Dynamic Autoscaling and Scheduling in Kubernetes Clusters with LSTM and ILP

Somashekar Patil, Narayan D G, and Ajeya Bhat

Original scientific article

Abstract—**Containerized applications provide benefits such as portability, security, and faster deployment, enabling organizations to adapt swiftly to dynamic business needs. Kubernetes automates the deployment, scaling, and management of these applications, with the Kubernetes autoscaler enhancing availability and scalability by dynamically adjusting capacity to handle unexpected traffic spikes or workloads. The Kubernetes scheduler is also crucial for application autoscaling, as it schedules pods across different nodes. However, most existing research addresses autoscaling and scheduling separately. This work aims to integrate these two aspects by developing a Mixed Integer Linear Programming (ILP) model to minimize overall response time while maximizing throughput in a Kubernetes cluster through dynamic pod autoscaling and optimal scheduling. Additionally, we design a Long-Short Term Memory (LSTM)-based horizontal autoscaler and scheduler to efficiently manage pods during autoscaling. We then integrate these algorithms and evaluate their performance on a 9-node Kubernetes testbed. Results show that this combined approach outperforms the default algorithms in terms of response time and throughput across various traffic scenarios.**

Index Terms—**Kubernetes, AutoScaling, Scheduling, LSTM, HPA, VPA.**

I. Introduction

Containerization is a technique for bundling all components of an application into a container, which can then be executed using an orchestration tool. It serves as an alternative to virtualization, involving the encapsulation of software code, libraries, packages, and dependencies. This encapsulation ensures that containers can run consistently and uniformly across various infrastructure environments. Furthermore, containerization enables the deployment of multiple applications on a single server with the same operating system. Kubernetes is an important container orchestration platform. It is an open-source solution designed to facilitate the management of containerized services and workloads while offering automation capabilities.

Kubernetes employs two distinct types of autoscalers. The Horizontal Pod Autoscaler (HPA) adjusts the pod count, increasing or decreasing it as the workload grows or diminishes, respectively. On the other hand, the Vertical Pod Autoscaler (VPA) focuses on dynamically altering the CPU or RAM allocation for pods based on workload demands. HPA primarily scales the number of pods to effectively manage computational workloads. It monitors resource utilization, including CPU, memory, or any custom metrics provided by metric servers. HPA then modifies the pod replica count in response to workload requests, creating new pods when demand exceeds a predefined threshold and terminating pods when requests fall below this threshold. However, default autoscalar has it own disadvantages. Thus, numerous research studies have been conducted on HPA, with a significant emphasis on leveraging machine learning techniques to improve autoscaler performance, as discussed in [1] and [2].
In the context of HPA, once the decision to scale has been made, the next step involves scheduling the pods onto nodes. The default scheduler in Kubernetes achieves this by maintaining the desired state of the pod within the API server. This default scheduling process relies on node ranking, wherein each pod is assigned to an appropriate node based on a set of criteria. These criteria encompass various node attributes such as CPU usage rate, memory usage rate, power consumption, and the total number of containers currently running on the node. Nodes are filtered and ranked according to these properties, and the pod is then assigned to the most suitable node, which is typically the highest-ranked one. However, default scheduler has it own disadvantages. Thus, recent advancements have introduced machine learning techniques to enhance the performance of the Kubernetes scheduler, as discussed in [3] and [4].

The majority of research in the literature has concentrated on addressing autoscaling and scheduling as separate components. The objective of this study is to develop a dynamic algorithm that integrates Long-Short Term Memory (LSTM)-based autoscaling and scheduling specifically for Kubernetes, and then evaluate its performance against the default algorithms. Our approach merges the LSTM-based autoscaler and scheduler to enhance the overall performance of Kubernetes clusters. To the best of our knowledge, this is only the second research work except the work carried out in [5], to explore the integration of both autoscaling and scheduling in the context of Kubernetes. The contributions of the work are:

- Developed Mixed Integer Linear Programming (ILP) model to minimize the overall response time while maximizing throughput in a Kubernetes cluster through dynamic pod autoscaling and optimal pod scheduling.
- Deployed a multi-node Kubernetes cluster using microk8s and created a Prometheus adaptor to fetch/send custom metrics from an two endpoints namely pod and node.

Manuscript received October 9, 2024; revised November 19, 2024. Date of publication November 12, 2025. Date of current version November 12, 2025.
Authors are with the School of Computer Science and Engineering, KLE Technological University, Hubli, India (e-mails: Somuinfo123@gmail.com, narayan_dg@kletech.ac.in, abajstyles@gmail.com).
Digital Object Identifier (DOI): 10.24138/jcomss-2023-0147

- Designed and implemented an integrated dynamic horizontal pod autoscaler and scheduler using LSTM.
- Evaluated the proposed autoscaler and scheduler with default algorithms of Kubernetes.

The paper is structured as follows: Section II discusses various techniques used for Kubernetes autoscaling and scheduling. Section III outlines the system design and provides algorithms for each module. The findings of the proposed work are presented in Section IV, while Section V contains the study's results and explores potential directions for future research.

II. Related Work

In this section, we discuss related works that deal with Kubernetes, Horizontal Pod Autoscaling and Scheduling. Authors in [6] combine grey system theory with the LSTM neural network prediction method for container scheduling. The experimental results demonstrate that this algorithm effectively mitigates resource fragmentation issues in cluster worker nodes and enhances the overall utilization of cluster resources. Scheduling based solely on a single criterion can often lead to suboptimal performance, as it limits the scheduler's understanding of both the state of the cloud infrastructure and the user's requirements. Authors in [7] propose KCSS (Multi-Criteria Container Scheduling System), which introduces a multi-criteria node selection approach. This helps the scheduler with a comprehensive view of the cloud environment and the user's demands. The core idea behind KCSS is to carefully choose the most suitable node for each newly submitted container, striking a well-balanced compromise among various criteria that pertain to both the cloud infrastructure and user requirements. Authors consider six key criteria in the decision-making process.

Authors in [8] formulate the problem of scheduling container-based microservices as a multi-objective optimization challenge. The aim of the work includes optimizing network latency among microservices, enhancing the reliability of microservice applications, and achieving load balancing within the cluster. Using multiple metrics authors proposed scheduling algorithm that leverages particle swarm optimization techniques. The proposed approach effectively balance the competing objectives and constraints in the microservice deployment process within edge computing environments. In [9], the authors introduced a custom Kubernetes scheduler tailored to accommodate the unique demands of scheduling tasks within Docker clusters. They outlined the scheduler's architectural and design details and assessed its performance using simulation experiments. The outcomes of their experiments demonstrated that their custom scheduler surpassed the default Kubernetes scheduler in both resource utilization and job completion time. The authors in [10] offer an extensive survey of the existing research landscape concerning Docker cluster scheduling. Additionally, they introduce a promising solution to address the associated challenges. Authors in [11] devised an energy-efficient container-based scheduling approach to efficiently handle a diverse range of tasks in both IoT and non-IoT networks. The proposed method leverages the accelerated particle swarm optimization technique to swiftly identify the most suitable container for each task while minimizing delays. The proposed approach excels in deploying containers onto optimal cloud servers through an optimal scheduling strategy, further enhancing resource utilization.

