An auto-scaling approach for microservices in cloud computing environments

Matineh ZargarAzad
Iran University of Science and Technology
Mehrdad Ashtiani (m_ashtiani@iust.ac.ir)
Iran University of Science and Technology

Research Article

Keywords: Web applications, microservice, cloud computing, auto-scaling, workload

Posted Date: June 7th, 2023

DOI: https://doi.org/10.21203/rs.3.rs-3020374/v1

Additional Declarations: No competing interests reported.

Version of Record: A version of this preprint was published at Journal of Grid Computing on November 27th, 2023. See the published version at https://doi.org/10.1007/s10723-023-09713-7.

An auto-scaling approach for microservices in cloud computing environments

Matineh ZargarAzad[1], Mehrdad Ashtiani[*2]

[1] School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran
m_zargarazad@cmps2.iust.ac.ir
[2] School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran
m_ashtiani@iust.ac.ir

Abstract

Today, web applications are one of the most common applications providing service to users. Web application providers have moved their applications to cloud data centers. In this regard, microservices have become a famous architecture for building cloud-native applications. Cloud computing provides flexibility for service providers, allowing them to remove or add resources depending on the workload of their web applications. If the resources allocated to the service are not aligned with its requirements, failure or delayed response will increase, resulting in customer dissatisfaction. This problem has become a significant challenge in microservices-based applications because thousands of microservices in the system may have complex interactions. Auto-scaling is a feature of cloud computing that enables resource scalability on demand. This allows service providers to deliver resources to their applications without human intervention under a dynamic workload to minimize resource cost and latency while maintaining the Quality of Service (QoS) requirements. In this research, we aimed to establish a computational model for analyzing the workload of all microservices. This is performed by considering the overall workload entered into the system and taking into account the relationships and call functions between microservices. This is because, in a large-scale application with thousands of microservices, it is usually difficult to accurately monitor all the microservices and gather precise performance metrics. Then, we introduce a multi-criteria decision-making method to select candidate microservices for scaling. The results of the conducted experiments show that the detection of input load toward microservices is performed with an average accuracy of 99% which is a significant value. Also, the proposed approach improves the maximum use of resources by an average of 22.73%, reducing the number of scaling times by 69.82%, and finally reducing the number of required resources which also affects the cost by 1.67% compared to existing approaches.
Keywords: Web applications, microservice, cloud computing, auto-scaling, workload.

1. Introduction

The cloud refers to a network of interconnected servers that enable sharing of computing resources and storage for running applications. This versatile computing model is available in different forms, including infrastructure-as-a-service, platform-as-a-service, and software-as-a-service [1]. Today, developers in most organizations build and deploy applications on the cloud to leverage hosting technologies such as auto-scaling and the resulted resource elasticity. Auto-scaling is a feature of cloud computing that enables application providers to dynamically add or remove

[*] Coresponding author. Address: School of Computer Engineering, Iran University of Science and Technology, Hengam St., Resalat Sq., Tehran, Iran, Postal Code: 16846-13114, Fax: +98-21-73021480.

computational resources based on the requirements and workload without human intervention. The misalignment of the allocated computational resources with the service's demands is considered a threat to the provided quality on the other hand, if the number of resources allocated to the service is more than needed, we will have a waste of resources, and as a result, the cost increases. So, the scaling mechanism must work as accurately as possible and effectively to maintain the service level agreement.

Microservices have become a well-known architectural pattern for building cloud-native applications. Cloud-native is an approach that emerged in 2015 focusing on how applications are built and executed to fully take advantage of the cloud computing model. Cloud-native technologies empower organizations to build and run scalable applications in modern and dynamic environments. Containers, orchestrators, service meshes, microservices, immutable infrastructure, and declarative APIs enable loosely coupled systems that are resilient, manageable, and observable. Based on the findings of the Cloud-native Computing Foundation (CNCF) survey conducted in 2018, the utilization of CNCF projects in production environments has experienced an average growth of over 200% since December 2017 [2]. However, auto-scaling in applications with microservice architecture still is a big challenge. This is due to the existence of hundreds or thousands of microservices that may have complicated interactions.

By separating different parts in integrated applications (i.e., bounded context) and turning the program into small components, each part would be responsible for performing an individual task. The implementation of different parts is performed independently of the others. Each part will be assigned its resources and the scaling of resources is performed separately for each microservice and based on its demands. Besides its clear advantages such as better potential for scalability and independent implementation, this architectural pattern has its downsides as well. For example, monitoring at the service level is difficult. Generally, only a high-level report of system metrics, such as the overall resource usage or the input workload to the entire system will be available. In addition, it can also be pointed out that by having the overall system metrics in hand, when the number of microservices is large, if the response times become larger, or if the resource usage amount violates the service-level agreement with the client, it will be challenging to determine which microservice such violation or prolonged response time is related to and which one lacks enough resources based on the input workload pattern toward that individual microservice.

In this paper, we provide a proactive solution to overcome some of these challenges. In this regard, in addition to finding and obtaining criteria related to microservices by considering the relationships between them, we also predict these criteria to act proactively in case of workload changes. Then, we use them to find the microservices that require more/fewer resources and make a decision to perform the scaling. This decision aims to keep the resource usage as high as possible in a way that the agreement with the client is not violated and no waste of resources is also present. Another point is that the importance of each criterion in the decision-making can be different from the others according to the type of application. Also, the interactions of microservices are modeled with a complex network. By considering the mathematical features of complex networks critical microservices with a high probability of increasing workload are identified. The last point is that in the structure of microservice-based software, the number of resources used in each component may differ. Therefore, in the conducted tests, we also considered this aspect and divided the microservices in the system into two categories: (1) CPU-intensive and (2) memory-intensive. The major contributions of this work are summarized as follows:

1. Considering the relationships and function calls between microservices as a graph.
2. Finding the workload of microservices using the workload input to the program and the relationships between them in the form of probabilistic hesitant fuzzy sets.
3. Using an efficient forecasting method to predict the workload of each microservice that can be well adapted to diverse workloads.
4. Using a multi-criteria decision-making method using system criteria calculated and predicted for each microservice to select the necessary microservices to change the scale at any time.
5. Considering the importance of each criterion in decision-making according to the program's requirements.
6. Considering the different resource usage in microservices with high CPU or memory consumption.

The remaining sections of this paper are organized as follows: Section 2 introduces the required background knowledge for a better understanding of the paper. Section 3 reviews the previous works on auto-scaling approaches and compares the related work with the existing literature. The proposed approach is introduced in section 4. In section 5, the results of the performed evaluations are given and discussed. Finally, section 6 concludes the paper and suggests future work.

2. Background Knowledge

In this section, we introduce the primary concepts and notations related to the main topic of the paper. We will discuss the concepts of cloud computing, auto-scaling, and microservices.

2.1. Cloud computing

Cloud computing entails providing computing resources over a network per user requirements [3]. The available resources encompass data storage, servers, databases, networks, and software. Cloud-based storage enables us to store files in a remote database rather than storing files on a dedicated hard drive or local storage device. Cloud hosting is frequently employed for web applications with a significant user base to meet desired quality of service standards, including reliability, availability, and performance [4]. Cloud computing offers users the flexibility to adjust the number of virtual machines based on the workload of their applications. Users can swiftly configure virtual machines for a specific duration and solely pay for their required usage. In essence, this eliminates the need for extensive time and financial investments in purchasing physical hardware and covering maintenance costs [5]. The characteristic of elasticity sets cloud computing apart from traditional IT infrastructures. Given the dynamic nature of cloud environments, workloads, and internal states, cloud software systems must continuously adjust to changing conditions to uphold their service-level agreements (SLAs) and optimize resource utilization [6].

As per the National Institute of Technology and Standards' definition, cloud computing represents a model that enables public, convenient, and on-demand network access to a shared pool of easily configurable computing resources. These resources encompass networks, servers, storage space, software, and services and can be rapidly provisioned with minimal managerial involvement or interaction with service providers [7]. One of the key distinctions between computational clouds and other computing technologies like grids and clusters, which necessitate resource reservations in advance, is the capability to dynamically scale resources as needed [8]. There are essentially two primary justifications for dynamic scaling: (1) to prevent violations of service-level agreements caused by an influx of requests and (2) to achieve cost savings during periods of low request volumes. [9].

2.1.1. Cloud computing characteristics

Cloud computing provides the following capabilities to users making this computational paradigm the optimal choice for modern applications: [7] [10]

1. **On-demand and self-service:** A consumer has the autonomy to independently allocate computing capabilities, including server time and network storage, as required, without the need for human interaction with individual service providers.
2. **Broad network access:** The capabilities are accessible over the network and can be accessed through standard mechanisms utilized by diverse client platforms, whether they are thin or thick, such as mobile phones, tablets, laptops, and workstations.
3. **Resource pooling:** The computing resources of the provider are consolidated into a shared pool to cater to multiple consumers using a multi-tenant model. This involves the dynamic assignment and reassignment of various physical and virtual resources based on the demands of the consumers. While there is a sense of location independence, where the customer typically lacks control or awareness of the precise resource location, they may have the ability to specify a higher-level abstraction, such as country, state, or data center. The available resources encompass storage, processing power, memory, and network bandwidth.
4. **Rapid elasticity:** Resources can be provisioned and released elastically, with the ability to scale up or down to match the demand rapidly. In certain cases, this scaling process can occur automatically. From the consumer's perspective, the available capabilities for provisioning often seem limitless and can be increased or decreased in any desired quantity at any given time.
5. **Measured service:** Cloud systems employ automated mechanisms to manage and optimize resource utilization by utilizing metering capabilities at an appropriate level of abstraction based on the type of service, such as storage, processing, bandwidth, and active user accounts. This allows for monitoring, control, and reporting of resource usage, ensuring transparency for both the service provider and the consumer of the utilized service.

6. **Autonomous:** The cloud operates as a self-governing system, providing transparent management for users. Through automated processes, hardware, software, and data can be dynamically reconfigured, orchestrated, and consolidated to present a unified image ultimately delivered to users.
7. **QoS guaranteed offer:** Cloud computing environments offer users the assurance of Quality of Service, which includes guarantees on hardware performance aspects such as CPU speed, I/O bandwidth, and memory capacity.
8. **Scalability and flexibility:** The emergence of cloud computing is primarily driven by its key features of scalability and flexibility. Cloud services and computing platforms can be easily scaled to address diverse concerns, including geographical locations, hardware performance, and software configurations. Moreover, the computing platforms need to be flexible enough to accommodate the varied requirements of a potentially extensive user base.
9. **Pay-as-you-go:** Referred to as a pay-per-use model, this feature implies that cloud service users exclusively rent and utilize the required resources without any unnecessary ownership or long-term commitments.

2.1.2. Cloud computing architecture

In the work of Zhang et al., cloud computing architecture is divided into four layers: (1) hardware or data-center layer, (2) infrastructure layer, (3) platform layer, and (4) application layer [11]. This schema is shown in Figure 1.

1. **Hardware layer:** The hardware layer assumes the responsibility of overseeing the physical resources, such as physical servers, routers, switches, power infrastructure, and cooling systems. In practical terms, the hardware layer is commonly deployed within data centers, which typically house thousands of servers organized in racks and interconnected through switches, routers, or similar infrastructure. Key concerns at the hardware layer encompass hardware configuration, fault tolerance, traffic management, and the efficient management of power and cooling resources.
2. **Infrastructure layer:** Referred to as the infrastructure layer or virtualization layer, this component forms a resource pool of storage and computing resources by leveraging virtualization technologies like Xen [12], KVM [13], and VMware [14] to partition the underlying physical resources. The infrastructure layer holds significant importance in cloud computing as it enables crucial features such as dynamic resource allocation that are made possible through virtualization technologies.
3. **Platform layer:** Situated above the infrastructure layer, the platform layer comprises operating systems and application frameworks. Its primary objective is to alleviate the complexities associated with deploying applications directly into virtual machine (VM) containers. An illustration of this is Google App Engine, which operates within the platform layer and offers API support for implementing storage, database, and business logic functionalities in typical web applications.

