# GraphOpticon: A Global Proactive Horizontal Autoscaler for Improved Service Performance & Resource Consumption

Theodoros Theodoropoulos[a,c,*], Yashwant Singh Patel[b], Uwe Zdun[a], Paul Townend[b], Ioannis Korontanis[c,d], Antonios Makris[d] and Konstantinos Tserpes[d]

[a]University of Vienna, Software Architecture Research Group, Vienna, 1090, Austria
[b]Umeå University, Dept. Computing Science, Umeå, 90187, Sweden
[c]Harokopio University of Athens, Omirou 9, Athens, 17778, Greece
[d]National Technical University of Athens, Heroon Polytechniou 9, Athens, 15780, Greece

ARTICLE INFO

*Keywords*:
Cloud Computing
Green Computing
Graph Neural Networks
Deep Learning
Resource Usage Forecasting
Resource Consumption
Service Performance

ABSTRACT

The increasing complexity of distributed computing environments necessitates efficient resource management strategies to optimize performance and minimize resource consumption. Although proactive horizontal autoscaling dynamically adjusts computational resources based on workload predictions, existing approaches primarily focus on improving workload resource consumption, often neglecting the overhead introduced by the autoscaling system itself. This could have dire ramifications on resource efficiency, since many prior solutions rely on multiple forecasting models per compute node or group of pods, leading to significant resource consumption associated with the autoscaling system. To address this, we propose GraphOpticon, a novel proactive horizontal autoscaling framework that leverages a singular global forecasting model based on Spatiotemporal Graph Neural Networks. The experimental results demonstrate that GraphOpticon is capable of providing improved service performance, and resource consumption (caused by the workloads involved and the autoscaling system itself). As a matter of fact, GraphOpticon manages to consistently outperform other contemporary horizontal autoscaling solutions, such as Kubernetes' Horizontal Pod Autoscaler, with improvements of 6.62% in median execution time, 7.62% in tail latency, and 6.77% in resource consumption, among others.

## 1. Introduction

The growing complexity of distributed computing environments introduces variability in workload demands, latency issues, and data management challenges, necessitating automated resource management strategies. Resource scaling (1) is the process of dynamically adjusting computational resource allocation, such as CPU and memory, based on workload fluctuations to optimize performance and cost efficiency (2). Horizontal scaling refers to the process of increasing or decreasing the number of instances in response to demand. Kubernetes[1], a widely adopted container orchestration framework, plays a crucial role in managing these scaling processes through mechanisms such as the Horizontal Pod Autoscaler (HPA) (3), ensuring efficient allocation of computational resources via pod replication.

In large-scale systems, resource utilization metrics typically exhibit time-series patterns with non-linear behaviors. Recurrent Neural Networks (RNNs) (4) have demonstrated effectiveness in modeling such data distributions. Advanced RNN-based architectures, such as Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) (5), can predict resource utilization trends, enabling more efficient system orchestration. While conventional time-series forecasting models focus on single-step-ahead predictions, multi-step-ahead forecasting strategies provide a sequence of future values, allowing for more granular resource management. This enables the implementation of proactive resource allocation and scaling techniques that mitigate bottlenecks and improve overall efficiency.

Encoder-Decoder (ED) Deep Learning (DL) architectures have shown superior performance in multi-step forecasting compared to traditional prediction models (6). The encoder processes a variable-length sequence, transforming it into a structured representation that the decoder then utilizes to generate predictions. Recent advancements in handling non-Euclidean data have also led to the emergence of Graph Neural Networks (GNNs) (7), which excel in solving problems with spatial components. This capability stems from their inherent ability to leverage and utilize the spatial characteristics of data related to a given problem. Spatio-temporal GNNs (8) have been particularly successful in forecasting problems, as their architecture allows them to simultaneously capture both spatial and temporal dependencies. This is achieved by using graph convolutions to model

*This research was funded in whole or in part by the Austrian Science Fund (FWF) project CQ4CD, Grant-DOI: 10.55776/I6510. For open access purposes, the author has applied a CC BY public copyright license to any author accepted manuscript version arising from this submission. This project has received funding from the European Union's Horizon 2020 research and innovation programmes under grant agreements No 101135775 (PANDORA) and No 101120990 (SOPRANO). The work reflects only the authors' view, and the EU Agency is not responsible for any use that may be made of the information it contains.

theodoros.theodoropoulos@univie.ac.at (T. Theodoropoulos); yashwant.patel@umu.se (Y.S. Patel); uwe.zdun@univie.ac.at (U. Zdun); paul.townend@umu.se (P. Townend); gkorod@hua.gr (I. Korontanis); antoniosmakris@mail.ntua.gr (A. Makris); tserpes@mail.ntua.gr (K. Tserpes)
ORCID(s): 0000-0002-4618-4891 (T. Theodoropoulos)

[1] https://kubernetes.io/

spatial dependencies and RNNs to capture temporal dependencies in alignment with the Encoder-Decoder paradigm. Proactive horizontal autoscaling strategies involve leveraging such models to enhance resource allocation efficiency by dynamically adjusting the number of processing nodes in anticipation of workload fluctuations.

A review of the scientific literature reveals that prior works on proactive horizontal autoscaling focus on improving resource consumption solely from the perspective of the workloads being executed. Hence, they do not consider the resource consumption of the autoscaling system itself. In proactive horizontal pod autoscaling, workload resource consumption refers to the resources utilized by the running application workloads within a Kubernetes cluster. This metric is essential for assessing system load and making informed scaling decisions. Conversely, autoscaling system resource consumption accounts for the resources used by the autoscaler itself. An efficient proactive autoscaling system should minimize its own resource footprint while effectively scaling workloads to maintain performance and optimize resource utilization. Many of the prior works are designed to employ multiple forecasting models (one per pod or group of pods), which would introduce significant overhead if integrated into the scaling decision-making process.

Employing multiple DL models in decision-making can improve resource management by enabling task specialization. However, this approach also increases resource usage, as each model requires computational resources for both training and inference. Training multiple models is particularly resource-intensive, and frequent updates can further amplify resource demands. Moreover, managing the outputs of multiple models introduces computational overhead and potential latency, affecting performance in real-time applications. Therefore, it is essential to minimize the number of forecasting mechanisms deployed in a system, ensuring efficient resource scaling and management while mitigating the subsequent resource consumption. This overhead could potentially offset the benefits of proactive resource allocation and deallocation.

These observations motivated us to propose GraphOpticon. GraphOpticon is a proactive horizontal autoscaling solution designed to enhance service performance while minimizing autoscaling system resource consumption by leveraging only a singular global forecasting model instead of numerous local specialized ones. Towards achieving this goal, GraphOpticon is based on the implementation of information fusion and distillation processes, driven by the characteristics of the deployed services, as well as by the ontological relations that these services form with each other in order to further refine the generalization capabilities of the employed forecasting model that is based on Spatiotemporal GNNs. The key contributions of our research are:

- We advocate for the use of a singular global forecasting model (instead of numerous local ones) in order to establish resource-efficient proactive horizontal autoscaling.
- We propose GraphOpticon, a novel global horizontal autoscaling solution that leverages the information fusion & distillation processes to improve both service performance and workload resource consumption while minimizing the underlying autoscaling system resource consumption.
- We extensively analyze the architectural decisions, results, and potential ramifications that are intertwined with the use of a global proactive horizontal autoscaling solution.

The remaining sections of this work are organized as follows. Section 2 provides the current research status of relevant proactive horizontal autoscaling solutions. Section 3 establishes the motivation behind this study. Section 4 presents the problem formulation. Section 5 describes the proposed 'GraphOpticon' solution. Section 6 discusses several implementation aspects that are intertwined with GraphOpticon. Section 7 focuses on the experimental results to evaluate the efficiency of the proposed solution. Finally, Section 8 concludes this study and discusses potential directions for future work.

## 2. Related Work

This section explores recent advancements in proactive horizontal autoscaling techniques. To achieve this, we meticulously analyze numerous scientific works in the frame of various aspects, such as forecasting models, workflows, datasets, evaluation tools, and evaluation metrics (as explained in Table 2). An overview of this analysis is presented in Table 1. Aside from showcasing the fact that GraphOpticon constitutes an advancement toward more efficient resource orchestration solutions, the purpose of this section is to guarantee that the subsequent experimental evaluation will be conducted in a manner that is aligned with the corresponding scientific literature.

Forecasting models are widely explored in the literature for resource orchestration. These models play a crucial role in proactive autoscaling, where they predict future resource needs and scale infrastructure accordingly. To improve the forecasting performance of container-based load prediction models, Tang et al. (9) design 'Fisher,' which consists of metric selection and neural network training components. The metric selection component identifies relevant metrics using a novel shape-based time series clustering technique. Subsequently, a Bidirectional LSTM (BiLSTM) is applied to predict the one-step-ahead workload. Patel et al. (10) propose a dynamic consolidation technique for cloud systems. They present a clustering-based stacked bidirectional LSTM model to forecast the future CPU and memory usage of machines. Utilizing the prediction results, they design different consolidation approaches to show improvements in energy, migrations, and SLA violations. Radhika et al.(11) present ARIMA and LSTM algorithms to predict future CPU usage from a 3-tier architecture of web applications hosted on a private cloud platform. The LSTM approach notably shows

the high accuracy and effectiveness in forecasting future web application demands. Theodoropoulos et al. (6) introduced a novel Encoder-Decoder architecture that utilizes stacked LSTM and BiLSTM layers at both the encoder and the decoder parts of the model. This approach managed to provide superior results in terms of service demand forecasting accuracy against a plethora of contemporary forecasting solutions, is referred to as the Hybrid LSTM Encoder-Decoder. Following works (12) have also demonstrated the superiority of the Hybrid LSTM Encoder-Decoder in the frame of resource consumption forecasting, surpassing competitors such as LTMS, Bi-LSTMs, GRUs, the CNN-LSTMs, as well as various other ED architectures, providing slightly worse results only when compared against Spatiotemporal GNNs that are based on Discrete-Time Dynamic Graphs (DTDG) (13) paradigm.

Proactive horizontal autoscaling strategies leverage machine learning and predictive models to enhance resource allocation efficiency by dynamically adjusting the number of processing nodes in anticipation of workload fluctuations. Various approaches have been proposed, each aiming to improve certain aspects of the autoscaling process. For instance, some of them focus on improving performance in the frame of evaluation metrics such as Inference Time, Latency, Availability, Elastic Speedup, Execution Time, and Response Time. Violos et al. (14) introduce an 'Intelligent Horizontal Proactive Autoscaling' strategy that utilizes resource usage metrics (CPU) of processing edge nodes to make timely and efficient scale-up and scale-down decisions. Their approach is based on a double-tower deep learning-driven topology, which simultaneously analyzes and distinguishes the time-series resource usage metrics for local processing nodes and the edge infrastructure's aggregated resource metrics. The framework shows improvements in latency and execution time. Kakade et al. (15) designed a Bi-LSTM model-based proactive autoscaler to predict future demands and automatically scale containers. Their experiments with a three-node Kubernetes setup indicate that Bi-LSTM outperforms stacked LSTM, while the proactive autoscaler achieves better performance than the default Kubernetes autoscaler. Marie-Magdelaine et al. (16) develop an LSTM-based proactive auto-scaling approach that dynamically adjusts the resource pool horizontally and vertically to optimize availability and minimize latency in cloud-native applications.

Aside from improving performance, other works aim to also enhance resource efficiency in the frame of Inference Time, Training Time, over-provisioning, under-provisioning, and Resource Underutilization. For instance, Imdoukh et al. (17) design an LSTM-based adaptive forecasting model that predicts future HTTP demands to determine the required number of containers, minimizing delays caused by starting and stopping active containers and eliminating oscillations during scaling operations. Dang-Quang et al. (18) propose a BiLSTM-based proactive autoscaler to predict future HTTP workloads, incorporating a 1-minute 'Cool Down Time' (CDT) interval to mitigate oscillations and a resource removal approach to handle underutilized resources. Similarly, Dogani et al. (19) introduce an attention-based GRU encoder-decoder (K-AGRUED) model for proactive autoscaling in Kubernetes, reducing scaling operations and under-provisioning compared to the Kubernetes horizontal pod autoscaler. Dogani et al. (20) devise a proactive auto-scaling approach for edge environments using FedAvg and multi-step workload forecasting with a Bidirectional Gated Recurrent Unit (BiGRU). Their results show improvements in resource overprovisioning, underprovisioning, and elastic speedup while reducing data transmission between edge nodes and cloud servers. Ahmad et al. (21) propose Smart HPA and ProSmart HPA, two resource-efficient horizontal pod autoscalers; Smart HPA applies a reactive scaling policy, while ProSmart HPA leverages Prophet-based machine learning for proactive scaling. Their results demonstrate improvements in resource utilization by mitigating underutilization, overprovisioning, and underprovisioning.

Aside from the aforementioned resource efficiency evaluation metrics, the number of pod replicas plays a significant role in the overall resource consumption. The number of pod replicas directly impacts resource consumption by determining how many instances of a specific application or service run simultaneously within a Kubernetes cluster. Each pod replica consumes CPU, memory, and other resources based on the application's requirements. Increasing the number of replicas spreads the workload, leading to higher resource utilization across the cluster, which can improve performance and availability but also increase overall resource consumption. Conversely, decreasing replicas reduces resource usage but may also impact the application's ability to handle traffic or provide high availability. Thus, managing pod replicas helps balance resource efficiency with performance needs. Following this line of thought, various works have emerged. Yadav et al. (22) design an autoscaler that enables horizontal scaling for Docker containers using a Support Vector Regression (SVR)-based proactive method, leveraging the IBM MAPE-K computing platform to optimize resource allocation in terms of the number of the deployed replicas. Nguyen et al. (23) introduce a graph-based proactive horizontal pod autoscaling strategy for microservices, employing an LSTM-GNN hybrid model that first predicts upcoming workloads and then determines the optimal number of pods required for efficient resource allocation, leading to significant resource savings. Goli et al. (24) propose 'Waterfall,' a predictive autoscaler using machine learning models such as linear regression, random forest, and SVR to estimate the required replicas for each microservice while considering interdependencies between services. Ju et al. (25) introduce the Proactive Pod Autoscaler for Kubernetes, utilizing multiple user-defined and customizable metrics for workload forecasting, dynamically scaling applications, and outperforming the default Kubernetes pod autoscaler in resource utilization efficiency. Theodoropoulos et al. (26) propose a GNN-LSTM-based approach that uses Spatiotemporal GNNs to forecast CPU consumption and then leverages

these forecasts to conduct proactive horizontal autoscaling, achieving performance gains in execution time and latency, while requiring fewer pod replicas.

