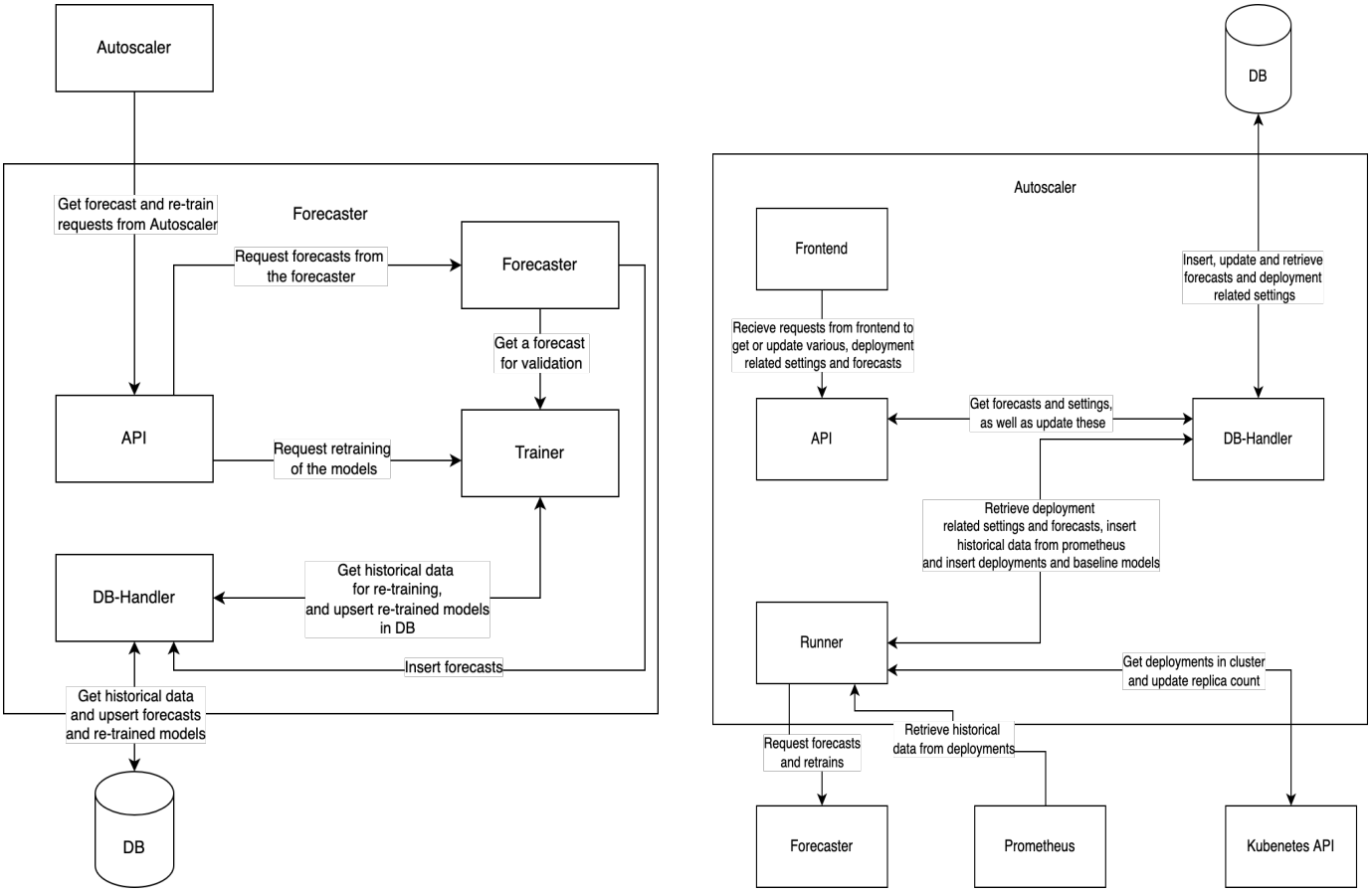
To train the baseline models used in the Autoscaling system we used the following cloud setup. We did not specifically decide on this configuration, but were granted this machine from the university and found that it was sufficient, after experiencing that the default machines did not perform sufficiently without a GPU.

The models were trained on a VM running on the hardware in the AAU Strato Cloud [25]. The hardware specifications can be seen in *Table III* (p. 5). We specifically requested a machine with GPU to leverage these in the model training process, as a measure to optimize and speed up the process.

GPU's: 2 X NVIDIA Corporation GA102GL [A10] (rev a1)
CPU: Intel® Xeon® Gold 6134 Processor 24.75M Cache, 3.20 GHz
RAM: 100GB

TABLE III: Strato Virtual Machine Specifications.

As described in [23] the models used to create the forecasts were trained on a dataset consisting of $\approx$ 5760 logs from a HTTP server. The full dataset consists of $\approx$ 8000 entries split such that $\approx$ 5760 entries are used for training and $\approx$ 2880 entries are used for validation. The data is processed to include only a timestamp and the number of requests at the given timestamp. The interval between two timestamps is one minute. The full training data is visualized in *Figure 7* (p. 8) (before MinMaxScaling has been applied).



(a) Architecture of the **Forecaster**.

(b) Architecture of the **Autoscaler**.

Fig. 2: Architectural diagrams of the main parts of the system.

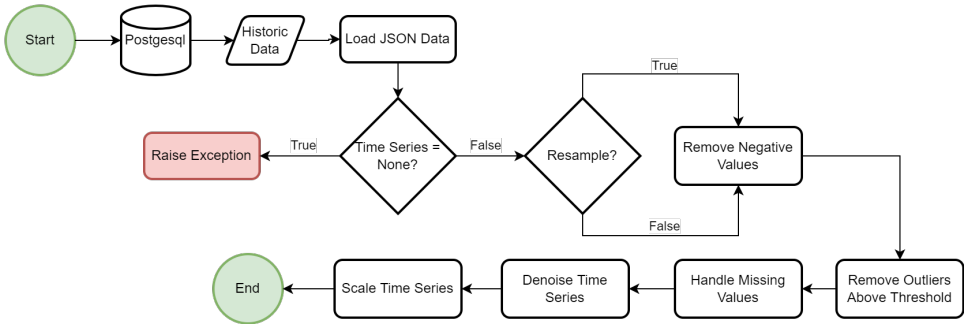


Fig. 3: ML preprocessing pipeline.

I. Experimental Setup

To evaluate our predictive Autoscaling system, we will deploy it to the following cloud setup since this is the set-up we had available at our university.

1) Test Environment:

- A Kubernetes (K3s) cluster running on the AAU Strato cloud [25].
- A node pool with five individual VM's. Specifications are described in *Table IV* (p. 6).
- Prometheus for metrics collection.
- Two deployments of the test application.
- Two deployments of the workload generator application using the Locust.io .framework [26] for real life workload simulation.
- Our Autoscaling system.

CPU: AMD EPYC Processor, Model family 23, model 1
Cores: 4 X 2.00 GHz (Base clock)
RAM: 12GB

TABLE IV: Cluster Nodes VM Specifications.

The reason for the chosen setup is that it minimizes the amount of manual setup as opposed to using a cluster consist-

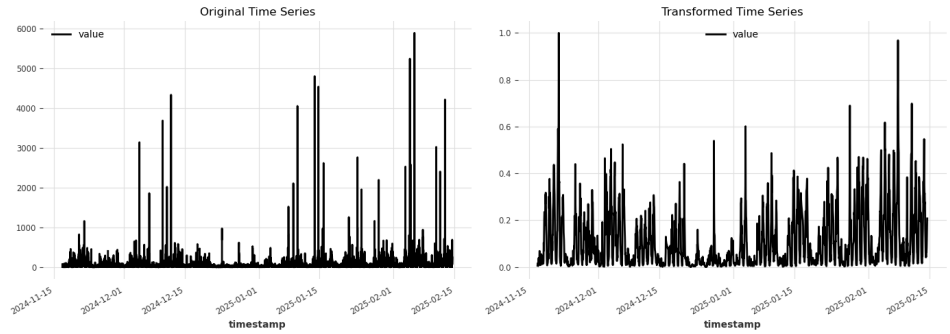


Fig. 4: Raw data and transformed data after being fed through the preprocessing pipeline.

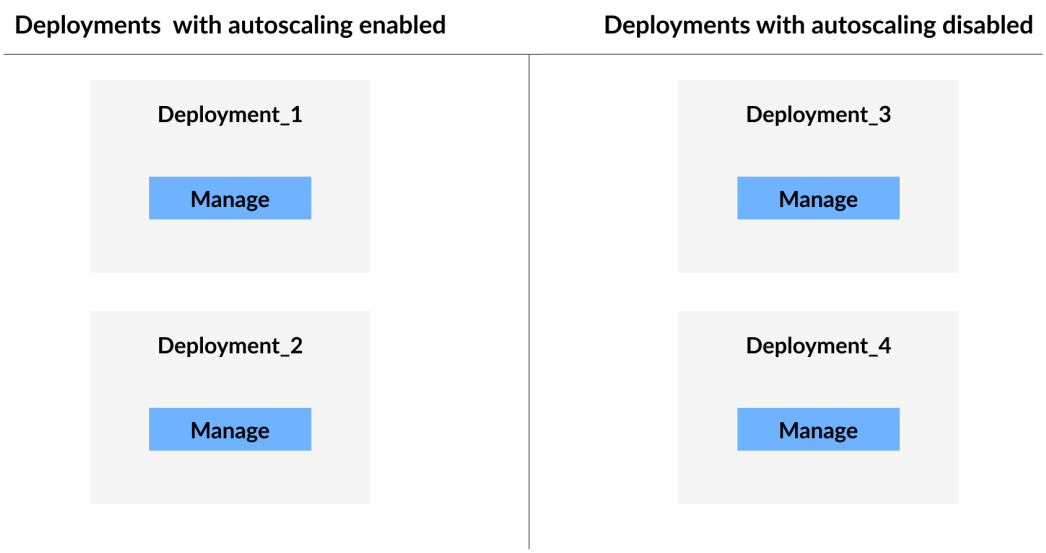


Fig. 5: A page displaying the overall view of all the deployments in the cluster.

