Proactive and Autonomic IoT Service Auto-scaling in Constrained Edge Computing Environments

by

Ahmed BALI

MANUSCRIPT-BASED THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
Ph.D.

MONTREAL, "JULY 24, 2023"

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

FOREWORD

This doctoral thesis consists of a compilation of articles, with three complementary solutions proposed for efficient and adaptive IoT service deployment on resource-limited edge devices, primarily using AI-based techniques. These articles are presented in their original published and submitted form, with only minor modifications to ensure compliance with the thesis template. While each article addresses different aspects of the research problem, they are closely interrelated and contribute to the overall goal of this thesis.

In addition to the incorporated articles, a dedicated section provides a comprehensive overview of the background and literature review on the research topic. The thesis concludes with a summary of the main findings and contributions of our work, along with potential avenues for future research in this field.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest appreciation and gratitude to my advisor, Professor Abdelouahed GHERBI, for his unwavering support, guidance, and mentorship throughout my Ph.D. journey. His expertise, encouragement, and patience have been instrumental in making this research possible. I am truly grateful for his mentorship, friendship, and invaluable contributions to my academic and personal growth.

I extend my heartfelt thanks to the members of my thesis committee, Professor Kim Khoa Nguyen, Professor Kaiwen Zhang, and Professor Mohamed Hamdaqa, for honoring me with their participation in my thesis jury and their efforts in reviewing my thesis.

I am grateful to my parents, Laid and Nafissa, for their endless love, support, and encouragement throughout my academic journey. Their unwavering belief in me and their sacrifices have been the driving force behind my success. They instilled in me the values of hard work, perseverance, and dedication, which have been vital to my accomplishments. I am thankful for their unwavering support.

I express my deepest appreciation to my wife, Anfal. Her unwavering support and encouragement throughout my PhD journey have been invaluable. Her patience, understanding, and love were the pillars that sustained me through the ups and downs of this challenging journey. Her sacrifice and dedication to our family made it possible for me to focus on my studies and achieve this significant milestone. Without her, this achievement would not have been possible, and for that, I am forever grateful.

Finally, I am grateful to all my beloved family for their constant support and encouragement, as well as my friends and colleagues in the LASI lab for their camaraderie and insightful discussions beyond research.

Mise à l’échelle automatique proactive des services IoT dans les environnements contraints de l’informatique périphérique

Ahmed BALI

RÉSUMÉ

L’internet des objets (IdO) s’est considérablement développé avec l’utilisation généralisée de capteurs dans divers domaines de la vie moderne, tels que les soins de santé et la construction. Le nombre croissant de dispositifs (objets) connectés génère des quantités massives de données qui ont un impact négatif sur les performances du système, en particulier sur le temps de réponse, qui est très important pour les applications sensibles à la latence. L’informatique en périphérie (Edge computing), qui délègue les tâches de l’informatique en nuage (Cloud) à des nœuds en périphérie plus proches des sources de données, a été adoptée pour atténuer ce problème. Cependant, les dispositifs IoT en périphérie sont limités en ressources et opèrent dans un environnement hautement hétérogène, ce qui est traité en utilisant des conteneurs (Containers) comme technique de virtualisation légère pour déployer des microservices.

La limitation des ressources constitue un défi important pour le déploiement de services à la périphérie des réseaux IoT. Une gestion efficace des ressources des dispositifs périphériques est essentielle pour répondre aux exigences des utilisateurs (par exemple, le temps de réponse) et optimiser l’utilisation des ressources, ce qui augmente la capacité de déploiement. La mise à l’échelle automatique des services (auto-scaling) est une solution interessante qui améliore l’utilisation des ressources en ajustant dynamiquement le nombre d’instances de services en fonction de la charge de travail. Cependant, les outils de la mise à l’échelle automatique actuels, tels que Kubernetes, s’appuient principalement sur des approches réactives basées sur des seuils, qui sont difficiles à configurer et moins efficaces pour traiter des charges de travail complexes. Ces approches réactives entraînent un gaspillage dû au surprovisionnement des ressources et une dégradation des performances lors de la libération des ressources. D’autre part, la mise à l’échelle automatique proactive qui anticipe la charge de travail future nécessite encore une amélioration en termes de précision des prévisions afin d’optimiser les performances et l’efficacité du système.

L’objectif principal de ce travail est de proposer une approche pour le déploiement de services IoT à la périphérie qui s’adapte aux exigences de performance, à la disponibilité des ressources et à la dynamique de la charge de travail. Notre approche de l’adaptabilité du déploiement des services est basée sur la boucle MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge). Par conséquent, chaque phase de la boucle présente une étape dans la réalisation de chaque contribution de notre travail. Chaque étape doit inclure une revue de la littérature, l’étude et l’utilisation des techniques existantes et leur amélioration ou même la proposition de nouvelles techniques, l’implémentation, l’expérimentation et la validation.

Notre première contribution vise à remédier aux limites des solutions basées sur les seuils. En se basant sur le modèle de déploiement que nous proposons, encodé dans la logique des prédicats, notre solution génère automatiquement des règles pour différentes phases, telles que l'analyse et la planification.

La deuxième contribution propose une solution proactive de mise à l'échelle automatique. Elle utilise un modèle des réseaux de neurones de type LSTM (Long Short-Term Memory), mémoire à long terme et court terme, pour la prévision de la charge de travail grâce à sa précision et à sa vitesse de prédiction. Pour améliorer davantage la précision, notre approche ajoute une phase de caractérisation (featurization) automatique qui extrait des caractéristiques des séries temporelles de données sur la charge de travail. Elle aborde également, de manière originale, le problème de l'oscillation causé par les actions de mise à l'échelle fréquemment générées. Les caractéristiques extraites par la phase de la caractérisation sont utilisées comme grille pour atténuer le problème d'oscillation.

La troisième contribution vise à améliorer encore davantage la précision de la prévision de la charge de travail tout en tenant compte de la limitation des ressources des dispositifs IoT. Notre approche proposée d'apprentissage en ensemble dynamique (Dynamique Ensemble learning) réduit efficacement les valeurs aberrantes et maintient une précision élevée dans la prévision de la charge de travail et les performances d'auto-scaling.

Dans toutes nos contributions, nous avons validé notre approche à l'aide de diverses implémentations. Nous avons également mené les expériences nécessaires en comparant nos résultats avec des travaux connexes. Dans l'ensemble, la validation montre la faisabilité et l'efficacité de nos différentes contributions. Enfin, nous avons mis en évidence plusieurs perspectives de recherche et proposé des travaux futurs potentiels dans le cadre de l'étude présentée.

Mots-clés: Internet des objets (IoT), informatique périphérique, virtualisation, conteneur, Mise à l'échelle automatique, boucle MAPE-K, LSTM, atténuation des oscillations, apprentissage automatique, prévision des séries temporelles

Proactive and Autonomic IoT Service Auto-scaling in Constrained Edge Computing Environments

Ahmed BALI

ABSTRACT

The Internet of Things (IoT) has greatly developed with the widespread use of sensors in various areas of modern life, such as healthcare, and construction. The increasing number of connected devices generates massive amounts of data that negatively impact system performance, especially the response time, which is very important for latency-aware applications. Edge computing, which delegates cloud tasks to edge nodes closer to data sources, has been adopted to alleviate this issue. However, IoT devices at the edge are resource-constrained and operate in a highly heterogeneous environment, which is addressed by using containers as a lightweight virtualization technique for deploying microservices.

The limitation of resources poses a significant challenge to deploying services at the edge of IoT networks. Effective resource management of edge devices is essential to meet user requirements (e.g., response time) and optimize resource usage, which increases the deployment capacity. Service auto-scaling is an interesting solution that improves resource utilization by dynamically adjusting the number of service instances to match the workload. However, current auto-scalers, such as Kubernetes, mostly rely on threshold-based reactive approaches, which are difficult to configure and less efficient in dealing with complex workloads. These reactive approaches result in wastage due to over-provisioning of resources and performance degradation during resource releases. On the other hand, proactive auto-scaling that anticipates future workload still needs improvement in forecasting accuracy to optimize system performance and effectiveness.

The main objective of this work is to propose an approach for deploying IoT services at the edge that is adaptive to performance requirements, resource availability, and workload dynamics. Our approach to service deployment adaptability is based on the MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge) loop. Therefore, each phase of the loop presents a step in the realization of each contribution of our work. Each step must include a literature review, study, and use of existing techniques and their improvement or even proposal of new techniques, implementation, experimentation, and validation.

Our first contribution aims at addressing the limitation of threshold-based solutions. Based on our proposed deployment model, encoded in predicate logic, our solution automatically generates rules for different phases, such as analysis and planning.

The second contribution proposes a proactive auto-scaling solution. It uses Long short-term memory (LSTM) for workload forecasting thanks to its accuracy and prediction speed. For further accuracy improvement, our approach adds an automatic featurization phase that extracts features from time-series workload data to improve the workload prediction accuracy. It also addresses, in an original way, the oscillation issue caused by the frequently generated scaling actions. The extracted features issued by the featurization phase are used as a grid to mitigate the oscillation issue.

The third contribution aims to improve further the accuracy of workload forecasting while considering the resource limitation of IoT devices. We have investigated a variety of known prediction algorithms according to the metrics of accuracy and prediction time. Our proposed Dynamic Ensemble learning approach effectively reduces outliers and maintains high workload forecasting accuracy and auto-scaling performance.

In all our contributions, we have validated our approach with several proof-of-concept implementations. We also conducted the necessary experiments by comparing our results with related work. Overall, the validation shows the feasibility and effectiveness of our different contributions. Finally, we highlighted several research perspectives and proposed potential future work within the scope of the presented study.

Keywords: Internet of Things (IoT), edge computing, virtualization, container, Auto-scaling, MAPE-K loop, LSTM, Oscillation mitigation, machine-learning, time series forecasting

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

ARIMA	Autoregressive Integrated Moving Average
BN	Bayesian Network
ETS	École de Technologie Supérieure
IoT	Internet of Things
IT	Information Technology
LSTM	Long Short-Term Memory
OS	Operating System
PC	Personal Computer
PMs	Power Management Systems
RF	Random Forest
RFID	Radio Frequency Identification
SVR	Support Vector Regression
VM	Virtual Machine

INTRODUCTION

0.1 Context and motivation

The Internet of Things (IoT) has brought about a technological revolution that has revolutionized various aspects of our daily lives, including health, transportation, and work. The innovative vision of IoT has made its applications ubiquitous, playing an essential role in several areas. These applications primarily rely on networks of small devices integrated into our surroundings, such as environmental, healthcare, and industrial sensors. With constant development and deployment of these devices, billions of connected devices are now present on the Internet (Evans, 2011).

To meet the demand for cost-efficient and responsive IoT applications, a portion of IoT services is being deployed at the network edge. Edge computing, a distributed computing model that brings computation and data storage closer to the data source and the devices where it is needed, is increasingly being used to support this deployment. In this topology, sensors transmit their data to an intermediary device that handles processing tasks like data pre-processing and aggregation (Venticinque & Amato, 2019). This approach ensures that only the relevant observations or results are communicated to the cloud, avoiding the transmission of a significant amount of data, which can reduce network traffic and enhance network performance. The need for real-time data processing is particularly critical in certain use cases such as IoT healthcare applications (Devarajan, Subramaniyaswamy, Vijayakumar & Ravi, 2019), where immediate action is necessary, making processing close to the resources of utmost importance.

To better understand the benefits of edge computing in IoT systems, consider a smart building where cameras are used for surveillance and control purposes. In this scenario, the camera captures real-time video footage that needs to be analyzed and processed to detect specific events or objects, such as intruders or unauthorized activities. Traditionally, this data would be sent to the cloud for processing and analysis. When it comes to surveillance and control applications, real-time responsiveness is essential. By processing the data locally at the edge, the camera can analyze the video feed in real-time and trigger immediate actions, when necessary, without relying on cloud-based processing, which may introduce latency due to data transmission and processing time.

Processing the data locally also reduces the burden on the network infrastructure. Video data is typically large and bandwidth-intensive, which can lead to network congestion and increased latency when transmitting to the cloud. By processing the data at the edge, only the relevant information or alerts need to be sent to the cloud, significantly reducing the amount of data transmitted and relieving the network traffic. This example demonstrates the importance of processing data locally at the edge instead of relying solely on cloud-based processing. Local processing improves responsiveness by analyzing the data in real-time, avoids network congestion and latency, and enhances privacy and security. It showcases how edge computing can enable efficient and effective IoT deployments, where IoT gateways, which can be resource-limited edge devices, play a crucial role in ensuring immediate action and timely responses in critical situations.

IoT devices at the network's edge operate in a highly heterogeneous environment due to the diverse processing capabilities, communication protocols, and software requirements. Containers, a lightweight virtualization technique, have emerged as a solution to address the heterogeneity challenge. By encapsulating application components and their dependencies, containers can facilitate the deployment of microservices at the edge level, making it easier to manage and scale applications while reducing the overhead associated with deploying traditional virtual machines. Moreover, container orchestration techniques like Swarm (Swarm, 2022) facilitate resource sharing among IoT edge devices that belong to the same cluster and provide ways for containers to communicate across virtual networks. Additionally, Kubernetes (kubernetes, 2022b) introduces the concept of a pod, which is a group of one or more containers that share network and storage resources and follow certain operating rules. A pod is the basic unit of deployment. Within this context, service deployment is achieved by deploying a set of replicas (i.e., containers or pods) on the available machines that are grouped into clusters. Overall, the orchestration techniques provide efficient means of communication and resource allocation for managing IoT devices.

0.2 Problem Statement

Deploying services to devices at the edge of IoT networks is a promising paradigm that can offer significant benefits, particularly in reducing service response time. However, this approach is often limited by resource constraints, which can restrict the ability to deploy services effectively. Consequently, these resource constraints can limit the ability to take full advantage of this novel paradigm, highlighting the critical need for efficient edge device resource management. Such resource management is necessary to meet user requirements, such as response time while reducing resource usage and increasing the deployment capacity for other microservices.

Service auto-scaling is an approach that addresses this need by improving resource utilization in response to changing demand (i.e., workload). There are essentially two types of autoscaling, vertical autoscaling and horizontal autoscaling (Al-Dhuraibi *et al.*, 2017). The distinction between these two types of scaling stems from the manner in which computing resources are added to the infrastructure. In vertical autoscaling, computing power is added to existing replicas/nodes. In contrast, horizontal autoscaling increases a system's capacity by adding more replicas (e.g., containers) to the environment, allowing for processing and memory load sharing across multiple devices. In the edge computing context, resources are often limited, which makes having a mechanism for increasing and decreasing computational resources (i.e., vertical auto-scaling) on the same node less useful or even unrealistic. Therefore, horizontal auto-scaling becomes more suitable by dynamically changing the number of replicas (e.g., containers or pods) to distribute the processing load among devices that constitute a so-called cluster. Increasing the number of service replicas increases the use of available computing resources. Conversely, reducing the number of service replicas increases the availability of computing resources, which can then be used for other deployed services.

Resource management is crucial at the edge level, where devices often face limited resources. Achieving a balance between the use of computing resources and meeting user requirements, such as response time, is essential. To achieve this, it is necessary to continuously and dynamically maintain a number of replicas that neither exceeds the demand (overprovisioning) nor fall short of it (under-provisioning). While increasing the number of replicas can improve the service's responsiveness (reducing latency), it also increases the usage of computational resources.

Reactive threshold-based approaches are commonly used by most current auto-scalers, including industrial solutions like Kubernetes HPA, Google Cloud Platform, Amazon EC2, and Oracle Cloud, due to their ease of implementation, as observed in Kovács (2019); Nguyen, Yeom, Kim, Park & Kim (2020); Taherizadeh & Stankovski (2019). These approaches react to the current system workload based on preconfigured thresholds. However, selecting appropriate thresholds can be a challenging task, especially when dealing with complex workloads, as noted in Imdoukh *et al.* (2019). Additionally, these reactive approaches lack proactivity, which limits the system's ability to adapt to the operational environment appropriately. The frequent changes in the requested workload, such as HTTP requests, make it challenging for reactive approaches to respond quickly, which can lead to performance issues.

The reactive approach responds to the current workload by adapting the system, such as provisioning new containers or virtual machines. However, this adaptation process takes time, which may not be suitable for the subsequent workload. This delay can result in a lag between the actual workload demand and the system's ability to scale or adjust accordingly.

Proactive auto-scaler anticipate future needs to adapt the system, whereas reactive auto-scalers react following workload changes. Thus, one key factor that makes behavior proactive is the ability to forecast future workloads and adapt the system accordingly at the right time. In proactive scaling, an algorithm is used to predict future workloads based on historical data (Lorido-Botran, Miguel-Alonso & Lozano, 2014).

There are two common categories of time series data analysis and forecasting methods used in the literature. The first category includes algorithms based on statistical time series analysis, such as ARIMA, which have been applied in various studies for workload forecasting, including Lorido-Botran *et al.* (2014); Sangpetch, Sangpetch, Juangmarisakul & Warodom (2017); Calheiros, Masoumi, Ranjan & Buyya (2014); Roy, Dubey & Gokhale (2011); Kan (2016); Li & Xia (2016); Ciptaningtyas, Santoso & Razi (2017); Meng, Rao, Zhang & Hong (2016). The second category involves using machine learning algorithms, which are increasingly applied to time series analysis for workload forecasting, as seen in studies such as Sangpetch *et al.* (2017); Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021). Deep learning-based solutions, including neural network (ANN) and LSTM algorithms, have also been used in studies such as Calheiros *et al.* (2014); Goli, Mahmoudi, Khazaei & Ardakanian (2021). However, it should be noted that most of these studies are intended for the cloud environment and do not consider resource limitations, which may make applying these techniques in edge computing challenging. For instance, in Imdoukh *et al.* (2019), authors mention that statistical approaches, such as ARIMA, are known to be slow in responding to dynamic workload demands and can suffer from resource overuse.

Furthermore, proactive auto-scalers using time series data analysis heavily rely on prediction accuracy (Doan, Zaharie & Petcu, 2019). Several factors, including the workload pattern, history windows, machine-learning models, and prediction horizon, affect the accuracy of predictions (Lorido-Botran *et al.*, 2014). Thus, to enhance auto-scalers' performance, it is crucial to propose solutions that improve prediction accuracy, enabling them to generate more appropriate actions, either scaling up or down, in response to the real workload.

Another important aspect to consider is the continuous generation of actions by the auto-scaler, which can lead to frequent changes in the number of replicas and result in oscillation issues that waste resources. For example, at time t, the auto-scaler may add a recently released resource from time t-1, or vice versa. Reactive auto-scaling can exacerbate to the oscillation of the number of replicas due to its delayed response to the current workload. As the system adjusts its resources based on the current workload, there is a lag between the workload change and the system's adaptation. This delay can result in an inadequate response to subsequent workloads, leading to undesirable consequences such as increased costs from over-provisioning or decreased performance from under-provisioning. Therefore, it is essential to consider oscillation mitigation as an important functionality in the auto-scaling process. By reducing the number of unnecessary changes in the number of replicas, system performance can be improved, and resource wastage can be reduced.

Unfortunately, oscillation mitigation has not received sufficient attention in the literature. While some works, such as Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021), have proposed solutions based on the cooling down principle. Simply introducing a delay before scaling down in response to decreasing workload volume, as proposed by this strategy, may not be enough to effectively mitigate oscillation. Its effectiveness depends on optimizing the cooling down period, which can be challenging. A longer period can lead to more over-provisioning, while a shorter period may reduce the efficiency of oscillation mitigation. Therefore, additional strategies are needed to effectively mitigate oscillation in auto-scaling systems.

Container-based autoscaling solutions are still an open issue that needs to be addressed (Cardenas, 2018). Designing and implementing an efficient and adequate auto-scaler for containerized services is challenging due to various factors, including dynamic workload characteristics, resource constraints, and the distributed nature of IoT edge nodes.

0.3 Research objectives

The research problem of this work pertains to the deployment of services on edge devices, which can offer significant benefits by reducing service response time. However, such deployments are often limited by resource constraints of edge devices, which hinder the ability to deploy services. Therefore, this research study aims to investigate solutions to improve the deployment of container-based IoT services on devices within the same edge cluster. Specifically, we aim to:

- Investigate the use of the MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge) loop for automatic service deployment and define a deployment model as shared knowledge. We will propose and test an automatic solution that responds to the limitation of reactive threshold-based approaches, which often suffer from the difficulty of defining threshold values.
- Propose and validate a novel time-series data pre-processing technique to improve the prediction accuracy of workload forecasting and test its effectiveness in a service auto-scaling solution. We will also investigate the possibility of using the same pre-processing technique for oscillation mitigation.
- Investigate algorithms and techniques to improve the accuracy of workload forecasting, taking into consideration the resource usage aspect, and test their impact on the auto-scaling process in the context of edge computing. We will evaluate the performance/resource usage ratio of various forecasting techniques, ranging from widely adopted techniques such as ARIMA and LSTM to less resource-intensive techniques such as SVR, RF, and BN. Finally, we will investigate, even improve, the feasibility and utility of combining these forecasting techniques.

0.4 Methodology

To achieve the aforementioned research objectives, this work follows a methodology inspired by the MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge) loop. It is worth mentioning that the MAPE-K loop is extensively used to make systems autonomous. The Monitor step allows for system monitoring, the Analyze step analyzes the system's state to determine if intervention is needed, the Planning step generates an intervention plan, and the final Execute step implements the generated plan. To ensure system autonomy, the process is iterative.

In terms of the research methodology, each activity in the MAPE-K loop represents a step in achieving an objective, where:

- The Monitor step includes a literature review and problem formulation. As this process is iterative, this step may also include a review of the proposed solution currently in development and validation;
- The Analyze step analyzes the nature of the problem and the intended solution. Depending on the nature of the problem, the solution may be a proposal for a new process, the use of a technique, or even its improvement to achieve the research objective. This step may include a study and learning of existing techniques and tools, such as learning algorithms and datasets;
- The Planning step involves designing the proposed approach in various forms, such as formal models, mathematical equations, algorithms, approaches, and strategies. The output of this step may include elements of fine granularity, such as the experimentation protocol;
- The Execution step involves implementing the proposed approach from the Planning step, conducting experiments, and validation;

In this methodology, the central element of the MAPE-K loop, Shared Knowledge, essentially represents the result of the documentation activity that takes place with all the loop steps. The output of the Monitoring step allows for the development of the problem description and formulation as well as the literature review. The documentation of the Analyze step further develops the problem and research gap description as well as a description of the selected techniques. The documentation of the Planning step describes the proposed approach, including different mathematical equations and algorithms. In the Execution step, the documentation logs the test results and their discussions.

As the MAPE-K process advances and the number of iterations increases, the documentation becomes increasingly rich and complete. This facilitates the process of compiling a research article specific to the objective being addressed.

0.5 Contributions

The proposed contributions were guided by the limitations of the existing approaches, as presented in the problem statement section. These limitations generally rely on the following elements: resource-consuming, automatic processing as the automatic setting of thresholds, workload forecasting accuracy, and oscillation mitigation.

The main objective of this work is to propose solutions for deploying containerized services at the edge level that meet the requirements of resource limitations, environmental heterogeneity, dynamic workload, and quality of service. To address the limitations of existing approaches, our contributions are articulated around the following requirements:

- Developing resource-efficient solutions that can handle dynamic workloads and ensure the quality of service without the need for resource-intensive processing;
- Automatic processing to maintain the deployed system in a suitable state. This automatic aspect includes the configuration task, such as the setting of rule thresholds;
- Investigating and even improving forecasting techniques to improve workload forecasting accuracy;

- Handling oscillation mitigation issues to ensure stable and efficient operation of containerized services;

Overall, our contributions were motivated by the need to explore alternative techniques that can improve the effectiveness and efficiency of containerized service deployment at the edge.

Our first contribution focuses on automating the deployment process to ensure service auto-scaling and fair distribution of containerized services on devices in the same edge cluster. To achieve this, we follow the MAPE-K (Monitor-Analyze-Plan-Execute-Knowledge) loop framework, which provides a systematic approach for managing and controlling the deployment process. A key aspect of our solution is the use of shared knowledge, which refers to the deployment and rule models we have developed.

The deployment model captures the different system elements involved in the deployment process, including the devices and services and their corresponding properties. The rule model, which builds on the deployment model, defines specific rule categories for each autoscaling process step, including evaluation, decision, scale, and verification. Both models are formulated in predicate logic form, which enables using algorithms and strategies to automatically determine thresholds and ensure efficient scaling.

We assessed the effectiveness of our approach based on various criteria, such as CPU and memory usage, network traffic, and service response time. Our results demonstrate that the proposed approach can efficiently adapt the system performance by adjusting the number of replicas to meet the service performance requirements and ensure the optimal utilization of system resources.

Our second contribution proposes an improved proactive service auto-scaling approach that aims to address the research gap in workload forecasting accuracy and oscillation issues. Similar to our first solution, it follows the MAPE-K loop to automatically maintain the required system performance with optimal resource utilization to cope with dynamic workload changes.

To improve the forecasting model accuracy, we propose to add a data featurization operation during the pre-processing data phase. Our proposed feature extraction is inspired by Japanese candlesticks, widely used in the trading domain. The candlestick representation provides an abstraction that allows traders to understand and predict stock evolution.

To mitigate oscillation, we propose a grid-based approach that benefits from the data and features used for workload prediction improvement. It is also based on an economic concept, namely a grid technique. Therefore, we call it grid-based oscillation mitigation, where the grid lines are determined by the features extracted by our featurization operation used to improve forecasting accuracy. Our original oscillation mitigation mechanism has the advantage of being parameterless compared to related work techniques.

We evaluate our approach using the WorldCup'98 (Arlitt & Jin, 2000) and NASA (Dang-Quang & Yoo, 2021) datasets, widely used in the literature on auto-scaling systems. The results show a considerable improvement in forecasting accuracy when transforming the original univariate time series data into a multivariate time series using our feature extraction approach. The experiments reveal that by combining our grid-based oscillation mitigation with a commonly used strategy, cool-down, outperforms the state-of-the-art techniques.

The third contribution addresses the complexity of designing and implementing an efficient auto-scaler for containerized services on the Edge level, which is impacted by dynamic workload characteristics and resource limitations. Consequently, our work focuses on addressing the forecasting accuracy issue while considering resource usage, as computing on the Edge is resource-limited. We propose a methodology to find suitable methods and techniques for workload forecasting. First, we investigate the convenience and accuracy of common time series data analysis and forecasting techniques such as ARIMA and LSTM. Second, we evaluate less resource-intensive techniques such as SVR, RF, and BN, considering the performance/resource usage ratio. Third, we evaluate the use of Ensemble learning to improve prediction accuracy, addressing the optimization problem under the constraint of the number of forecasting algorithms that can be applied. We formulate this problem as the Knapsack problem, which is an NP-complete problem, and propose a heuristic technique to reduce the solution complexity. Our proposed dynamic Ensemble learning approach effectively handles predicted data outliers and maintains high accuracy and stability in workload forecasting.

As a complement to our first contribution, we conducted a separate study (Ahmed, Seghir, Al-Osta & Abdelouahed, 2019) to evaluate the benefits of service deployment based on lightweight virtualization technologies, such as Docker containers. Containerization offers advantages such as reusability and easy duplication of deployments. Additionally, containers can help alleviate the heterogeneity issue in IoT environments by enabling deployment and communication between different software modules, regardless of the underlying framework. Containers can be customized to cope with resource constraints on the target node. Container-based solutions can also take advantage of orchestration techniques such as Swarm and Kubernetes, which facilitate resource sharing between IoT devices and provide a means of communication between containers across virtual networks. Our experiments demonstrate the potential of virtualization tools in resource-constrained contexts such as edge computing.

Another paper (Bali, Al-Osta, Ben Dahsen & Gherbi, 2020) addresses the issue of overload in monitoring and auto-scaling solutions, considering the resource limitations of Edge computing. We argue that existing monitoring tools can consume a significant amount of resources, which is often overlooked in the auto-scaling literature. To demonstrate this, we conducted an experiment that showed the high resource consumption of the Cadvisor monitoring tool (cAdvisor, 2022) when deployed on a resource-limited device such as the Raspberry Pi 3.

Our proposed monitoring approach relies on a rule-based model to evaluate the significance of measured data metrics, such as CPU utilization exceeding 80%, before forwarding them to a higher-level cluster node component for data aggregation. This mechanism helps reduce the volume of measured data to be reported, reducing the computing workload required to process this data. Our rule model features dynamic rule updating, including automatically set threshold ranges. This automatic update of monitoring rules is based on the analysis of data metrics collected from different cluster nodes, using a data mining technique, namely, the association rule (Bali *et al.*, 2020). Our evaluation shows a significant reduction in the communication volume of monitoring metrics, which can translate into reduced resource consumption.

It is important to note that our last two contributions are not included in this manuscript, as they were published separately as conference papers. Additionally, while we have other publications related to the context of our work, but do not directly address our research problem, in the following publication section, we will provide a comprehensive list of all our relevant publications.

0.6 Publications

The publications are organized into two categories: Articles and Communications. They are listed in chronological order based on their submission date.

0.6.1 Articles

- Al-Osta Mahmud, Ahmed Bali, and Abdelouahed Gherbi. "Event driven and semantic based approach for data processing on IoT gateway devices." Journal of Ambient Intelligence and Humanized Computing 10 (2019): 4663-4678;
- Bali, A., Al-Osta, M., Ben Dahsen, S., & Gherbi, A. (2020). Rule based auto-scalability of IoT services for efficient edge device resource utilization. Journal of Ambient Intelligence and Humanized Computing, 11, 5895-5912;

- Elrotub, M., Bali, A., & Gherbi, A. (2021). Sharing VM resources with using prediction of future user requests for an efficient load balancing in cloud computing environment. International Journal of Software Science and Computational Intelligence (IJSSCI), 13(2), 37-64.
- Bali, Ahmed, Gherbi, A, and Yassine El Houm., Improved Forecasting based on Data Featurization for Proactive Service Auto-Scaling. Submitted to: Journal of King Saud University - Computer and Information Sciences. Minor revisions.
- Bali, Ahmed and Gherbi, A, Proactive IoT Service Auto-Scaling for an Efficient Edge Resource Utilization: Methods and Improvements. Submitted to: Future Generation Computer Systems. Under Review.

0.6.2 Communications

- Al-Osta, M., Ahmed, B., & Abdelouahed, G. (2017). A lightweight semantic web-based approach for data annotation on IoT gateways. Procedia computer science, 113, 186-193;
- Bali, A., Al-Osta, M., & Abdelouahed, G. (2017). An ontology-based approach for IoT data processing using semantic rules. In SDL 2017: Model-Driven Engineering for Future Internet: 18th International SDL Forum, Budapest, Hungary, October 9–11, 2017, Proceedings 18 (pp. 61-79). Springer International Publishing;
- Ahmed, B., Seghir, B., Al-Osta, M., & Abdelouahed, G. (2019). Containe based resource management for data processing on iot gateways. Procedia Computer Science, 155, 234-241;
- Bali, A., & Gherbi, A. (2019, December). Rule based lightweight approach for resources monitoring on IoT edge devices. In Proceedings of the 5th International workshop on container technologies and container clouds (pp. 43-48).

0.7 Thesis Organization

As this work is article-based, this thesis is structured as follows: Chapter 1 provides a comprehensive literature review and general background on the research topic. Chapters 2, 3, and 4 detail each publication in chronological order. Finally, we conclude the thesis by summarizing the main findings and contributions of our work and discussing their implications for future research in the field.

CHAPTER 1

BACKGROUND AND LITERATURE REVIEW

This chapter provides a comprehensive literature review of the state-of-the-art deployment of IoT services on edge devices, serving as a foundation for our research contributions in this area. We explore different aspects of deploying IoT systems on edge devices, including virtualization, auto-scaling, and workload forecasting, examining each aspect in a separate section. Each section starts with a brief overview of the relevant knowledge background, followed by a review of existing related work to contextualize our research. Finally, we position our contributions by identifying gaps and limitations within the existing literature, highlighting the originality and significance of our proposed approach.

1.1 Virtualization techniques for the Edge computing

The potential benefits of containers for application deployment and management have garnered significant attention in academic and industrial research communities. The section is divided into three subsections. In the following subsections, we review existing literature on utilizing containers for service deployment and resource management in edge computing. The first subsection provides a brief background on virtualization and its use in edge computing. The second subsection discusses the related work on using containers for service deployment and resource management in the context of edge computing, considering the unique challenges and opportunities presented by this environment. Finally, the third subsection describes the positioning of our approach within the existing literature.

1.1.1 Brief background

Before we review virtualization techniques, this section briefly overviews some fundamental concepts related to the topic.

1.1.1.1 Internet of Things

Kevin Ashton first introduced the concept of the Internet of Things (IoT) in 1999, which referred to a tracking system that utilized RFID tags connected to the Internet and attached to physical objects. Over time, this concept has evolved to encompass a broader range of interconnected devices and systems that enable communication and data exchange between physical objects and digital networks. This development has led to new applications and services that leverage the vast amounts of data generated by connected devices, thereby improving efficiency, enhancing decision-making, and enabling new business models across various industries and domains, such as healthcare, transportation, and manufacturing. Consequently, IoT has emerged as a significant area of research and innovation with crucial implications for the future of technology and society.

IoT architecture design is a crucial process in the development of IoT systems, and several factors influence this process, including scalability, interoperability, data storage reliability, and Quality of Service (QoS), (Wu, Lu, Ling, Sun & Du, 2010; Gubbi, Buyya, Marusic & Palaniswami, 2013). Several proposals have been presented in the literature to address this process, including different IoT architectural views, such as Mashal *et al.* (2015); Gubbi *et al.* (2013); Wu *et al.* (2010). Typically, these proposals rely on a three-layer architecture comprising Perception, Network, and Application layers.

The perception layer, or the device layer, consists of sensors and actuators that run various functionalities to acquire data like temperature, humidity, location, weight, and pressure. The network layer maintains bidirectional communication between the perception and application layers by assigning unique addresses to all connected objects. This layer may incorporate various short and long-distance communication protocols, including Bluetooth, ZigBee, WiFi, and Lora, to facilitate the seamless transmission of information. The role of the application layer is to provide customized services to customers upon requests.

This architecture can leverage cloud and edge computing to enhance its ability to offer advanced services that cater to users' needs. Cloud computing enables the IoT application to offer services that require large-scale data processing and analysis. On the other hand, by utilizing edge computing, the application IoT can perform data processing and analysis at the edge of the network, reducing latency and enhancing the responsiveness of the system.

1.1.1.2 Edge computing

The growth in IoT networks and the exponential growth of associated data generated have created new challenges in meeting the quality of service (QoS) requirements of various IoT applications. The inadequacy of traditional cloud computing-based solutions in fulfilling the time-sensitive and context-aware service needs of IoT applications has led to the emergence of edge computing. It involves extending cloud capabilities to the network edge, closer to the data source, to address these requirements. As a result, edge computing has become a crucial element in IoT solutions, especially for handling time-sensitive applications and minimizing the amount of data transmitted to the cloud. By adopting edge computing, IoT solutions can achieve lower latency and higher bandwidth, enabling faster and more efficient data processing at the network edge (Khan, Ahmed, Hakak, Yaqoob & Ahmed, 2019).

The various levels of Edge computing are depicted in Figure 1.1, encompassing devices ranging from low-resource to high-performance servers in terms of computational capacity.

It is worth noting that our study in this work specifically targets resource-limited devices, including IoT devices and gateways.

IoT devices: At the lowest level of IoT systems, IoT devices or sensor nodes comprise limited resources, such as microcontrollers and sensors. Sensors measure environmental parameters such as temperature and gas detection, and their accuracy is crucial for the overall performance of IoT systems (Poongodi, Rathee, Indrakumari & Suresh, 2020; Yu *et al.*, 2017). They detect changes and events in the environment, serving as the digital backbone of IoT systems and relaying information to higher layers for analysis and storage.

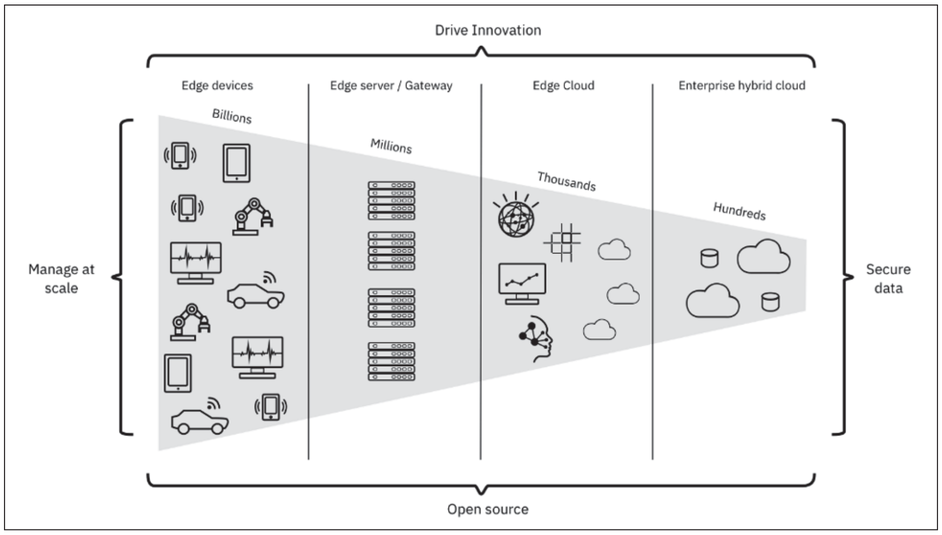


Figure 1.1 Levels of Edge computing and IoT
(Ashok & Christine, 2023)

Gateway devices: In the context of IoT, gateway devices play a crucial role in connecting sensor nodes to cloud services or other remote networks. They can collect, aggregate, and partially process sensory data from multiple nodes, as well as receive controlling instructions and perform decision-making tasks. Recent improvements in gateway devices, which have more computing resources than IoT devices (i.e., sensor nodes), have fueled the trend toward shifting computing tasks to the network edge. By assigning data preprocessing tasks to IoT gateways, system responsiveness can be improved for time-sensitive use cases (Yu *et al.*, 2017; Al-Fuqaha, Guizani, Mohammadi, Aledhari & Ayyash, 2015).

1.1.1.3 Virtualization techniques

Virtualization is a technology that enables the creation of useful IT services using resources typically tied to hardware. It allows the full capacity of a physical machine to be utilized by distributing it among multiple users or different environments (Hat, Accessed 2023). The virtualization of systems has significantly evolved in recent years, providing system architects and developers with a plethora of tools to leverage.

Virtualization and containerization are two widely used techniques for hosting applications in a computing system. Virtualization involves creating virtual versions of computing resources, while containerization involves encapsulating an application in a container with its operating environment (Hat, Accessed 2023). These mechanisms enable the efficient use of resources and the isolation of applications, improving their reliability and scalability.

Virtualisation Virtualization is a technique that allows a physical computer or server to be partitioned into multiple virtual machines (VMs). Each VM is a self-contained computer resource that runs on software rather than a physical machine. A "host" physical machine can run one or more "guest" VMs, each with its operating system and applications. The VMs operate independently of each other, even if they are running on the same host. This enables, for example, a MacOS VM to run on a Windows host machine.

The process of virtualization is facilitated by hypervisor software. A hypervisor can be installed directly on the hardware or on top of an operating system. It divides physical resources into virtual environments for use by virtual machines. Figure 1.2 depicts a virtualization architecture.

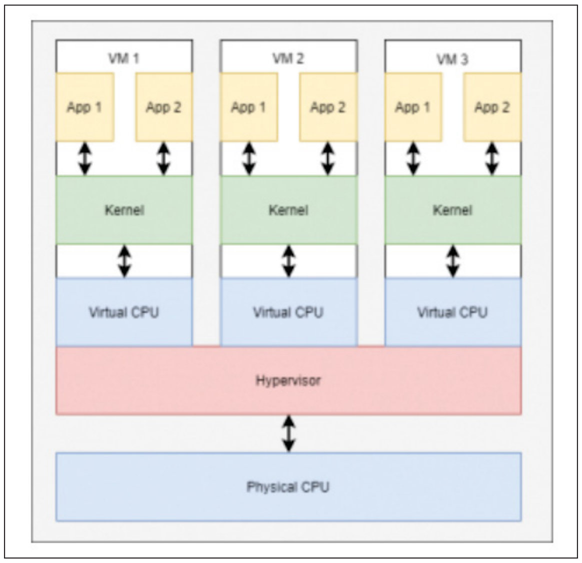


Figure 1.2 Virtualization explanation scheme (Baeldung, Accessed 2023)

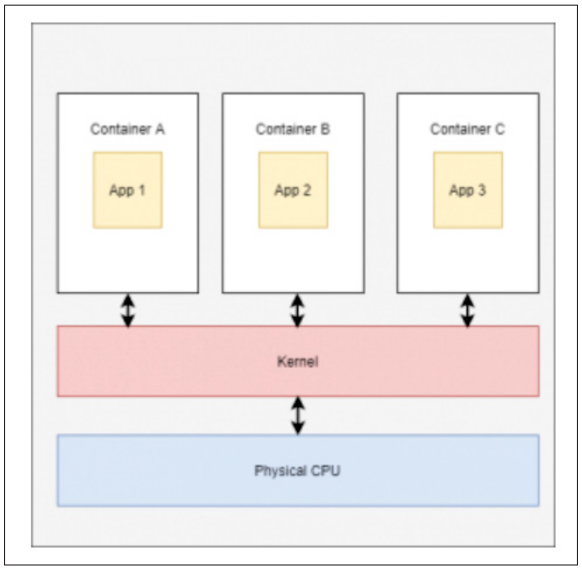


Figure 1.3 Containerization explanation scheme (Baeldung, Accessed 2023)

Containerization, a lightweight virtualization technique The containerization provides a lightweight alternative to virtualization by using the host OS instead of installing a separate OS for each VM (Hat, Accessed 2023). Containers provide a popular virtualization method that allows applications to operate independently of the underlying host operating system. They can host microservices, software processes, or large-scale applications, bundling all necessary files such as executables, binary code, and configuration files. Compared to traditional virtualization, containers are more lightweight, portable, and deploy faster with reduced overhead as they do not need an entire operating system image. Container clusters, managed by a container orchestrator like Kubernetes, can be used for large-scale application deployments (NetApp, Accessed 2023). The containerization architecture is presented in Figure 1.3, which has been sourced from Baeldung (Accessed 2023).

In the context of IoT environments, lightweight virtualization techniques, especially containers, have gained widespread adoption for deploying and managing services on IoT devices to address the challenges of a heterogeneous stack of technologies. Containers offer advantages by packaging IoT services and their dependencies into independent and autonomous modules. Additionally, container orchestration techniques, such as Swarm and Kubernetes, enable resource management at the cluster level and provide a means of communication among containers over virtual networks. In the following subsection, we review the literature on adopting virtualization techniques in IoT applications.

1.1.2 Related work

Several studies have explored the use of container technologies, such as Docker, for virtualization and resource management on edge devices in IoT networks. Ismail et al. (Ismail *et al.*, 2015) evaluated Docker's effectiveness as an edge computing platform and found that it can provide significant advantages for IoT applications regarding deployment, management, and fault tolerance. Similarly, Morabito (Morabito, 2017) conducted a performance evaluation of container virtualization on IoT edge devices and demonstrated the potential for reducing resource overheads and improving scalability. Ruchika (Ruchika, 2016) also evaluated Docker for IoT applications and found that it can provide effective means for managing resources and scaling IoT applications on the edge. However, effective resource management in container-based edge computing environments is a pressing need, as highlighted by these studies.

Given the growing importance of edge computing in IoT networks, there is a critical need to explore the development of resource management strategies using container technologies. This is an important research area that requires further investigation.

In Morabito & Beijar (2016), a design approach is proposed for processing data at the network edge using lightweight virtualization technologies like Docker containers. The approach is customized for IoT applications, where functional components are implemented as reusable containers, providing a flexible environment. It enables dynamic provisioning of various device and data management functions along with orchestration capabilities. However, evaluating the orchestration and data processing modules is not concrete, although CPU, memory, and network usage are assessed.

A container-based approach for resource allocation in IoT, aimed at improving the utilization of resources offered by edge devices while reducing network traffic, is proposed in Renner, Meldau & Kliem (2016). The proposed approach facilitates the dynamic allocation of resources to various applications and users, thereby optimizing data processing at the edge level rather than transmitting it to the cloud. Although the feasibility of the approach has been assessed in terms of performance, considering the resource-constrained nature of edge devices, the case study presented primarily aims to demonstrate its feasibility rather than providing a thorough analysis of resource utilization enhancements, such as CPU and memory.

The proposed approach in Brogi, Mencagli, Neri, Soldani & Torquati (2017) utilizes containers for autonomic data stream processing application management and orchestration in the Fog layer. Docker containers encapsulate Autonomic Applications (Apps), and Application Controllers (ACs) interact with the Fog Node Controllers (FNCs) to manage resource allocation. Although this approach enhances scalability and agility, it assumes a rich-resource infrastructure and does not address resource-limited scenarios common in IoT applications.

1.1.3 Positioning of our approach

The reviewed literature highlights the positive impact of container technology on resource utilization and its potential to address the challenge of heterogeneity. Virtualization is a crucial component of cloud computing, allowing multiple work environments to run on a single server. Detailed explanations of virtualization architecture, including technologies and trends, are available in various sources, such as Varghese & Buyya (2018) and Pahl, Brogi, Soldani & Jamshidi (2017). Containers, another virtualization property, enable microservices to be deployed on cloud servers. Containers offer significant advantages, including scalability, resource efficiency, and portability, making them an attractive option for modern application development and deployment.

However, introducing container concepts brings new challenges for deploying container-based applications. For instance, in monitoring, as containers work together to provide microservices, they become a distributed system. Especially at a larger scale, they require monitoring many dynamic parts, generating an explosion of metrics. For example, a monolithic application may have 50 metrics to monitor, but its deployment based on containers multiplies the number of metrics to monitor by the number of necessary containers (e.g., 30 containers). Assuming we have 30 containers, the total number then becomes 50 * 30 = 1500 metrics.

As the use of container-based deployment continues to increase, there is a need for a new monitoring approach that can effectively observe the health of the system. However, it is not sufficient to only evaluate the feasibility and effectiveness of container-based deployment. An effective approach for resource management using this technology also needs to be described to ensure optimal utilization of resources. In our first contribution (Chapter 2), we define a model describing the concepts related to service deployment using containers. Due to the ephemeral nature of containers and their increasing scale, our model considers cluster and service concepts instead of monitoring only individual container state. Our model aims to make the deployment process automatic and optimize resource utilization fairly.

1.2 Auto-scaling techniques

This section focuses on the key functionality of distributed systems, auto-scaling, especially in the context of edge computing. Edge computing involves delegating specific tasks from the cloud to peripheral devices, such as gateways located closer to the data source. To ensure efficient task distribution among computing devices in a network, automatic scaling is essential to determine how computing and processing capabilities increase when extended to multiple machines.