Authors in [12] propose autoscaling mechanism based on the both Service Level Objective (SLO) violation prevention and recovery. The SLO violation mechanism adapts autoscaling thresholds according to SLO compliance. This component dynamically selects the appropriate CPU threshold, cooldown intervals, and the number of replicas based on the velocity of the load. The recovery component of the solution rectify SLO violations that may occur due to delays or resource underestimation. It achieves this by provisioning additional resources to bring the SLO back into compliance. Authors in [13] authors propose an adaptive HPA of POD resources within Alibaba cloud. Authors leverages a resilient decomposition forecasting algorithm and a performance training model to provide an optimal solution for adjusting the number of pods which minimizes POD resource usage while simultaneously ensuring the stability of the business operations.

In [14], the authors propose a proactive custom autoscaler that dynamically manages workloads during runtime using deep learning techniques. Among the various prediction models they tested, the Bidirectional long short-term memory (Bi-LSTM) model demonstrated the best performance. The approach proposed by authors in [15] addresses the limitations of the existing Kubernetes HPA by introducing a proactive approach that aims to reduce response delays and improve the efficiency of resource scaling. This approach combines empirical mode decomposition and AutoRegressive Integrated Moving Average (ARIMA) time series forecasting methods to make more informed scaling decisions. In [16], authors propose a new methodology called Microscaler. Authors design a system for services requiring scaling requirements and optimize their scaling to align with the agreed Service Level Agreement (SLA) while maintaining an optimal cost structure within micro-service ecosystems. Microscaler gathers QoS metrics with the assistance of a service mesh-enabled infrastructure. Subsequently, it employs a metric known as "service power" to check whether services are under-provisioned or over-provisioned.

The authors of [17] propose a mechanism to enhance the efficiency of resource scaling in an active Kubernetes autoscaling system. Mechanism is based on predicting pod replica requirements and proactively adjusting resource allocation, thereby improving overall scaling efficiency. The authors in [18] focuses on the challenge of selecting the most suitable performance metrics to trigger auto-scaling actions effectively. In particular, the study explores the use of both relative and absolute metrics. The results reveal that, especially in scenarios with CPU-intensive workloads, using absolute metrics leads to more precise scaling decisions. In [19], authors introduce Reinforcement Learning (RL) based approach to manage the horizontal and vertical scalability of container-based applications, aiming to enhance adaptability in response to varying workloads. To speed up the learning process and identify more effective adaptation strategies, authors present

RL solutions that leverage different levels of understanding of the system. Authors evaluate the proposed approach in elastic docker swarm.

In [20], the authors employ both reactive and predictive scaling techniques. The core scaling approach is reactive, where a scaling manager dynamically adjusts resource allocation based on real-time metrics such as the number of requests per minute or active users. Scaling up or down is performed through the cloud provider's API, depending on the observed usage. Authors in [21] introduces an effective multivariate autoscaling framework for cloud computing, employing Bi-LSTM models. The framework was developed following the monitor-analyze-plan-execute loop. Experimental results, conducted using various real workload datasets, demonstrate that the proposed multivariate Bi-LSTM approach achieved a significantly lower Root Mean Square Error(RMSE) prediction error, approximately 1.84 times smaller than that of the univariate approach. Authors in [22] introduces similar approach using Bi-LSTM. The similar approach is used in [23] which introduces a custom LSTM-based autoscaler for horizontal pod autoscaling, comparing its performance to the default autoscaler using response times and SMAPE error values. The results show that the LSTM model outperforms the ARIMA model in predicting pod scaling. Authors use Kubernetes cluster for the evaluation.

Recent advancements in Kubernetes optimization frameworks have introduced innovative approaches to resource scheduling and management. For instance, KOptim offers a flexible schema for resource scheduling using SLA classes, enhancing adaptability to dynamic workloads and aligning container resource allocation with predefined SLA requirements [24]. CAROKRS integrates cost-aware optimization strategies, achieving significant reductions in deployment costs and improvements in load balancing compared to Kubernetes' native scheduler, highlighting the increasing focus on cost efficiency in resource allocation [25]. UniSched, tailored for deep learning applications, improves throughput and reduces job completion time by specializing resource allocation for diverse user demands, while SAGE leverages real-time metrics and predictive analytics to automate optimal workload placement and improve resource utilization [26][27]. Other studies, such as the Kubernetes Scheduler Optimization Based on Real Load, further emphasize the role of real-time workload data in enhancing resource utilization and reducing latency [28].

These studies collectively contribute to the ongoing evolution of Kubernetes scheduling and autoscaling frameworks, each focusing on different aspects such as cost efficiency, workload-specific optimization, and real-time decision-making. The proposed framework in this work builds upon these advancements by integrating predictive autoscaling using LSTM models with an ILP-based scheduler, offering a comprehensive solution for dynamic cloud-native environments. This approach addresses both workload prediction and optimal resource allocation, aiming to improve scalability, efficiency, and overall performance in containerized applications [29]. The authors in [30] propose an efficient placement of Service Function Chains (SFC) utilizing LSTM and ILP; however, their work primarily emphasizes container scheduling. In summary, the studies reviewed in the literature focus on scheduling and autoscaling separately. In contrast, our work integrates both autoscaling and scheduling using ML.

III. Proposed Methodology

In Section A, we discuss the mathematical formulation for the proposed system. In Section B, we discuss the proposed system model for the combined horizontal pod autoscaling and scheduling model in Kubernetes. Finally, we discuss the algorithms used in the implementation in Section C.

A. Mathematical Formulation

The Mixed Integer Programming (MIP) model is designed to optimize the autoscaling and scheduling of pods across nodes in a Kubernetes cluster. The objective is to minimize response time while maximizing throughput. The model uses binary variables to represent whether a pod is scheduled on a particular node and whether a node is active. It also includes integer variables for the number of pods allocated to each node. The objective function combines minimizing response time with maximizing throughput. Constraints ensure that resource demands of pods do not exceed node capacities, each pod is assigned to one node, and the overall system throughput meets the target. Autoscaling conditions are applied to activate or deactivate nodes based on workload demands.

Model Variables

The model employs several key variables to represent the decision-making process for scheduling pods and activating nodes:

$$
\begin{aligned}
x_{ij} &: \text{Binary variable indicating if pod } i \text{ is scheduled on node } j \\
y_j &: \text{Binary variable indicating if node } j \text{ is active} \\
z_i &: \text{Binary variable indicating if pod } i \text{ is active} \\
p_j &: \text{Integer variable for pods assigned to node} j.
\end{aligned}
$$

Model Parameters

Parameters represent fixed values that guide the model's decision-making.

$$
\begin{aligned}
R_i &: \text{Resource requirement (CPU, memory) of pod } i. \\
C_j &: \text{Capacity of node } j \ (CPU, \ \text{memory}). \\
&\quad \text{This represents the total available resources of a node.} \\
T_i &: \text{Estimated response time for pod } i. \\
D &: \text{Desired throughput.} \\
S_{ij} &: \text{Scheduling delay of pod } i \text{ on node } j. \\
\lambda_i &: \text{Workload arrival rate for pod } i. \\
\mu_j &: \text{Service rate of node } j.
\end{aligned}
$$

Constraints

The constraints enforce the logical and physical limits of the system:

Resource Capacity Constraint:
Ensure that the total resource demand of the scheduled pods does not exceed the node's capacity:

$$\sum_i x_{ij} R_i \leq y_j C_j \quad \forall j \tag{1}$$

This constraint ensures that a node is not overloaded beyond its capacity when scheduling pods.