Despite their various scientific contributions, all aforementioned works examine resource consumption only from the perspective of workloads. However, they do not account for the potential autoscaling system resource consumption. In proactive horizontal pod autoscaling, workload resource consumption refers to the resource consumption caused by the running application workloads within the Kubernetes cluster. This metric helps determine how much load the system is handling and is crucial for making scaling decisions. On the other hand, autoscaling system resource consumption pertains to the resource consumption caused by the autoscaling system itself. An optimal proactive autoscaling system should minimize its own resource footprint while effectively scaling workloads to maintain performance and resource efficiency. Since various of these works leverage numerous forecasting models (one for each pod or set of pods), the overhead that would derive from their incorporation into the scaling decision-making process would be significant and more than likely negate the improvements in terms of workload resource consumption caused by proactively allocating, and de-allocating compute resources. To the best of our knowledge, no work exists in the corresponding scientific literature that aims at constructing a proactive horizontal autoscaling solution that improves both service performance and workload resource consumption while minimizing the underlying autoscaling system resource consumption. Our work is dedicated to mitigating this research gap.

## 3. Motivation

The use of numerous DL models comes in stark contrast with our goal to construct a proactive horizontal autoscaling solution that improves both service performance and workload resource consumption and minimizes the underlying autoscaling system resource consumption. Having many local models in a system allows for task specialization, which can enhance forecasting accuracy by tailoring each model to capture the unique characteristics of individual compute nodes (27). Local models are designed with architectures, features, and training datasets that optimize them for particular computing environments. However, a critical aspect of enabling reduced resource consumption in the frame of autonomic computing lies in the need to keep the number of forecasting models to a minimum. Each model requires resources to operate, meaning that a system with many local models will generally consume more resources than one with a single, consolidated global model. This demand can be substantial when the models are run frequently, as with real-time CPU forecasting in autonomic systems. Training multiple local models can be very resource-intensive, especially if each model is regularly updated to ensure optimal accuracy. Training typically consumes far more resources than inference, and multiple training cycles for different models can lead to significantly higher energy costs, impacting the sustainability of the system. Balancing the outputs of multiple local models requires additional computational overhead, as the system must aggregate, reconcile, or prioritize forecasts from each model. This integration process can introduce latency, particularly in real-time scenarios, where autonomic responses are critical for maintaining system stability or optimizing resource allocation.

Motivated by the aforementioned drawbacks of leveraging numerous local DL models, we propose the use of a singular global forecasting model for proactive horizontal autoscaling in order to improve service performance & resource consumption. Our main assumption is that one of the key advantages of using a global forecasting model in distributed computing environments is its ability to leverage shared underlying patterns across multiple pods. In many containerized applications, pods exhibit similar usage trends due to common workload types, synchronized traffic patterns, or shared infrastructure constraints. By recognizing these patterns, a global model can generalize across pods, leading to improved forecasting accuracy compared to individual pod-specific models that may overfit to transient fluctuations or noise.

Furthermore, since pods operate within a cluster to host services, their resource consumption is often influenced by their service type and cluster membership. By incorporating these factors, a global forecasting model can effectively capture workload correlations across pods, leading to improved accuracy compared to isolated, pod-specific models that may overfit to transient variations. Pods hosting the same service tend to exhibit similar CPU usage trends due to synchronized request patterns, common workload dependencies, and shared execution environments. For instance, multiple pods serving the same API requests will likely experience correlated resource consumption trends, with peaks during high-traffic periods and lower usage during off-hours. A global forecasting model trained on aggregated data from such pods can identify these recurring patterns and seasonal fluctuations, allowing for more reliable predictions even when dealing with newly deployed or short-lived pods. The impact of shared underlying patterns is particularly valuable in autoscaling scenarios, where Kubernetes dynamically adjusts the number of pods in response to workload changes (28). When a new pod is spawned, it lacks historical resource utilization data, making forecasting difficult for local models. However, a global model can instantly predict its expected resource consumption by leveraging data from existing pods running the same service. This reduces the lag time in making accurate forecasts, ensuring that resource allocation remains efficient and preventing overprovisioning and underprovisioning.

## 4. Problem Formulation

Multi-cluster deployments involve managing and orchestrating applications across multiple independent clusters of pods. These clusters are denoted by the set $C =$

**Table 1**
Analysis of horizontal pod autoscaling approaches.

| Contributors & Year | Forecasting Model | Workflow | Datasets | Evaluation Tools | Evaluation Metrics |
|---|---|---|---|---|---|
| Marie-Magdelaine et al. (16) (2020) | LSTM | Applying LSTM model to dynamically adjust the resource pool, number of replicas and pool of resources | Simulated traffic dataset | Digital Ocean cloud provider's VMs | Availability, Latency, Number of Pod Replicas |
| Imdoukh et al. (17) (2020) | LSTM | Utilizing proactive machine learning method for auto-scaling of Docker containers in response to dynamic workload | HTTP request (logs of Worldcup98) | Python | MSE, $R^2$, RMSE, MAE, Resource Underprovisioning & Overprovisioning, Inference Time |
| Goli et al. (24) (2021) | Linear regression, random forest, support vector regression | Apply ML models to forecast the required number of replicas for each microservice and considers the impact of scaling one microservice on others within a given workload | Generated custom dataset using Teastore application | Compute Canada Arbutus Cloud platform | Response Time, Number of Pod Replicas |
| Ju et al. (25) (2021) | ARMA, LSTM | Incorporates multiple metrics for workload forecasting and later utilizes these predictions to dynamically scale the applications | Random access, NASA datasets | Edge computing cluster | MSE, Response Time, Resource Underutilization |
| Dang-Quang et al. (18) (2021) | BiLSTM | Applying a proactive custom autoscaler using BiLSTM model for handling the dynamic workload | NASA and FIFA World Cup 98 log traces | Python, Google Colab environment | MSE, RMSE, $R^2$, MAE, Resource Underprovisioning & Overprovisioning, Elastic Speedup |
| Yadav et al. (22) (2021) | SVR | Applying SVR model to perform horizontal elasticity for Docker containers | Web service logs of the Complutense University of Madrid | Python, Computing cluster | RMSE, MAE, MSE, Resource Consumption, Elastic Speedup |
| Nguyen (23) (2022) | LSTM, GNN | Applying LSTM for workload prediction and then Graph Convolution Networks for relationship modeling between workload and resource consumption | Microsoft's Azure traces | AWS EC2 instances | MSE, MAE, Number of Pod Replicas |
| Violos et al. (14) (2022) | Feedforward+RNN | Applying horizontal proactive autoscaling to provide scale up and scale down decisions | Alibaba cluster traces | Python, CloudSim Plus | RMSE, MSE, MAE, Execution Time, Tail Latency |
| Dogani et al. (19) (2022) | Attention+GRU | Adapted proactive autoscaling method to predict the multi-step resource usage based on cooldown time | FIFA Worldcup Dataset, NASA Log | Python, Computing cluster | MAE, MAPE, RMSE, Resource Underprovisioning & Overprovisioning, Elastic Speedup |
| Kakade et al. (15) (2023) | Bi-LSTM | Predict future demands and automatically scale containers accordingly | Generated custom dataset | Computing cluster | MAE, RMSE, Latency |
| Theodoropoulos et al. (26) (2023) | GNN-LSTM | Uses Spatiotemporal GNNs to forecast CPU consumption, and then leverage said forecastings to conduct proactive horizontal autoscaling | Generated custom dataset | Python, CloudSim Plus | MAE, RMSE, Latency, Execution Time, Number of Pod Replicas |
| Dogani et al. (20) (2024) | FedAVG-BiGRU | Design proactive auto-scaling for a multi-step prediction model | FIFA World Cup 98 Web Server | Computing cluster | MAE, MAPE, RMSE, Resource Underprovisioning & Overprovisioning, Elastic Speedup |
| Ahmad et al. (21) (2025) | Prophet | introduce ProSmart HPA, a resource-efficient horizontal pod auto-scaler, which utilizes machine learning for proactive scaling to mitigate resource mismanagement. | FIFA World Cup 98 Web Server | AWS EC2 instances | Resource Underutilization, Resource Underprovisioning & Overprovisioning, Elastic Speedup |
| **Proposed: GraphOpticon (2025)** | GNN-LSTM, Information Fusion & Distillation Algorithms | Utilizes Spatiotemporal GNNs and information fusion & distillation algorithms to improve service performance and resource consumption | Google cluster traces | Python, CloudSim Plus | RMSE, MAE, Execution Time, Latency, Number of Pod Replicas, Training Time, Inference Time, Number of Parameters |

$c_1, c_2, ..., c_c$, with $c_c$ indicating the $c^{th}$ cluster, where $1 \leq c \leq |C|$. In the context of clusters, pods are the fundamental units that encapsulate and run one or more containers. They provide isolation, resource management, and a consistent deployment model across different clusters in container orchestration systems like Kubernetes. A pod is the smallest deployable unit in Kubernetes and can host one or more containers. Pods provide a layer of abstraction and encapsulation for its containers. These pods are denoted by the set $P = p_1, p_2, ..., p_p$, with $p_p$ indicating the $p^{th}$ pod, where $1 \leq p \leq |P|$. Furthermore, they are used to host services of various types. These types of services are denoted by the set $S = s_1, s_2, ..., s_s$, with $s_s$ indicating the $s^{th}$ type of service, where $1 \leq s \leq |S|$. Much like the service types mentioned above, each pod is intertwined with a specific replication time that highly depends on the service this pod hosts. The replication time for pods refers to the duration it takes for a pod in a container orchestration system (such as Kubernetes) to be replicated. This process involves pulling container images, setting up the environment, and initializing the application. Faster start-up times are desirable for efficient scaling and responsiveness in dynamic environments. These replication times are denoted by the set $T = t_1, t_2, ..., t_t$, with $t_t$ indicating the $t^{th}$ replication time, where $1 \leq t \leq |T|$. Each pod $p$ is associated with a service type $S$ and a replication time $T$.

**Table 2**
Evaluation metrics used in contemporary works on proactive horizontal autoscaling.

| Evaluation Metrics | Descriptions |
|---|---|
| Mean Absolute Error (MAE) | Average magnitude of errors between predicted and actual values |
| Mean Elasticity Index (MEI) | Ratio of the minimum to the maximum of actual vs. predicted resources, averaged over time |
| Root Mean Squared Error (RMSE) | Evaluates the standard deviation of the prediction errors |
| Mean Absolute Percentage Error (MAPE) | Average percentage error between predicted and actual values |
| R-squared ($R^2$) | Square of the multiple correlation coefficient between the observed outcomes and the predicted values |
| Mean Squared Error (MSE) | Average squared difference between the forecast and the observed values |
| Availability | Percentage of successfully processed requests out of the total user requests issued. |
| Training time | Time required to train the model on a given dataset |
| Inference time | Time required for a trained model to make predictions on unseen data. |
| Number of parameters | Total count of trainable weights and biases in the model |
| Resource underprovisioning | Resources that a microservice needs but is unavailable. |
| Resource overutilization | Resource utilization exceeding a predefined threshold value. |
| Resource overprovisioning | The residual resource not utilized by a microservice |
| Elastic speedup | The rate at which a system dynamically adjusts resources to meet workload demands |
| Workload Resource Consumption | Resource consumption caused by the running application workloads |
| Autoscaling System Resource Consumption | Resource consumption caused by the autoscaler |

Finally, each pod exhibits a certain type of resource consumption. In the frame of this work, resource consumption corresponds to the ongoing percentage of CPU utilization, as it is the most widely applied resource metric for CPU-intensive applications in Kubernetes (29). The ongoing resource consumption at pod $p$ at time $t$ is denoted as $R_t^p$. Table 3 summarizes the notations used in this work. Our work aims to introduce an advanced forecasting model that, through information refinement & fusion, is capable of producing more accurate resource consumption predictions regarding multiple pods.

In time-series analysis, the multi-step formulation involves predicting future values of a time series by forecasting multiple time steps ahead. This approach contrasts with the single-step approach, which only estimates the next point in time. In the present challenge's context, the output vector's dimensional space is denoted as $R^{|P|*M}$, where $|P|$ represents the number of pods for which traffic predictions are intended at time point $t$, and $M$ represents the number of future steps for these projections. Similarly, the input vector's dimensional space is defined as $R^{|P^{'}|*M^{'}}$, with $|P^{'}|$ corresponding to pods whose resource consumption variations depend on those of $P$, and $M^{'}$ indicating the number of preceding time steps contributing to the retrospective observation window (look-back window). It's essential to note that in the frame of this work, the value of $M^{'}$ is equivalent to that of $M$.

To delve further into our analysis, we concentrate on a specific time point $t_i$ and define the input vector $X$ as:

$$X = \{x_{i-M'+1}, ..., x_{i-z'}, ..., x_i\}, z' \in M', \tag{1}$$

where $x_{i-z'} = R^1_{t_{i-z'}}, R^2_{t_{i-z'}}, ..., R^{|P^{'}|}_{t_{i-z'}}$ represents the resource consumption of each pod $p \in P^{'}$ at time $t_{i-z'}$. Similarly, we model the output vector $Y$ as:

$$Y = \{y_{i+1}, ..., y_{i+z}, ..., y_M\}, z \in M, \tag{2}$$

where $y_{i+z} = R^1_{t_{i+z}}, R^2_{t_{i+z}}, ..., R^{|P|}_{t_{i+z}}$ represents resource consumption at pod $p \in P$ at time $t_{i+z}$.