1.2.1 Brief background

Elasticity is one of the fundamental properties of a cloud environment, allowing for managing unpredictable changes in workloads (Fernandez, Pierre & Kielmann, 2014). In their article (Al-Dhuraibi *et al.*, 2017), Al-Dhuraibi et al. address the primary problems and research challenges related to elasticity in the cloud, emphasizing the critical importance of optimizing the auto-scaling process to ensure elasticity.

1.2.1.1 Auto-scaling concept

Auto-scaling is a technique commonly used in distributed computing, particularly in the context of cloud computing, to dynamically adjust the allocation of computing resources across a cluster of servers based on fluctuations in traffic load. Scaling is a key orchestration feature, providing policy, description, and flexibility for managing containers and virtual machines in the cloud environment (Kovács, 2019).

Auto-scaling is a powerful feature that enables dynamic management of server resources, allowing them to be allocated or released in response to changes in user activity. For instance, when user traffic surges, additional servers can be assigned to accommodate the increased load, while a reduction in active servers during periods of low demand optimizes resource utilization. To demonstrate this functionality, an example similar to the study Mishra, Sahoo & Parida (2020) is presented, where the number of servers running behind a web application can be automatically adjusted based on the number of active users. Since server requirements fluctuate throughout the day, and servers are finite resources, it is critical to maintain an adequate number of servers to support the current load.

The auto-scaling methodology is closely linked to the concept of load balancing, drawing inspiration from it in order to maintain optimal performance across the computing environment (Mishra *et al.*, 2020). The benefits of auto-scaling extend beyond improved performance to include increased efficiency and cost-effectiveness in providing computing resources. By optimizing resource allocation based on real-time demand, auto-scaling minimizes wastage and reduces the overall resource management cost.

1.2.1.2 Auto-scaling classification

Auto-scaling approaches can be classified into two categories:

Horizontal vs. vertical auto-scaling There are essentially two types of auto-scaling, vertical and horizontal (Al-Dhuraibi *et al.*, 2017), as presented in Figure 1.4. The difference between these two types of scaling stems from the manner in which computing resources are added to our infrastructure. In vertical auto-scaling, additional computing power is added to existing replicas/nodes. In horizontal auto-scaling, additional capacity is achieved by adding more instances (i.e., replicas) to the environment and sharing processing and memory load across multiple devices, such as duplicating containers.

Reactive vs proactive auto-scaling Furthermore, auto-scaling is also categorized into two types: proactive and reactive (Lorido-Botran *et al.*, 2014).

Reactive auto-scaling algorithms respond to changes in workload or resource usage by adjusting resource allocation at demand based on predefined rules and thresholds. This approach has been used to develop numerous solutions, including industrial solutions such as Autoscaler Horizontal Pod (HPA) of Kubernetes (Kubernetes, Accessed 2023b) and Amazon EC2 (Services", Accessed 2023).

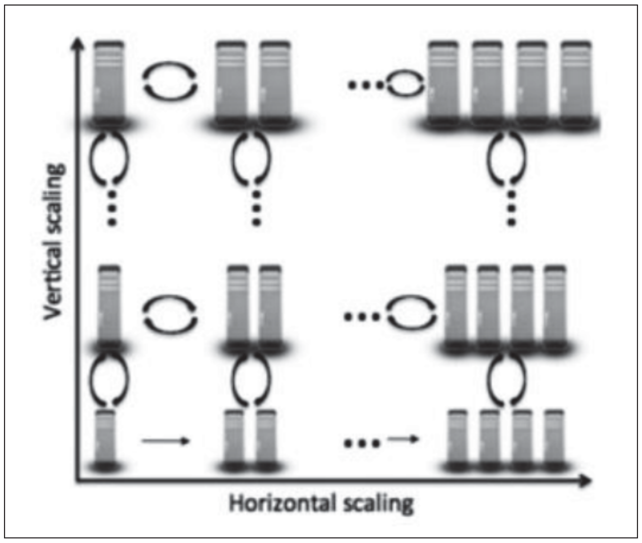


Figure 1.4   Types of Auto-scaling (Al-Dhuraibi *et al.*, 2017)

In the proactive auto-scaling approaches, the future resource demand is predicted based on historical data, using techniques such as time-series analysis. For that, various statically and machine learning algorithms are used, such as AutoRegressive Integrated Moving Average (ARIMA) and Long Short-Term Memory (LSTM). The proactive approach has been implemented in numerous solutions, such as those presented in Goli *et al.* (2021), Imdoukh *et al.* (2019), and Calheiros *et al.* (2014).

1.2.2 Related Work

Auto-scaling is a popular research area, particularly in the context of cloud computing. In Kovács (2019), auto-scaling is defined as the dynamic and automatic adjustment of computing resources in a set of servers based on traffic workload. Scaling is also an essential aspect of orchestration in terms of policy and flexibility for cloud containers and virtual machines. Auto-scaling is beneficial for meeting customer resource requirements by reducing the number of active servers when activity is low and launching new servers when activity is high.

The popularity of container-based deployment is increasing in various domains, including edge computing, as demonstrated by research such as Khazaei, Bannazadeh & Leon-Garcia (2017) and Wong, Zavodovski, Zhou & Kangasharju (2019). It is worth noting that in the case of edge computing with limited resources, having a mechanism for increasing and decreasing computational resources (i.e., vertical auto-scaling) on the same node is less useful or even unrealistic. Therefore, horizontal auto-scaling becomes more suitable by dynamically changing the number of replicas (e.g., containers or pods) to distribute the processing load among devices that constitute a so-called cluster.

We present a comprehensive review of the related work in the field of auto-scaling techniques for edge computing. We categorize this related work into four elements: Reactive Auto-Scaling Approaches, Proactive Auto-Scaling Approaches, Featurization for Forecasting Accuracy Improvement, and Oscillation Mitigation.

1.2.2.1 Reactive Auto-Scaling

For reactive auto-scaling that react to current system workloads, often based on predefined thresholds, we examine various approaches and highlight their strengths and limitations.

In Khazaei *et al.* (2017), an Autonomic Management System (AMS) is proposed as a module for managing resources and providing container-based auto-scaling solutions for IoT applications. The resource management tasks are divided into three layers: Core-Cloud, Edge-Cloud, and Aggregator (IoT gateways), and the system utilizes the Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) loop. The AMS scales IoT applications by analyzing the workload and internal state of the nodes.

In their study Wong *et al.* (2019), the authors aim to address the absence of location-awareness features in container-based clustering and deployment techniques. To achieve this, they propose a two-module approach consisting of classification and edge scheduling modules. The classification module categorizes scaling requests based on user-defined metrics like CPU utilization. In contrast, the edge scheduling module deploys containers to the edge node with the lowest possible latency based on location. However, one notable limitation of these modules is that they do not consider the overall system state when determining the need for scaling operations.

These previously presented works and others in Kovács (2019); Nguyen *et al.* (2020); Taherizadeh & Stankovski (2019), are a reactive approach. A reactive auto-scaler reacts following workload changes. The reactive approaches are generally based on rules with predefined thresholds, making these approaches easy to implement. However, the reactive approaches still lack proactivity, which limits the system's ability to adapt to the operational environment appropriately. The frequent changes in the requested workload, such as HTTP requests, make it challenging for reactive approaches to respond quickly, which can lead to performance issues. More precisely, it generates oscillations due to sudden and unpredictable workload changes. As a result, the reactive approach results in waste due to the over-provisioning of resources and degradation of the system's performance when releasing the resources that the system needs.

To overcome the limits of the reactive approaches in response to changes in system workload demands, Klinaku et al. (Klinaku, Frank & Becker, 2018) proposed CAUS (Custom AUtoScaler) tool, which is an elasticity control tool for containerized microservices. In addition to the reactive mechanism, CAUS uses a second mechanism that manages additional containers as a pool to meet changing needs during the silent period. Tests of CAUS have shown an improvement over a purely reactive approach. Nevertheless, the use of the pool mechanism can generate unneeded containers. Additionally, the performance of this approach to deal with load peaks depends on custom threshold values.

1.2.2.2 Proactive Auto-Scaling Approaches

Proactive auto-scaler anticipates future needs to adapt the system. Thus, one key factor that makes behavior proactive is the ability to forecast future workloads and adapt the system accordingly at the right time. In the study Dang-Quang & Yoo (2021), the authors have shown that their proactive auto-scaler surpasses Kubernetes' default horizontal autoscaling pod (HPA) in accurately and quickly provisioning and de-provisioning resources. This capability is a significant aspect of proactive auto-scaling, which can forecast future workloads and adjust the system accordingly at the optimal time. Proactive scaling involves using an algorithm to forecast future workloads based on historical data (Lorido-Botran *et al.*, 2014). The subsequent paragraphs will discuss some proactive approaches.

Kan et al. (Kan, 2016) attempt to integrate both reactive and proactive approaches in their "Docloud" approach, which is an elastic cloud platform based on the Docker container manager. This platform adapts to changes in workload by adding or removing containers to meet the system's performance needs. The control that manages this automatic scaling operates in hybrid mode, integrating proactive mode for "scaling down" and reactive mode for "scaling up". The architecture of this platform consists of an HAProxy application load balancer, which serves as the entry point for the web application, linked to a set of Docker containers for the web application. In their approach, to perform proactive scaling down, the system needs to predict future workloads by using the ARMA method (Roy *et al.*, 2011). After obtaining the predicted workload, the system must check if the previous workloads are lower than the provisioning level. If this is the case, it introduces a delay before executing the scale-in to reduce the oscillation effect. Similarly, Li and Xia (Li & Xia, 2016) also used the same scaling approach, but this time they used an autoregressive moving average, which also gave good test results. However, Li and Xia (Li & Xia, 2016) did not take into account the management of the oscillation problem in their studies.

Ciptaningtyas et al. (Ciptaningtyas *et al.*, 2017) proposed a resource elasticity controller approach for containerized systems. The work experimented with different combinations of parameters of the ARIMA "AutoRegressive Integrated Moving Average" model to optimize the accuracy of future workload prediction. The test results in the study of Ciptaningtyas et al. (Ciptaningtyas *et al.*, 2017) show that the ARIMA model was able to achieve the lowest error rate. However, the study overlooked the issue of oscillation mitigation and did not take into account time series data analysis for the parameterization of the ARIMA model. This aspect was partially discussed in the work of Meng et al. (Meng *et al.*, 2016), which proposes the tool CRUPA "Container Resource Utilization Prediction Algorithm". CRUPA is a container management platform that uses the ARIMA algorithm to make predictions on future workloads to meet the needs of automatically scaling containers in its platform. However, the approach's efficiency is only compared with the threshold-based (reactive) approach, without addressing the oscillation mitigation or considering resource utilization during tool execution.

In Goli *et al.* (2021), the authors propose a proactive auto-scaling approach called Waterfall for microservices applications. This approach uses machine learning models to predict Request Rate and average CPU Utilization under a specific load. To achieve the target performance objective, the approach employs Linear Regression (LR), Random Forest (RF), and Support Vector Regressor (SVR) models. Additionally, Waterfall considers the impact of scaling one microservice on other microservices under a given workload by utilizing the microservice graph that shows the dependencies between microservices. Experimental results demonstrate that Waterfall outperforms Kubernetes' default HPA auto-scaler regarding response time and throughput, indicating that considering the impact of dependencies and taking appropriate measures in a timely manner can enhance the performance of microservices applications. However, the study does not compare the performance of the proposed approach to the alternative of independently auto-scaling each microservice. Moreover, the approach's consideration of dependencies can introduce complexity, which is not thoroughly examined in the study. For example, cyclic dependencies and multiple dependencies for a microservice can occur, necessitating the need to ensure the predecessors' microservices before initiating auto-scaling. Additionally, Waterfall does not address the issue of oscillation, which can occur due to rapid changes in workloads.

In their work, Imdoukh et al. (Imdoukh *et al.*, 2019) proposed a proactive approach for automatic scaling of Docker containers using deep machine learning, specifically short and long-term memory (LSTM). Their proposed auto-scaler architecture follows the MAPE-K control loop, with a prediction model based on LSTM used to anticipate future HTTP workload and determine the number of containers required to process incoming requests, thus avoiding delays from starting or stopping containers. Similarly, authors of Dang-Quang & Yoo (2021) proposed an approach based on the MAPE-K loop. They employed Bidirectional Long Short-term Memory (Bi-LSTM) to predict the number of future HTTP workloads.

1.2.2.3 Featurization for Forecasting Accuracy Improvement

While many proactive auto-scaling studies have traditionally focused on univariate time series data, such as workload, it has been widely recognized in the literature that incorporating multivariate data (i.e., features) can significantly improve forecasting accuracy. Cetinski & Juric (2015) demonstrated the importance of extending training data with relevant features, such as the time of day and weekends.

In the context of auto-scaling, LSTM models have been shown to effectively capture complex non-linear feature interactions when applied to multivariate data with numerous dimensions and a substantial volume of data (Ogunmolu, Gu, Jiang & Gans, 2016). Laptev, Yosinski, Li & Smyl (2017) proposed a novel LSTM architecture that leverages an autoencoder for feature extraction, achieving superior performance compared to the vanilla LSTM model. In their data preparation process, they incorporated additional specific features such as weather information (e.g., precipitation, wind speed, temperature) and city-level information (e.g., current trips, current users, local holidays). However, most of these additional features cannot be automatically extracted and need to be logged during data collection.

Various classical statistical time series features have been considered in the literature to improve forecasting accuracy. Hyndman, Wang & Laptev (2015) explored features such as mean, variance, ACF (Auto-correlation Function), trend strength, linearity, peak, and season. Di, Kondo & Cirne (2012) focused on important and predictive statistical properties of host load, including mean load, load fairness index, noise-decreased fairness index, and N-segment pattern. However, these derived features, particularly those related to trend and seasonality, usually require manual analysis to identify their parameters. Chakraborty, Mehrotra, Mohan & Ranka (1992) emphasized the significance of considering correlations among different metrics to improve prediction accuracy and avoid the distortion of forecast models. They attempted to select appropriate features, such as disk space, disk IO time, memory, and CPU.

To further illustrate the significance of incorporating relevant features, Wang *et al.* (2021) established a dataset by collecting features of complex system simulation to improve the resource prediction performance of simulation applications in the cloud. These features include average, maximum, and minimum values of usage metrics such as CPU, memory, file system, network (receive and send bytes), communication delay, and execution time. Similarly, Kao, Chang, Cho & Shun (2020) focused on communication metrics, specifically incoming traffic, outgoing traffic, number of connections, and network traffic load (per day). These features need to be obtained during data logging since they are not derived automatically.

1.2.2.4 Oscillation Mitigation

Another important aspect to consider is the continuous generation of actions by the auto-scaler, which can lead to frequent changes in the number of replicas and result in oscillation issues that waste resources. For example, at time t, the auto-scaler may add a recently released resource from time t-1, or vice versa. Therefore, it is essential to consider oscillation mitigation as an important functionality in the auto-scaling process. By reducing the number of unnecessary changes in the number of replicas, system performance can be improved, and resource wastage can be reduced.

Unfortunately, oscillation mitigation has not received sufficient attention in the literature. To overcome this limitation, the authors of Imdoukh *et al.* (2019) introduced a gradual decrease technique (GDS), as a Cooling-Down Strategy (CDT), which introduces a delay before a resource de-provisioning. The approach presented in Dang-Quang & Yoo (2021), employs also a resource removal strategy (RRS), which removes some resources during workload bursts.

The two previous works propose GDS and RRS strategies to mitigate oscillation. Although these strategies, GDS and RRS, have demonstrated efficacy in reducing oscillation, their effectiveness depends on certain parameters such as delay, which require optimization to achieve optimal results.

1.2.3 Positioning of our Work

In this section, we present the positioning of our research contributions within the context of existing approaches for IoT service auto-scaling in the edge computing. Our contributions aim to address the limitations of existing approaches and provide more efficient and effective solutions, namely, a reactive approach, a proactive approach, a novel featurization technique for improved forecasting accuracy, and an innovative oscillation mitigation strategy.

1.2.3.1 Our Reactive Approach

According to the reviewed literature, several works have addressed reactive auto-scaling using rules with thresholds based on monitoring data collected at the server component level, such as CPU and RAM (Kubernetes, Accessed 2023a; Taherizadeh & Stankovski, 2019). Reactive approaches are often based on a predefined set of rules with thresholds, making them easy to implement. Moreover, this simplicity makes the Reactive threshold-based approaches are commonly used by most current auto-scalers, including industrial solutions like Kubernetes HPA. However, selecting appropriate thresholds can be challenging, especially when dealing with complex workloads, as noted in Imdoukh *et al.* (2019). To optimize the configuration of thresholds, auto-scalers can use static heuristic techniques offline according to predefined workloads (Zhong & Buyya, 2020). These strategies are unable to cope with highly dynamic workloads in which applications must scale at runtime (Zhong, Xu, Rodriguez, Xu & Buyya, 2022).

To address the aforementioned gap, our approach presented in our first contribution employs descriptive models to facilitate service deployment on IoT edge devices. It defines a rule set for the various stages of the MAPE-K loop. Additionally, in conjunction with our work Bali *et al.* (2020), our approach utilizes the association rule technique to automatically determine optimal thresholds for reactive auto-scaling. Furthermore, our approach considers the overall cluster load to promote resource sharing among devices within the same cluster. Unlike related works, our approach considers the system state from individual elements, such as containers and nodes, to the overall state of the cluster and service.

1.2.3.2 Our Proactive Approach

The proactive auto-scaling approach, which can anticipate future needs to adapt the system, has been studied in several articles using time series data analysis Calheiros *et al.* (2014), Messias *et al.* (2016), Prachitmutita, Aittinonmongkol, Pojjanasuksakul, Supattatham & Padungweang (2018), Imdoukh *et al.* (2019), and Tang, Liu, Dong, Han & Zhang (2018). The literature uses two common categories of time series data analysis and forecasting methods.

First, algorithms are based on statistical time series analysis (e.g., ARIMA) such as those presented in Lorido-Botran *et al.* (2014); Calheiros *et al.* (2014); Roy *et al.* (2011); Kan (2016); Li & Xia (2016); Ciptaningtyas *et al.* (2017); Meng *et al.* (2016). These statistical approaches are slow in the case of dynamic workload demands and suffer from resource overuse (Imdoukh *et al.*, 2019). Since most of this work is intended for the cloud environment, the application of these techniques in edge computing is restricted by resource limitations.

Second, there are deep learning-based solutions such as the neural network (ANN) and LSTM algorithms, for instance, in Calheiros *et al.* (2014); Goli *et al.* (2021). In Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021), the experimental results show that LSTM model predicts as accurately as the ARIMA model but with a faster prediction speed.

However, the efficiency of auto-scalers using time series data analysis is highly dependent on prediction accuracy (Doan *et al.*, 2019). This accuracy, in turn, depends on parameters such as the workload pattern, history windows (Lorido-Botran *et al.*, 2014) as well as a machine-learning model and the prediction horizon. Thus, it is required to propose solutions that improve prediction accuracy. As noted by Qu, Calheiros & Buyya (2018) and Cardenas (2018), there is still a need to address the issue of container-based autoscaling, which remains an open research problem.

To enhance prediction accuracy, our second contribution (Chapter 3) introduces an automated approach based on feature extraction, or featurization, of the data. By extracting features such as maximum and minimum values, a general description of the data window is obtained and used to predict future workload. Incorporating these features results in significant improvements in prediction accuracy. Unlike other solutions (e.g., Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021)), our model is multivariate, meaning that it utilizes both historical workload values and the automatically generated features. This automatic generation reduces the need for data preparation compared to classical multivariate approaches that require the collection of additional features. In the subsequent subsection, we further emphasize the advantages of employing our featurization approach.

1.2.3.3 Our Featurization Approach

In our approach, instead of relying on pre-existing multivariate features, we propose the automatic extraction of features from the univariate data presented within each data window using non-linear functions. Inspired by Japanese Candlesticks, a well-known technique used in the trading domain, we apply this featurization technique to each data window, deriving features automatically. This approach eliminates the need for manual analysis of statistical properties, especially trend and seasonality parameters, which simplifies the data preparation process and enhances the accuracy of time series forecasting.

By focusing on the data window, which serves as the prediction model (LSTM) input, we capture window-specific features that contribute to improved short-term predictions. Moreover, the set of generated features from a sequence of data windows helps shape the time series data patterns, leading to enhanced long-term predictions.

Finally, it is worth noting that our featurization approach proposed in this study can be used in other scientific fields where there is a need to make more accurate time-series forecasting, such as transportation domain (Nguyen, Kieu, Wen & Cai, 2018), where there is a need for traffic flow prediction.

1.2.3.4 Our Oscillation Mitigation

Our approach also benefits from the generated features to address the oscillation behavior. The feature values form a value grid, where each line (i.e., value) represents a reference action. This grid enables matching the actions generated by our auto-scaling system to the reference actions to reduce the oscillation issue. We have called this original method of handling oscillation 'grid-based oscillation mitigation'.

Our approach has a further advantage, as the grid values change dynamically according to the historical data window used. It is a parameterless mechanism that offers better oscillation mitigation compared to the common used strategy, the cooldown timer (CDT). Additionally, combining our approach with the CDT mechanism leads to even greater improvement in oscillation mitigation. This makes our approach less demanding and more effective in handling the challenging problem of oscillation in auto-scaling.

In the upcoming section, we will delve deeper into the aspect of prediction accuracy while also considering the constraints imposed by resource limitations.

1.3 Forecasting methods for the edge computing

As the demand for real-time data processing and analysis continues to grow, edge computing has emerged as a promising solution to address the challenges of latency, bandwidth limitations, and data privacy. Accurate forecasting methods are essential to optimize resource allocation and improve the quality of service in edge computing environments.

In this section, we will begin by introducing fundamental concepts of machine learning (ML) and relevant concepts for forecasting methods in edge computing. We will then review existing literature on forecasting methods, highlighting their strengths and limitations. Finally, we will present our approach, which leverages machine learning techniques to enhance the accuracy of resource forecasting in edge computing.

1.3.1 Brief background

1.3.1.1 Machine Learning for IoT

Machine learning (ML) techniques have become ubiquitous across many fields in computer science, including pattern recognition and predictive modeling applications. These techniques can perform classification, prediction, and clustering tasks by using statistical models trained on data samples with measurable features. ML has gained attention in the IoT community for improving system efficiency, as it can analyze data and generate scores that identify patterns or indicate events of interest. ML algorithms can be categorized into four types: classification, regression, clustering, and reinforcement, each with specific usage scenarios (Jagannath, Polosky, Jagannath, Restuccia & Melodia, 2019; Samie, Bauer & Henkel, 2019).

Regression algorithms are classified as supervised learning, in which a predictive model is created to show the relationship between input and output variables. This type of algorithm is particularly useful for predicting numerical or continuous values, such as predicting future energy usage in a smart grid by analyzing historical data (Samie *et al.*, 2019; Zantalis, Koulouras, Karabetsos & Kandris, 2019). Time series models of data are often used in these types of applications.

1.3.1.2 Methods for Time Series Prediction

A time series is a sequence of data points recorded at regular intervals over time (Shumway & Stoffer, 2017). For example, the collected time series data represent the historical workload records in our case. In dependence on the number of variables being observed at each time point, there are two types of time series, univariate and multivariate (Hyndman & Athanasopoulos, 2018). While univariate time series data involves a single variable being observed, multivariate time series data involves multiple variables being observed simultaneously. An example of multivariate time series data, environmental data that contains the measurements of temperature, rainfall, and air quality levels. The main goal of time series analysis is to predict future values of the sequence based on its past values.

Time series prediction can be accomplished using linear regression (LR), a basic and uncomplicated model. The choice of method depends on the characteristics of the time series, such as its level of stationarity and seasonality. In the literature, two commonly used methods are statistical, as ARIMA model (Miller, 2015) and the deep learning techniques, such as LSTM model (Greff, Srivastava, Koutník, Steunebrink & Schmidhuber, 2017).

Autoregressive Integrated Moving Average (ARIMA) is a popular method for modeling stationary time series. It involves fitting a linear regression model to the time series after differencing it to make it stationary. ARIMA models are characterized by three parameters: p, d, and q. The order of the autoregressive part is represented by p, the order of differencing is denoted by d, and the order of the moving average part is indicated by q (Miller, 2016).

Long Short-Term Memory networks (LSTM) is a type of deep learning model well-suited for modeling complex, non-linear patterns in time series data. LSTMs use a series of memory cells and gates to store and manipulate information about past values of the time series, allowing them to capture long-term dependencies and patterns in the data. LSTMs have been used to model a wide range of time series patterns, including multi-step ahead forecasting, multivariate time series forecasting, and time series with irregular patterns (Greff *et al.*, 2017).

In addition, Bayesian networks (BNs) (Mohammadi, Frick & Vouros, 2019), Random forest (RF), support vector regression (SVR) (Pratama & Yang, 2019), and K nearest neighborhood (KNN) (Zhang, Qi & Zhang, 2018) are used for regression tasks.

In order to improve the prediction accuracy, the Ensemble learning technique involves combining multiple models to make predictions, as in Sommer, Klink, Tomforde & Hähner (2016). This combination can be realized by averaging the predictions of multiple models, such as a neural network and a decision tree. The idea behind ensemble learning is to take advantage of the strengths of different models by combining their predictions. This can lead to a more robust and accurate forecast than any single model could provide.

IoT-based prediction solutions have widely embraced regression models for various applications, such as predicting demand in a bike-sharing system (Xu *et al.*, 2020), detecting early signs of heart disease (Kumar & Gandhi, 2018), and real-time monitoring of power consumption (Arce & Macabebe, 2019). Similarly, we utilized regression models to predict future workload in this thesis.

1.3.2 Related Work

As the number of IoT applications continues to grow, integrating artificial intelligence (AI) algorithms into the network edge has gained popularity. In Merenda, Porcaro & Iero (2020), the authors conducted a comprehensive study on the requirements for implementing machine learning models and architecture at the IoT edge. The research also incorporates various intensive experiments aimed at deploying machine learning models on edge devices with limited resources.

1.3.2.1 Machine learning at the Network Edge

The paper Murshed *et al.* (2019) presents a comprehensive survey on the utilization of machine learning at the network edge, covering various models and use cases. The authors also discuss popular frameworks and hardware platforms that can run ML models on resource-constrained edge devices. Two other surveys Cui *et al.* (2018) and Samie *et al.* (2019) also emphasize the significance of incorporating machine learning in IoT applications, enhancing the devices' capabilities for tasks such as information inference and data processing.

The integration of ML techniques with IoT systems has been studied extensively, spanning from sensor nodes at the lowest level to interactive end services at the highest level. This integration has highlighted the potential of machine learning methods to improve IoT applications. These techniques can be employed for various purposes, such as optimizing the network, preventing congestion, and efficiently allocating resources at the network level.

1.3.2.2 Widely used Forecasting Techniques

Many works used algorithms are based on statistical time series analysis (e.g., ARIMA and exponential smoothing) such as those presented in Lorido-Botran *et al.* (2014); Sangpetch *et al.* (2017); Calheiros *et al.* (2014); Roy *et al.* (2011); Kan (2016); Li & Xia (2016); Ciptaningtyas *et al.* (2017); Meng *et al.* (2016). These techniques are based on historical data and rely on the assumption that past behaviors (e.g., trends) will continue. Second, another approach that has gained popularity in recent years is the use of deep learning-based techniques such as the neural network (ANN) and LSTM algorithms, for instance, in Zhu, Zhang, Chen & Gao (2019); Saxena & Singh (2022, 2021); Kumar, Saxena, Singh & Mohan (2020); Kumar, Singh & Buyya (2021a); Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021).

In Saxena & Singh (2021) work, a novel framework for energy-efficient resource allocation in cloud data centers, based on an Online Multi-Resource Feed-forward Neural Network (OM-FNN). The proposed framework predicts the resource requirements of incoming tasks and matches them with the available capacity of virtual machines (VMs), consolidating the workload onto energy-efficient physical machines. To achieve accurate resource predictions, the OM-FNN utilizes task resource utilization data collected in 5-minute intervals to forecast usage for the next prediction interval. Overall, this approach reduces energy consumption and enhances resource utilization in cloud data centers. In their recent work, Saxena & Singh (2022) present a new approach for workload forecasting in dynamic cloud environments, utilizing an adaptive neural network model with a novel Auto Adaptive Differential Evolution (AADE) algorithm. This univariate prediction model is designed to learn workload traces for consecutive prediction intervals based on historical data, while the AADE algorithm is used to train a feed-forward neural network with three-dimensional adaptation. This approach has the potential to improve resource allocation and utilization.

A neural network model based on adaptive learning called BiPhase is proposed by Kumar *et al.* (2020) for workload forecasting in cloud datacenters. The model utilizes an adaptive evolutionary learning algorithm to improve the accuracy of the workload predictions. The framework can adaptively learn from historical data to forecast workload for the future, providing an accurate method for workload prediction in cloud data centers. This approach has the potential to improve resource allocation and utilization. Kumar *et al.* (2021a) propose a method called self-directed workload forecasting (SDWF) that improves future predictions by capturing forecasting error trends. The model utilizes self-directed learning to adaptively enhance its predictions without external guidance, potentially improving resource allocation and utilization in cloud resource management.

The work Imdoukh *et al.* (2019) proposes a machine learning-based approach for auto-scaling containerized applications, which aims to optimize performance and resource utilization. By training a LSTM model using historical performance data, the proposed approach predicts future resource requirements. Based on realistic workload experiments, it has been observed that the LSTM model provides prediction accuracy comparable to that of the Auto-Regressive Integrated Moving Average model while also offering a prediction speed that is 600 times faster. Furthermore, the LSTM model outperforms the Artificial Neural Network model in autoscaler metrics. Additionally, it was observed that using the LSTM model enables the prediction of future workload, which in turn helps to optimize the use of replicas for handling workload.

On the other hand, in their study, Dang-Quang & Yoo (2021) utilized bidirectional long short-term memory (BiLSTM), a deep learning technique for auto-scaling in Kubernetes. The BiLSTM model was used to forecast future workload and determine the optimal number of replicas for each application in the Kubernetes cluster. The BiLSTM model outperformed both the LSTM model and the state-of-the-art statistical ARIMA model in short and long-term forecasting accuracy on realistic workloads. It also provides significantly faster prediction speeds, ranging from 530 to 600 times faster than ARIMA models for different workloads. Furthermore, the BiLSTM model demonstrates better resource provision accuracy and elastic speedup than the LSTM model.

Proposed in Zhu *et al.* (2019), a novel approach to enhance workload prediction accuracy in cloud computing employs an LSTM encoder-decoder network with an attention mechanism. The model can effectively learn from historical data to handle the dynamic nature of workloads and refine predictions by focusing on critical input sequence components. This approach has the potential to enhance resource utilization and allocation.

Deep learning's inherent capacity to extract meaningful relationships from complex data makes it an appropriate choice for accurate forecasting. In addition, deep-learning techniques can be more flexible and powerful than traditional time series methods, but they often require more data and computational resources. However, as mentioned in Imdoukh *et al.* (2019), the commonly used forecasting approaches can be slow in responding to dynamic workload demands and may lead to resource overuse. In our work, we specifically focus on forecasting techniques for edge computing, taking into consideration both prediction accuracy and resource usage.

1.3.2.3 Lightweight Forecasting Techniques

Despite the widespread adoption of deep learning (DL) techniques, traditional ML techniques like Bayesian Network, Support Vector Regression (SVR), linear regression (LR), and K-Nearest Neighbors (KNN) continue to attract research interest due to their algorithmic simplicity and efficacy, as noted in studies by Gao, Wang & Shen (2020); Kumar *et al.* (2021b).

The article by Raghunath & Annappa (2015) employs SVR to estimate load and predicted performance factors for virtual machine migration. Similarly, Zhong et al. (2018) (Zhong, Zhuang, Sun & Gu, 2018) combine SVR with the PSO optimization algorithm to estimate a load of physical machines in cloud data centers. In Cetinski & Juric (2015), authors used the random forest classifier to provide a model for workload forecasting in the data centers. Bayesian-based methods are used in Dietrich, Nunna, Goswami, Chakraborty & Gries (2010); Di *et al.* (2012) for forecasting purpose. In Tong, Hai-Hong, Song & SONG (2014), the Bayes methods have nearly the same performance as SVM method.

However, these traditional techniques, such as Support Vector Machines (SVM) and Random Forest (RF), are likely to have less accuracy compared to commonly used approaches such as AutoRegressive Integrated Moving Average (ARIMA) and Long Short-Term Memory (LSTM). Additionally, the effectiveness of auto-scaling solutions that utilize time series data analysis heavily relies on the accuracy of the forecasting model, as emphasized in Doan *et al.* (2019).

The challenge of achieving high prediction accuracy is not limited to lighter techniques but also applies to commonly used methods such as ARIMA and LSTM. Therefore, improving the accuracy of resource forecasting in container-based auto-scaling remains a significant challenge that requires attention, as highlighted in Qu *et al.* (2018); Cardenas (2018).

1.3.2.4 Ensemble-Based Forecasting Learning

In the pursuit of improved forecasting accuracy, ensemble learning techniques emerge as a compelling solution. Ensemble learning has demonstrated effectiveness in various fields, such as wind gusts and electricity consumption forecasting (Wang *et al.*, 2021). In the context of auto-scaling, ensemble learning has also been explored in some works in the literature. For example, a method proposed by Cao, Fu, Li & Chen (2014) for predicting CPU load employs an ensemble approach that uses multiple models (called predictors), including Autoregression and Exponential smoothing models, to generate predictions. The approach includes a prediction optimization layer that dynamically adjusts the predictor parameters using the Adaptive Step Size Random Search (ASSRS) strategy, (Schumer & Steiglitz, 1968), to maintain the predictor performance.

In their study, Shariffdeen, Munasinghe, Bhathiya, Bandara & Bandara (2016) proposed an ensemble-based load forecasting approach to improve the accuracy of auto-scalers. The authors evaluated multiple prediction models to forecast various load patterns. To forecast cloud resource usage, Rahmanian et al. (Rahmanian, Ghobaei-Arani & Tofighy, 2018) proposed a learning-automata-based ensemble. The approach presented in Sommer *et al.* (2016) utilizes load forecasting techniques to identify critical parameters for cloud data centers. It is capable of handling non-stationary workloads by updating learning parameters without requiring re-training of prediction models. The authors employed Weighted Majority and Simulatable Experts to effectively manage large-scale and non-stationary workloads with massive streaming data.

The authors of Sommer *et al.* (2016) proposed an ensemble-based module that predicts virtual machine (VM) utilization and incorporates a proactive VM migration policy utilizing predictive overload detection. To forecast workloads, they developed two online learning ensemble learning approaches (Singh & Rao, 2014). Similarly, Chen, Yuan, Liu & Li (2020) proposed a weighted random forest model with an error correction mechanism for workload prediction. The model used a set of random forests, each trained on different training sets, and the final prediction was calculated by weighting the forecasts of each model.

In Gao *et al.* (2020), the authors compare several workload prediction methods and introduce a clustering-based approach to enhance accuracy. This method involves categorizing tasks into clusters and creating a prediction model for each category using either Prototype-based Clustering Method (PCM) or Density-based Clustering Method (DCM) with ARIMA, Bayesian Ridge Regression (BRR), or LSTM models. To allow for scheduling based on predicted workload, the authors recommend performing predictions a specific time before the target time point.

In Kumar *et al.* (2021b), the authors conduct a comparative study of machine learning methods such as LR, KNN, SVR, and ARIMA for web application workload forecasting. The proposed prediction model aims to select the most suitable algorithm for workload features. Furthermore, other studies such as Jiang, Perng, Li & Chang (2013); Liu *et al.* (2015); Cetinski & Juric (2015) have employed hybrid forecasting methods that involve integrating multiple machine learning models.

However, it should be noted that these works do not take into account the resource usage aspect, which is a crucial constraint considered in our approach. The use of multiple models in the Ensemble learning technique can significantly increase resource consumption, which is not feasible in resource-limited edge computing environments.

In general, since the presented state-of-the-art solutions are intended for cloud computing workload forecasting, the application of these techniques in edge computing can be restricted by resource limitations. Unfortunately, this resource-constrained aspect is commonly overlooked in the related work on auto-scaling.

1.3.3 Positioning of our approach

Given the limited resources available in the Edge computing context, our third contribution (Chapter 3) places great emphasis on resource usage. We assess the suitability of state-of-the-art forecasting techniques with respect to resource constraints. Specifically, our approach investigates the feasibility of common, albeit potentially heavy, techniques, as well as the usefulness of traditional machine learning methods that are potentially lighter, such as Support Vector Regression (SVR), Random Forest (RF), and Bayesian Networks (BN). Furthermore, to improve the prediction accuracy, we propose using Ensemble techniques that combine the predictions of multiple models. This approach can enhance the robustness of the forecasting process and provide more accurate and reliable predictions.

In Kumar *et al.* (2021a), the authors noted that using a single model may not be optimal for modeling and forecasting various data types, as these models were created and trained for specific workload types. Therefore, to address this issue, a combination of multiple methods was employed to more effectively model and forecast workloads.

However, it is important to note that using Ensemble learning techniques can result in increased resource consumption, as it involves using multiple models. Therefore, it is crucial to consider the trade-off between prediction accuracy and resource usage. To address this challenge, we first used a lightweight solution based on the weighted average, similar to the Weighted Majority and Simulatable Experts techniques presented in Singh & Rao (2014). Second, we formalized this optimization problem and proposed a heuristic solution that considers resource limitations when selecting a limited number of models for the Ensemble learning process.

Lastly, our approach enables the distribution of training across multiple edge devices by training a local ML model for each device. This improves scalability and addresses the challenges posed by federated learning, a recently proposed distributed training approach that has communication overhead, interoperability issues with heterogeneous devices, and resource allocation constraints, as discussed in Lim *et al.* (2020). In our approach, the local models generate predictions that are transmitted to another edge device, which computes a weighted average of all local predictions using our optimization solution. This improves the accuracy of the predictions while reducing the burden on individual edge devices.

CHAPTER 2

RULE BASED AUTO-SCALABILITY OF IOT SERVICES FOR EFFICIENT EDGE DEVICE RESOURCE UTILIZATION

Ahmed Bali[1] , Mahmud Alosta[1] , Soufiene Ben Dahsen[1] , Abdelouahed Gherbi[1]

[1] Department of Software Engineering and IT, École de Technologie Supérieure
1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3
Paper published in the Journal of Ambient Intelligence and Humanized Computing on June 2020.

ABSTRACT Conveying the workload of IoT systems from the cloud to edge nodes have been widely adopted by industrial and academic sectors. This tendency is generally promoted to meet the requirements of some time-sensitive use cases such as IoT healthcare applications. However, IoT devices at the edge network are likely to be resource-limited, as well as, they perform under an extremely heterogeneous environment in terms of the connected devices and the deployed software modules. Thus, both of the aforementioned concerns have considerably led to hindering the deployment process of services on IoT edge devices. In this paper, we propose an approach to facilitate a scalable and lightweight solution for service deployment for efficient resource utilization on IoT edge nodes. Our solution is based on the container concept, and we adopt the cluster concept to define a group of IoT edge devices. Containers are lightweight virtualization technique that enables services to be packaged and deployed with their dependencies regardless of the host's infrastructure, as well as, they facilitate the service communication and the update process. Furthermore, containers are supported by some means of orchestration such as swarm. These orchestration tools can be configured to enable services deployment and resources sharing among IoT edge devices falling within the same cluster. However, they lack elasticity in terms of auto-scaling up/down of services' instances in corresponding to the resource utilization of all cluster elements, as well as, service performance metrics. Our approach overcomes these limitations by following an auto-scaling process based on MAPE-K loop, which is based on our proposed rule model to generate a scaling plan by analyzing collected performance metrics of a cluster. Our evaluation shows the efficiency of the proposed approach in adapting the system performance to meet service performance requirements, and to meet the availability of system resources.

Keywords: Edge Computing, Container, Resource Management, Auto-Scalability

2.1 Introduction

The innovative vision of the Internet of Things (IoT) has led to a technological revolution in all the aspects of our lives, i.e., health, transportation, and work. IoT applications have become ubiquitous and their influence in several areas of our daily life is indispensable. These applications are mostly reliant on networks composed of tiny devices dipped in our surroundings, for instance, in the form of environmental, healthcare, and industrial sensors. These devices are in an ongoing process of development and deployment, leading to billions of devices connected to the Internet (Evans, 2011). In traditional IoT network topology, these devices communicate their captured observations directly or indirectly to central cloud services for processing and decision making tasks. The latter is likely to be impacted by the latency resulted from transferring data and requests bidirectionally between cloud services and sensor nodes. Furthermore, transferring the ever-growing volumes of data generated by IoT devices following the abovementioned topology has led to relative IoT costly solutions, in addition to network bottleneck (Al-Osta, Bali & Gherbi, 2019).

The demand for cost-efficient and responsive IoT applications has recently pushed toward the deployment of a portion of IoT services at the side of the network edge. In this topology, sensors are configured to transmit their data to an intermediary device, which mainly handles some processing tasks such as data pre-processing, aggregation and filtering (Venticinque & Amato, 2019). Afterward, it communicates the results or only the significant observations to the cloud. Thus, a considerable amount of data is expected to be avoided to be transmitted to the cloud, which implicitly would lighten the traffic. This would be reflected in the enhancement of the network performance. Moreover, in some use cases such as IoT healthcare applications

(Devarajan *et al.*, 2019), an immediate action is required; thus, real time data processing close to their resources is of an utmost significance.

Nevertheless, devices at the edge are characterized by their limited resources compared to the cloud, as well as, they are likely to be connected to several sensor nodes that periodically push their data. By default, these sensor nodes are composed of set of heterogeneous devices that, in some cases, rely on different protocols and technology stacks to communicate with devices on the edge. Thus, the heterogeneity characteristics in the hardware level is reflected on various services and applications deployed on top of the edge layer. This indeed would lead to the amplification of the lack of the interoperability issue. In addition, handling this type of technological diversity is likely to consume a considerable amount of resources on the edge devices and to hinder the opportunity of fully taking advantages of the underlying technology.

To overcome aforementioned issues, lightweight virtualization technologies, such as containers, have been widely adopted to facilitate service deployment and management on IoT edge devices (Ahmed *et al.*, 2019). Containers provide the advantage of packaging and running IoT services and their dependencies in isolated and self-contained modules. Employing such technology in IoT edge devices is likely to lead to the avoidance of the strict reliant on specific technology since it has the capability to merge extreme different technologies in one virtualized software component. In addition, the communication between these modules is maintained by some kind of container management tools such as the docker engine. This is likely to contribute to the mitigation of the heterogeneity challenge, as well as, to enable rational deployment of IoT services at the network edge. In addition, containers can be customized to cope with resources available on the target node. Moreover, container orchestration techniques such as Swarm enable resource sharing between IoT edge devices falling within the same cluster and provide means of communication between containers across virtual networks. However, they lack elasticity in terms of auto-scaling up/down of services' instances in corresponding to the resource utilization of all cluster elements, as well as, service performance metrics.

In this work, we emphasize on the utmost necessity to investigate solutions to optimize the deployment of container based IoT services on network edge devices taking into consideration their resources restrictions. As well as, to automate the process of a fair distribution of these services on devices falling within the same edge-cluster.

Our solution follows MAPE-K(Monitor-Analyze-Plan-Execute over a shared Knowledge) loop (Computing *et al.*, 2006). The shared knowledge is represented by our proposed rule and deployment models. In the rule model, we defined four categories of rules namely evaluation, decision, scale, and verification. Each one of them corresponds to a specific step of the proposed auto-scaling process. While the deployment model captures the different system elements with their properties involved in the deployment process such as the device and the service. The proposed approach has been evaluated and several criteria are considered namely CPU and memory utilization, and network traffic, as well as, the response time as a service performance metric.

The main contributions presented by this work are briefly described as follows:

- Deployment Model: It contains concepts and properties to describe all possible elements included within the deployment process.
- Rule Model: It uses concepts from the deployment model, and it contains several categories of rules used to evaluate the need to scale and to generate the scaling plan.
- Proposing algorithms and strategies based on the rule model that defines our auto-scaling process steps inspired from MAPE-K adaptation control loop introduced by IBM (Computing *et al.*, 2006).

The reminder of the paper is organized as follows: First, Section 2.2 discusses the related work with respect to the approach proposed in this paper. Then, the preliminary concepts behind the proposed approach will be presented in Section 2.3. Afterward, Section 2.4 discusses the knowledge models used by the proposed auto-scaling process which presented in Section 2.5. Followed by the evaluation and experimental results discussion in Section 2.6. Finally, Section 2.7 concludes the paper and highlights directions for future work.

2.2 Related Work

The use of containers has been recently an attractive subject of research for both industry and academia. In this context, some research initiatives have been proposed to evaluate the feasibility of employing containers on the edge of IoT networks taking into consideration their limited resources. For instance, Ismail *et al.* (2015), Morabito (2017), and Ruchika (2016) have carried out research studies to explore container virtualization technologies and their role as an effective means to facilitate the development and to adapt (scale) IoT applications. To assess the impact of the container on the IoT devices' resources, IoT applications were developed and deployed in the form of Docker containers, then Benchmark tools were used to measure their performance. Some evaluation criteria have been considered such as deployment, resource and service management, and fault tolerance. The results show that the use of Docker containers offers good performance. However no clear approach is described for effective resource management. Although Docker brings advantages for resource utilization in IoT context, it is only a working tool. Thus, it is necessary to describe an effective approach for resource management purpose using this technology. In the following, we investigate some recent work in comparison with the approach presented in this study.

A design of data processing approach is proposed in Morabito & Beijar (2016), it is an edge oriented and its functional components are customized as reusable Docker containers. This facilities building versatile environment for IoT applications, and enables elastic provisioning of distinctive device and data management, as well as, orchestration capabilities. While their implementation is determined on the evaluation of the CPU, Memory, and Network usage, it lacks a concrete assessment of the orchestration and data processing modules, which is extensively considered in our work.

A container-based resource allocation model was proposed in Renner *et al.* (2016) to increase the use of resources offered by IoT devices and reduce the generated network traffic. The proposed approach allows different applications and users to dynamically allocate the resources offered by IoT devices and thus maximize data processing at the source level instead of sending them to the cloud. The feasibility of this approach has been evaluated in terms of performance since IoT devices are limited in terms of resources. However, the case study of the work is focused on presenting the feasibility of the approach and not on improving the performance of resources such as CPU and memory.

A resource management and orchestration approach based on containers is proposed in Brogi *et al.* (2017) for supporting autonomic data stream processing applications on the Fog layer. Fog nodes (FNs) are equipped by three main components namely Fog Node Controllers (FNCs), Autonomic Applications (Apps), and Application Controllers (ACs). Apps are encapsulated in the form of Docker containers; each of them runs an AC. The AC interacts with the FNC of the corresponding FN to facilitate scaling up/down the set of resources (e.g., number of cores, CPU time, and bandwidth) assigned to the Docker container. Unlike our work, their implementation relies on rich-resource infrastructure and does not take into consideration scenarios where the Fog node is resource-limited, which is the case in a considerable portion of IoT applications.