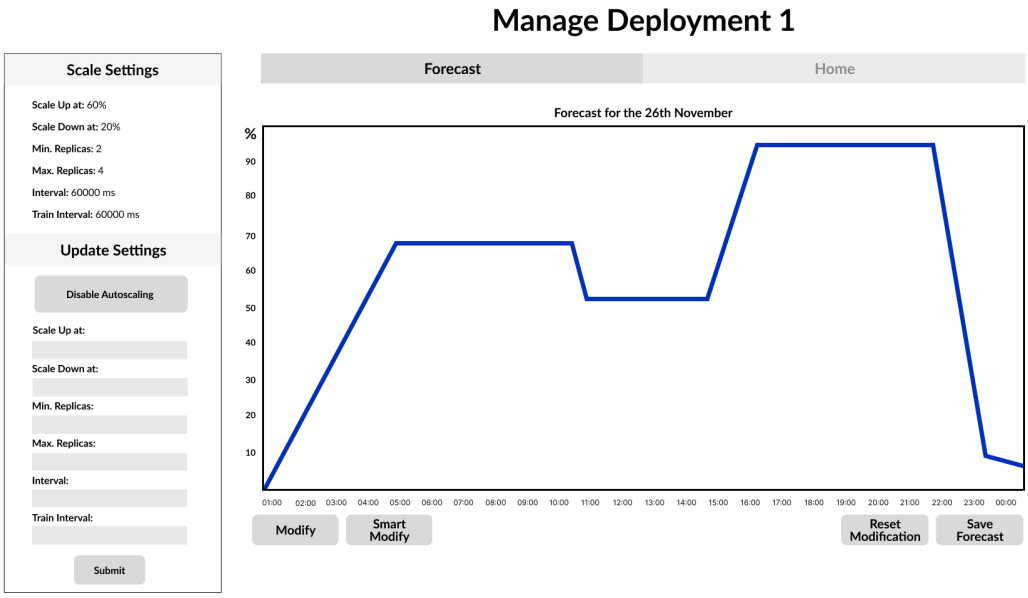


Fig. 6: A page displaying the forecast and settings for a single deployment.

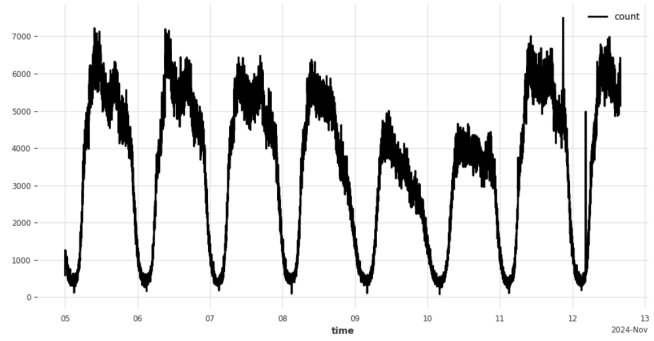


Fig. 7: Data used to train the baseline models. The training data is from time 05 - 09. Validation data is from 11 - 13.

ing of equivalent physical machines.

The forecasting part of the Autoscaling system will be deployed outside the cluster on the same machine as used in *Section III-H* (p. 5) to allow for fast model retraining.

2) Test Applications: We deploy two instances of our test application which will be a basic web API with various endpoints. These endpoints will do different tasks to simulate different user requests which has different durations. The endpoints will include simple get requests as well as post requests where various matrix multiplications will be done to simulate heavy computational tasks.

3) Workload generator application: The workload generator application consists of an application written in Python using the Locust.io framework [26]. The framework is set up to follow a diurnal curve which matches the usual utilization curve for a web application. The reason for using the Locust framework is that it simulates real user behavior, by picking an endpoint based on a percentage specified for each endpoint so that some endpoints are chosen more often than others to simulate a real system. In addition to this it will also wait a random amount of time within a given interval between sending the requests to further add to the real life likeness of the user behavior. We configure the workload generator application with the parameters listed in *Table V* (p. 8) to make the data closely resemble the data which was used to train the models.

VARIABLE	DEFAULT	Description
GENERATOR_MAX	2000	Workload amplitude (i.e. maximum amount of requests)
GENERATOR_MIN	50	Workload baseline (i.e. minimum amount of requests)
GENERATOR_PEAK	16.0	Timestamp of workload peak
GENERATOR_X	100	Size of X in matrix multiplication
GENERATOR_Y	100	Size of Y in matrix multiplication
GENERATOR_MIN_DELAY	1	Minimum delay between requests
GENERATOR_MAX_DELAY	3	Maximum delay between requests
GENERATOR_SHAPE	mapped	Shape of the generated workload

TABLE V: Configuration parameters for the workload generator application.

We will be deploying different workload generator applications for the different test applications, and in addition to this, we will be using different amounts of users to simulate different loads on the services and our Autoscaling systems ability to adapt to the individual deployments. We use the data we trained the models on to tune the Locust framework to produce a similar curve. We will however also tune the Locust framework to produce a sinusoidal curve which is different from the data the models were trained on to test the systems adaptability. The difference lies in the peak load, as well as the gradient between peak and valley.

J. Evaluation Metrics

To evaluate our system we have chosen to use the following metrics:

- Response time
- Pod count
- Node power usage (Watt)
- Forecast RMSE
- Model used for forecast

a) Response time: was chosen as it is the primary motivator behind the project, to optimize the response time compared to the Kubernetes HPA and no autoscaling.

b) Pod count: was chosen as it is another metric that is directly related to the motivation of the project. The goal is to have a pod count that is proportionally similar to the request count from the Workload Generator, while still being able to handle all incoming requests.

c) Node power usage: was chosen as it gives a view of the computational impact of our Autoscaling system compared to the industry standard (HPA). Since we will have the values for both HPA and our solution, a direct comparison of power usage will be useful to determine the efficiency of this system.

d) Forecast RMSE: is collected to measure the deviation of the prediction compared to the validation set. The model is trained on 80% of the input data from Prometheus, and the remaining 20% of the input data is used as validation set. The RMSE is calculated by comparing the validation set to the same amount of minutes of the forecast starting from the beginning.

e) Model used: was chosen so we can test whether the system needs the functionality to supply forecasts from multiple models. If the same model always performs better than all the other models, we have found out that the need to support so many models is not necessary, and selecting one or two models would be sufficient at least for the workloads under consideration.

IV. Implementation

Section IV-A presents the implementation of the **Autoscaler** described in *Section III-E* (p. 4) as well as the database implementations. Section IV-B presents the implementation of the **Forecaster** as described in *Section III-C* (p. 4). In addition to this it describes various modules used by the **Forecaster**. Section IV-C presents the frontend implementation. Sections IV-D and IV-E describes the tools used to test the system and the implementation of the system tests described in *Section III-I* (p. 6).

A. Autoscaler / Backend

The **Autoscaler** is implemented in C# 8.0 and is split into four sub-projects: **Api**, **Persistence**, **DbUp** and **Runner**. The frontend co-exists in the **Api** sub-project, and is hosted by the

ASP.NET core framework. The **Persistence** project is a structured database sub-project, that abstracts database calls using the repository pattern, in the rest of the **Autoscaler**. The **DbUp** sub-project constitutes a database migration tool with SQL scripts for initialization of our database schemas. This project builds on [27]. When run, it checks the **schemaversions** table in the database to check wether the script in question has already been run. If not, it applies the script to the database, if it has, the script is simply skipped and it goes on to next one. This is proper for a production setting since, the scripts will only be executed if they have not been run. We have also prefixed all scripts with a sequence number **xxxx_script.sql** such that they are run in a specific order. This allows us to have the ability to take down the database in case of bad data, or similar, and execute all these scripts in a manner that would give us the same exact database again. Finally the **Runner** project is the brain of the application e.g. where the autoscaling decisions are made. The **API** sub-project serves as the entry-point of the application, where all other sub-projects except for **DbUp** are instantiated, where also the **Frontend** is started. Other than being the entry-point of the application, it also contains the controller, that exposes the necessary endpoints to accommodate the functionality provided by the **Frontend**. When the **Autoscaler** is started and/or when the **start** endpoint provided by the **API** is called, the **Runner** will be created and discover deployments in Kubernetes through the Kubernetes API as shown in *Figure 1* (p. 5). Then it will start a **Monitor** thread for each detected deployment which will monitor the deployment and scale periodically according to the forecast if autoscaling is enabled for the deployment. At the start of the **Monitor** thread it will query Prometheus for the last hour of historical data which will be used by the **Forecaster** when calling both the train endpoint and the predict endpoint. After querying Prometheus and storing the data in the database, the **Runner** checks whether autoscaling is enabled, if it is enabled it then checks if it is the first iteration of the loop. If this condition is met, it retrains the models on the historic data to get correct predictions from the models since they begin the prediction from the last seen timestamp that was available in the training set. It also retrains the models at user specified intervals on all other iterations. The **Runner** then proceeds to check if there is a forecast in the database and if not requests one from the **Forecaster**. Then it finds the forecast horizon by looking at the average pod startup time. We get this by requesting the creation time and the status of the pods from the Kubernetes API where we can calculate the time it takes to start up, from the creation time until the status is **Ready**. The runner then looks that far into the future of the forecast rounded up to nearest minute, and then determines if it should scale the amount of pods up or down. It does so by using the same algorithm used by the Horizontal Pod Autoscaler in Kubernetes [28] while making sure to be within the **min_replicas** and **max_replicas** determined by the user. Lastly it waits for the rest of the forecast horizon before starting the next iteration of the loop. A flowchart of the runner can be seen in: *Figure 8* (p. 10).

The dataset retrieved from Prometheus is of cpu seconds consumed, which are directly proportional to the core count associated with a given pod. We are using **container_cpu_usage_seconds_total** for our calculations [29]. The dataset resembles the combined CPU usage of all the pods of the deployment which gives a general view of the CPU usage regardless of the number of pods. Since we communicate with various pieces of software we have chosen to spend some time creating a development mode for the **Autoscaler**. We have done this so that we do not have to deploy the **Autoscaler** to Kubernetes each time we would like to test a new feature. To do this we have created **Mock** services meaning that we have the following service mocks in the **Autoscaler**:

- MockPrometheusService
- MockKubernetesService
- MockForecasterService

When running the **Autoscaler** in development mode, these services will return generated responses from all the services mocking their behavior, and allowing to test the **Autoscaler**'s logic locally.