4. **Application layer:** At the topmost tier of the hierarchy, the application layer encompasses tangible cloud applications. Distinguishing themselves from traditional applications, cloud applications have the advantage of harnessing the automatic-scaling capability to enhance performance, availability, and reduce operational expenses.

2.1.3. Cloud computing services

The cloud architecture can be divided into the backend and frontend sections. The frontend is made visible to the user through connections to the Internet, allowing user interactions with the system.

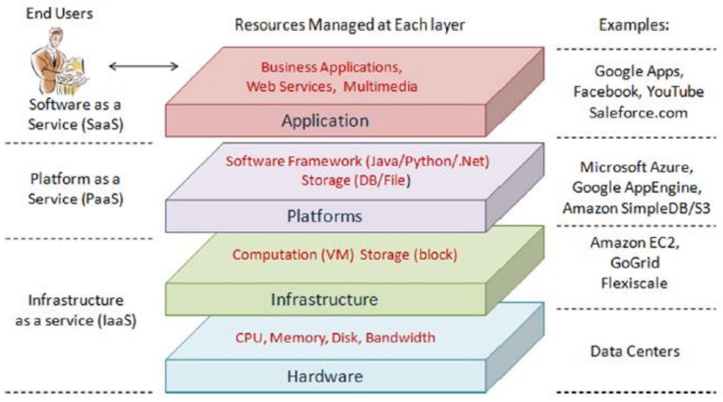

End Users
Resources Managed at Each layer
Examples:
Software as a Service (SaaS)
Business Applications, Web Services, Multimedia
Application
Google Apps, Facebook, YouTube Saleforce.com
Platform as a Service (PaaS)
Software Framework (Java/Python/.Net) Storage (DB/File)
Platforms
Microsoft Azure, Google AppEngine, Amazon SimpleDB/S3
Computation (VM) Storage (block)
Infrastructure
Amazon EC2, GoGrid Flexiscale
Infrastructure as a service (IaaS)
CPU, Memory, Disk, Bandwidth
Hardware
Data Centers

Figure 1. Cloud computing architecture [11].

The back end comprises the various cloud services models [15]:

1. **Software-as-a-Service (SaaS):** The user is provided with a software suite hosted on a platform and infrastructure owned by the cloud service provider (CSP). Applications are specifically designed and developed to be accessible by multiple cloud consumers concurrently via the internet. The CSP is responsible for managing and maintaining the hosted application, ensuring that it operates in an up-to-date manner. The hosted application supports multitenancy, can be accessed on-demand, and offers scalability options. In some instances, Software-as-a-Service (SaaS) providers may utilize the Platform-as-a-Service (PaaS) or Infrastructure-as-a-Service (IaaS) offerings of other cloud providers. Examples of applications falling into this category include email applications, text editors and word processors, spreadsheet programs, and presentation software.
2. **Platform–as–a–Service (PaaS):** PaaS is an internet-based development service provided to users, eliminating the need for software installation or hardware requirements and resulting in cost savings. It acts as middleware upon which applications can be built. PaaS offers integrated tools, security features, and web service interfaces for the deployed applications. These deployed applications can be seamlessly integrated with other applications within the same platform and can interface with external applications. PaaS includes software components such as databases, middleware, and development tools. Examples of PaaS applications include business intelligence systems, databases, development environments, and testing environments.
3. **Infrastructure–as–a–Service (IaaS):** Refers to delivering servers, storage, networks, and

operating systems as a service. With IaaS, users are provided with a pre-configured and installed abstract machine that includes an operating system. It allows data to be stored in different geographical locations. IaaS providers maintain control over activities within the cloud data centers, while users can deploy and manage their own software services. Users have access to a virtual computer, storage, network infrastructure, and computing resources for deploying and running software. The cloud provider takes care of managing the software and hardware components, such as servers, storage devices, the host operating system, and the hypervisor for virtualization. Examples of IaaS applications include backup and recovery systems, compute resources, and storage services.

2.2. Auto-scaling

In the present day, developers often encounter the task of constructing and deploying extensive applications that utilize the resources of multiple virtual machines on cloud infrastructure. Adequately provisioning these machines continues to be a challenge. Workloads are in a constant state of flux, and even minor modifications to application logic can lead to unpredictable changes in resource demands. Over-provisioning applications results in resource wastage and higher expenses. However, under-provisioning leads to sluggish execution and a subpar user experience. Tenants aim to optimize resource utilization while ensuring a satisfactory user experience. The scalable infrastructures offered by cloud providers aid in accommodating the needs of tenants. [16].

Autoscaling is an inherent capability of cloud computing, enabling the automatic adjustment of resources based on demand. This feature enhances fault tolerance, availability, and cost management. [3]. The process of dynamically adding or removing Virtual Machines (VMs) to optimize costs and minimize latency is referred to as auto-scaling.

2.3. Microservice-base architecture

Cloud services have recently shifted from monolithic applications to micro-services because of the benefits they provide for application providers such as scalability, reliability, availability, and dynamic development so that a large number of them together make up an end-to-end application. Spotify, LinkedIn, Soundcloud, and Netflix are examples of programs developed through microservices.

Microservices offer several appealing advantages. Firstly, they simplify and expedite deployment by leveraging modularity, with each microservice responsible for a specific subset of the overall application's functionality. Secondly, microservices can capitalize on language and programming framework diversity, as they only require a shared cross-application API, typically facilitated through remote procedure calls (RPC) or a RESTful API. Thirdly, microservices streamline the process of debugging for correctness and performance issues, as bugs can be isolated to specific components, unlike monolithic architectures, where troubleshooting often involves the entire service. Lastly, microservices align well with the container-based data center model, with each microservice encapsulated within its own container [17]. Finally, when scaling microservices, resources can be added only to desired services that require more resources, not the entire system, and this feature optimizes scaling.

One of the primary challenges posed by microservices is that performance issues cannot be solely reliant on client reports, as is the case with traditional client-server applications. Addressing performance concerns involves identifying the microservice responsible for a Quality of Service violation. This task is commonly achieved through distributed tracing, which helps trace and analyze the flow of requests across multiple microservices [17]. Additionally, appropriate tools to monitor each microservice's efficiency do not exist.

3. Related Work

By reviewing a wide-range of approaches that provide methods for the automatic scaling of resources, we first present an overview of these strategies currently introduced in this field. This taxonomy is shown in Figure 2. Then, we will discuss in more detail each of these categories.

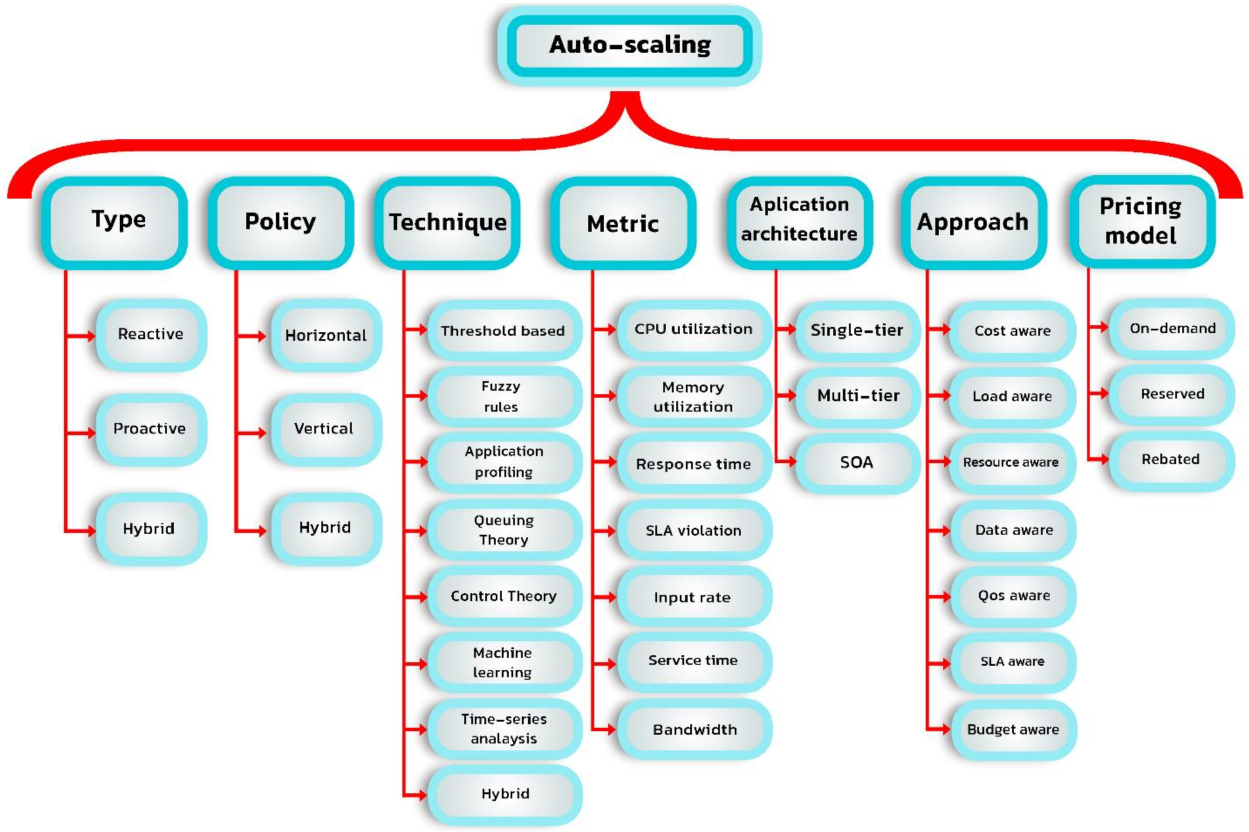


Figure 2. Taxonomy of auto-scaling resources in web applications.

Various researchers have focused on one or several of these aspects. Some articles have emphasized the horizontal, vertical, or hybrid nature of scaling. For example, Gias et al. in [18] used both horizontal and vertical scaling in their approach, equivalent to a hybrid approach. They construct a layered queueing network model of the application at run-time to find resource usage. Some have focused on using or providing appropriate novel techniques to obtain thresholds for reactive scaling. Rudrabhatia [19] proposed a quantitative approach for estimating the scaling thresholds in microservices. In this work, after finding the total resource consumption for the peak load scenario, an optimal upper threshold value for the step-up functions is calculated by considering the network and startup latencies. However, this method is reactive and reactive methods take action when an SLA violation is detected. In this way, researchers have examined some of these classifications or all of them. In [5], the authors proposed a reactive approach with fuzzy rules and dynamic thresholds.

Abdullah et al. have proposed a hybrid of a reactive and proactive approach to find the most appropriate resource demands in microservice-based applications [20] [21]. In both articles, they used approximately similar strategies. They initially proposed an approach using a reactive auto-scaling method to gather sufficient performance traces to learn the predictive resources provisioning policy model. After training the model with dynamic tensor re-materialization (DTR), they forecast the future workload using recent observations. Then, the number of sufficient containers is detected using predicted workload values to satisfy these service-level objectives (SLO) requirements. Bouabdallah et al. proposed an approach to scale up resources reactively with user-defined thresholds and scale down proactively using the simple exponential smoothing method (SES) [22].

Proactive approaches focused on metrics such as workload, resource utilization, bandwidth usage, etc. Furthermore, different techniques are used. Messias et al. combined time-series prediction models such as Naïve, AR, ARMA, ARIMA, and ETS using genetic algorithms to auto-scale web applications hosted in the cloud infrastructure [9]. After predicting demand, an M/m/m queue model is used to calculate the number of resources. However, genetic algorithms are slow and have computational overhead. In several papers, researchers presented machine learning techniques to predict future demands of applications. Al Qassem et al. designed a random forest (RF) model to forecast the future CPU and memory utilization values required by the microservice workload. Then, these predicted values are used to adjust the resource pool vertically and horizontally [23]. Prachitmutita et al. proposed an auto-scaling framework based on predicted workload, with an artificial neural network, recurrent neural network, and resource scaling optimization algorithm to create an automated system for managing the whole application via performing scale-out or scale-in in the IaaS layer [24]. Radhika et al. used a deep learning technique termed recurrent neural network with long short-term Memory (RNN-LSTM) to predict future demand based on historical data. Based on the result, the predictive autoscaling is performed on OpenStack by triggering an instance using Terraform [25].