In Khazaei *et al.* (2017) the author proposed resource management module referred as the Autonomic Management System (AMS). The main objective of this work is to provide container-based IoT application auto scalability solution by means of distributing the resources management task on three varied resources layers namely: Core-Cloud, Edge-Cloud, and Aggregator (IoT gateways). Like our approach, this approach is based on Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) loop, and it scales IoT applications mainly based on both workloads and node internal state. In addition to both aforementioned scaling criteria, our approach takes into consideration the overall cluster overload, which promotes resources sharing among devices within the same level. For the knowledge base representation, our approach proposes a descriptive model for service deployment on IoT edge devices as well as a rule set used by the different steps of the MAPE-K loop. In addition, the service performance metrics such as response time is considered.

The proposed approach in Wong *et al.* (2019) is presented to overcome some container based clustering and deployment technique shortcomings, mainly, the lack of location-awareness deployment feature. The solution is composed of two main modules namely classification, and edge scheduling. The former module is proposed to classify received scaling requests based on predefined user metrics such as CPU level. While the latter module is proposed to facilitate the deployment process with the lowest possible latency by enabling closest location based selection of containers to be deployed on the target edge node. This approach constraint on dealing with scaling request without conducting an analysis on whether the current state requires scaling or not, which is addressed by our approach.

2.3 The Preliminary concepts of the proposed approach

In this work, a solution based on container virtualization technology has been proposed to overcome device limitation and deployment issues. This approach relies on some container clusters concepts to promote resource sharing between devices and enable easy and independent deployment of the hosting infrastructure. Furthermore, an auto-scaling mechanism of services is proposed to meet both the resource limitation and the service requirements (e.g., service response time). Typically, IoT topology is designed on the top of three essential layers namely sensor nodes, gateways and the cloud. The sensor nodes continually send the data to the gateways that store and process it if necessary, and then send it to the cloud for use as needed. Mostly, gateway receives data from multiple sensor nodes, and due to resource limitations of IoT devices, some gateways become overloaded and can no longer perform data processing tasks. Meanwhile, it is likely to find some gateways that are not overloaded and have some available resources within the same cluster. As a result, the work on resource management targets the workload distribution between IoT devices at the network edge represents the essential element in our approach.

2.3.1 Overall Deployment Architecture

In order to fully take benefits of resources available on IoT edge devices, they will be grouped into clusters. A cluster includes a number of edge devices with one as manager while the rest will be workers. The manager node, as indicated by its label, will take care of administering the other worker nodes, as for the other gateways, they will just employ their resources to deploy the services. Each gateway will be equipped with a container framework and its orchestration mechanisms that are responsible on services deployment and communication within the same cluster. This system decomposition will allow the gateways belonging to the same cluster to share resources between them and communicate through a virtual network. Usually each gateway receives data from one or more sensor nodes, but in the case of an overload; depending on our approach the data processing could be delegated to another gateway within the same cluster in order to fairly distribute the load. Each gateway in the system is encompassed of three different categories of services namely reading service, processing service and sending service, these constitute the workflow of data processing coming from the sensor nodes.

In order to facilitate the deployment of services on the gateways, their images are created with all their dependencies (libraries, version, etc.) and stored in a registry. The latter is a centralized service directory shared between all the gateways. So to deploy a service in a gateway, we need just to access the registry and pull the corresponding image. With such a procedure, it would be possible to overcome the problem of heterogeneity of the gateways, since the services are created with all their dependencies and ready to be executed in containers regardless of the infrastructure of the host. Having a centralized repository will also simplify the update, because users just need to make a new build of the image subject to an update, instead of going through all the gateways and make an update.

Figure 2.1 shows the high-level architecture of our system, the details of the essential components will be detailed in the following sections. As illustrated, gateways are grouped into a set of clusters. In each cluster, there is a manager gateway (in green) that manages other gateways. The sensor nodes (in red) communicate directly with the gateways. The gateways of the same cluster communicate with each other through an overlay (i.e., virtual) network. While they communicate directly with the registry to instantiate services and execute them in containers from existing images.

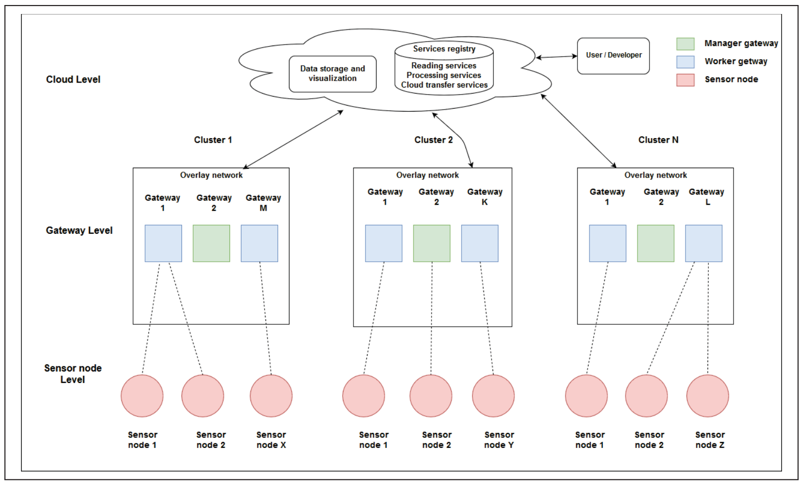


Figure 2.1 The Overall Architecture

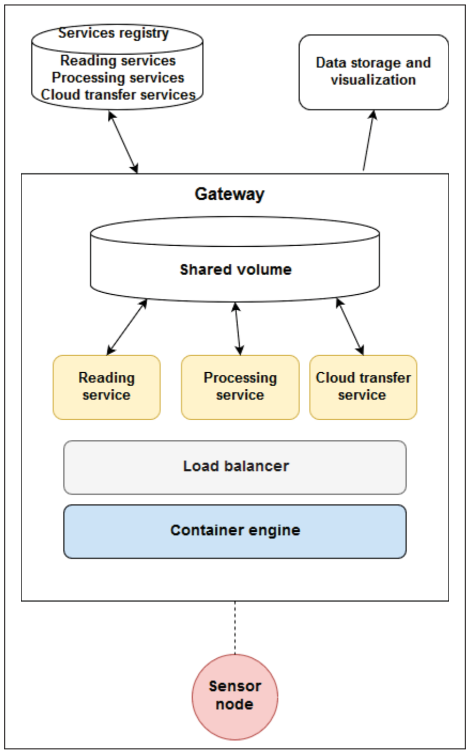


Figure 2.2 The Gateway level Architecture

2.3.2 Gateway level

In this work, more importance is given to the gateway as it represents the core element of our solution. A gateway could be either Manager or Worker, but both share the same architecture. The sole difference lays on the extra functionalities assigned to the manager gateway; so it can run commands to orchestrate services deployment tasks, as well as, to mange the communication between worker nodes.

Figure 2.2 shows the main elements of the gateway architecture. Each gateway is equipped with a container engine to enable services to run as packaged containers. The chosen engine must be able to support the cluster composition. Also, each gateway is equipped with a load balancer, which intended to enable load distribution of similar service instances located in the gateways of the same cluster.

A gateway contains three services that run continuously as containers and constitute the data processing workflow from the sensor nodes. These three services are:

- Data Reading Service: This service receives data packets from sensor nodes on predefined interval time; then it will store them in a given directory of the volume shared between the services deployed on the gateway. Each data packet received includes SensorID, measured value, and Timestamp.
- Data Processing Service: This service communicates with the volume directory to access collected data and to apply data processing tasks required by the user.
- Cloud data transfer service: The main objective of this service is to facilitate communication between the gateway level and the cloud level. This communication aims at enabling messages exchange among the two levels. It includes tasks such as sending processed data to the cloud, and receiving deployment and updating requests from the cloud.

As shown in Figure 2.2, the three aforementioned services use a shared volume to communicate. The latter contains directories for storing the collected data and also for those that are processed. So each service uses the proper directory to accomplish its tasks. While in the case of a gateway manager, it communicates directly with the registry to deploy the services on the gateways, save new images or update existing ones.

2.3.3 Cluster Functionality

Figure 2.3 illustrates the distribution of requests in a cluster consisting of three gateways. Each gateway is identified by an IP address. As we explained in the gateway architecture, there are three services shaping the data handling workflow. Since the data reading service is the first component of the data treatment services chain, so it is implicitly realizable that the distribution of reading tasks would automatically lead to decreasing the amount of data to be treated in the following services on each gateway individually. With the launch of the system, the gateways begin to receive data from the sensor nodes. The load balancer on each gateway receives requests from the sensor nodes, then transfers them to the data read service of the same gateway, or redirects them to another read service instance of another gateway. This division of tasks is based on an overlay network. The data processing service of a gateway accesses the shared volume to retrieve the data delivered by the read service of the same gateway, then it will process them and insert them again in the shared volume. Afterwards, the 'cloud send service' accesses the shared volume to retrieve the data processed by the 'data processing service' deployed on the same gateway, and then transfers it to the cloud. Thus, in the case of IoT system composed of clusters of gateway devices, each cluster will work independently following the aforementioned steps. The goal behind this division of tasks is to have a balanced system; thus avoiding situations where some gateways are overloaded; while others are relatively unburdened. The next section presents our proposed deployment model that captures the different concepts presented in this section.

2.4 The container based IoT service deployment model

This section demonstrates the abstract model proposed in this work to capture the set of concepts involved in the deployment process. This model is intended to serve as a referenced knowledge base to be used by our auto-scaling solution. Thus, the representation of this model is mainly

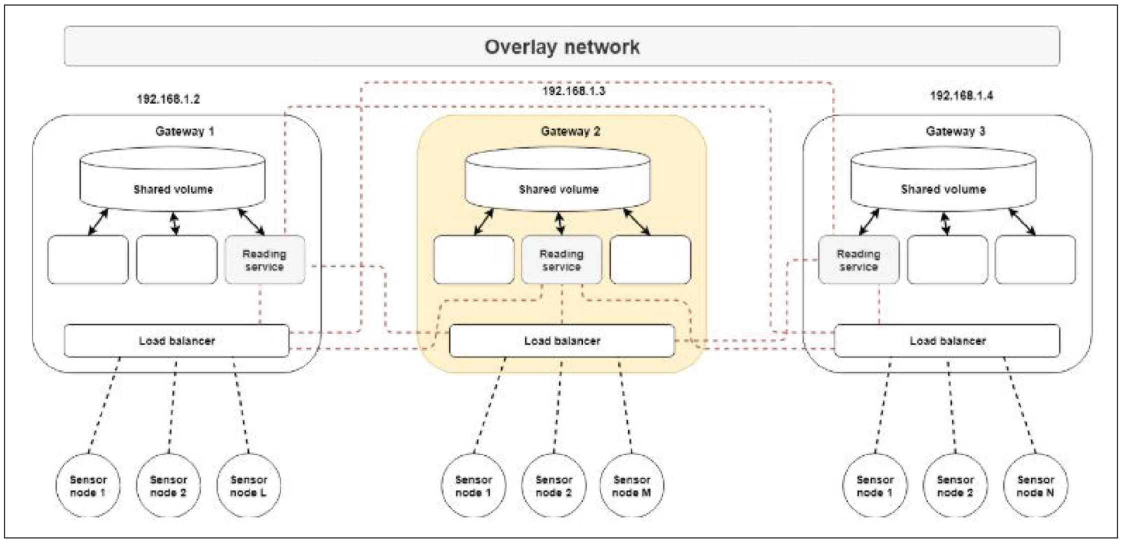


Figure 2.3 Cluster operation

focused to reveal the concepts related to the scalability process namely the devices and the services as well as the assignment of services to the devices (i.e., the deployment). Gateways can be grouped into clusters, to make it possible the sharing of resources, which contributes to alleviating the problems of resource limitation.

For the purpose of simplicity and comprehension, we subdivide the proposed model into different submodels as parts of the total view namely, Device, Service, and Deployment. Thus, our model is formulated out of the union of these submodels, which are detailed in the following subsections.

2.4.1 Device submodel

As previously mentioned (Figure 2.1), a typical IoT network topology is consisted of Edge network devices and clouds. The hierarchy of Edge devices is essentially divided into two layers, mainly the sensor and actuator nodes layer and gateway nodes layer. The solution presented in this work is mainly targeting IoT gateway devices. These devices have recently witnessed a significant technological leap in terms of improved computing and communication resources. This has considerably contributed to facilitating the deployment of software modules and services in the form of containers. Figure 2.4 shows the device part of our proposed model. It contains

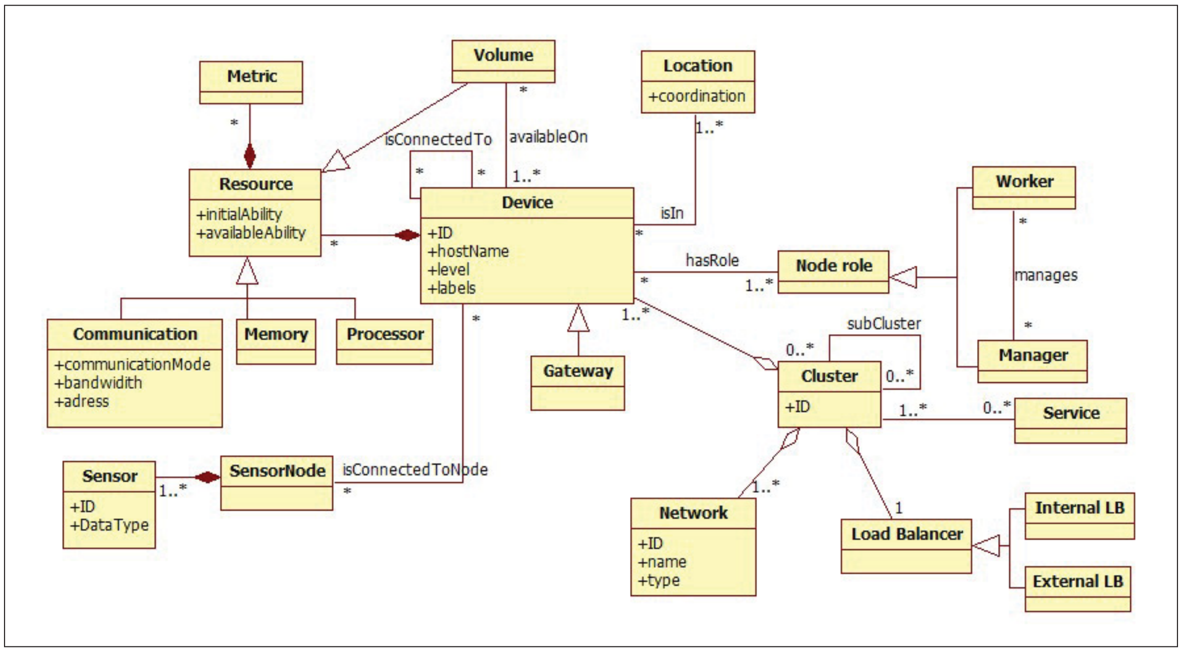


Figure 2.4 Device submode

concepts and properties that represent the device specification. More specifically, these concepts describe the device computing resources such as the processor and memory. This information about resources is useful to evaluate the ability of a device to deploy a service. The device, as part of a cluster, has by default the worker role. It can also play the role of manager who controls and manages the other nodes. The reflexive relationship (i.e., property) ‘isConnectedTo’ defines the existence of a connection between devices. The specification of connection between devices is useful for the creation of the cluster networks. It is possible to define a set of networks, which can be virtual (i.e., overlay network) that are used to deploy services. For example, two services may use two different virtual networks for isolation purposes, while the instances of the same service use the same network.

In the context of IoT, the device might be connected sensor/actuator nodes. Moreover, geographical location information is helpful in the service deployment process. It allows specifying a specific region or even node for deploying a service. The following subsection gives more description of the ’Service’ concept as a part of our model.

2.4.2 Service submodel

Following the microservices architecture model, an application consists of a set of services. These services might be of different types such as sensing, actuating, data transferring and data processing as presented in section 2.3.

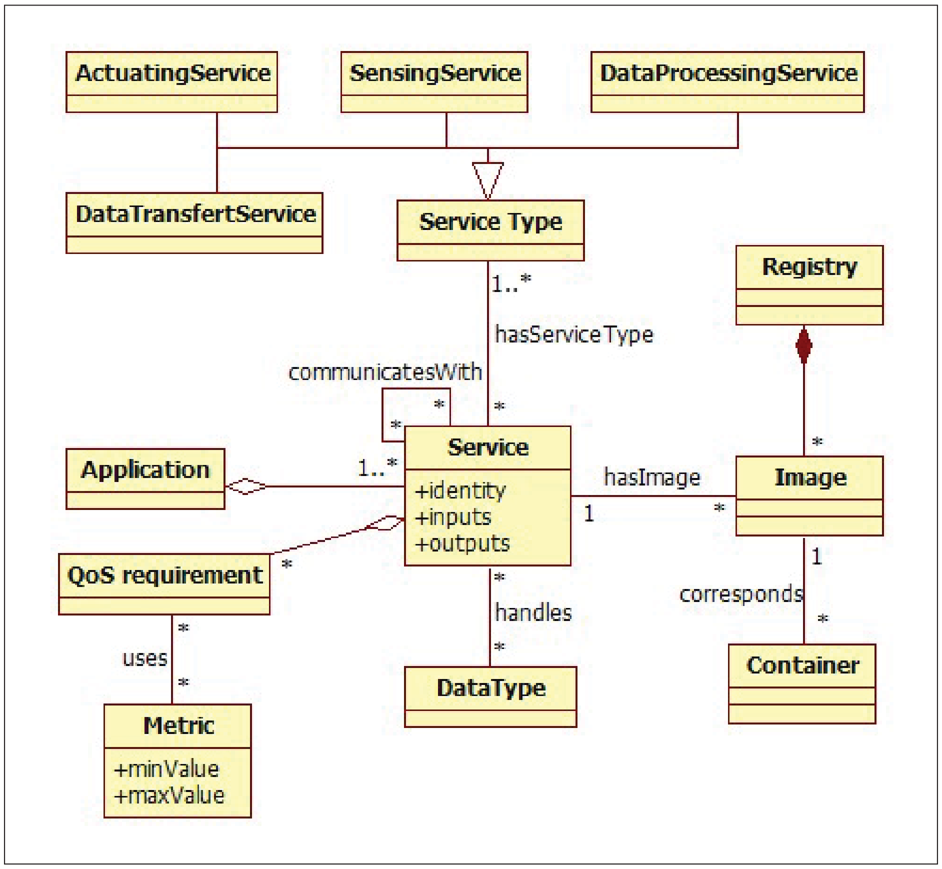


Figure 2.5 Service submodel with the container concept

Figure 2.5 shows our service specification model part. It defines different concepts and properties from a service deployment point of view. The relationship ‘communicatesWith’ is used to define the communications between services. This information is useful for the deployment process in order to deploy these services on devices that belong to the same cluster network, or that have the relationship ‘isConnectedTo’ as presented in subsection 2.4.1. In addition the service model contains a ‘Container’ concept which intended to facilitate the service deployment. To run the container, the system needs to know its image, which is usually stored in a central registry.

Moreover, the service is characterized by a set of QoS metrics (i.e., criteria) that represents the non-functional user requirements. Our approach is mainly focus on the service performance metrics such as the response time, latency and the throughput. The proposed service model specifies two metric delimiters (i.e., thresholds): the minimum and maximum values (minValue and maxValue). For example, Metric=‘Response time’, minValue=30 ms, and maxValue=120 ms. The minimum threshold of the response time criterion (30 milliseconds) defines the best requirement. In other words, the service does not require that the value to be lower than this threshold. However, it requires that the value to be less than the maximum delimiter (120 milliseconds).

Noting that, there are two categories of criteria, negative as the response time and positive as the throughput. The higher value of the response time indicates the lower quality (i.e., negative quality). Contrariwise, the higher value of the throughput is the higher quality (i.e., positive quality).Therefore, to treat these two metric categories, we can scale the positive criteria values by using a decreasing utility function. We can just invert (multiplied by -1) the different criterion values (including the delimiters $minValue$ and $maxValue$). By this way, we can present only the case of negative criteria such as the response time..

2.4.3 Deployment submodel

The previous submodels (Figures 2.4 and 2.5) describe the system components (i.e., elements) namely the device and the service. The deployment submodel presented in Figure 2.6 represents the semantic relation between submodels of the ‘Device’ and ‘Service’ concepts. It presents the assignment of services to the different devices. An actual use case of the assignment represents an instantiation of our general model. Each deployed service represents a number of instances, which corresponds to the service replicas. Each service instance (i.e., replica) is deployed as a container, which is placed on one of the cluster’s devices. In our model, the operator can specify the ‘minReplica’ and ‘maxReplica’ properties to define the minimum and the maximum number of replicas respectively. The values of this properties is an important parameter to our scalability solution, where it automatically updates the number of replicas ('nbrReplicas' service property in Figure 2.6).

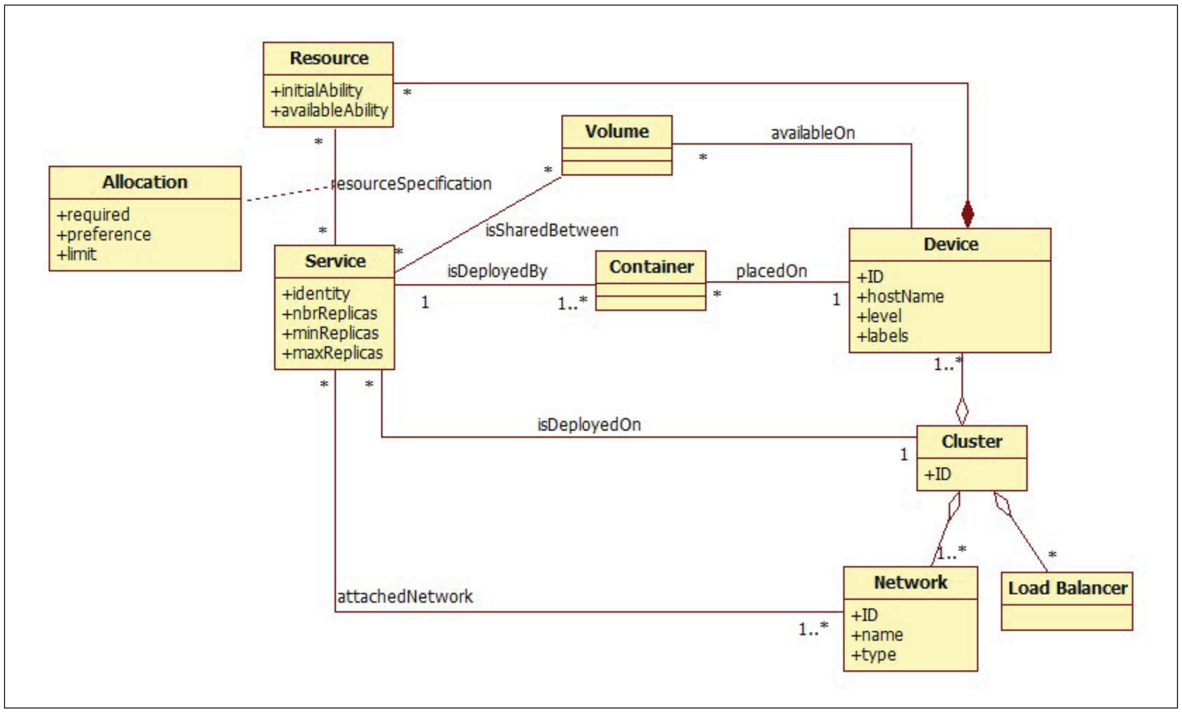


Figure 2.6 Container based Service deployment submodel

The placement of the service instance (i.e. replica) depends on its requested resource specification. Our model specifies for each service (i.e. implicitly its instances) a set of properties namely the required, the preference, and the limit of each resource. The 'required' property means the minimum of the resource value that is necessary to perform the service. For an efficient performing, the 'preference' property value is specified by the developer, the system operator or by history data analysis. Finally, the 'limit' property specifies the limit value that the service cannot exceed. For example, a database service 'db_service' might require 100 Mbytes of memory resource. The preference value is 120 Mbytes and its limit is 150 Mbytes.

2.5 Auto-scaling solution process

This section presents our solution of auto-scaling of services deployed on IoT edge devices. To perform the auto-scaling, our approach follows a process inspired from MAPE-K (Monitor-

Analyze-Plan-Execute over a shared Knowledge) loop (Computing *et al.*, 2006). Figure 2.7 presents our auto-scaling process, which is composed of five (05) steps, namely: Monitoring, Evaluating, Making decision, Generating a scale plan, and Executing the scale plan. The different steps and elements will be explained in the following subsections.

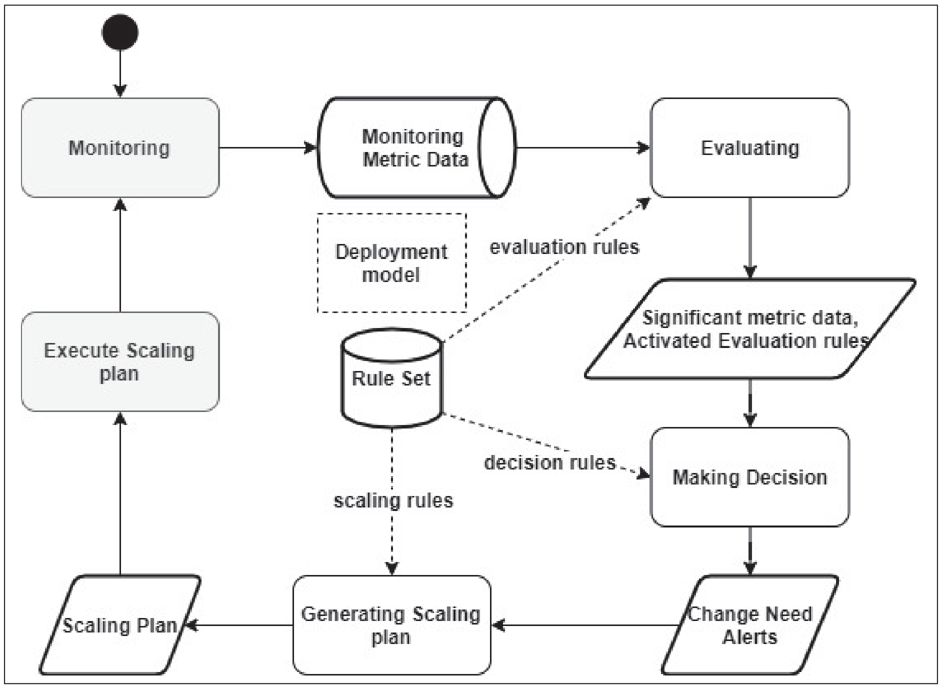


Figure 2.7 Rule Based Autoscaling Process

2.5.1 Knowledge reference

The knowledge reference represents the central element of the auto-scaling process loop. It consists mainly of the proposed deployment model and a set of rules. The deployment model makes the deployment information available to the scalability process. As an example of this information, the resource availability of the device and the deployed services on the cluster. These values of the deployment properties present an instantiation of our deployment model presented in Section 2.4. In addition, we have proposed a rule model that serves the deployment process in the steps of the evaluation, the taking decision of the scalability and the generation of the scalability plan.

2.5.1.1 General Rule model

Our auto-scaling solution relies on the use of rules to maintain the functionality of services and to meet their performance requirements while minimizing the utilization of resources.

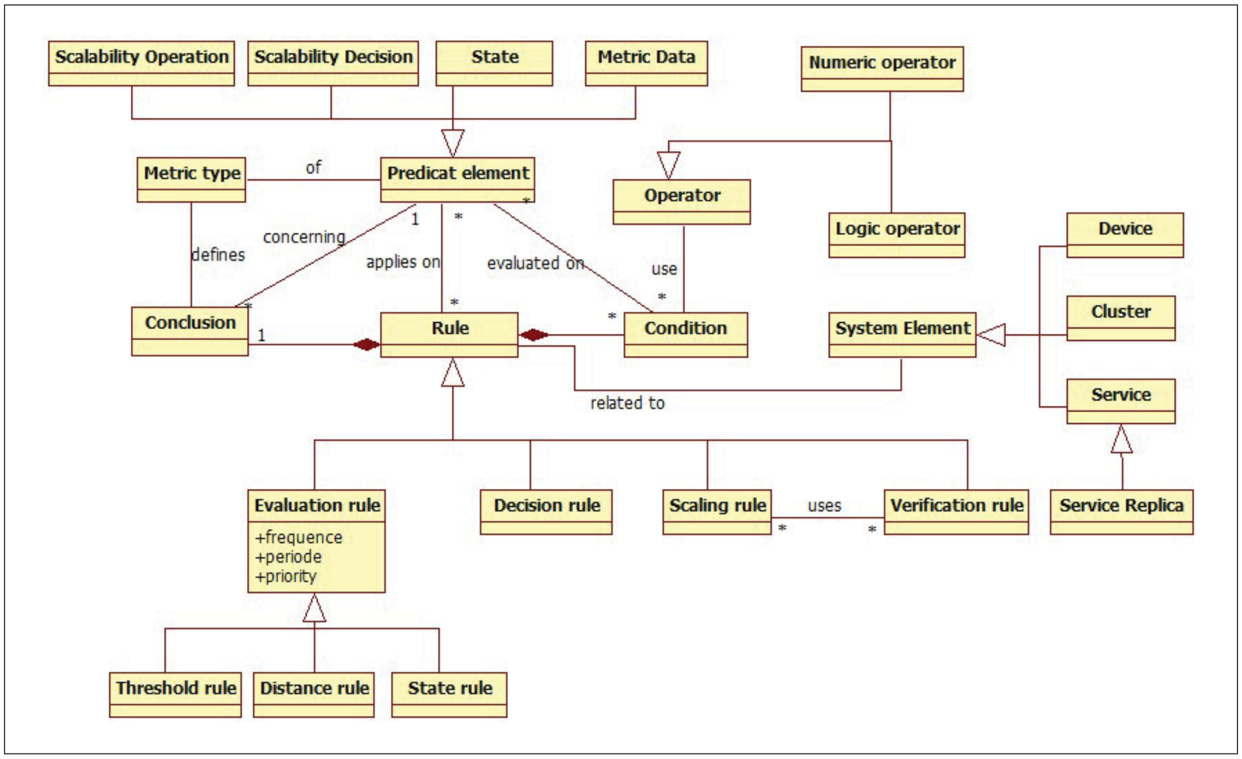


Figure 2.8 Rule model

Figure 2.8 presents our rule model. In general, a rule evaluates the condition to generate the conclusion. In the condition of a rule, predicates can be combined with the use of logical operators such as AND, OR, and Not. In addition, numeric operators may be also used such as $+$, $-$, sum and $Average$. In addition, the rules are related to the system elements such as the Device, Cluster, Service, and Service instance (i.e., replica). Based on that, the elements of condition and the conclusion are diversified (e.g. Metric data and Element state) according to the type of rules, which are the Evaluation, Decision, Scaling, and Verification rules. We will give more details about these types of rules in the description of our auto-scaling process steps.

2.5.1.2 Formal deployment information model

For performing its different steps, the auto-scaling process needs the deployment information such as the number of nodes on the cluster and the current usage of a node's memory. The first information is provided from the deployment specification; while the second information is obtained from the monitoring collected data (monitoring metric data in Figure 2.7). This information constitutes an instance of our abstract deployment model presented in Section 2.4. For that, we propose a formal representation to present this instantiation. It presents the references (i.e., concepts and properties in our deployment model) of the information used by the scalability process.

We have defined two types of information: the primitive and calculated information. The primitive information is extracted directly from the monitoring data or the deployment specification. Our formal representation called this extraction operation the 'primitive function'. Whereas the calculating function returns a calculated information. Noting that, we use the set theory for the representation of the elements with their relationships.
We start by the definition of the set of Elements E. Given the set of elements

$$E = C \cup D \cup Cl \cup S \tag{2.1}$$

Where, C, D, Cl, and S are respectively the set of containers, nodes, clusters (in the case of managing more than one cluster), and services.

Furthermore, R is the set of monitored resources as CPU and Memory. Each resource has a specific set of metrics, which is a part of a metric set M. For example, the CPU resource can have the CPU time and CPU usage percent as metrics. In addition, P is the set of performance metrics such as the response time that concerns subsets of E namely $S \cup C$.

Each recorded measured data of an element represents the values of the metrics of its monitored resources as presented in the following mathematical function:

$$\begin{aligned}
&monitorDataValues: \\
&E \times (R \times M^m)^n \to \mathbb{R}^{m \times n} \\
&(e, \{(r_1, \{m_1, \ldots, m_m\}), \ldots (r_n, \{m_1, \ldots m_m\})\}) \\
&\mapsto (v_1, \ldots, v_m, \ldots, v_{m \times n})
\end{aligned} \tag{2.2}$$

To obtain the current value of resource metric for a specific element from the monitoring data, we propose the primitive function (i.e., mathematical function) $currentValue$

$$\begin{aligned}
&currentValue: \\
&\qquad E \times R \times M \to \mathbb{R} \\
&\qquad (e, r, m) \mapsto v
\end{aligned} \tag{2.3}$$

We also define other primitive functions (mathematical function), which are presented in the following equations.

$$\begin{aligned}
instance :& S \to P(C) \\
& s \mapsto instance(s) = \{c_1, c_2, \ldots, c_k\}
\end{aligned} \tag{2.4}$$

Where $k = |\{c_1, c_2, \ldots, c_k\}|$ is the number of replicas of service (as presented in Figure 2.6). It corresponds the number of deployed containers corresponding the service image.

$$\begin{aligned}
type :& C \to S \\
& c \mapsto type(c) = s
\end{aligned} \tag{2.5}$$

Where type(c) gives the service name (i.e., identification) of the deployed container c. i.e., $c \in instance(S)$.

As presented on the model of container based service deployment in Figure 2.6. Each service $s \in S$ has a specification of resource allocation namely, required, limit, and preference as follows:

$$\begin{aligned}
require :& S \times R \times M \to \mathbb{R} \\
& (s, r, m) \mapsto= require((s, r, m)) = value
\end{aligned} \tag{2.6}$$

For example $require(db\_service, Memory, usage) = 100Mb$. The developer or operator of the system estimate that each instance of the service $db\_service$ needs at least 100 Mb of memory to work correctly. For simplicity of the representation, we will omit the metric parameters.

Similarly, we define the $preference$ and $limit$ functions. Knowing that, the preference function means the estimated resource that allows the service to work efficiently. The limit function represents the limitation (i.e., maximum) of resources that attributed to the instance (container) of the service. For example, $preference(db\_service, Memory, usage) = 120Mb$ and $limit(db\_service, Memory, usage) = 150Mb$

If these values are not defined, we can define them based on the current values or/and the history values. For example, in the case of the limit property, its value can be defined by

$$limit(s,r) = \frac{availableAbility(d,r) \times (currentValue(s,r)}{currentValue(d,r) \times |instance(s,d)|} \tag{2.7}$$

To simplify, we consider the hypothesis that these values are defined as presented in the service deployment model in Figure 2.6. Contrariwise, the currentValue can be extracted from the monitoring data as presented in Equation 2.3.
Noting that these resource specifications are the same for each instance (replicas).
For example, $require(s,r)$=$require(c,r)$.
For the set of devices (i.e., nodes), we define the following functions: $intialAbility(d,r)$ and $availableAbility(d,r)$. These functions have the same relation definition :
$D \times R \longrightarrow \mathbb{R}$
For the cluster set Cl, we define the following functions:

$$\begin{aligned} compose :& Cl \rightarrow D \\ & cl \mapsto compose(cl) = \{d_1, d_2, \dots, d_n\} \end{aligned} \tag{2.8}$$

Where, $n = |compose(cl)|$ is the number of nodes in the cluster.

$$\begin{aligned} deployed :& Cl \rightarrow S \\ & cl \mapsto deployed(cl) = \{s_1, s_2, \ldots, s_m\} \end{aligned} \tag{2.9}$$

Where, $m = |deployed(cl)|$ is the number of deployed services.

$$\begin{aligned} & instancesInDevice : S \times D \rightarrow P(C) \\ & (s, d) \mapsto instancesInDevice(s, d) = \{c_1, c_2, \ldots, c_h\} \end{aligned} \tag{2.10}$$

Where, $h = |instancesInDevice(s, d)|$ is the number of service replicas in the device d.

2.5.2 Monitoring step

To minimize the resource utilization and to ensure meeting the system requirements, our auto-scaling solution, firstly, needs to periodically monitor the system. This task can be fulfilled by any tool of monitoring (e.g. cAdvisor and Prometheus) that reports the different measured metrics such as CPU, Memory and Network usage. In our solution, we are interested in two categories of measured data. The first category is related to the resources of the device such as the CPU usage as well as the resources of the containers that are running on the device (i.e., the resource set R). The second category is related to the application or service performance metrics such as the response time and latency (i.e., the performance metric set P). The system analyzes the collected metric data in order to maintain the proper functioning of the system and to adapt it in case of necessity as we will present in the following subsections.

2.5.3 Evaluation step

Based on the metric data obtained by the previous step, this step determines the current state of different elements of the system and therefore its overall state. This step performs a preliminary operation that filters the data taking into consideration only non-redundant data. This due to the fact if a metric has the same value, then it does not make any change in the state. As we mentioned in the previous step, the input data represent metric measures of the nodes and services deployed on the cluster. The evaluation step is interested in the data concerning the following elements (Container, Node, Cluster, and Service), which could be the sources of scalability needs.

- Container: the resource attributed to the container is insufficient to perform the assigned task.
- Node: the node is overloaded in a manner that cannot scale any existing container or add/activate new one.
- Cluster: the resource of the all cluster is not enough to perform well the different deployed tasks.
- Service: the number of replications (i.e., instances) of the service is under/over the need of the service to meet the user requirement like the response time.

Now we present the evaluation of the states of the different elements of C, D, Cl, S (Equation 2.1) by using the evaluation rules. The evaluation is based on the required, preference, and limit usage properties presented in the deployment model (Figure 2.6) as well as equations 2.6 and 2.7.

In addition, our approach is flexible and the system operator can add new rules or modify the different thresholds. To represent the different rules used in our solution, this paper adopts the first order logic format. This format uses quantifiers such as $\forall$ and $\exists$, Terms (functions) that begin with a lowercase letter (e.g. $currentValue(e, r)$), and Predicates that start with an uppercase letter (e.g. $LowUsage(e, r)$).

The following equations represent a general model of the evaluation rules of the states:

$$\begin{aligned} &\forall e \in E, \forall r \in R, currentValue(e, r) \leq require(e, r) \\ &\rightarrow LowUsage(e, r) \end{aligned} \tag{2.11}$$

$$\begin{aligned} &\forall e \in E, \forall r \in R, require(e, r) < currentValue(e, r) \\ &\rightarrow MediumUsage(e, r) \end{aligned} \tag{2.12}$$

$$\forall e \in E, \forall r \in R, preference(e,r) < currentValue(e,r) \\ \rightarrow HighUsage(e,r) \quad (2.13)$$

By definition the predicat $currentValue(e,r) \leq limit(e,r)$ is always verified.

The following subsections provide more details about the limit, preference, and require properties for each element type such as Container, Device, Cluster and Service as well as their usage in the evaluation rules.

2.5.3.1 Container(instance)

This type of rule is used to evaluate the current state of the resources used by the container. The currentValue and availableResource information can be extracted directly from the monitoring data. The initialAbility information is available on the deployment model.

- $limit(c,r)$, $preference(c,r)$, and $require(c,r)$ information is available in the deployment model.
- $availableAbility(c,r) = limit(c,r)\check{}currentValue(c,r)$

2.5.3.2 Device

This category of rules is intended to evaluate the state of the used resources of the device. The $currentValue(d,r)$ information can also be extracted directly from the monitoring data and $initialAbility(d,r)$ from the deployment model.

The calculated functions are as follows:

$$\begin{aligned}
&availableAbility(d,r) = initialAbility(d,r)\breve{}currentValue(d,r)\\
&otherUsage(d,r) = currentValue(d,r) - \sum currentValue(c_i,r)\\
&require(d,r) = otherUsage(d,r) + \sum require(c_i,r)\\
&preference(d,r) = otherUsage(d,r) + \sum preference(c_i,r)\\
&limit(d,r) = otherUsage(d,r) + \sum limit(c_i,r)\\
&availableForDeployment(d,r) = initialAbility(c,r) - limit(d,r)
\end{aligned} \tag{2.14}$$

Where, the limits of each sum ($\sum$) is from $i = 1$ to $|instancesInDevice(S,d)|$ which means the number of all service instances deployed in the device 'd'(as in Equation 2.10). The function $otherUsage(d,r)$ represents the resource usage used by process other than the system's services like the operating system.

2.5.3.3 Cluster

A cluster is composed of set of devices. For that, all the information functions (e.g., $currentValue$ and $limit$) have the same format, which is the sum of those of the devices.

$infoFunction(cl) = \sum_{i=1}^{|compose(cl)|} infoFunction(d_i,r)$, where

$$\begin{aligned}
&infoFucntion \in \{initialAbility, availableAbility, currentValue,\\
&otherUsage, require, preference, limit, availableForDeploy\}
\end{aligned} \tag{2.15}$$

$compose(cl)$ is defined in Equation 2.8.

2.5.3.4 Service

The service corresponds to all its deployed instances (replica containers). Two categories of metrics are taken into consideration for the service:

- Metrics of resources (e.g. CPU and memory):
  This case is similar to the cluster resource information.
  $infoFunction(s) = \sum_{i=1}^{instance(s)} infoFunction(c_i, r)$ where,

$$infoFucntion \in \{availableAbility, currentValue, require, preference, limit\} \tag{2.16}$$

  The function $compose(cl)$ is defined in Equation 2.4.

- Performance metrics:
  These metrics that are related to the service performance such as the response time, latency, and throughput. In our actual approach, we propose to use two delimiters (i.e., thresholds): the minimum and maximum values (minValue and maxValue), as defined in the service presented in Figure 2.5. For example, the minimum threshold of the response time criterion (e.g., 30 milliseconds) defines the best requirement. In other words, the service does not require that the value to be lower than this threshold. However, it does require that the value to be less than the maximum delimiter (e.g., 120 milliseconds).
  The following rules, represent a general model to define the evaluation rules related to performance metrics of services:

$$\forall s \in S, \forall p \in P, currentValue(s,p) \leq minValue(s,p) \rightarrow MoreExpected(s,p) \tag{2.17}$$

$$\forall s \in S, \forall p \in P, minValue(s,p) < currentValue(s,p) \leq maxValue(s,p) \rightarrow AsExpected(s,p) \tag{2.18}$$

$$\forall s \in S, \forall p \in P, maxValue(s,p) < currentValue(s,p) \rightarrow UnderExpected(s,p) \tag{2.19}$$

Our approach introduces and focuses on the concept of states. It retains the use of other types of rules, most notably those based on thresholds, which are mostly used in the literature and in the technical tools.

Direct threshold rules:

In this category, the rules are based on specific metric values. The preconditions of the rule are based on a basic comparison (i.e., <, ≤,=, ≥, >) as presented in the followed general formula:

$$\begin{aligned}
&\forall e \in E, \forall r \in R, \exists (e, r, comp, thV) \in (E \times R \times CompO \times \mathbb{R}), \\
&CompO(currentValue(e, r), thresholdValue(e, r)) \\
&\rightarrow Significant(currentValue(e, r))
\end{aligned} \tag{2.20}$$

Where the quadruplet $(E \times R \times CompO \times \mathbb{R})$ defines the set of thresholds which is based on the specification of the element (e.g. a device), the resource (e.g. the memory), the comparison operator (e.g. '≥'), and the threshold value (e.g. $120 Mbytes$).

Distance change rule:

The distance change rule is activated when the difference between the current metric value and the previous one exceeds a predefined and configurable threshold. Consequently, this can be interpreted as the speed of value change. Therefore, by this type of rule, the system can deduce the necessity of scaling even the current metric values are reasonable (i.e., they do not activate the state and direct threshold values). The following equation presents the general model of this type:

$$\begin{aligned}
&\forall e \in E, \forall r \in R, \exists (e, r, thV) \in (E \times R \times \mathbb{R}), \\
&CompO(|currentValue(e, r) - oldValue(e, r)|, distanceThValue(e, r)) \\
&\rightarrow Significant(currentValue(e, r))
\end{aligned} \tag{2.21}$$

Where, 'CompO' verifies whether the difference between the current value of the resource usage with the old usage exceeds the distance threshold value defined by 'distanceThValue'.

2.5.3.5 Evaluating algorithm

Algorithm 2.1 follows mainly three steps:

1. *Data filtering*(Lines 1 to 2): by the $filter$ operation, the algorithm eliminates the useless data such as the repeated data(i.e., duplicated data). For that, it needs to update the old data after each filtering operation (Line 2)).
2. *Data extraction*: in Line 3, the algorithm directly extracts the primitive data (corresponding to primitive functions) from the monitoring data MD. Based on the extracted data, Line 4 calculates the other data $Calcul\_Data$ such as those related to the cluster element, by using the different equations presented in this paper.
3. *Apply evaluation rules*(Lines 5 to 13): The algorithm evaluates the condition of rules on the prepared data (primitive and calculated data). As indicated in Line 6, it uses the deployment model $DeplModel$ to check other properties of the rule such as the period (ex. 3 minutes) and the rate(95%). These last metrics indicate that the rate of its verification is greater than the rate value during the period value.

   In the case of its verification, it adds the rule to the activated rule list ($Activated\_rules$) After that, it updates the state of the related element with the specified resource in the rule.

Algorithm 2.1 Evaluating

```
Input: $MD \langle E, R, currentValues, oldValues \rangle$,
       $DeplModel \langle E, Properties, Values \rangle$, $Rules \langle R \rangle$
Output: $ActivitedRules(R) = \{\}$, $ElementStateByRes \langle E, R, State \rangle = \{\}$

1  $Filtred\_MD = filter(MD)$;
2  $updateOld\_MD(MD)$;
3  $Primit\_Data \leftarrow getPrimitives(Filtred\_MD)$;
4  $Calcul\_Data \leftarrow calculate(Primit\_Data, DeplModel)$;
5  for $r \in Rules$ do
6      if $check\_rule(r, Primit\_Data, Calcul\_Data, DeplModel)$ then
7          $Activated\_rules = Activated\_rules \cup \{r\}$;
8          if $type(r) = `State'$ then
9              $(e, res, s) = getElementState(r)$;
10             $ElementStateByRes.update(e, res, s)$ ;
11         end if
12     end if
13 end for
```

2.5.4 Making decision step

This step analysis the result of the evaluating step to take decision about the need of the scalability (whether scale up or down) of each system element. It based on the decision rules to allow mapping between the results of the evaluation rules (especially the element state) and the decision of the scalability.