B. Forecaster

The **Forecaster** is responsible for abstracting python dependencies, machine learning model training and predicting future timestamps and their values. The **Forecaster** is implemented using Python 3.11, Darts 0.32.0, psycopg2 and Flask. It exposes a REST API with endpoints for training, tuning and prediction. The **Forecaster** expects a set of models to already be available for each deployment, and a single entry of historical data for each deployment. All data must be preprocessed using the preprocessing pipeline described in *Section III-D* (p. 4) to ensure its validity and avoid exceptions during training. The **Forecaster** is implemented to simultaneously train and create forecasts for all models using per-deployment multiprocessing to ensure that the training of one of the deployments models does not bottleneck the training of another deployments models, which could be fatal since we train all models when the **TrainInterval** is exceeded. Since we run the training in parallel we could encounter memory issues, but since the machine we are using has 100GB of RAM we do not consider this to be an issue in our case ($>$5 GB of RAM should be sufficient see: *Section B* (p. 19) but more may be needed depending on the models). When a forecast has been created for each model, the best forecast according to the RMSE, calculated between the predicted values and the historical data as described in *Section III-J* (p. 8), is inserted into the database. When the **TrainInterval** is exceeded each model for each deployment is trained on the historical data, i.e. the last hour, and the aforementioned forecast procedure is executed to update the forecast in the database according to the newly trained models. Furthermore each updated model is inserted in the database as well to allow for further training. The preprocessing pipeline as shown in *Figure 3* (p. 6) is implemented using various data processing measures including **sklearns MinMaxScaler** and **darts fill_missing_values**, a custom outlier detection mechanism and a de-noiser mechanism using the **darts KalmanFilter** model.

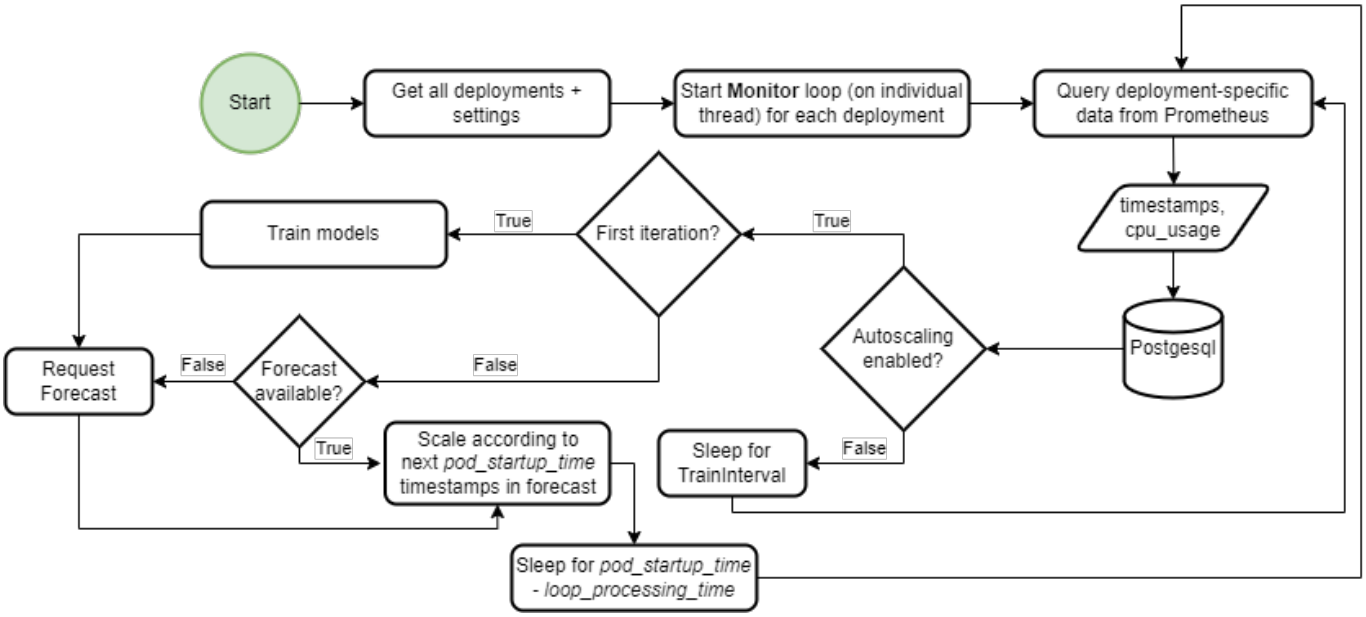


Fig. 8: Runner/Monitor functionality visualized.

C. Frontend

The solution includes frontend GUI from which the user can inspect and make manual changes to the forecast. This could be used if the user knows that abnormal loads will occur in the future, which are impossible for the models to predict, since they may not follow the pattern which the models expect. The frontend is developed using ReactJS and consists of two pages shown in *Figure 9* (p. 11) and *Figure 10* (p. 11); A deployment selection page and a dashboard showing the forecast and a number of settings which can be adjusted to the users preference. These include how much the CPU load should increase or decrease before scaling occurs, the minimum and maximum amount of pods, and how often training and prediction should occur. The deployment selection page displays the active deployments and whether they have autoscaling enabled. Selecting one of these redirects the user to the dashboard page mentioned above.

D. Evaluation Tools

To evaluate the performance of our Autoscaling system in comparison to no autoscaling as well as the HPA from Kubernetes, we have developed a robust platform where we have a test application and a workload generator application as described in *Section III-I* (p. 6) and a system testing program to generate datasets for the evaluation metrics described in *Section III-J* (p. 8).

1) Test application: The test application consists of an API that exposes three endpoints which the generator can send requests to

- Matrix Multiplication (POST)
- Matrix Sum (POST)
- Instant response (GET)

Matrix Multiplication is implemented as a heavy multi-threaded workload, while the matrix sum is implemented as a heavy single threaded workload. Finally the instant response immediately returns a HTTP 200 response. The workload application does nothing on its own, and the only configuration it has, is hosting configuration. The test application will therefore just await incoming requests which means that the only thing that will have an impact on the CPU utilization is the requests and the CPU usage will therefore scale linearly to the request count [10].

2) Generator application: The generator is implemented using the Locust library, with configurable variable load. Since the models were trained on data from a real life system, the generator will follow the curve of the same real life system. Granting the system a workload that is almost identical to the real data. The generator further has the ability to create other types of curves, e.g. a sinusoidal curve, also used in the system tests to test the adaptability of the system. Scaling the curves up by the same scalar it was scaled down with produces very similar curves as shown in *Figure 11* (p. 12). This imply that our workload generation is similar to the load on the system the models were trained on. This is done by looking at each timestep in the trained data and then determining the amount of users at this point in time to make the gradient match the trained data. The amount of users is scaled down since our small Kubernetes cluster wont be able to handle the amount of requests in the original data. If we take a look at *Figure 12* (p. 12) we can see that the relation between the two datasets is rather linear, which means that they are fairly similar. In addition to this, we calculated various metrics to confirm the similarity of the datasets which can be seen in: *Table VI* (p. 12).

E. System Tests

The system tests were implemented as its own project where it manages setup of:

- A base case.

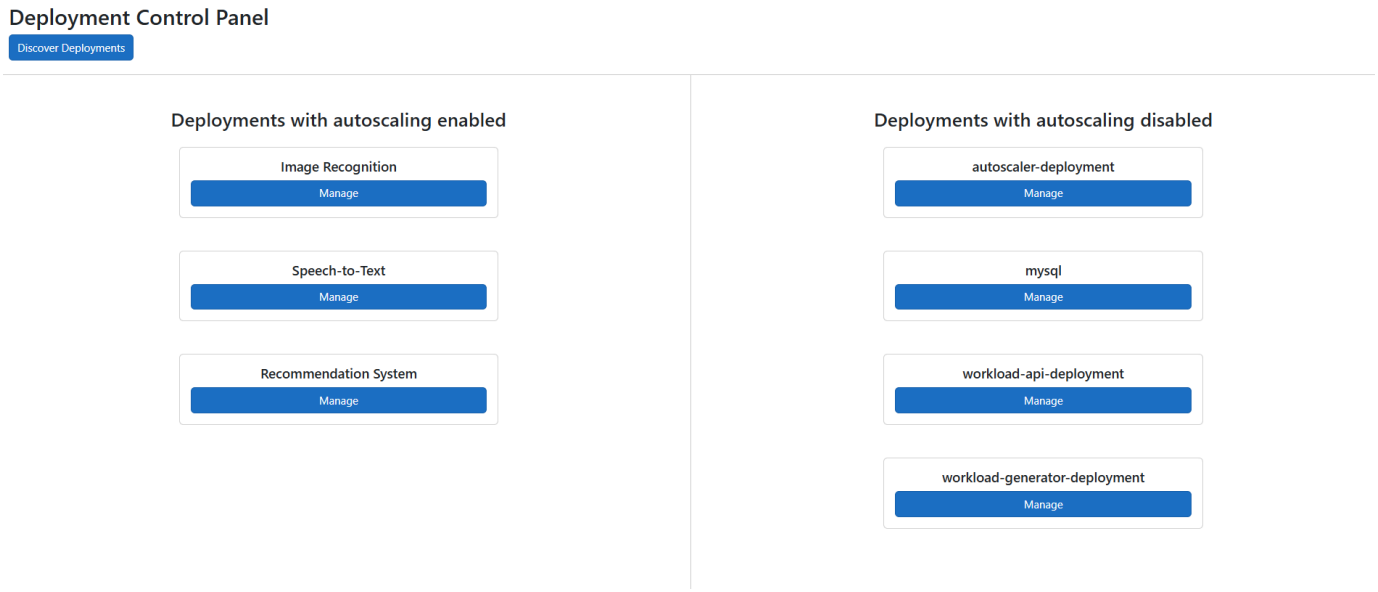


Fig. 9: Dashboard displaying all deployments in the cluster.