As our proposed solution is based on the information fusion & distillation properties of graph neural networks, it is crucial to transform the previous problem formulation into a graph format. There are various graph types, with a significant distinction being whether the considered graph structures are static or dynamic. Dynamic graphs can be categorized into Discrete-Time Dynamic Graphs (DTDG) (13) and Continuous-Time Dynamic Graphs (CTDG) (30). This work adopts the DTDG approach to represent resource consumption across pods dynamically, since its ability to capture spatiotemporal dependencies in the frame of resource consumption forecasting has been documented in the corresponding scientific literature, as discussed previously in the Related Work section of this work. In the DTDG paradigm, a dynamic graph is defined as a sequence of snapshots of a static graph, each corresponding to a specific time-step $t$, with the duration between consecutive time-steps termed as $t_{window}$. These snapshots create a temporal continuum, enabling the emergence of temporal patterns.

Each static graph consists of multiple nodes and edges representing spatial relations. In our context, each graph corresponds to a computational infrastructure that consists of $|P|$ Pods that exhibit a certain resource consumption behavior at each time-step $t$.

Given an undirected graph $\overline{G}$ with $|P|$ nodes and $E$ edges, nodes correspond to pods, and edges represent the correlations in resource consumption that emerge within the context of the various pods. This graph can be described by a weighted Adjacency Matrix $\mathbf{A} \in \mathbb{R}^{|P|\times|P|}$ incorporating edge weights $w_{ij}$ and a Feature Matrix $\mathbf{F} \in \mathbb{R}^{|P|\times V}$, where $V$ is the dimension of each feature vector.

The Feature Matrix represents the collective of Feature Vectors. Each of the $|P|$ rows in the Feature Matrix corresponds to a Feature Vector describing node-specific attributes. In this study, graph node (pod) is characterized by a Feature Vector with a dimension equal to $M^{'}$, representing resource consumption values recorded at the respective pod over the last $M^{'}$ time intervals, hence $V = M^{'}$. Moreover,

**Table 3**
Notations used in this paper.

| Notations | Descriptions |
|---|---|
| $\mathbb{R}^n$ | n-dimensional euclidean space |
| $C$ | Set of clusters |
| $P$ | Set of pods |
| $R_t^p$ | Resource consumption of $Pod_p$ at time-step $t$ |
| $S$ | Set of services |
| $S_p$ | Type of service deployed in $Pod_p$ |
| $T_p$ | Replication time for $Pod_p$ |
| $M$ | Number of input time-steps |
| $M'$ | Number of prediction time-steps |
| $t_{window}$ | Duration between consecutive time-steps |
| $V$ | Dimension of Feature Vector |
| $F$ | Feature Matrix |
| $X$ | Input Matrix |
| $\overline{G}$ | Undirected Graph |
| $A$ | Adjacency matrix with edge weights $w_{ij}$ |
| $I$ | Identity Matrix |
| $W$ | Learnable weight matrix |
| $\|E\|$ | The number of elements in a given set $E$ |

each of the $M^{'}$ columns in the Feature Matrix $F$ represents a distinct time interval $t$ within the input sequence. This setup allows instances of the Feature Matrix to be treated as time-series data. The Adjacency Matrix $A$ remains constant, representing the resource consumption correlation among the various Pods. Conversely, the Feature Matrix $F$ is dynamic and varies for each time interval $t$. Consequently, snapshots of the Feature Matrix $F$ are conceptually analogous to the aforementioned input vector $X \in \mathbb{R}^{|P| \times M^{'}}$.

## 5. GraphOpticon

This work aims to establish a proactive horizontal autoscaling solution that is capable of simultaneously achieving better service performance and reduced resource consumption. The proposed solution operates based on accurate resource consumption predictions that involve numerous pods across numerous time steps. To that end, the authors of this work propose GraphOpticon, a proactive horizontal autoscaling solution that leverages the information fusion & distillation properties of graph neural networks. GraphOpticon consists of 4 components. These components are:

- **Monitoring** Component
- **Input Construction** Component
- **Forecasting** Component
- **Output Distillation** Component

These components are designed to operate as parts of a singular pipeline. This pipeline is depicted in Fig. 1.

### 5.1. Monitoring Component

This component is built upon the functionalities of modern monitoring frameworks, such as Prometheus. It is designed to perform three distinct operations related to information retrieval. First, it periodically scrapes CPU utilization values for each pod at every time-step $t$ to construct the Input Matrix $X$. The second operations involve the construction of the weighted Adjacency Matrix $A$. While the Monitoring Component does not directly compute the weights of $A$, it is responsible for gathering the necessary data and transmitting it to the Input Construction Component, which performs the weight calculations. Towards assisting in the construction of $A$, the Monitoring Component retrieves information about the cluster $C$ each pod $p$ belongs to and the service $S$ it hosts. This process occurs once during the initialization of GraphOpticon. The third and final operation of the Monitoring Component involves conveying the estimated replication times $T$ of all examined services to the Output Distillation Component.

### 5.2. Input Construction Component

This component uses the name of the cluster $C$ that a pod $p$ belongs to, along with the name of the service $S$ the pod hosts, to create the Adjacency Matrix $A$. The aforementioned input information is provided by the Monitoring Component. Graph edges are crucial for illustrating the relationships among pods and determining which nodes shall be involved in the feature aggregation process that shall be performed by the encoder part of the Forecasting Component. The encoder part of the Forecasting Component is based on Graph Convolutional Networks (GCNs) (31). Feature aggregation, in the frame of GCNs, is the process of combining node features from a node's neighbors (and sometimes itself) using weighted summation or averaging to capture local graph structure and propagate information. In this paper, we aim to leverage the relations that are present among the various pods in order to construct the corresponding Adjacency Matrix $A$. According to this approach, the Adjacency Matrix $A$ is formed based on the service $S$ and cluster $C$ similarities between each pair of pods (nodes). The algorithm for calculating the corresponding weights is detailed in Algorithm 1. If two pods do not facilitate the same type of service $S_p$, then the corresponding weight associated with edge $e_{ij}$ equals zero. If two Pods facilitate the same type of service $S_p$, but they do not belong to the same cluster $C$, then the corresponding weight associated with edge $w_{ij}$ equals 0.5. Finally, if two Pods facilitate the same type of service $S_p$, and they belong to the same cluster $C$, then the corresponding weight associated with edge $e_{ij}$ equals 1.

An example of this approach is showcased in Figure 2. This figure depicts 4 clusters (1,2,3,4) with 16 pods that host 4 different types of services (red, green, blue, yellow). According to the proposed approach, pods that belong to the same cluster $C$ and host the same type of service $S$ (inside black rectangles) shall have edge weights $w$ equal to 1. Pods that host the same service $S$ and belong to different clusters

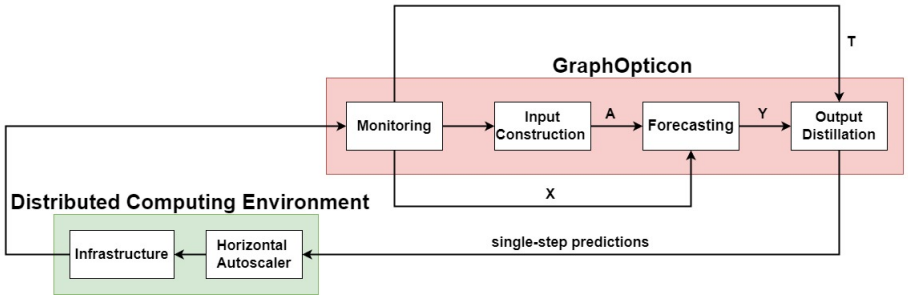


**Figure 1:** The pipeline of the proposed solution.

$C$ (connected by a colored edge) shall have edge weights $w$ equal to 0. The rest shall have edge weights equal to 0.

The underlying rationale for leveraging the Input Construction Component is to streamline the feature aggregation process by restricting it to pods hosting the same service type. This simplifies the model's complexity by focusing on distilled spatial correlations inherent in the input structure. Moreover, the forecasting model can capture more refined dependencies by introducing the weight differentiation between pods that inhabit the same cluster and pods that do not. The proposed weight assignment derives from the assumption that pods that host the same type of service are expected to present more similarities in terms of their resource (CPU) consumption patterns when compared with pods that do not host the same type of service. Furthermore, pods that host the same the same type of service and belong to the same cluster are expected to present more similarities in terms of their resource (CPU) consumption patterns when compared with pods that do host the same type of service but do not belong to the same cluster. This assumption serves as an extension to our line of thought that was presented in Section 3.

**Algorithm 1** Input Construction Algorithm.

**Input:** The $|P|$ Pods, alongside their corresponding $C_p$ and $S_p$ attributes, which describe the cluster that each pod belongs to and the type of service each pod facilitates, respectively.
**Output:** The weighted Adjacency Matrix $A$.
**Begin algorithm**
**1.** For each pair of pods $i, j$ in $P$:
**2.** If $S_i \neq S_j$:
**3.** Then $A_i, j \leftarrow 0$.
**4.** If $(S_i = S_j) \wedge (C_i \neq C_j)$:
**5.** Then $A_i, j \leftarrow 0.5$.
**6.** If $(S_i = S_j) \wedge (C_i = C_j)$ :
**7.** Then $A_i, j \leftarrow 1$.
**8.** Return $A$
**End algorithm**

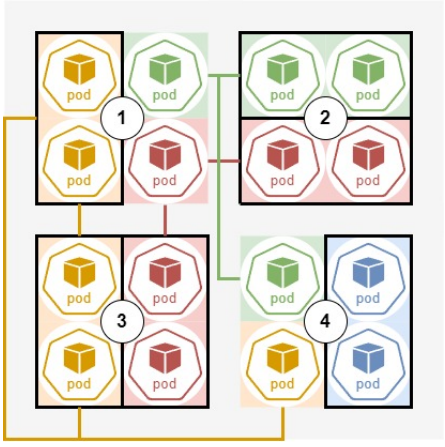


**Figure 2:** Input construction process.

### 5.3. Forecasting Component

We have combined weighted Graph Convolutional Network (GCN) (31) and LSTM layers to construct an encoder-decoder architecture capable of predicting resource consumption. Encoder-decoder architectures for time-series forecasting involve an encoder that processes input sequences into a fixed representation and a decoder that uses this representation to predict future sequences. The weighted GCNs serve as the encoder to extract structural features from the input sequence to generate a consolidated representation. This operation is conducted as follows:

$$H^{\text{encoder}} = \text{Weighted GCN}_{\text{encoder}}(X, A) \tag{3}$$

Here, $h^{\text{encoder}}$ denotes the consolidated representation post the application of weighted stacked graph convolution, where $A$ signifies the weighted Adjacency Matrix of the graph, and $X$ indicates the Feature Matrix, which is the input of the forecasting entity. The weighted Adjacency Matrix $A$ is provided by the Input Construction Component, while the Feature Matrix $X$ is provided by the Monitoring Component.

Subsequently, the constructed representation is channeled into the LSTM component of the model, enabling the capture of temporal patterns at the level of graph snapshots. Functioning as a decoder, the LSTM component generates the desired forecasts through the following process:

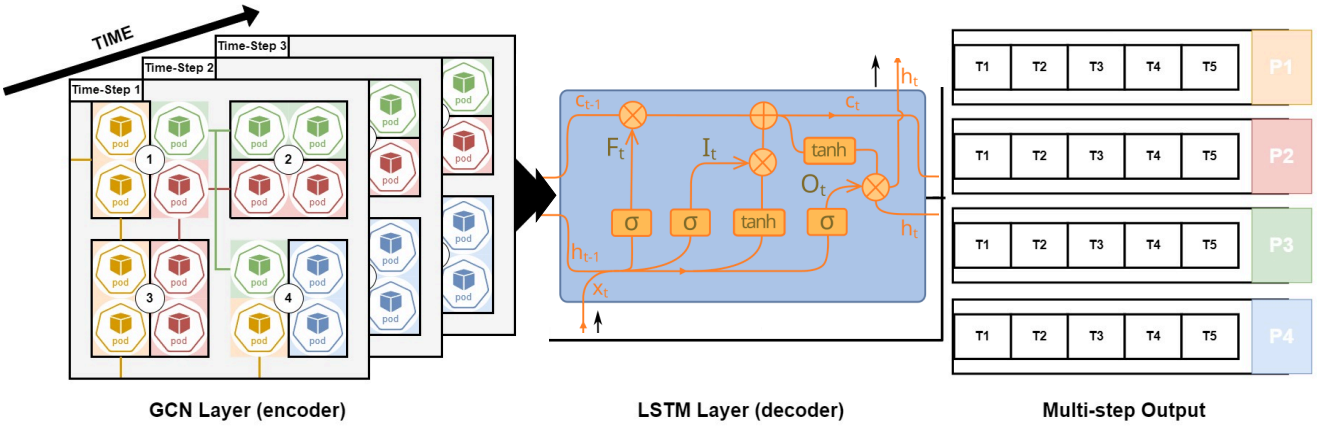


**Figure 3:** Architecture of the Forecasting Component.

$$Y = \text{LSTM}_{\text{decoder}}(H^{\text{encoder}}) \tag{4}$$

The term $\text{LSTM}_{\text{decoder}}$ denotes the LSTM network, which accepts the aggregated representation from the encoder to produce an output that is then passed through a dense layer for multi-step prediction generation. In the context of multi-step time-series prediction, the Forecasting Component takes a sequence of graph signals as input, where each signal represents a different time-step and is depicted as a graph signal on a consistent graph. The objective is to forecast future time-series values based on the graph signals from preceding time steps.

### 5.3.1. *Encoder (Weighted GCN Layer)*

The encoder part of the Forecasting Component is based on the use of weighted Graph Convolutional Networks (GCNs). Weighted GCNs are a powerful framework for learning representations of nodes in graphs, considering both the graph structure and features associated with nodes. Let $\mathbf{X}$ denote the feature matrix of size $|P| \times M$, where $N$ is the number of nodes and $M$ is the number of features per node. The weighted adjacency matrix $\mathbf{A}$ represents the connections between nodes in the graph. The propagation rule in a weighted GCN is defined as:

$$\mathbf{H}^{(l+1)} = \sigma\left(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}\right) \tag{5}$$

Where:

- $\mathbf{H}^{(l)}$ is the feature representation matrix at layer $l$,
- $\tilde{\mathbf{A}}$ is the weighted adjacency matrix with added self-connections,
- $\tilde{\mathbf{D}}$ is the degree matrix of $\tilde{\mathbf{A}}$,
- $\mathbf{W}^{(l)}$ is the weight matrix of layer $l$,
- $\sigma$ is the ReLU activation function.