Aslanpour et al. worked on the execution phase of auto-scaling [26]. They claim that there is no specific strategy in the execution phase to remove virtual machines during scaling-down, and in the best case, from the selection mechanism, they remove the machine that went up earlier, but this strategy must be executed manually.

We have compared several works that have been studied in this section in Table 1. The pricing model of all investigated works is on-demand. Several researchers have also focused on microservice-based architecture which is also mentioned in the table.

Table 1. A comparison of the reviewed research in the auto-scaling domain.

Article	Type	Policy	Metrics	Technique	Service-based	Approach
J. Novak et al. [16]	Proactive	Hybrid	Memory	Application profiling / Rule-based	×	Resource and workload aware
S. Srirama et al. [27]	Reactive	Horizontal	CPU and memory utilization	Rule-based	✓	Container aware
E. G. Radhika et al. [3]	Proactive	Horizontal	CPU and RAM utilization	Time series	×	Workload aware
B. Liu et al. [5]	Reactive	Horizontal	CPU utilization / Response time	Fuzzy rules	×	Resource and SLA aware
V. Messias et al. [9]	Proactive	Horizontal	Workload	Time series	×	Resource and SLA aware
I. Prachitmutita et al. [24]	Proactive	Horizontal	Workload	Machine learning	✓	Cost aware / SLA
A. U. Gias et al. [18]	Proactive	Hybrid	Resources utilization	Queuing model	✓	Resource aware
A. Khaleq et al. [28]	Reactive	Horizontal	memory and number of undelivered messages in the queue.	Rule-based	✓	QoS aware
M. S. Aslanpour et al. [26]	Hybrid	Horizontal	Resources utilization / SLA	Rule-based	×	Cost aware
M. Abdullah et al. [20]	Proactive	Horizontal	SLA	Machine learning	✓	Resource and SLA aware
f. Zhang et al. [29]	Reactive	Horizontal	CPU utilization / Response time/scaling speed	Rule-based	✓	Container aware
M. Abdullah et al. [21]	Proactive	Horizontal	Workload	Machine learning	✓	Burst aware
A. Shahidinejad et al. [30]	Reactive	Horizontal	Response time/cost / CPU utilization	Workload clustering	✓	QoS aware
J. Benifa et al. [8]	Proactive	Horizontal	CPU utilization / Response time / Throughput	Reinforcement learning	×	Resource aware
G. Yu et al. [31]	Reactive	Horizontal	Service delay	Rule-based	✓	SLA aware
B. Zhu et al. [32]	Reactive	Horizontal	Resources utilization	Rule-based	✓	Container aware
Y. Li et al. [33]	Hybrid	Hybrid	CPU and memory utilization	Time series / Rule-based	✓	QoS aware
R. Bouabdallah et al. [22]	Hybrid	Horizontal	CPU utilization	Time series / Rule-based	×	Resource aware

4. The Proposed Approach

In software with a microservice-based architecture, there may be dozens or even hundreds of different microservices, each of which may have thousands of instances. These microservices have complex communication patterns and call each other through particular protocols. As a result, the workload is transferred from one microservice to another. So, scaling the resources in these environments is complicated because a change in the resources of one of them may affect a large number of other microservices. The investigation of this issue has rarely been seen in similar studies.

Our goal is to decide which microservices, among all, should be scaled and need more resources. After identifying the microservices whose scaling is critical, we will determine the importance and necessity of scaling each of the remaining microservices. Finally, based on that, determine the number of resources needed by each, do the scaling work, and apply the changes. In addition to these cases, achieving the existing criteria separately for each microservice takes time, cost, and effort. We will achieve these criteria for each microservices using a mathematical model presented in this section.

In the presented approach, it is assumed that there are many microservices, and we will decide and prioritize the scaling of critical microservices. To make a decision, we use a multi-criteria decision-making method. This solution has four main components: (1) workload calculation, (2) workload prediction in the future time window (t+1), (3) resource consumption calculation, and finally, (4) making the final decision using the previously mentioned components. A mathematical model is presented in the workload calculation component. It calculates the input workload toward each microservices according to their complex relationships and the workload that enters the input service.

After obtaining the current workload of each service, we use machine learning methods to predict their individual workload at the time (t+1). Then, the difference of additional workload in the future time window compared to the current time is obtained where the higher this value means the need for more resources. This component is one of the decision criteria in the proposed approach. As a result, the more accurate this prediction is the better the overall results would be. Another criterion used in this research is calculating resource utilization. We monitor each microservice and provide its data as another input to the decision-maker component. Each microservice includes several instances or containers, each with its resources. As a result, several containers form a cluster, representing a microservice. The amount of resources utilization for a microservice represents the average resources consumed in its respective containers. The more CPU and memory resources is used, the more resources it additionally needs. In the decision-making section, using the values and criteria calculated for each microservice according to the explanations given earlier, we make a suitable decision using an efficient decision-making method and finally perform the correct scaling.

In Figure 3, we see the general architecture of the proposed approach. There are a total of three layers in this architecture. At first, requests are sent from the customers to the system and received by the API gateway. Then, each input workload goes to the desired service according to its needs. As it is known, microservices are all implemented in the cloud environment. Each microservice has some containers to which a load balancer sends requests. According to the calls between upstream microservices and downstream microservices, some connections exist between them. We intend to use these relationships and the information obtained from them to make the right decision. On the other hand, using existing sensors and monitoring tools, the requested information goes from the resource side to the workload calculation components. After calculating the input workload toward each microservices, the amount of resource usage and the predicted workload is calculated. Finally, the calculations related to these components are entered into the decision-maker component. The decision made about resource scaling goes through the controller to the load balancer, and the changes are applied. In the following sections, we will explain each component in detail.

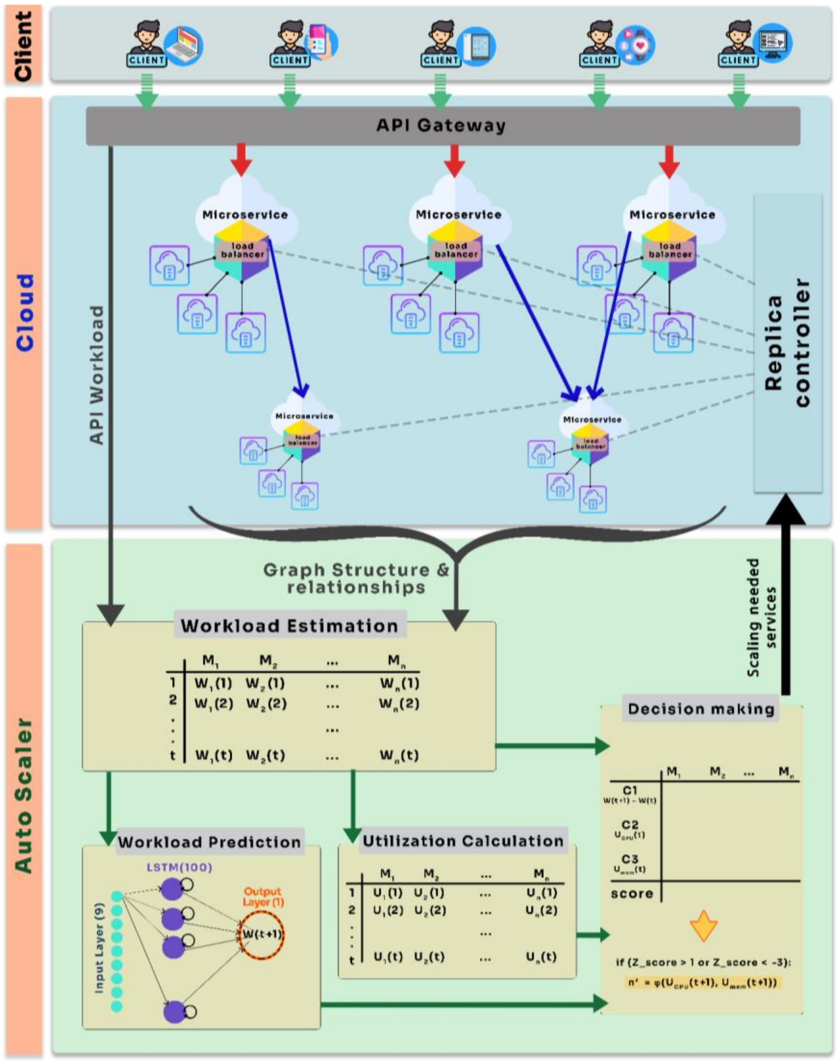


Figure 3. Overview of the proposed approach.

Figure 4 shows the flowchart of the steps in the proposed solution. These steps will be explained gradually in the following sections. In this figure, the main phases of automatic scaling, which include monitoring, analysis, planning, and execution, are specified.