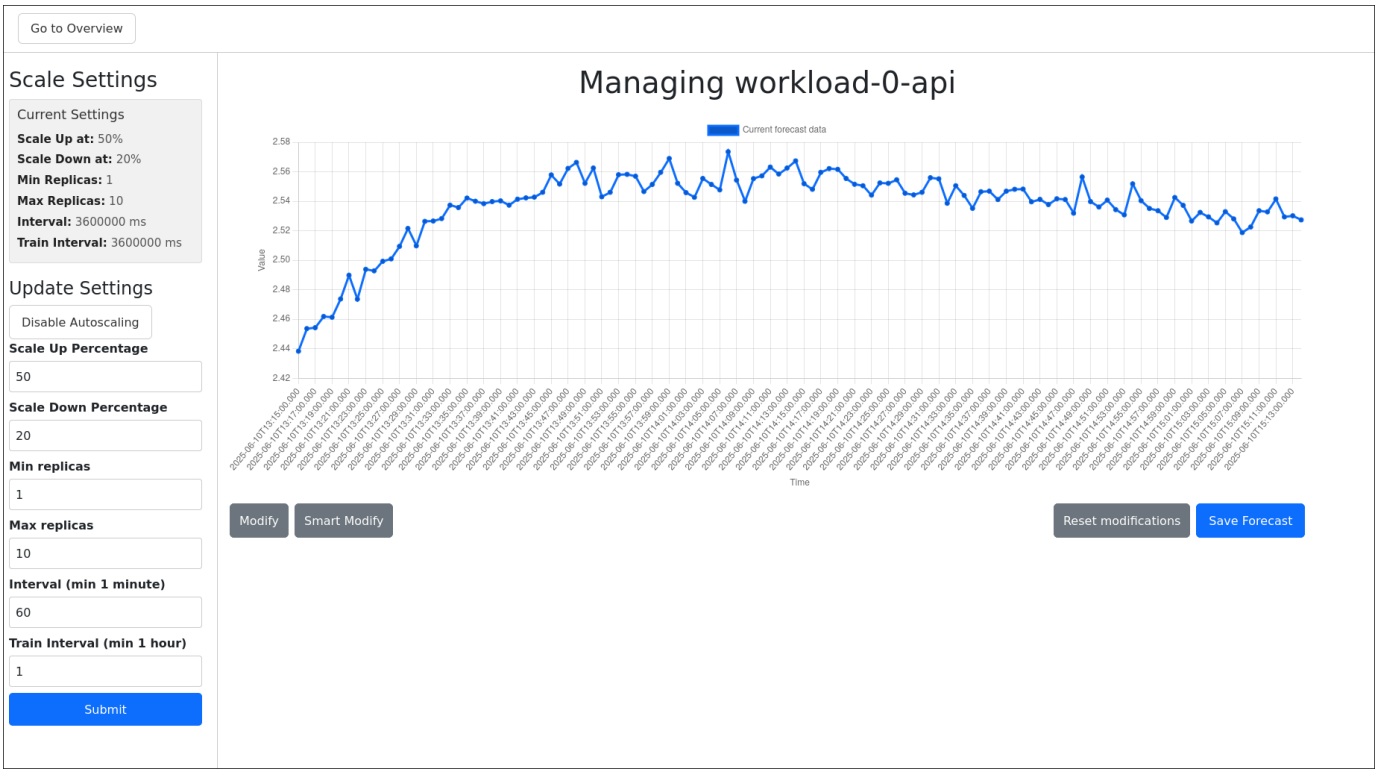


Fig. 10: View of a specific deployment.

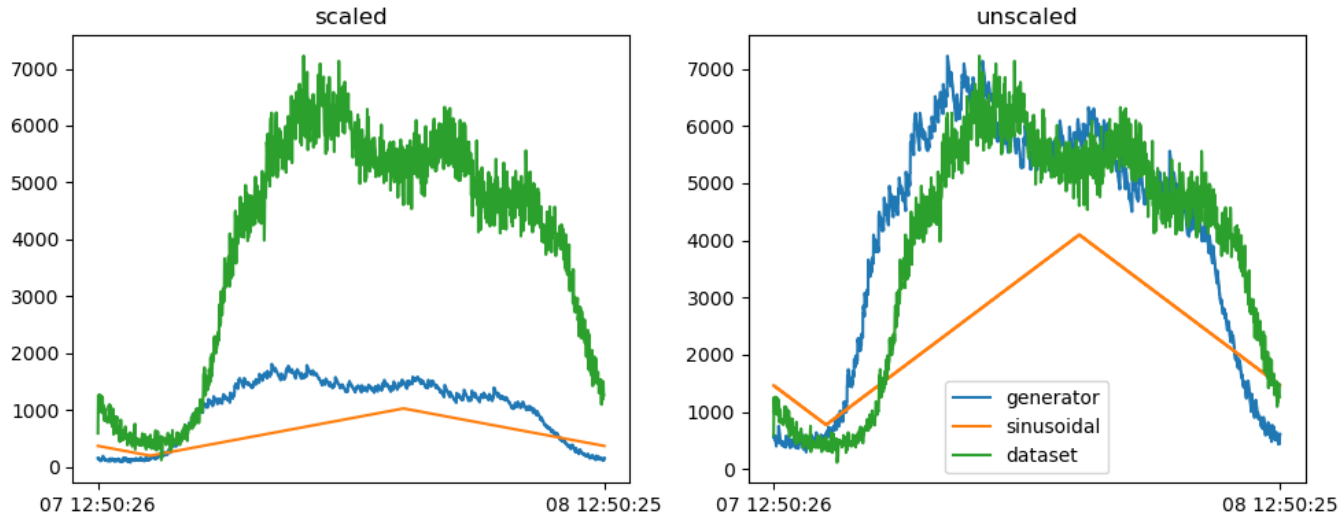


Fig. 11: The request count curve produced by the generator compared to the dataset (green), for both the curve that closely resembles the training set (blue), and the curve that is vastly different from the dataset (orange).

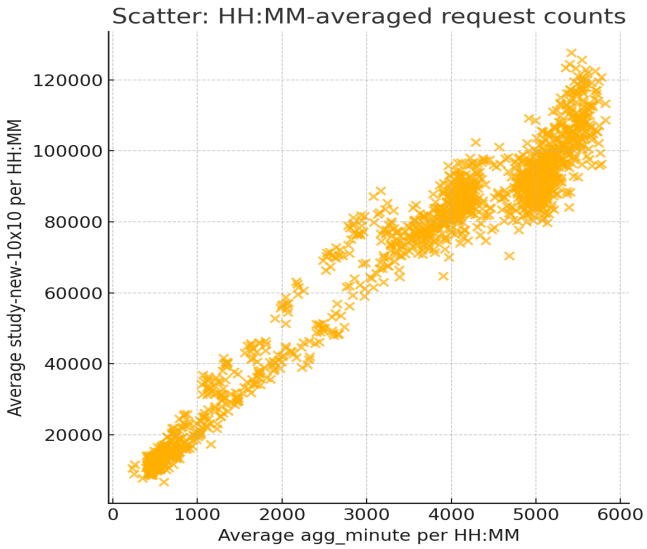


Fig. 12: Scatter plot displaying the similarity of the generated request count and the training data.

Metric	Value
Pearson r	0.9760
R^2	0.9526
Spearman ρ	0.9492

TABLE VI: Similarity metrics for minute-aggregated, time-of-day averaged request counts.

- A Ground Truth (HPA).
- The proposed solution.
- Workload applications.
- Workload generator applications.

The base case is defined as a case where no autoscaling is performed, it should be configured to lead to a worse response time than the rest, showing a general need to autoscale. The Ground Truth is defined as a well tested case that is proven in an industry setting. For this the standard Horizontal Pod Autoscaler in Kubernetes can be considered sufficient, since it is the default in Kubernetes. The system tests will along with the workload generator application, send requests to the workload-api. The system test only sends very few requests to record the response times, by sending similar request to that of the workload generator application. The system test can be configured through various parameters, the parameters are shown in *Table VII* (p. 13).

The purpose of these parameters is to have as similar a configuration as that of the HPA for the predictive **Autoscaler** as possible. The system tests will then generate a set of kube-configs in JSON format with all the adjustments applied. The **workload_configs** are a list of tuples containing the minimum and maximum amount of simulated users for the workload generator, while also defining the type of workload, whether its mapped based on the training data or a generic curve. The system tests also have some parameters unique to the predictive **Autoscaler**, namely the **forecaster_remote_config** and **deployment_settings**. The **forecaster_remote_config** sets the target url for the remote **Forecaster**, while **deployment_settings** updates the settings table with **train_interval** or **scale_period** and more based on a dictionary passed to the tests. The millicores (the unit for amount of CPU used by Kubernetes [30]) used for cpu utilization calculations within Kubernetes, are defined in what Kubernetes calls a **Resource Request** and **Resource Limits**. In cloud environments 1000 millicores correspond to a single vCPU (Virtual CPU) which similarly corresponds to a thread/core on the physical CPU (depending on whether simultaneous multithreading is enabled). This allows cloud service providers to distribute the hardware computation power among recipients [31]. A **Resource Request** is the CPU and memory allocation that a pod requests from Kubernetes. As such it is critical for good data that the **Resource Requests** are well recorded, and omitting them lets Kubernetes allocate dynamically, which can

Name	Description	Default value
size	Size of the workload i.e. matrix size	10x10
period	The testing period	86400s
scale_up	Percentage to scale up	50%
scale_down	Percentage to scale down	20%
min_replicas	Minimum amount of replicas for the deployment	1
max_replicas	Maximum amount of replicas for the deployment	10
workload_configs	A list of configurations for n deployments	[(50,2000,"mapped")]
forecaster_remote_config	Config for optionally running forecaster remotely	None
deployment_settings	Autoscaler database overrides	{}

TABLE VII: System Test Parameters.

severely impact the results of our evaluation. Measuring the millicores used during tests allows us to compare the default HPA's ability to scale with the proposed Autoscaling system, based on how well they minimize this specific metric. Finally this portion of the implementation serves as the primary way the predictive Autoscaling system, HPA, and workloads are deployed to our testing cluster, meaning it acts as our deployment pipeline. The tests are conducted with the parameters shown in *Table VII* (p. 13). The resource requests of the pods are listed in *Section C* (p. 19).

V. Results and Evaluation

A. Dataset

The model tuning was attempted using two similar datasets, one with minutely data and the same dataset aggregated by hour. The models tuned on the minutely data show differing results, some models performing well, while some models did not. The models tuned on hourly data were generally able to capture the patterns better. While tuning each of individual dataset we plotted the forecasts on top of the training data. In *Figure 13* (p. 14) an example of both data aggregations can be seen overlaid by a single weeks forecast. As can be seen in the figure both the model trained on the minutely and hourly data are able to capture the general patterns, but the hourly aggregated data seems to capture the patterns better. Although, proving promising results for multiple models for both datasets, the hourly being the better one, we decided that it would be more optimal to use the models which were trained as a part of [23], since these perform very well on minutely data. If we were to use the models trained on the hourly data, we would leave some of the more fine-grained inference to the models interpolation capabilities, which could perhaps neglect important data points.