The input feature matrix $H^{(0)}$ is typically initialized as $X$, which is the input sequence provided by the Monitoring Component. Furthermore, in the frame of this work, $l = 1$.

### 5.3.2. *Decoder (LSTM Layer)*

The decoder part of the Forecasting Component is based on Long Short-Term Memory (LSTM) networks. LSTM networks employ the Hidden State mechanism to capture dynamic temporal patterns. What sets LSTM networks apart is their utilization of the Cell State structure, introducing Cell State manipulation through regulatory mechanisms known as Gates. Each LSTM node comprises three gate-related elements, all incorporating sigmoid layers to ensure differentiation within the range of $0 - to - 1$. The sigmoid activation function scales values to facilitate the assessment of data importance and decision-making regarding retention or omission. Gate structures include two sets of weight matrices, labeled $W$ and $U$, associated with Hidden State and input and additional matrices for Cell State. The input $X_t$ corresponds to timestamp $t$. Gates utilize these matrices, input, and prior Hidden State ($hidden_{t-1}$).

The Forget Gate determines which historical information from past timestamps to exclude from the Cell State. Its output is computed using Eq. 6. The Input Gate assesses the significance of recent input, updating the Cell State using Eq. 7. Cell State calculation employs the $\overline{\text{C}}$ vector, generated as per Eq. 8, with the tanh activation function mitigating gradient issues. The Cell State update process is described in Eq. 9, combining the output of the Forget Gate and the Input Gate with $\overline{\text{C}}$. The Output Gate computes the subsequent hidden state using Eq. 10. The new Hidden State is calculated according to Eq. 11. Updated Cell State

and Hidden State are then propagated to subsequent LSTM nodes for the next time-step (32).

$$\text{forget}_t = \text{sigmoid}(X_t \cdot W_f + \text{hidden}_{t-1} \cdot U_f) \quad (6)$$

$$\text{input}_t = \text{sigmoid}(X_t \cdot W_i + \text{hidden}_{t-1} \cdot U_i) \quad (7)$$

$$\overline{C} = \tanh(X_t \cdot W_c + \text{hidden}_{t-1} \cdot U_c) \quad (8)$$

$$C_t = \text{forget}_t \cdot C_{t-1} + \text{input}_t \cdot \overline{C}_t \quad (9)$$

$$\text{output}_t = \text{sigmoid}(X_t \cdot W_o + \text{hidden}_{t-1} \cdot U_o) \quad (10)$$

$$\text{hidden}_t = \text{output}_t \cdot \tanh(C_t) \quad (11)$$

In the frame of the decoder, $output_t$ is passed through a dense layer to construct the desired 2-dimensional output $Y$ shape.

### 5.4. Output Distillation Component

The Forecasting Component produces a multi-step output $Y$ corresponding to $P$ pods. The Output Distillation Component is designed to receive the multi-step prediction $Y$ generated by the Forecasting Component as input, along with the replication time $T$ corresponding to each pod $p$. Upon receiving these inputs, the Output Distillation Component determines, based on each pod, which step from the multi-step prediction should be retained to produce a single-step prediction. This functionality is conducted using a dedicated list ($N$), which associates each replication time $T$ with a corresponding time-step of the multi-step prediction. Calculating $N$ is showcased using Algorithm 2.

This decision is guided by the necessity of selecting a time step further into the future than the anticipated completion time of the deployment process. Additionally, it prioritizes a time step closer to the expected completion moment of the deployment process, as looking too far ahead compromises prediction accuracy. This process tailors predictions to each pod. Figure 4 depicts an example of how the proposed component selects the prediction step to retain.

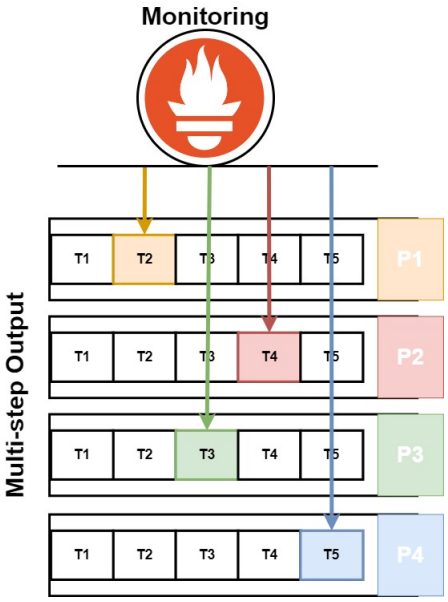


**Figure 4:** Output distillation process.

**Algorithm 2** Output Distillation Algorithm

**Input:** List of replication times $T = [t_1, t_2, \ldots, t_{|P|}]$, and list of prediction time-steps $M = [m_1, m_2, \ldots, m_{|M|}]$. The temporal difference between each time step equals $t_{window}$.
**Output:** List of singular time-steps $N$ corresponding to each replication time $T$.
**Begin algorithm**
**1.**Sort $M$ in ascending order
**2.**Initialize $N$ as an empty list
**3.**For each $t$ in $T$:
**4.** Initialize $next\_step \leftarrow None$
**5.** For each $m$ in $M$:
**6.** If $m > t$:
**7.** Then $next\_step \leftarrow m$
**8.** **break**
**9.** Append $next\_step$ to $N$
**10.**Return $N$
**End algorithm**

## 6. System Implementation

Before proceeding to the experimental evaluation section of this work, it is of paramount importance to address any potential limitations that the architecture of GraphOpticon may present. These potential limitations involve aspects such as the compatibility of GraphOpticon with contemporary frameworks, as well as its scalability, and the underlying formalism that is used in the frame of this work. This section is dedicated to addressing these limitations.

### 6.1. Integration

The aforementioned components that were described in the previous section of this work are designed to be integrated into cloud and edge computing frameworks such as Kubernetes or K3S[2], inheriting foundational orchestration and management mechanisms. GraphOpticon does not seek to replace these established frameworks but aims to enhance them, specifically regarding improved service performance and resource consumption. By limiting the required changes to the scaling mechanisms, GraphOpticon maintains the overall stability and reliability of the existing orchestration frameworks. This strategy minimizes disruption and complexity, allowing for a smoother integration while still achieving the desired improvements. Enhancing this aspect without altering other parts of the framework ensures that GraphOpticon can offer performance improvements without the need for major changes or overhauls, thus preserving the integrity and usability of the underlying orchestration system.

The same principles have also been applied in regard to its compatibility with the leveraged monitoring framework.

[2]https://k3s.io/

The Monitoring Component is designed in a manner that enables it to seamlessly integrate with the Prometheus[3] monitoring system. Its primary functions include monitoring application components, whether deployed as pods or virtual machines, and monitoring hosts, whether physical or virtual, on Kubernetes or K3s clusters located in the cloud or at the edge. For our case, it is necessary to retrieve the CPU utilization of pods. In the Monitoring Component's setup, Prometheus periodically pulls metrics from monitoring agents named Prometheus exporters. Particularly for pod monitoring, the Monitoring Component utilizes kube-state-metrics[4] exporter to perform pod monitoring. This exporter provides metrics by interfacing with the Kubernetes API server and incorporating data from both Kubelet and cAdvisor. The cAdvisor exporter collects resource usage statistics, including CPU utilization, for all running pods. Kubelet is an agent on each node in a Kubernetes cluster, working with the Kubernetes API server to collect information on node events, pod statuses, and resource usage.

The leveraged architectural design (33) enables the Monitoring Component to detect the replication time of pods by utilizing the kube-state-metrics exporter and its above-mentioned capability to monitor pod statuses. The detection of a running pod produces an alert that includes the deployment time as a timestamp, the number of replicas, the namespace to which the pod belongs, the host node, node IP, pod IP, pod status, external IPs, and the applied pod labels. The above alert was configured to meet our case requirements by including the master node IP indicating the cluster's IP and a master's label indicating the cluster name. This feature can match pods with the cluster they are running in. This addition was also achieved based on metrics retrieved by kube-state-metrics. A naming convention is essential to accurately identify the name of the service provided by a pod. Specifically, the pod name should be associated with the service name it provides, ensuring that the monitoring mechanism generates meaningful alerts for the entire system.

## 6.2. Scalability Analysis

The centralized architecture of GraphOpticon raises several scalability concerns. Chief among them is the potential for performance bottlenecks, as the autoscaler must process vast amounts of data from numerous pods, which could lead to increased latency and delayed scaling decisions. There is the risk that as the system scales, the overhead associated with data collection and processing could become a significant drain on resources, further exacerbating performance issues. Calculating the complexity of the proposed autoscaler provides a structured method to address these concerns. By analyzing the corresponding complexity, one can quantify how the autoscaler's performance degrades as the number of pods and metrics increases. This quantitative insight allows for the identification of potential bottlenecks and critical thresholds, guiding the design towards more efficient algorithms and data handling methods.

Towards examining the scalability of the proposed solution, we analyze the computational complexity of the data collection and processing algorithms that are leveraged in the frame of GraphOpticon. These include the Monitoring process, the Input Construction process, and the Output Distillation process. Since scalability is investigated in relation to the number of pods, complexity shall be calculated on the basis of the number of pods $P$. In order to construct Input Matrix $X$, the Monitoring component periodically retrieves CPU utilization metrics from the Metrics Server or an external monitoring system. Given that the Monitoring component queries metrics for each running pod individually, the time complexity of this operation is $O(P)$, where $P$ is the number of pods being monitored. The same complexity applies to the retrieval of the required information that is later used by the Input Construction Component to create the Adjacency Matrix $A$.

The Input Construction Component is responsible for generating the Adjacency Matrix $A$ based on relationships between pods. This process involves iterating over all $P$ pods and performing pairwise comparisons to establish edge weights. In the worst case, each pod is compared against every other pod, leading to a complexity of $O(P^2)$. The quadratic growth arises due to the necessity of examining all potential connections within the system. However, it is of paramount importance to point out that this process is fully carried out only when GraphOpticon is instantiated.

The Output Distillation component involves selecting the most relevant prediction step for each pod. Given $M$ sorted prediction time-steps, the nearest step can be identified using a binary search operation with complexity $O(\log|M|)$. Since this operation is performed for each of the $P$ pods, the total complexity is given by $O((P + |M|)\log|M|)$. Based on these results, the overall complexity of GraphOpticon shall be equal to $O(P^2)$ for its instantiation and equal to $O((P+|M|)\log|M|)$ during its operation. In other words, when the Adjacency Matrix $A$ is calculated, the complexity is dominated by the quadratic term, yielding $O(P^2)$. However, after the initial matrix is constructed, the subsequent complexity is equal to $O((P + |M|)\log|M|)$. Since we expect that $M < P$, the term $P+|M|$ is dominated by $P$. So it simplifies to $O(P)$. As a result, the complexity becomes approximately $O(P\log|M|)$. In this case, $log|M|$ grows slower than $P$, and the overall complexity will still be influenced by $P$, with a logarithmic factor that does not change the linear dependence on $P$. So, this complexity can be characterized as linear with a logarithmic correction. Thus, it is safe to conclude that GraphOpticon constitutes a scalable solution, the complexity of which grows in an almost linear manner as the number of pods increases.

## 6.3. Formalism

In the frame of Kubernetes' autoscaling operation, the number of pods dynamically increases and decreases. As a result, the size and structure of the Adjacency Matrix $A$

[3] https://prometheus.io/

[4] https://github.com/kubernetes/kube-state-metrics

would constantly change over time. However, this would contradict the CTDG paradigm on which GraphOpticon is based. Towards resolving this issue, in the frame of this work, we consider that the Adjacency Matrix $A$ is constructed on the basis of the maximum number of pods that can be deployed for each type of service, given the resource limitations of the underlying computational infrastructure, instead of simply considering just the pods that are deployed during the instantiation phase of GraphOpticon. As such, the size of the Adjacency Matrix $A$ (in terms of graph nodes) is bound to always be greater or equal to the number of deployed pods. Furthermore, when a pod is not currently operating, its corresponding Feature Matrix $X$ values should be equal to 0 since its CPU consumption is equal to 0. In the frame of GCNs, a zero feature vector value means the node is present but inactive in feature propagation. So, even if the structural graph connectivity is still intact (edges), a zero feature vector value has no effect on its neighbors in terms of information contribution during the feature aggregation process. In other words, in the context of information contribution, a zero feature vector value has the same effect as the removal of the corresponding edge would have. As a result, the information contribution for a node shall only involve nodes that have both an edge formed with each other and nonzero feature vector values (corresponding to pods that are currently deployed). Subsequently, the forecasting accuracy of GraphOpticon is not affected at all by this design choice.

## 7. Experimental Evaluation

To evaluate the efficiency of GraphOpticon, we conducted a large-scale experiment in a simulated distributed computing environment using the CloudSim Plus[5](34) framework on a machine that utilizes an AMD Ryzen 9 4900HS processor, and 16GBs of RAM. Given the modular and extendable nature of this simulation framework, it constitutes a widely adopted approach to conducting refined distributed computing simulations (35). The simulated distributed computing environment consists of 20 compute nodes that process tasks assigned by a dedicated scheduling algorithm. There are 5 compute nodes available for task processing at all times and 15 additional ones that can be allocated based on resource demand via horizontal autoscaling. In the frame of the experimental evaluation of this work, each compute node corresponds to a pod.

Across all scaling solutions, the scaling decision is produced once every 1 second. Furthermore, the duration between each time-step $t_{window}$ equals 1 second. In the frame of this experiment, there are 5 distinct types of services whose replication times $T$ are equal to 1, 3, 5, 7, and 9 seconds. To incorporate various types of service scenarios, the task generation patterns for 3 of these services were modeled as work-related services, and the other 2 services were modeled as leisure-time services. This type of modeling is geared towards establishing distinct temporal patterns in terms of task creation rates, depending on the time of day, as well as the day of the week. For instance, leisure-time-related services are designed to peak in terms of task creation after work hours and during the weekends. On the other hand, work-related services are designed to peak in terms of task creation rate during work hours. Such a peak is depicted in Fig. 5, which associates task production rates of a work-related service with the CPU usage of a compute node that is dedicated to handling tasks that correspond to this service. As one can see, task production rates (and the corresponding CPU) usage peaks around $12:00$.