Pod Scheduling Constraint:
Each pod is scheduled on exactly one node:

$$\sum_j x_{ij} = z_i \quad \forall i \tag{2}$$

This constraint guarantees that each pod is assigned to one and only one node.

Node Activation Constraint:
A node is activated if any pod is scheduled on it:

$$y_j \geq x_{ij} \quad \forall i, j \tag{3}$$

This ensures that a node is only activated (turned on) if at least one pod is scheduled to run on it.

Throughput Constraint: The total service rate must meet or exceed the desired throughput:

$$\sum_j y_j \mu_j \geq D \tag{4}$$

This constraint ensures that the system's overall processing power is sufficient to meet the throughput requirement.

Pod Count Constraint:
The total number of pods allocated to a node should match the number of active pods:

$$p_j = \sum_i x_{ij} \quad \forall j \tag{5}$$

This ensures that the count of pods assigned to each node is consistent with the scheduling decisions.

Objective Function

The objective function combines minimizing response time with maximizing throughput:

$$\text{Minimize} \quad \sum_{i,j} x_{ij} T_i S_{ij} - \alpha \sum_j y_j \mu_j \tag{6}$$

- The first term, $\sum_{i,j} x_{ij} T_i S_{ij}$, aims to minimize the overall response time by considering the scheduling delay and the estimated response time of each pod.
- The second term, $-\alpha \sum_j y_j \mu_j$, seeks to maximize throughput by encouraging the activation of nodes that can handle higher workloads, where α is a weighting factor balancing the two objectives.

Autoscaling Condition

The autoscaling condition activates or deactivates nodes based on the workload demand:

$$y_j = \min\left(1, \frac{\sum_i x_{ij} R_i}{C_j}\right) \quad \forall j \tag{7}$$

This condition ensures that a node is scaled up (activated) when necessary to handle the resource demands of the scheduled pods.

Optimizer

To solve this MIP model efficiently, the Gurobi optimizer is used. Gurobi is well-suited for handling complex MIP problems due to its advanced branch-and-bound algorithms, efficient presolve techniques, and cutting planes that reduce the solution space. It is optimized for performance, making it ideal for large-scale, real-time applications like Kubernetes scheduling.

B. System Model

The proposed system model consists of two components, namely autoscaler and scheduler as shown in Fig. 1. Both components use ML model for prediction. The target resource utilization level, as well as the minimum and a maximum number of replicas for the target Deployment or ReplicaSet, are then specified for HPA. When the HPA determines that scaling is necessary, the scheduling model is then utilized to make scheduling decisions for the new pods as can be seen in Figure 1. This model incorporates information such as the projected performance and resource utilization of an application, as well as other considerations like affinity and anti-affinity rules. The Kubernetes API is used to manage all of the communication between the various modules, and the entire operation is controlled by an HPA control loop. For instance, to change the target deployment's or ReplicaSet target's number of replicas.

The experimentation was conducted on a Kubernetes cluster deployed in a controlled environment to evaluate the performance of the proposed framework. To monitor and collect performance metrics, Prometheus version 2.24.0 was deployed across the cluster. A custom Prometheus adapter was implemented to query resource utilization metrics such as CPU and memory usage and application-specific metrics like response time and throughput. Grafana dashboards were used to visualize the collected data in real-time, aiding in performance analysis.

The workload for the experiments was designed to test the cluster under various traffic scenarios. A combination of a publicly available dataset and a custom testbed dataset was used. The Web Traffic Time Series dataset provided by Google was utilized for training the LSTM-based autoscaler. This dataset contained time-series data of daily views of Wikipedia articles from July 2015 to December 2016. The LSTM model was trained to predict resource utilization based on historical workload patterns. Additionally, a custom testbed dataset was generated by simulating traffic using the "hey" HTTP load generator. This dataset captured CPU usage, memory

consumption, and request throughput over a 3-hour period, during which varying volumes of HTTP requests were sent to the cluster to emulate real-world workload fluctuations.

The experiments involved deploying a containerized web application on the Kubernetes cluster. The application was subjected to dynamic workloads to evaluate the effectiveness of the proposed framework in autoscaling and scheduling. The Horizontal Pod Autoscaler (HPA) was configured to adjust the number of replicas based on the predicted workload. Simultaneously, the custom scheduler, incorporating the ILP model, was tasked with optimizing pod placement across the nodes. Metrics such as response time, throughput, and the number of active pods were measured to compare the performance of the proposed framework against the default Kubernetes mechanisms.

To ensure repeatability and accuracy, the cluster was set up using a lightweight Kubernetes distribution. Node resource allocation and application deployment configurations were managed through YAML files. Performance metrics were collected in real time using Prometheus, enabling detailed analysis of the system's behavior under different traffic conditions. By employing this setup, the experimentation effectively demonstrated the scalability, efficiency, and resource optimization capabilities of the proposed LSTM-based autoscaling and ILP-based scheduling framework.

From Fig. 2, we can observe that the system's overall steps include gathering and processing data, training an LSTM model, establishing the HPA and scheduling module, and setting up the control loop to continually track resource utilization and initiate scaling events. Overall, the system requires gathering and processing data, training an LSTM model, establishing the HPA and scheduling module, and setting up the control loop to continually monitor resource utilization and initiate scaling events depending on the expected resource demand.

C. *Proposed Algorithms*

In this section, we initially discuss the algorithm used in the implementation of autoscaling and scheuler separately. Later, we discuss the combined algorithm of autoscaling and scheduling.

Horizontal Pod AutoScaling Algorithm:

The objective of the autoscaling algorithm is to find the optimum number of pods to scale to, based on the predicted metrics and the HPA configuration. As shown in Algorithm 1, the autoscaling algorithm takes the following parameters as inputs, the prediction model, the identifier for the deployment(label), and the deployment target metric value for that deployment.

A control loop is used to do polling and scaling at regular intervals of t seconds. Inside the loop, we fetch the number of currently Active Pods associated with that deployment and the metric values forecasted by the prediction model. Finally, we calculate the desired number of replicas using the same formula that is being used by the default HPA.

Algorithm 1 Horizontal Pod AutoScaling Algorithm

// target metric value, the model for prediction and deployment identifier
Input: $U_t, model, deploymentIdentifier$
Output: P // Target number of Pods
while true **do**
$currentDeploymentReplicas$:= $getCurrentReplics(deploymentIdentifier)$;
// get predicted metrics using the prediction model, Kubernetes, and Prometheus APIs.
U_i = $getPredictedMetricValue(model)$;
P = $ceil[currentDeploymentReplicas$ * (U_i / U_t)]
// wait t seconds after which the control loop repeats. $wait(t)$
end while

Scheduling Algorithm

As discussed, pod scheduling is triggered during autoscaling of applications. As shown in Algorithm 2, the scheduling algorithm takes the target pod configuration and its affinities to find the optimal node on which it is to be deployed, based on the predicted metrics. When the scheduling algorithm is triggered, we obtain the configuration of the pod to schedule along with the list of nodes available for scheduling ad the prediction model. In the scoring step, we get the forecasted metrics of each node. Then, in the filtering step, we take node affinity, and resource requirements into account and then select the best node using the scores and the filtered nodes.