B. Evaluation

We attempted to carry out the training on one of the worker nodes described in *Table IV* (p. 6), but experienced that the model training became slow, especially for the PyTorch Models, since the resources (no GPU) of these nodes are not sufficient for these types of models. We shifted the model training to the machine on which the models were initially trained, described in *Table III* (p. 5), which allowed us to utilize cuda and GPUs. This proved efficient and optimized the model training time from around 10 minutes to three minutes for the slowest model (ExponentialSmoothing), while other models train within seconds.

C. Base case

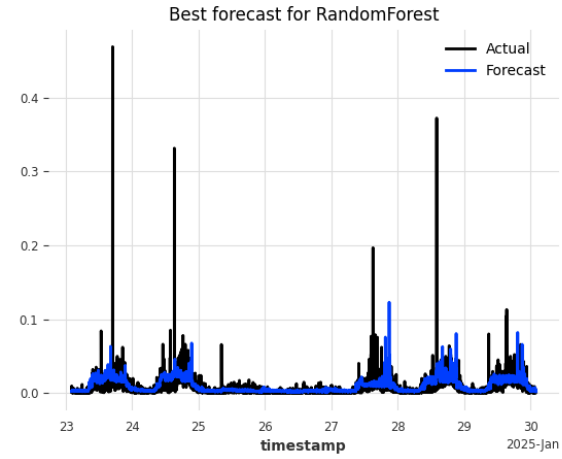
The base case is a test where there is NO autoscaling applied to the cluster, which verifies the need for autoscaling. If we take a look at the results show in *Figure 14* (p. 14) we can see that the response times in the 95th percentile really shows the need for scaling. While the response times in *Figure 14a* (p. 14) mostly fall between zero and one second, we can also see that when the system is under a much higher load, the response times start to increase, where most response times falls between one and five seconds with as slow response times as 14 seconds as shown in *Figure 14b* (p. 14), which means that with a high system load, the need for scaling becomes much more apparent.

D. Comparison of the Predictive Autoscaling system and HPA

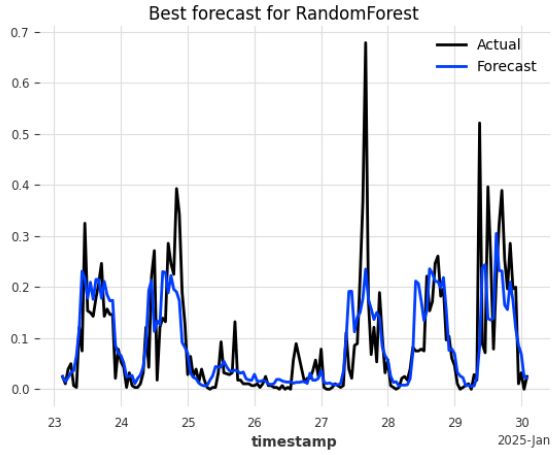
In this section we will be presenting and comparing the results of our Autoscaling system, and the Kubernetes HPA.

1) Response times: If we take a look in regards to the 95th percentile response times of the HPA we can see that it generally lies between almost instant and one second while some requests takes as much as three seconds for the sinusoidal workload while up to four seconds on the real workload trace. If we compare that to our Autoscaling system we can see that it mostly follows the same pattern for the sinusoidal workload as well as the real workload trace. We can however see that the amount of requests that take one second is significantly lower than the HPA and the slowest response time of the test application with our Autoscaling system is also no more than around one second. If we look at *Table VIII* (p. 17) we can see that the test of our system shows that it outperforms the HPA when it comes to average response time in the 95th percentile, with about 20.53% in the workload trace and 13.93% in the workload with a sinusoidal curve. We can also see for the workload trace that the percentage of requests between zero and one second is roughly equivalent, while decreasing the amount of requests above one second by 95.28%. In addition to this, the response time of the application does not go much higher than one second with our Autoscaling system running. If we take a look at the different response time ranges for the sinusoidal workload the amount of requests is like the workload trace roughly equivalent in the zero to one second range, while reducing the amount of request above one second by 93%, and also eliminating response times much longer than one second.

2) Node power usage: The power usage of our Autoscaling system is around 3% higher than that of the Kubernetes HPA,

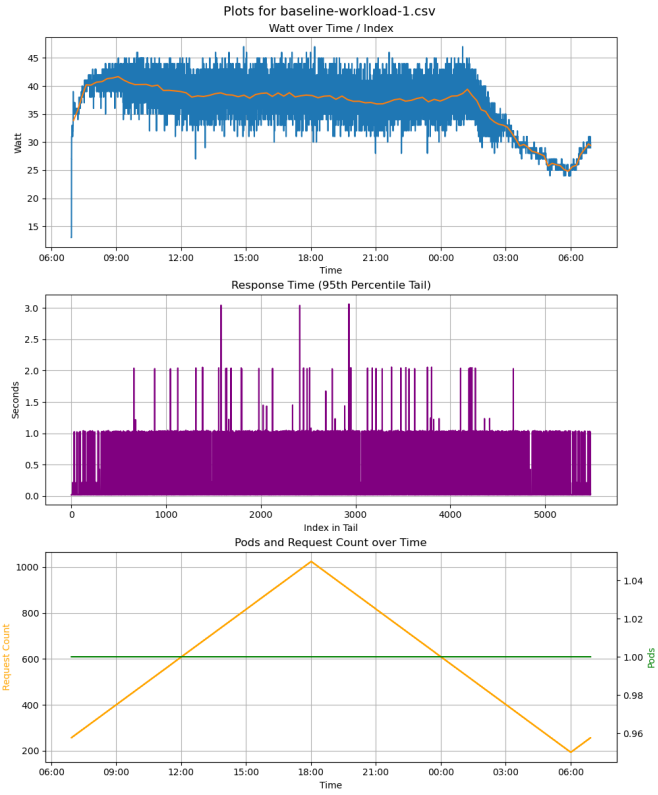


(a) Minutely aggregated data.

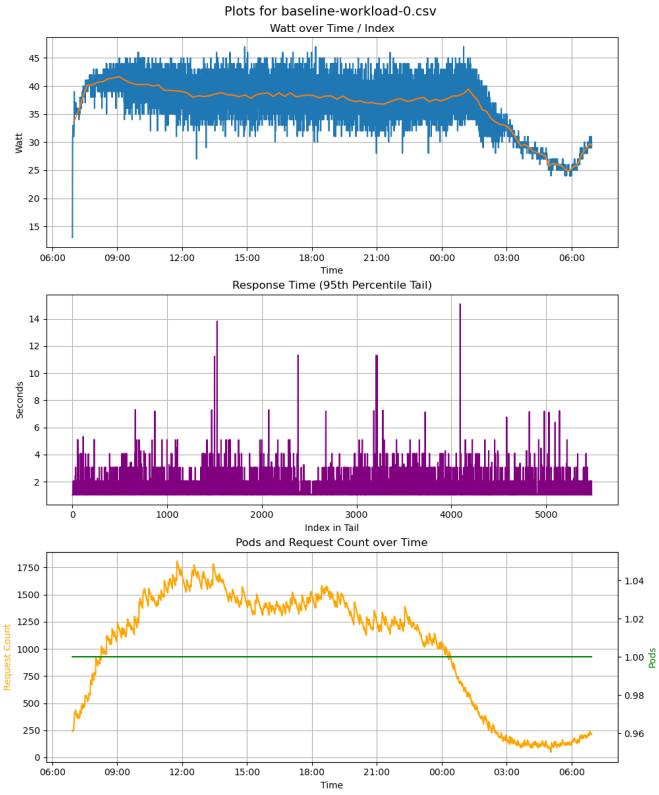


(b) Hourly aggregated data.

Fig. 13: Comparison between forecasts from minutely and hourly aggregated data.



(a) Results from base case test with the sinusoidal workload.

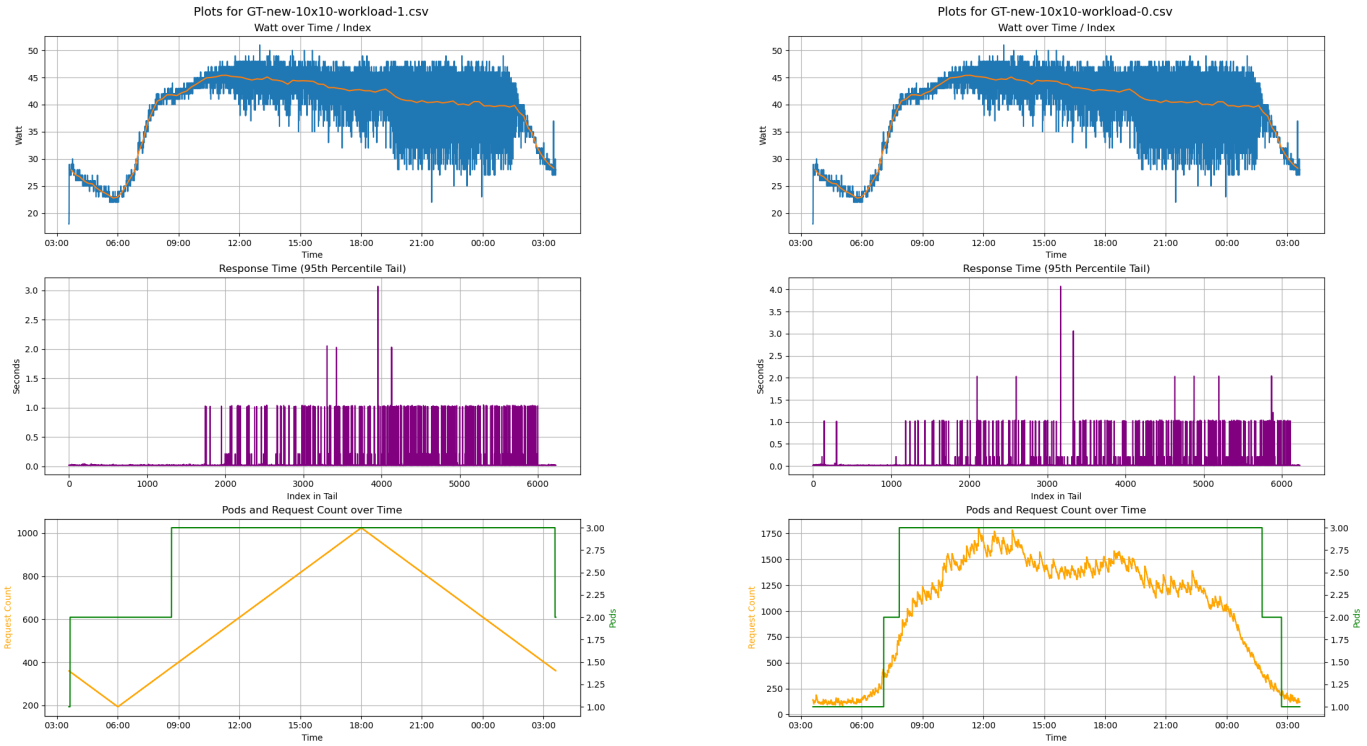


(b) Results from base-case test with the real workload trace.