For each type of service, a maximum of 4 compute nodes is allocated to handle its corresponding tasks. Tasks produced during the simulation are assigned to processing nodes based on a scheduling algorithm. The combination of scheduling policies and resource allocation strategies can greatly affect resource & energy consumption, operational costs, and service quality (36). Three task-scheduling algorithms were examined to ensure that GraphOpticon outperforms its competitors across various scenarios. The algorithms include Round-Robin, MinMin, and MaxMin.

Round-robin is a preemptive scheduling method where tasks are assigned to computational resources in a rotating sequence without considering specific task or node characteristics. MinMin prioritizes short-length tasks by identifying the task with the fewest Million Instructions (MI) in two phases. First, it selects the task with the shortest length, and second, it assigns this task to the processing node that can execute it in the least amount of time. This process repeats for each task to minimize overall execution time. MaxMin, on the other hand, prioritizes long-duration tasks. It begins by identifying the task with the longest duration in terms of MI and then allocates it to the processing node that can complete it in the shortest time possible. This algorithm considers the computational load and capabilities of tasks and processing nodes to ensure efficient task allocation.

Finally, to ensure experimental integrity, the allocation of each pod to a specific cluster $C$ was performed in a randomized manner. There are 4 available clusters. During each second, numerous tasks are generated and sent to the available compute nodes for processing. Given that service demand and resource consumption are influenced by periodic phenomena occurring over hours, days, and even weeks, the experiment was designed to span a 28-day period and to include more than 8 million tasks that were generated based on multiple Poisson probability distributions and statistical properties that were derived from the Google Cluster dataset[6](37; 38), to capture those patterns, using statistical techniques. The incorporation of Google Cluster traces into simulation frameworks (39), as well as the use of statistical techniques to improve the modeling capabilities of these frameworks, constitute practices found in the corresponding scientific literature (40). More specifically, in accordance with the Google Cluster dataset, the task arrival rate can be modeled using a Poisson process, which captures the burstiness of frequent, short tasks and idle periods when task

[5] https://cloudsimplus.org/

[6] https://github.com/google/cluster-data

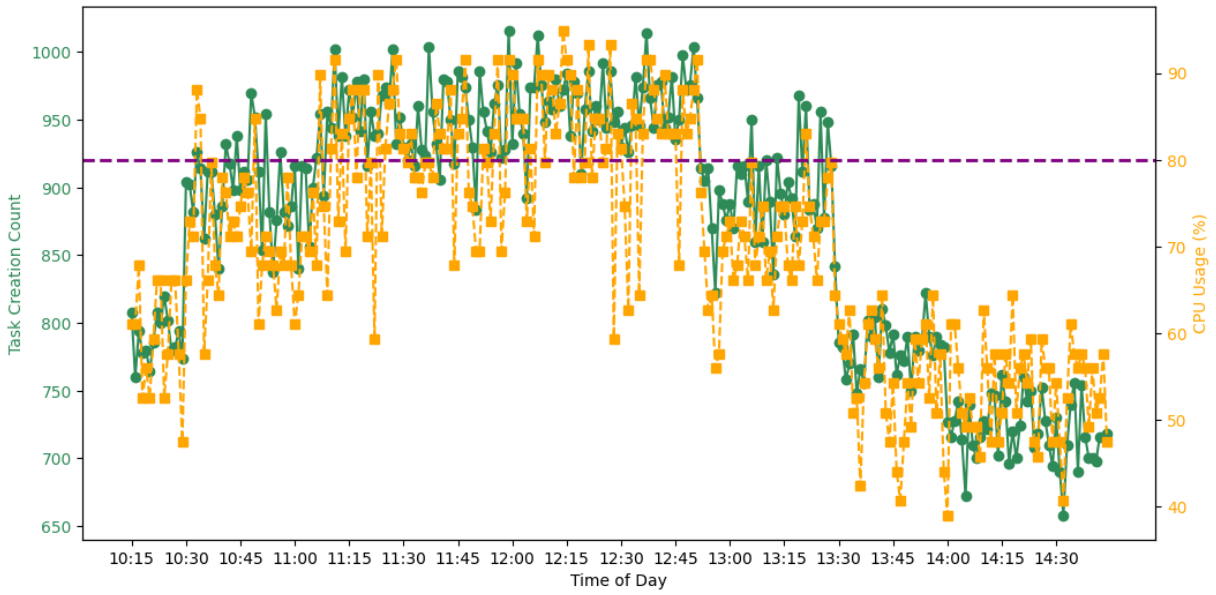


**Figure 5:** Number of tasks and the corresponding CPU usage.

demand is low. The dataset is dominated by light tasks (short, low-resource-consuming) but includes a smaller number of heavy tasks (long-running, resource-intensive), following a heavy-tailed distribution. Additionally, peak and off-peak trends in workload intensity, such as hourly or daily fluctuations, can be identified to optimize resource management and system performance during varying task loads.. Aside from the various periodic phenomena, resource consumption is heavily influenced by sudden bursts in resource demand. To simulate the sudden bursts in resource demand, the number of tasks created at each time-step $t$, is dynamically calculated using a combination of the process mentioned above that is capable of simulating periodic phenomena and a function that generates random numbers that are within an acceptable range considering $t$. Furthermore, tasks were categorized into three sizes based on Millions of Instructions Per Second (MIPS): 0.5, 1, and 1.5 MI, with each compute node handling 1 MIPS. Out of the 8 million tasks in total, about 4 million were 0.5 MIPS, 3 million 1 MIPS, and 1 million 1.5 MIPS. The type of each task created at any given moment was random, potentially overwhelming compute nodes during bursts of lengthy tasks. Thus, we were able to evaluate the efficiency of the proposed approach in handling both sudden bursts and periodic phenomena (as depicted in 5).

### 7.1. Forecasting Models

The evaluation of the proposed solution consists of two parts. In the first part, we compare the forecasting accuracy of the proposed GraphOpticon (a singular model for all pods) and the Hybrid LSTM Encoder-Decoder (ranging from using only a singular model for all pods to having a dedicated model for each pod). As was previously discussed in the Related Work section of this work, the Hybrid LSTM Encoder-Decoder (6) has shown promising results in the field of multi-step service demand forecasting, surpassing many contemporary forecasting solutions. Aside from its forecasting accuracy, it was selected due to the fact that this model and GraphOpticon both constitute DL Encoder-Decoder solutions, and thus, it would be highly appropriate to compare them.

Towards evaluating the accuracy of various forecasting solutions, we utilized the traces derived from the aforementioned large-scale simulation, employing the Standard scaling approach. The Standard reactive horizontal autoscaling approach requests allocating an additional compute node when CPU consumption exceeds the 80% mark and deallocates a compute node when its CPU consumption falls below 20% These traces encapsulate the CPU consumption for each of the 20 pods throughout the duration of the experiment. Using these traces, we performed multi-step CPU consumption forecasting based on 7 scenarios. The first 6 scenarios explored different configurations of the Hybrid LSTM Encoder-Decoder. In scenario 1, 20 models were used, each predicting CPU consumption for a single pod. Scenario 2 involved 10 models, each predicting CPU consumption for 2 pods, while scenario 3 used 5 models, each covering 4 pods. Scenario 4 employed 4 models to predict CPU consumption for 5 pods each, and scenario 5 utilized 2 models, each predicting for 10 pods. Finally, scenario 6 used a single Hybrid LSTM Encoder-Decoder model to predict CPU consumption across all 20 pods. Finally, scenario 7 employed GraphOpticon, which is designed to predict CPU consumption for all 20 nodes.

Scenarios $(1-6)$ serve a dual purpose. On the one hand, they are used to identify the optimal configuration of the Hybrid LSTM Encoder-Decoder, which is then compared against GraphOpticon. On the other hand, these scenarios represent different layers of information fusion and deep

learning model specialization. Scenario 1 examines the use of a distinct forecasting model for each pod, thereby operating with a localized context. Scenarios 2 and 3 focus on the service type level. In scenario 3, a single forecasting model is assigned to all pods facilitating the same type of service, whereas in scenario 2, two forecasting models are assigned to all pods facilitating the same type of service. Scenarios 4 and 5 focus on the cluster level. In scenario 4, a single forecasting model is assigned to all pods within the same cluster, whereas in scenario 5, one forecasting model is assigned to all pods spanning a pair of clusters. Finally, scenario 6 explores the use of a single forecasting model for all pods, thereby incorporating a global context. Thus, by comparing the experimental results, one can useful insights into which levels of information fusion are able to provide the best results.

### 7.2. Performance & Resource Consumption

The second part of the experimental evaluation process involves examining the impact of the various horizontal autoscaling approaches (proactive & reactive) on service performance and resource consumption. This part aims to compare the results in terms of execution time, latency, and resource consumption that correspond to four horizontal autoscaling approaches. The first is the aforementioned Standard reactive horizontal autoscaling approach. The second one is the Kubernetes' Horizontal Pod Autoscaler (HPA) approach, set to have the $Target - Metric - Value$ equal to 50% in terms of CPU consumption.HPA automatically scales the number of pods based on the observed resource usage. The HPA controller operates by comparing the target metric value with the current metric value, as expressed in the following equation:

$$\text{Target-Replicas} = \left( \frac{\text{Current-Metric-Value}}{\text{Target-Metric-Value}} \right) \times \text{Current-Replicas} \tag{12}$$

This equation calculates a scaling factor by dividing the Current-Metric-Value by the Target-Metric-Value, which indicates how far the current metric value deviates from the target. This ratio is then multiplied by Current-Replicas to determine the Target-Replicas, which is the desired number of replicas Kubernetes should scale to bring the metric value closer to the target. In practical terms, if the observed metric (CPU utilization) exceeds the target value, the equation will calculate a higher number of Target-Replicas, prompting Kubernetes to scale up the deployment to handle increased load and maintain performance. Conversely, if the metric is below the target, it will calculate a lower number of Target-Replicas, potentially triggering the scaling down to conserve resources. The third one is based on the GraphOption framework that performs proactive horizontal autoscaling based on the produced predictions using the same scaling in and out thresholds as the standard reactive approach. GraphOpticon receives the 10 more recent CPU values as input to construct a singular prediction for each compute node per its associated replication time, while its forecasting horizon is equal to 10 time-steps. Finally, the fourth one is based on the use of the best-performing configuration of the Hybrid LSTM Encoder-Decoder (as shall be decided by the first part of the experimental evaluation) for proactive horizontal autoscaling. Within the frame of this experimental evaluation process, the proactive approach that leverages the best configuration of the Hybrid LSTM Encoder-Decoder shall be referred to as Proactive. Proactive receives the 10 more recent CPU values, while its forecasting horizon is equal to 10 time-steps.

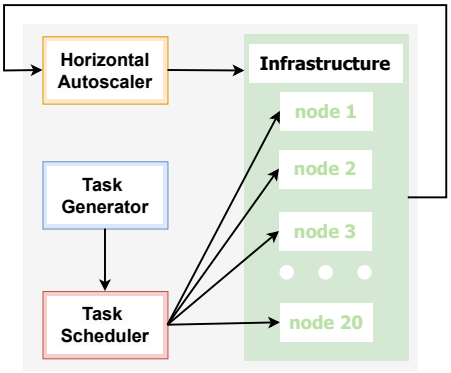


**Figure 6:** Simulation workflow.

A brief overview of the simulation workflow that constitutes the cornerstone of the experimental evaluation process is depicted in Fig. 6. Tasks are generated by a dedicated Task Generator and then sent to the available compute nodes on the basis of the decisions made by the Task Scheduler (Round Robin, MinMin, MaxMin). The dedicated Horizontal Autoscaler (Standard, Kubernetes, Proactive, GraphOpticon) is in charge of adjusting the number of processing nodes in anticipation of workload fluctuations. The Task Generator, all Task Scheduler algorithms, and the reactive Horizontal Autoscaler algorithms (Standard, Kubernetes) were constructed using Java as extensions to the CloudSim Plus simulation framework. The proactive Horizontal Autoscaler algorithms (Proactive, GraphOpticon) required the use of deep learning models that were implemented in Python 3, utilizing libraries such as NumPy, pandas, statistics, Scikit-learn, SciPy, Scikit-Optimize, TensorFlow 2, and its high-level API, Keras.

### 7.3. Evaluation Metrics

To assess forecasting accuracy, we used two metrics. These metrics are the **Mean Absolute Error** (MAE) and the **Root Mean Squared Error** (RMSE). The MAE quantifies the mean of the absolute differences between the predicted and actual values, capturing the average error magnitude. Equation 13 shows the calculation of MAE:

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{13}$$

In contrast, the RMSE evaluates the standard deviation of the prediction errors, giving higher weight to larger discrepancies by squaring them before averaging. The formula for RMSE is shown in Equation 14:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{14}$$

Selecting between MAE and RMSE depends on the prioritization of errors. MAE is generally preferred when equal importance is assigned to all errors, whereas RMSE is useful for penalizing larger errors more heavily. Furthermore, aside from examining the accuracy of the various forecasting solutions, this part is dedicated to exploring aspects such as **Training Time**, **Inference Time**, and the **Number of Parameters** that are associated with each forecasting solution. Long training times increase computational resource demand, consuming significant energy and resources. The larger the model, the longer it generally takes to train, resulting in higher power consumption. Inference, or the process of making predictions with a trained model, also requires resources. Faster models require fewer resources per prediction, while models with longer inference times consume more. Efficiency during inference is crucial for real-time applications due to latency concerns or deployment on resource-limited devices. Training and inference times are calculated in seconds. Models with more parameters are more complex, requiring more memory and computational power for both training and inference. This increases resource usage significantly.