Algorithm 2 Scheduling Algorithm

Input: $AvailableNodes, model, podConfig$
Output: K // Node to be selected
$N_a = []$
for all $i \epsilon AvailableNodes$ **do**
N_i = $getPredictedMetrics(model, i)$; //scoring
$N_a.push(N_i)$ // Push the predicted metrics to array
end for
$K = getNodeToBeSelected(N_a, podConfig)$
$wait(t)$ //wait till the control loop period

Combined Algorithm

Algorithm 3 represents a high level overview of the Combined autoscaling and scheduling model. The combined algorithm involves both scaling and scheduling. First, we create the deployment and apply the custom HPA configuration to that deployment. Then, we keep monitoring the deployment for autoscaling purposes. Once scaling occurs, we check whether the desired number of replicas is greater than the currently active replicas. If yes, then new pods are created and those pods need to be scheduled. The scheduling module is then triggered to schedule those pods.

D. *Algorithm Complexity Analysis*

The proposed framework is analyzed in terms of algorithmic complexity by examining its two primary components: the LSTM-based autoscaler and the ILP-based scheduler. The LSTM-based autoscaler predicts future resource utilization based on historical workload data. Its complexity arises from the internal operations of the LSTM model, where computations involve processing input sequences over multiple time steps. Specifically, for each time step, the LSTM processes features through matrix multiplications that scale with the number

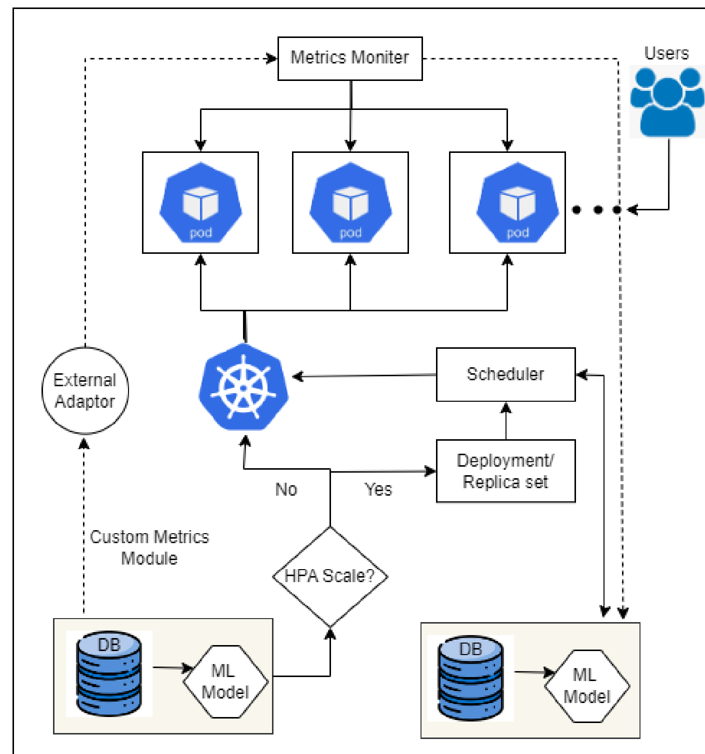

Fig. 1. Proposed Scaling and Scheduling Architecture

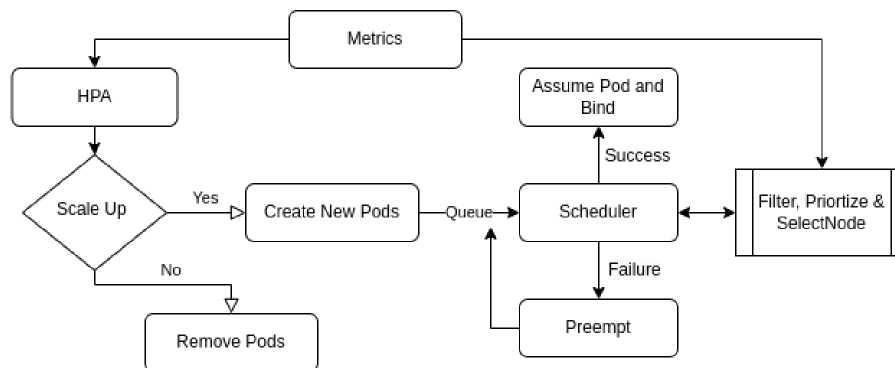

Fig. 2. Flowchart for Proposed Scaling and Scheduling

Algorithm 3 Combined Autoscaling and Scheduling Algorithm with ILP Model

Input: $model$, $deploymentConfig$, $HPAConfig$
$deployment \leftarrow createDeployment(deploymentConfig)$
$hpa \leftarrow createHPA(HPAConfig)$
while true **do**
$nodes \leftarrow getAllNodes()$
`// Target pods to be autoscaled`
$TP \leftarrow ScalingModule(hpa.target, model, deployment.label)$
  for all $i \in TP$ **do**
`// Solve the ILP model to find optimal scheduling`
$optimalSchedule \leftarrow solveILP(nodes, TP, model)$
`// Apply the optimal schedule to the pods`
$applySchedule(optimalSchedule, deployment.podCfg)$
  end for
end while

of features and hidden units. When extended across all time steps, the overall complexity is proportional to the product of the sequence length, the number of features, and the number of hidden units. However, in practice, this predictive process is highly efficient due to the relatively small sequence lengths and neural network dimensions used in this application. As a result, the autoscaling component introduces only a slight overhead with a computational complexity of O(n), where n represents the number of pods in the system.

In contrast, the ILP-based scheduler focuses on optimizing pod placement across the available nodes in the cluster. The scheduler solves an optimization problem that seeks to minimize latency and maximize throughput while adhering to various resource constraints. The computational complexity of this process grows exponentially with the number of decision variables, which is determined by the number of pods and nodes in the cluster. Specifically, the complexity of solving the ILP problem can be expressed as O(2nm) where n is the number of pods and m is the number of nodes. While this represents a significant theoretical complexity, modern ILP

solvers, such as Gurobi, leverage advanced techniques like branch-and-bound algorithms and heuristics to significantly reduce the practical runtime. Furthermore, the scheduler operates asynchronously, limiting the computation to specific intervals or periods of low cluster activity, which helps mitigate its impact on system performance.

The default Kubernetes methods for autoscaling and scheduling offer simpler computational models. The Horizontal Pod Autoscaler (HPA) evaluates resource utilization metrics for each pod independently, resulting in a linear complexity of O(n), where n is the number of pods. Similarly, the default Kubernetes scheduler filters and ranks nodes for pod placement using predefined criteria. The complexity of node filtering scales linearly with the number of nodes, while the overall scheduling process, which considers all pods and nodes, scales as O(nm).

IV. Results and Discussions

Within this section, we discuss the experimental setup and results obtained for autoscaler, scheduler and combined approach.

A. Experimental Setup

For experimentation, the Kubernetes cluster is set up with Master Node with 16 GB RAM and 100 GB storage. The experimental setup also had 8 slave nodes with 2 of them with 4 GB RAM and rest had 2 GB RAM. All of the slave nodes had 50 GB storage. Common software and hardware configurations used in this experiment are mentioned in Table I. In this work, LSTM model is used to predict the scaling of pods by creating the replicas of pod based on the workload. Table II shows the parameters used in LSTM model.

TABLE I
Experimental Configuration.