Fig. 14: Results from base-case tests.

which means that our solution does not add too much computational overhead to the system compared to that of the HPA. It is however important to note that this is the power usage of the cluster, which does not include the **Forecaster**, hence the power usage of our system would possibly be significantly higher since the **Forecaster** is the most computationally heavy part of our system.

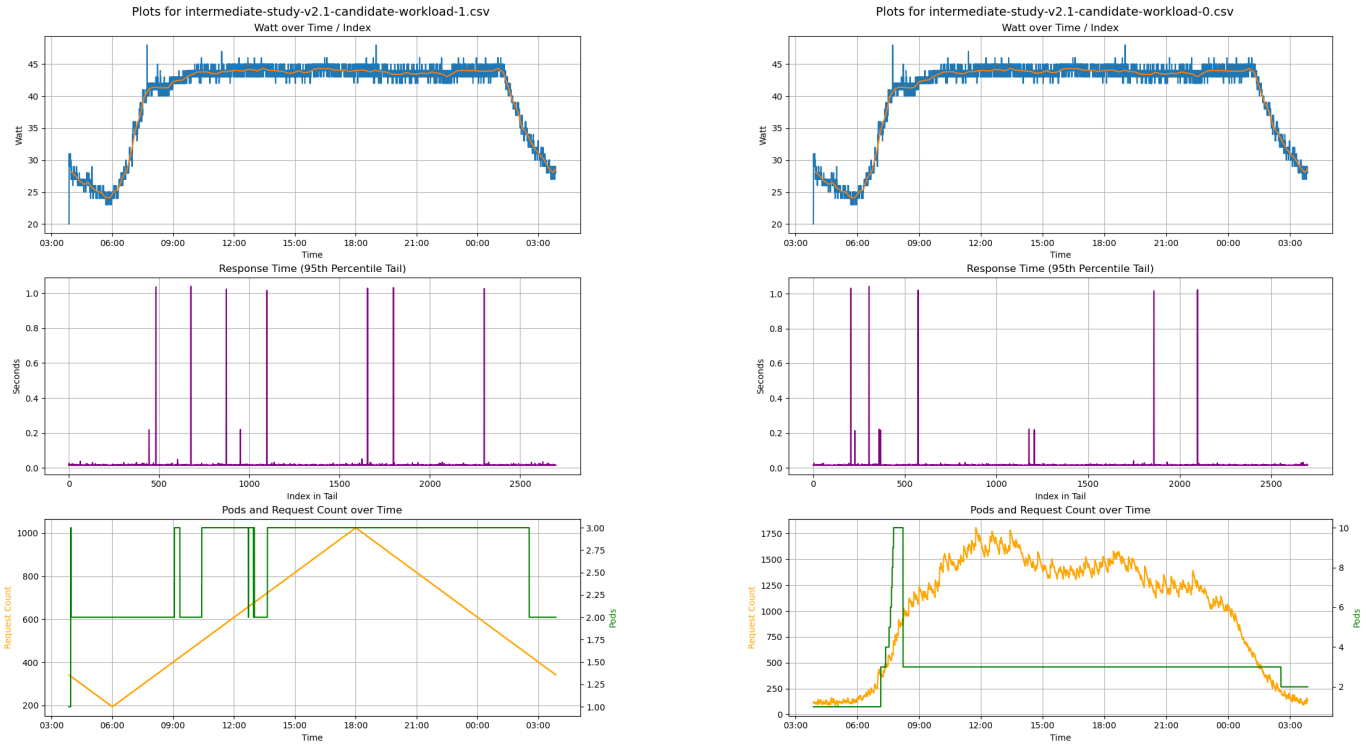
3) Pod Count: If we take a look at the results for the Kubernetes HPA as shown in *Figure 15* (p. 15) we can see that the pod count generally follows the curve of the incoming requests in both workloads, where it rises to three pods during peak load and one pod during the lowest load. If



(a) Results for Ground Truth on the sinusoidal workload.

(b) Result for Ground Truth on the real workload trace.

Fig. 15: Results for the Ground Truth tests (HPA).



(a) Results for our system using the sinusoidal workload.

(b) Result for our system using the real workload trace.

Fig. 16: Results of the test on our Autoscaling system.

we compare this to the test of our Autoscaling system as shown in *Figure 16* (p. 15) we can see that for the workload trace, our system roughly follows the same pattern, with a difference being the beginning of the rise in load, where our systems prediction was a bit off and therefore scaled incrementally to 10 pods, but after retraining came down to the same level as the HPA. If we then look at the sinusoidal workload, we can see that it also follows roughly the same pattern as the HPA, but fluctuates a bit more during the rise towards the peak load. If we then take a look at the average pod count in *Table VIII* (p. 17), we can observe that both the sinusoidal as well as the workload trace, have very similar pod counts compared to that of the HPA, where it only uses between 2% and 5% more pods respectively.

E. RMSE and model choice

As discussed in III-J, the RMSE should be considered for the forecasts, therefore in this section we will evaluate the RMSE and compare it to the RMSE of the same models in our preliminary research. In table IX the chosen model(s) is shown along the average RMSE value for the duration of that test. If we compare the RMSE of the models to that of our preliminary research [23] we can see that the RMSE of the TCN model used on the sinusoidal workload was 59.44% lower than the previous results of the same model. The FFT model used on the workload trace, however performed around 53.74% worse than it did in the preliminary research, while still having a very low RMSE. In addition to this, even though all the models kept being retrained, the same model was chosen for both traces on all forecasts during the duration of the test.

VI. Discussion

Our Autoscaling system, including ML pipeline, model training and deployment, is able to scale the deployments before the loads increases, minimizing the average response time by between 13.95% and 20.53% compared to the HPA, while lowering the response times above 1 second with between 93% and 95.28%. The Autoscaling system selects the same models forecast continuously for the same deployment, but the selected model differs between workloads. The Autoscaling system exhibits an average pod count of between a 2% to 5% increase compared to the Kubernetes HPA.

As confirmed above, the proposed Autoscaling system outperforms the Kubernetes HPA, but some things to consider include: For deployments with highly stochastic workloads, which appears to produce seemingly random historical data, it may be difficult for the models to recognize a pattern, which can potentially degrade the performance of the Autoscaling system, which also explains why the workload trace test ended up using the FFT model since this excels in cases like that. Further training and a broader time frame may help the baseline models recognize the patterns with time. Furthermore, the performance of the models, and thus the ability to scale accordingly may be limited for certain deployments, until enough historical data has been acquired and the models have had time to recognize the patterns for the specific deployment. The complexity of our system may explain the generally larger power usage. Since the model training and tuning requires a fair amount of computational power this may introduce a computational overhead in maintaining the models and updating the forecasts. This may be a problem for edge clusters with limited hardware, since this can increase the model training time and in the worst case inhibit the training process. This is especially prominent for the PyTorch- and Neural Network models, since these have much better efficiency in the presence of a GPU. If a GPU is not available, these may take a long time to train. This could be solved by pinning the ML deployments to a node with more computational power, or a GPU. Noisy data with a large granularity can make it hard to predict small steps into the future, due to the stochastic nature of such data, which can fluctuate a lot during short periods of time, resulting in patterns only becoming visible when inspected in a broader perspective or perhaps not at all. As a result the Autoscaling system should only be enabled for deployments which exhibit clear seasonality and patterns. The **Forecaster** supports true ensembling using the Darts ensembling methods, but is not yet supported as an endpoint in our API, since this requires careful model selection, as to which models should actually comprise the ensemble model. A naive way to decide which models to use in the ensemble model, is by simply selecting the top three compatible best-performing models as candidates. Since all models are trained continuously regardless, these can be used as candidate models for the ensemble model. Since the ensemble model in Darts have the same methods as the regular models, it can be also be saved in the database, and thus compete along the other models, as ensembling does not always guarantee a better forecast [32]. This also allows for *multi-level ensembling*, where ensemble models are used as candidate models for new ensemble models [33]. When deciding which model performs the best, we only consider the RMSE. Considering other error metrics such as MAPE, MAE and SMAPE to provide a more robust performance measurement could be considered. The different metrics could then be weighted and added together to get a single error metric which considers all the error metrics. The Autoscaling system provides forecasts from different models which is, due to the fact that different models are chosen for different workloads, a solid addition to the system. However, the first model chosen persists throughout the entire test, which imply that the historical data, may be slightly influenced by the models forecast and thus the scaling decision. Since our test applications takes a short amount of time to spin up, the HPA can relatively quickly recover from the under-provisioning, but if the pod creation time was larger our Autoscaling system may prove even better with regards to the response time, since the HPA's pod creation delay is more influential. The power consumption of the Autoscaling system is slightly larger than that of the HPA, which is understandable since this is directly tied to the slightly higher pod count. In addition, the complexity of the Autoscaling system may further add to the increased power consumption. Since the measured power consumption only considers the cluster, and the model training takes place on a separate machine outside the cluster, this may be larger than what is measured, and does thus not truly reflect the actual power consumption of the Autoscaling system.

Test	Response Time (s)	Avg. Pods	Avg. Watts	% (0–1) (s)	% (1–2) (s)	% (>2) (s)
HPA sinusoidal	0.01385	2.5006	38.5748	99.8146	0.1790	0.0064
HPA workload trace	0.01450	2.7612	38.5734	99.8050	0.1918	0.0032
Our autoscaling (sinusoidal)	0.01192	2.6246	39.7131	99.9870	0.0129	0
Our autoscaling (trace)	0.01203	2.8088	39.7130	99.9907	0.0092	0
Base case (sinusoidal)	0.11899	1.0000	35.9998	92.9697	5.1089	1.9213
Base case (trace)	0.02674	1.0000	35.9988	98.7565	1.2052	0.0383

TABLE VIII: Average response time, pod count, power usage, and latency distribution by workload.