Execution time corresponds to the interval between the beginning and the end of task processing. Some of the most notable metrics used to evaluate latency and execution time include **Average Execution Time**, which offers a comprehensive perspective on service performance. Nevertheless, solely depending on the average might cause one to overlook the intricacies within the latency value distribution. **Median Execution Time** is valuable for assessing the central point of the latency distribution, as it is not sensitive to outliers. A significant difference between the median and the average may indicate outliers disproportionately impacting latency. **Standard Deviation of Execution Time** indicates the spread of execution times, with a higher standard deviation suggesting greater variability in latency. Monitoring standard deviation helps identify consistency or inconsistency in response times, which is crucial for user experience and detecting potential issues in the service infrastructure. **Maximum Execution Time** represents the longest duration for task completion within a system, serving as an upper limit on acceptable execution time. Lower maximum execution time is desirable for timely and responsive performance, especially in real-time or time-sensitive applications. **Range of Execution Time** measures the difference between the shortest and longest execution times, providing insight into the variability and potential extremes in task completion times. A smaller range suggests more consistent performance, while a larger range can indicate fluctuations that might need addressing.

On the other hand, latency corresponds to the interval between task creation and the beginning of its processing. **Skewness of Latency** assesses the asymmetry of latency distribution. A right-skewed distribution indicates that some requests experience significantly longer delays than average, guiding optimizations to mitigate outliers. **Kurtosis of Latency** measures the tails of the distribution, with higher kurtosis suggesting heavy tails and the presence of extreme values. Understanding kurtosis helps anticipate and manage rare but impactful events affecting service latency. **Tail Latency** (98th percentile) focuses on extreme values in latency distribution, identifying the 2% of requests with the longest response times. Monitoring tail latency ensures that even under adverse conditions, a small percentage of users do not experience unacceptably long delays, directly impacting user satisfaction and Service Level Agreements. In combination, these metrics can serve as good indicators of service latency. For instance, high Average or Median Execution Times, coupled with high Maximum Execution Time, Range of Execution Time, Skewness of Latency, and Kurtosis of Latency, may indicate performance issues that need attention. Conversely, a low Average or Median Execution Time, combined with a low Maximum Execution Time, a narrow range of Execution Time, and well-behaved Skewness of Latency and Kurtosis of Latency suggests a more stable and predictable service. By regularly analyzing and interpreting these metrics, service providers can identify areas for improvement and optimize performance accordingly.

Aside from the aforementioned metrics that examine execution time and latency, the authors of this work have also examined the **Resource Consumption** that manifests at each distinct combination of horizontal autoscaling and task scheduling approaches. In the frame of this section, the resource consumption corresponds to the average number of compute nodes used in each examined scenario. Across all examined scenarios, 1 node is always allocated for hosting the Monitoring Component. Furthermore, in the case of the proactive horizontal autoscaling scenarios, 1 additional node is allocated at all times for hosting each of the corresponding forecasting models.

### 7.4. Forecasting Models: Experimental Results & Discussion

Figures 7 and 8 present the experimental results in terms of RMSE and MAE. For the Hybrid LSTM Encoder-Decoder scenarios (1-6), the results are displayed independently for all three task scheduling approaches. In these figures, the blue line corresponds to Round Robin, the orange line to MinMin, and the green line to MaxMin. To reduce visual clutter, for the GraphOpticon scenario (7), we included the average performance across the three task scheduling approaches. Furthermore, although these results correspond to 3 runs, toward ensuring that there is no overlap between

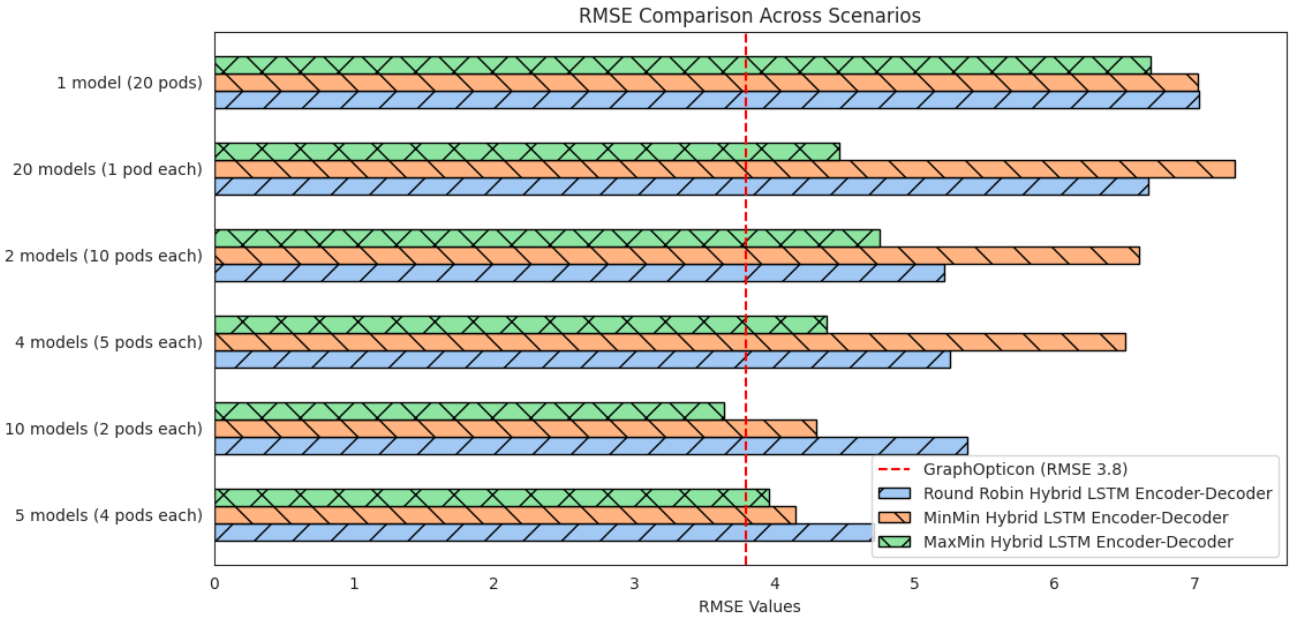


**Figure 7:** RMSE comparison between the Hybrid LSTM ED and GraphOpticon, across various experimental configurations.

competitors, we have chosen to keep the worst-performing run for GraphOpticon, and the best-performing runs for its competitors.

GraphOpticon demonstrates superior performance across almost all explored configurations, with the sole exception being the 10 models (2 pods each) configuration. These experimental results not only highlight GraphOpticon's forecasting accuracy but also offer a wealth of intriguing insights. By analyzing the performance of various configurations, we gain valuable indicators regarding the importance of interrelations among pods. Among the configurations, the 5 models (4 pods each) setup delivers the best results. It is closely followed by the 10 models (2 pods each) configuration. This finding is particularly notable given that the 5 models (4 pods each) setup employs a single forecasting model for each distinct type of service. In contrast, the 10 models (2 pods each) configuration uses two forecasting models per service type. Ranking next in terms of forecasting accuracy are the 4 models (5 pods each) and 2 models (10 pods each) configurations. The 4 models (5 pods each) configuration relies on a single forecasting model for each distinct cluster, while the 2 models (10 pods each) configuration adopts a similar approach but involves two clusters instead of one. Lastly, the poorest results are observed in the 20 models (1 pod each) and 1 model (20 pods) configurations. These results indicate that attempts at performing resource consumption forecasting at a purely local or global level are not as effective as ones that target specific conceptual groups (such as service type level or cluster level).

Considering these results, it can be concluded that, for resource consumption forecasting, it is more efficient to focus primarily on the service type level, followed by the cluster level. This approach is perfectly aligned with the philosophy of GraphOpticon, which considers the relations at the service type level more important compared to the relations at the cluster level. As stated before, according to the Input Construction Component of the GraphOpticon approach, if two pods do not provide the same type of service $S_p$, the weight associated with the edge $e_{ij}$ is set to zero. If two pods provide the same service $S_p$ but are in different clusters $C$, the weight of the edge $w_{ij}$ is set to 0.5. Finally, if two pods provide the same service $S_p$ and belong to the same cluster $C$, the weight of the edge $e_{ij}$ is set to 1. This design choice enables GraphOpticon to perform information fusion and distillation in an optimal manner, thus surpassing the various configurations of the highly sophisticated Hybrid LSTM Encoder-Decoder, while requiring a lower number of computational resources. According to the experimental results, GraphOpticon managed to surpass the best overall configuration, which requires 5 distinct Hybrid LSTM Encoder-Decoder models to be implemented instead of the single one that is used in the case of GraphOption.

On top of that, GraphOpticon is considerably more lightweight compared to even a single instance of the Hybrid LSTM Encoder-Decoder. Table 4 depicts the experimental results in terms of the Number of Parameters, Training Time, and Inference Time, GraphOpticon requires 67,847, 118.663, and 0.0394, respectively. By contrast, the Hybrid LSTM Encoder-Decoder requires 426,694, 320.199, and 0.092 for the same metrics. An NVIDIA GeForce RTX 3060 GPU was used for the training and inference processes. It is worth mentioning that these results can be attributed to the fact that the Hybrid LSTM Encoder-Decoder leverages 4 LSTM layers. More specifically, its encoder part consists of a Bi-LSTM layer (256 units) and an LSTM layer (128 units), while its decoder part consists of an LSTM layer (256 units) and a Bi-LSTM layer (128 units). On the other

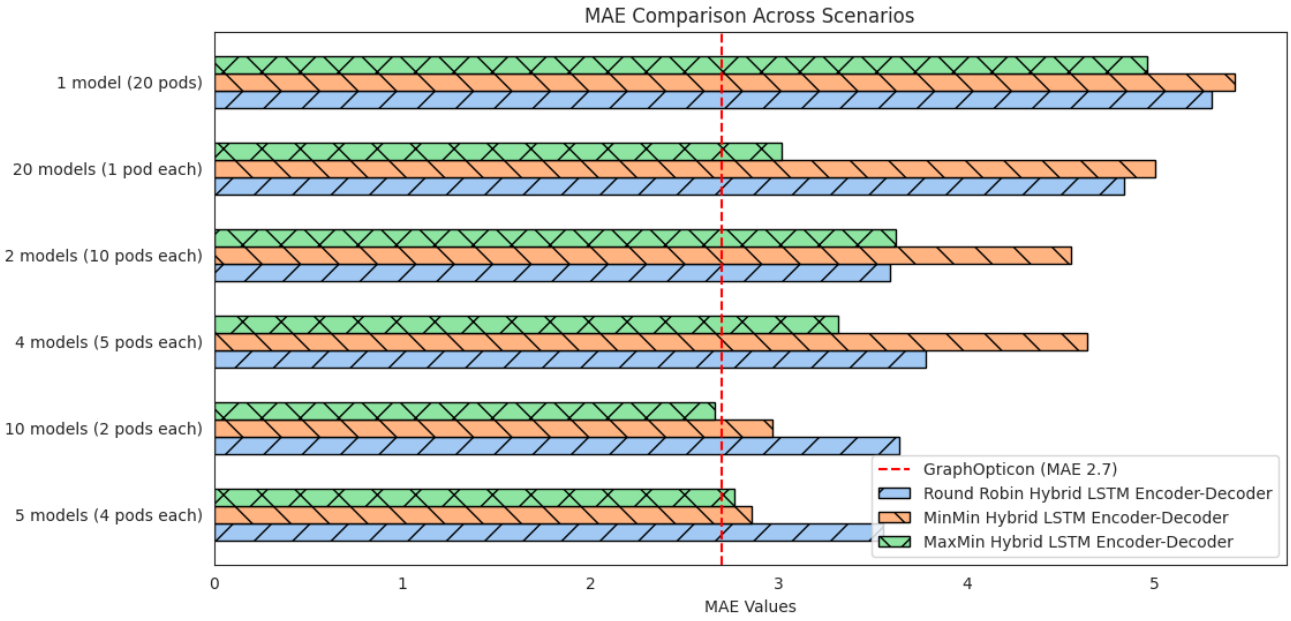


**Figure 8:** MAE comparison between the Hybrid LSTM ED and GraphOpticon, across various experimental configurations..

| Model | #Param. | Train. Time | Inf. Time |
|---|---|---|---|
| Hybrid LSTM ED | 426,694 | 320.199 | 0.092 |
| **GraphOpticon** | **67,847** | **118.663** | **0.0394** |

**Table 4**
Comparison of GraphOpticon and Hybrid LSTM Encoder-Decoder in terms of parameters, training time, and inference time.

hand, only the decoder of the GraphOpticon forecasting model leverages LSTM units (128). In terms of service performance, GraphOpticon's fewer parameters and lower inference and training times suggest that it can provide faster responses with reduced latency and quicker execution times compared to the Hybrid LSTM Encoder-Decoder model. This is beneficial for real-time or near-real-time applications where low latency is critical. The reduced number of parameters also implies that GraphOpticon consumes less computational power, leading to lower resource usage during both the training and inference phases. Thus, potentially requiring fewer hardware resources, it is a more resource-efficient choice for proactive horizontal autoscaling.

## 7.5. Performance & Resource Consumption: Experimental Results

Table 5 presents experimental results comparing execution times across different scheduling algorithms and scaling approaches. GraphOpticon consistently outperforms both Standard, Kubernetes, and Proactive methods, achieving lower average execution times, standard deviations, medians, maximums, and ranges in most cases. GraphOpticon achieves the lowest average execution times across all scheduling algorithms. For Round-Robin scheduling, GraphOpticon records an average execution time of 1.45 seconds, which is 3.33% lower than Standard (1.50 seconds), 1.36% lower than Kubernetes (1.47 seconds), and 0.68% lower than Proactive (1.46 seconds). In MinMin scheduling, GraphOpticon's average is 1.42 seconds, outperforming Standard (1.49 seconds) by 4.70%, Kubernetes (1.45 seconds) by 2.07%, and Proactive (1.45 seconds) by 2.07%. For MaxMin scheduling, GraphOpticon achieves an average of 1.52 seconds, a reduction of 7.32% compared to Standard (1.64 seconds), 4.40% compared to Kubernetes (1.59 seconds), and 0.66% compared to Proactive (1.53 seconds).