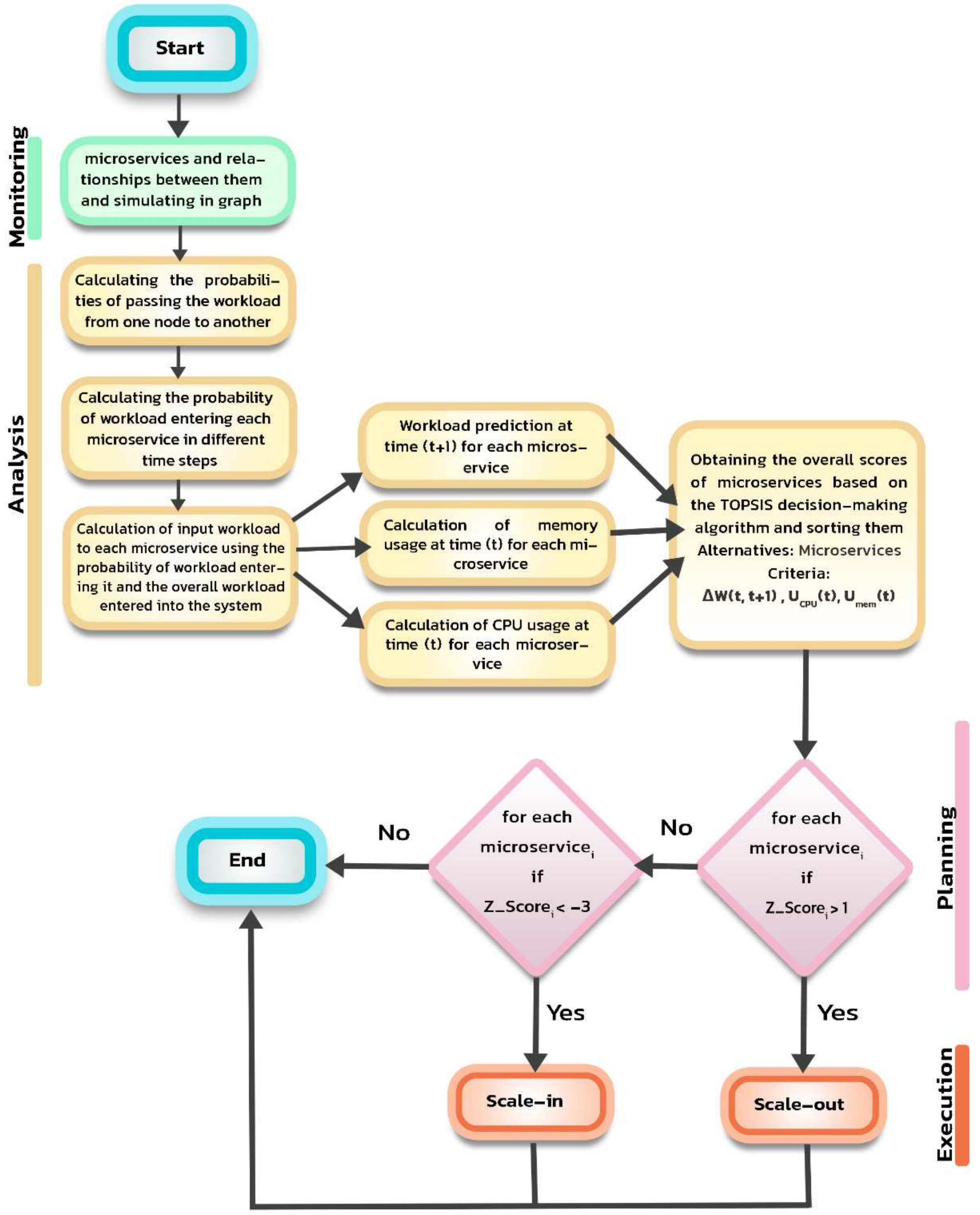

Start
Monitoring
microservices and relationships between them and simulating in graph
Analysis
Calculating the probabilities of passing the workload from one node to another
Calculating the probability of workload entering each microservice in different time steps
Calculation of input workload to each microservice using the probability of workload entering it and the overall workload entered into the system
Workload prediction at time (t+1) for each microservice
Calculation of memory usage at time (t) for each microservice
Calculation of CPU usage at time (t) for each microservice
Obtaining the overall scores of microservices based on the TOPSIS decision-making algorithm and sorting them
Alternatives: Microservices
Criteria:
$\Delta W(t, t+1)$, $U_{CPU}(t)$, $U_{mem}(t)$
Planning
for each microservice$_i$ if Z_Score$_i$ > 1
for each microservice$_i$ if Z_Score$_i$ < -3
No
No
Yes
Yes
Execution
Scale-out
Scale-in
End

Figure 4. Flowchart of the proposed approach.

4.1. Decision-making

In this decision-making problem, we use the multi-criteria decision-making method to choose which existing microservices need scaling based on some predetermined criteria. So, each of the different options (i.e., alternatives) represents a microservice. On the other hand, we have to determine the criteria around which this decision should be made. These criteria are:

1- The changes in workload until the future time window.

2- The amount of CPU usage.

3- The amount of memory usage.

For explaining why these criteria have been chosen, the higher the number of resources currently used or in case they are fully utilized, the more likely we will need to increase the scale in the next window. These resources include CPU and memory. On the other hand, another essential criterion is how much workload is supposed to be added or reduced for each microservice which is the basis for prediction.

We chose horizontal scaling in our experiments. This way, we will change the number of containers by increasing or decreasing the scale. Also, the service level agreement specifies how much the percentage of system metrics can be utilized. For this purpose, we also considered a threshold limit for the use of resources so that the amount of resource usage does not exceed this value. A challenge in this section is determining the weight of the introduced indicators. These values may vary according to the type of program and data. Assume that the workload changes from one time period to the next are not significant. In this case, services that have fully utilized their resources need to scale out with a slight change. If the services are generally of this category, the weight related to the amount of resource usage should be considered higher than the changes in the number of requests. On the other hand, suppose that the changes generally face a significant increase or decrease. In this case, although the amount of resource usage is not high, with a significant change, the service may need to increase or decrease the scale. So, we consider the weight related to load changes more than the amount of resource usage. Finally, we should have a matrix like Table 2 where we will make the necessary decision using the points obtained for each microservice in the decision-making section.

Table 2. The decision-making table in the proposed approach.

	$A_1(mic_1)$	**$A_2(mic_2)$**	**…**	**$A_n(mic_n)$**
C1($\Delta W^{(t,t+1)}$)	A_{11}	A_{12}	…	A_{1n}
C2(U^t_{mem})	A_{21}	A_{22}	…	A_{2n}
C3(U^t_{CPU})	A_{31}	A_{32}	…	A_{3n}
μ_1	μ_1	μ_2	…	μ_n

4.2. Calculating the microservices' workloads

Workload means the number of incoming requests per unit of time. In this section, it is necessary to calculate the probability of the workload entering each microservice in the time intervals between 1 and t in the form of a probabilistic hesitant fuzzy set. After obtaining the workload, we predict its amount for the time interval $(t + 1)$. As we mentioned earlier, microservices call each other and have complex relationships. So, we can model the microservices and the relationships between them with a graph and more specifically a Bayesian belief network. We want to calculate the probability of workload entering each microservice from this mathematical representation.

Park et al. state that several current auto-scalers [24] [34] [35] [36] manage the resources of each microservice individually, which leads to cascading effects and significant performance degradation during periods of high traffic. These auto-scalers only become aware of workload changes once the deepest microservice in the chain is affected, resulting in a delayed response to traffic surges. Additionally, the time taken for instance creation of each microservice further exacerbates this issue. Such immature behavior of auto-scalers has a noticeable impact on tail latency and should not be overlooked. [37]. Creating the Bayesian graph of relationships between microservices and extracting inference with it can be very useful. So far, few researchers have studied the relationships graph between microservices [34] [37] [38]. Moreover, all of these researches have performed experiments on a small graph with a small number of nodes and few edges which is not necessarily realistic.

First, we must provide a precise definition of the desired graph and the possible set and values that are on each edge. Each graph is defined by a binary component $G\ (V, E)$, where V is the set of graph nodes, and E is the set of edges which are shown as the set of $E\ =\ (v_i, v_j)$ which means that the edge is drawn between two nodes v_i and v_j. Two nodes that have an edge between them are called neighbors.

Assume a graph like the one shown in Figure 5 representing the microservices' relationships. Each node V represents a microservice, and the edges between them represent the connection between the two nodes. A microservice relation graph is a directed graph. For example, in this graph, an edge is drawn from node A to B, C, and D. This means that microservices B, C, and D are called by microservice A. Finally, a probabilistic hesitant fuzzy set is obtained for each node that represents the probability of input workload toward them. This set is formulated as Equation 1. By calculating these values for each node, aggregating these values, and having the input workload to the system and API gateway, we will obtain the input workload for each microservice. For example, suppose an edge is drawn from A to B. Its value means that if the workload enters A, with what probability it also enters the microservice B? (i.e., $P(B|A)$). In other words, what percentage of the input workload of A goes to B?

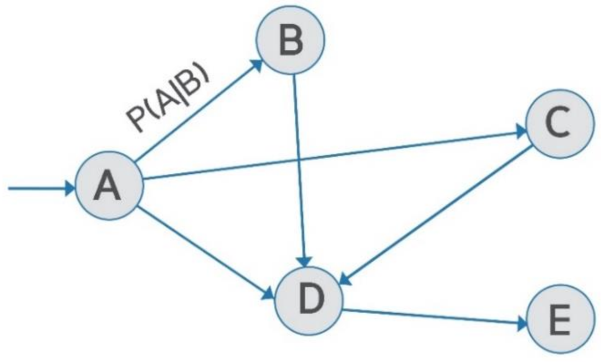


Figure 5. Example of relationships graph.

$$A_p = \{<x, h_x(p_x)> \mid x \in X\} \tag{1}$$

Suppose the following statement is obtained for node D:

$$\mathrm{P(D)} = \{0.65(0.2), 0.8(0.6), 0.9\ (0.2)\}$$

It means that in the incoming workload toward D, which may come from the API calls generated by nodes A, B, or C, 65% of workloads from A will enter D with a probability of 20%. 80% of workloads from B will enter D with a probability of 60%. Finally, in 20% of the cases, 90% of the input workload from C reaches D. The work steps are as follows.

1) In the first step, we must have the data related to the workload and the number of requests coming from one node to another at any moment or the number of calls from one microservice to another. These numbers are related to the edges, and the aim is to reach the possible workload values of each node by changing and aggregating these values during several phases.
2) In the second part, we will calculate the possible values for edges, meaning that if, for example, an edge is drawn from microservice A to B, C, and D, how much of the input workload toward A will go to each of these microservices. In other words, if a workload enters A, with what probability this workload will be redirected to each of the microservices that are connected to A. The algorithm for calculating this value in each relationship graph is shown as Algorithm 1. In this algorithm, we first obtain the sum of the weights for the input edges connected to the neighbors of each node. Then, the weight of each edge is divided by the sum of the weights of the input workload toward the source node, which is given in Equation 2.

$$P_{(A,B)} = {W_{(A,B)}}\Big/{\textstyle\sum_{X \in adj(A)} W_{(X,A)}} \tag{2}$$

Algorithm 1: Edge_probability

Input: Graph G with weights of edges

Output: Graph G with edge probability

```
1: for each $node$ do:
2:     weight($node$) = $\sum_{X \in \mathrm{adj}(node)} W_{(X,node)}$
3:     for each $edge$ ($node$, $Y$) do:
4:         $P_{edge} \leftarrow \mathrm{weight}_{edge}$ / weight($node$)
5: return $G$
```

3) In the third phase, we will obtain the possible values of the input workload for each microservice according to the edges connected to it. We assume that we have a starting node to which all workloads first enter, which in real-world deployments is usually the API gateway

in a microservice architecture. In Figure 5, this node is assumed to be node A. Each node may receive a workload from one or more nodes.

To aggregate these values for each node, we consider all the possible paths from the starting node toward the desired node and calculate the probabilistic value corresponding to that path from the beginning to the end. Finally, by summing these values for each node, the work aggregation is done, and we reach the desired result. For example, for node D in Figure 5, we obtain the expression in Equation 3, where the probability of the workload arriving from the starting node to the desired node is obtained by calculating all possible paths.

$$P(D) = P(D|A) + P(D|B) * P(B|A) + P(D|C) * P(C|A) \tag{3}$$

The pseudo-code for calculating this value is presented as a recursive algorithm in Algorithm 2. In this algorithm, it is assumed that the list or neighbors are the nodes from which an edge is drawn to the desired node.

Algorithm 2: Node_ aggregation

```
Input: node I, Graph G
//node I = Initial node && graph G for p(edge) && adjacency list for each node
Output: Result_list // probabilities of all nodes in a graph
1:  Initialize adj_list of graph G
2:  Initialize result_list
3:  for each node n do:
4:      P = 0
5:      if (n == I) then
6:          add 1 to result_list
7:      else
8:          for each adj of node n in adj_list:
9:              initialize p(n|adj) from G
10:             p = p + Node_aggregation(adj) * p(n|adj)
11:         add p to result_list
12: end for
13: return result_list
```

4) We assume that we have a large number of graphs, each of which has its information and represents the state of the system at any moment. We want to reach the probabilistic hesitant fuzzy values for the nodes of a new graph by aggregating some graphs recorded in consecutive moments, obtained from aggregating several existing graphs in a time interval.

In the problems related to hesitant fuzzy sets, some experts initially specify the desired values. Then, the opinions of these people are aggregated and turned into a hesitant fuzzy set. However, we obtain these values from recorded graphs at each time interval instead of specifying the values by humans. Here, for each node in a time interval (t) that has been monitored several times, in each monitoring cycle, we have a new graph where the nodes and edges do not change, but the weight on the edges changes each time according to the information obtained from the previous phase. We use Equation 4 used in [39] to calculate this amount and gather information.

$$h_x^T(p_x) = \{h_i(p_i) \mid \mathrm{i} = 1, 2, \ldots, \#h_x\} \tag{4}$$

Here, the final value is equal to the set of $h_i(p_i)$ calculated in multiple time intervals from section (3) such that h_i are all possible values that appeared in each graph for a node. The corresponding p_i is the probability of its occurrence, and their number is equal to $\left\lceil\frac{WS}{\xi}\right\rceil$ or the number of graphs. In fact, in calculating the workload in a time interval WS, several observations are made in smaller time intervals ξ, and all values are aggregated. A single set is obtained for each time window (WS) and each microservice (i.e., Figure 6). The proposed approach sets the number of time intervals in a time window to 10 which is based on try and error in the evaluations. We need to obtain a final score from the PHFS corresponding to each microservice. This score shows the probability of the workload reaching each microservice at any moment. To calculate the function score for a microservice in a time window, Algorithm 3 is introduced.