Test	Avg. Error (RMSE)	Model Chosen
Our autoscaling (sinusoidal)	0.119073155877611 18	TCN
Our autoscaling (trace)	0.1204161519889773	FFT

TABLE IX: Error metric and selected forecasting model for each workload scenario.

VII. Conclusion

This thesis presented a comprehensive approach to predictive autoscaling in Kubernetes environments using machine learning time series forecasting models. The research in this paper demonstrates that anticipating resource demands and proactively adjusting resources can significantly improve the performance compared to current solutions such as the HPA. We developed and evaluated a complete Autoscaling system, that act as a custom Kubernetes controller for predictive scaling, and established a methodology for evaluating autoscaling systems. Our results show that our predictive Autoscaling system reduces the average response time by between 13.93% and 20.53% while reducing the amount of requests above one second by between 93% and 95.28% depending on the workload according to our testing. It also shows very promising results in terms of adaptability since the system was able to outperform the HPA on the sinusoidal workload which it had not been pre-trained on. The system has been designed with Kubernetes as well as production readiness in mind, incorporating monitoring, and integration capabilities with the broader Kubernetes ecosystem as well as the use of as few dependencies as possible. While some limitations exist, particularly for applications with highly unpredictable workloads, the benefits of predictive autoscaling is very promising for an array of different applications running in Kubernetes. As the amount of containerized applications continue to increase and the complexity of cloud application architectures also increases, predictive resource management will become more and more important for maintaining performance, efficiency and cost-effectiveness. Our system shows that it is possible to improve the QoS significantly while only using slightly more power compared to the HPA, but more testing is still needed to validate this further, due to the nature of ML as well as the variance in cloud applications. Our contribution with this research represents a significant step toward realizing the full potential of predictive autoscaling, to provide more elasticity to cloud systems, while also being open source and reproducible.

A. Future work

In the following section, various future improvements to the Autoscaling system will be discussed.

1) More testing: While our tests show very promising results we limited our investigations to a single test case due to time constraints. In the future we would like to conduct a variety of different tests of the system. For instance, it would be highly beneficial to expose the system to tests with even more varying workloads, harder to predict workloads, longer test periods, as well as tests with scaling of a different test application that could for instance take much longer to start up, to further validate the robustness and adaptability of the Autoscaling system.

2) Meta model: train a model to select the best model: In our current solution we have implemented a method for selecting the best forecast, where we get a forecast from all available models and then we select the best model based on the RMSE. Instead of taking this approach a meta model could be trained that received all the forecasts from the models, and then made various error calculations as well as looked at historical data to determine the best forecast provided by the models, making the forecast selection more precise.

3) Feature selection: To further improve the precision of the forecasts as well as the training time, we would in the future also like to implement feature selection provided by the already implemented tuning functionality. The feature selection works by, ranking the hyperparameters by importance for the tuning. This makes it possible to narrow the hyperparameter search space by excluding less important parameters and therefore decrease the tuning time while also minimizing the possibility of the models spending too much time tuning hyperparameters that do nothing.

4) Seasonal models: It would also be interesting to look into having various models for different seasons. Since loads on different deployments tends to follow various patterns depending on different parts of the week, and also different parts of the year, it could improve the performance of the system to have different models that are trained specifically on for instance weekdays and weekends or various weeks during the year where there is more or less traffic than usual. This would mean that when a certain part of the week or year arrives, the system would select the appropriately trained model for this season, hence make the forecasts more precise.

5) Implement self retuning: While we have provided the backend functionality in the **Forecaster** to retune the hyperparameters of the models, due to time constraints we didn't implement the functionality for the user to issue a model retuning manually from the GUI. The retuning differs from the retraining process by working with completely new model instances. The process simply carry out the search for optimal hyperparameters again. This should function as a fallback, should the performance of the models not increase solely

by retraining. In the future we would like to implement this feature, to make the forecasts from the models even more precise.

6) Vertical Autoscaling: Like our preliminary research [23] mentioned it would be interesting to train a model choosing between horizontal and vertical scaling. After selecting which way to scale, another set of models could be used to predict the actual values, which could be beneficial since it is much faster to do vertical scaling and also if sudden unpredictable spikes occur, the system would be able to quickly vertically scale all the already running pods to accommodate this.

7) Distributed retraining: To allow for even faster retraining of the models, it could be beneficial, if available, to utilize multiple GPU's to retrain the models during deployment. This could be done in a way where the training part of the **Forecaster** is abstracted away and deployed on one or more machines that has one or multiple GPU's available while still keeping the forecasting part of the **Forecaster** in the cluster. This would allow for creating a queue where the **Forecaster** could request retraining of the models for a certain deployment, and then it would get done when the GPU's on one of the machines is available, while still providing new forecasts from the models in their existing states. This would also scale very nicely, since the retraining simply would happen when the GPU's are available.

8) Take Power Usage into Account: Our preliminary research also mentions taking power usage into account. It would be interesting, to see if the models would be able to predict the CPU usage and determine the least amount of pods necessary for the amount of incoming requests to minimize the power usage, and make the Autoscaling system a more environmentally friendly option than current solutions.

9) Overall system complexity: The predictive Autoscaling system has a lot of communication which can be avoided if the architecture is designed using more robust patterns. If this project were to have a third iteration, the SQL database and the repeated API calls to Prometheus and the **Forecaster** could be avoided by designing the system around an event-based database like Apache Kafka and have both the **Autoscaler** and the **Forecaster** consume and produce on topics rather than sending API calls and checking status codes constantly. The system would effectively act the same but the complexity of our code would be considerably reduced with the absence of a traditional database and predefined scaling, forecasting and training timing.

10) GUI Extensions / Improvements:

- Deploying the GUI in the same container as the API is not ideal in production environments, where it should be deployed separately to allow for load balancing. For our use case, we consider it to be sufficient, since independent scaling of the backend and frontend would not contribute further to our test cases than concurrent scaling does.
- In the future we would also like to provide the possibility to view the forecasts in the frontend by different time windows, such that the user would be able to select between a variety of different windows and by doing that obtaining more fine-grained knowledge of the behavior of the system. The GUI could be further extended by displaying previous time windows with the historical data displayed along the forecast, such that the user is able to gain insight in the forecasting performance and verify whether it is sufficient or if some of the settings should perhaps be modified.
- The GUI could further include a hyperparameter configuration page for each deployment, where the user is able to define the hyperparameter search space themselves. This could, if configured correctly, reduce the time it takes to tune by reducing the search space or simply leaving out parameters which are known to be indifferent.

11) Usability Tests: In the future we would also like to test the usability of the frontend. Currently it provides only the features we thought ourselves would be relevant as well as the design we thought works well that has been implemented, but it would be very beneficial to make some usability tests with actual system administrators which could provide valuable insights into how a great user interface would be, if this was to be an actual product.

PROCESS

From our past experiences from our preliminary research [23] we chose a different approach for managing the project, to make the project progress more smooth. We choose to use a mix of daily check-ins from scrum as well as setting and planning various milestones throughout the semester. We had a status meeting at the end of each month where we looked at what we had accomplished and how the state of the development was. We then planned the upcoming monthly milestone where the next steps in development would be discussed and prioritized in order to full fill our problem statement. We used again a Kanban board in GitHub which can be seen in *Figure 17* (p. 19) where we made an issue representing the milestone, and then added issues from our backlog to that milestone. We also made a Gantt chart which can be seen in *Figure 18* (p. 19) where all the milestones for the entire work period could be seen. In addition to this we tried to prioritize the different issues by both importance and time-frame, which meant that the planning was much easier than last semester. This worked out great, and we were able to reach all of our milestones. However, the act of trying to predict the time-frame turned out to quite difficult in the beginning and we were often finished with the milestone earlier than expected. This however got better during the semester and we were able to match the amount of work much better to the milestones as time went on. This process allowed us to make very good progress all throughout the semester. However at different points we encountered that some of the tasks made, was very large which meant that some of the group members encountered some idle time waiting for the task to be finished. So in the future we would choose a similar process, but make sure that the tasks made were smaller such that we could take advantage of all the group members at all times.

ACKNOWLEDGMENT

The authors would like to thank Turftank for providing production data to train and evaluate our models even though

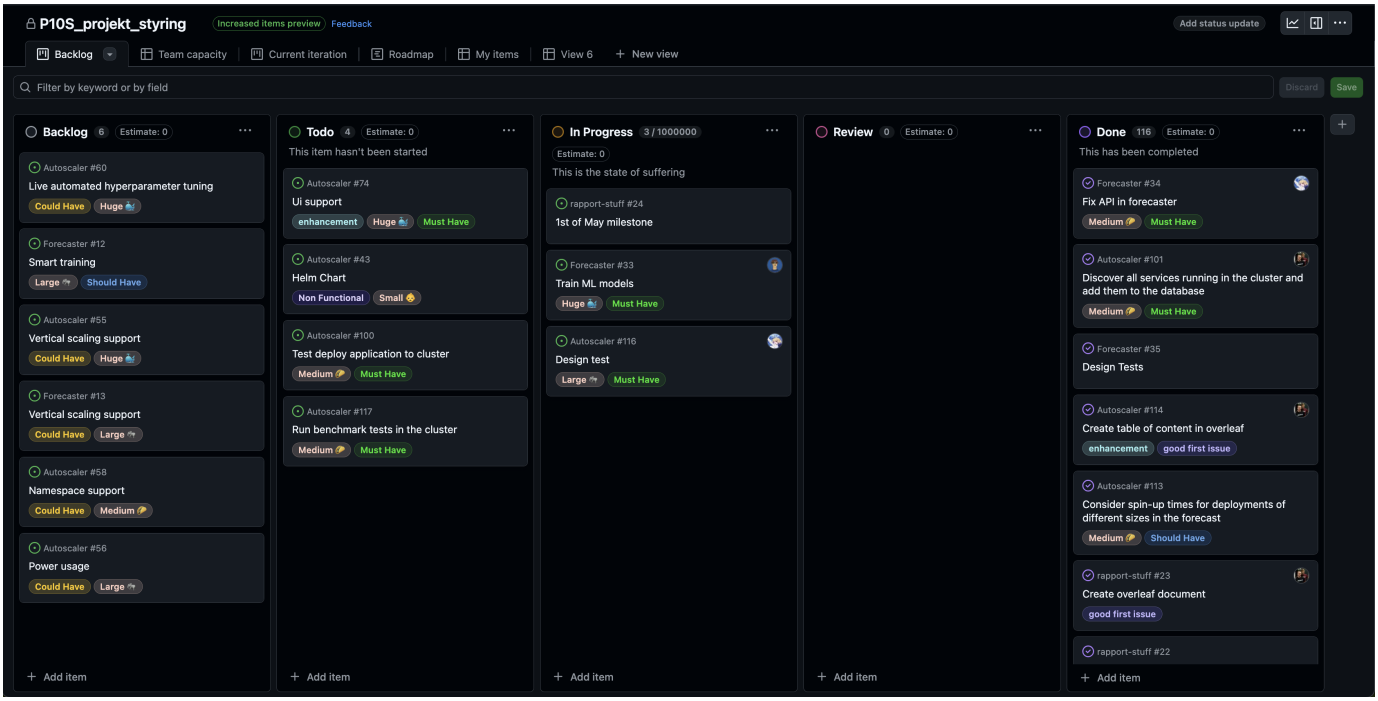


Fig. 17: A screenshot of the Kanban board used during the semester.