GraphOpticon also demonstrates greater consistency in execution times, as indicated by lower standard deviations. In Round-Robin scheduling, GraphOpticon's standard deviation is 0.89 seconds, 7.29% lower than Standard (0.96 seconds), 9.18% lower than Kubernetes (0.98 seconds), and 1.11% lower than Proactive (0.90 seconds). For MinMin scheduling, GraphOpticon achieves a standard deviation of 0.85 seconds, which is 3.41% lower than Standard (0.88 seconds), 15.84% lower than Kubernetes (1.01 seconds), and 2.30% lower than Proactive (0.87 seconds). In MaxMin scheduling, GraphOpticon has a standard deviation of 1.04 seconds, which is 12.61% and 11.11% lower than Standard (1.19 seconds) and Kubernetes (1.17 seconds), respectively, and 1.89% lower than Proactive (1.06 seconds).

GraphOpticon presents the lowest median execution times across all scheduling algorithms. For Round-Robin scheduling, GraphOpticon's median is 1.27 seconds, an improvement of 13.61% over Standard (1.47 seconds), 6.62% over Kubernetes (1.36 seconds), and 0.78% over Proactive (1.28 seconds). In MinMin scheduling, GraphOpticon's median is 1.26 seconds, 14.86% lower than Standard (1.48 seconds), 7.46% lower than Kubernetes (1.36 seconds), and 2.33% lower than Proactive (1.29 seconds). In MaxMin

| | | Execution Time | | | | |
|---|---|---|---|---|---|---|
| | | **Average** | **Standard Deviation** | **Median** | **Max** | **Range** |
| **Round-Robin** | Standard | 1.50 | 0.96 | 1.47 | 15.79 | 15.29 |
| | Kubernetes | 1.47 | 0.98 | 1.36 | 27.05 | 26.55 |
| | Proactive | 1.46 | 0.90 | 1.28 | 15.52 | 15.02 |
| | **GraphOpticon** | **1.45** | **0.89** | **1.27** | **15.22** | **14.72** |
| **MinMin** | Standard | 1.49 | 0.88 | 1.48 | 19.16 | 18.66 |
| | Kubernetes | 1.45 | 1.01 | 1.33 | 20.73 | 20.23 |
| | Proactive | 1.45 | 0.87 | 1.29 | 16.53 | 16.03 |
| | **GraphOpticon** | **1.42** | **0.85** | **1.26** | **13.05** | **12.55** |
| **MaxMin** | Standard | 1.64 | 1.19 | 1.54 | 21.65 | 21.15 |
| | Kubernetes | 1.59 | 1.17 | 1.52 | 38.30 | 37.80 |
| | Proactive | 1.53 | 1.06 | 1.45 | 21.10 | 20.60 |
| | **GraphOpticon** | **1.52** | **1.04** | **1.43** | **20.73** | **20.23** |

**Table 5**
Experimental results for Execution Time.

scheduling, GraphOpticon's median of 1.43 seconds is 7.14% lower than Standard (1.54 seconds), 5.92% lower than Kubernetes (1.52 seconds), and 1.39% lower than Proactive (1.45 seconds).

GraphOpticon reduces the maximum execution time in each algorithm, helping to minimize extreme outliers. In Round-Robin scheduling, GraphOpticon's maximum execution time is 15.22 seconds, 3.57% lower than Standard (15.79 seconds), 43.73% lower than Kubernetes (27.05 seconds), and 1.93% lower than Proactive (15.52 seconds). For MinMin scheduling, GraphOpticon achieves a maximum of 13.05 seconds, which is 31.99% and 37.11% lower than Standard (19.16 seconds) and Kubernetes (20.73 seconds), respectively, and 21.04% lower than Proactive (16.53 seconds). In MaxMin scheduling, GraphOpticon's maximum is 20.73 seconds, 4.16% lower than Standard (21.65 seconds), 45.83% lower than Kubernetes (38.30 seconds), and 1.75% lower than Proactive (21.10 seconds).

GraphOpticon has the lowest range of execution times, which reflects a more predictable performance. For Round-Robin scheduling, GraphOpticon's range is 14.72 seconds, a reduction of 3.73% compared to Standard (15.29 seconds), 44.72% compared to Kubernetes (26.55 seconds), and 2.00% compared to Proactive (15.02 seconds). In MinMin scheduling, GraphOpticon's range is 12.55 seconds, which is 32.74% lower than Standard (18.66 seconds), 38.05% lower than Kubernetes (20.23 seconds), and 24.35% lower than Proactive (16.03 seconds). For MaxMin scheduling, GraphOpticon achieves a range of 20.23 seconds, which is 4.21% lower than Standard (21.15 seconds), 46.59% lower than Kubernetes (37.80 seconds), and 1.80% lower than Proactive (20.60 seconds).

Table 6 illustrates latency and resource consumption metrics across different scheduling algorithms. GraphOpticon shows consistent improvements over the Standard, Kubernetes, and Proactive approaches, excelling in skewness, kurtosis, tail latency, and resource consumption across all scheduling configurations. GraphOpticon achieves the lowest skewness values, reflecting a more balanced latency distribution. For Round-Robin scheduling, GraphOpticon's skewness of 3.67 is lower than Standard (3.84), Kubernetes (4.36), and Proactive (3.82), achieving reductions of 4.43%, 15.8%, and 3.92%, respectively. In MinMin scheduling, GraphOpticon's skewness of 3.01 outperforms Standard's 3.85, Kubernetes' 4.43, and Proactive's 3.05, with reductions of 21.82%, 32.05%, and a minor improvement of 1.31% over Proactive. For MaxMin, GraphOpticon records a skewness of 4.07, which is 14.68% lower than Standard (4.77), 53.52% lower than Kubernetes (8.76), and 4.24% lower than Proactive (4.25). This reduction in skewness demonstrates GraphOpticon's superior ability to stabilize latency distributions across workload types.

GraphOpticon also exhibits the lowest kurtosis values, indicating fewer extreme outliers in latency. For Round-Robin scheduling, GraphOpticon achieves a kurtosis of 22.68, representing reductions of 20.66% compared to Standard (28.59), 36.48% compared to Kubernetes (35.70), and 12.93% compared to Proactive (26.04). With MinMin scheduling, GraphOpticon achieves a kurtosis of 15.93, significantly outperforming Standard (29.57), Kubernetes (73.85), and Proactive (17.85) by 46.11%, 78.42%, and 10.77%, respectively. In MaxMin scheduling, GraphOpticon's kurtosis of 30.05 shows improvements of 19.57% over Standard (37.36), 48.21% over Kubernetes (58.04), and 5.51% over Proactive (31.80). These results underscore GraphOpticon's robustness in handling extreme latency values, leading to more predictable performance.

GraphOpticon consistently achieves lower tail latency than Standard, Kubernetes, and Proactive models, reflecting fewer high-latency occurrences. In Round-Robin scheduling, GraphOpticon's tail latency of 4.09 is lower than Standard (4.40), Kubernetes (4.72), and Proactive (4.22), showing reductions of 7.05%, 13.34%, and 3.08%, respectively.

| | | Skewness of Latency | Kurtosis of Latency | Tail Latency | Resource Consumption |
|---|---|---|---|---|---|
| **Round-Robin** | Standard | 3.84 | 28.59 | 4.40 | 11.174 |
| | Kubernetes | 4.36 | 35.70 | 4.72 | 11.854 |
| | Proactive | 3.82 | 26.04 | 4.22 | 14.857(9.857+5) |
| | **GraphOpticon** | **3.67** | **22.68** | **4.09** | **10.747**(9.747+1) |
| **MinMin** | Standard | 3.85 | 29.57 | 4.16 | 10.907 |
| | Kubernetes | 4.43 | 73.85 | 4.08 | 11.109 |
| | Proactive | 3.05 | 17.85 | 3.92 | 14.608(9.608+5) |
| | **GraphOpticon** | **3.01** | **15.93** | **3.87** | **10.591**(9.591+1) |
| **MaxMin** | Standard | 4.77 | 37.36 | 5.06 | 11.419 |
| | Kubernetes | 8.76 | 58.04 | 5.12 | 11.793 |
| | Proactive | 4.25 | 31.80 | 4.85 | 15.080(10.080+5) |
| | **GraphOpticon** | **4.07** | **30.05** | **4.73** | **10.993**(9.993+1) |

**Table 6**
Experimental results for Latency and Resource Consumption.

With MinMin scheduling, GraphOpticon achieves a tail latency of 3.87, marking a 6.73% improvement over Standard (4.16), a 5.15% reduction compared to Kubernetes (4.08), and a slight improvement of 1.28% over Proactive (3.92). In MaxMin scheduling, GraphOpticon's tail latency of 4.73 seconds is 6.52% lower than Standard (5.06), 7.62% lower than Kubernetes (5.12), and 2.47% lower than Proactive (4.85). These findings highlight GraphOpticon's ability to minimize latency extremes effectively.

GraphOpticon proves to be the most resource-efficient option, requiring fewer resources across all scheduling algorithms compared to Standard, Kubernetes, and Proactive. In Round-Robin scheduling, GraphOpticon's resource consumption is 10.747, reducing consumption by 3.82% compared to Standard (11.174), 9.34% compared to Kubernetes (11.854), and an impressive 27.65% compared to Proactive (14.857). In MinMin scheduling, GraphOpticon uses 10.591 resources, representing reductions of 2.89% from Standard (10.907), 4.66% from Kubernetes (11.109), and 27.51% compared to Proactive (14.608). For MaxMin scheduling, GraphOpticon consumes 10.993 resources, showing a 3.73% reduction compared to Standard (11.419), a 6.77% reduction compared to Kubernetes (11.793), and a 27.06% reduction compared to Proactive (15.080). This efficiency in resource allocation positions GraphOpticon as the most cost-effective option.

## 7.6. Discussion

In terms of the three scheduling algorithms that were leveraged in the context of this work, some clear conclusions can be drawn. The evaluation of scheduling algorithms in this study reveals that the MaxMin approach, which prioritizes longer tasks, yields the poorest performance across all assessed metrics, including latency, execution time, and resource consumption. This inefficiency is particularly pronounced in workloads dominated by short tasks, as seen in the experimental framework. The inherent strategy of MaxMin scheduling causes short tasks to experience increased wait times, leading to higher overall execution time and elevated system latency.

In contrast, the MinMin scheduling algorithm, which prioritizes shorter tasks, significantly improves performance by quickly clearing the majority of the workload. This results in a lower average execution time and reduced tail latency, particularly in environments where short tasks are prevalent. Similarly, the Round-Robin approach, by evenly distributing tasks across available resources, ensures a balanced execution time and mitigates the monopolization of resources by any specific task type.

Moreover, the resource consumption is notably higher under MaxMin scheduling in short-task-heavy workloads due to prolonged resource occupation by longer tasks. This necessitates the activation of additional compute nodes to manage the backlog of shorter tasks, leading to inefficiencies in system resource utilization. By contrast, MinMin and Round-Robin scheduling optimize resource usage by either prioritizing short tasks for rapid completion or ensuring an equitable task distribution, respectively. These approaches contribute to more efficient system operation, minimizing the number of active nodes and reducing overall resource expenditure.

In terms of horizontal scaling algorithms, let us begin our analysis by focusing on Standard, Kubernetes, and GraphOpticon. While Kubernetes often achieves slightly lower average and median execution times compared to the Standard approach, it suffers from significantly higher maximum execution times, range of execution time, tail latency, resource consumption, skewness, and kurtosis of latency, indicating greater inconsistency and more frequent outliers in task execution times. This discrepancy can be attributed to the different methods used for horizontal autoscaling in Kubernetes and the Standard approach. The Standard approach utilizes two distinct thresholds for scaling in and out, providing a more controlled and predictable mechanism for resource allocation. When the system load reaches the

upper threshold, additional resources are allocated to manage the increased demand, and when the load drops below the lower threshold, excess resources are released. This binary threshold system ensures that resources are scaled in a predictable manner, which helps maintain a more consistent performance across varying loads.

In contrast, Kubernetes employs HPA, which uses a more dynamic formula for scaling. The HPA adjusts the number of running pods based on the observed CPU utilization against a target utilization defined by the user. The formula calculates the desired number of pods as a ratio of current CPU utilization to the target utilization. While this method allows for more fine-tuned and responsive scaling, it can also lead to greater variability in execution times due to the rapid and frequent adjustments in resource allocation. This can result in a system that is more susceptible to sudden spikes in demand, leading to higher maximum execution times, larger ranges, and increased skewness and kurtosis in the distribution of latency.

The proactive horizontal autoscaling approach employed by GraphOpticon is a key factor in its superior performance. Unlike the Standard and Kubernetes approaches, which react to changes in demand, GraphOpticon anticipates demand fluctuations and scales resources accordingly. This proactive approach leads to lower execution time and latency by scaling resources ahead of anticipated demand spikes, minimizing the occurrence of high-latency events, and maintaining a high quality of service. In addition, proactive scaling ensures that resources are allocated precisely when needed, avoiding both underutilization and overprovisioning. This efficient resource management reduces the average number of compute nodes used, lowering operational costs, and improving system efficiency.

GraphOpticon's efficiency can be attributed to two factors. The first is its ability to encapsulate temporal patterns throughout the temporal continuum. The second is its ability to further refine the encapsulation of these temporal patterns through information fusion and distillation. GraphOpticon, contrary to the other examined approaches, takes into consideration the ongoing and past CPU consumption values across numerous compute nodes to produce the values upon which the horizontal autoscaling process shall take place. These produced values provide valuable information regarding the number of pods that will be deployed in the near future. These insights can be leveraged to conduct the horizontal autoscaling process in an advanced manner that is associated with enhanced service performance and reduced resource consumption.

A sudden burst in resource consumption can be followed by: i) a further increase in resource consumption and ii) a decrease in resource consumption. GraphOpticon is capable of leveraging various temporal patterns to calculate which one of the scenarios is more likely to play out and formulate its predictions accordingly. In case a gradual increase follows the burst in resource consumption, GraphOpticon will proactively allocate additional computational resources to mitigate the ramifications on service performance that would derive from the insufficiency of resources. In case a decrease follows the burst in resource consumption, GraphOpticon will either de-allocate a compute node or maintain the ongoing compute node configuration as is. This is essential for achieving reduced resource consumption.