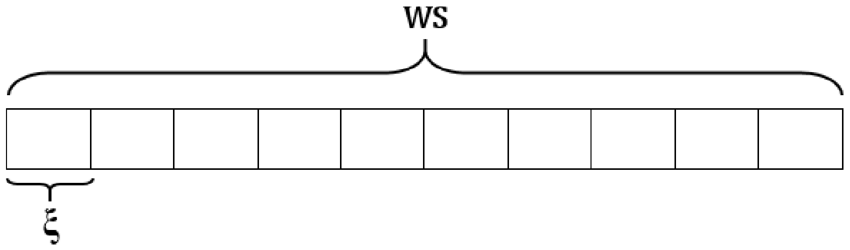


Figure 6. The time window in the proposed approach.

At the end of all calculations, we will have a table similar to Table 3. A workload value for each node in several time intervals is calculated according to the input workload toward the entire system and the relationships between microservices. Finally, we predict them for time $(t + 1)$. Now we want to calculate the workload at any moment for a microservice. This value is calculated by having the probability of the workload reaching each node or the percentage of the workload entering it in a specific time window and the workload entering the API gateway in the same interval. This will be calculated using Equation 5. In the next step, we will predict these values.

$$W^{\;T} = P^T \; * {W_{API\_Gateway}}^T \tag{5}$$

```
Algorithm 3: node_score
input: Monitoring interval (ξ), window size (WS)
output: Final workload value in a window size for all microservices
 1: list, result_list = []
 2: while (time < WS) do:
 3:     wait for $\xi$
 4:     Construct graph G based on microservices and their relationships in time $t + \xi$
 5:     $m_1$ ← API gateway node
 6:     $P$   = node_aggregation ($m_1$, $G$)
 7:     add $P$   to list
 8:     time ← time +1
 9: for each microservice do:
10:     $h_x^T(p_x) = \{h_i(p_i) \mid i = 1, 2, \ldots, \#h_x\}$
11:     score_function ← $\sum_{i=1}^{\#h_x} h_i \; p_i$
12:     add score_function to result_list
13: end for
14: return result_list
```

Table 3. workloads and expected values of microservices.

Microservices Time intervals	A_1	A_2	…	A_n
1	$W_1^{(1)}$	$W_2^{(1)}$	…	$W_n^{(1)}$
2	$W_1^{(2)}$	$W_2^{(2)}$	…	$W_n^{(2)}$
⋮	…	…	…	…
t	$W_1^{(t)}$	$W_2^{(t)}$	…	$W_n^{(t)}$
t+1	$W_1^{(t+1)}$	$W_2^{(t+1)}$	…	$W_n^{(t+1)}$

4.3. Predict workload

In this step, we will use the values obtained from the previous step such as the workloads related to microservices in different time windows, to predict and calculate each node's values in the next step $(t + 1)$. To predict these values in a time window $(t + 1)$, we applied and evaluated various machine learning algorithms like KNN regression, Linear Regression, Lasso regression, LassoCV, Ridge regression, Elasticnet regression, Decision tree, random forest, and gradient-boosted regression. We also used RNN neural networks based on LSTM.

We initially have a list of available values. Depending on what time window and how many previous values we want to consider for predicting the subsequent values, we shift the initial list forward one place at a time and use that value as the previous value. Finally, we used the LSTM model that showed the highest accuracy and the lowest error among all the tested methods. The parameters used in this neural network are as follows:

1. The value of the time window for prediction is considered to be 9.
2. We have a hidden LSTM layer with 100 units.
3. We used Adam as an optimization function, and the learning rate is variable. Initially, this value is equal to 0.01, and as the number of epochs increases, the learning coefficient decreases. In some cases, this value may reach 10^{-4}. A learning rate scheduler function is used in this part.
4. The mean absolute error is used as the loss function, which aims to minimize this value. Equation 6 shows the calculation of the mean absolute error.
$$\text{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} \quad (6)$$
where, y_i is the predicted value, x_i is the actual value of samples, and n is their total number.
5. The number of epochs is set to a maximum of 1200 but learning and changing the weights will continue until there is no progress in reducing the loss function in the last 100 cycles. So, in each data series, the number of epochs differs from others. In this section, the early-stopping function is used. Algorithm 4 represents the pseudo-code of this prediction.

Algorithm 4: Prediction

input: *Workload_list*

output: *W(t+1)*

```
1: w_i = workload_list.shift(+i)  {i = 1, 2, ..., T}
2: Normalize the workload dataset
3: Set RNN_LSTM parameters
4: Define call_back functions
5: for n_epochs and batch_size do:
6:     | Train RNN_LSTM model based on parameters and functions
7: end for
8: W(t+1) ← LSTM (w_i) {i = 1, 2, ..., T}
9: return W(t+1)
```

4.4. Calculate resources utilization

In this section, we calculate the resource utilization proportional to the workload calculated in each time window. Since the evaluation dataset is self-generated, we will also generate resource data. Yo et al. claim that the relationship between workload and resource utilization is linear [31]. To calculate the amount of resource utilization, first, it is determined what the maximum number of workloads a CPU or memory unit can handle. Then, it would be possible to determine how much of these resources each workload consumes.

4.5. Decision-making algorithm

In this section, Algorithm 5, the decision-making algorithm that uses the stated calculations to determine the workloads, the criteria, and the weights of the criteria, is presented. In this algorithm, previous components such as workload calculation, future workload prediction, and resource usage calculation are used to provide proper input. Then, after determining the scores of the microservices at each time, we obtain the Z-Score of each microservice. This is because in our experiments, scores demonstrate a normal distribution. Figure 7 shows a histogram plot of the scores obtained from the decision-making process for microservices in the first step of the test data. If this value is greater than 4, an anomaly or burst is assumed to be detected; This means that the volume of requests toward that service increases significantly. As a result, we allocate the maximum possible number of resources to that microservice.

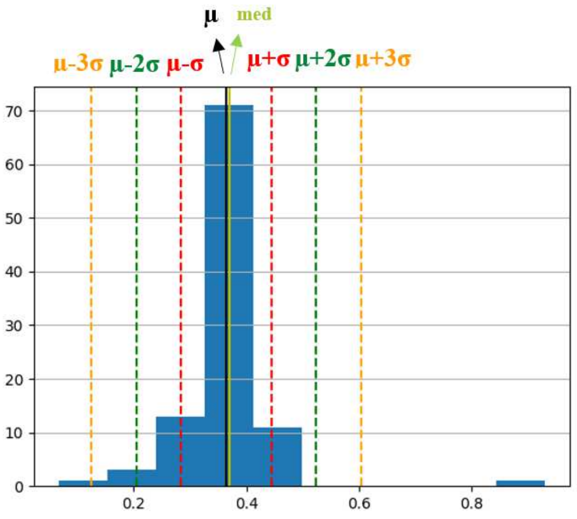


Figure 7. Histogram of the scores obtained from the decision-making component for microservices in the first step.

5. Evaluations

We evaluate the performance of the proposed approach using a simulation testbed. We introduce the dataset in section 5.1. Then, we discuss the performed evaluations, assumptions, and outcomes.

5.1. Environment and dataset

We have used the Python programming language to implement and evaluate the proposed algorithm. Codes and datasets are publicly available on a GitHub repository [40]. The proposed approach requires the workload and resource usage of the CPU and memory to make decisions. According to the previously discussed process, we first need a graph of relationships between microservices to obtain the workload. In this graph, nodes represent microservices, and edges represent functions or API calls between them. The relationship graph should be a directed acyclic graph. Two graph models were produced for the experiments. The first graph has 12 nodes and 16 edges. Node zero is assumed to be the API gateway. This graph is related to the online store named Acmezon [41], with minor changes. The initial reference graph is shown on the left side of Figure

8. This graph was converted to the one on the right side of the figure in the performed simulations.

```
Algorithm 5: Decision_making
input: Monitoring interval ($\xi$), window size ($WS$), list of microservices {1…m}
output: Microservices in need of scaling
1:  initialize criteria weights ($W_1$, $W_2$, $W_3$)
2:  is_burst = False
3:  $p^T \leftarrow$ node_score ($\xi$, $WS$)
4:  $W^T \leftarrow W_{\text{API}}^T * p^T$
5:  $W^{t+1} \leftarrow$ workload_prediction ($W^{t-WS}$ to $W^t$)
6:  $U_{CPU}^t \leftarrow \varphi_1(W^t)$
7:  $U_{memory}^t \leftarrow \varphi_2(W^t)$
8:  decision_matrix = $\begin{bmatrix} \Delta W^{(t+1,t)}(1) & \Delta W^{(t+1,t)}(2) & \dots & \Delta W^{(t+1,t)}(\text{m}) \\ U_{CPU}^t(1) & U_{CPU}^t(2) & \dots & U_{CPU}^t(\text{m}) \\ U_{mem}^t(1) & U_{mem}^t(2) & \dots & U_{mem}^t(\text{m}) \end{bmatrix}$
9:  for i in range 1 to the number of microservices do:
10:     Calculate $\mu_i$ from TOPSIS (decision_matrix)
11:     Add $\mu_i$ to score_list
12: end for
13: score_list ← sort (score_list)
14: for $score_i$ in score_list do:
15:     $z\_score_i = \frac{score_i - \mu}{\sigma}$
16:     if ($z\_score_i > 1$) then:
17:         add i to scale_out_list
18:         if ($z\_score_i > 4$) then:
19:             is_burst = True
20:     else if ($z\_score_i < -3$) then:
21:         add i to scale_in_list
22: end for
23: for each microservice in scale_out_list or scale_in_list:
24:     $n' \leftarrow \varphi(U_{CPU}^{(t+1)}, U_{mem}^{(t+1)})$, $n \leftarrow \varphi(U_{CPU}^{(t)}, U_{mem}^{(t)})$
25: if (is_burst == True) then:
26:     $n' \leftarrow n_{max}$
27: return microservices in need of scaling and ($n' - n$) for them
```

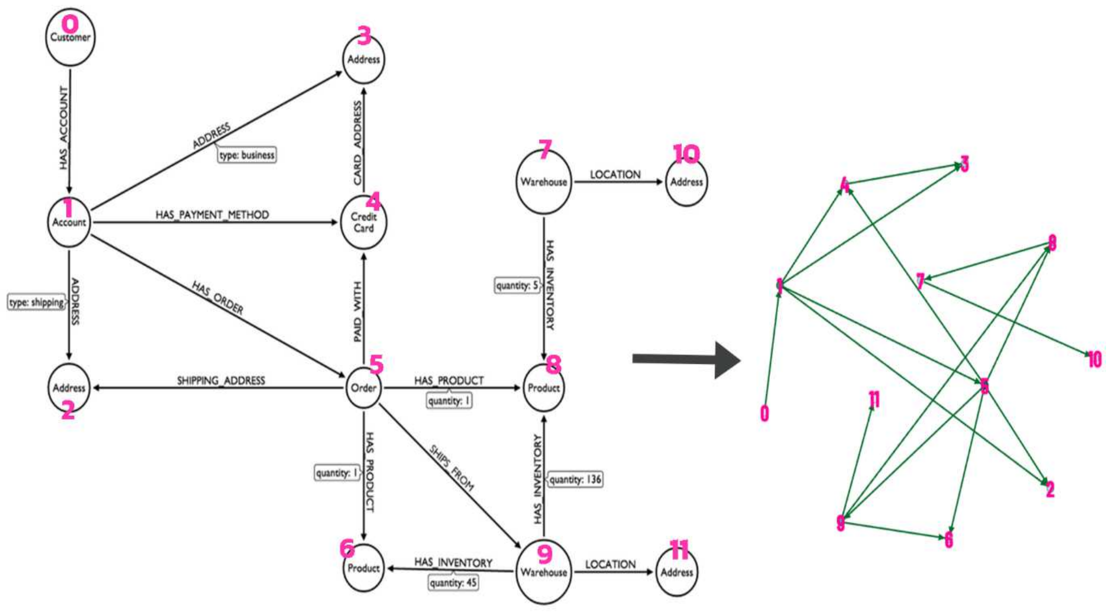


Figure 8. The structure of the first microservices' relationships graph (i.e. small graph).