Resource Name	Specifications
The Operating System	Ubuntu 22.04
docker	20.10.3
k8s	1.23
prometheus	2.24.0
CPU	Intel core i5

TABLE II
LSTM Model Configuration.

Parameter	Quantity
num$_lstm$	10
epochs	10
Loss Function	SMAPE, RMSE
Optimizer	Adam
Batch Size	64

B. Autoscaling Results

In this section, we intially discuss the prediction results for autoscaler using standard dataset as well as testbed dataset using ARIMA and LSTM models. Later, we compare the LSTM based autoscaler with default Kubernetes autoscalar.

Prediction Results:

To train and test the autoscaling prediction models, we used 2 different datasets, namely, the 'Web Traffic Time Series' dataset by Google and our very own emulated test-bed dataset.

Standard Dataset

The training dataset consists of approximately 145k time series. Each time series instance represents a number of daily views of a different Wikipedia article, from July, 1st, 2015 up until December 31st, 2016. LSTM and ARIMA models are compared and to evaluate the model, SMAPE values are considered for the analysis. Table III gives prediction results of web traffic dataset. Results reveal that LSTM performs better than ARIMA.

TABLE III
Prediction Results of web traffic dataset.

	LSTM Model	ARIMA Model
Training Error(SMAPE)	31.02	55.68
Training Error(SMAPE)	14.99	43.44

Testbed Dataset

This testbed dataset is similar to the standard dataset in terms of the parameters considered. It was collected by sending a similar volume of requests in a 3-hour timeframe to the experimental setup using 'hey', an open-source HTTP load generator. The cluster contains one master and 8 slave nodes and have the same configuration as mentioned in Section A.

TABLE IV
Prediction Results of testbed dataset.

	LSTM Model	ARIMA Model
Training Error(SMAPE)	44.656	59.68
Tesr Error(SMAPE)	10.317	51.83

Tables III and IV describe the Symmetric Mean Absolute Percentage Error (SMAPE) values of machine learning models LSTM and ARIMA for each of the datasets. In both the datasets, we observe that the LSTM model has less error rate compared to the ARIMA model. This is because, LSTM is designed to capture long-term dependencies in time series data, which is particularly useful for forecasting complex patterns which can be used for better forecasting.

Performance analysis of dynamic HPA:

The performance of the default and faster dynamic HPA is analyzed based on the response time for HTTP requests. As shown in Fig. 3, the default HPA responds significantly slower to changes in request volume or idle time compared to the proposed HPA. In contrast, the custom autoscaler demonstrates improved performance over the default horizontal pod autoscaler. By predicting future workload, the faster custom HPA accelerates pod scaling decisions, both up and down.

To efficiently manage future workload reductions, the custom HPA preemptively removes some replicas when demand decreases.

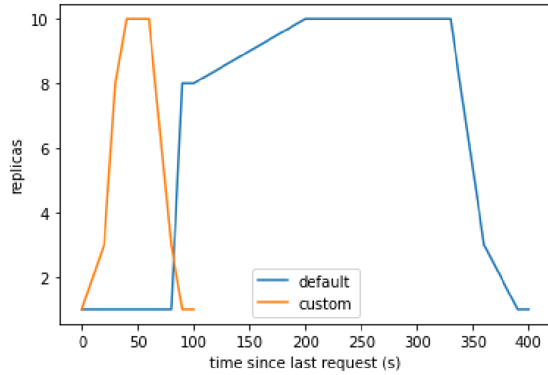

Fig. 3. Performance of default and dynamic HPA.

C. Scheduling Results

For scheduling, we firstly created a time series dataset by taking parameters from all the nodes while sending dummy HTTP requests at intervals that follow a similar seasonality that was found in the previously mentioned 'Web Traffic' dataset. Then, we trained and tested the prediction models using that dataset.

Prediction Results:

Figures 4 and 5 represent the training and testing performances of the LSTM model. Finally, Table V shows the Root-mean-square deviation(RMSE) values of LSTM and ARIMA models. From this, we can infer that the RMSE value of LSTM is better than ARIMA. Thus, we can validate that the LSTM model is better than the ARIMA model. This can be attributed to the LSTM's handling of non-stationary data. LSTM can handle non-stationary time series data, which is data that has a changing mean, variance, or other statistical properties over time. ARIMA, on the other hand, assumes that the time series data is stationary, meaning that its statistical properties do not change over time. From Table V, we can observe that LSTM performs better than ARIMA in prediction load.

TABLE V
RESULTS OF LSTM AND ARIMA MODELS.

Model	RMSE Value
LSTM	2.01
ARIMA	5.90

Performance Analysis of Dynamic Scheduler:

From Table VI, we can observe that LSTM-based scheduler response time is less compared to the default scheduler. Hence, we can conclude that LSTM performs well compared to the default scheduler.

The Fig. 6 depicts the response times of the different schedulers. From the graph, we can see that the average response time of LSTM is lesser compared to the average response time of the default scheduler. Hence we can validate

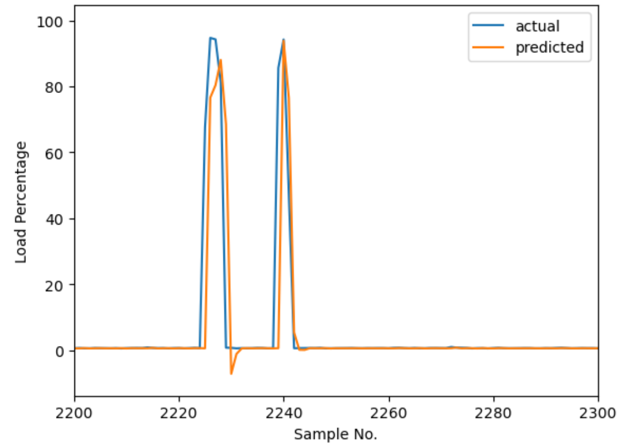

Fig. 4. Training performance of LSTM Model.

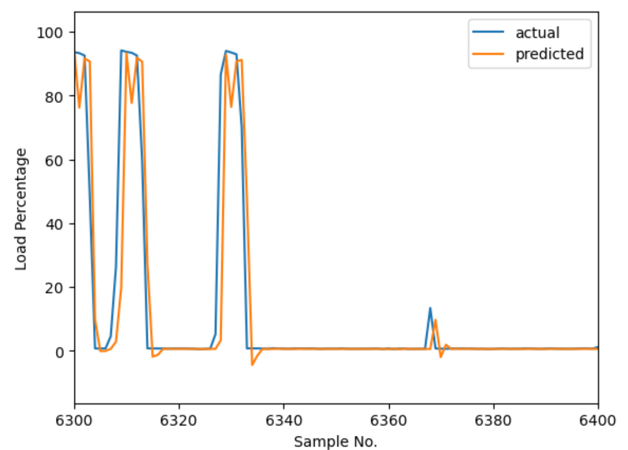

Fig. 5. Test performance of LSTM Model.

TABLE VI
RESPONSE TIME RESULTS OF LSTM AND DEFAULT SCHEDULERS.