Aside from reduced resource consumption, this approach can achieve better service performance. In the scenario of a sudden burst in resource consumption, the reactive approaches would allocate more compute nodes. Newly produced tasks would then be assigned to these newly added compute nodes. However, if the sudden burst is just a random event, which is followed by a decrease in resource consumption, then resource overprovisioning would emerge, and subsequently, the newly added compute nodes would be de-allocated. The tasks that were assigned to the de-allocated nodes shall have to be re-assigned to other nodes, thus increasing latency and the overall execution time.

Furthermore, when encountering a sudden drop in resource demand, GraphOpticon can effectively manage two possible scenarios: i) a further decrease in resource usage or ii) a subsequent increase. By analyzing various temporal patterns, GraphOpticon predicts which scenario is more likely and adjusts its resource allocation strategy accordingly. If the drop is expected to be followed by an additional decrease, GraphOpticon proactively de-allocates compute nodes or maintains the current configuration without adding more resources. This approach prevents resource overprovisioning, reduces unnecessary costs, and enhances resource efficiency. By scaling down resources only when necessary, GraphOpticon ensures a lean operational state, which is crucial for minimizing resource consumption.

On the other hand, if the drop is anticipated to be temporary and followed by a gradual increase in resource consumption, GraphOpticon retains or slightly increases resource allocation. This proactive strategy helps mitigate potential performance issues arising from resource shortages when demand rises again. These performance issues also include the fact that the tasks of the previously de-allocated nodes would have to be re-assigned to alternative compute nodes, thus increasing latency and the overall execution time. By forecasting the need for more resources in advance, GraphOpticon maintains stable and efficient service performance, avoiding the drawbacks of reactive scaling. This constant reassignment process increases latency and overall execution time, negatively impacting service performance. Additionally, the rapid scaling up and down of nodes can introduce instability and additional overhead, further exacerbating performance issues. This phenomenon is referred to as resource oscillation. As showcased in the frame of this work, GraphOpticon is capable of mitigating resource oscillation by leveraging predictive analytics.

The Proactive manages to consistently outperform the Standard and Kubernetes approaches in terms of service performance, yet it performs slightly worse than GraphOpticon. This is due to the fact that although it is also a proactive approach and thus can outperform reactive approaches, as is evident by the prior discussion, its forecasting process is not

as accurate as that of GraphOpticon. Proactive exhibits the higher resource consumption among all approaches when accounting for the computational resources that are required to host the various forecasting models. In the event that we do not account for these additional resources, Proactive manages to outperform the Standard and Kubernetes approaches. Similarly to GraphOpticon, it is able to de-allocate resources proactively, which can lead to reduced resource consumption compared to reactive approaches. However, the additional resource overhead that is required in order to facilitate the 5 forecasting models negates the benefits of proactive horizontal autoscaling in terms of resource consumption. These experimental results, alongside the fact that GraphOpticon is considerably more lightweight than even a singular instance of the Proactive approach, enable us to safely conclude that GraphOpticon is the superior option in terms of improving service performance while reducing the resource consumption that is associated with workloads and the autoscaling system itself.

## 8. Conclusions & Future Research Directions

In this work, we introduced GraphOpticon, a global proactive horizontal pod autoscaling solution aimed at optimizing service performance and reducing resource consumption. It integrates four key components: Monitoring, Input Construction, Forecasting, and Output Distillation, utilizing GNNs for information fusion and distillation. The Input Construction component refines input sequence representations, enabling accurate resource consumption predictions across multiple pods and time steps through a global Forecasting mechanism. Output Distillation then leverages these predictions to generate tailored insights specific to pod characteristics. By leveraging predictive analytics, GraphOpticon maintains efficient resource usage and enhanced service performance, avoiding the pitfalls of overprovisioning and reassignment delays seen in reactive approaches. This proactive management ensures stability and efficiency, delivering optimal results in varying demand scenarios. These findings highlight the advantages of proactive scaling strategies in contemporary distributed systems. GraphOpticon not only reduces latency and execution time but also improves workload source consumption while minimizing autoscaling system resource consumption, ensuring a more reliable and efficient distributed computing environment.

The aforementioned decrease in workload resource consumption and minimization of autoscaling system resource consumption is expected to result in enhanced cost-savings and energy efficiency. As organizations increasingly rely on cloud-based infrastructure and distributed systems, the need for more sustainable and cost-effective solutions becomes critical. As such, future work shall focus on developing more detailed models to quantify the financial and environmental benefits of GraphOpticon's proactive autoscaling approach. This will involve conducting large-scale evaluations across various industries and workloads to measure the real-world impact of improved resource utilization.

## References


[1] E. F. Coutinho, F. R. de Carvalho Sousa, P. A. L. Rego, D. G. Gomes, J. N. de Souza, Elasticity in cloud computing: a survey, annals of telecommunications-annales des télécommunications 70 (2015) 289–309.

[2] N. Roy, A. Dubey, A. Gokhale, Efficient autoscaling in the cloud using predictive models for workload forecasting, in: 2011 IEEE 4th International Conference on Cloud Computing, IEEE, 2011, pp. 500–507.

[3] A. https://kubernetes.io/docs/tasks/run-application/horizontal-pod autoscale/., Horizontal pod autoscaling, section: docs. [online]. (2024).

[4] B. Shiva Prakash, K. Sanjeev, R. Prakash, K. Chandrasekaran, A survey on recurrent neural network architectures for sequential learning, in: Soft Computing for Problem Solving: SocProS 2017, Volume 2, Springer, 2019, pp. 57–66.

[5] T. Theodoropoulos, J. Violos, S. Tsanakas, A. Leivadeas, K. Tserpes, T. Varvarigou, Intelligent proactive fault tolerance at the edge through resource usage prediction, arXiv preprint arXiv:2302.05336 (2023).

[6] T. Theodoropoulos, A.-C. Maroudis, J. Violos, K. Tserpes, An encoder-decoder deep learning approach for multistep service traffic prediction, in: 2021 IEEE Seventh International Conference on Big Data Computing Service and Applications (BigDataService), IEEE, 2021, pp. 33–40.

[7] Y. Zhou, H. Zheng, X. Huang, S. Hao, D. Li, J. Zhao, Graph neural networks: Taxonomy, advances, and trends, ACM Transactions on Intelligent Systems and Technology (TIST) 13 (1) (2022) 1–54.

[8] Z. A. Sahili, M. Awad, Spatio-temporal graph neural networks: A survey, arXiv preprint arXiv:2301.10569 (2023).

[9] X. Tang, Q. Liu, Y. Dong, J. Han, Z. Zhang, Fisher: An efficient container load prediction model with deep neural network in clouds, in: 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), IEEE, 2018, pp. 199–206.

[10] Y. S. Patel, R. Jaiswal, R. Misra, Deep learning-based multivariate resource utilization prediction for hotspots and coldspots mitigation in green cloud data centers, The Journal of Supercomputing 78 (4) (2022) 5806–5855.

[11] E. Radhika, G. S. Sadasivam, J. F. Naomi, An efficient predictive technique to autoscale the resources for web applications in private cloud, in: 2018 Fourth International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB), IEEE, 2018, pp. 1–7.

[12] T. Theodoropoulos, A. Makris, I. Kontopoulos, A.-C. Maroudis, K. Tserpes, Multi-service demand forecasting using graph neural networks, in: 2023 IEEE International Conference on Service-Oriented System Engineering (SOSE), IEEE, 2023, pp. 218–226.

[13] B. Yu, H. Yin, Z. Zhu, Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting, in: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, International Joint Conferences on Artificial Intelligence Organization, 2018. doi:10.24963/ijcai.2018/505.
URL https://doi.org/10.24963%2Fijcai.2018%2F505

[14] J. Violos, S. Tsanakas, T. Theodoropoulos, A. Leivadeas, K. Tserpes, T. Varvarigou, Intelligent horizontal autoscaling in edge computing using a double tower neural network, Computer Networks 217 (2022) 109339.

[15] Kakade, Soham and Abbigeri, Gurutej and Prabhu, Om and Dalwayi, Akash and Patil, Somashekar Patil and Sunag, Bhagya and others, Proactive horizontal pod autoscaling in kubernetes using bi-lstm, in:

2023 IEEE International Conference on Contemporary Computing and Communications (InC4), Vol. 1, IEEE, 2023, pp. 1–5.

[16] N. Marie-Magdelaine, T. Ahmed, Proactive autoscaling for cloud-native applications using machine learning, in: GLOBECOM 2020-2020 IEEE Global Communications Conference, IEEE, 2020, pp. 1–7.

[17] M. Imdoukh, I. Ahmad, M. G. Alfailakawi, Machine learning-based auto-scaling for containerized applications, Neural Computing and Applications 32 (13) (2020) 9745–9760.

[18] N.-M. Dang-Quang, M. Yoo, Deep learning-based autoscaling using bidirectional long short-term memory for kubernetes, Applied Sciences 11 (9) (2021) 3835.

[19] J. Dogani, F. Khunjush, M. Seydali, K-agrued: a container autoscaling technique for cloud-based web applications in kubernetes using attention-based gru encoder-decoder, Journal of Grid Computing 20 (4) (2022) 40.

[20] J. Dogani, F. Khunjush, Proactive auto-scaling technique for web applications in container-based edge computing using federated learning model, Journal of Parallel and Distributed Computing 187 (2024) 104837.

[21] H. Ahmad, C. Treude, M. Wagner, C. Szabo, Towards resource-efficient reactive and proactive auto-scaling for microservice architectures, Journal of Systems and Software (2025) 112390.

[22] M. P. Yadav, Rohit, D. K. Yadav, Maintaining container sustainability through machine learning, Cluster Computing 24 (4) (2021) 3725–3750.

[23] Nguyen, Hoa X and Zhu, Shaoshu and Liu, Mingming, Graph-PHPA: graph-based proactive horizontal pod autoscaling for microservices using LSTM-GNN, in: 2022 IEEE 11th International Conference on Cloud Networking (CloudNet), IEEE, 2022, pp. 237–241.

[24] Goli, Alireza and Mahmoudi, Nima and Khazaei, Hamzeh and Ardakanian, Omid, A Holistic Machine Learning-based Autoscaling Approach for Microservice Applications, CLOSER 1 (2021) 190–198.

[25] Ju, Li and Singh, Prashant and Toor, Salman, Proactive autoscaling for edge computing systems with kubernetes, in: Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion, 2021, pp. 1–8.

[26] T. Theodoropoulos, A. Makris, I. Kontopoulos, J. Violos, P. Tarkowski, Z. Ledwoń, P. Dazzi, K. Tserpes, Graph neural networks for representing multivariate resource usage: A multiplayer mobile gaming case-study, International Journal of Information Management Data Insights 3 (1) (2023) 100158.

[27] D. Jakubovitz, R. Giryes, M. R. Rodrigues, Generalization error in deep learning, in: Compressed Sensing and Its Applications: Third International MATHEON Conference 2017, Springer, 2019, pp. 153–193.

[28] Tamiru, Mulugeta Ayalew and Tordsson, Johan and Elmroth, Erik and Pierre, Guillaume, An experimental evaluation of the kubernetes cluster autoscaler in the cloud, in: 2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2020, pp. 17–24.

[29] E. Casalicchio, V. Perciballi, Auto-scaling of containers: The impact of relative and absolute metrics, in: 2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W), IEEE, 2017, pp. 207–214.

[30] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, M. Bronstein, Temporal graph networks for deep learning on dynamic graphs (2020). doi:10.48550/ARXIV.2006.10637.
URL https://arxiv.org/abs/2006.10637

[31] Y. Zhao, J. Qi, Q. Liu, R. Zhang, Wgcn: graph convolutional networks with weighted structural features, in: Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, 2021, pp. 624–633.

[32] J. Chung, C. Gulcehre, K. Cho, Y. Bengio, Empirical evaluation of gated recurrent neural networks on sequence modeling, arXiv preprint arXiv:1412.3555 (2014).

[33] I. Korontanis, A. Makris, T. Theodoropoulos, K. Tserpes, Real-time monitoring and analysis of edge and cloud resources, in: Proceedings of the 3rd Workshop on Flexible Resource and Application Management on the Edge, FRAME '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 13–18. doi:10.1145/3589010.3594892.
URL https://doi.org/10.1145/3589010.3594892

[34] Silva Filho, Manoel C. and Oliveira, Raysa L. and Monteiro, Claudio C. and Inácio, Pedro R. M. and Freire, Mário M., CloudSim Plus: A cloud computing simulation framework pursuing software engineering principles for improved modularity, extensibility and correctness, in: 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), 2017, pp. 400–406.

[35] Zakarya, Muhammad and Gillam, Lee and Khan, Ayaz Ali and Rahman, Izaz Ur, Perficientcloudsim: a tool to simulate large-scale computation in heterogeneous clouds, The Journal of Supercomputing 77 (4) (2021) 3959–4013.

[36] Zakarya, Muhammad and Gillam, Lee and Salah, Khaled and Rana, Omer and Tirunagari, Santosh and Buyya, Rajkumar, CoLocateMe: Aggregation-based, energy, performance and cost aware VM placement and consolidation in heterogeneous IaaS clouds, IEEE Transactions on Services Computing 16 (2) (2022) 1023–1038.

[37] Verma, Abhishek and Pedrosa, Luis and Korupolu, Madhukar and Oppenheimer, David and Tune, Eric and Wilkes, John, Large-scale cluster management at Google with Borg, in: Proceedings of the Tenth European Conference on Computer Systems, EuroSys '15, Association for Computing Machinery, New York, NY, USA, 2015.

[38] Tirmazi, Muhammad and Barker, Adam and Deng, Nan and Haque, Md E. and Qin, Zhijing Gene and Hand, Steven and Harchol-Balter, Mor and Wilkes, John, Borg: the next generation, in: Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys '20, Association for Computing Machinery, New York, NY, USA, 2020.

[39] Zakarya, Muhammad and Gillam, Lee and Ali, Hashim and Rahman, Izaz Ur and Salah, Khaled and Khan, Rahim and Rana, Omer and Buyya, Rajkumar, epcAware: A game-based, energy, performance and cost-efficient resource management technique for multi-access edge computing, IEEE Transactions on Services Computing 15 (3) (2020) 1634–1648.

[40] Zakarya, Muhammad and Gillam, Lee, Modelling resource heterogeneities in cloud simulations and quantifying their accuracy, Simulation Modelling Practice and Theory 94 (2019) 43–65.