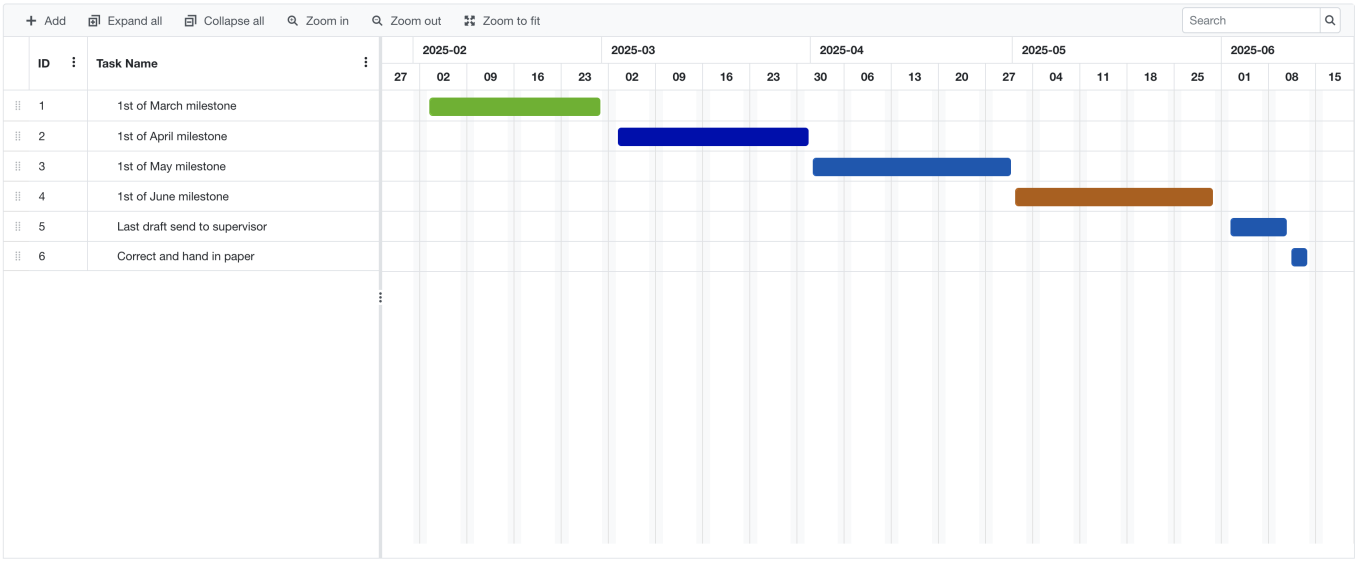


Fig. 18: A screenshot of our Gantt chart.

we found the patterns in the data too subtle. We also express our gratitude to our academic advisor Michéle Albano at Aalborg University of Denmark for his guidance and support throughout this research.

Appendix A
Source code

The source-code for the system, as well as the tests, datasets for results, generator etc. can be found here: https://github.com/orgs/aau-p9s/repositories

Appendix B
RAM usage when training

We have tested the simultaneous training of all the models on two different deployments on a Lenovo Thinkpad P14S Gen 5 with 32 GB of RAM, the peak memory usage can be seen in: *Figure 19* (p. 20).

Appendix C
System tests resource requests

The system tests was conducted on a cluster consisting of pods with resource requests listed in *Table X* (p. 20).

```
CONTAINER ID   NAME                        CPU %     MEM USAGE / LIMIT     MEM %     NET I/O           BLOCK I/O         PIDS
08d1b669e191   deployment-forecaster-1     100.15%   3.877GiB / 27.06GiB   14.33%    56.2MB / 41.3kB   219MB / 2.59MB    61
639e5ad6364e   deployment-db-1             0.00%     66.81MiB / 27.06GiB   0.24%     41.8kB / 56.2MB   549kB / 573kB     8
```

Fig. 19: RAM usage when training the models of two deployments.

Pod	vCPU	Memory	Pod counts
Generator	500m	1000Mb	n
API	500m	100Mb	n * m
Autoscaler	500m	1000Mb	1
Forecaster	unlimited*	unlimited*	1

TABLE X: System tests resource requests (n = workload type count, m = workload api replicas) *system dependant

REFERENCES

[1] (March 2025) State of cloud native development q1 2025. [Online]. Available: https://www.cncf.io/wp-content/uploads/2025/04/Blue-DN29-State-of-Cloud-Native-Development.pdf

[2] (December 2023) What is auto scaling? [Online]. Available: https://www.ibm.com/think/topics/autoscaling

[3] (March 2025) Horizontal pod autoscaling. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[4] (January 2025) What is automl? [Online]. Available: https://www.ibm.com/think/topics/automl

[5] S. Alla and S. K. Adari, *What Is MLOps?* Berkeley, CA: Apress, 2021, pp. 79–124. [Online]. Available: https://doi.org/10.1007/978-1-4842-6549-9_3

[6] Automl: Getting started. [Online]. Available: https://developers.google.com/machine-learning/crash-course/automl/getting-started

[7] Automl: Benefits and limitations. [Online]. Available: https://developers.google.com/machine-learning/crash-course/automl/benefits-limitations

[8] Kubernetes website. [Online]. Available: https://kubernetes.io/

[9] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Adaptive ai-based auto-scaling for kubernetes," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 599–608.

[10] L. Ju, P. Singh, and S. Toor, "Proactive autoscaling for edge computing systems with kubernetes," *Association for Computing Machinery*, 2022. [Online]. Available: https://doi.org/10.1145/3492323.3495588

[11] What is time-series forecasting? [Online]. Available: https://cloud.google.com/learn/what-is-time-series?hl=en

[12] M. Shah, Dhawani & Thaker, "A review of time series forecasting methods," *INTERNATIONAL JOURNAL OF RESEARCH AND ANALYTICAL REVIEWS. 11. 749. 10.1729/Journal.38816.*, 2024. [Online]. Available: https://www.researchgate.net/publication/379862462_A_Review_of_Time_Series_Forecasting_Methods

[13] (18 March 2024) What is ensemble learning? [Online]. Available: https://www.ibm.com/think/topics/ensemble-learning

[14] O. Cordon, P. Kazienko, and B. Trawinski, "Hybrid ensemble machine learning approach for cancer prediction," *Bima Journal of Science and Technology*, vol. Vol. 8 No. 1b (2024), pp. 206–217, 07 2024.

[15] ——, "Special issue on hybrid and ensemble methods in machine learning," *New Generation Comput.*, vol. 29, pp. 241–244, 07 2011.

[16] (2025) Keda website. [Online]. Available: https://keda.sh/

[17] (February 14, 2022) Introducing predictkube - an ai-based predictive autoscaler for keda made by dysnix. [Online]. Available: https://keda.sh/blog/2022-02-09-predictkube-scaler/

[18] (2025) Kubeflow website. [Online]. Available: https://www.kubeflow.org/

[19] S. Yuan, H.; Liao, "A time series-based approach to elastic kubernetes scaling," *Electronics 2024*, 2024. [Online]. Available: https://www.mdpi.com/2079-9292/13/2/285

[20] P. Naayini, "Building ai-driven cloud-native applications with kubernetes and containerization," *International Journal of Scientific Advances (IJSCIA), Volume 6— Issue 2*, Apr 2025. [Online]. Available: https://www.ijscia.com/wp-content/uploads/2025/04/Volume6-Issue2-Mar-Apr-No.862-328-340.pdf

[21] Oss mlops platform. [Online]. Available: https://github.com/OSS-MLOPS-PLATFORM/oss-mlops-platform

[22] J. Koskinen, "Designing a machine learning pipeline with continuous training for time series forecasting," Master's thesis, Faculty of Science - University of Helsinki, 2024. [Online]. Available: https://helda.helsinki.fi/server/api/core/bitstreams/fa6cb5a7-d5b2-4aa6-9c54-6c7aa8857636/content

[23] J. W. Fog, J. J. T. Møller, T. M. Jensen, D. Taibi, and M. Albano, "Comparing neural and statistical time-series models for proactive auto-scaling in kubernetes," in *Proceedings of the 2025 IEEE International Congress On Intelligent And Service-Oriented Systems Engineering (CISOSE 2025)*, 2025, to be presented, July 21-24, 2025, Tucson, Arizona, USA.

[24] A. Tawakuli, B. Havers, V. Gulisano, D. Kaiser, and T. Engel, "Survey:time-series data preprocessing: A survey and an empirical analysis," *Journal of Engineering Research*, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2307187724000452

[25] Strato. [Online]. Available: https://hpc.aau.dk/strato/

[26] Locust github repository. [Online]. Available: https://github.com/locustio/locust

[27] Dbup docs. [Online]. Available: https://dbup.readthedocs.io/en/latest/

[28] Horizontal pod autoscaling. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[29] Kubernetes metrics reference — kubernetes. [Online]. Available: https://kubernetes.io/docs/reference/instrumentation/metrics

[30] Resource management for pods and containers. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

[31] Cpu platforms. [Online]. Available: https://cloud.google.com/compute/docs/cpu-platforms

[32] Ensembling models. [Online]. Available: https://unit8co.github.io/darts/examples/19-EnsembleModel-examples.html

[33] W. F. Satrya, R. Aprilliyani, and E. H. Yossy, "Sentiment analysis of indonesian police chief using multi-level ensemble model," *Procedia Computer Science*, vol. 216, pp. 620–629, 2023.