A second graph, with 101 nodes and 406 edges, was also generated for testing on larger scales. This graph also has the characteristics of a relationships graph in a microservice-based architecture. The last node represents the API gateway. This graph is shown in Figure 9. Because in these programs, each of the microservices has its specific task, each may have different resource usage patterns. Services with heavy calculations have high CPU usage, and services that work with large data, such as those connected to databases, have high memory resource usage.

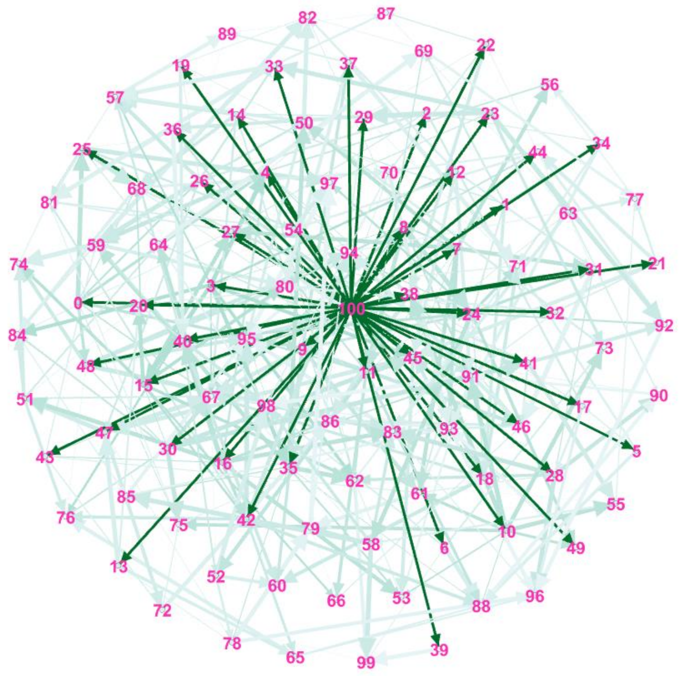


Figure 9. The structure of the second graph of microservices relationships (i.e. large graph).

Therefore, to make the model more accurate and closer to reality, in both graphs, it was considered that some nodes (i.e., nodes with even numbers) are related to microservices with high processor consumption and a small amount of memory, and others (odd-numbered nodes) have high memory consumption and a small amount of CPU usage.

5.2. Experiments and results

The experiments from calculating the workloads of all microservices to making the final decision and selecting the microservices that require scaling are summarized in the following steps:

1. Finding the frequency of function calls between microservices.
2. Calculate the probability of calling a microservice by another in each time window.
3. Calculating the probability of workloads entering each microservice in each time step.
4. Finding the hesitant fuzzy set from the microservices' workload probability in multiple time steps and calculating the score function in each time window.
5. Calculating the workload of each microservice by assigning probabilities to the created graph and based on the workload toward the API gateway.
6. Predicting future workloads.
7. Making decisions based on the predicted values and the current state of nodes.

5.3. Evaluations and comparisons

To check the accuracy and performance of the proposed model, the scenarios discussed in this section have been considered.

5.3.1. Evaluation of calculating microservices' workloads

The current tools lack the ability to identify the specific microservices that require scaling before initiating the scaling process. Consequently, these tools are not well-suited for microservice systems. As modern microservice applications continue to increase in scale and complexity, it becomes increasingly challenging to pinpoint the under-provisioned or over-provisioned microservices. Moreover, making effective scaling decisions to optimize resource utilization poses its own set of challenges [31]. In all similar research, it has been assumed that we already have the available workload for all microservices. It means that a monitoring tool has been placed separately for each of the microservices, and these tools also consume resources. Moreover, it will take time to monitor them individually, and as mentioned, it is not easy to monitor them at the service level. So, we provided a probabilistic mechanism to obtain these metrics previously discussed. In Figure 10, we can see the diagram of the workload calculated using the proposed approach and the actual workload of node 0 in the final 300 samples, which has increased significantly in the final time windows. It is also evident that these values are very close to each other.

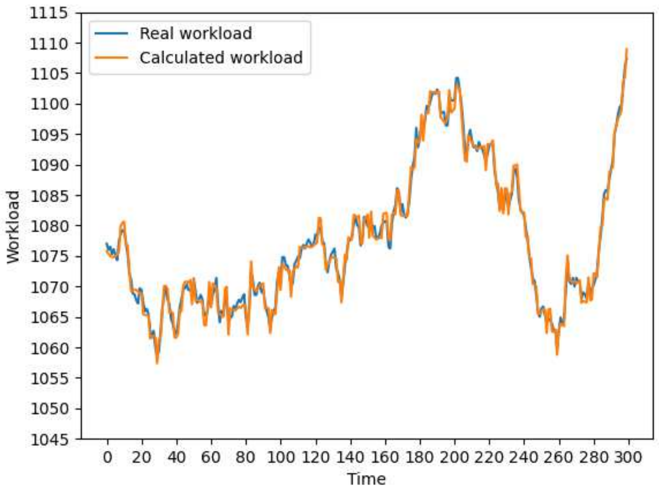
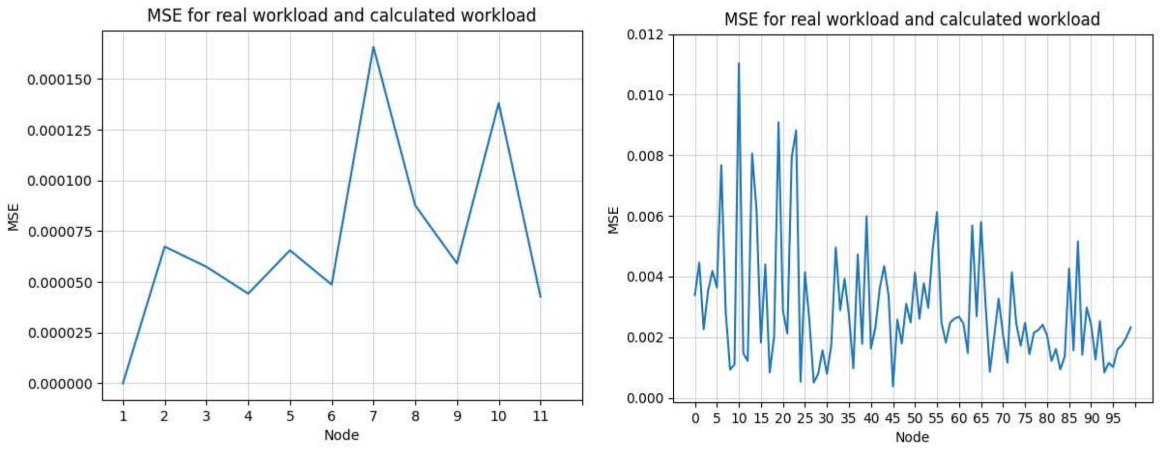


Figure 10. Comparison of the actual and calculated workload of node 0 using the proposed method in the last 300 time windows.

The diagrams in Figure 11, show the mean squared error of the actual workload values and the calculated workload of all nodes in the last 300 time windows. The first graph is for the small graph, and the second one is for the large graph.



(a) **(b)**

Figure 11. The mean squared error of real and calculated workloads using the proposed method. (a) first dataset (b) second dataset.

5.3.2. Evaluation of predicted workload

In this section, we compare prediction accuracy in the proposed approach with actual data and other common machine-learning approaches. To evaluate the predicted values, considering that the information about the actual amount of input workload at the time $(t+1)$ is available, we can compare this value with the predicted value in the proposed approach. We have used RNN LSTM-based prediction model. In this experiment, we used a time window of length 9, which means that each value is predicted according to its previous nine values in the past time windows. Time window lengths of 3 to 9 were tested, and 9 generated the best results.

Also, 80% of the data have been used for training and 20% for testing. This method, with all configurations proposed in section 4, can fit all data of all nodes in the performed experiments. The graphs in Figure 12 show the actual and predicted values by the LSTM method for node 0 in the large graph.

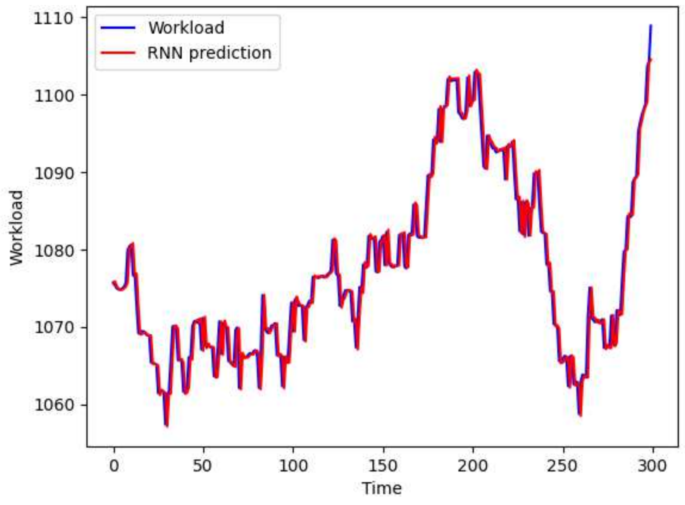


Figure 12. Comparison of real workload and predicted value for node 0.

Table 4 compares the mean squared error between the prediction method for the initial nodes of the large graph. We have determined the most suitable parameters for each tested method to get the best results.

Table 4. Comparison of mean square error of workload prediction using machine learning methods, in the first 5 nodes.

	MSE in node 0	**MSE in node 1**	**MSE in node 2**	**MSE in node 3**	**MSE in node 4**
KNN	21.1953	11.1705	8.2620	11.4717	6.6061
LR	2.5034	2.3651	2.5867	2.5568	3.0523
Lasso	2.4843	2.3416	2.6891	2.5221	3.1397
LassoCV	2.48200	2.322	2.6340	2.5178	3.0558
Ridge	2.50349	2.3651	2.5865	2.5564	3.0523
Elasticnet	2.5015	2.3499	2.5909	2.5392	3.0456
Polynomial	3.2366	2.7	2.9118	2.7371	3.0907
DT	20.9337	11.1686	9.6611	11.0145	5.6716
RF	21.2230	11.3885	8.3678	10.6923	4.9710
XGBoost	21.5627	11.3950	7.9388	10.9317	4.3101
SVR	93.7860	36.2896	23.1080	6.6929	8.4130
LSTM	2.4991	2.334	2.5921	2.5206	2.9894

5.3.3. Evaluation of decision-making and selecting microservices

One of the objectives outlined in this study is to reach a compromise according to the criteria defined in the previous sections to decide between the nodes that need scaling. The parameters and assumptions used in the decision-making section are as follows:

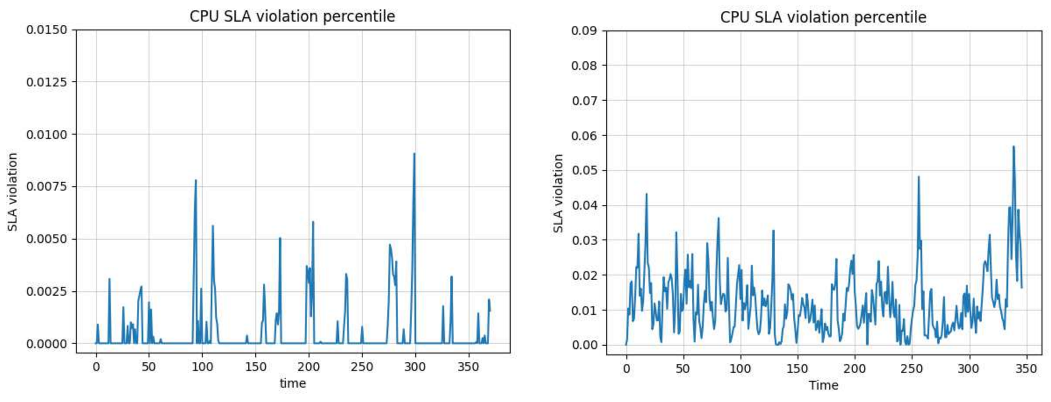
1. The alternatives are the nodes or microservices.
2. The criteria for decision-making are the amount of workload added to the microservice, the amount of CPU usage, and the amount of memory usage.
3. The weights considered for each criterion differ according to whether the node is memory-intensive or CPU-intensive. This mechanism has also been put in place so that when the usage amount of each resource exceeds the threshold, and the agreements are violated, the weight of that criterion is doubled to give it more importance.
4. The CPU and memory threshold values mentioned in the customer service level agreement are assumed to be 90%.