In our solution, we focus on the state evaluation rule type that brings more autonomy to the system for the scalability concern. Otherwise, in the case of the evaluation generated by the other rule types (i.e., the direct threshold and distance change rule), the making decision step translates directly their results such as the need of scalability that is specified by the system operator.

The following evaluation presents some generic decision rules:

$$\begin{aligned}&\forall e \in (Cl \cup D \cup C), \forall r \in R, State(e,r) = \text{‘}HighUsage' \\ &\rightarrow NeedScalability((e,r), \text{‘}Up') \end{aligned} \tag{2.22}$$

$$\begin{aligned}&\forall e \in (Cl \cup D \cup C), \forall r \in R, State(e,r) = \text{‘}LowUsage' \\ &\rightarrow NeedScalability((e,r), \text{‘}Down') \end{aligned} \tag{2.23}$$

$$\begin{aligned}&\forall s \in (S \cup C), \forall p \in P, State(s,p) = \text{‘}UnderExpectation' \\ &\rightarrow NeedScalability((e,r), \text{‘}Up') \end{aligned} \tag{2.24}$$

$$\begin{aligned}&\forall s \in (S \cup C), \forall p \in P, State(s,p) = \text{‘}OverExpectation' \\ &\rightarrow NeedScalability((e,r), \text{‘}Down') \end{aligned} \tag{2.25}$$

The rules are based on the state evaluations generated in the previous evaluation step, which is used as an input in the taking decision algorithm:

Algorithm 2.2 Taking decision algorithm

```
Input: $ElementStateByRes\,\langle E, R, State\rangle$, $DecisionRules\,\langle P(R)\rangle$
Output: $ActivitedRules(R) = \{\}$, $Scal\_Decision\,\langle E, R, Sense\rangle = \{\}$

1 for $r \in DecisionRules$ do
2     if $check\_rule(r, ElementStateByRes)$ then
3         $e = getElement(r)$;
4         $res = getResource(r)$;
5         $sense = getSense(apply(r))$;
6         $Scal\_Decision.update(e, res, sense, `Worst')$;
7         $ActivitedRules(R) = ActivitedRules \cup \{r\}$;
8     end if
9 end for
```

Algorithm 2.2 checks the different decision rules (Lines 1-9). For each verified rule, it extracts (Lines 4-6) the concerned elements namely the element and the resource as well as the result of the applying rule. The algorithm uses the worst choice strategy to manage the case of multiple and conflict scalability decision generated by different rules for the same element and resource. It chooses the scale-up option instead of the scale-down, because it reflects a need to continue operating the system and/or to meet user requirements. On the contrary, the scale-down is important in terms of unloading the system's resource usage, but it has less priority than the user requirements.

2.5.5 Generating scaling plan step

This step generates the overall scalability plan based on the elementary decisions of the system elements obtained in the previous step. To do that, it aggregates these elementary decisions and manages the potential conflicts between them.

Example, an instance c_1 of a service s wants to scale-up for one performance metric $p \in P$ (i.e. $NeedScalability((c_1, p), 'Up')$). On the other hand, another instance c_2 of the same service s wants to scale-down for the same metric p (i.e., $NeedScalability((c_2, p), 'Down')$) because its performance metric is over expectation. Furthermore, this step should verify, via verification rules, that the generated plan meets the system constraints such as the available resources. In order to reduce the complexity of the task of generating the scalability plan, we follow the following heuristic strategies:

- *Order of element types:* This strategy firstly considers the aggregating elements such namely the service and the cluster. The primitive and calculation functions of the aggregating elements give to the system an overall indicators of the need and the ability of the scalability. For example, if one service wants to scale-up by adding a new instance and the available resources of the cluster are sufficient to satisfy this need; the system can accept its request, as presented in the following scalability rules:

$$\begin{aligned} &\forall s \in S, \forall r \in R, \exists p \in P, \exists th \in \mathbb{R} \\ &NeedScalability((s,p), `Up') \wedge (availableAbility(cl,r) > th) \\ &\rightarrow ScaleUp(s) \end{aligned} \tag{2.26}$$

  Where the service s is deployed in cl, which is the current cluster. The th presents a resource availability threshold (e.g., 0) that allows to scale up a service s.

- *Order of the scalability validation:* the first order is from the aggregating element to elementary elements as presented in the previous point. The second one is between the type of elements, where the generator starts from software element type (i.e., service and container) to the hardware element type (i.e., node and cluster).
  This order is justified by the impact of the scalability of software elements on the hardware element.

- *Order of the scalability operation:* in this strategy, we give the scale-down operation a higher priority compared to the scale-up operation if they do not concern the same element. The scale-down allows to drop the corresponding used resource. By taking into consideration the dropped resources, the system can satisfy the new scale-up requests. But the system takes the reverse order if the scalability operations concern the same element or the same type such as the instances of the same service. For example, if a service needs to scale-up for one performance metric and to scale-down for another one, the system cannot choose the scale-down before finishing the scale-up request.

2.5.6 Execute the scalability plan step

The role of this step is to execute the scalability operations recorded in the generated plan. This task can even be performed by an external tool. For example, in the validation of our approach, we have used Jenkins pipeline tool.

2.6 Evaluation

This section highlights the steps followed to assess the feasibility and the performance of our approach with resource-constrained devices as in the case of IoT edge environment. We used Docker, Docker-Machine [1], and Docker-Swarm [2] to carry out our experiment. Docker-Machine is used to create a set of limited resource virtual machines (VMs) configured with Docker engine to simulate IoT edge nodes. Docker-Engine is used to create containers that run services deployed on the VMs, as well as, to run the deployed management and monitoring tools. The Docker-Swarm is used to create a cluster of VMs, one of them is deemed as a manager node, while the rest is worker nodes. Some criteria are considered to evaluate the obtained results namely the scalability, the response time, and the resource utilization.

2.6.1 Evaluation Architecture

The implementation and the deployment of our approach is based on a set of tools that are widely used in the DevOps and Container community namely the monitoring tools NodeExporter, cAdvisor, Prometheus [3], and Grafana [4] as well as the management tools such as Jenkins [5] and AlertManager [6].

[1] https://docs.docker.com/v17.09/machine/get-started/

[2] https://docs.docker.com/engine/swarm/

[3] https://prometheus.io/

[4] https://grafana.com/

[5] https://jenkins.io/

[6] https://prometheus.io/docs/alerting/alertmanager/

As shown in Figure 2.9, our evaluation architecture is encompassed of set of modules that are formed as virtual machines (VMs) and docker containers. Namely, three VMs are used including one as a manger node, while the other two machines as worker nodes. Each node in the cluster is equipped with Linux 2.6 as an operating system, 1-core CPU, and 1 GB of RAM. In addition, it is equipped by a set of services deployed as docker containers. CAdvisor and Node-exporter are deployed in each node to collect and to export the different metrics. In the node manager, Prometheus service scrapes the collected data from all Node-exproter and Cadviser instances. Communications between edge nodes and services are maintained by means of ports run on the top of the host machine's IP. In addition, containers are used to run management tools namely

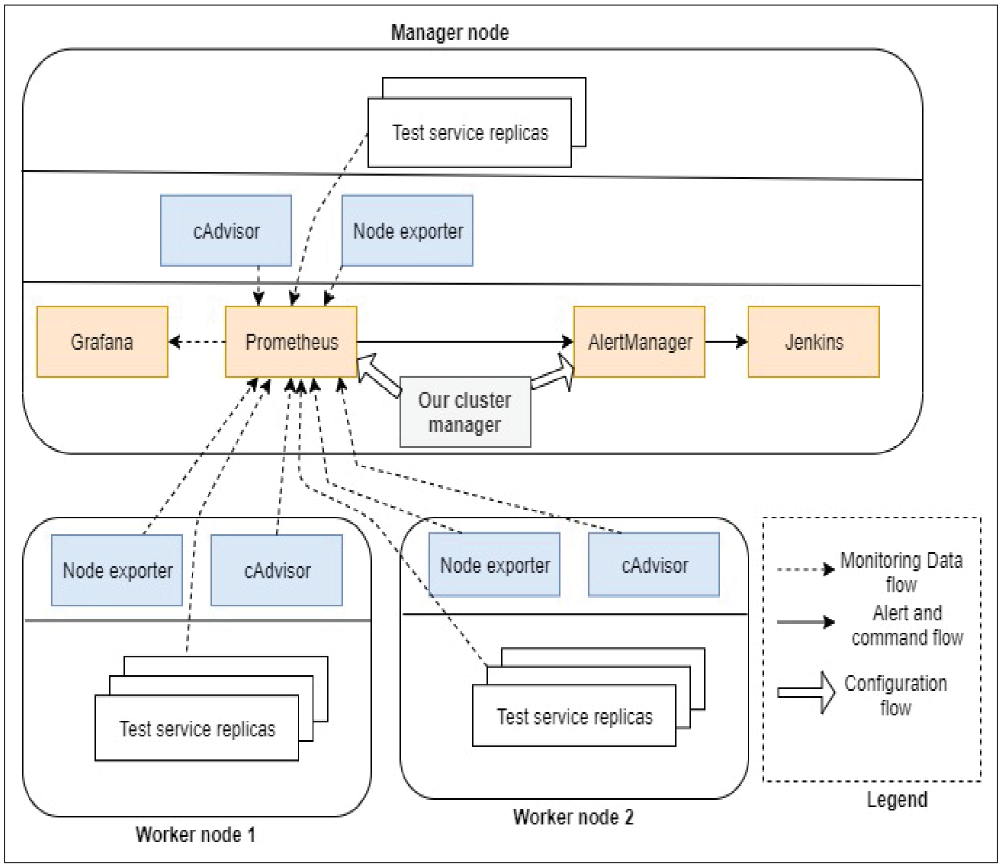


Figure 2.9 The evaluation architecture

AlertManager and Jenkins to handle the different alerts and adapt the client service (i.e., Test service). Our cluster manager service ensures that the architecture functioning is consistent with our proposed approach.

From this perspective, we explain more this evaluation architecture in line with our approach:

- Monitoring step: to monitor the different nodes and the deployed services, Prometheus scrapes the monitoring data metrics from the exporter tools namely CAdvisor and Node-exporter as well as from the instances of our instrumented test service. While Grafana is in charge of visualizing these metrics. Grafana tool is used to present in graphical way the monitoring data.
- Evaluating step: our rule management service configures Prometheus to evaluate the monitoring data as specified in our rule model. To do that, our cluster manager service creates Prometheus alert rules based on the conditions and conclusions of the evaluation rules.
- Taking decision and scaling steps: Alertmanger receives alerts from Prometheus. Our cluster manager service also configures the mapping between the Prometheus alerts and their Alertmanger handlers. The existence of the alert's handler means that there is a scaling decision. The handlers generate the corresponding actions. In our implementation, these actions are the scalability operation such as the scale-up and scale-down, which we have implemented in the Jenkins tool. Consequently, Jenkins executes the scaling operation by adapting the current test service deployment.

2.6.2 Evaluation Scenario

The scenario of our experimentation is as follows:

1. Configure the cluster nodes with the aforementioned tools such as our manager service, cAdvisor, and Prometheus. Also, start monitoring the utilization of the different resources such as CPU, Memory, and Network traffic.
2. Deploy a test service with 03 replicas. The test service uses an instrumented metrics [7] to mainly send the response time metric, as well as the service name and the response

[7] https://prometheus.io/docs/practices/instrumentation/

code. The service receives HTTP requests and record the performance metrics when the response is sent back to the client. By this way the system (via Prometheus) can monitor the performance metrics of the deployed service.

3. Impose a pressure on the service to increase significantly its response time and observe the scalability of the system by monitoring the number of test service replicas (i.e., instances) and the system resource utilisation. This is achieved by sending HTTP requests (10000 requests) with a delay parameter (500 ms). The test service has for the minReplicas and maxReplicas properties, 01 and 05 respectively. In addition, it has 0.025 ms and 0.1 ms values for the minValue and maxValue response time metrics.

2.6.3 Results

This section presents the results of our experimentation, which carried out based on the above mention evaluation configurations. Two main evaluation objectives that our approach seeks to demonstrate namely the auto-scalability of services and the reduction of resource utilization. The first objective is intended to meet the performance metrics; while the second is intended to meet the resource-limitation requirement.

2.6.3.1 Auto-scalability behavior

To monitor the auto-scalability behaviour we have used the number of service replicas metric. Figure 2.10 presents an extract during the second step of the evaluation scenario. The service descales to the one replica (i.e., minReplicas) because the response time is over expectation. In other words, the response time value < minValue (i.e., 0.025). In contrast, in Figure 2.11 the service scale-up to 5 replicas as long as the response time is under expectation because its value $> 0.1ms$ (i.e., maxValue).

Obviously, in our current implementation, the system scales by only one replica each time. This point is to be improved by taking into account the historical behaviour of the system in order to adapt to the corresponding degree of scalability (number of replicas).

2.6.3.2 Resource utilization

Figures 2.12, 2.13, and 2.14 represent, respectively, the utilization of CPU and Memory, and Network traffic. They show the average resource utilization by the nodes during the auto-scalability of the service. The figures show that the resource utilization is directly proportional to the number of replicas. Consequently, the experimentation proves the notable enhancement in the resource utilization when the service is descaling. Since the nodes run an operating system and other services such as monitoring services, there is still a remarkable utilization with the minimum of service replicas (one instance per node).

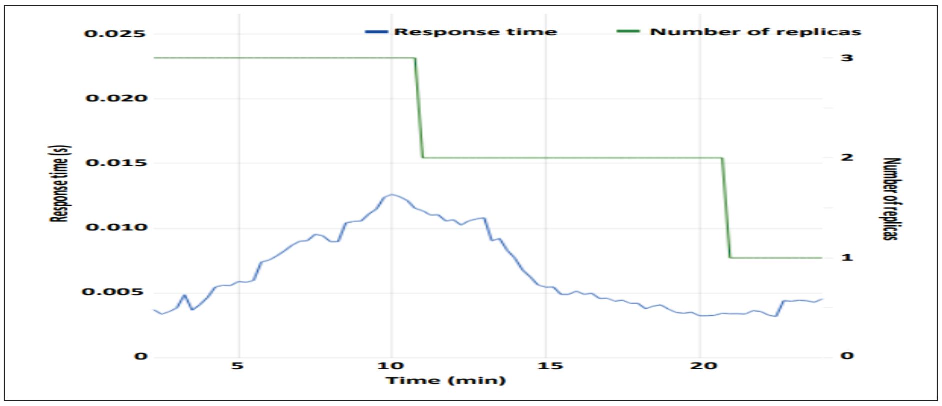


Figure 2.10 Scalability when the response time is Overexpectation

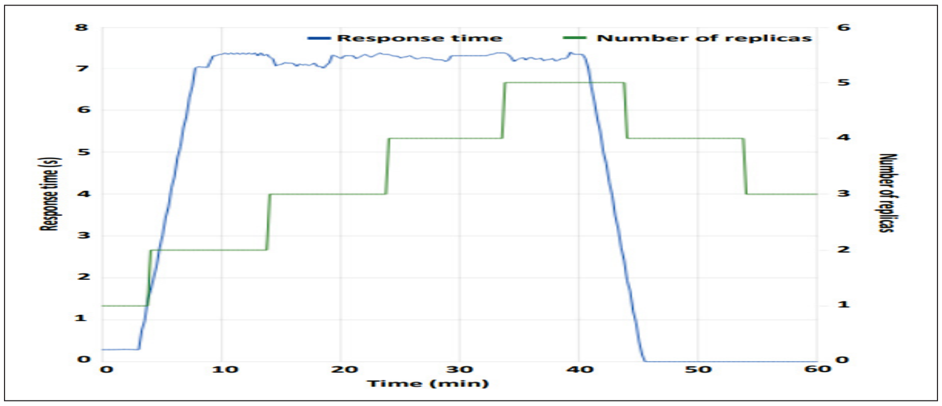


Figure 2.11 Scalability when the response time is Underexpectation

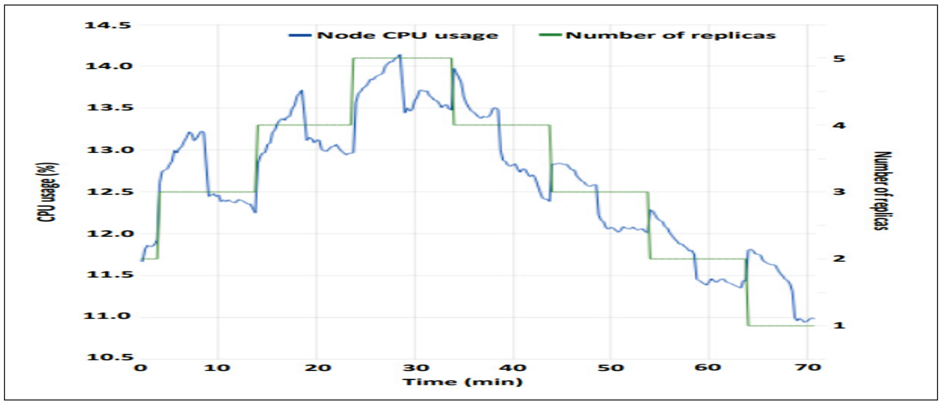


Figure 2.12 CPU usage during the scalability

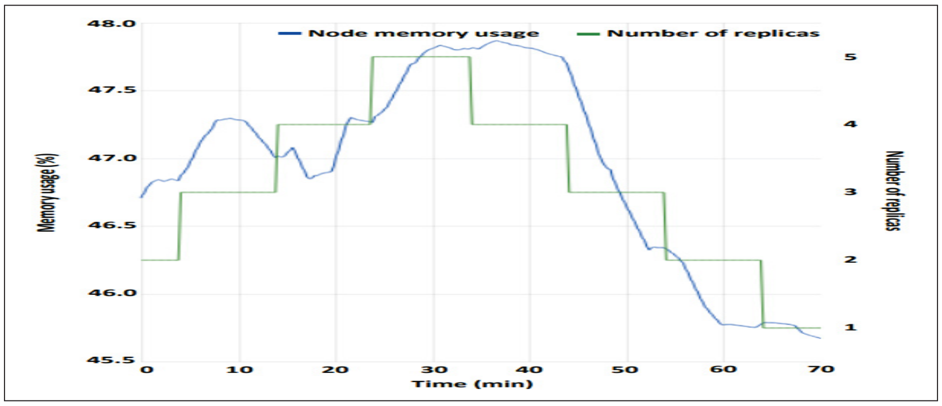


Figure 2.13 Memory usage during the scalability

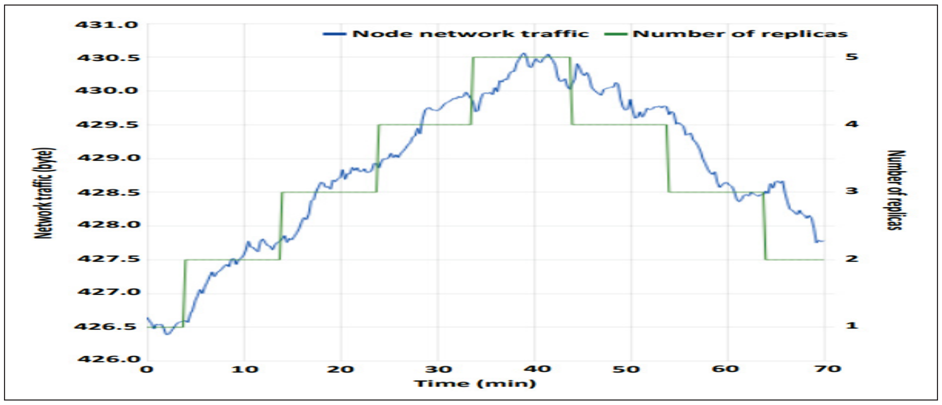


Figure 2.14 Network traffic during the scalability

2.7 Conclusion

Edge devices have conveyed significant benefits to IoT networks by taking the charge of performing some cloud tasks, by which alleviating the workload imposed on the network and improve system responsivity. However, devices at the edge of IoT networks are featured by its resource limitation, as well as, it is likely to perform within an extreme heterogeneous environment. This has imposed more restrictions especially in terms of service deployment and resource management. This paper presents a solution based on lightweight virtualization technologies namely containers to facilitate efficient auto-scaling of service deployment on IoT edge devices. The proposed approach adopts the cluster concept to define grouped IoT edge devices, and it follows the MAPE-K loop to monitor the state of the cluster and the performance of services deployed on its devices. In addition, the rule model is proposed to facilitate the auto-scaling process of services. More specifically, it is designed to serve as reference knowledge base used by the monitoring and analysis modules of our solution. Thus, a deployment plan will be generated and transferred to the execution module. The obtained results proved the efficiency of our approach in terms of auto scaling of services in corresponding to service performance metrics (e.g. response time); while optimizing the resource utilization namely CPU, memory and network traffic. As future work, we plan to perform more experimentation in real IoT scenarios use cases considering more performance metrics such as throughput and latency. In addition, to expand the feasibility of our approach, it would be of great significant to investigate multi-level clusters, which will be of our interest as future work. Furthermore, we are planning to improve the scaling plan generation step by integrating machine learning techniques to analysis the historical behaviour of the system.

CHAPTER 3

IMPROVING FORECASTING ACCURACY AND OSCILLATION MITIGATION FOR PROACTIVE SERVICE AUTO-SCALING USING AUTOMATIC DATA FEATURIZATION

Ahmed Bali[1] , Yassine El Houm[1] , Abdelouahed Gherbi[1]

[1] Department of Software Engineering and IT, École de Technologie Supérieure
1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3

Paper submitted to the Journal of King Saud University - Computer and Information Sciences.

ABSTRACT Edge computing has gained widespread adoption for time-sensitive applications by offloading a portion of IoT system workloads from the cloud to edge nodes. However, the limited resources of IoT edge devices hinder service deployment, making auto-scaling crucial for improving resource utilization in response to dynamic workloads. Recent solutions aim to make auto-scaling proactive by predicting future workloads, overcoming the limitations of reactive approaches. These proactive solutions often rely on time series data analysis and machine learning techniques, especially Long Short-Term Memory (LSTM), thanks to its accuracy and prediction speed. However, existing auto-scaling solutions often suffer from oscillation issues, even when using a cooling-down strategy. Consequently, the efficiency of proactive auto-scaling depends on the prediction model accuracy and the degree of oscillation in the scaling actions.

This paper proposes a novel approach to improve prediction accuracy and deal with oscillation issues. Our approach involves an automatic featurization phase that extracts features from time-series workload data, which improves forecasting accuracy. These extracted features also serve as a grid for controlling oscillation in generated scaling actions. Our experimental results demonstrate the effectiveness of our approach in improving prediction accuracy, mitigating oscillation phenomena, and thereby enhancing the overall auto-scaling performance.

Keywords: Container, Auto-scaling, LSTM, Oscillation mitigation, Data featurization, Time series Forecasting

3.1 Introduction

The Internet of Things (IoT), which promotes integration between objects in the real world and services in the digital world, is influencing many aspects of our lives, such as health, education, and construction. These uses rely primarily on networks composed of a large number of tiny devices immersed in our environment, for example, in the form of environmental and health sensors. The use of these devices is constantly growing, leading to a massive number of devices connected to the internet (Evans, 2011). The massive data generated by IoT devices and their limitations in computing and connectivity capabilities may increase the latency of services.

Edge computing, which offloads the workload of IoT systems from the cloud to edge nodes, improves system responsiveness by minimizing latency. However, IoT devices at the edge network are typically resource-constrained and heterogeneous, which can constrain the ability to deploy services to IoT devices.

To address the heterogeneity of IoT technologies, lightweight virtualization technologies, such as containers, have been extensively employed to facilitate the deployment and management of microservices on edge IoT devices (Ahmed *et al.*, 2019). The container can package the service program with all its dependencies into a single module. Thus, services can be run stably and faster, regardless of the operating environment.
Additionally, IoT edge devices within the same cluster can share resources and communicate with each other via virtual networks thanks to container orchestration techniques such as Swarm (Swarm, 2022). Furthermore, Kubernetes (kubernetes, 2022b) considers the concept of a pod, which is a collection of one or more containers that share network and storage resources and adhere to certain operating rules. A Pod presents the deployable primitive unit of computation. In this context, the service deployment is presented by deploying a set of replicas (i.e., containers or pods) on the available machines that are grouped into clusters. Each replica presents an instance of the microservice to be deployed. Increasing the number of replicas improves the service's responsiveness (i.e., reduces latency) but consequently increases the usage of computation resources.

Resource usage should be further considered at the edge level, where devices are often limited in terms of resources. This requires a reasonable use of computing resources while meeting user requirements (e.g., response time). Therefore, it is necessary to have, dynamically and continuously, a number of replicas that does not exceed the need (i.e., over-provisioning) and that is not less than the need (i.e., under-provisioning).

However, the current container tools (e.g., Swarm (2022) and kubernetes (2022b)) need to be more elastic to scale deployed services automatically. Therefore, this reduces their ability to continuously adapt in response to the operational environment, such as the frequent changes in the requested workload (e.g., HTTP requests). In addition, the existing approaches lack the proactivity aspect, which limits the system's ability to adapt appropriately to the operational environment.

Container-based autoscaling solutions are still an open issue that needs to be addressed, as mentioned in Qu *et al.* (2018). Designing and implementing an efficient and adequate auto-scaler for containerized services is challenging due to various factors, including dynamic workload characteristics, resource constraints, and the distributed nature of IoT nodes at the edge.

Workload forecasting is an important factor in making auto-scaling behavior proactive. Therefore, forecasting accuracy improvement is necessary to increase the auto-scaler performance in generating actions (scale up or down) more adequately to the workload.

Moreover, the continuous generation of actions by the auto-scaler leads to a dynamic change in the number of replicas, which generates an oscillation issue. The oscillation of the number of replicas has considerable consequences on the system's performance, such as cost increases in resource usage in the case of over-provisioning and performance declines in the case of under-provisioning. Moreover, adapting the number of replicas can generate latency in the system if performed reactively.
Related work generally adopts the solution based on the cooling down strategy, which introduces a delay before carrying out a scale-down request following a decrease in the workload volume. However, this strategy is demanding since it depends on optimizing the cooling down period

parameter. If the period is long, it generates more over-provisioning; if it is short, that reduces the efficiency of the oscillation mitigation.

To address the above issues, this work proposes a service auto-scaling approach that maintains the desired performance with optimal resource utilization to cope with cases of dynamic workload change (i.e., increase/decrease). Our approach is built upon several key contributions, which are as follows:

- Adoption of the MAPE-K algorithm (Monitor, Analyze, Plan, and Execute) as a controlling loop: We have implemented the different steps of the MAPE-K algorithm, allowing us to effectively monitor system behavior, analyze data, plan appropriate actions, and execute scaling operations.
- Proposal of data featurization approach: We introduced a data featurization operation that enhances the accuracy of the forecasting model. By extracting automatically relevant features from the data, we transform it from univariate to multivariate, thereby capturing more information and improving the predictive capabilities of the model. The features extracted are inspired by Japanese candlesticks, a widely used technique in the trading domain.
- Proposal of a grid-based approach for oscillation mitigation: Our approach includes a novel grid-based mechanism for mitigating oscillations in system scaling. This mechanism takes advantage of the data and features used for workload prediction improvement and is based on an economic concept known as the grid technique. By aligning the grid lines with the extracted features, we achieve effective oscillation mitigation. Notably, our mechanism is parameterless compared to existing techniques proposed in related work.
- Evaluation of our approach using widely-used datasets: To validate the effectiveness of our approach, we conducted comprehensive evaluations using datasets commonly used in the literature on the auto-scaling of systems.

The remainder of the paper is organized as follows: Section 3.2 discusses the related work to the approach proposed in this paper. Then, Section 3.3 shows the overall architecture of our auto-scaling approach, followed by the presentation of the collection and pre-processing of data in Section 3.4. Afterward, Section 3.5 presents our featurization approach. Our LSTM model for forecasting the future workload is presented in Section 3.6. Section 3.7 presents our oscillation mitigation approach. Section 3.8 discusses the experiments to evaluate our contributions. Finally, we conclude this study and highlight our directions for future work in Section 3.9.

3.2 Related Work and Background

Many works are performed on auto-scaling at the cloud level in the literature. In (Kovács, 2019), auto-scaling is defined as a method used in distributed computing, especially in the cloud, to dynamically and automatically adjust the amount of computing resources in a set of servers based on traffic workload. Additionally, scaling is a crucial component of orchestration in terms of policy and flexibility for cloud containers and virtual machines. To explain the auto-scaling feature, we take as an example (similar to that presented in (Mishra *et al.*, 2020)) the number of servers running behind a web application that can be automatically increased or decreased depending on the number of active users. Since these measurements vary considerably during the day and servers are a limited resource, it is often worthwhile to operate a sufficient number of servers to support the current load. Auto-scaling is very useful for meeting customer service requirements. It reduces the number of active servers when activity is low and launches new servers when activity is high.

3.2.1 Virtualization and Autoscaling

The cloud is based on virtualization technology, which allows performing multiple work environments on the same server. In this regard, (Varghese & Buyya, 2018; Pahl *et al.*, 2017) explain, in a simple and detailed way, the virtualization architecture, presenting the trends and technologies involved in the cloud. Containers are considered a fundamental virtualization property, which allows deploying microservices on the cloud server. Several domains increasingly use containers, including service meshes, edge and fog computing, IoT, smart cars, and smart cities, as in Ahmed *et al.* (2019); Jamshidi, Pahl, Mendonça, Lewis & Tilkov (2018); Khazaei *et al.* (2017); Morabito *et al.* (2017).

There are essentially two types of autoscaling, vertical autoscaling and horizontal autoscaling (Al-Dhuraibi *et al.*, 2017). The distinction between these two types of scaling stems from the manner in which computing resources are added to the infrastructure. In vertical autoscaling, computing power is added to existing replicas/nodes. In contrast, horizontal autoscaling increases a system's capacity by adding more replicas (e.g., containers) to the environment, allowing for processing and memory load sharing across multiple devices.

In the edge computing context, resources are often limited (e.g., IoT devices and Gateway devices), which makes having a mechanism for increasing and decreasing computational resources (i.e., vertical auto-scaling) on the same node less useful or even unrealistic. Therefore, horizontal auto-scaling becomes more suitable by dynamically changing the number of replicas (e.g., containers or pods) to distribute the processing load among devices that constitute a so-called cluster.

Furthermore, auto-scaling approaches are classified into two types: reactive and proactive. As in our previous work (Bali *et al.*, 2020), the algorithm reacts to workload or resource utilization in real-time according to a set of predefined rules and thresholds. In proactive scaling, an algorithm is used to forecast future workload based on historical data (Lorido-Botran *et al.*, 2014). The difference is that, in proactive mode, the auto-scaler must predict workloads to adapt the system to future needs, while in reactive mode, the system reacts in real-time to workload changes (Lorido-Botran *et al.*, 2014).

Due to the ease of implementation, most current auto-scalers use reactive threshold-based approaches, as used in Kubernetes HPA, Google Cloud Platform, Amazon EC2, and Oracle Cloud. For this purpose, many studies (Klinaku *et al.*, 2018; Taherizadeh & Stankovski, 2019; Nguyen *et al.*, 2020) suggest using the reactive auto-scaling functionality offered by cloud servers. However, selecting appropriate thresholds is difficult, especially when dealing with complex workloads (Imdoukh *et al.*, 2019). To optimize the configuration of thresholds, auto-scalers can use static heuristic techniques offline according to predefined workloads (Zhong & Buyya, 2020). These strategies are unable to cope with highly dynamic workloads in which applications must scale at runtime (Zhong *et al.*, 2022).

In addition, although this improved reactive approach is simple, it is less efficient since it generates oscillations due to sudden and unpredictable changes in workloads. As a result, the reactive approach results in waste due to the over-provisioning of resources and degradation of the system's performance when releasing the resources that the system needs.

In (Dang-Quang & Yoo, 2021), the authors demonstrate that their proactive auto-scaler outperforms Kubernetes' default horizontal autoscaling pod (HPA) in terms of accuracy and speed when provisioning and de-provisioning resources.

3.2.2 Workload Forecasting Techniques

In proactive auto-scaling as in (Sangpetch *et al.*, 2017; Imdoukh *et al.*, 2019; Dang-Quang & Yoo, 2021), machine learning algorithms are often applied in time series analysis for workload forecasting. Different ML algorithms are used to predict the future from historical data (Lorido-Botran *et al.*, 2014). The time series-based forecasting approaches bring more performance compared to regular regression approaches, as they are specifically designed for forecasting tasks. These approaches explicitly consider the sequential nature of the data and incorporate past observations to make predictions, taking into account temporal patterns. In contrast, regular regression models, such as Linear and Polynomial Regression, typically assume a constant relationship between the input variables and the target variable, without explicitly accounting for these temporal patterns. The literature uses two common categories of time series data analysis and forecasting methods.

First, algorithms are based on statistical time series analysis (e.g., ARIMA) such as those presented in Lorido-Botran *et al.* (2014); Sangpetch *et al.* (2017); Calheiros *et al.* (2014); Roy *et al.* (2011); Kan (2016); Li & Xia (2016); Ciptaningtyas *et al.* (2017); Meng *et al.* (2016). These statistical approaches are slow in the case of dynamic workload demands and suffer from resource overuse (Imdoukh *et al.*, 2019). Since most of this work is intended for the cloud environment, the application of these techniques in edge computing is restricted by resource limitations.

Second, there are deep learning-based solutions such as the neural network (ANN) and LSTM algorithms, for instance, in Calheiros *et al.* (2014); Goli *et al.* (2021); Zhu *et al.* (2019); Saxena & Singh (2022); Kumar *et al.* (2021a); Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021). In Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021), the experimental results show that LSTM model predicts as accurately as the ARIMA model but with a faster prediction speed.

However, the efficiency of auto-scalers using time series data analysis is highly dependent on prediction accuracy (Doan *et al.*, 2019). This accuracy, in turn, is dependent on parameters such as the workload pattern, history windows (Lorido-Botran *et al.*, 2014) as well as a machine-learning model and the prediction horizon. Thus, it is required to propose solutions that improve prediction accuracy. Therefore, container-based autoscaling is still an open issue that needs to be addressed, as mentioned in Qu *et al.* (2018); Cardenas (2018).

In order to improve the accuracy of the prediction, our approach proposes an automatic solution based on the feature extraction (i.e., featurization) of the data. The extracted features (such as maximum and minimum) give a general description of the data window to be used to predict the future workload. By adding these features, the accuracy of the prediction model has been greatly improved, as shown in the evaluation section (Section 3.8). Unlike other solutions (such as Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021)), our model becomes multivariate, which means that it uses, in addition to historical workload values, the automatically generated features. This automatic generation makes our model less required in terms of data preparation compared to other classical multivariate approaches that often need the collection process of other features as we will discuss in the next element.

3.2.3 Univariate and Multivariate Time Series

While many auto-scaling studies have traditionally focused on univariate time series data, such as workload, it has been widely recognized in the literature that incorporating multivariate data (i.e., features) can significantly improve forecasting accuracy. Cetinski & Juric (2015) demonstrated the importance of extending training data with relevant features, such as the time of day and weekends.

In the context of auto-scaling, LSTM models have been shown to effectively capture complex non-linear feature interactions when applied to multivariate data with numerous dimensions and a substantial volume of data (Ogunmolu *et al.*, 2016). Laptev *et al.* (2017) proposed a novel LSTM architecture that leverages an autoencoder for feature extraction, achieving superior performance compared to the vanilla LSTM model. In their data preparation process, they incorporated additional specific features such as weather information (e.g., precipitation, wind speed, temperature) and city-level information (e.g., current trips, current users, local holidays). However, most of these additional features cannot be automatically extracted and need to be logged during data collection.

Various classical statistical time series features have been considered in the literature to improve forecasting accuracy. Hyndman *et al.* (2015) explored features such as mean, variance, ACF (Auto-correlation Function), trend strength, linearity, peak, and season. Di *et al.* (2012) focused on important and predictive statistical properties of host load, including mean load, load fairness index, noise-decreased fairness index, and N-segment pattern. However, these derived features, particularly those related to trend and seasonality, usually require manual analysis to identify their parameters. Chakraborty *et al.* (1992) emphasized the significance of considering correlations among different metrics to improve prediction accuracy and avoid the distortion of forecast models. They attempted to select appropriate features, such as disk space, disk IO time, memory, and CPU.

To further illustrate the significance of incorporating relevant features, Wang *et al.* (2021) established a dataset by collecting features of complex system simulation to improve the resource prediction performance of simulation applications in the cloud. These features include average, maximum, and minimum values of usage metrics such as CPU, memory, file system, network (receive and send bytes), communication delay, and execution time. Similarly, Kao *et al.* (2020)

focused on communication metrics, specifically incoming traffic, outgoing traffic, number of connections, and network traffic load (per day). These features need to be obtained during data logging since they are not derived automatically.

In our approach, instead of relying on pre-existing multivariate features, we propose the automatic extraction of features from the univariate data presented within each data window using non-linear functions. Inspired by Japanese Candlesticks, a well-known technique used in the trading domain, we apply this featurization technique to each data window, deriving features automatically. This approach eliminates the need for manual analysis of statistical properties, especially trend and seasonality parameters, which simplifies the data preparation process and enhances the accuracy of time series forecasting. By focusing on the data window, which serves as the prediction model (LSTM) input, we capture window-specific features that contribute to improved short-term predictions.

3.2.4 Oscillation Mitigation

Another important aspect to consider is the continuous generation of actions by the auto-scaler, which can lead to frequent changes in the number of replicas and result in oscillation issues that waste resources. For example, at time t, the auto-scaler may add a recently released resource from time t-1, or vice versa. Therefore, it is essential to consider oscillation mitigation as an important functionality in the auto-scaling process. By reducing the number of unnecessary changes in the number of replicas, system performance can be improved, and resource wastage can be reduced.

Unfortunately, oscillation mitigation has not received sufficient attention in the literature. To overcome this limitation, Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021) have integrated the oscillation mitigation technique into the autonomous and self-adaptive MAPE-K loop system (Arcaini, Riccobene & Scandurra, 2015). Therefore, we compared the results in our study with those in Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021), as they had good results on the data analysis and implementation of an auto-scaling system, as well as they considered the oscillation mitigation issue.

Our grid-based oscillation mitigation approach benefits from the generated features. The feature values form a value grid, where each line (i.e., value) represents a reference action. This grid enables matching the actions generated by our auto-scaling system to the reference actions to reduce the oscillation issue. We have called this original method of handling oscillation 'Grid-based oscillation mitigation'. Our approach has a further advantage, as the grid values change dynamically according to the historical data window used. The literature approaches often use the cooldown timer (CDT) principle (Imdoukh *et al.*, 2019), which delays the execution of a scaling-down request due to decreased workload volume. However, this CDT solution requires finding an optimized delay timer value. In contrast, our oscillation processing approach is less demanding since it is a parameterless mechanism. In addition, it can improve the oscillation mitigation compared to the related work approaches. Moreover, combining our grid-based approach with the CDT mechanism significantly improves the oscillation mitigation.

Finally, it is worth noting that our approach proposed in this study can be used in other scientific fields where there is a need to make more accurate time-series forecasting, such as transportation domain (Nguyen *et al.*, 2018), where there is a need for traffic flow prediction.

3.3 Overall architecture

Our approach follows MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge) (Computing *et al.*, 2006) loop to ensure the auto-scaling. In our context, MAPE-K presents the general process of the auto-scaling as presented in Figure 3.1. First, the auto-scaler monitors the system by logging all the measurement data (e.g., number of HTTP requests and CPU) captured during the monitoring. This historical data represents the time-series data, which will be used for model training and forecasting purposes. Second, the auto-scaler analyses the monitoring data to evaluate the system state. It forecasts the future workload, while the planning phase generates the plan that contains scaling actions, especially the increasing/decreasing of service replicas. The generated plan takes into consideration the oscillation issue. The last step of the MAPE-K module will execute the generated plan. In our case, the prediction model is part of the knowledge central to the MAPE-K loop, as shown in Figure 3.1.

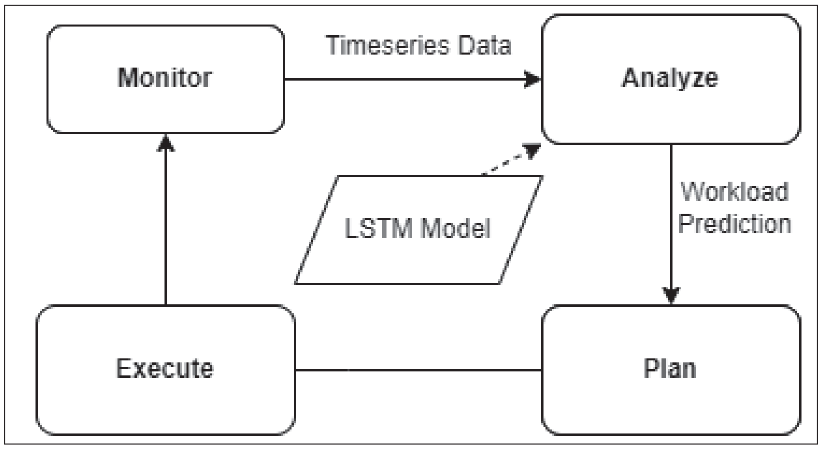


Figure 3.1 General Auto-scaling Process

3.4 Data collection and pre-processing

The data collection is based on the monitoring phase. In our case, we use univariate time series data. We organized the dataset to have only two useful pieces of information, the time (period) and the count of HTTP requests. So, the overall dataset is aggregated and transformed such that each record represents the total workload (i.e., number of HTTP requests) per minute.

Time-series data can be collected and stored by the Prometheus tool, which aggregates metrics from monitoring tools such as CAdvisor and Node Exporter.

We use the Worldcup98 Dataset (Arlitt & Jin, 2000) and NASA dataset (Dang-Quang & Yoo, 2021) to evaluate and compare our forecasting approach to literature models. The Worldcup98 Dataset contains the HTTP request logs for approximately 1.3 billion total requests made to the FIFA World Cup Website between April 30 and July 26, 1998. In contrast, NASA'95 Dataset contains a two-month log of all HTTP requests made to the Florida NASA Kennedy Space Center web server. These datasets were used extensively to evaluate auto-scalers in the cloud computing literature (Imdoukh *et al.*, 2019). We present further the used datasets in the evaluation section (Section 3.8).

3.4.1 Data scaling

The scaling of data can increase the performance of some ML algorithms, such as LSTM, in our case. Scaling is changing the values of numeric variables to have relevant properties. As a result, data are transformed to be bounded within a newly defined range, such as [0, 1], using the min-max scaling mechanism as presented in Equation 3.1.

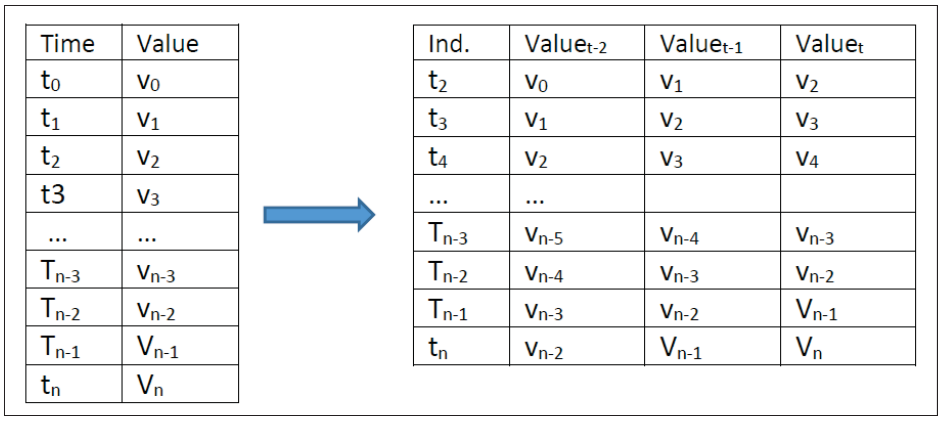
$$x^{'} = \frac{x - min(x)}{max(x) - min(x)} \tag{3.1}$$

Where $x^{'}$, x, min(x), and max(x) represent the scaled value, original value, minimum value of the feature in the dataset, and maximum value, respectively.

3.4.2 Data reframing and Horizon of prediction

The data reframing step aims to transform the data into a format suitable for a supervised learning task. In this process, we use the previous time steps as input variables, while the next time step is used as the output variable. As shown in Figure 3.2, the data are represented as a list of sequences of time series data. It corresponds to a sliding window concept (e.g., a window of two values in Figure 3.2) that is used to predict the value at the next moment (i.e., step).

The prepared data in the previous step is sufficient for the single-step prediction. For the case of multi-step prediction (e.g., next fifth-time step), we put the corresponding output value in the training process. However, this approach is not practical since it may require many models that match the size of the forecasting horizon. For example, five models are needed in the case of a prediction horizon of the length of five-time steps. Another alternative is the recursive multi-step prediction (Imdoukh *et al.*, 2019). However, its limitation is the degradation of accuracy as the size of the horizon increases.



Time	Value
t_0	v_0
t_1	v_1
t_2	v_2
t3	v_3
…	…
T_{n-3}	v_{n-3}
T_{n-2}	v_{n-2}
T_{n-1}	V_{n-1}
t_n	V_n

Ind.	$Value_{t-2}$	$Value_{t-1}$	$Value_t$
t_2	v_0	v_1	v_2
t_3	v_1	v_2	v_3
t_4	v_2	v_3	v_4
…	…		
T_{n-3}	v_{n-5}	v_{n-4}	v_{n-3}
T_{n-2}	v_{n-4}	v_{n-3}	v_{n-2}
T_{n-1}	v_{n-3}	v_{n-2}	V_{n-1}
t_n	v_{n-2}	V_{n-1}	V_n

Figure 3.2 Time Series reframing, with window size =2

3.5 Forecasting based on data featurization

The analysis phase of the MAPE-K loop (in Figure 3.1) is based on forecasting the future workload. Predicting the future workload is essential in making the auto-scaling process proactive. This prediction is based on quantitative forecasting using the collected and pre-processed data of the monitoring step. The collected data represents the historical workload data. To improve the forecasting algorithm (LSTM in our case) accuracy, we propose to add a featurization phase. The time-series featurization allows transferring the model from univariate (e.g., contains only the workload data) to multivariate by extracting new information from data, as we will present in the following subsection.

3.5.1 Our Time Series Featurization

Our approach adds a step of time series featurization (i.e., feature extraction) to improve prediction accuracy. The added features are derived from the input data. In particular, the features summarize the time series window data.

We formulate the data featurization as follows. First, we define the workload data of an instant t as d_t. Accordingly, a time series data TS can express as:

$TS =< d0, d1, \cdots, dn >$. The featurization can modeled as function:

$$\begin{aligned} &F : D^s \rightarrow D^k \\ &Wi =< d_i, d_{i+1}, \cdots d_{i+s} > \rightarrow < f_{i1}, f_{i2}, \cdots, f_{ik} > \end{aligned} \tag{3.2}$$

Where, D represents the domain of data value (e.g., integer or real), W_i contains s values, and the function generates k features.