	Average Response Time
LSTM based Scheduler	1.094
ARIMA based Scheduler	1.2111
Default Scheduler	1.456

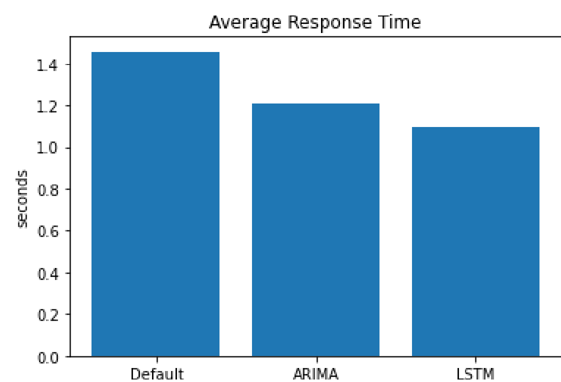

Fig. 6. Comparison of Response Time of Different Schedulers.

that LSTM optimizes the response time.

D. Combined Scaler and Scheduler

Performance evaluations of the default and proposed models are conducted taking into account a variety of parameters, namely latency, throughput, and the number of actively running pods while responding to HTTP requests. The custom scheduler and the autoscaler are integrated into the combined model to obtain the best performance. In order to perform the assessment, HTTP requests were sent to deployments in a way that the volume of requests over time exhibited some seasonality.

For one of the tests, the characteristics for each of the models, the LSTM model, the ARIMA model, and the default system, were obtained by sending varied amounts of HTTP requests to the deployment for 30 minutes. Based on the typical volume of requests submitted during a given period, the test results were then averaged out at the minute mark and then plotted and split into two broad groups, low and high load, to help draw conclusions.

Underloaded Scenario:

In this scenario, we send low volumes of HTTP requests and observe the average latency, throughput, and active pods/replicas for the course of the test.

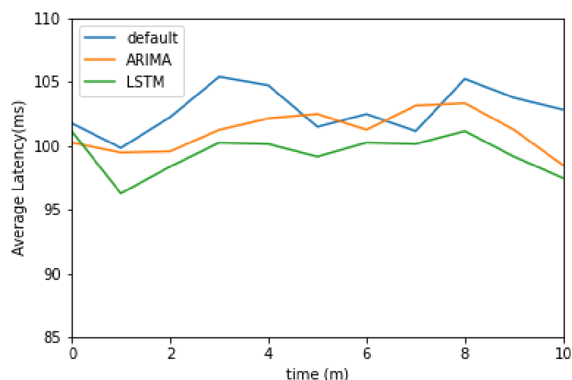

Fig. 7. Average latency of various models.

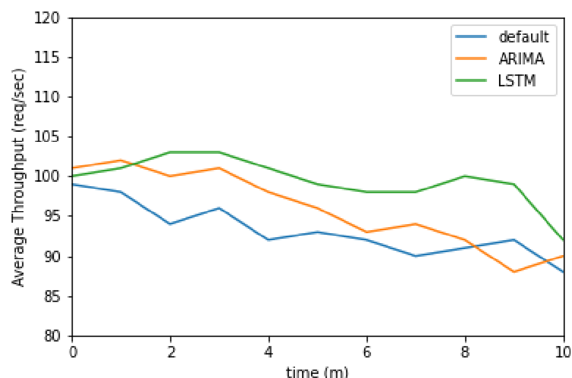

Fig. 8. Average throughput of various models.

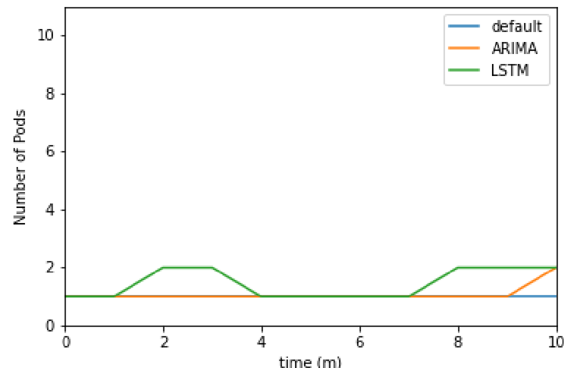

Fig. 9. Average active pods of various models.

Since the machine learning models try and forecast potential future metrics, they are able to take scaling decisions earlier than the default system. This can be seen in Figures 7 and 8 where these models perform better in terms of latency and throughput. Referring to Fig. 9, we can see that the improvement in performance does come at a cost of more active pods at work, where the dynamic models have a higher average number of replicas than the default system over time.

Overloaded Scenario:

In contrast to the default system, the proposed model's Scaling and Scheduling are quicker and smoother, resulting in no significant throughput loss. Over time, this may assist in satisfying a greater number of requests. This can be observed from Fig. 11.

As illustrated in Figures 10 and 11, the latency and throughput of all three systems eventually converge to comparable conditions. This is because the maximum number of pods set to the autoscaler has been reached and there is no more room for scaling as seen in Fig. 12. In this experiment, the minReplicas was set at 1 pod and the maxReplicas was 10. So the HPA cannot set the number of replicas to a number that is out of that range.

Randomly Loaded Scenario:

In previous scenarios discussed the behavior of the combined model under distinct conditions like underloaded and overloaded scenarios. In this scenario, we have considered varying loads for a longer duration of time so that we get the combination of both underloaded and overloaded scenarios. We will now discuss the results of the analysis on the random loaded scenario. The parameters considered are average latency, throughput and active pods.

Latency:

Table VII describes the Percentage Decrease in latency of LSTM and ARIMA models with respect to the default model. As shown in Fig. 13, the proposed model's latency is almost always lower compared to that of the default system. The average latency of the LSTM, ARIMA, and default system

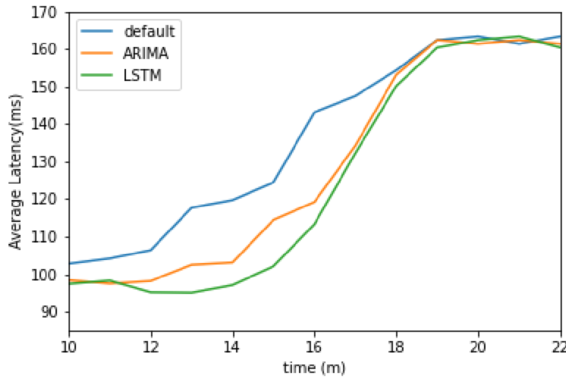

Fig. 10. Average latency of various models at high load.

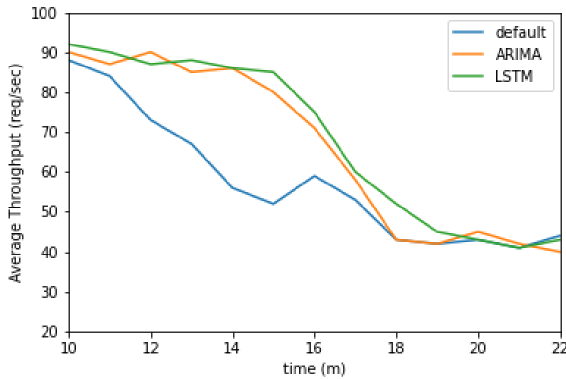

Fig. 11. Average throughput of various models at high load.