In each time window, the selected nodes may differ from those in need of scaling. Because firstly, the forecasted workload may differ slightly from the actual amount. Second, in the decision-making process, we only select nodes for scaling whose Z-Score is greater than 1, and we do not care about nodes with a lower score. Although some of these nodes have high resource utilization, the added workload is usually low based on experiments. So, they have earned smaller scores. These nodes may have violated the agreements due to resource consumption exceeding the threshold, but their workload may be reduced again in subsequent time windows. However, if the agreements are violated again in the future time windows, the scaling of these services will be prioritized. So, we may skip the scaling of cases that violate the threshold with a slight difference at first, but in the next time steps, if this amount increases, the importance of changing its scale becomes higher. As a result, we have reduced the number of scaling times and unwanted overhead while the efficiency is not decreased. This feature is the main difference between the proposed mechanism and approaches that do not have a conscious choice in the decision-making phase.

The diagrams in Figure 13 show the average percentage of SLA violations in CPU usage across all microservices at any time. Each point in these graphs shows the average percentage of SLA violations in all nodes. In parts (a) and (b), these values are shown for the small and large graphs respectively. As is evident from the figures, these values are very small.

In the diagrams of Figure 14, we see the average SLA violations for using memory in the small and large graphs, respectively in the test data. To compare the decision-making component of the proposed approach with other decision-making methods in similar studies, these studies can be analyzed from a reactive or proactive viewpoint in general without having a microservice-focus in their approach.

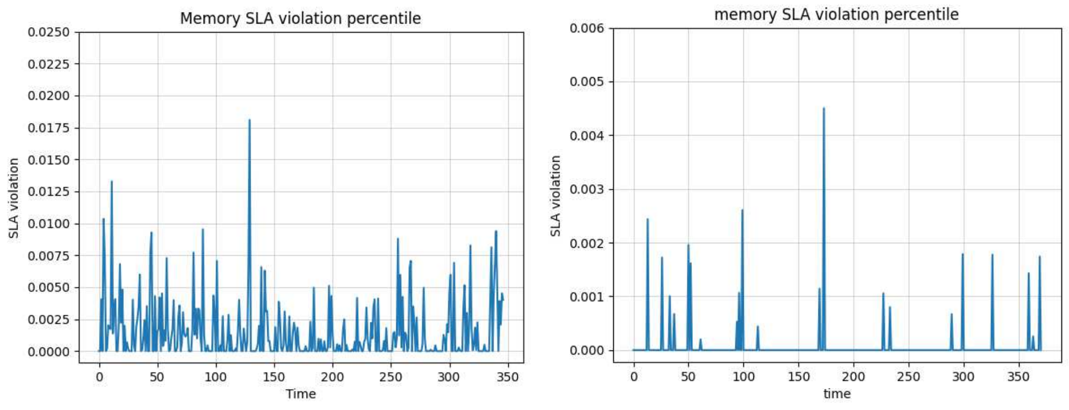
The default auto-scaling algorithm used in Kubernetes is reactive. Since prediction does not exist in this method, a decision is made when the agreement has been violated. Therefore, the amount of SLA violations in this approach is very high. On the other hand, the accurate and correct determination of the threshold limits can never correctly occur. When the workload fluctuations are high, the scale increases and decreases continuously, creating overhead for the system every time. This reactive method is used in [19] [28] [29] [30] [42].



(a) **(b)**

Figure 13. The average percentage of SLA violations of microservices' CPU utilization in each time window. (a) first dataset (b) second dataset.

In proactive methods without a focus on microservices, the workload or the amount of resource usage is predicted first. Then, they calculate the resources needed proportional to the predicted workload for all node's resources for the decision-making process. This method is used in [10] [20] [21] [24]. The hybrid approaches predict the workload, but rule-based or reactive methods are used to make decisions. For example, the rule is set such that if the usage amount of the predicted resources is more than 70%, additional resource is added, and if it is less than 30%, resources are reduced. Researchers use this method in [22] [26] [33] [25]. In none of the approaches presented in similar research, a focus on microservices exist and also no prioritization is performed when a large number of services exist in the environment each having different workload patterns or resource demands. However, in the proposed approach, we prioritize node scaling by setting scores for them. By calculating the Z-Score criterion, we select the necessary nodes for changing the scale.



(a) **(b)**

Figure 14. The average percentage of SLA violations of microservices' memory utilization in each time window. (a) first dataset (b) second dataset.

When there is a limitation on the amount of available computational resources, proper prioritization can allocate those limited resources to more sensitive services first. Figure 15 compares the decision-making in the proposed approach and hybrid approaches without any focus on selecting microservices. As it is clear, the number of allocated resources and subsequently the costs of hybrid approaches are significantly larger. This difference becomes more notable in the case of a large number of microservices. As an average, the total number of resources in the proposed approach has decreased by 1.67%.

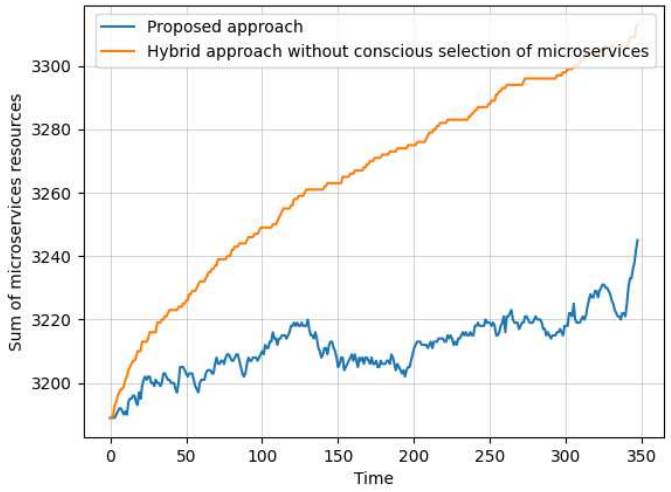
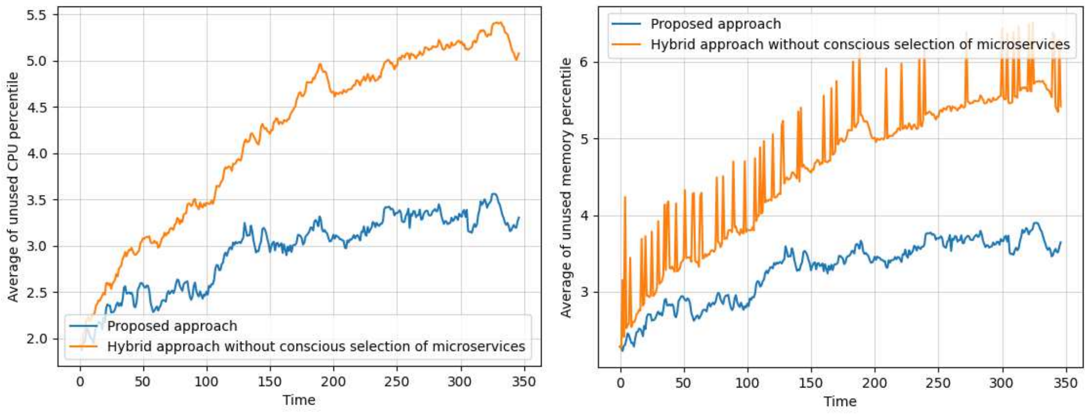


Figure 15. Comparison of the total allocated resources between the proposed approach with a focus on microservice scaling and hybrid approaches without any awareness of microservice existence.

As we see in Figure 15, in hybrid approaches without microservice-awareness, there is a continuous increase in the number of resources, leading to a rise in unused resources. It is also worth mentioning that the overall workload of the input to the system is increasing. Consequently, with each increase in resource allocation, the amount of unused resources also grows. On average, the amount of unused CPU and memory in the proposed approach has decreased over the hybrid approach without considering microservices by 24.09% and 21.37%, respectively (shown in Figure 16).



(a) **(b)**

Figure 16. Comparison of the the average percentage of unused (a) CPU (b) memory, in the proposed approach and the hybrid approach without considering the microservices.

In most proactive solutions, the resources needed in the next steps are calculated, and the necessary resources are allocated. In these cases, the average number of resources may decrease, but the number of scaling will increase compared to the proposed approach which will result in a significantly larger scaling-overhead. The total number of scales in each time step in the proposed approach and the proactive methods without focus on microservices are shown in Figure 17. This amount has improved by 69.82% on average in the proposed method.

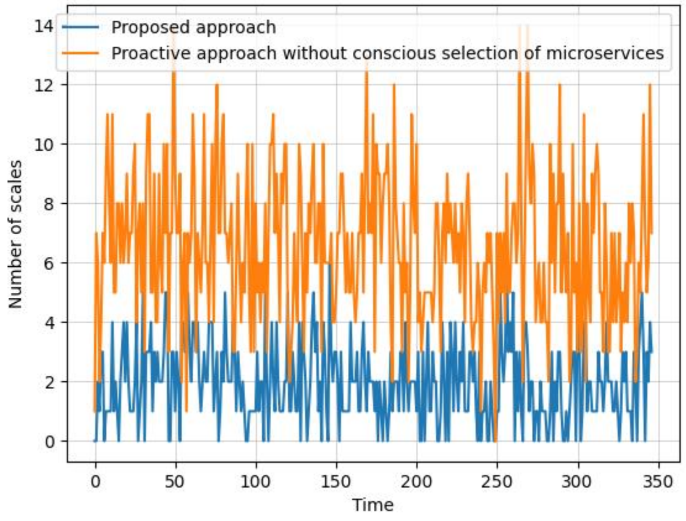


Figure 17. Comparison of the total number of scales in the proposed approach and the proactive approach without considering the microservices.

5.3.4. Sensitivity of the model to the chosen criteria

In this section, we design a scenario to measure the correctness of the proposed mechanism in several situations. For example, suppose that in a period, the requests for a microservice increase. In this case, all the nodes that are placed in the path between the starting node and the stated node may be affected, and their workload will increase as well. This means that a request may have to go through a path of nodes to reach the destination node. Of course, it should also be checked what other services are affected by these nodes. So, the auto-scaler should prioritize its scaling, predict the workload before the event occurs, and increase the resources to the extent required. For each scenario, the following values are reported:

1. Time step.
2. Node number.
3. Workload sequence over the past 10 time-steps for microservices that have decided to scale in that time frame.
4. The average percentage of processor and memory resource consumption at the present time (t).
5. The actual workload at the future time $(t + 1)$.
6. The expected amount of workload in the future time $(t + 1)$.
7. Scaling decision.
8. Reason for decision.

The results of the tests performed in the large graph with 100 nodes and the values and decisions related to the last two time-steps are listed in Table 5. In the first time-step, five nodes need to be scaled up. In this Table, for the nodes where the resource consumption rate is more than 90%, the weight of the relevant criteria will increase, and the scale will increase as well.