In our approach, the features are inspired by the Japanese Candlestick concept (Tam, 2015) used in the economic domain for trading different assets such as Stocks and Futures. Four useful pieces of information are provided by a candlestick, namely, open, low, high, and close, as presented in Figure 3.3.

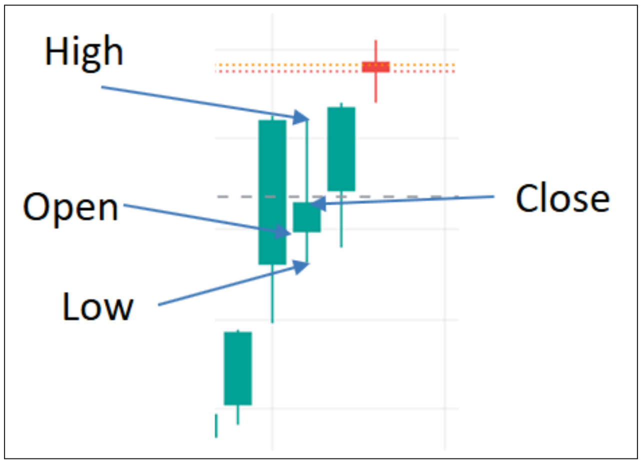


Figure 3.3 Japanese candlestick information

Furthermore, the Japanese candlestick gets an abstraction, allowing the trader to comprehend the stock evolution better. Figure 3.4 presents the line representation corresponding to the sequence of Tesla stock price values (i.e., time series data). In contrast, Figure 3.5 presents the equivalent representation based on the Japanese candlesticks, where each candlestick summarizes one day period.

Inspiring from Japanese Candlestick provides interesting advantages to our featurization approach that provides additional information about the data windows. The featurization incorporates

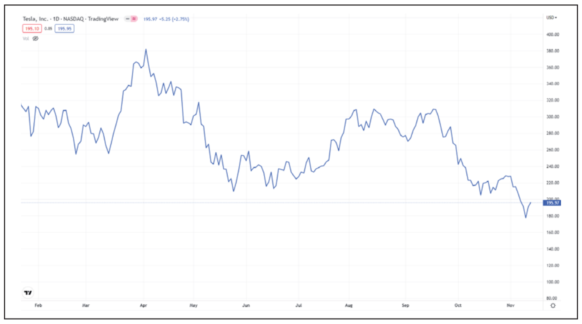


Figure 3.4 Line representation of Tesla stock

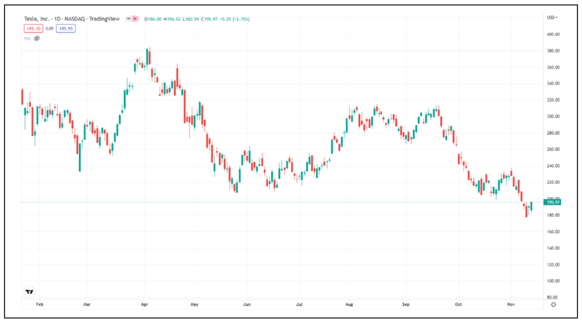


Figure 3.5 Representation of Tesla stock using Japanese candlesticks

time-based features, namely open (i.e., the first value of a window) and close (i.e., the last value of a window), as well as value-based features, namely, maximum and minimum. The extracted features abstract and generally describe the data presented in the window, which allows capturing of short-term dependencies.

In addition, our featurization approach allows the transfer of univariate data to multivariate data. Our approach avoids generating linearly dependent features, which potentially cause instability or overfitting in the models. The non-linear functions (e.g., max and min) applied to the data have introduced variations in the derived features, leading to a diversity of information, which is beneficial for the forecasting model as it captures different aspects of the underlying patterns.

Besides adopting the Japanese Candlestick concept, our approach has the advantage of automatic feature generation, which makes it less requiring compared to most related work approaches that need to collect the features on the data preparation phase, as in Wang *et al.* (2021) (e.g., CPU, Memory, and network), or even make manual analysis for feature generation such as the statistical properties as in Hyndman *et al.* (2015) (e.g., trend and seasonality). The following subsection describes our featurization process.

Algorithm 3.1 Feature Extraction Algorithm

```
**Input:**
$TS =< d_0, d_1, ..., d_n >$
$s$ : window size
**Output:** $reformulatedData$

$reformulatedData \leftarrow \{\}$;
**for** $i \leftarrow 0$ *to* $n - s$ **do**
    $w \leftarrow< d_i, ..., d_{i+s} >$;
    $open \leftarrow d_i$;
    $close \leftarrow d_{i+s}$;
    $low \leftarrow min(w)$;
    $high \leftarrow max(w)$;
    $w \leftarrow$ reformulateData(w, [open, close, low, high, average]);
    reformulatedData.append(w);
**end for**
```

3.5.2 Time Series Featurization Algorithm

To map the candlestick concept to our context, we consider that the window is the equivalent to the period (e.g., 3 m, 15 m and 1 day).

Given the window: $w = \{d_{t-n}, d_{t-n-1}, ..., d_t\}$, so the candlestick futures are obtained by the equations 3.3, 3.4, 3.5, and 3.6.

$$Open = d_{t-n} \tag{3.3}$$

$$Close = d_t \tag{3.4}$$

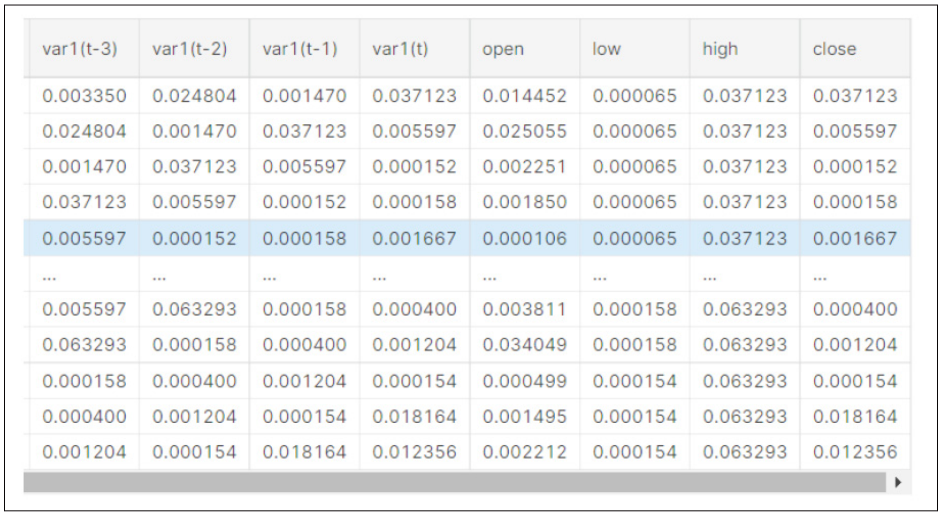
$$Low = \min(w) \tag{3.5}$$

$$High = \max(w) \tag{3.6}$$

Figure 3.6 presents an extract of time-series data organized in periods with its features.

Other information, as an indicator, can be added as the average (Equation 3.7):

$$Average = \frac{1}{n}\sum_{i=0}^{n-1} d_{t-i} \tag{3.7}$$



var1(t-3)	var1(t-2)	var1(t-1)	var1(t)	open	low	high	close
0.003350	0.024804	0.001470	0.037123	0.014452	0.000065	0.037123	0.037123
0.024804	0.001470	0.037123	0.005597	0.025055	0.000065	0.037123	0.005597
0.001470	0.037123	0.005597	0.000152	0.002251	0.000065	0.037123	0.000152
0.037123	0.005597	0.000152	0.000158	0.001850	0.000065	0.037123	0.000158
0.005597	0.000152	0.000158	0.001667	0.000106	0.000065	0.037123	0.001667
...	...	...	...	...	...	...	...
0.005597	0.063293	0.000158	0.000400	0.003811	0.000158	0.063293	0.000400
0.063293	0.000158	0.000400	0.001204	0.034049	0.000158	0.063293	0.001204
0.000158	0.000400	0.001204	0.000154	0.000499	0.000154	0.063293	0.000154
0.000400	0.001204	0.000154	0.018164	0.001495	0.000154	0.063293	0.018164
0.001204	0.000154	0.018164	0.012356	0.002212	0.000154	0.063293	0.012356

Figure 3.6 Overview of the proposed multivariate data structure.

Our time series featurization process is presented in Algorithm 3.1. The data sequence is passed in parameters. The window size is used to reformulate the data in Line 3. Lines 4-7 extract the features, which are concatenated to the initial window in Line 8.

The time complexity of the feature extraction part of the algorithm is linear (i.e., $O(n)$) to the input size, which corresponds to $|features| \times w_z$, where $|features|$ is the number of features to extract and w_z represents the window size.

3.6 LSTM Model

LSTM is a deep learning algorithm, an advanced type of recurrent neural network (RNN). Recurrent neural networks are distinguished by an ability to memorize from prior inputs to influence the current input and output. Due to the vanishing gradient problem (Hu, Huber, Anumula & Liu, 2018), RNNs forget what they have seen in previous layers and do not learn appropriately with long-term dependency cases. LSTM overcomes this drawback of RNN by using gate mechanisms that control the information flow. Thus, LSTM is highly adapted to predicting the following sequence in time-series data, such as workload over time. An LSTM network is formed by linking many LSTM units (cells).

An LSTM neuron comprises an internal memory (cell) controlled by the three gates of input, forget, and output. The role of any gate is to regulate the volume of data that passes through it. The input gate decides whether the input should change the cell's content. Its output C'_t is obtained by equations 3.9 and 3.10. The forget gate determines whether to reset the cell content to 0; its output corresponds to f_t calculated by Equation 3.8. These outputs of the two gateways present the base to calculate the new cell state C_t calculated by Equation 3.11. Finally, the output gate determines if the cell content (i.e., C_t) should impact the output of the cell h_t as presented in equations 3.12 and 3.13.

$$f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f) \tag{3.8}$$

$$i_t = \sigma(W_i.[h_{t-1}, x_t] + b_c) \tag{3.9}$$

$$C'_t = tanh(W_i.[h_{t-1}, x_t] + b_c) \tag{3.10}$$

$$C_t = f_t \times C_{t-1} + i_t \times C'_t \tag{3.11}$$

$$o_t = \sigma(W_o.[h_{t-1}, x_t] + b_o) \tag{3.12}$$

$$h_t = o_t \times tanh(C_t) \tag{3.13}$$

Where W and b represent the weights and bias, respectively. In addition, σ (sigmoid) and $tanh$ denote the used activation functions.

Table 3.1 summarizes the configuration of our LSTM model. We have chosen a Sequential LSTM architecture (i.e., a pipeline architecture). The input layer corresponds to the sliding window size (30 in our case), whereas the output layer contains only one neural cell. We configured the input and the hidden layers with a 20% dropout after each layer. Dropout is a regularization technique used to prevent data overfitting. We used backpropagation and the Adam optimizer to fit the model. Our epochs are set to 20, and our batch size is set to 10.

Table 3.1 Hyperparameters of our LSTM learning algorithm

Parameter	*Value*
Number of layers	2 layers
Input size	17
Output size	1
Units	30
Loss function	MAE
Optimizer	adam
Batch size	512
Epochs	50

3.7 Oscillation mitigation

The sudden and constant change in the number of replicas the auto-scaler generates can be costly, impacting the system's efficiency. This frequent change generates the oscillation phenomenon. To address this issue, auto-scaling related works, as in (Dang-Quang & Yoo, 2021; Imdoukh *et al.*, 2019), limit the oscillation by using the cooldown timer (CDT) and the rate of change mechanisms by using scaling down ratio (SDR). Therefore, the performance of these oscillation mitigation approaches depends on tuning the values of these parameters (i.e., CDT and SDR).

In contrast, our oscillation mitigation approach has the advantage of being parameterless due to using features extracted for the forecasting phase. The extracted features generate a selection grid to reduce the action volatility. This grid consists of elements representing the features extracted from the values of the time series window (e.g., 15 previous workload values). Figure 3.7 shows an example of a grid composed of four features: open, low, high, and close. Our approach selects the top line closest to the calculated action value.

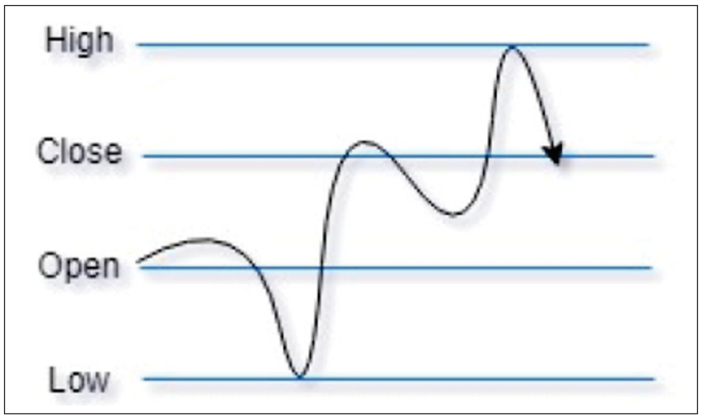


Figure 3.7 A grid based on four features with the workload curve

The grid lines defined by the features are used to cap the value of the number of replicas calculated by the planner. During a requested resource reduction (i.e., down-scaling), the least great line of the value issued by the planner is selected.

Algorithm 3.2 gives an overview of these steps. At each execution, the algorithm estimates (Line 2) the future workload using a prediction model (e.g., LSTM), which needs a data window as a parameter. A downscale is needed if the obtained number is lower than the current workload. In this case, to reduce the oscillation effect, our approach changes the workload value by a value belonging to the feature set (obtained as input by Line 3.2). The feature set values, which represent the grid lines, are generated by Algorithm 3.1. The function ’getNextProof’ (lines 7-10) returns a value from the grid that is the least value greater than the predicted workload value.

Afterward, the algorithm calculates (Line 6) the corresponding number of replicas needed by dividing the predicted workload value by the replica capacity (i.e., replicaCapacity), which represents the capacity in terms of workload. In other words, the replicaCapacity represents the number of requests a replica can handle in a given period (e.g., one minute).

Algorithm 3.2 Grid-based Oscillation Mitigation

```
**Input:**
$w =< d_{t-w_z}, ..., d_{t-1}, d_t >$
$propertySet = \{low(w), open(w), close(w), high(w)\}$
**Output:** $newReplicas$

$currentWorkload \leftarrow d_t$;
predictedNextWorkload ← predict(ForecastingModel, w);
if (predictedNextWorkload < currentWorkload) then
    predictedNextWorkload ← getNextRoof(predictedNextWorkload,propertySet);
end if
newReplicas ← predictedNextWorkload / replicaCapacity;
Function getNextRoof($elt$, $\{el_1, el_2, ...el_n\}$):
    $subSet \leftarrow$ inferior($elt$, $\{el_1, el_2, ...el_n\}$);
    roof ← min(subSet);
    return roof;
```

3.8 Experiments and Results

In this section, the main objective of the evaluation is to present the feasibility and utility of our two contributions, namely time series featurization, which improves the accuracy of the forecasting model, and our oscillation mitigation approach, which improves the auto-scaling efficiency. To achieve this objective, each of the following subsections presents an important aspect of the evaluation.

3.8.1 Simulator

To examine our approach, we developed a simulation program in Python (using the NumPy, Pandas, Scikit-learn, and Keras libraries) to test and compare our results with related work (Imdoukh *et al.*, 2019; Dang-Quang & Yoo, 2021). This simulator allows us to scale replicas according to the prediction data generated by the machine learning model presented in Section 3.6. As shown in Figure 3.8, the simulation tool consists of 5 phases, following the MAPE-K loop, which includes monitoring the auto-scaling and evaluating the results to verify the impact of models on the scaling process.

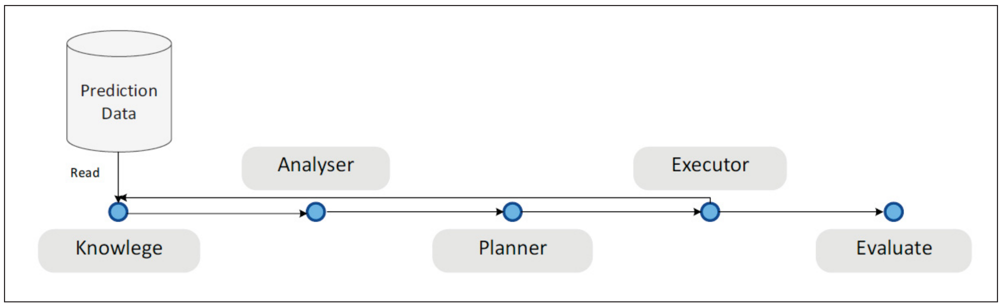


Figure 3.8 Architecture of the simulator

3.8.2 Performance metrics

Two types of metrics are considered: forecasting model metrics and auto-scaling metrics.

3.8.2.1 Evaluation metrics for the prediction model

The regression models selected in this study, LSTM and Bi-LSTM, allow the prediction of future sequences of a dataset according to the parameters acquired during its training. The efficiency of these forecasting models is evaluated according to their generalization error rate (i.e., accuracy). The accuracy evaluation in regression analysis consists of comparing the original target with the predicted one. For that measures, we can use different metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-Squared (R2), which help to explain the errors and predictive ability of the model (Chicco, Warrens & Jurman, 2021). These measures are defined as follows:

- MAE (Mean absolute error) represents the difference between the actual and predicted values calculated by averaging the absolute errors over the dataset (Eq. 3.14).
- MSE (Mean Squared Error) represents the average squared difference between the actual and predicted values (Eq. 3.15).
- RMSE (Root Mean Squared Error) represents the standard deviation of the prediction by the square root of MSE (Eq. 3.16).
- Coefficient of determination (R-Squared) represents the proportion of the variance for a dependent variable that is predictable from one or more independent variables in a regression

model. The proportion value, which normally ranges from 0 to 1, explains the correlation between variables that correspond to actual and predicted values in the forecasting case. Thus, the higher the value, the better the model (Eq. 3.17).

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{3.14}$$

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{3.15}$$

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2} \tag{3.16}$$

$$R^2 = 1 - \frac{\sum i\left(y_i - \hat{y}_i\right)^2}{\sum i\left(y_i - \bar{y}_i\right)^2} \tag{3.17}$$

Where y, $\hat{y}_i$ and $\bar{y}_i$ represent actual value, predicted value, and mean of actual values y, respectively.

As shown by equations, MSE and MAE are error metrics that quantify the difference between predicted and actual values. A lower value for these metrics indicates better performance, meaning the model's predictions are closer to the actual values. Root Mean Squared Error (RMSE), as it is the square root of MSE, penalizes further larger errors, giving a better representation of the overall prediction performance. Thus, a model having a lower MSE value implies that it has a lower value of RMSE. Additionally, we included the R-squared (R2) metric to assess the proportion of variance in the predicted values, obtained by forecasting models, compared to actual values. It represents an informative metric of the model goodness-of-fit, where a higher value indicates a better fit.

3.8.2.2 Auto-scaler evaluation metrics

The performance evaluation of our auto-scaling simulator is based on two essential points: elasticity and provisioning rate. For this purpose, we have chosen to use the metrics proposed in Herbst et al. (Herbst *et al.*, 2016) and Bauer et al. (Bauer, Grohmann, Herbst & Kounev, 2018a). These metrics are used in several literature studies on auto-scaling, such as Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021); Bauer, Herbst, Spinner, Ali-Eldin & Kounev (2018b):

- Under-provisioning metric (θ_u) indicates the number of missing replicas (e.g., containers) needed to reach the requested number of replicas in a time interval (Eq. 3.18).
- Over-provisioning metric (θ_o) represents the supplied replicas that exceed the demanded number, as shown in Eq. 3.19.
- Under-provisioning time (T_u) reflects the time during which the simulator was under-provisioning (Eq. 3.20)
- Over-provisioning time (T_o) reflects the time during which the simulator was over-provisioning (Eq. 3.21)
- Elasticity speedup (ϵ_n) reveals the performance gain obtained by using a proactive auto-scaler. In this work, the elasticity speedup is calculated by a ratio between two cases: using a proactive auto-scaler and a reactive auto-scaler, which are represented in Eq. 3.22, by the p and r indices, respectively. In contrast to the previous metrics, the higher the ϵ_n value, the higher the auto-scaler performance. In other words, the best auto-scaler has less $\theta_u, \theta_o, T_u, T_o$ values and essentially a higher ϵ_n value.

$$\theta_u = \frac{100}{T} \sum_{i=1}^{T} \frac{max(required(t) - provided(t), 0)}{required(t)} \Delta t \tag{3.18}$$

$$\theta_o = \frac{100}{T} \sum_{i=1}^{T} \frac{max(provided(t) - required(t), 0)}{required(t)} \Delta t \tag{3.19}$$

$$T_u = \frac{100}{T} \sum_{i=1}^{T} \max(sgn(required(t) - provided(t)), 0)\Delta t \tag{3.20}$$

$$T_o = \frac{100}{T} \sum_{i=1}^{T} \max(sgn(provided(t) - required(t)), 0)\Delta t \tag{3.21}$$

$$\epsilon_n = \left(\frac{\theta_{u,r}}{\theta_{u,p}} . \frac{\theta_{o,r}}{\theta_{o,p}} . \frac{T_{u,r}}{T_{u,p}} . \frac{T_{o,r}}{T_{o,p}} \right)^{\frac{1}{4}} \tag{3.22}$$

Where, $required(t)$ represents the correct number of replicas corresponding to the actual workload. In contrast, $provided(t)$ represents the number of replicas offered by the auto-scaler according to the predicted value of the workload. Δt corresponds to the time interval (e.g., each 1 minute) to check the change in the workload. T represents the entire evaluation period and $sgn()$ is the sign function.

3.8.3 Datasets

3.8.3.1 Worldcup 98 Dataset

The Worldcup 98 Dataset contains the HTTP request logs of more than 1.3 billion requests from the 1998 FIFA World Cup website in France between April 30 and July 26 (Arlitt & Jin, 2000). This high request load is due to the number of spectators worldwide who followed this event. The Worldcup'98 Dataset is often used by researchers working on auto-scaling in the cloud. Moreover, this will allow us to compare our results with other studies such as Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021).

The data structure of this dataset contains the following properties: timestamp, clientID, objectID, size, method, status, type, and server Arlitt & Jin (2000). In order to better manage this data, we performed a preprocessing by grouping all logs occurring in the same minute into a single cumulative record, as in Imdoukh *et al.* (2019). As a result, the information about the number of requests corresponds to the number of received requests in one minute. Figure 3.9 plots the obtained dataset.

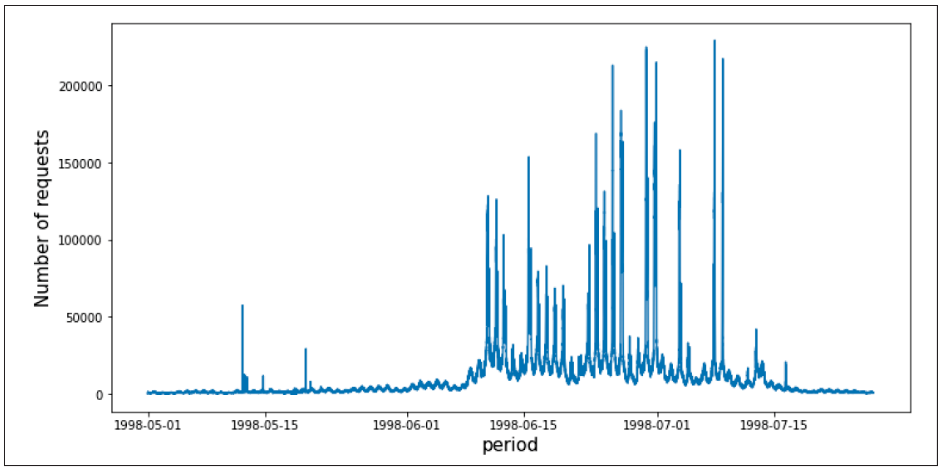


Figure 3.9 Representation of Worldcup 98' dataset by number of requests per minute

3.8.3.2 NASA 95' Dataset

The NASA 95' Dataset provides a two-month log of HTTP requests to the NASA Kennedy Space Center web server in Florida. This dataset contains 3,461,612 requests collected between July 1, 1995, at 00:00:00 and August 31, 1995, at 23:59:59. The timestamps are accurate to one second. It should note that no accesses were reported from 01/Aug/1995:14:52:01 to 03/Aug/1995:04:36:13 since the web server was shutdown due to Hurricane Erin (Dang-Quang & Yoo, 2021). Figure 3.10 plots a part of the dataset.

3.8.3.3 Data Distribution

The distribution of the NASA'95 dataset is less complicated than that of Worldcup 98' (Dang-Quang & Yoo, 2021). Figure 3.11 presents the data distribution for two datasets: Worldcup and NASA. The Worldcup dataset exhibits a higher degree of variation and the presence of outliers compared to the NASA dataset. This is evident from the wider spread of data points in the Worldcup dataset. Notably, the high difference between the median value, 3884, and the

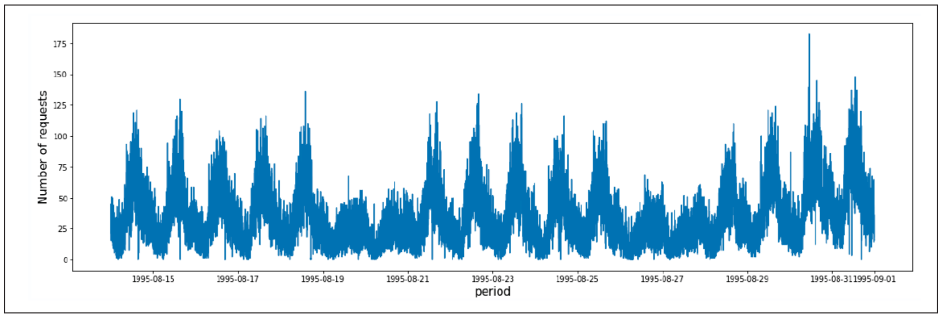


Figure 3.10 Representation of NASA 95' dataset by number of requests per minute

maximum value that exceeds 200,000 indicates the presence of extreme values in the Worldcup dataset. On the other hand, the NASA dataset shows less variation and a smaller range of values, with the maximum value being around 300. The median value of the NASA dataset is 29, indicating a relatively concentrated distribution of data points.

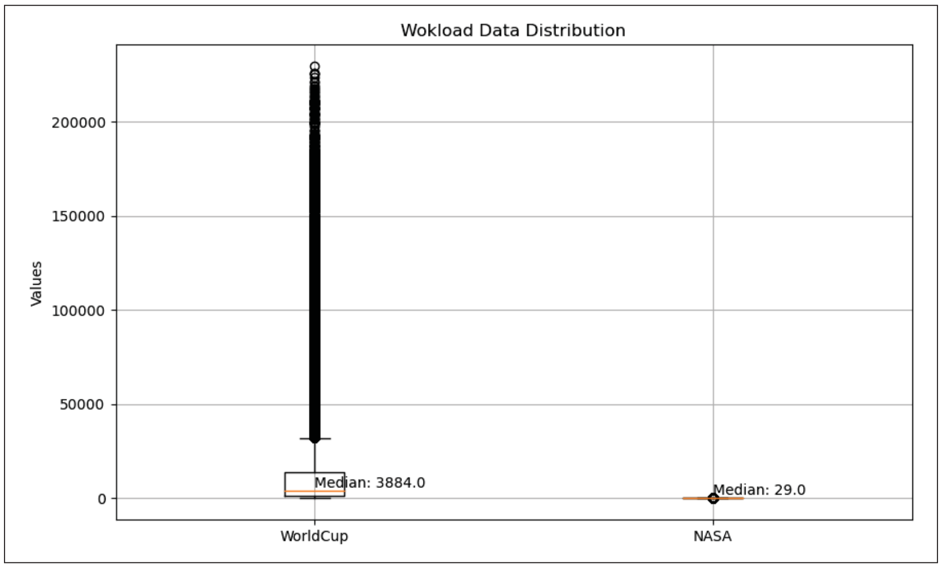


Figure 3.11 Data Distribution Comparison of Worldcup and NASA Datasets

Indeed, the Worldcup 98' dataset has a high degree of variation in its value, with peaks that are difficult to predict, as well as a less well-defined pattern compared to the NASA 95' dataset, in terms of trend and seasonality.

3.8.4 Evaluation Protocol

First, we have rebuilt the forecasting models proposed in the literature to reproduce these tests in the same operating environment and execution conditions. This will allow us to have reproducible and non-biased results. Consequently, we have kept the same configuration hyperparameters proposed in each approach.

Second, the efficiency evaluation of each forecasting algorithm will be based on its performance results. However, the configuration parameters proposed for these algorithms (Imdoukh et al. algorithm in (Imdoukh *et al.*, 2019), Dang et al. algorithm in (Dang-Quang & Yoo, 2021) and our algorithm in Table 3.1) are not sufficient to cover all our test cases, whether during the prediction stage or the auto-scaling of the system.

Therefore, we need to test several combinations of the key hyper-parameters of these models to verify their impact on the different approaches' contributions by analyzing the change in the forecasting results. In addition, this study, according to multiple combinations of settings and parameters, dedicates the best model with the optimal configuration.

Third, every dataset we have chosen (i.e., Worldcup'98 and NASA'95) will be divided into two parts: train the model and do the tests. For this purpose, we were inspired by the "Pareto (80-20)" principle (Dunford, Su & Tamang, 2014) to make our division. Thus, we chose to keep 80% of the data available for training the algorithms against 20% for testing for each dataset. The execution results of the test data were then analyzed, following the performance metrics, to evaluate the success rate of these forecasts.

Finally, we experimented with our predicted data as well as those of Imdoukh et al. (Imdoukh *et al.*, 2019) and Dang et al. (Dang-Quang & Yoo, 2021) in our simulator, to choose the best auto-scaling approach, which deals most effectively with the oscillation mitigation. These approaches are evaluated according to the metrics presented in Subsection 3.8.2.

3.8.5 Evaluation of our forecasting approach

We first need to reproduce forecasting approaches proposed in the forecasting literature and test them in the same operating conditions. Specifically, we consider for comparison the two approaches (Imdoukh *et al.*, 2019; Dang-Quang & Yoo, 2021) since they both use the deep learning technique and treat the oscillation issue. Imdoukh et al. (Imdoukh *et al.*, 2019) use LSTM model, whereas Dang et al. (Dang-Quang & Yoo, 2021) use Bi-LSTM model.

Next, we aim to ensure that our featurization-based approach can produce accurate forecasts by comparing it to these related works. Then, we will further evaluate the featurization impact on the forecasting results.

3.8.5.1 Comparison to related work

For the evaluation of the regression model, we used the metrics presented in Subsection 3.8.2, namely RMSE, MSE, MAE, and R^2. Table 3.2 records the evaluation metrics compared to related work.

Table 3.2 Our featurization-based forecasting approach vs. related work

Dataset	*Approach*	*RMSE*	R^2	*MSE*	*MAE*
NASA	LSTM (Imdoukh *et al.*, 2019)	0.0811	0.6430	0.0066	0.0607
	Bi-LSTM (Dang-Quang & Yoo, 2021)	0.0813	0.6409	0.0066	0.0609
	Our Model	**0.0016**	**0.9993**	**2.6343e-06**	**0.0015**
Worldcup'98	LSTM (Imdoukh *et al.*, 2019)	0.0023	0.9862	5.1755e-06	0.0016
	Bi-LSTM (Dang-Quang & Yoo, 2021)	0.0020	0.9898	3.8043e-06	0.0013
	Our Model	**0.0006**	**0.9990**	**3.5793e-07**	**0.0004**

During these tests, we noticed that the results of the two related work approaches were consistently close, with occasional variations in performance where one approach outperformed the other. Nevertheless, our approach with the featurization always performed well in these tests and outperformed other approaches.

In the case of the NASA dataset, the improvement of our approach is more impressive. As demonstrated in Subsubsection 3.8.3.3 of the data description, the NASA dataset has a more stable pattern, which can positively impact the forecasting error. In the case of the Worldcup'98, there are a lot of workload peaks, which increases the number of outliers in the difference between the predicted and actual data. These significant and frequent differences hide in part the improvement in the overall evaluation metrics, such as RMSE and MAE.

Our model exhibited remarkable performance for the NASA dataset, surpassing the best values reported in the related works. Notably, our model demonstrated approximately improvements of 98%, 55%, 99%, and 97% in RMSE, R2, MSE, and MAE, respectively.

This reduction in the error rate has, therefore, impacted the quality and accuracy of the forecasting as presented in figures 3.12 and 3.13, which visualize the prediction of our model on Worldcup'98 and NASA dataset respectively. The red color represents the actual dataset, whereas the prediction of our model is plotted using the blue color. It is clear that our model fits well with the test dataset.

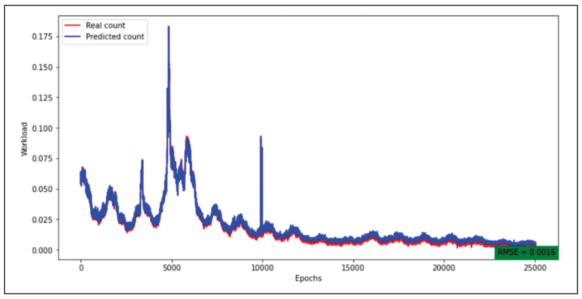


Figure 3.12 Our model prediction with WorldCup'98 dataset

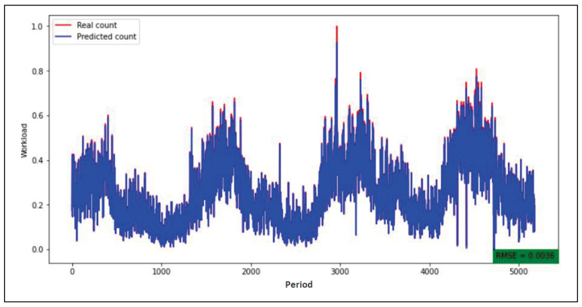


Figure 3.13 Our model prediction with NASA dataset

3.8.5.2 Result of the hyperparameter combination tests

To further evaluate the effect of the featurization mechanism, we decided to change the LSTM hyperparameters of the different approaches to see if that impacts our findings. We have chosen the following hyperparameters: the number of units and the number of epochs. The number of units corresponds to the dimension of the inner cells. Whereas the number of epochs defines the number of times, the learning algorithm will change the network's weights.

Table 3.3 shows that our approach, regardless of the changed hyperparameters, outperforms related work thanks to our featurization mechanism. As in the previous test category, the improvement with the NASA dataset is more apparent and significant.

Table 3.3 Forecasting results with varying the hyper-parameters: the numbers of units and epochs. Our Model is compared to LSTM Model of Imdoukh *et al.* (2019) and Bi-LSTM Model of Dang-Quang & Yoo (2021)

Case	*Dataset*	*Approach*	*RMSE*	R^2	*MSE*	*MAE*
30 units with 50 Epochs	NASA	LSTM	0.0811	0.6430	0.0066	0.0607
		Bi-LSTM	0.0813	0.6409	0.0066	0.0609
		Our Model	**0.0024**	**0.9997**	**6.1537-06**	**0.0017**
	Worldcup'98	LSTM	0.0023	0.9861	5.1755e-06	0.0016
		Bi-LSTM	0.0020	0.9898	3.8042e-06	0.0013
		Our Model	**0.0006**	**0.9990**	**3.5793e-07**	**0.0004**
20 units with 120 Epochs	NASA	LSTM	0.0807	0.6468	0.0065	0.0606
		Bi-LSTM	0.0817	0.6381	0.0067	0.0610
		Our Model	**0.0044**	**0.9989**	**1.9779e-05**	**0.0027**
	Worldcup'98	LSTM	0.0038	0.9609	1.4625e-05	0.0035
		Bi-LSTM	0.0021	0.9884	4.3315e-06	0.0017
		Our Model	**0.0017**	**0.9916**	**3.1499e-06**	**0.0015**
10 units with 50 Epochs	NASA	LSTM	0.0816	0.6389	0.0067	0.0614
		Bi-LSTM	0.0823	0.6326	0.0068	0.0614
		Our Model	**0.0073**	**0.9971**	**5.2620e-05**	**0.0051**
	Worldcup'98	LSTM	0.0044	0.9485	1.9246e-05	0.0040
		Bi-LSTM	0.0023	0.9858	5.3124e-06	0.0018
		Our Model	**0.0019**	**0.9907**	**3.4721e-06**	**0.0017**

3.8.5.3 Evaluation of the impact of our featurization mechanism on the related work approaches

In the previous test category, we tested the impact of our featurization mechanism by tuning two key hyperparameters: the number of units and the number of epochs. In this test category, we consider the overall hyperparameters of related work models by applying our featurization mechanism to the data before using the forecasting algorithms of the related work (Imdoukh *et al.*, 2019; Dang-Quang & Yoo, 2021).

As presented in Table 3.4, our featurization mechanism improves the forecasting accuracy of the related work, especially in the case of the NASA dataset. For example, our approach achieves an improvement of approximately 92% in $RMSE$ and 55% in R^2 for the LSTM model. Additionally, it significantly improves the performance of the Bi-LSTM model with approximately 84% improvement in $RMSE$ and 55% improvement in R^2.

Table 3.4 The impact of our featurization mechanism on the related work approaches (LSTM of Imdoukh *et al.* (2019) and Bi-LSTM of Dang-Quang & Yoo (2021))

$Dataset$	$Approach$	$RMSE$	R^2	MSE	MAE
NASA	LSTM	0.0811	0.6430	0.0066	0.0607
	LSTM ⊕ our featurization	**0.0056**	**0.9983**	**3.1902e-05**	**0.0043**
	Bi-LSTM	0.0813	0.6409	0.0066	0.0609
	Bi-LSTM ⊕ our featurization	**0.0130**	**0.9908**	**0.0002**	**0.0099**
Worldcup'98	LSTM	0.0022	0.9861	5.1755	0.0016
	LSTM ⊕ our featurization	**0.0016**	**0.9935**	**2.4442e-06**	**0.0010**
	Bi-LSTM	0.0019	0.9898	3.8043e-06	0.0013
	Bi-LSTM ⊕ our featurization	**0.0052**	**0.9289**	**2.6619e-05**	**0.0051**

3.8.5.4 Finding and Analysis Summary of Our Featurization Approach Evaluation

Our featurization approach has demonstrated significant improvements in forecasting accuracy. This conclusion is based on a comprehensive evaluation, including: comparison to related work, comparison to models with varied hyperparameters, and an assessment of the impact of our featurization on existing forecasting works.

Figure 3.14 illustrates the percentage improvement achieved by comparing our forecasting model to the best values of compared related works, whose accuracy values are presented in Table 3.2. In both datasets, our approach outperformed other models regarding accuracy metrics such as RMSE, MSE, and MAE. In the WorldCup dataset, the R2 value of our model was similar to that

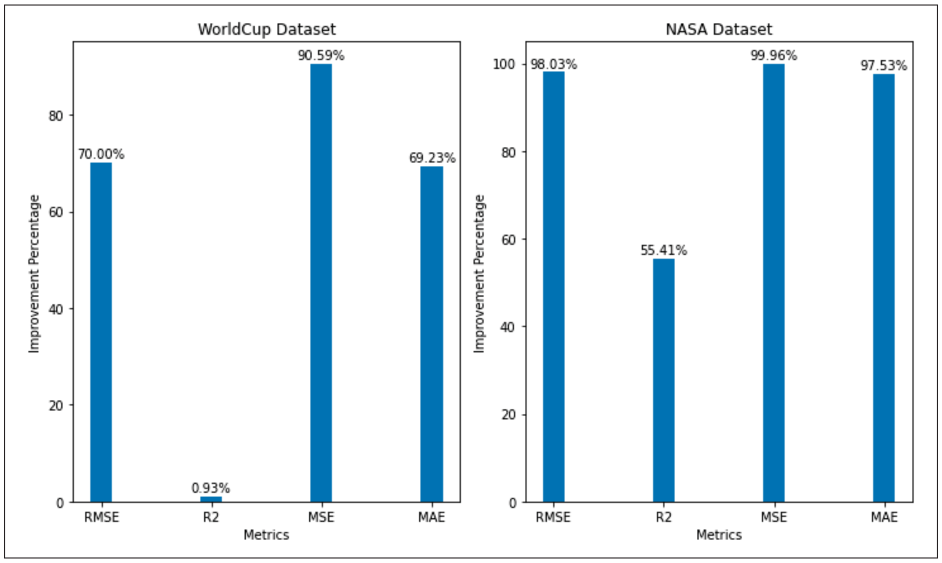


Figure 3.14 Relative Improvement Percentage of Metrics: Our Approach vs. Related Works

of other models that already have high R2 values (around 0.99). This similarity indicates that our model captures similar levels of variance in the data as the other models while still achieving superior performance in error metrics.

It is worth noting that the different metric values are calculated based on scaled data. In the case of descaling the data (i.e., inverting the scaling process), the metric values increase while maintaining the same improvement percentage.

3.8.6 Evaluation of the auto-scaler

This subsection aims mainly to evaluate our oscillation mitigation approach through the evaluation of the auto-scaler performance. Therefore, we will discuss our strategies and the results of the tests carried out on the auto-scaling simulator. We have implemented our auto-scaling simulator in two modes: the reactive and the proactive modes. Like most commercial solutions, the reactive mode does not use the prediction functionality.

The reactive approach will be used as a baseline to compare the improvement offered by the proactive approaches, as well as in the calculation of the autoscaling performance metrics, notably the ϵ_n metric (Elasticity speedup in Eq. 3.22).

The proactive mode allows us to analyze the performance of our proactive approach and compare it to related work, namely Imdoukh et al. (Imdoukh *et al.*, 2019) and Dang et al. (Dang-Quang & Yoo, 2021).

To allow comparison of our auto-scaling approach with those proposed in the literature, we need to run our auto-scaling simulator for all the approaches under the same conditions proposed in the literature. The auto-scaling simulator uses the predicted data collected during the previous tests.

In these tests, we considered five auto-scaling approaches:

- The reactive approach: scaling uses the current monitoring data without the forecasting ability. This simple category represents most industrial auto-scaling solutions.
- Our proactive approach ($OurProactive$): represents our proactive approach that uses the data generated from our featurization-based forecasting algorithm without processing the oscillation issue.
- Our Proactive & Grid approach ($OurProWithGrid$): represents our proactive approach with the use of our grid-based oscillation mitigation approach.
- Work1 : is the approach proposed in the literature (Imdoukh *et al.*, 2019) using our predicted data. It suggested a cooldown timer (CDT) of 10 seconds with a scaling down ratio (SDR) of 40%.
- Work2 : the approach proposed in the literature (Dang-Quang & Yoo, 2021) using the predicted data generated by our featurization-based approach. It chooses 60 seconds for the CDT and 60% for the SDR.

3.8.6.1 Reactive vs. our proactive approach

In this first test category, we show the improvement provided by our proactive approach based on the featurization forecasting mechanism.

Comparing two figures 3.15 and 3.16, we see that our proactive approach can shorten the response time due to its prediction of the changes in resource requirements in the system. In addition, our forecasting approach has substantially improved the metrics values of test results for both datasets (NASA and WorldCup) compared to the reactive approach, as presented in Table 3.5.

Thanks to the forecasting mechanism, our proactive approach reduces the variation between the current and the needed resources (i.e., replicas), which improves the auto-scaling performance. For instance, in the case of Worldcup dataset, our model improves the metric values of over-provisioning (θ_o), under-provisioning (θ_u), over-provisioning time (T_o), under-provisioning time (T_u) and elasticity speedup (ϵ_n) by about 55%, 76%, 55%, 33%, and 140%, respectively. Moreover, our proactive approach improves the overall auto-scaling performance (i.e., ϵ_n) by about 206% in the case of the NASA dataset. This significant improvement demonstrates the importance of the proactive behavior of our approach, which uses our featurization-based forecasting approach.

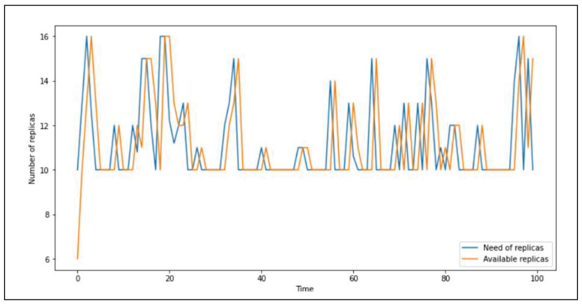


Figure 3.15 reactive Auto-scaler behavior using NASA'95 dataset

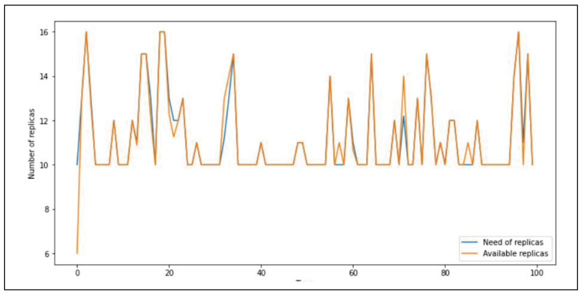


Figure 3.16 Auto-scaler behavior based on our proactive approach using NASA'95 dataset

Table 3.5 Reactive vs. our Proactive auto-scaling approaches with Worldcup 98' and NASA Datasets

Dataset	*Metric*	*Reactive*	*OurProactive*
NASA	θ_o	2.8139	**0.0493**
	θ_u	**1.1794**	1.1993
	T_o	13.3070	**9.5947**
	T_u	13.2231	**11.6587**
	ϵ_n	1	**3.0659**
Worldcup'98	θ_o	10.1007	**4.4949**
	θ_u	4.6103	**1.0628**
	T_o	20.2554	**9.0138**
	T_u	20.4295	**13.5262**
	ϵ_n	1	**2.3983**

3.8.6.2 Our approach with vs. without oscillation mitigation

To evaluate the impact of our oscillation mitigation approach on the auto-scaling behavior, we compare our proactive approach with and without the oscillation mitigation. Table 3.6 summarizes the test results. Our oscillation mitigation approach was able to improve the performance of the auto-scaler enormously. It improved the overall Elasticity Speedup metric (ϵ_n) by about 400% with the NASA'95 dataset and 108% with the WorldCup'98 dataset.

Our grid-based mechanism can increase the over-provisioning metric, θ_o, as it can choose a higher value than the predicted one in the case of downscaling needs. For example, in the case of the NASA dataset, the θ_o metric is increased by about 46%. Moreover, against expectation, our approach has reduced the time when the system is over-provisioning (T_o metric). The dynamics of our grid and the predicted workload explains this improvement. As a result, the overall performance of auto-scaling is greatly improved.