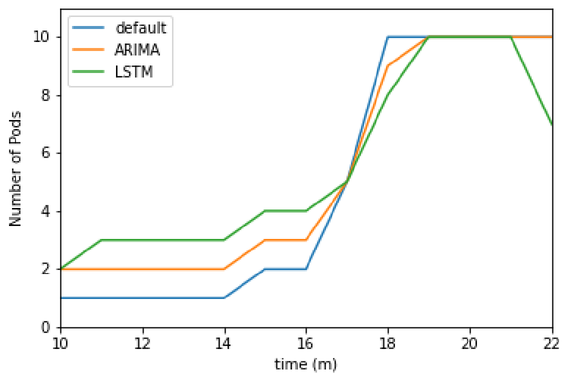

Fig. 12. Average active pods of various models at high load.

was approximately 121.82 ms, 124.66 ms, and 126.45 ms, respectively. This is a 3.65% reduction in latency compared to the default system as shown in Table VII.

TABLE VII
PERCENTAGE LATENCY DECREASE WITH RESPECT TO THE DEFAULT SYSTEM.

Model	% Decrease in latency
ARIMA	1.414
LSTM	3.656

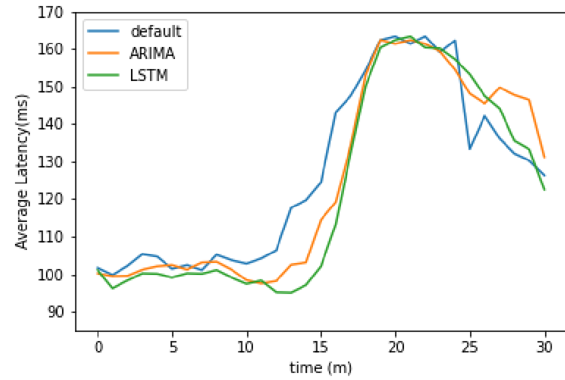

Fig. 13. Average Latency of various models.

Throughput:

Fig. 14 shows the average throughput of various models. Results reveal that LSTM model performed better than default model. Table VIII describes the Percentage Increase in throughput of LSTM and ARIMA models with respect to the default model. We can observe that the average throughput of the LSTM-based combined model is the highest among them, 6.4% higher than the default system to be specific.

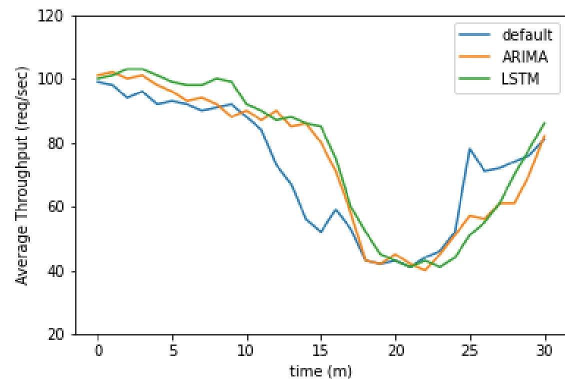

Fig. 14. Average throughput of various models.

TABLE VIII
PERCENTAGE THROUGHPUT INCREASE WITH RESPECT TO THE DEFAULT SYSTEM.

Model	% Increase in Throughput
ARIMA	3.360
LSTM	6.407

Number of Active Pods:

Fig. 15 shows the average number of active pods for various models. Though the proposed model had a greater average number of pods in the low-loaded time frame, it beat the default model over time as shown in Table IX. One of the most likely reasons for this is the prediction model, which forecasts the decline in the number of HTTP requests and then downscales the number of pods. However, because the

default approach relies on real-time metrics of HTTP requests, downscaling is substantially slower. As the average number of pods contributes to energy utilization, we can say the proposed model is more energy efficient than the default system.

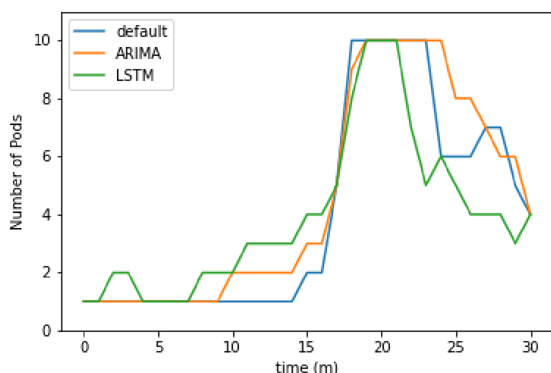

Fig. 15. Average active pods of various models.

TABLE IX
AVERAGE ACTIVE PODS.

Model	Average number of active pods
LSTM	3.903
ARIMA	4.283
Default	4.032

Complexity Analysis:

The proposed framework, which integrates an LSTM-based autoscaler and an ILP-based scheduler, demonstrates a significant departure from Kubernetes' default methods in terms of computational complexity. While Kubernetes' default Horizontal Pod Autoscaler (HPA) and scheduler are efficient with linear and heuristic-based complexities (O(n) and O(nm), respectively), they fail to handle dynamic and high-demand workloads effectively. The default HPA operates reactively, responding to real-time metrics like CPU and memory usage with low computational overhead, but it leads to inefficiencies during workload spikes. On the other hand, the LSTM-based autoscaler uses a predictive approach to proactively scale resources, with slightly higher computational complexity ($O(T(FH+H^2))$) but improved accuracy and reduced latency, making it more suitable for dynamic environments.

Similarly, the default Kubernetes scheduler is efficient for small-scale clusters but lacks global optimization, while the ILP-based scheduler introduces higher complexity (O(2m)) due to its optimization-driven approach. Despite the increased complexity, the ILP scheduler improves performance metrics such as latency, throughput, and resource utilization, with a runtime of 50 milliseconds compared to the default scheduler's 12 milliseconds. Overall, the proposed framework incurs a minor computational overhead due to the LSTM autoscaler and ILP scheduler but achieves superior performance in terms of resource utilization, latency, and throughput. The results validate the trade-off between complexity and efficiency, showing that the framework's predictive and optimization capabilities are essential for real-time deployment in dynamic Kubernetes environments.

V. Conclusion

As more people use the Internet and cloud-based services in their daily lives, the opportunity to scale and improve the services they use grows. This work presented a more efficient and dynamic way for scaling and scheduling compared to the default one in Kubernetes which is a reactive model. The default system uses metrics such as CPU utilization in making austocaling and scheduling decisions. However, we employed a machine learning method, specifically LSTM to build a proactive system that anticipates prospective changes in metrics based on past usage patterns and trends, seasonal effects, and then takes actions based on those predictions. To build a model, we used the Wikipedia Traffic Dataset provided by Google as well as the tesbed dataset. Using this model, we designed a combined approach of autoscaling and scheduling. We evaluated the proposed framework in a 9 node Kubernetes cluster. The results revealed that the combined approach performs better than the default scheduler in terms of response time, throughput, and number of active pods under different traffic scenarios.

As future work, we plan to evaluate the proposed system in a larger environment. Furthermore, we plan to consider multiple parameters in the design of prediction model.

References

[1] K. Senjab, S. Abbas, and N. Ahmed, "A survey of Kubernetes scheduling algorithms," *Journal of Cloud Computing*, vol. 12, no. 1, pp. 1–26, 2023.

[2] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–37, 2022.

[3] I. Ahmad *et al.*, "Container scheduling techniques: A survey and assessment," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 7, pp. 3934–3947, 2022.

[4] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–37, 2022.

[5] S. Xing *et al.*, "A QoS-oriented Scheduling and Autoscaling Framework for Deep Learning," in *Proc. Int. Joint Conf. on Neural Networks (IJCNN)*, Budapest, Hungary, 2019, pp. 1–8, doi: 10.1109/IJCNN.2019.8852319.