3.8.6.3 Our approach with oscillation mitigation vs. related work approaches

This test category aims to compare our oscillation mitigation mechanism to the related work (Imdoukh *et al.*, 2019; Dang-Quang & Yoo, 2021) that both proposed an oscillation mitigation

Table 3.6 Our proactive approach with vs. without oscillation mitigation with Worldcup 98' and NASA Datasets

Dataset	*Metric*	*OurProactive*	*OurProWithGrid*
NASA	θ_o	**0.0493**	0.0722
	θ_u	1.1993	**0.2543**
	T_o	9.5947	**0.2916**
	T_u	11.6587	**2.0095**
	ϵ_n	3.0659	**15.2634**
Worldcup'98	θ_o	4.4949	**0.6567**
	θ_u	1.0628	**1.0508**
	T_o	9.0138	**4.5799**
	T_u	13.5262	**9.8195**
	ϵ_n	2.3983	**4.9919**

strategy by restricting the periodicity and rate of change. Work1 (Imdoukh *et al.*, 2019) proposed a cooldown timer (CDT) of 10 seconds with a scaling down ratio (SDR) of 40 percent, while Work2 (Dang-Quang & Yoo, 2021) chooses 60 seconds for the CDT and 60 percent for the SDR. To neutralize the forecasting effect of each approach, we use the same forecasting data generated by our forecasting approach.

It is interesting to note that our oscillation mitigation approach proposes an original mechanism that has the advantage of being agnostic to any parameters to optimize (i.e., parameterless), unlike other related work mechanisms. Considering this originality, our approach slightly improved the overall auto-scaling performance compared to established and widely used mechanisms, as presented in Table 3.7.

These findings led us to consider combining these approaches for potential improvement, as discussed in the following evaluation element.

3.8.6.4 Combination of oscillation solutions

In this test category, we investigate the feasibility and utility of combining our grid-based oscillation mitigation approach with the related work commonly used mechanism, namely, the

Table 3.7 Our proactive approach with our grid-based oscillation mitigation vs. related work (Work1 of Imdoukh *et al.* (2019)) and Work2 of Dang-Quang & Yoo (2021)

$Dataset$	$Metric$	$OurProWithGrid$	$Work1$	$Work2$
NASA	θ_o	**0.0722**	0.0828	0.0732
	θ_u	0.2543	0.2565	**0.2501**
	T_o	**0.2916**	0.2954	0.2954
	T_u	2.0095	2.0202	**2.0082**
	ϵ_n	**15.2625**	14.6498	15.2237
Worldcup'98	θ_o	**0.6567**	0.9125	0.8357
	θ_u	1.0508	1.3457	**0.9761**
	T_o	**4.5799**	4.3529	4.6044
	T_u	**9.8195**	10.0019	9.8665
	ϵ_n	**4.9919**	4.3572	4.7753

cooldown time (CDT). The CDT adds a delay when a downscaling request is received to confirm the persistence of this need.

To deduce general findings on the performance of the combination of mechanisms, we performed intensive experiments taking into consideration different contextual parameters, namely, the predicted data and the CDI values as presented in tables 3.8, 3.9 and 3.10.

Table 3.8 Using our predicted data, a comparison of our grid-based approach ($OurProWithGrid$) to its combination with CDT mechanism ($OurProWithGrid \oplus CDT$)

$Case$	$Metric$	$OurProWithGrid$	$OurProWithGrid \oplus CDT$
Case1: CDT = 10s	θ_o	**0.0722**	0.4526
	θ_u	0.2543	**0.0366**
	T_o	**0.2916**	2.0167
	T_u	2.0095	**0.5605**
	ϵ_n	**15.2625**	13.2899
Case1: CDT = 60s	θ_o	**0.0722**	0.1539
	θ_u	0.2543	**0.0061**
	T_o	**0.2916**	0.4268
	T_u	2.0095	**0.0694**
	ϵ_n	15.2625	**67.6469**

Table 3.9 Using the predicted data of Imdoukh *et al.* (2019) approach, a comparison of our grid-based approach ($OurProWithGrid$) to its combination with CDT mechanism ($OurProWithGrid \oplus CDT$)

$Case$	$Metric$	$OurProWithGrid$	$OurProWithGrid \oplus CDT$
Case1: CDT = 10s	θ_o	0.0722	**0.0336**
	θ_u	**0.2543**	0.5066
	T_o	0.2916	**0.1973**
	T_u	**2.0095**	7.0110
	ϵ_n	**15.2625**	12.5497
Case1: CDT = 60s	θ_o	0.0722	**0.0119**
	θ_u	0.2543	**0.1280**
	T_o	0.2916	**0.0480**
	T_u	2.0095	**2.2090**
	ϵ_n	15.2625	**43.6019**

Table 3.10 Using the predicted data of Dang-Quang & Yoo (2021) approach, a comparison of our grid-based approach ($OurProWithGrid$) to its combination with CDT mechanism ($OurProWithGrid \oplus CDT$)

$Case$	$Metric$	$OurProWithGrid$	$OurProWithGrid \oplus CDT$
Case1: CDT = 10s	θ_o	0.0722	**0.0337**
	θ_u	**0.2543**	0.4981
	T_o	0.2916	**0.1973**
	T_u	**2.0095**	6.9737
	ϵ_n	**15.2625**	12.6064
Case1: CDT = 60s	θ_o	0.0722	**0.0119**
	θ_u	0.2543	**0.1261**
	T_o	0.2916	**0.0480**
	T_u	**2.0095**	2.1983
	ϵ_n	15.2625	**43.8206**

Two cases of CDT values are considered: CDT=10ms and CDT=60ms. In the case of CDT=10ms, the combination is less efficient. The case of a small value of CDT (e.g., 10ms) pushes to make more auto-scaling operations, which worsens the situation by increasing the oscillation issue.

In contrast, the combination with the CDT=60 case significantly improved the performance of the auto-scaling. It improves the overall performance metric (ϵ_n) by approximately 343%, 185%, and 187% considering the predicted data used in tables 3.8, 3.9 and 3.10, respectively.

In addition, this fairly high value of CDT (i.e., 60ms) can increase the over-provisioning metrics, T_o and θ_o, as the case in Table 3.8. Since the CDT mechanism adds a delay time, our approach reduces the number of resources to shrink. However, despite this possible increase in over-provisioning metrics, the combination significantly improves the under-provisioning metrics, which is reflected in the overall performance of the auto-scaling represented by Elasticity speedup (ϵ_n).

3.8.6.5 Finding and Analysis Summary of Our Oscillation Mitigation Approach Evaluation

First, our proactive approach, which builds upon our featurization approach to improve workload forecasting, delivers significantly enhanced auto-scaling performance compared to the reactive approach.

Second, the improved performance is further multiplied by combining our featurization approach with our proposed oscillation mitigation mechanism, referred to as OurProWithGrid. The overall auto-scaling performance of our approach surpasses that of the related works for both datasets. Notably, our grid-based oscillation mitigation approach offers the advantage of being parameterless, unlike common mechanisms employed in related works.

Finally, we explored the combination of our grid-based oscillation mitigation approach with established mechanisms, particularly the CDT (Cool Down Timer). The results demonstrate that the combination, when paired with an optimized CDT value, leads to multiplied auto-scaling performance compared to using only the CDT mechanism. It is important to note that this combination requires the use of the CDT parameter, which implies it is not parameterless. As a result, we plan to explore the utilization of optimization techniques to determine the optimal CDT value as part of our future work, as discussed in the conclusion section.

3.9 Conclusion

In this work, we addressed the resource management of IoT systems, specifically service auto-scaling at the edge computing level. Edge devices are equipped with relatively powerful resources (processors, memories, and graphics cards), capable even of running machine learning applications. These (mini) computers are close to IoT devices and can make decisions without cloud infrastructure. Our study aims to optimize edge devices' resource management for better service performance and efficient resource utilization. To do so, we chose to rely on existing techniques and methods used in the cloud, such as containerization and proactive auto-scaling. Our main contribution focused on two mechanisms. First, we implemented the machine learning method LSTM, which allowed us to predict upcoming resource change requests. This helped to adapt to changes in resource requirements proactively. In addition, we made an original improvement to our LSTM prediction model by adding data featurization techniques, notably based on the Japanese Candlestick concept. This technique allowed our prediction model to extract the most correlations between the input data, to have more accurate prediction results. This study compared our model to other prediction models proposed in the literature to prove its efficiency rate. Then, our second contribution focused on the oscillation mitigation of auto-scaling. Our approach introduced a selection grid concept in the action generation phase to control the autonomous and self-adaptive systems based on MAPE-K loop. This grid is built from the features of historical workload data preceding the moment of the change request. This technique allows having the number of replicas closest to our objectives during the planning phase of the feedback MAPE-K loop. Compared to related work, our approach to oscillation mitigation has the advantage of being parameterless, requiring no parameter optimization. In addition to this advantage, it could improve performance compared to related work. Combining the promising mechanisms proposed in this study can considerably improve the service auto-scaling performance. Compared to the existing studies in the literature, this efficient solution can also be implemented in different domains (e.g., network) to benefit from better autonomous and self-adaptive resource management in a proactive mode.

For our future work, while our featurization method based on Japanese candlesticks has shown promising results, we plan to investigate using the candlestick patterns in our feature grid to improve the accuracy and power of our forecasting model. Candlestick Patterns are widely used in technical analysis and can provide valuable insights into market trends and price movements. Additionally, we plan to apply optimization techniques to enhance the combination of our grid-based oscillation mitigation mechanism with the commonly used CDT mechanism, as our experimentation has indicated the potential for significant improvement through this combination.

CHAPTER 4

PROACTIVE SERVICE AUTO-SCALING IN EDGE ENVIRONMENTS: METHODOLOGY, QUANTITATIVE ANALYSIS AND IMPROVEMENTS

Ahmed Bali[1] , Khalifa Serraye[1] , Abdelouahed Gherbi[1]

[1] Department of Software Engineering and IT, École de Technologie Supérieure
1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3
Paper submitted to the journal of Future Generation Computer Systems.

ABSTRACT Edge computing has become increasingly popular in addressing the time-sensitive demands of IoT applications by offloading workload from the cloud to edge nodes. However, the limited resources of IoT edge devices pose challenges for service deployment, necessitating auto-scaling to optimize resource utilization in response to dynamic workloads. Proactive auto-scaling, which leverages workload prediction to anticipate future demands, has emerged as an interesting approach.
In this study, we assess the performance of forecasting techniques in a resource-constrained environment using a proposed methodology. We evaluate established methods such as Auto-Regressive Integrated Moving Average (ARIMA) and Long Short-Term Memory (LSTM), as well as lightweight alternatives including Support Vector Regression (SVR), Random Forest (RF), and Bayesian Network (BN). Furthermore, we investigate the effectiveness of Ensemble learning in improving forecasting accuracy and optimizing resource utilization. The experimental results demonstrate the effectiveness of our proposed Dynamic Ensemble learning technique, consistently achieving high forecasting accuracy and auto-scaling performance.

Keywords: Edge computing, Virtualization, Auto-scaling, Time series Forecasting, ARIMA, LSTM, SVR, RF, BN, Ensemble learning

4.1 Introduction

The pervasive nature of Internet of Things (IoT) applications has profoundly impacted various aspects of our daily lives. These applications heavily rely on networks of small devices embedded in our surroundings, such as environmental, healthcare, and industrial sensors. As a result, billions of devices are now interconnected (Evans, 2011), generating massive amounts of data. However, this abundance of data can negatively impact network performance, leading to increased service latency. To address these challenges, edge computing has emerged as a promising solution. By offloading the workload of IoT systems from the Cloud to edge nodes, edge computing improves system responsiveness and minimizes latency. However, the edge network is often characterized by resource-limited devices and the heterogeneity of technologies used, which limits service deployment capabilities.

To address the heterogeneity of the IoT technology stack, lightweight virtualization technologies such as containers have gained significant adoption (Ahmed *et al.*, 2019). Containers offer a convenient solution for packaging and executing IoT services along with their dependencies in self-contained and isolated modules. This approach helps mitigate compatibility issues and enhances portability across different IoT devices. Moreover, container orchestration techniques like Swarm enable resource sharing among IoT edge devices within a cluster. They facilitate the efficient utilization of resources and provide seamless communication capabilities between containers across virtual networks. In this context, service deployment involves allocating a set of replicas (e.g., containers or pods) to distribute the processing load among available devices organized into clusters. Each replica represents an instance of the service to be deployed. Increasing the number of replicas improves service response time by reducing latency, but it also increases the utilization of computational resources. Thus, resource utilization needs to be optimized while ensuring efficient service performance.

As previously mentioned, devices at the edge level are typically characterized by limited resources, which necessitates a delicate balance between meeting Quality of Service (QoS) requirements, such as response time, and efficiently utilizing computing resources. Auto-scaling plays a crucial role in dynamically adjusting the number of replicas, reducing both the over-provisioning and under-provisioning of resources.

However, reactive auto-scaling approaches commonly employed in industrial solutions like Kubernetes HPA, Google Cloud Platform, Amazon EC2, and Oracle Cloud lack proactivity and struggle to adapt to frequent and dynamic workload changes. In contrast, proactive auto-scalers anticipate future needs (Lorido-Botran *et al.*, 2014) and dynamically adjust the system, leading to improved responsiveness and performance. While widely used time series data analysis and forecasting techniques such as Autoregressive Integrated Moving Average (ARIMA) and Long Short-Term Memory (LSTM) have been successfully applied for workload forecasting in auto-scaling studies like Sangpetch *et al.* (2017); Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021), their applicability in resource-constrained edge devices remains uncertain due to the predominant focus on cloud environments in existing auto-scaling studies.

Developing an efficient auto-scaling system for containerized services in the edge environment is challenging owing to factors including dynamic workload characteristics, resource limitations, and the distributed nature of IoT nodes. Therefore, it is imperative to explore less resource-intensive techniques such as Support Vector Regression (SVR), Random Forest (RF), and Bayesian Network (BN) and evaluate their performance/resource usage ratio, as well as their impact on the service auto-scaling process. Additionally, the potential of ensemble learning in improving prediction accuracy and its suitability for resource-constrained environments need to be investigated. Therefore, this study aims to address the following research questions:

- **RQ1**: Are widely adopted time series data analysis and forecasting techniques, such as ARIMA and LSTM, relevant and effective in resource-constrained edge environments?
- **RQ2**: Can less resource-intensive techniques like SVR, RF, and BN outperform widely adopted approaches regarding the performance/resource usage ratio?
- **RQ3**: Does ensemble learning potentially improve forecasting accuracy, and is it appropriate for resource-constrained environments?
- **RQ4**: To what extent do resource-aware workload forecasting influence the service auto-scaling process?

By investigating these aspects, this study aims to enhance the understanding of forecasting techniques, resource optimization, and auto-scaling mechanisms in the context of edge computing, ultimately contributing to developing efficient auto-scaling solutions for containerized services at the edge environments. In summary, this work makes the following contributions:

- Proposing a methodology that allows for combining and evaluating different forecasting algorithms to enable proactive service auto-scaling in the context of edge computing.
- Conducting an in-depth analysis of workload characteristics and their impact on forecasting techniques, specifically focusing on ARIMA.
- Proposing the Dynamic Ensemble learning technique, a weighted-average Ensemble learning approach that dynamically adapts to the resource-constrained edge environment by continuously adjusting the included forecasting algorithms.
- Performing a comparative study of forecasting techniques, including the proposed Dynamic Ensemble learning technique, considering prediction accuracy, resource usage, and auto-scaling performance in the constrained edge environment.

The remainder of the paper is structured as follows: Section 4.2 provides an overview of the related work in auto-scaling for containerized services. The methodology followed to address the research questions is presented in Section 4.3. The different phases of the methodology are then detailed in separate sections: data processing and analysis (Section 4.4), forecasting algorithms (Section 4.5), evaluation of models (Section 4.6), application of the ensemble learning technique (Section 4.7), model selection (Section 4.8), and the use of the selected model in the auto-scaling process (Section 4.9). The experimental evaluation is presented in Section 4.10, followed by a discussion of the results in Section 4.11. Finally, Section 4.12 concludes the paper and highlights directions for future work.

4.2 Related Work

The cloud relies on virtualization technology, which enables the execution of multiple work environments on the same server. The trends and technologies involved in cloud virtualization architectures are discussed in Varghese & Buyya (2018); Pahl *et al.* (2017). Containers considered a fundamental aspect of virtualization, facilitate the deployment of microservices on cloud servers.

Containers also offer advantages in the edge computing context, providing a lightweight and portable method to deploy services across various edge devices. This deployment ability simplifies application management and updates at the edge. Several research studies have evaluated the feasibility of using containers in edge computing while considering the devices' limited computing and storage resources. Ismail et al. (Ismail *et al.*, 2015), Morabito et al. (Morabito, 2017), and Ruchika et al. (Ruchika, 2016) have examined the use of container virtualization technology for developing and scaling IoT applications. These studies have demonstrated that Docker containers can offer good performance. Furthermore, the work proposed by Morabito et al. (Morabito & Beijar, 2016) presents a container-based data processing approach for edge computing, where the functional components are designed as reusable containers. This container-based design enables elastic provisioning of data management with orchestration capabilities. Other research studies address the limitations of container-based clustering and deployment techniques. For instance, Wong et al. (Wong *et al.*, 2019) address the lack of location-aware deployment features and aim to minimize deployment latency by selecting containers to deploy on the nearest edge node.

The auto-scaling process is highly important for optimizing resource usage while meeting customer requirements. It enables the automatic adjustment of computing entities (e.g., web servers, virtual machines, containers, or pods) based on dynamic computing needs, which can be measured using various metrics such as the number of users (Mishra *et al.*, 2020), workload (Elrotub, Bali & Gherbi, 2021), and resource usage like CPU usage (Bali *et al.*, 2020).

Several studies have explored auto-scaling at the cloud level. Kovács et al. (Kovács, 2019) define auto-scaling as a method used in distributed computing, particularly in the cloud, to dynamically and automatically adjust computing resources based on traffic workload. Scaling is also crucial for orchestration in terms of policy and flexibility for cloud containers and virtual machines. Brogi et al. (Brogi *et al.*, 2017) propose a resource management and orchestration approach based on containers to support autonomic data stream processing applications in the fog layer. At the edge level, other works address the challenges of limited-resource IoT devices. Renner et al. (Renner *et al.*, 2016) propose a container-based resource allocation model to maximize the utilization of IoT device resources and reduce network traffic by enabling dynamic resource allocation at the source level. This approach enables diverse applications and users to utilize the resources provided by IoT devices, leveraging local data processing instead of transmitting data to the cloud.

There are two main types of auto-scaling: vertical auto-scaling and horizontal auto-scaling (Al-Dhuraibi *et al.*, 2017). The distinction between these types lies in how computing resources are added to the infrastructure. In vertical auto-scaling, computing power is added to existing replicas or nodes. On the other hand, horizontal auto-scaling increases system capacity by adding more replicas (e.g., containers) to the environment, enabling load sharing across multiple devices in terms of processing and memory. In the edge computing context, where devices operate within resource-constrained environments, all available resources are dedicated to the service deployment process. As a result, horizontal auto-scaling, which dynamically adjusts the number of replicas (e.g., containers or pods) to distribute the processing load among devices in a cluster, becomes particularly suitable.

Furthermore, auto-scaling approaches can be classified into two types: reactive and proactive. Reactive auto-scalers respond to current workload changes. Due to its simplicity, the reactive approach is used in most current auto-scalers, including popular solutions like Kubernetes HPA, Google Cloud Platform, Amazon EC2, and Oracle Cloud. However, the reactive approach is unable to cope with highly dynamic workloads in which applications must scale at runtime (Zhong *et al.*, 2022). Consequently, it leads to resource over-provisioning and performance degradation when releasing necessary resources. In contrast, proactive scaling involves forecasting future workload using historical data (Lorido-Botran *et al.*, 2014). In Dang-Quang & Yoo (2021), the authors demonstrate that their proactive auto-scaler outperforms Kubernetes' default horizontal autoscaling pod (HPA) regarding accuracy and speed during resource provisioning and de-provisioning. Proactive auto-scaling, as in Sangpetch *et al.* (2017); Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021), leverages machine learning (ML) algorithms for workload forecasting using time series analysis. Different ML algorithms are used to predict the future from historical data (Lorido-Botran *et al.*, 2014). The success of future workload forecasting relies on factors such as the workload pattern, history windows, the chosen ML model, and the prediction horizon.

Workload forecasting in the context of auto-scaling often relies on time series methods, broadly categorized into two main categories: statistical time series analysis and deep learning-based techniques. Statistical time series analysis algorithms, such as ARIMA (AutoRegressive Integrated Moving Average) and exponential smoothing, have been widely used in workload forecasting (Lorido-Botran *et al.*, 2014; Sangpetch *et al.*, 2017; Calheiros *et al.*, 2014; Roy *et al.*, 2011; Kan, 2016; Li & Xia, 2016; Ciptaningtyas *et al.*, 2017; Meng *et al.*, 2016). These techniques make predictions based on historical data and assume that past behaviors and trends will continue. On the other hand, deep learning-based techniques, such as Artificial Neural Networks (ANN) and Long Short-Term Memory (LSTM) algorithms, have gained popularity in recent years for workload forecasting, such as Calheiros *et al.* (2014); Goli *et al.* (2021); Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021).

However, these commonly used approaches are slow in dynamic workload demands and resource-constrained environments (Imdoukh *et al.*, 2019). They can be computationally intensive and require large amounts of data, notably deep-learning techniques, making them less suitable for edge computing scenarios. The related work on auto-scaling often overlooks the resource-constrained aspect since most of them are intended for the cloud environment. Our work evaluates the suitability of using these techniques in a resource-constrained environment.

In addition, we also assess alternative machine learning techniques that are potentially lighter, such as Support Vector Regression (SVR), Random Forest (RF), and Bayesian Networks (BN). It is worth mentioning that these techniques have also been employed in the literature for workload prediction purposes. For instance, SVR has been applied in workload estimation and performance prediction for tasks like virtual machine migration (Raghunath & Annappa, 2015). RF has been used as a classifier for workload forecasting in data centers (Cetinski & Juric, 2015). Chen et al. (Chen *et al.*, 2020) Chen et al. proposed a workload prediction scheme that utilized a weighted random forest, which involved employing multiple random forest models, each trained on distinct training sets. The final forecast was computed by weighing the forecasts of each model. Bayesian-based methods have also been employed for workload forecasting purposes (Dietrich *et al.*, 2010; Di *et al.*, 2012). The performance of Bayesian methods has been found to be comparable to the Support Vector Machines (SVM) in load forecasting (Tong *et al.*, 2014).

However, older techniques like SVM and RF may have lower accuracy than widely used techniques like ARIMA and LSTM. Auto-scalers' efficiency utilizing time series data analysis depends on prediction accuracy (Doan *et al.*, 2019) regardless of the forecasting techniques used. Therefore, container-based auto-scaling remains an open issue that needs to be addressed, as mentioned in Qu *et al.* (2018); Cardenas (2018).

To improve prediction accuracy, this study explores the Ensemble learning technique, which combines the predictions of multiple models. It has been observed that relying on a single model may not effectively model and forecast different data types, as these models are often developed and trained for specific workload patterns (Kumar *et al.*, 2021a). Therefore, the approach of combining various methods in an Ensemble learning model is employed to enhance the modeling and forecasting of workloads.

Ensemble learning techniques have shown effectiveness in various fields, including wind gust and electricity consumption forecasting (Wang *et al.*, 2021). Ensemble learning has also been explored in the literature in the context of auto-scaling. For example, Cao et al. (Cao *et al.*, 2014) proposed an ensemble method for forecasting CPU load, which utilizes multiple models such as Autoregression and Exponential smoothing. Their approach dynamically replaces predictors to maintain the overall performance of the predictor set. They also introduced a prediction optimization layer that adjusts predictor parameters using the Adaptive Step Size Random Search (ASSRS) strategy (Schumer & Steiglitz, 1968). Similarly, in their study, Sommer et al. (Sommer *et al.*, 2016) proposed an ensemble-based forecasting tool to predict the future utilization of virtual machines (VMs) to enable proactive VM migration. Shariffdeen et al. (Shariffdeen *et al.*, 2016) presented an ensemble-based load forecasting approach specifically designed to enhance auto-scalers' accuracy. To achieve this goal, the authors evaluated various prediction models, considering different load patterns. The work conducted by Rahmanian et al. (Rahmanian *et al.*, 2018) introduced a learning-automata-based ensemble approach for forecasting cloud resource usage. By employing load forecasting techniques, Singh et al. (Singh & Rao, 2014) aimed to reduce power usage, cooling, and CO2 emissions in cloud infrastructures. They utilized Weighted Majority and Simulatable Experts to handle large-scale workloads with extensive non-stationarity and massive online streaming data.

However, many existing hybrid forecasting methods, such as those discussed in Jiang *et al.* (2013); Liu *et al.* (2015); Cetinski & Juric (2015), often encounter challenges related to high computational complexity, especially during the training phase. Moreover, these methods typically do not explicitly consider resource usage as a significant constraint, a crucial aspect addressed in this study. It is worth noting that Ensemble learning techniques, which involve utilizing multiple models, can potentially result in increased resource consumption.

To strike a balance between prediction accuracy and resource usage, this work proposes a lightweight Ensemble learning solution based on the weighted average. By considering the resource limitations, the optimization problem is formally modeled, and a heuristic solution is proposed to dynamically select the models in the Ensemble learning process. This approach ensures that the proposed Ensemble learning method used for forecasting is optimized to achieve highly accurate predictions while efficiently utilizing available resources.

4.3 Methodology

In this study, we follow the methodology presented in Figure 4.1, which contains mainly the steps of the general forecasting process: Data collection, processing, and analysis, training of the forecasting algorithms, and evaluation of models. For the evaluation, we consider the prediction accuracy and the time consumption. In addition, our process applies Ensemble learning for the purpose of improving prediction accuracy. The best model is selected according to evaluation metrics, such as accuracy and resource utilization. The last step evaluates the impact of the chosen model on the auto-scaling process.

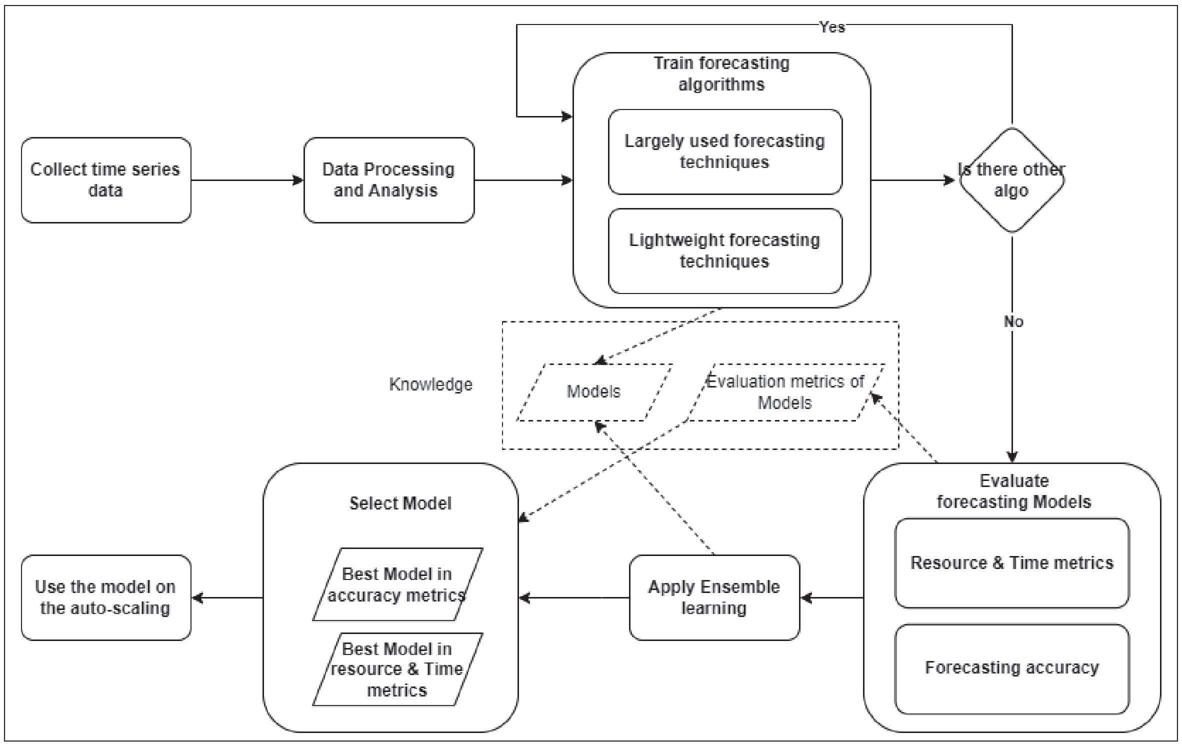

Yes
Train forecasting algorithms
Largely used forecasting techniques
Lightweight forecasting techniques
Collect time series data
Data Processing and Analysis
Is there other algo
No
Knowledge
Models
Evaluation metrics of Models
Select Model
Best Model in accuracy metrics
Best Model in resource & Time metrics
Use the model on the auto-scaling
Apply Ensemble learning
Evaluate forecasting Models
Resource & Time metrics
Forecasting accuracy

Figure 4.1 Followed Forecasting Methodology

As mentioned in the related work section (Section 2.2), forecasting the future workload is essential to make the auto-scaling process proactive. The forecasting operation is based on the quantitative forecasting of collected and pre-processed data in the first step. After that, depending on the forecasting algorithm (e.g., ARIMA) to apply, specific analysis may be required, such as the trend and seasonality. Different forecasting algorithms are considered, some are widely used (e.g., ARIMA and LSTM) and others are investigated for reducing resource utilization. After the training phase of the different forecasting algorithms, we investigate using the Ensemble learning technique that combines the predictions of multiple models to improve forecasting accuracy. Considering the resource-limitation of devices, we also asses the resource consumption metrics in the evaluation. Consequently, we consider the resource availability, in addition to the accuracy, for selecting the best model. As present in Figure 4.1, the best model is one of the following models: The best model in accuracy, the most lightweight model, the best in accuracy/time ratio, and the Ensemble learning model. Finally, we evaluate the application of the chosen model on our auto-scaling solution. The following sections give more details about the different methodology steps.

4.4 Data processing and analysis

To predict the workload, the forecasting algorithms need, as input, the time series data of the historical workload. A Time series dataset is a specialized form of data collection specifically optimized to store data with timestamps, enabling efficient analysis and understanding of trends, patterns, and fluctuations over time.

4.4.1 Selected Dataset

The auto-scaling system uses the time series data, such as CPU usage, HTTP requests) that is collected from the current service deployment. Different tools (such as CAdvise and Node Exporter) are usually used to collect different metrics of containerized services. Other tools as Prometheus, collect data from different cluster nodes. Prometheus stores data in the form of time series data. However, using this kind of collected data make our evaluation related to the data collected from a specific deployed system.

To avoid the limitations of small-scale evaluations, we conducted a large-scale assessment using the Worldcup'98 dataset (Arlitt & Jin, 2000). This approach ensures reproducibility, comparison with existing research, and a more robust thorough of our evaluation methodology. This dataset was used extensively to evaluate auto-scalers in the cloud computing literature (Imdoukh *et al.*, 2019). The dataset used in this study comprises the HTTP request logs from the FIFA World Cup Website in 1998, spanning from April 30 to July 26. It contains approximately 1.3 billion total requests. Each log entry includes various information such as a timestamp, client ID, HTTP method, and status.

4.4.2 Data Pre-processing

As mentioned, our auto-scaling approach uses a system historical data, representing time series data. WorldCup'98 dataset is a univariate time series forecasting. We organized the dataset to have only two useful pieces of information the time (period) and the count of HTTP requests. So, the overall dataset is aggregated and transformed such that each record represents the total workload, HTTP requests, per minute. Noting that, in the evaluation phase, we will consider different dataset sizes (10000, 20000 and 40000). Each case will be split into two parts training, 75%, and testing, 25%. Figure 4.2 presents a reduced data of 40000 points. One distinguishing characteristic of the WorldCup'98 DataSet, setting it apart from other workload datasets like NASA (Dang-Quang & Yoo, 2021), is its significant variation characterized by large peaks. This wide variation poses challenges for various prediction techniques, making it particularly valuable for our analysis in this study.

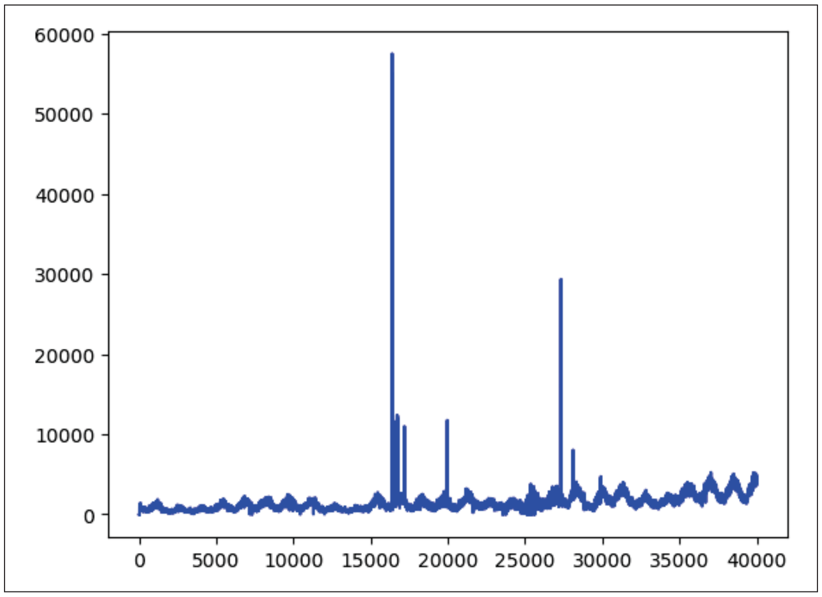


Figure 4.2 WorldCup'98 Dataset: 40,000 Points (Minutes)

4.4.2.1 Data scaling

In our case study, data scaling can enhance the performance of specific machine learning algorithms like Support Vector Regression (SVR) and LSTM (to converge faster). This process adjusts the range of independent variables or features in the dataset to a common scale, promoting equal weighting of each feature during model training. In this case, we employ min-max scaling, which transforms the data such that all feature values fall within a defined range, such as [0, 1], allowing the algorithm to converge more efficiently. A Min-Max scaling is typically performed using the Equation 4.1.

$$x^{'} = \frac{x - min(x)}{max(x) - min(x)} \tag{4.1}$$

Where, x is the original value of feature f and $x^{'}$ is the scaled value.

4.4.3 Data Analysis

In the case of ARIMA algorithm, the time-series data must be stationary. The trend and seasonality that affect the value of the time series at different times (Hyndman & Athanasopoulos, 2018) make the time series non-stationary. For that, we need to perform additional analyses in order to extract more information about the time series, which can be composed of a trend, seasonal components, and Residuals (i.e., a reminder component).

As illustrated in Figure 4.2, it can be observed that the variation of the time series is not dependent on its level, indicating that an additive decomposition is the most suitable (Hyndman & Athanasopoulos, 2018) as presented in Equation 4.2.

$$y_t = S_t + T_t + R_t \tag{4.2}$$

Where, at period t, y_t is the data, S_t is the seasonal component, T_t is the trend-cycle component, and R_t is the residual component. Figure 4.3 depicts the shape of the used time series dataset when broken down into the components mentioned above.

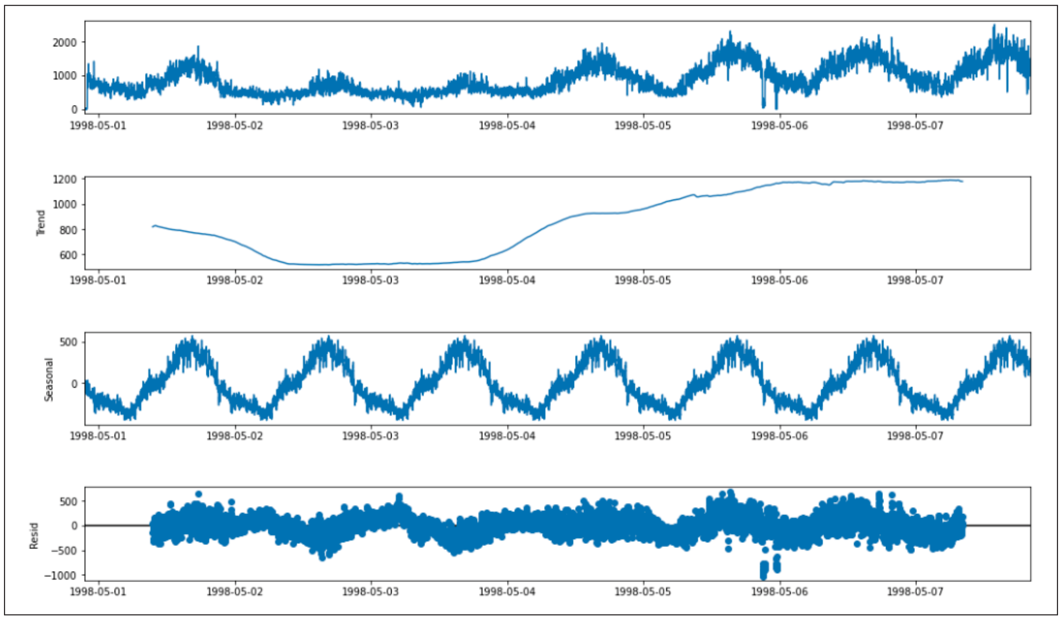


Figure 4.3 Data Decomposition

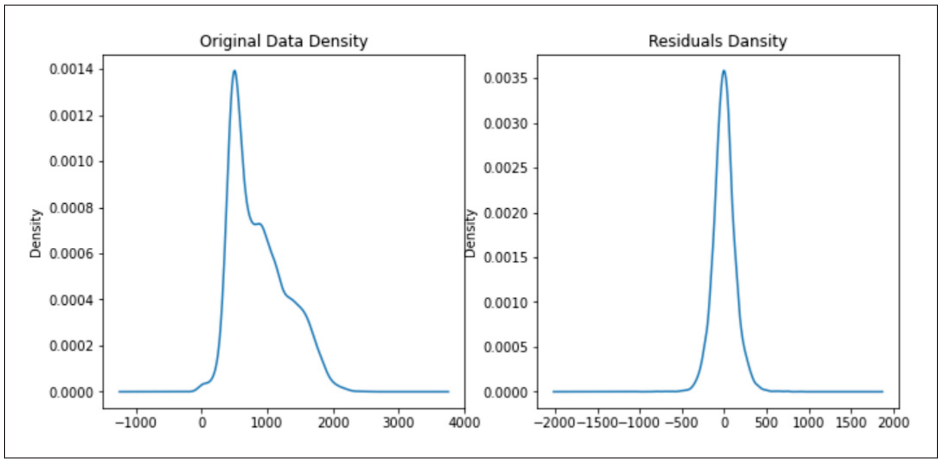


Figure 4.4 Data Density Plot

Our time series shows a fluctuating trend with a slight upwards. The residuals' variance is less than the historical data, and it is a little stable through time with a slight increase at the end, which shows that the series exhibits more random behavior at the end. Figure 4.4 shows the distribution of the residual data (right plot) compared to the original data distribution. The residual data is a normal distribution (Gaussian) centered on zero. Therefore, the residuals have zero mean and constant variance, meaning white noise.

To analyze the seasonality further, we consider different seasonality resolutions: hourly, daily, and weekly. Figure 4.5 plots individually these seasonal components. To accurately calculate the daily component, it is first essential to subtract the hourly component and then apply the weekly decomposition step after all other relevant seasonal components have been removed from the data. The seasonal patterns in the data reveal a relatively insignificant hourly seasonality,

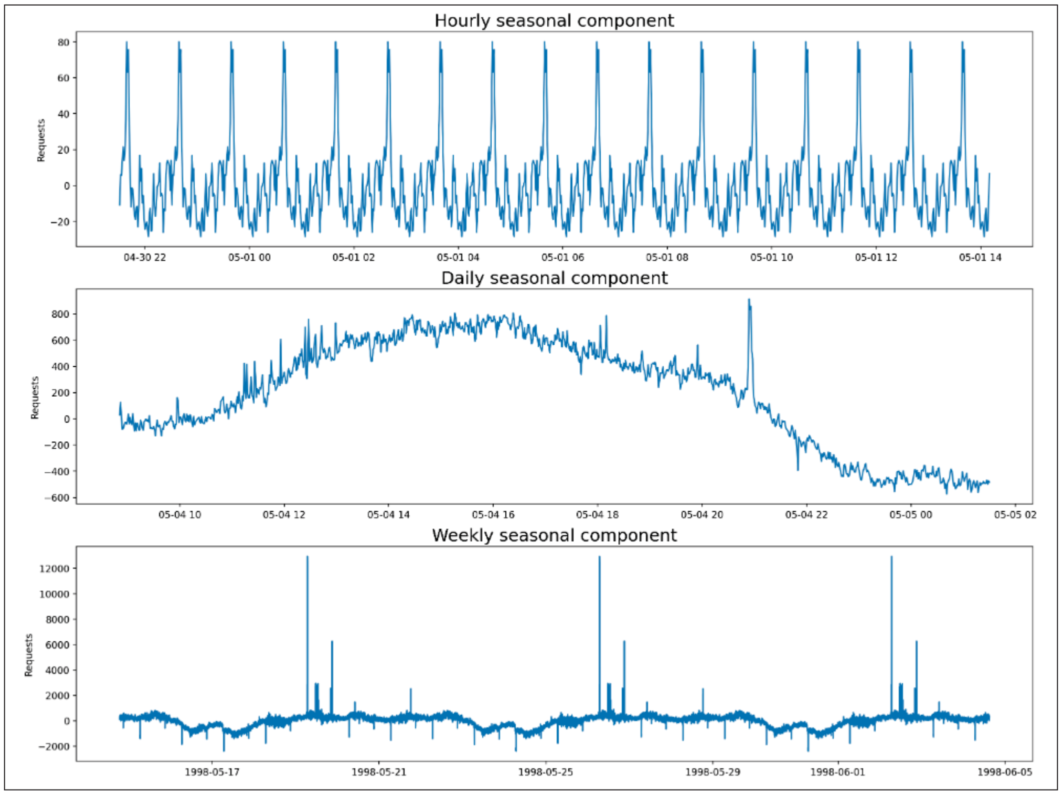


Figure 4.5 Seasonality Analysis

while no discernible daily or weekly seasonality can be observed.

To further investigate whether the time series data is stationary, we perform further analyses on the mean, variance, and autocorrelation features. Figure 4.6 visualizes the evolution of these features. It is important to mention that a stationary time series requires a constant mean,

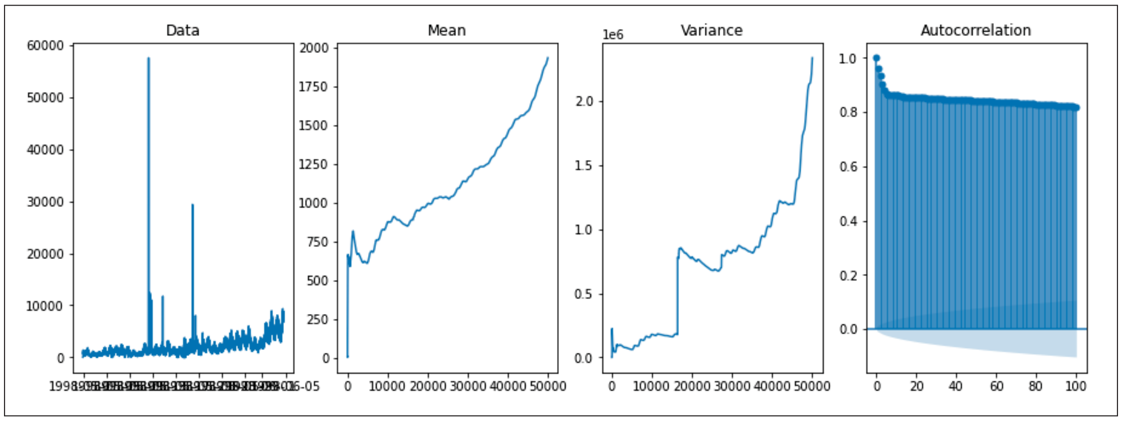


Figure 4.6 Stationary Analysis: Original Data

variance, and autocorrelation, which is not true for our time series dataset.

Some forecasting algorithms, such as ARIMA (Hyndman & Athanasopoulos, 2018), require converting the time series data to be stationary. Even for LSTM, removing non-stationary elements from the data, it will make the prediction task less complex and thus make it more efficient (Brownlee, 2016). Since we have a slight trend, the log transformation can help remove the trend component. If we consider the weak hourly seasonal, we can differentiate the data.

To evaluate the logarithm transformation and the differencing impact, Figure 4.7 plots statistical features of the 1^{st} differencing on the original time series data, its logged transformation and the 1^{st} differencing on the logged data. As in Figure 4.6, the presented features are the mean, variance and the autocorrelation, which need to be stable to make the transformed data stationary.

In the first differencing over the original data, the variance is not stable, in contrast to the two other cases of the logged data. Moreover, the differencing of the logged data makes the value of means close to zero. By this analysis, we conclude that the first differencing of the log transformation of the used dataset ensures that the dataset is stationary, which is helpful for forecasting algorithms.

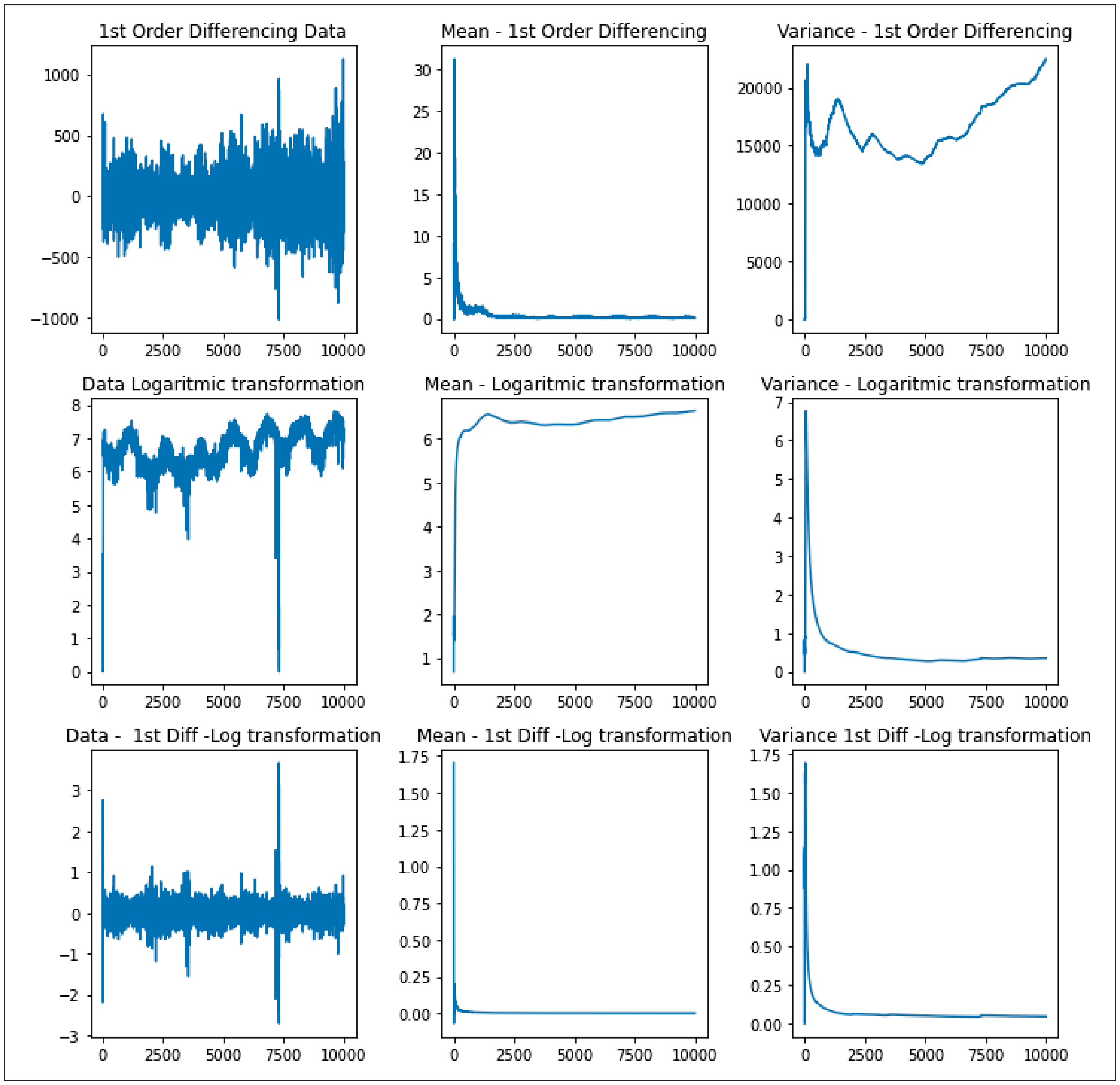


Figure 4.7 Stationary Analysis of Data: 1st Differenced Data, Logged Data, 1stDifferenced of Logged Data

It is worth mentioning that Dicky-Fuller test (Dickey & Fuller, 1979) considers the data non-stationary since the p-value is greater than 0.05. However, since our time series data has slight trend and seasonal components, the p-value of 0.056638 is very close to the reference value of 0.05. In addition, based on the statistical critical values, Dicky-Fuller test considers the data stationary with 90% certain. We will discuss this aspect further in the evaluation of ARIMA algorithm.

4.4.4 Data reframing

The data reframing step aims to convert the time series prediction task into a supervised learning task by utilizing previous time steps as input variables and the subsequent time step as the output variable. In simpler terms, it is possible to predict the value at the next time step by using a sequence of numbers from a time series dataset, known as a sliding window.

Formally, let $TS =< d_1, d_2, \ldots, d_n >$ represents the original time series data, where each d_i corresponds to a data point at time i. The window size is defined by S a positive integer. The data reformulation involves partitioning the original time series into overlapping windows of size S. The i^{th} input-output pair consist of the window $W_i =< d_i, d_{i+1}, \ldots, d_{i+S-1} >$ and the corresponding value $y_i = d_{i+S}$.

4.4.5 Horizon of prediction

The data prepared in the previous step is suitable for single-step prediction. For multi-step prediction, a straightforward approach is to include the corresponding output value in the training process. However, this approach is impractical as it would require multiple models, one for each horizon step. For example, five models would be needed for a prediction horizon of five-time steps. An alternative approach is recursive multi-step prediction, but its limitation is that accuracy decreases as the horizon size increases.

4.5 Forecasting algorithms

In this study, our initial focus is on two commonly employed prediction methods: ARIMA and LSTM. We also consider a less resource-intensive technique, namely SVR. The following subsections present these three techniques and their use in our working context.

4.5.1 ARIMA

ARIMA models are designed to capture the autocorrelation and stationary of time series data. They have proven to be highly accurate for short-term forecasting of statistically dependent time series (Box, Jenkins, Reinsel & Ljung, 2015). To utilize ARIMA for workload forecasting, the historical workload data is fitted to the model by specifying the hyper-parameters p, d, and q of the ARIMA model. The AR parameter (p) represents the number of lags in the autoregressive term, the differencing parameter (d) determines the degree of differencing required to make the time series stationary, and the MA parameter (q) signifies the size of the moving average window used in the model. The AR and MA terms are presented in equations 4.3 and 4.4, respectively.

$$Y_t = \alpha + \beta_1 y_{t-1} + \beta_2 y_{t-2} + \cdots + \beta_p y_{t-p} + \epsilon_t \tag{4.3}$$

Where, $yt - i$ is the i^{th} lag of the series, β_i is its coefficient, and α is the intercept term. The values of these coefficients are estimated by training the model.

Moving Average (MA) term of order q depends on the lagged forecast errors as expressed in Equation 4.4.

$$Y_t = \alpha + \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \cdots + \theta_p \epsilon_{t-q} \tag{4.4}$$

Where, ϵ_t is white noise (Hyndman & Athanasopoulos, 2018) and θ_i is a coefficient.

After making the time series stationary by differencing it (y') in the number of d, the ARIMA(p,d,q) model combines AR and MA terms Auto-Regressive (AR) term, as presented in Equation 4.5.

$$Y_t = \alpha + \beta_1 y'_{t-1} + \cdots + \beta_p y'_{t-p} + \theta_1 \epsilon_{t-1} + \cdots + \theta_p \epsilon_{t-q} + \epsilon_t \tag{4.5}$$

4.5.2 LSTM

Long Short-Term Memory (LSTM) is an advanced version of recurrent neural networks (RNN) that addresses the vanishing gradient problem, often hampers the learning capability of RNNs.

LSTM models effectively predict sequences in time-series data, such as workload patterns over time.

The LSTM network comprises interconnected LSTM units. Each unit has an internal memory cell and incorporates three gates: the input gate, the forget gate, and the output gate. The gates employ sigmoid functions and multiplication operations to regulate the flow of information. The input gate determines whether the input should modify the cell's content, while the forget gate decides whether to reset the cell's content to 0. The output gate governs whether the cell's content should influence the neuron's output.
Figure 4.8 depicts the architecture of an LSTM unit, illustrating the various outputs of the gates as expressed in the following equations.

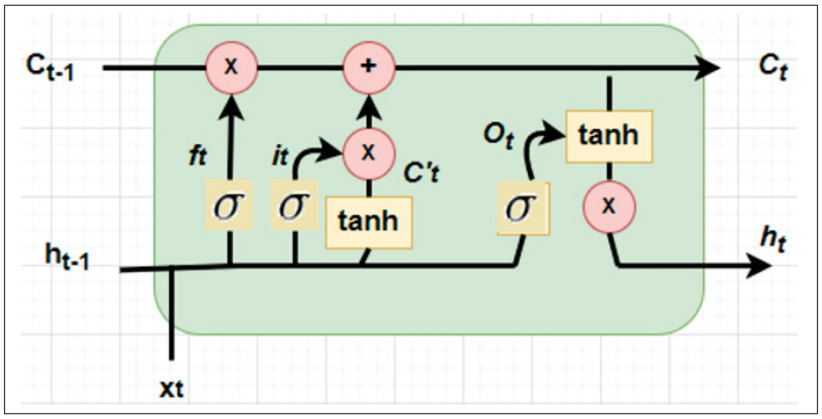


Figure 4.8 LSTM Unit Architecture

$$f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f) \tag{4.6}$$

$$i_t = \sigma(W_i.[h_{t-1}, x_t] + b_i) \tag{4.7}$$

$$C'_t = \tanh(W_c.[h_{t-1}, x_t] + b_c) \tag{4.8}$$

$$C_t = f_t \times C_{t-1} + i_t * C'_t \tag{4.9}$$

$$o_t = \sigma(W_0.[h_{t-1}, x_t] + b_0) \tag{4.10}$$

$$h_t = \sigma(o_t \times tanh(C_t)) \tag{4.11}$$

Where, W and b represent the weights and bias, respectively. In addition, σ (sigmoid) and $tanh$ denote the used activation functions.

4.5.3 SVR

Support Vector Regression (SVR) is another machine learning algorithm that can be used for time series forecasting. It is a regression algorithm based on the principles of support vector machines, which are commonly used for classification tasks. In SVR, the goal is to find a function that best maps the input features to the target variable. This function best fits the data while maximizing the margin, i.e., the distance between the function and the closest data points. These closest data points are known as support vectors (Steinwart & Christmann, 2008), and they play a critical role in determining the final regression function. To simplify, the prediction for a new instance, X', can be made by plugging it into the learned regression function, f(X') as presented in Equation 4.12.

$$f(X') = W' \times X' + b \tag{4.12}$$

Where, W is the weight vector and b is the bias term, learned during the training process, and X' is the feature vector for the new instance.

4.5.4 BN

Bayesian networks, which can be used for time series forecasting, represent probabilistic graphical modeling for building models from data. A Bayesian network is an acyclic-directed graph, where each node is associated with quantitative probabilistic information. Each node is attached to a random variable and to links (arcs) connecting it to other nodes. A link from the node X to the node Y means that X is the parent of Y and implies a conditional dependency relationship between them. Each node A_i is characterized by a conditional probability distribution, expressed in Equation 4.13.

$$P(A_i|parents(A_i)) \tag{4.13}$$

Bayes' theorem, described by Equation 4.14, allows the calculation of conditional probability distributions for a set of interacting variables.

$$P(H|E) = \frac{P(E|H) \times P(H)}{P(E)} \tag{4.14}$$

Where, H is the hypothesis and E is the evidence. $P(A|B)$ is generally interpreted as "the probability of A given B", where A is the dependent variable and B is the independent variable. The theorem of Bayes determines the probability of the hypothesis H based on the evidence E.

4.5.5 RF

Random Forest (RF) is a tree-based ensemble machine learning algorithm that operates by constructing a set of decision trees from a random subset of the training data and then aggregating their predictions to produce the final prediction. The prediction for a new instance, X', is made by passing it through each of the T decision trees and aggregating the predictions. The aggregation can be done by taking the average of the individual trees' predictions, as presented in Equation 4.15.

$$f(X') = \frac{1}{T}\sum_{n=1}^{T} f_i(X') \tag{4.15}$$

Where, $f_i(X')$ is the prediction made by the decision tree number i, and T is the number of trees in the forest.

4.6 Model Evaluation

We used the prediction accuracy and time consumption metrics for the algorithm performance evaluation.

4.6.1 Prediction Accuracy Metrics

The efficiency of the forecasting models is based on the error evaluation between the actual and predicted data. Different models are considered to explain the errors and predictive ability of models (Chicco *et al.*, 2021), the Root Mean Square Error (RMSE) presented in eq. 4.18 and the coefficient of determination (R^2) presented in Eq. 4.19. The Mean Absolute Error (MAE) and Mean Square Error (MSE) can also be considered for further comparison (equations 4.16 and 4.17). In addition, we considered Mean Absolute Percentage Error (MAPE) 4.20, which gives an idea of the magnitude of the error in percentage form. The percentage values, as scaled quality values, serve in applying the Ensemble learning technique (Section 4.7).

$$MAE = \frac{1}{n}\sum_{i=1}^{n} \|y_i - \hat{y}_i\| \tag{4.16}$$

$$MSE = \frac{1}{n}\sum_{i=1}^{n} \left(y_i' - y_i\right)^2 \tag{4.17}$$

$$RMSE == \sqrt{MSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} \left(y_i' - y_i\right)^2} \tag{4.18}$$

$$R^2 = 1 - \frac{\sum i\left(y_i - \hat{y}_i\right)^2}{\sum i\left(y_i - \bar{y}_i\right)^2} \tag{4.19}$$

$$MAPE = \frac{100}{n}\sum_{i=1}^{n} \|\frac{y_i - \hat{y}_i}{y_i}\| \tag{4.20}$$

Where, y, $\hat{y}_i$ and $\bar{y}_i$ represent the actual value, predicted value, and mean values of the variable y, respectively.

Since the MAPE error is a percentage value (i.e., from 0 to 1), we used is it to calculate the model quality, as presented in Equation 4.21.

$$Quality(m_i) = 1 - MAPE(m_i) \tag{4.21}$$

It is worth noting that we utilize this Quality metric in our work, particularly when applying the Ensemble learning method.

4.6.2 Runtime Metrics

To evaluate the resource usage aspect, we considered the runtime for both the model training (i.e., training time) and its prediction speed (i.e., prediction time).

- Training time: its value informs if the model can be trained on the device. This time corresponds to the execution time of the training without a testing phase.
- Prediction time: it also verifies the possibility of using the trained model on the resource-constrained device. Moreover, it gives an idea about how much of the device's prediction capacity has been used. For example, consider an auto-scaler that updates the state of a system (e.g., scaling the number of replicas) every minute. It may reserve only 10 seconds for workload forecasting and the rest (i.e., 50 seconds) for planning and executing the scaling plan. If the prediction operation using a forecasting algorithm takes 5 seconds, then the prediction by the model uses half of the device's prediction capacity.

In addition, if the auto-scaling system uses more than one forecasting model, the prediction time is the sum of the prediction times of each used model as presented in Equation 4.22. The assumption of worst-case sequential execution of forecasting models is driven by resource limitations, especially memory constraints, which may restrict the parallel execution of models.

$$overall\_PT = \sum_{i=1}^{|models|} PT_i \tag{4.22}$$

4.6.3 Evaluation Process of Models

Using the different evaluation metrics, the evaluation process of forecasting models is presented in Algorithm 4.1. It calculates the prediction time of each model based on the testing time. The training time of models is supposed to be calculated in the previous step of our methodology (i.e., Train forecasting algorithms).

The algorithm has, as inputs, the set of forecasting models ($Models$) and the test dataset (TS_{test}) reformulated in the form of a sequence of time series windows (W_i). The next values of different time series windows are stored in list $actualData$ (Line 1). In the nested loop (lines 5-8), the list of predicted values adds the predicted value obtained by using a forecasting model. In addition, the algorithm calculates the testing time and the average of the predicting time in lines 10 and 11, respectively. Finally, it calculates, in Line 12, the different error metrics based on the $actualData$ and $predictedData$. The model quality is calculated based on the MAPE metric, as presented in Equation 4.21.

Algorithm 4.1 Evaluate Models

```
   **Input:**
   $Models = \{M_1, M_2, \cdots, M_n\}$
   $TS_{test} =< W_1 =< x_1, x_2, \cdots, x_s >, W_2 =< x_2, x_3, \cdots, x_{s+1} >, \cdots, W_m =< x_{m-s-1}, \cdots, x_{m-1} >>$
   **Output:** $Models$

1  $actualData \leftarrow< x_{s+1}, x_{s+2}, \cdots, x_m >$;
2  $predictedData \leftarrow<>$;
3  **for** $model\ in\ Models$ **do**
4      $startTime \leftarrow Now()$;
5      **for** $Each\ W_i\ in\ TS_{test}$ **do**
6          $pred \leftarrow model.predict(W_i)$;
7          $predictedData.append(pred)$;
8      **end for**
9      $endTime \leftarrow Now()$;
10     $model.TestingT \leftarrow endTime - startTime$;
11     $model.PT \leftarrow model.TestingT / |actualData|$;
12     $MAE, MSE, RMSE, R^2, MAPE \leftarrow errors(actualData, redictedData)$;
13     $model.Q \leftarrow 100 - MAPE$;
14 **end for**
```

4.7 Ensemble Learning Technique Application

Ensemble learning is a technique that involves combining the predictions of multiple models in order to improve forecasting accuracy. The idea behind ensemble learning is to take advantage of the strengths of different models by combining their predictions together. This can lead to a more robust and accurate forecast than any single model could provide (Zhou, 2012). Ensemble methods can be powerful for time series forecasting as it allows combining the strength of different algorithms, reduce overfitting, and improve generalization (Wheelwright, Makridakis & Hyndman, 1998). Several ensemble learning can be used to improve forecasting accuracy, such as Bagging, Boosting, Stacking, and Blending (Zhou, 2012).

4.7.1 Our Application of Ensemble Learning

In our case, we consider the multiple trained models, such as ARIMA, LSTM, RF, BN, and SVR. Our Ensemble learning combines their predictions to improve the overall performance. Considering the resource limitation, we use a basic technique, Weighted Average, which avoids training a new model for the combination. Different weights are assigned to forecasting models to define the prediction quality (defined in Equation 4.21).

The weight of each model is its scaled prediction quality value, calculated by Equation 4.23.

$$Weight(m_i) = \frac{Quality(m_i)}{\sum_{i=1}^{|models|} Quality(m_i)} \tag{4.23}$$

Algorithm 4.2 presents our ensemble learning solution. The algorithm takes as input the set of models to consider, along with their prediction quality. It also receives the time series window for predicting the next value or the prediction horizon.

In Line 7, it calculates the weights of different models according to Equation 4.23. Afterward, in lines 6-9, it calculates the prediction of the ensemble learning based on the predictions

Algorithm 4.2 Ensemble Learning Forecasting

```
Input:
$Models = \{M_1, M_2, \ldots, M_k\}$
$W =< x_{t-s}, x_{t-s-1}, ..., x_{t-1}, x_t >$
Output: $ELPrediction$
1 $sum \leftarrow 0$;
2 for $model$ in $Models$ do
3     $sum \leftarrow sum + model.Quality$;
4 end for
5 $ELPrediction \leftarrow 0$;
6 for $model$ in $Models$ do
7     $weight \leftarrow model.Quality/sum$;
8     $ELPrediction \leftarrow ELPrediction + Model.predict(W) * weight$;
9 end for
```

of different models. Finally, the algorithm generates the prediction set of different models, including Ensemble learning, for their use in model selection.

It should be noted that if the models' predictions have already been calculated in the previous step, they can be sent to the algorithm that uses them instead of calculating them. As a result, the execution time of the algorithm becomes significantly reduced. For the sake of understanding, we included calculating the predictions of the models in the algorithm. Also, for the same reason, we presented how to calculate the forecast of the ensemble learning for a single window. To calculate the Ensemble learning model's quality and average prediction time on the test dataset, we use the same Algorithm 4.1. Except for the prediction operation (Line 6) should call Algorithm 4.2.

In the evaluation section (Section 4.10), we will evaluate the utility of adding ensemble learning. However, using Ensemble learning can increase the complexity of models (Seni & Elder, 2010). It requires the execution of all models instead of only one model. In the context of resource-constrained devices, the problem becomes worse. Resource limitation may prevent the execution of all models, either for memory limitation or exceeding the time allocated for prediction. As a result, we aim to propose a solution that adapts the set of models to be considered in Ensemble learning according to the availability of resources.

4.7.2 Problem Formulation

Given multiple models, each with a prediction time and a quality value. In addition, given a maximum prediction time for the workload prediction, the optimization problem aims to answer the question: which forecasting models should be integrated into the ensemble learning model to maximize the total value of quality without exceeding the maximum prediction time allowed by the auto-scaler?

We have a maximum prediction time for the Ensemble learning model and n forecasting models. For each model i (m_i), we have a prediction time PT_i and a quality value Q_i.

The decision variable xi associated with the model m_i is defined as follows: $xi = 1$ if the model m_i is considered in the ensemble learning model, and $x_i = 0$ if the model m_i is not selected.

In our problem, we define a single constraint: the sum of the prediction times, PT_i, of all the models in the ensemble learning must be less than or equal to the maximum prediction time, $MaxTP$, as presented in Equation 4.24.

$$Weight(m_i) = \sum_{i=1}^{n} x_i \times PT_i \leq MaxPT \tag{4.24}$$

Finally, the objective function aims to maximize the total quality value of the models integrated into the ensemble learning (Equation 4.25).

$$\max \sum_{i=1}^{n} x_i \times Q_i \tag{4.25}$$

Our problem can be mapped to the knapsack problem (Kellerer *et al.*, 2004). Where, the prediction models to be considered in the Ensemble learning correspond to the objects to be put in the knapsack. A model's quality and prediction time correspond to the value and weight of an object, respectively. Whereas the total allowed prediction time corresponds to the maximum allowed weight for the knapsack. It should be noted that the knapsack problem is one of the 21 NP-complete problems identified by Richard Karp (Karp, 1975).

4.7.3 Proposed Dynamic Ensemble learning (DEL)

Given the resource constraints, we solve this optimization problem using a heuristic method instead of exact methods, which can be time-consuming and demand more resources. However, heuristic solutions allow us to obtain an approximate solution quickly, but not necessarily optimal.

Our heuristic solution, presented in Algorithm 4.3, is based on sorting the different models (Line 4) according to the values of the ratio Q_i/PT_i of the models calculated in the first loop (lines 1-3). Then, it selects, in order, the models one by one (Line 7) and if the maximum prediction time is still respected (Line 7), it adds the selected model to the set of models in the ensemble learning (Line 8).

4.8 Model Selection

For the model selection phase of our methodology, we expect to have as output a set of models, including the Ensemble learning model. The model selection process is similar to the application of Ensemble learning, where we consider both the quality and prediction time (PT) of models. The selected model has the best forecasting accuracy compared to the feasible models, which can be executed on the restricted-resource device. A feasible model means that its time prediction is less or equal to the maximum prediction time authorized by the auto-scaling system. Equations 4.26 and 4.27 present the extraction of the feasible models and the model selection, respectively.

$$feasibleModels = \{m \in Models \mid PT(m) \leq MaxPT\} \tag{4.26}$$

Algorithm 4.3 Ensemble Learning Optimization

```
Input:
$Models = \{M_1, M_2, \ldots, M_n\}$
$MaxPT$
Output: $selectedModels$
1  $selectedModels \leftarrow \{\}$for $model\ in\ Models$ do
2      $model.opt\_value \leftarrow model.Q/model.PT$
3  end for
4  $Sorted\_Models \leftarrow Sort(Models, opt\_value, descending)$;
5  $current\_PT \leftarrow 0$
6  for $i = 0\ in\ |Sorted\_Models|$ do
7      $model \leftarrow Sorted\_Models[i]$ $current\_PT \leftarrow current\_PT + model.PT$ if
         $current\_PT \leq MaxPT$ then
8          $selectedModels \leftarrow selectedModels \cup model$
9      else
10         $break$
11     end if
12 end for
```

Where, $Models$ is the set of all models, PT is the prediction time and $MaxPT$ is the maximum time reserved for the prediction operation.

$$\forall m \in feasibleModels,\ Quality(selectedModel) \geq Quality(m) \tag{4.27}$$

4.9 Integration of the Model in the Auto-scaling Process

In this last step, we apply the selected model in the auto-scaling process. The role of the selected model is to predict the future workload. According to the predicted workload, the auto-scaler proactively adapts the system resource by scaling up\down the service replicas.

To assess the efficiency of auto-scaling, we adopt the metrics proposed by Herbst et al. (Herbst *et al.*, 2016) and Bauer et al. (Bauer *et al.*, 2018a), which are widely utilized in the literature on auto-scaling, such as Imdoukh *et al.* (2019); Dang-Quang & Yoo (2021).

The under-provisioning metric, denoted as θ_U, quantifies the number of replicas (e.g., containers) required to meet the desired number of replicas, as represented in Equation 4.28. Additionally, the under-provisioning time (T_u) captures the duration during which the simulator experienced under-provisioning, as indicated in Equation 4.29.

$$\theta_u = \frac{100}{T} \sum_{i=1}^{T} \frac{max(required(t) - provided(t), 0)}{required(t)} \Delta t \tag{4.28}$$

$$T_u = \frac{100}{T} \sum_{i=1}^{T} \max(sgn(required(t) - provided(t)), 0) \Delta t \tag{4.29}$$

Where, $required(t)$ represents the correct number of replicas corresponding to the actual workload, while $provided(t)$ represents the number of replicas offered by the auto-scaler based on the predicted workload value. The time interval for checking the workload change is denoted as Δt, typically set at a minute level. The evaluation period is represented by T, and the function $sgn()$ denotes the sign function.

On the other hand, the over-provisioning metric, denoted as θ_O, quantifies the number of replicas supplied that exceed the desired number, as expressed in Equation 4.30. Additionally, the over-provisioning time (T_o) indicates the duration during which the auto-scaler experienced over-provisioning, as illustrated in Equation 4.31.

$$\theta_o = \frac{100}{T} \sum_{i=1}^{T} \frac{max(provided(t) - required(t), 0)}{required(t)} \Delta t \tag{4.30}$$

$$T_o = \frac{100}{T} \sum_{i=1}^{T} \max(sgn(provided(t) - required(t)), 0) \Delta t \tag{4.31}$$

Finally, the Elasticity speedup (ϵ_n) reveals the performance gain obtained by using a proactive auto-scaler. In this work, the elasticity speedup is calculated by a ratio between two cases: using a proactive auto-scaler and a reactive auto-scaler, which are represented, in Eq. 4.32, by the p

and r indices, respectively.

$$\epsilon_n = \left(\frac{\theta_{u,r}}{\theta_{u,p}} . \frac{\theta_{o,r}}{\theta_{o,p}} . \frac{T_{u,r}}{T_{u,p}} . \frac{T_{o,r}}{T_{o,p}} \right)^{\frac{1}{4}} \tag{4.32}$$

In contrast to the other metrics, the higher the ϵ_n value, the higher the auto-scaling performance. In other words, the best auto-scaling has less $\theta_u, \theta_o, T_u, T_o$ values and essentially a higher ϵ_n value.

4.10 Evaluation Results

In this section, we outline the steps taken to evaluate the feasibility and performance of our approach within the constraints of resource-limited devices, specifically in the context of IoT edge environments. The evaluation focuses on forecasting algorithms, including ARIMA, LSTM, SVR, RF, and BN. It considers the accuracy and execution time metrics discussed in Section 4.6, such as MSE, MAE, $RMSE$, R^2, and $MAPE$. Furthermore, we evaluate the impact of predictions on the auto-scaling process using the metrics presented in Section 4.9. The evaluation process follows the steps of the MAPE-K loop: Monitoring, Analysis, Planning, and Execution. In the monitoring phase, prediction data is obtained from the forecasting techniques applied to the WorldCup'98 dataset. The analysis phase involves the auto-scaler assessing the need for scaling, whether up or down, based on the difference between the current workload and the predicted workload generated by the forecasting algorithms. The planning phase calculates the number of replicas (e.g., containers) required to meet the predicted workload, considering the maximum capacity of a replica (e.g., one container can process 500 requests per minute). Finally, the execution phase scales the system to the new number of replicas.

For the implementation of the prediction algorithms and the auto-scaler, we utilized Python with the Scikit-learn and Keras libraries. To evaluate the performance of forecasting algorithms on resource-constrained devices, we conducted experiments on a Raspberry Pi 3 Model B, equipped with the Raspbian operating system, a 1-core CPU, and 1 GB of RAM.

To ensure diverse experimentation and comprehensive analysis, we considered three classes of dataset sizes: 10,000 points (10k), 20,000 points, and 40,000 points, classified as Small, Medium, and Large data sizes, respectively. Each data point represents the number of requests received in a minute. Figure 4.12 illustrates these three size classes. In the case of 10k size Subfigure 4.12a, the testing data exhibits patterns similar to the training data. For the 20k size Subfigure 4.12b, the testing data contains additional patterns not present in the training data, characterized by diverse request peaks, which make them challenging to predict. In the case of 40k size Subfigure 4.12c, we introduced highly varied data into the training data portion. It is important to note that a 75%-25% ratio was used to split the data for training and testing the models.

4.10.1 Configuration of Models

Table 4.2 summarizes the configuration parameters for the various forecasting models used in this study. In the following subsubsections, we briefly explain the model parametrization for each algorithm, highlighting the key parameters tuned to optimize the model performance.

4.10.1.1 ARIMA

To utilize ARIMA, we need to determine the model order ARIMA(p,d,q), where p represents the order of the Auto-regressive (AR) part (i.e., the number of AR terms), d represents the order of differencing ("I" part), and q represents the order of the Moving-average (MA) part. In Section 4.4, we discussed the aspect of data differencing to make the data stationary. We demonstrated that applying first-order differencing (d=1) on the logged data suffices.

In addition, we can determine the values of p and q, by analyzing the autocorrelation and partial autocorrelation (PACF) of the stationarized data represented, respectively, in Figure 4.9 and Figure 4.10. This blue area indicates the significance threshold that corresponds to the 95% confidence interval, which is statistically non-zero.

The auto-ARIMA function selects the best model based on the AIC value during the training phase. However, the selected model, ARIMA(8,0,1), has a high autoregressive order (p = 8). Consequently, this increased value of p can potentially result in overfitting of the model to the

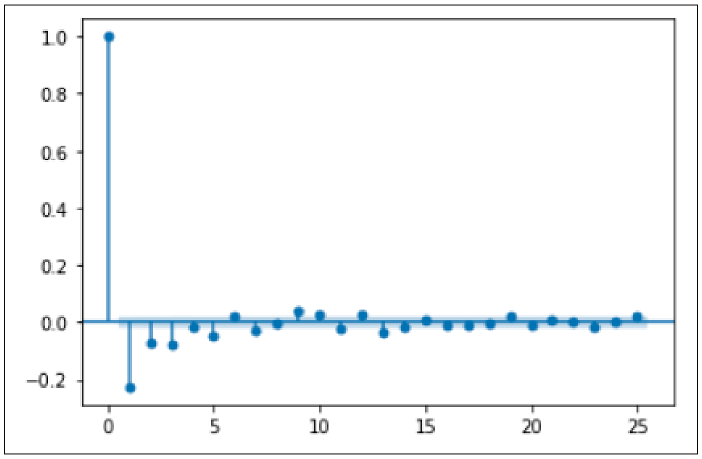


Figure 4.9 Autocorrelation Function (ACF)

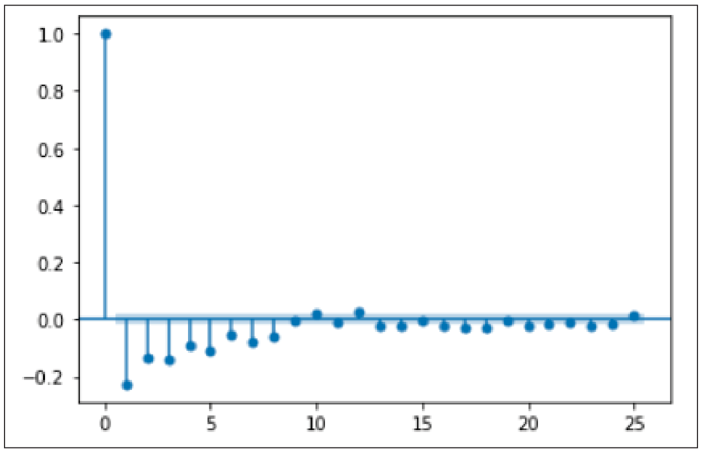


Figure 4.10 Partial Autocorrelation Function (PACF)

training data, leading to less efficient performance on the test data. This observation helps explain the results presented in Table 4.1, where our analysis-based model outperforms the auto-ARIMA model across most accuracy metrics. Furthermore, increasing the values of p or q

Table 4.1 Comparison between ARIMA Models based on AIC and Error Metrics

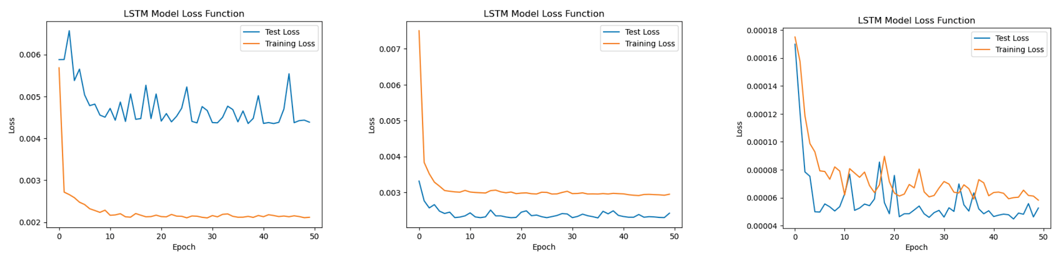
ARIMA Model	Order	AIC	MSE	RMSE	MAE	$\hat{R2}$	MAPE
Analysis-based	(2,1,2)	-1577.6352	**27435.0543**	**165.6353**	**127.6111**	**0.8279**	**0.1159**
Auto-ARIMA	(8,0,1)	**-1630.592**	29167.4965	170.7849	130.7615	0.8171	0.1164

can increase the complexity of the model, resulting in longer training times. Considering these factors, we use our analysis-based model (ARIMA(2,1,2)) for our evaluation.

4.10.1.2 LSTM

For the LSTM model, we employed a sequential architecture. The input layer corresponds to a sliding window size of 30, while the output layer consists of a single neural cell. We utilized the Adam optimizer to train the model. To optimize the hyperparameters of our LSTM model, we performed a grid-search technique. The values for the grid were obtained from previous literature works (Imdoukh *et al.*, 2019; Dang-Quang & Yoo, 2021) and an initial model. After evaluating different combinations, we determined that setting the number of units to 100, the number of epochs to 50, and the batch size to 64 yielded the best results (Table 4.2).

It is worth noting that the LSTM model can achieve higher accuracy when increasing the data volume, as it is a deep learning algorithm, as shown in Figure 4.11, where the test loss and train loss gradually become closer when increasing the data volume. However, the increase in data is limited by resource availability, as presented in the previous subsection. It is important to mention that in the 10k case, the model is not well trained, as there is a considerable difference between the training and loss plots. Additionally, the plateauing effect in the 40k case suggests that the model might not be overtrained.



a) Case of Data size=10k b) Case of Data size=20k c) Case of Data size=40k

Figure 4.11 LSTM Model Training

4.10.1.3 SVR

For SVR (Support Vector Regression), two main hyperparameters are to consider: C and gamma. A higher value of C indicates a smaller-margin hyperplane. We experimented with several values of C, including 0.1, 1, 10, and 100. As for gamma, we tested the values of 0.01, 0.1, and 1. A higher gamma value focuses only on points close to the boundary lines when determining the position of the hyperplane. In contrast, a lower gamma value considers points both close and far from the boundary lines. Additionally, SVR offers various kernel function options such as linear, RBF (Radial Basis Function), poly, and sigmoid. We also considered the epsilon hyperparameter, which can take values of 0.01, 0.1, and 1. Similar to other models, we conducted a grid search to optimize the model configuration by exploring different combinations of hyperparameter values. The selected SVR model and its hyperparameter settings are presented in Table 4.2.

4.10.1.4 RF

For the Random Forest algorithm, there are two main hyperparameters to consider: the number of trees (number_trees) and the maximum depth of each decision tree in the forest (max_-depth). Increasing the number_trees can improve the model's performance but also increases computation time and the risk of overfitting. On the other hand, the max_depth parameter limits the depth of the trees, which helps prevent overfitting and enhances the model's generalization performance. To tune these hyperparameters, we conducted a grid search with the following values: number_trees (50, 100, 150) and max_depth (5, 10, 15). As shown in Table 4.2, the best configuration is number_trees = 150 and max_depth = 10. We evaluated the models' performance based on the negative mean squared error. The results may vary since the Random Forest algorithm randomly selects subsets of features and data points at each split. To ensure reproducibility, we set the random_state hyperparameter to 42.

4.10.1.5 BN

For the BN (Bayesian Network) algorithm, we utilized the PyBATS package in Python for Bayesian time series analysis. PyBATS is based on Dynamic Generalized Linear Models (DGLM), which allows coefficients to change dynamically over time, generalizes based on the observation distribution (e.g., Normal or Poisson), and linearly combines coefficients multiplied by predictors. Similar to other models, we performed a grid search to determine the best configuration of model parameters. Table 4.2 presents the considered parameters, including the distribution family (family = "Poisson") for analyzing count-based time series, the number of samples (nsamps = 100) for obtaining credible intervals and point estimations, the prior length (prior_length = 6) used to define the prior distribution, the random effect extension (rho = 0.9) that increases forecast variance, and the discount factors for trend and regression components (deltrend = 0.5 and delreg = 0.9).

4.10.2 Forecasting Resource Usage Evaluation

We assessed the viability of different models on resource-constrained devices, specifically the Raspberry Pi in our case. The assessment considered the resource requirements for training the models and using them for predictions. The assessment focused on evaluating the models' performance within the resource limitations. Considering the constrained resources, we limited the training and testing data to the first 10,000 points (minutes) of the dataset. It is important to note that we maintained a 75%-25% ratio for training and testing the models. Table 4.3 presents the average training and prediction times for the different models. Among the models, the LSTM algorithm required the longest training time as a deep learning algorithm. Moreover, when dealing with larger datasets (e.g., 20,000 points), we observed the need to reduce the complexity of the LSTM model, such as by reducing the epoch value, to ensure successful training on the Raspberry Pi. The training times for the remaining models were reasonable, as they did not exceed one minute. Regarding prediction time, all models could quickly predict a single value, usually in less than one millisecond.

Table 4.2 Key Hyperparameters of Different Algorithms

Model	*Parameter*	*Value*
ARIMA	Order of differencing (d)	1
	AR order (p)	2
	MA order (q)	2
	Goodness-of-fit statistics	AIC
LSTM	Number of layers	2
	Input size	30
	Output size	1
	Units	100
	Optimizer	adam
	Batch size	64
	Epochs	50
	Loss function	MSE
SVR	Kernel function	RBF
	Regularization parameter (C)	10
	Width of the Gaussian radial basis function (γ)	0.01
	epsilon (ϵ)	0.01
	Goodness-of-fit statistics	Negative MSE
RF	Number of trees	150
	Maximum depth of each tree	10
	Random_state	42
	Goodness-of-fit statistics	Negative MSE
BN	Distribution family	Poisson
	Number of samples	30
	Prior length	6
	Random effect extension	0.9
	Discount factor for trend	0.5
	Discount factor for regression	0.9

However, it is important to consider the aspect of model retraining during the prediction operation. For example, in the case of ARIMA model, retraining is necessary for each prediction to maintain accuracy. If the model is not retrained by incorporating new values, the forecast will generate a stable value for long-term predictions. Table 4.4 demonstrates that iterative retraining (after each prediction) outperforms the model without retraining by a factor of 13 in terms of MSE metric, for example. Considering this, the effective prediction time of the ARIMA model on the

Raspberry Pi would be the sum of the prediction and retraining times, resulting in approximately 27 seconds. We also explored the accuracy-to-prediction time ratio by incorporating periodic retraining after a certain number of predictions (e.g., every 100 or 30 predictions), as shown in Table 4.4.

Furthermore, our evaluation revealed that iterative retraining could also improve the prediction accuracy of the RF and BN models. Therefore, if forecasting accuracy is a high priority, it is necessary to consider the retraining time in addition to the prediction time for these models.

Table 4.3 Training Time and Prediction Time of Different Models

Dataset volume	*Time metrics*	*ARIMA*	*LSTM*	*SVR*	*RF*	*BN*
Small	Training time	27.2963	3349.0086	34.1058	**5.6457**	14.7178
	Prediction time	0.0098	0.0093	0.0059	0.0825	9.2128×10^{-06}

Table 4.4 Retraining Necessity to Improve the Prediction of ARIMA Model

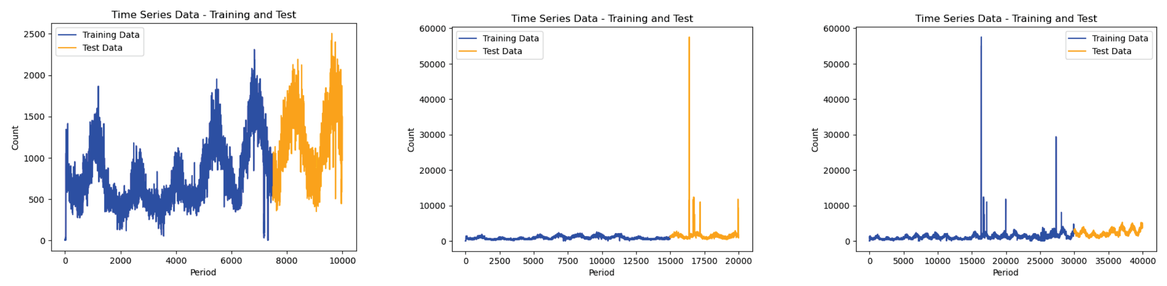
Retraining type	*MSE*	*MAE*	*RMSE*	R^2	*MAPE*
Without retraining	357651.5200	494.7020	598.0397	-1.2424	0.3680
With periodic retraining (after 100 predictions)	61597.9105	196.0479	248.1892	0.6137	0.1841
With periodic retraining (after 30 predictions)	49186.2131	172.1206	221.7796	0.6916	0.1607
Iterative training (after each prediction)	**27340.1252**	**127.5014**	**165.3484**	**0.8285**	**0.1158**

4.10.3 Forecasting Accuracy Evaluation

Table 4.5 shows the accuracy of different trained models using different data size cases: 10k, 20k, and 40k. Unexpectedly, the commonly used models, ARIMA and LSTM, do not outperform other lightweight models, except in the case of 10k, where ARIMA and LSTM have the best results, excluding the Ensemble learning model (EL). In other cases, the lightweight models, namely BN, could achieve better results. Even in the 10k case, the SVR model has a better RMSE metric value compared to LSTM. In general, we observe that the results are close, making it sensible to consider the accuracy/time ratio.

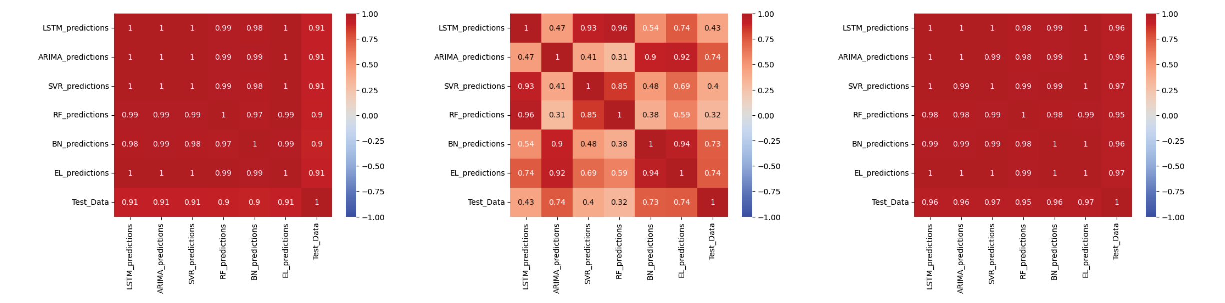
To explain the similarity in accuracy among different models, we investigated the correlation between the actual data and predicted data for different data size cases, as presented in Figure 4.13. The correlation matrices for the 10k and 40k cases (Figures 4.14a and 4.14c) clearly demonstrate high correlation, which is reflected in the accuracy results, particularly the R^2 metric shown in Table 4.5. Notably, the 40k case exhibits higher correlation than the 10k case. Thus, incorporating additional patterns in the training data, as shown in Figure 4.12c, improves the overall performance of the models. Adding data patterns to the training data can reduce or even avoid overfitting effects. Table 4.5 shows improved MAPE metric values. It is important to remember that the MAPE metric is a percentage value (between 0 and 1), allowing for comparisons between different data size cases.

In the 20k case (Figure 4.12b), which contains significant variation in the testing data, Figure 4.14b demonstrates a considerable decrease in the correlation between the actual data (Test_Data) and predicted data of different models, which significantly impacts the R^2 value. However, this decrease in correlation does not significantly affect the values of the other accuracy metrics: MSE, MAE, RMSE, and MAPE. This can be explained by the fact that the distance between the actual and predicted values plays a more important role in these accuracy metrics than correlation. For instance, a correlated pair of actual and predicted values may have a greater distance than a less correlated pair. This aspect can be further clarified with the data distribution analysis, which will be presented in the next element of the Ensemble learning evaluation.



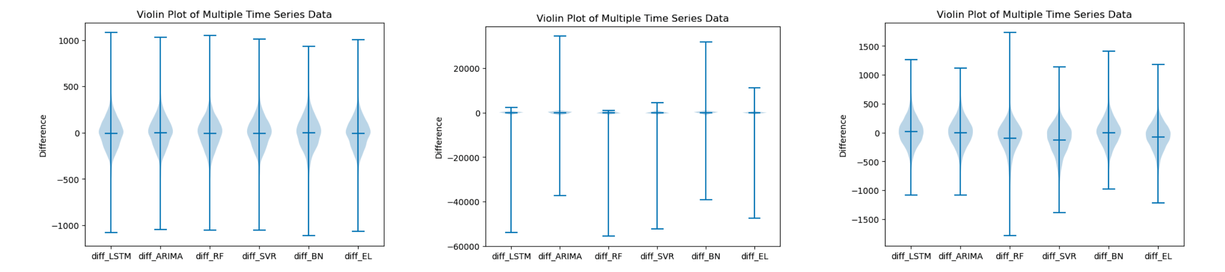
a) Small Data=10k   b) Medium Data=20k   c) Large Data=40k

Figure 4.12 Training and Testing Data of Different Data Sizes



a) Small Data=10k b) Medium Data=20k c) Large Data=40k

Figure 4.13 Correlation Matrix: Actual vs Predicted Data for Different Models and Data Sizes



a) Small Data=10k b) Medium Data=20k c) Large Data=40k

Figure 4.14 Distribution of Predicted Data: Violin Plot for Different Data Sizes

4.10.4 Evaluation of Our Ensemble Learning (EL) Model

For the evaluation of the proposed Ensemble Learning model, we consider the accuracy and time consumption of the EL model, as well as its dynamic aspect, which selects the models to consider based on the availability of resources determined by the available forecasting time.

4.10.4.1 Accuracy and Time Consumption

Table 4.5 presents the accuracy of the Ensemble Learning model (EL) in a separate line. In all cases, the EL model demonstrates good accuracy compared to the single models. Despite being

Table 4.5 Accuracy Results across Different Dataset Sizes

Data Size	*Models*	*MSE*	*MAE*	*RMSE*	R^2	*MAPE*	*Quality*
10k	ARIMA	**27435.0543**	127.6111	**165.6353**	**0.8279**	0.1159	0.884
	LSTM	29259.0771	130.5017	171.0528	0.8151	0.1145	0.8854
	SVR	27826.8364	128.2523	166.8137	0.8241	0.1154	0.8845
	RF	29067.5173	130.7628	170.4919	0.8163	0.1173	0.8826
	BN	31179.6508	136.7604	176.5776	0.8045	0.1228	0.8771
	EL	27564.5471	**127.4891**	166.0257	0.8258	**0.1143**	**0.8856**
20k	ARIMA	1469209.2302	256.1708	1212.1094	0.3876	0.1514	0.8485
	LSTM	1938833.7423	227.0645	1392.4201	0.1964	0.1273	0.8726
	SVR	2039238.156	243.6274	1428.0189	0.1548	0.1397	0.8602
	RF	2190145.9282	227.3979	1479.9141	0.0923	**0.1217**	**0.8782**
	BN	**1190752.2868**	236.1388	**1091.2159**	**0.5036**	0.1446	0.8553
	EL	1235524.0926	**219.23636**	1111.5413	0.4879	0.1285	0.8714
40k	ARIMA	58753.3252	189.5252	1212.1094	0.9243	0.0874	0.9125
	LSTM	91801.5052	231.9231	302.9876	0.8819	0.0957	0.9042
	SVR	84693.7665	219.7276	291.0219	0.8911	0.0919	0.9080
	RF	94786.2199	219.5011	307.8737	0.8781	0.0903	0.9096
	BN	**56052.0033**	**184.4365**	**236.7530**	**0.9278**	0.0849	0.9150
	EL	61987.3997	190.0018	248.9726	0.9203	**0.0829**	**0.9171**

based on the weighted average of predictions, the EL model achieves the best results for some metrics, namely MAE and MAPE. This improvement can be attributed to averaging prediction values, which helps reduce outliers (i.e., extreme differences between actual and predicted data).