[6] Y. Yang and L. Chen, "Design of Kubernetes scheduling strategy based on LSTM and grey model," in *Proc. IEEE 14th Int. Conf. on Intelligent Systems and Knowledge Engineering (ISKE)*, Dalian, China, 2019, pp. 701–707.

[7] T. Menouer, "KCSS: Kubernetes container scheduling strategy," *The Journal of Supercomputing*, vol. 77, no. 5, pp. 4267–4293, 2021.

[8] G. Fan *et al.*, "Multi-objective optimization of container-based microservice scheduling in edge computing," *Computer Science and Information Systems*, vol. 18, no. 1, pp. 23–42, 2021.

[9] Z. Wang *et al.*, "Research and implementation of scheduling strategy in Kubernetes for computer science laboratory in universities," *Information*, vol. 12, no. 1, p. 16, 2021.

[10] J. M. Bernabé, F. J. Quiles, and A. F. Gómez-Skarmeta, "A study of Kubernetes scheduling policies and their relation with application performance," 2021.

[11] B. Adhikari and S. N. Srirama, "Multi-objective accelerated particle swarm optimization for container-based scheduling in IoT-enabled cloud environments," *Journal of Ambient Intelligence and Humanized Computing*, vol. 9, no. 6, pp. 1881–1894, 2018, doi: 10.1007/s12652-018-0864-4.

[12] O. Pozdniakova, A. Cholomskis, and D. Mažeika, "Self-adaptive autoscaling algorithm for SLA-sensitive applications running on the Kubernetes clusters," *Cluster Computing*, pp. 1–28, 2023.

[13] Z. Zhou *et al.*, "AHPA: Adaptive Horizontal Pod Autoscaling Systems on Alibaba Cloud Container Service for Kubernetes," *arXiv preprint arXiv:2303.03640*, 2023.
[14] N.-M. Dang-Quang and M. Yoo, "Deep learning-based autoscaling using bidirectional long short-term memory for Kubernetes," *Applied Sciences*, vol. 11, no. 9, p. 3835, 2021.
[15] A. Zhao *et al.*, "Research on resource prediction model based on Kubernetes container auto-scaling technology," in *Proc. IOP Conf. Series: Materials Science and Engineering*, vol. 569, no. 5, 2019.
[16] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *Proc. IEEE Int. Conf. on Web Services (ICWS)*, 2019, pp. 186–193.
[17] T. Hu and Y. Wang, "A Kubernetes Autoscaler Based on Pod Replicas Prediction," in *Proc. Asia-Pacific Conf. on Communications Technology and Computer Science (ACCTCS)*, 2021, pp. 238–241, doi: 10.1109/ACCTCS52002.2021.00053.
[18] E. Casalicchio and V. Perciballi, "Auto-scaling of containers: The impact of relative and absolute metrics," in *Proc. IEEE 2nd Int. Workshops on Foundations and Applications of Self* Systems (FAS*W)*, 2017, pp. 207–214.
[19] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proc. IEEE 12th Int. Conf. on Cloud Computing (CLOUD)*, 2019, pp. 329–338, doi: 10.1109/CLOUD.2019.00061.
[20] B. Baliś, A. Broński, and M. Szarek, "Auto-scaling of scientific workflows in Kubernetes," in *Proc. Int. Conf. on Computational Science*, 2022, pp. 33–40.
[21] N.-M. Dang-Quang and M. Yoo, "An efficient multivariate autoscaling framework using Bi-LSTM for cloud computing," *Applied Sciences*, vol. 12, no. 7, p. 3523, 2022.
[22] S. Kakade *et al.*, "Proactive Horizontal Pod Autoscaling in Kubernetes using Bi-LSTM," in *Proc. IEEE Int. Conf. on Contemporary Computing and Communications (InC4)*, vol. 1, 2023.
[23] S. Patil, D. G. Narayan, A. Bhat, A. Hungund, and D. M. Patil, "Horizontal Pod Autoscaling in Kubernetes Cluster Using Long Short-Term Memory," in *Data Engineering and Applications: Proc. IDEA 2K22*, vol. 2, pp. 461, 2024.
[24] T. Menouer, C. Cérin, and P. Darmon, "KOptim: Kubernetes Optimization Framework," in *Proc. IEEE Int. Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2024, pp. 900–908.
[25] T. Li, L. Qiu, F. Chen, H. Chen, and N. Zhou, "CAROKRS: Cost-Aware Resource Optimization Kubernetes Resource Scheduler," in *Proc. IEEE Int. Conf. on Cloud Computing and Big Data Analytics (ICCCBDA)*, 2024, pp. 112–120.
[26] W. Gao, Z. Ye, P. Sun, T. Zhang, and Y. Wen, "UniSched: A Unified Scheduler for Deep Learning Training Jobs With Different User Demands," *IEEE Transactions on Computers*, vol. 73, no. 6, pp. 1050–1060, 2024.
[27] V.-I. Luca and M. Eraşcu, "SAGE – A Tool for Optimal Deployments in Kubernetes Clusters," in *Proc. IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom)*, 2023, pp. 45–54.
[28] X. Cheng, E. Fu, C. Ling, and L. Lv, "Research on Kubernetes Scheduler Optimization Based on Real Load," in *Proc. Int. Conf. on Electronic Information Engineering and Computer Science (EIECS)*, 2023, pp. 350–360.
[29] L. Zhang, X. Ji, F. Peng, J. Li, Y. Chen, and K. Zhang, "Flow-Graph Based Optimization of Containerized Resource Scheduling: An Efficient Scheduling Algorithm for Kubernetes Clusters," in *Proc. Int. Conf. on Intelligent Computing and Robotics (ICICR)*, 2024, pp. 78–88.
[30] P. Vishesh *et al.*, "Optimized Placement of Service Function Chains in Edge Cloud with LSTM and ILP," *SN Computer Science*, vol. 6, no. 1, p. 44, 2024.

Somashekar Patil is pursuing a Ph.D. in Cloud Computing at KLE Technological University, Hubballi. He holds an M.Tech. in Computer Networks (2013) and a B.E. in Computer Science (2009). With more than ten years of teaching and research experience and three years in industry, he has worked as an Assistant Professor and as a Senior Consultant at HCL Technologies. His expertise covers cloud computing, software-defined networking (SDN), IoT, load balancing, secure communication, and cross-platform development.

Narayan D. G. is a Professor in the School of Computer Science and Engineering at KLE Technological University, Hubballi, India. He obtained a Ph.D. and M.Tech. from Visvesvaraya Technological University, Belgaum, in 2017 and 2004, respectively, and a B.E. in Computer Science and Engineering from Karnataka University in 2000. With 25 years of teaching and research experience, his interests include cloud computing, cybersecurity, software-defined networks, and blockchain. He is a reviewer for many SCIE-indexed journals and has published more than 80 papers in conferences and journals. ORCID: 0000-0002-2843-8931.

Ajeya Bhat completed his B.Tech. in Computer Science in 2023, focusing on intelligent workload scheduling in cloud environments. His early research involved optimizing resource allocation on virtual machines, followed by heterogeneous Kubernetes cluster scheduling. He emphasized reinforcement learning for predictive and adaptive workload management. Post-graduation, he has been engaged in infrastructure engineering, building scalable and reliable PaaS solutions.