Figure 4.14 employs the Violin Plot visualization technique to gain insights into the nature of these differences between actual and predicted data (i.e., predicted data - actual data) and compare their distribution characteristics across the models. In all cases, especially the 20k case, the central tendency of the differences, particularly the median, is close to zero, indicating that most actual and predicted values are similar.

In the 20k case, the narrower distribution observed for the EL model demonstrates its capability to reduce the occurrence of extreme differences compared to other models with different distribution patterns. The extent to which EL has reduced the maximum and minimum limitation values can be observed. However, for the 10k case, the EL model does not significantly improve the outlier aspect since the data distributions of the different models are similar. On the other hand, in the 40k data size case, EL significantly reduces the distribution compared to the RF model.

Regarding time consumption, there is no training time for the Ensemble Learning model based on the weighted average if the involved models are already trained. However, if the models require training, the training time in the worst case will be the sum of the training times of the involved models. The prediction time corresponds to the sum of the prediction times of the different models, along with the computing time of the weighted average, which is very small ($2.1511e - 05$). The experiments found that the average prediction time for EL is 0.1075. However, if we consider the iterative retraining of the models, the prediction time will increase significantly. This highlights the importance of considering the dynamic aspect of Ensemble Learning, as we will discuss in the next element.

4.10.4.2 Dynamic Ensemble Learning (DEL)

The evaluation of the proposed dynamic Ensemble Learning is based on the prediction time of each forecasting model discussed in Subsection 4.10.2. Specifically, we consider the utility of model retraining to improve forecasting accuracy. As a result, the prediction time can be extended for certain models, such as ARIMA (from 0.0089 to 27.3052).

Table 4.6 presents various cases based on the defined maximum prediction time (Max PT). The limitation on prediction time is used in our optimization model as a constraint (Equation 4.24) to select the models that form the Ensemble Learning model (referred to as "Used Models" in Table 4.6). The case that includes all five models (Max PT = 50) achieves the highest accuracy. However, as shown in Table 4.6, increasing the number of models does not automatically lead to improved forecasting accuracy, as observed in the cases of Max PT = 10 and Max PT = 30. Therefore, the addition of the BN model did not enhance the overall performance (ϵ_n) of the Ensemble Learning model. Finally, the "Effective PT" in Table 4.6 represents the actual time

Table 4.6 Results of Dynamic Ensemble Learning

Max PT	**Used Models**	**Eff.PT**	MSE	MAE	$RMSE$	R^2	$MAPE$
5	SVR, LSTM	0.016	28002.7269	129.2408	167.3401	0.8230	0.1177
10	SVR, LSTM, RF	5.666	27649.4064	128.2412	166.2811	0.8253	0.1164
30	SVR, LSTM, RF, BN	20.386	27668.7384	128.2982	166.3392	0.8251	0.1161
50	SVR, LSTM, RF, BN, ARIMA	47.686	**27570.1312**	**128.0019**	**166.0425**	**0.8258**	**0.1158**

consumed by the forecasting models comprising the Ensemble Learning model on the Raspberry PI.

4.10.5 Auto-scaling Performance

To evaluate the effectiveness of different forecasting algorithms for auto-scaling, we utilized the WorldCup'98 dataset as a consistent input for the auto-scaler. The auto-scaling process dynamically adjusts the number of replicas (e.g., containers) based on the incoming request volume, following the iterative phases of the MAPE-K (Monitoring, Analyzing, Planning, and Execution) loop. Our experimental evaluations comprised two categories: simulation experiments and real experiments implemented with Kubernetes.

4.10.5.1 Simulation Results

In the first category of experiments presented in Table 4.7, we evaluated various forecasting algorithms using auto-scaling evaluation metrics. The experiments involved testing datasets with 10k, 20k, and 40k data points, where each data point represented the number of requests received per minute. The maximum replica capacity was set to 500 requests per minute.

Based on the results presented in Table 4.7, the LSTM model demonstrated better overall performance (as measured by the ϵ_n value) for the 10k and 40k data point cases, despite not having the highest forecasting accuracy. This discrepancy can be attributed to the nature of the

Table 4.7 Auto-scaling Results for Different Dataset Sizes

Size	*Models*	θ_o	θ_u	T_o	T_u	ϵ_n
10k	Reactive	6.4803	4.2837	14.2162	14.2162	1.0000
	LSTM	7.7395	**2.5344**	15.8299	**9.2712**	**1.1790**
	ARIMA	6.6869	3.1248	13.2388	11.3765	1.1530
	SVR	**6.3022**	3.1821	**12.5506**	11.4574	1.1784
	RF	6.4372	3.2557	12.6720	11.7813	1.1547
	BN	6.9062	3.6686	14.4129	12.672	1.0470
	EL	6.7105	3.1167	13.2793	11.2550	1.1549
20k	Reactive	7.2345	4.3284	14.9325	**14.7112**	1.0000
	LSTM	5.4080	4.4877	11.5291	16.0362	1.1127
	ARIMA	7.2800	4.6590	14.5271	15.2716	0.9777
	SVR	6.6274	4.4410	13.4406	15.7746	1.0246
	RF	**4.5372**	4.9730	**10.1207**	17.5251	**1.1450**
	BN	7.1517	4.2633	14.2253	14.8692	1.0162
	EL	5.8294	**4.1751**	12.3340	14.7283	1.1168
40k	Reactive	4.4953	3.5546	18.9788	18.9888	1.0000
	LSTM	**1.6531**	6.8376	**5.5767**	37.9438	**1.2456**
	ARIMA	4.4533	3.4629	18.7061	18.6659	1.0169
	SVR	2.8295	5.4982	9.1374	32.4172	1.0573
	RF	3.2776	4.7051	12.2668	26.4292	1.0359
	BN	4.2743	**3.4211**	18.024	**18.5255**	1.0421
	EL	2.9466	4.4765	10.8224	25.8375	1.1178

auto-scaling process, which focuses on the difference between demanded and supplied replicas rather than relying on the difference between actual and predicted workload data. Thus, even if the difference between actual and predicted data is within the capacity limits of the replicas, no adjustment in the number of replicas is necessary.

In all cases, there was consistently a lightweight algorithm that performed well in either over-provisioning or under-provisioning scenarios, as well as in the overall performance metric (ϵ_n). Notably, the RF model yielded the best results for the 20k dataset.

Regarding stability, the Ensemble Learning (EL) model achieved good performance, consistently ranking around the 2nd position across all cases. This stability in result quality makes the EL model particularly appealing, as it outperforms other models that may excel in one case but perform poorly in others. For example, the RF model ranked first for the 20k case but third for the remaining two cases.

4.10.5.2 Experiments on Kubernetes

In the experiments conducted on Kubernetes, we utilized the Nginx service using its DockerHub image, a widely used web server and proxy server software. The following parameters were set for the experiments:

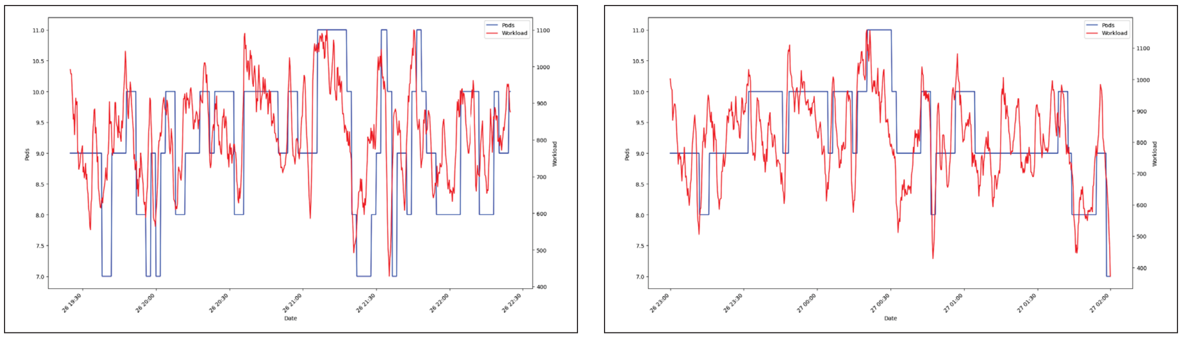
- Minimum number of pods: 1
- Maximum number of pods: 100
- Workload per replica (maximum capacity): 100

To mitigate the impact of oscillations, the PRR (Pods Removal Rate) parameter was set to 0.9. This parameter determines the proportion of pods to be removed when there is a decrease in the number of HTTP requests. By setting PRR to 0.9, the system is designed to remove a significant fraction of pods in each iteration, gradually adjusting in response to diminishing workload.

To perform the experiments, HTTP requests were sent to an application deployed on Kubernetes. The Nginx service was deployed in Kubernetes's "default" namespace and exposed on a specific port, 30651. Jmeter was configured to send requests corresponding to the WorldCup'98 dataset periodically. The requests were sent over 3 hours.

Figure 4.15a illustrates the response of the reactive autoscaler to changes in workload. It shows the number of pods (blue line) in relation to the number of HTTP requests received by the Nginx pods (red line). The reactive autoscaler often exhibits a delay before adjusting the number of pods in response to workload changes.

On the other hand, Figure 4.15b demonstrates the changes applied to the number of pods by the proactive autoscaler based on the predicted future workload. In this experiment, the LSTM model was used for prediction, as it had the best value of the MAPE metric and the best overall auto-scaling performance in the simulation with the 10k data size. The proactive autoscaler



a) Reactive Approach b) Proactive Approach

Figure 4.15 Dynamic Pod Auto-scaling Based on Workload

frequently adapts the number of pods in advance to workload changes, leveraging the forecasting model. Overall, the proactive autoscaler demonstrates more proactive behavior in adjusting the number of pods, while the reactive autoscaler tends to have a delay in response.

4.11 Discussion

Based on the evaluation results, the discussion section presents our observations and findings related to the research questions in the Introduction section. The following key points emerge from our analysis:

- *The training of LSTM model, as a widely used deep-learning technique for time series forecasting, is unsuitable for resource-constrained devices such as Raspberry Pi. In contrast, ARIMA, a statistical technique, can be trained reasonably for limited data sizes (e.g., 10k).* As highlighted by research question **RQ1**, we evaluated the performance of LSTM and ARIMA as widely adopted techniques for time series forecasting. In the context of resource-constrained devices, our evaluation revealed that training an LSTM model is not practical, even with small data sizes. For instance, when using a data size of 10k, LSTM training on a Raspberry Pi took approximately 3349 seconds. Furthermore, attempting to increase the data size (e.g., 20k) proved challenging, as the device struggled to support the training process unless we significantly reduced the model's complexity (e.g., by minimizing the epoch value), which can impact the model's accuracy.

Therefore, it is advisable to train LSTM models on more powerful devices, whether in Edge, Fog or even the cloud, with the possibility of utilizing techniques such as federated learning. In contrast, the prediction time of the LSTM model was sufficiently fast and suitable for resource-constrained devices.

On the other hand, the statistical technique ARIMA exhibited reasonable training times (around 27 seconds) on a Raspberry Pi for small datasets (e.g., 10k). However, it is important to note that the ARIMA model needs to be retrained after each prediction to maintain its forecasting accuracy, which can be time-consuming on resource-constrained devices. Moreover, if a large volume of data is involved, training ARIMA on more powerful devices is more practical.

Regarding forecasting accuracy, our evaluation indicates that the accuracy of these models is relatively close regardless of the dataset size. Furthermore, our experiments demonstrate that these heavy techniques, such as LSTM and ARIMA, perform well when the current data patterns are similar to the training data patterns (as observed in Figure 4.12a). However, in cases where significant differences exist between the patterns, alternative lighter techniques such as Support Vector Regression (SVR), Random Forest (RF), and Bayesian Networks (BN) or their combinations could be considered, as we will discuss in the following points.

- *Alternative techniques that are less resource-intensive, such as Support Vector Regression (SVR), Random Forest (RF), and Bayesian Networks (BN), demonstrate comparable accuracy and overall performance to widely used techniques like ARIMA and LSTM.*

  The exploration of alternative models is motivated by the need to find lighter models that require fewer resources, particularly in terms of training time. The evaluation assesses the suitability of these models for resource-limited devices, taking into account the iterative retraining required in the forecasting operation.

  In terms of time consumption, Random Forest (RF) proves to be the lightest model among the evaluated techniques while also achieving comparable or even superior accuracy compared to Support Vector Regression (SVR) and Bayesian Networks (BN).

  Against our expectations, the accuracy of these alternative models is comparable to widely used techniques like ARIMA and LSTM, which are more resource-intensive. Consequently, this observation emphasizes the importance of considering the performance-to-resource usage ratio when selecting a forecasting model. It encourages further exploration and consideration of these alternative techniques, particularly in resource-constrained scenarios where efficient resource utilization is crucial.

- *Our dynamic and weighted-average based Ensemble learning maintains a high level of accuracy and stable performance compared to individual models, primarily due to its ability to effectively reduce outliers, representing extreme differences between actual and predicted workload data.*

  Our Dynamic Ensemble learning model (DEL) considers both accuracy and computational time during the model selection process, ensuring compatibility with available resources while aiming to improve prediction accuracy.

  Regarding training time, the evaluation demonstrates that the Ensemble learning approach is lightweight when utilizing pre-trained models. However, if new models are included, they would need to be trained beforehand to be incorporated into the Ensemble learning model.

  Our Ensemble learning approach consistently achieves high accuracy in most cases, particularly when evaluated using the Quality metric (i.e., MAPE). In the remaining cases, it maintains a commendable rank, typically around 2nd place, compared to individual models. This can be attributed to the Ensemble learning's ability to reduce the presence of outliers, which are extreme differences between the actual and predicted data. These outliers can negatively impact the performance of auto-scaling. Furthermore, the evaluation reveals that the accuracy of the Ensemble learning model is not solely dependent on the number of included models but also on their quality. Consequently, adding a new model to the Ensemble learning may decrease the overall accuracy and performance in certain cases.
- *The evaluation highlights the significant performance improvement achieved by adopting proactive auto-scaling, leveraging workload forecasting, compared to a reactive approach.*

  Our evaluation highlights the significant impact of adopting a proactive approach, known as proactive auto-scaling, on the performance of auto-scaling systems. Proactive auto-scaling leverages workload forecasting to anticipate future demands and enables the system to adapt before the new workload arrives.

In our evaluation, we compared the performance of a reactive model, which reacts to changes in workload without forecasting, to a proactive model that utilizes LSTM, the best forecasting model based on the MAPE metric. The results demonstrate a substantial improvement in the proactive model compared to the reactive model. It is important to note that a higher forecasting accuracy does not necessarily guarantee better auto-scaling performance. Our analysis indicates that minor differences between the actual and predicted data, below the capacity of the replicas or containers, may have minimal or no effect on the auto-scaling performance. In addition, other elements can also impact the auto-scaling performance, such as the oscillation of scaling actions.

4.12 Conclusion

Edge computing has brought notable benefits to IoT networks by offloading tasks from the cloud, reducing network workload, and enhancing system responsiveness. However, operating within a resource-constrained and heterogeneous environment presents challenges in service deployment and resource management. To address these issues, this paper proposes a proactive approach that predicts future workload to adjust the number of deployed service replicas dynamically. A research methodology is followed to select an appropriate forecasting algorithm capable of effectively handling auto-scaling challenges in Edge environments.

The study conducted in this paper includes experiments and discussions that address research questions concerning the suitability of widely used forecasting algorithms, the exploration of lightweight alternatives, the efficacy of Ensemble learning, and the impact of predictions on the auto-scaling service process.

Future work involves conducting additional experiments to further evaluate the presented algorithms, considering optimization techniques for hyper-parameter selection, and incorporating additional performance metrics such as CPU and memory usage. Additionally, exploring techniques within Ensemble learning, such as blending, is planned to improve prediction accuracy. Furthermore, investigating alternative techniques to address the optimization problem presented in this work, aiming to optimize the trade-off between prediction quality and resource usage in the model selection process of the proposed Dynamic Ensemble learning, is part of the future research direction.

CONCLUSION AND RECOMMENDATIONS

The deployment of Internet of Things (IoT) services at the network edge through edge computing has become increasingly popular due to its cost-effectiveness and improved responsiveness. However, the resource constraints and heterogeneity of edge devices in IoT networks present significant challenges for efficient service deployment and resource utilization. This study proposes using lightweight virtualization technologies, specifically containers, to enable the auto-scaling of services on edge devices. Furthermore, our work investigates the optimization of shared resource management on edge devices, including the integration of machine learning techniques. Enhancing service performance and resource utilization improves IoT services' overall efficiency and responsiveness at the Edge.

Consequently, this research aims to improve the deployment of container-based IoT services on edge devices within the same cluster. Specifically, this work aims to address the limitations of existing approaches and improve the effectiveness and efficiency of deploying containerized services at the Edge. Our contributions include developing resource-efficient solutions that can handle dynamic workloads and ensure the quality of service, automatic processing for maintaining the system in a suitable state, investigating and improving forecasting techniques to enhance workload forecasting accuracy, and addressing oscillation mitigation issues to ensure the stable and efficient operation of containerized services.

Our first contribution automates the deployment process for service auto-scaling and fair distribution of containerized services on devices in the same edge cluster using the MAPE-K loop framework and our deployment model and rule model as shared knowledge. This enables efficient scaling through automatic threshold determination and evaluation based on system performance criteria. Our evaluation shows the effectiveness of this approach in adapting system performance by changing the number of replicas to meet service performance requirements and the availability of system resources.

Our second contribution proposes an improved proactive service auto-scaling approach that addresses the research gap in workload forecasting accuracy and oscillation issues. We add data featurization inspired by Japanese candlesticks to improve forecasting accuracy and propose a grid-based oscillation mitigation approach that uses the same features. Our evaluation shows considerable improvement in forecasting accuracy and outperformance of state-of-the-art techniques in combination with a cool-down strategy, which is a commonly used strategy.

Our third contribution proposes a methodology to address the complexity of designing an efficient auto-scaler for containerized services on the Edge, which is impacted by dynamic workload characteristics and resource limitations. We investigate the accuracy and convenience of common time series data analysis and forecasting techniques, evaluate less resource-intensive techniques, and explore the use of Ensemble learning to improve prediction accuracy while considering the performance/resource usage ratio. We formulate the optimization problem of selecting the appropriate forecasting algorithms as a Knapsack problem and propose a heuristic technique to reduce the solution complexity. The experiments demonstrate that the proposed Dynamic and weighted-average Ensemble learning approach maintains high forecasting accuracy and auto-scaling performance in resource-constrained context.

This thesis has presented novel solutions to significant challenges in the field of containerized service deployment at Edge computing. We believe that our contributions have the potential to significantly advance the state of the art and open up new avenues of research. However, there is still much work to be done to fully address the complex and evolving challenges in this field. As such, we propose several avenues for future research that build upon the work presented in this thesis:

Improving the use of our deployment and rule model: This improvement focuses on the deployment knowledge component. It addresses the issue of the many existing Description Specific Languages (DSLs) that describes the deployment for a specific tool. We plan to model our proposed IoT service deployment model using semantic techniques as we have previous work in this domain (e.g., Bali, Al-Osta & Abdelouahed (2017)). Semantic Web techniques can bring various advantages, such as automatic instantiation and verification of the deployment models. By using ontologies, we can represent the deployment knowledge in a machine-understandable format and enable automated reasoning to validate the consistency of the deployment model. This will allow for more efficient and reliable deployment of IoT services, as well as easier integration with other systems.

Exploring other feature extraction techniques: While our featurization method based on Japanese candlesticks has shown promising results, other feature extraction techniques may further improve forecasting accuracy. We plan to investigate other trading techniques (i.e., indicators), such as Relative Strength Index (RSI) and Moving average convergence/divergence (MACD), which can provide more rich information on the time series window.

Enhancing LSTM architecture: One area for future improvement is the architecture of our LSTM model. Specifically, we plan to explore ways to better incorporate the featurization principle into the LSTM model or add attention mechanisms to give more weight to important features.

Optimizing oscillation mitigation: Another avenue for future work is to optimize further the combination of our grid-based oscillation mitigation approach and the commonly used cool-down technique (CDT). One potential approach is to apply optimization techniques, such as genetic algorithms or particle swarm optimization, to find the optimal values of the grid parameters and CDT cool-down period for different workload characteristics and system configurations.

Improving the Planning phase of the MAPE-K loop: In our work, we generate the plan of auto-scaling actions after deducing the predicted workload in the previous phase, the Analysis phase. However, to further enhance the generated plan, we plan to explore more advanced techniques, particularly reinforcement learning. This will involve developing and training reinforcement learning models to learn the optimal actions to take in response to predicted workloads.

Optimizing the accuracy of our workload forecasting methodology: We aim to investigate other optimization techniques for (hyper)parameter selection and incorporate more performance metrics such as CPU and memory usage to comprehensively understand our approach's performance. Moreover, we will explore other techniques used in Ensemble learning, such as blending, to improve the prediction accuracy and assess their effectiveness compared to the existing techniques. Additionally, we plan to investigate other approaches to address the optimization problem presented in our work, which aims to optimize the prediction quality/resource usage ratio in the model selection of the Ensemble learning.

Extending to other auto-scaling scenarios: To gain a more comprehensive understanding of our approach's performance, we plan to experiment with large-scale industry use cases with different hardware platforms. This will involve collecting and analyzing data from actual usage scenarios to assess the accuracy and efficiency of our approach under varying conditions. In this scope, we aim to investigate how our approach can be extended to other auto-scaling scenarios beyond containerized services on the Edge. For example, investigating multi-level clusters and managing resources in hybrid cloud-edge environments. By applying our approach to different scenarios, we can assess its versatility and potential for wider adoption, leading to the development of more efficient and effective auto-scaling systems that can meet the demands of modern computing environments.

Implementing a fully built auto-scaler: Lastly, we aim to further enhance the practical applicability of the research findings by developing a fully built auto-scaler solution that can be readily used in practice. This solution should incorporate the proposed deployment and rule models, the improved proactive auto-scaling approach, and the optimized oscillation mitigation techniques.

Overall, by implementing a fully built auto-scaler solution, conducting real-world validations, and incorporating machine learning algorithms, the research outcomes can have a direct impact on the industry, facilitating the deployment of efficient and effective containerized services at the edge, and paving the way for advancements in edge computing and IoT deployments.

Finally, as a memorable statement: our work is a vital step towards making auto-scaling containerized services on the IoT Edge more efficient and effective. By pushing the boundaries of what is possible, we may unlock the full potential of this technology, paving the way for a brighter and more connected future. Let us strive towards a world where the IoT empowers us to work smarter, faster, and more innovatively than ever.

BIBLIOGRAPHY

Ahmed, B., Seghir, B., Al-Osta, M. & Abdelouahed, G. (2019). Container based resource management for data processing on IoT gateways. *Procedia Computer Science*, 155, 234–241.

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N. & Merle, P. (2017). Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on Services Computing*, 11(2), 430–447.

Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M. & Ayyash, M. (2015). Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE communications surveys & tutorials*, 17(4), 2347–2376.

Al-Osta, M., Bali, A. & Gherbi, A. (2019). Event driven and semantic based approach for data processing on IoT gateway devices. *Journal of Ambient Intelligence and Humanized Computing*, 10(12), 4663–4678.

Arcaini, P., Riccobene, E. & Scandurra, P. (2015). Modeling and analyzing MAPE-K feedback loops for self-adaptation. *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 13–23.

Arce, J. M. M. & Macabebe, E. Q. B. (2019). Real-Time Power Consumption Monitoring and Forecasting Using Regression Techniques and Machine Learning Algorithms. *2019 IEEE International Conference on Internet of Things and Intelligence System (IoTaIS)*, pp. 135–140.

Arlitt, M. & Jin, T. (2000). A workload characterization study of the 1998 world cup website. *IEEE network*, 14(3), 30–37.

Ashok, I. & Christine, O. [https://www.ibm.com/cloud/architecture/architectures/edge-computing/]. (2023). Edge Computing Architecture.

Baeldung. [https://www.baeldung.com/cs/virtualization-vs-containerization]. (Accessed 2023). Virtualization vs Containerization.

Bali, A., Al-Osta, M. & Abdelouahed, G. (2017). An ontology-based approach for IoT data processing using semantic rules. *SDL 2017: Model-Driven Engineering for Future Internet: 18th International SDL Forum, Budapest, Hungary, October 9–11, 2017, Proceedings 18*, pp. 61–79.

Bali, A., Al-Osta, M., Ben Dahsen, S. & Gherbi, A. (2020). Rule based auto-scalability of IoT services for efficient edge device resource utilization. *Journal of Ambient Intelligence and Humanized Computing*, 1–18.

Bauer, A., Grohmann, J., Herbst, N. & Kounev, S. (2018a). On the value of service demand estimation for auto-scaling. *International Conference on Measurement, Modelling and Evaluation of Computing Systems*, pp. 142–156.

Bauer, A., Herbst, N., Spinner, S., Ali-Eldin, A. & Kounev, S. (2018b). Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field. *IEEE Transactions on Parallel and Distributed Systems*, 30(4), 800–813.

Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes. *IEEE cloud computing*, 1(3), 81–84.

Box, G. E., Jenkins, G. M., Reinsel, G. C. & Ljung, G. M. (2015). *Time series analysis: forecasting and control*. John Wiley & Sons.

Brogi, A., Mencagli, G., Neri, D., Soldani, J. & Torquati, M. (2017). Container-Based Support for Autonomic Data Stream Processing Through the Fog. *European Conference on Parallel Processing*, pp. 17–28.

Brownlee, J. (2016). *Machine learning mastery with Python: understand your data, create accurate models, and work projects end-to-end*. Machine Learning Mastery.

cAdvisor, G. [https://github.com/google/cadvisor]. (2022). cAdvisor Open source.

Calheiros, R. N., Masoumi, E., Ranjan, R. & Buyya, R. (2014). Workload prediction using ARIMA model and its impact on cloud applications' QoS. *IEEE transactions on cloud computing*, 3(4), 449–458.

Cao, J., Fu, J., Li, M. & Chen, J. (2014). CPU load prediction for cloud environment based on a dynamic ensemble model. *Software: Practice and Experience*, 44(7), 793–804.

Cardenas, Y. M. R. (2018). Scaling policies derivation for predictive autoscaling of cloud applications.

Cetinski, K. & Juric, M. B. (2015). AME-WPC: Advanced model for efficient workload prediction in the cloud. *Journal of Network and Computer Applications*, 55, 191–201.

Chakraborty, K., Mehrotra, K., Mohan, C. K. & Ranka, S. (1992). Forecasting the behavior of multivariate time series using neural networks. *Neural networks*, 5(6), 961–970.

Chen, M., Yuan, J., Liu, D. & Li, T. (2020). An adaption scheduling based on dynamic weighted random forests for load demand forecasting. *The Journal of Supercomputing*, 76, 1735–1753.

Chicco, D., Warrens, M. J. & Jurman, G. (2021). The coefficient of determination R-squared is more informative than SMAPE, MAE, MAPE, MSE and RMSE in regression analysis evaluation. *PeerJ Computer Science*, 7, e623.

Ciptaningtyas, H. T., Santoso, B. J. & Razi, M. F. (2017). Resource elasticity controller for Docker-based web applications. *2017 11th International Conference on Information & Communication Technology and System (ICTS)*, pp. 193–196.

Computing, A. et al. (2006). An architectural blueprint for autonomic computing. *IBM White Paper*, 31(2006), 1–6.

Cui, L., Yang, S., Chen, F., Ming, Z., Lu, N. & Qin, J. (2018). A survey on application of machine learning for Internet of Things. *International Journal of Machine Learning and Cybernetics*, 9(8), 1399–1417.

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

Devarajan, M., Subramaniyaswamy, V., Vijayakumar, V. & Ravi, L. (2019). Fog-assisted personalized healthcare-support system for remote patients with diabetes. *Journal of Ambient Intelligence and Humanized Computing*, 1–14.

Di, S., Kondo, D. & Cirne, W. (2012). Host load prediction in a Google compute cloud with a Bayesian model. *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11.

Dickey, D. A. & Fuller, W. A. (1979). Distribution of the estimators for autoregressive time series with a unit root. *Journal of the American statistical association*, 74(366a), 427–431.

Dietrich, B., Nunna, S., Goswami, D., Chakraborty, S. & Gries, M. (2010). LMS-based low-complexity game workload prediction for DVFS. *2010 IEEE International Conference on Computer Design*, pp. 417–424.

Doan, D. N., Zaharie, D. & Petcu, D. (2019). Auto-scaling for a streaming architecture with fuzzy deep reinforcement learning. *European Conference on Parallel Processing*, pp. 476–488.

Dunford, R., Su, Q. & Tamang, E. (2014). The pareto principle.

Elrotub, M., Bali, A. & Gherbi, A. (2021). Sharing VM resources with using prediction of future user requests for an efficient load balancing in cloud computing environment. *International Journal of Software Science and Computational Intelligence (IJSSCI)*, 13(2), 37–64.

Evans, D. (2011). The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, 1(2011), 1–11.

Fernandez, H., Pierre, G. & Kielmann, T. (2014). Autoscaling web applications in heterogeneous cloud infrastructures. *2014 IEEE international conference on cloud engineering*, pp. 195–204.

Gao, J., Wang, H. & Shen, H. (2020). Machine Learning Based Workload Prediction in Cloud Computing. *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, pp. 1–9.

Goli, A., Mahmoudi, N., Khazaei, H. & Ardakanian, O. (2021). A Holistic Machine Learning-based Autoscaling Approach for Microservice Applications. *CLOSER*, pp. 190–198.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R. & Schmidhuber, J. (2017). LSTM: A Search Space Odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10), 2222–2232.

Gubbi, J., Buyya, R., Marusic, S. & Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7), 1645–1660.

Gupta, V., Kaur, K. & Kaur, S. (2017). Performance comparison between light weight virtualization using docker and heavy weight virtualization. *International Journal of Advanced Technology in Engineering and Science, Volume*, 1(05), 509–514.

Hat, R. [https://www.redhat.com/en/topics/virtualization]. (Accessed 2023). Virtualization: What it is and why it matters.

Herbst, N., Krebs, R., Oikonomou, G., Kousiouris, G., Evangelinou, A., Iosup, A. & Kounev, S. (2016). Ready for rain? A view from SPEC research on the future of cloud metrics. *arXiv preprint arXiv:1604.03470.*

Hu, Y., Huber, A., Anumula, J. & Liu, S.-C. (2018). Overcoming the vanishing gradient problem in plain recurrent networks. *arXiv preprint arXiv:1801.06105.*

Hyndman, R. J. & Athanasopoulos, G. (2018). *Forecasting: principles and practice*. OTexts.

Hyndman, R. J., Wang, E. & Laptev, N. (2015). Large-scale unusual time series detection. *2015 IEEE international conference on data mining workshop (ICDMW)*, pp. 1616–1619.

Imdoukh, M., Ahmad, I. & Alfailakawi, M. G. (2019). Machine learning-based auto-scaling for containerized applications. *Neural Computing and Applications*, 1–16.

Iram, T., Shamsi, J., Alvi, U., ur Rahman, S. & Maaz, M. (2019). Controlling Smart-City Traffic using Machine Learning. *2019 International Conference on Frontiers of Information Technology (FIT)*, pp. 203–2035.

Ismail, B. I., Goortani, E. M., Ab Karim, M. B., Tat, W. M., Setapa, S., Luke, J. Y. & Hoe, O. H. (2015). Evaluation of docker as edge computing platform. *2015 IEEE Conference on Open Systems (ICOS)*, pp. 130–135.

Jagannath, J., Polosky, N., Jagannath, A., Restuccia, F. & Melodia, T. (2019). Machine learning for wireless communications in the Internet of Things: A comprehensive survey. *Ad Hoc Networks*, 93, 101913.

Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J. & Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3), 24–35.

Jiang, Y., Perng, C.-S., Li, T. & Chang, R. N. (2013). Cloud analytics for capacity planning and instant VM provisioning. *IEEE Transactions on Network and Service Management*, 10(3), 312–325.

Kan, C. (2016). DoCloud: An elastic cloud platform for Web applications based on Docker. *2016 18th international conference on advanced communication technology (ICACT)*, pp. 478–483.

Kao, C.-C., Chang, C.-W., Cho, C.-P. & Shun, J.-Y. (2020). Deep learning and ensemble learning for traffic load prediction in real network. *2020 IEEE Eurasia Conference on IOT, Communication and Engineering (ECICE)*, pp. 36–39.

Karp, R. M. (1975). On the computational complexity of combinatorial problems. *Networks*, 5(1), 45–68.

Kellerer, H., Pferschy, U., Pisinger, D., Kellerer, H., Pferschy, U. & Pisinger, D. (2004). *Multidimensional knapsack problems*. Springer.

Khan, W. Z., Ahmed, E., Hakak, S., Yaqoob, I. & Ahmed, A. (2019). Edge computing: A survey. *Future Generation Computer Systems*, 97, 219–235.

Khazaei, H., Bannazadeh, H. & Leon-Garcia, A. (2017). SAVI-IoT: A self-managing containerized IoT platform. *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 227–234.

Klinaku, F., Frank, M. & Becker, S. (2018). CAUS: an elasticity controller for a containerized microservice. *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 93–98.

Kovács, J. (2019). Supporting programmable autoscaling rules for containers and virtual machines on clouds. *Journal of Grid Computing*, 17(4), 813–829.

kubernetes. [Accessed: 2022-11-30]. (2022a). Production-grade container orchestration. Retrieved from: https://kubernetes.io/.

kubernetes. [https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/]. (2022b). What is Kubernetes.

Kubernetes. [https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/]. (Accessed 2023a). Horizontal Pod Autoscaling.

Kubernetes. [https://kubernetes.io/docs/concepts/overview/]. (Accessed 2023b). Kubernetes Overview.

Kumar, J., Saxena, D., Singh, A. K. & Mohan, A. (2020). BiPhase adaptive learning-based neural network model for cloud datacenter workload forecasting. *Soft Computing*, 24, 14593–14610.

Kumar, J., Singh, A. K. & Buyya, R. (2021a). Self directed learning based workload forecasting model for cloud resource management. *Information Sciences*, 543, 345–366.

Kumar, K. et al. (2021b). Forecasting of cloud computing services workload using machine learning. *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, 12(11), 4841–4846.

Kumar, P. M. & Gandhi, U. D. (2018). A novel three-tier Internet of Things architecture with machine learning algorithm for early detection of heart diseases. *Computers & Electrical Engineering*, 65, 222–235.

Laptev, N., Yosinski, J., Li, L. E. & Smyl, S. (2017). Time-series extreme event forecasting with neural networks at uber. *International conference on machine learning*, 34, 1–5.

Li, Y. & Xia, Y. (2016). Auto-scaling web applications in hybrid cloud based on docker. *2016 5th International conference on computer science and network technology (ICCSNT)*, pp. 75–79.

Lim, W. Y. B., Luong, N. C., Hoang, D. T., Jiao, Y., Liang, Y.-C., Yang, Q., Niyato, D. & Miao, C. (2020). Federated learning in mobile edge networks: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 22(3), 2031–2063.

Liu, C., Shang, Y., Duan, L., Chen, S., Liu, C. & Chen, J. (2015). Optimizing workload category for adaptive workload prediction in service clouds. *Service-Oriented Computing: 13th International Conference, ICSOC 2015, Goa, India, November 16-19, 2015, Proceedings 13*, pp. 87–104.

Lorido-Botran, T., Miguel-Alonso, J. & Lozano, J. A. (2014). A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4), 559–592.

Lu, Y., Panneerselvam, J., Liu, L. & Wu, Y. (2016). RVLBPNN: A workload forecasting model for smart cloud computing. *Scientific Programming*, 2016.

Mashal, I., Alsaryrah, O., Chung, T.-Y., Yang, C.-Z., Kuo, W.-H. & Agrawal, D. P. (2015). Choices for interaction with things on Internet and underlying issues. *Ad Hoc Networks*, 28, 68–90.

Meng, Y., Rao, R., Zhang, X. & Hong, P. (2016). CRUPA: A container resource utilization prediction algorithm for auto-scaling based on time series analysis. *2016 International conference on progress in informatics and computing (PIC)*, pp. 468–472.

Merenda, M., Porcaro, C. & Iero, D. (2020). Edge Machine Learning for AI-Enabled IoT Devices: A Review. *Sensors*, 20(9), 2533.

Messias, V. R., Estrella, J. C., Ehlers, R., Santana, M. J., Santana, R. C. & Reiff-Marganiec, S. (2016). Combining time series prediction models using genetic algorithm to autoscaling web applications hosted in the cloud infrastructure. *Neural Computing and Applications*, 27, 2383–2406.

Miller, T. W. (2015). *Forecasting Time Series Data with ARIMA*. Pennsylvania State University.

Miller, T. W. (2016). *Forecasting Time Series Data with ARIMA*. Pennsylvania State University.

Mishra, S. K., Sahoo, B. & Parida, P. P. (2020). Load balancing in cloud computing: a big picture. *Journal of King Saud University-Computer and Information Sciences*, 32(2), 149–158.

Mohammadi, K., Frick, R. & Vouros, G. A. (2019). Time series forecasting using Bayesian networks: A survey and systematic review. *Expert Systems with Applications*, 137, 286–305.

Morabito, R. (2017). Virtualization on internet of things edge devices with container technologies: a performance evaluation. *IEEE Access*, 5, 8835–8850.

Morabito, R. & Beijar, N. (2016). Enabling data processing at the network edge through lightweight virtualization technologies. *2016 IEEE International Conference on Sensing, Communication and Networking (SECON Workshops)*, pp. 1–6.

Morabito, R., Petrolo, R., Loscrì, V., Mitton, N., Ruggeri, G. & Molinaro, A. (2017). Lightweight virtualization as enabling technology for future smart cars. *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 1238–1245.

Murshed, M., Murphy, C., Hou, D., Khan, N., Ananthanarayanan, G. & Hussain, F. (2019). Machine learning at the network edge: A survey. *arXiv preprint arXiv:1908.00080*.

NetApp. [https://www.netapp.com/devops-solutions/what-are-containers/]. (Accessed 2023). What are Containers?

Nguyen, H., Kieu, L.-M., Wen, T. & Cai, C. (2018). Deep learning methods in transportation domain: a review. *IET Intelligent Transport Systems*, 12(9), 998–1004.

Nguyen, T.-T., Yeom, Y.-J., Kim, T., Park, D.-H. & Kim, S. (2020). Horizontal pod autoscaling in Kubernetes for elastic container orchestration. *Sensors*, 20(16), 4621.

Ogunmolu, O., Gu, X., Jiang, S. & Gans, N. (2016). Nonlinear systems identification using deep dynamic neural networks. *arXiv preprint arXiv:1610.01439*.

Pahl, C., Brogi, A., Soldani, J. & Jamshidi, P. (2017). Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing*, 7(3), 677–692.

Poongodi, T., Rathee, A., Indrakumari, R. & Suresh, P. (2020). IoT Sensing Capabilities: Sensor Deployment and Node Discovery, Wearable Sensors, Wireless Body Area Network (WBAN), Data Acquisition. In *Principles of Internet of Things (IoT) Ecosystem: Insight Paradigm* (pp. 127–151). Springer.

Prachitmutita, I., Aittinonmongkol, W., Pojjanasuksakul, N., Supattatham, M. & Padungweang, P. (2018). Auto-scaling microservices on IaaS under SLA with cost-effective framework. *2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)*, pp. 583–588.

Pratama, M. F. & Yang, B. S. (2019). Random forest for time series forecasting: A review. *Expert Systems with Applications*, 116, 456–472.

Qu, C., Calheiros, R. N. & Buyya, R. (2018). Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR)*, 51(4), 1–33.

Raghunath, B. R. & Annappa, B. (2015). Virtual machine migration triggering using application workload prediction. *Procedia Computer Science*, 54, 167–176.

Rahmanian, A. A., Ghobaei-Arani, M. & Tofighy, S. (2018). A learning automata-based ensemble resource usage prediction algorithm for cloud computing environment. *Future Generation Computer Systems*, 79, 54–71.

Renner, T., Meldau, M. & Kliem, A. (2016). Towards container-based resource management for the internet of things. *2016 International Conference on Software Networking (ICSN)*, pp. 1–5.

Roy, N., Dubey, A. & Gokhale, A. (2011). Efficient autoscaling in the cloud using predictive models for workload forecasting. *2011 IEEE 4th International Conference on Cloud Computing*, pp. 500–507.

Ruchika, V. (2016). Evaluation of Docker for IoT Application. *International Journal on Recent and Innovation Trends in Computing and Communication*, 4(6), 624–628.

Samie, F., Bauer, L. & Henkel, J. (2019). From cloud down to things: An overview of machine learning in internet of things. *IEEE Internet of Things Journal*, 6(3), 4921–4934.

Sangpetch, A., Sangpetch, O., Juangmarisakul, N. & Warodom, S. (2017). Thoth: Automatic Resource Management with Machine Learning for Container-based Cloud Platform. *CLOSER*, pp. 75–83.

Saxena, D. & Singh, A. K. (2022). Auto-adaptive learning-based workload forecasting in dynamic cloud environment. *International Journal of Computers and Applications*, 44, 541–551.

Saxena, D. & Singh, A. K. (2021). A proactive autoscaling and energy-efficient VM allocation framework using online multi-resource neural network for cloud data center. *Neurocomputing*, 426, 248–264.

Schumer, M. & Steiglitz, K. (1968). Adaptive step size random search. *IEEE Transactions on Automatic Control*, 13(3), 270–276.

Seni, G. & Elder, J. F. (2010). Ensemble methods in data mining: improving accuracy through combining predictions. *Synthesis lectures on data mining and knowledge discovery*, 2(1), 1–126.

Services", A. W. [https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-auto-scaling.html]. (Accessed 2023). Service Auto Scaling.

Shariffdeen, R., Munasinghe, D., Bhathiya, H., Bandara, U. & Bandara, H. D. (2016). Adaptive workload prediction for proactive auto scaling in PaaS systems. *2016 2nd International Conference on Cloud Computing Technologies and Applications (CloudTech)*, pp. 22–29.

Shumway, R. H. & Stoffer, D. S. (2017). *Time series analysis and its applications: with R examples*. Springer.

Singh, N. & Rao, S. (2014). Ensemble learning for large-scale workload prediction. *IEEE Transactions on Emerging Topics in Computing*, 2(2), 149–165.

Sommer, M., Klink, M., Tomforde, S. & Hähner, J. (2016). Predictive load balancing in cloud computing environments based on ensemble forecasting. *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 300–307.

Steinwart, I. & Christmann, A. (2008). *Support vector machines*. Springer Science & Business Media.

Swarm. [https://docs.docker.com/engine/swarm/]. (2022). Swarm mode overview.

Taherizadeh, S. & Stankovski, V. (2019). Dynamic multi-level auto-scaling rules for containerized applications. *The Computer Journal*, 62(2), 174–197.

Tam, F. K. (2015). *The Power of Japanese Candlestick Charts: Advanced Filtering Techniques for Trading Stocks, Futures, and Forex*. John Wiley & Sons.

Tang, X., Liu, Q., Dong, Y., Han, J. & Zhang, Z. (2018). Fisher: An efficient container load prediction model with deep neural network in clouds. *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)*, pp. 199–206.

Tong, J.-j., Hai-Hong, E., Song, M.-n. & SONG, J.-d. (2014). Host load prediction in cloud based on classification methods. *The Journal of China Universities of Posts and Telecommunications*, 21(4), 40–46.

Varghese, B. & Buyya, R. (2018). Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems*, 79, 849–861.

Venticinque, S. & Amato, A. (2019). A methodology for deployment of IoT application in fog. *Journal of Ambient Intelligence and Humanized Computing*, 10(5), 1955–1976.

Wang, S., Zhu, F., Yao, Y., Tang, W., Xiao, Y. & Xiong, S. (2021). A computing resources prediction approach based on ensemble learning for complex system simulation in cloud environment. *Simulation Modelling Practice and Theory*, 107, 102202.

Wheelwright, S., Makridakis, S. & Hyndman, R. J. (1998). *Forecasting: methods and applications*. John Wiley & Sons.

Wong, W., Zavodovski, A., Zhou, P. & Kangasharju, J. (2019). Container deployment strategy for edge networking. *Proceedings of the 4th Workshop on Middleware for Edge Clouds & Cloudlets*, pp. 1–6.

Wu, M., Lu, T.-J., Ling, F.-Y., Sun, J. & Du, H.-Y. (2010). Research on the architecture of Internet of Things. *2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, 5, V5–484.

Xu, T., Han, G., Qi, X., Du, J., Lin, C. & Shu, L. (2020). A Hybrid Machine Learning Model for Demand Prediction of Edge-Computing based Bike Sharing System Using Internet of Things. *IEEE Internet of Things Journal*.

Yu, W., Liang, F., He, X., Hatcher, W. G., Lu, C., Lin, J. & Yang, X. (2017). A survey on the edge computing for the Internet of Things. *IEEE access*, 6, 6900–6919.

Zantalis, F., Koulouras, G., Karabetsos, S. & Kandris, D. (2019). A review of machine learning and IoT in smart transportation. *Future Internet*, 11(4), 94.

Zhang, Y., Qi, Y. & Zhang, L. (2018). Time series forecasting using k-nearest neighbor regression. *Applied Soft Computing*, 69, 425–437.

Zhang, Y., Wang, Y. & Luo, G. (2020). A new optimization algorithm for non-stationary time series prediction based on recurrent neural networks. *Future Generation Computer Systems*, 102, 738–745.

Zhong, W., Zhuang, Y., Sun, J. & Gu, J. (2018). A load prediction model for cloud computing using PSO-based weighted wavelet support vector machine. *Applied Intelligence*, 48, 4072–4083.

Zhong, Z. & Buyya, R. (2020). A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources. *ACM Transactions on Internet Technology (TOIT)*, 20(2), 1–24.

Zhong, Z., Xu, M., Rodriguez, M. A., Xu, C. & Buyya, R. (2022). Machine learning-based orchestration of containers: A taxonomy and future directions. *ACM Computing Surveys (CSUR)*.

Zhou, Z.-H. (2012). *Ensemble methods: foundations and algorithms*. CRC press.

Zhu, Y., Zhang, W., Chen, Y. & Gao, H. (2019). A novel approach to workload prediction using attention-based LSTM encoder-decoder network in cloud environment. *Eurasip Journal on Wireless Communications and Networking*, 2019(1), 1–15.