As stated earlier, even-numbered nodes have high CPU consumption, and odd-numbered nodes have high memory consumption. So, as we can see in the Table, according to the 90% threshold limit set for resources, the even nodes have used almost the maximum amount of their processor resources, and the odd nodes have also used the maximum amount of memory allocated to them. Therefore, in some nodes, due to the high amount of resource usage (nearly 90%) and also the significant increase in the workload, it has been decided to increase the scale.

Table 5. Test results of checking the sensitivity of the large graph model in the last two time steps.

Time step	Node number	Last 10 workload sequence	CPU utilization	Memory utilization	The real workload in time (t+1)	The predicted workload in time (t+1)	Auto-scaler decision	Reason
1	9	2734.805, 2735.968, 2737.164, 2733.831, 2746.121, 2753.229, 269.819, 2773.769, 2787.71, 2785.576	89.857	89.857	2791.464	2790.012	Scale-out	Added workload & high resource utilization
	28	2577.08, 2578.123, 2583.3, 2583.976, 2587.925, 2594.857, 2597.177, 2607.017, 2612.715, 2614.532	90.782	90.056	2620.308	2618.697	Scale-out	SLA violation
	15	1820.506, 1821.28, 1826.126, 1826.604, 1833.452, 1835.486, 1837.126, 1838.333, 1843.551, 1848.905	90.19	90.19	1846.033	1848.847	Scale-out	SLA violation
	12	1674.866, 1675.578, 1676.31, 1676.749, 1679.312, 1689.297, 1694.871, 1695.984, 1705.183, 1710.441	90.98	90.253	1703.404	1705.64	Scale-out	SLA violation
2	45	5961.658, 5964.265, 5969.876, 5991.169, 6001.877, 6007.242, 6031.524, 6047.499, 6055.777, 6059.718	89.773	89.773	6074.595	6075.619	Scale-out	Added workload & high resource utilization
	27	3670.892, 3664.399, 3669.408, 3679.073, 3687.215, 3686.447, 3705.137, 3719.660, 3730.391, 3749.119	90.34	90.34	3758.911	3758.097	Scale-out	SLA violation
	30	2246.246, 2247.228, 2251.867, 2263.421, 2269.992, 2272.022, 2281.648, 2283.373, 2284.661, 2290.223	90.881	90.15	2295.3	2294.507	Scale-out	SLA violation
	36	1015.869, 1016.314, 1016.580, 1022.19, 1015.202, 1007.981, 996.442, 988.925, 981.467, 969.88	86.596	85.903	962.15	964.891	Scale-in	Reduced workload

5.3.5 Influence of complex networks criteria on the decision-making

By considering the mathematical framework of complex networks, a set of criteria can be used to extract interesting features from the desired graph. One of these criteria is the centrality measurement. This measure can help us understand how vital each node is or to find essential nodes among all existing nodes. Well-known centrality measures are degree centrality, closeness, and betweenness. Using the conducted experiments, we concluded that more important nodes have more resource requirements. Finally, by finding the more critical nodes, we find the nodes that may face more changes and therefore need more scaling.

In our experimental analysis, we observed that node 45 exhibited more frequent changes than other nodes during multiple periods. Furthermore, the workload directed towards node 45 was consistently higher than that of other nodes. As depicted in Table 5, subsequent observations indicated that additional nodes also required changes during specific time windows. Specifically, a subset of microservices experienced significant fluctuations compared to the remaining nodes. Figure 18 represents a portion of the comprehensive graph illustrating the interconnections among microservices, with the majority of these nodes present within this subset. This figure displays a subgraph showcasing all potential pathways for the workload to traverse from the API gateway (node 100) to node 45. Nodes such as 24, 15, 30, 18, 8, and others are located along this pathway. Intriguingly, in the later time steps, as the workload directed towards node 45 increased, the majority of these nodes along the pathway also exhibited an upward trend in workload and necessitated scaling up.

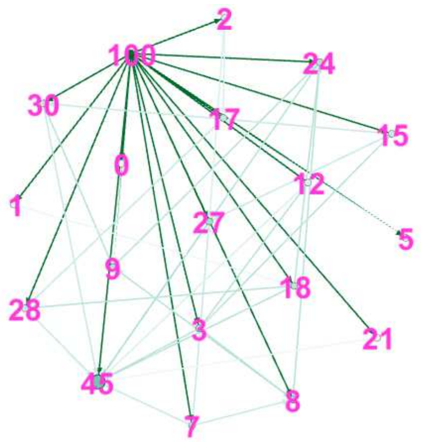


Figure 18. A subgraph of the larger graph consisting of all the workload paths ending in node 45.

The colorization of nodes in Figure 19 is performed based on implementing a greedy algorithm to find communities detected in this graph. Here, six communities have been found. As it is clear, the nodes mentioned in Figure 18 were all detected in the same group. This observation suggests that nodes belonging to the same community generally exhibit similar patterns of workload fluctuations and experience similar workload increases or decreases. Node degree is a crucial centrality measure because we consider input workload as an input edge to a node. To evaluate this criterion, we focused on examining the input degree.

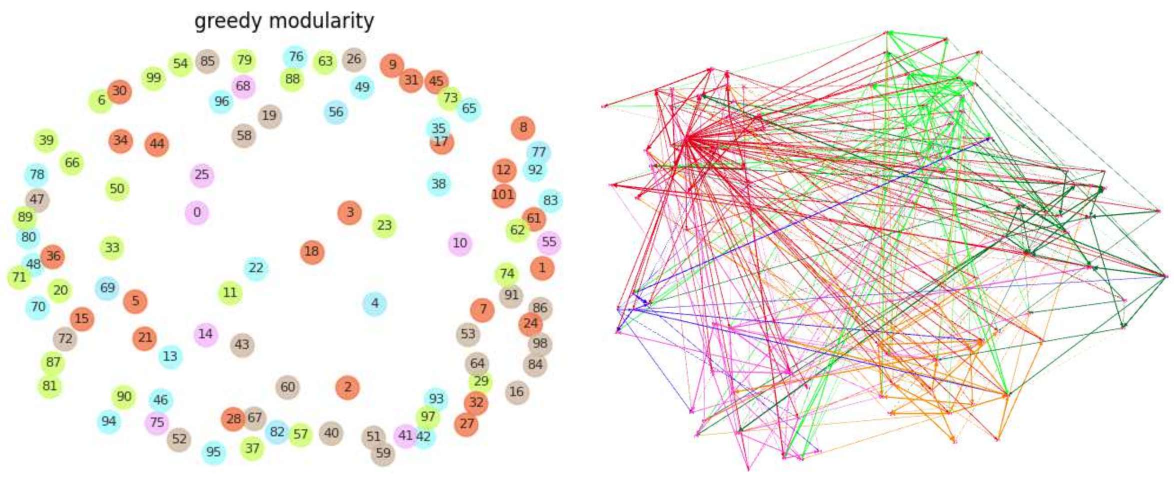


Figure 19. The communities detected in a large graph by the greedy modularity algorithm.

The performed analysis revealed that node 45 exhibited high centrality, ranking fourth in terms of this measure. This indicates that the node receives a substantial number of incoming edges and can be accessed through multiple routes. Consequently, it holds a critical position within the network. Notably, node 45 is associated with a microservice that is utilized by numerous other microservices. Therefore, nodes with high input degree centrality often correspond to nodes with significant input workloads, indicating a higher need for scaling.

$$\text{In\_degree\_centrality} = \{(95, 0.11), (67, 0.1), (93, 0.1), (45, 0.09), \dots\}$$

Another criterion is the betweenness centrality, which indicates how much a node is located in the shortest path from one node to another. This value is calculated by Equation 7.

$$C_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t\,|v)}{\sigma(s,t)} \tag{7}$$

In this Equation, the betweenness centrality of node v is the total fraction of all pairs of paths between s and t that pass through node v to all paths between s and t. By measuring this value, we observed that node 45 obtained the second rank. An interesting observation is that other nodes, such as nodes 24 and 28, which mainly obtained a high rank in scores, also obtained a high value in this criterion. This means that when multiple microservices call each other in different paths, nodes with high betweenness centrality are often seen among these paths and are considered a bridge to reach other nodes. So, this criterion is also one of the important criteria for determining essential nodes.

$$\text{Betweenness\_centrality} = \{(47, 0.0222), (45, 0.0156), (51, 0.0153), (41, 0.0125), \dots\}$$

As we have seen, according to the analysis related to the network, it is possible to find a subgraph of the complex and large microservices' relationship graph, in which nodes receive a significant number of user requests and have numerous references. Consequently, we can prioritize monitoring and scaling of these services. However, it should be mentioned that in addition to the characteristics of the network, the distribution and dispersion of workload among nodes at any given time also play a significant role in this determination. Therefore, it is essential to consider both the network characteristics and the workload dispatching mechanisms discussed earlier when making informed decisions.

6. Threats to validity

To apply the proposed mechanism in a real-world scenario, there may be some challenges and limitations. The first issue is that in the proposed approach a model with specific parameters to predict the incoming workload is created which is used to predict the workload of all microservices. The more the number of microservices with different patterns, the more difficult it is to find a suitable model that does not overfit or underfit any of them.

The second limitation of the proposed approach is to determine the exact number of microservices that need to be scaled using their scores in the multi-criteria decision-making component. Some services may need to be scaled but are not selected by the decision algorithm due to earning a lower score. For example, let's assume that by setting the value of 90% for the threshold of resources utilization, in a time interval, this amount for a microservice is 89.9%, and in the next time window, it is 90.01%. Here, this microservice has violated the threshold limit. It may need to be scaled as a general rule. However, it scored very little because the workload added to it was very small and may decrease again in a later time window. So, as we hypothesized, scaling, in this case, can be ignored to avoid additional scaling overhead. Therefore, we give more importance to scaling microservices with high scores in all three decision-making criteria. Another challenge is when in an application with a large number of microservices, the workload of all microservices suddenly increases significantly. In this case, the proposed approach only selects some microservices for scaling which are more critical, and therefore scaling other microservices is deferred to subsequent time steps. This issue may cause overall performance degradation. Therefore, considering the appropriate approach in such particular circumstances is one of the improvements that can be investigated in future works. Additionally, to simplify the proposed approach, we assumed that each of the microservices had enough resources in their containers to be scaled. In this case, scaling is only performed based on load fluctuations. So, in cases where a microservice belonging to a specific container does not have additional resources to be scaled, considering the migration of microservices to other containers or even changing the physical machine might be needed.

7. Conclusions

Auto-scaling is a major feature of cloud computing that enables us to allocate resources to applications on demand. On the other hand, microservice architecture is becoming popular because of scalability, fault tolerance, high availability, domain isolation, and simple debugging. However, in applications with many microservices, auto-scaling becomes a challenge. Microservices interact with each other over remote procedure calls (RPC) or RESTful APIs, and workload flows between microservices. Most of the time, they have complex relationships, and if customers report latency in the system, finding the microservice causing such a violation is almost impossible. In addition, we cannot monitor metrics such as input workload or resource usage at the service level.

In this paper, we introduced a mathematical model to calculate the probability of input workload in each time step toward individual microservices based on their relationships graph. Then, by predicting the workload besides calculating other metrics such as CPU and memory utilization, we have proposed an algorithm to perform scaling for candidate microservices in each time window. Finally, we compared and evaluated the proposed algorithm by designing several experiments. In these experiments, it is assumed that initially, we have several function calls between microservices. This data can be generated by randomly calling neighbor nodes for each node and obtaining the call rate of each microservice by neighboring nodes for every time interval. In the end, a multi-criteria decision-making component will make the final decision based on the predicted incoming workload and current resource utilization. The outcomes of the performed evaluations demonstrate that the proposed approach improves the correct decision ratio for scaling, decreases the resource scaling cost, and increases utilization compared to existing approaches (e.g. hybrid approaches).

In future expansions, we can consider the history of customer requests, which specifies the requested address or URI. Additionally, we can store what path is taken and what microservices will be involved in each request from a specific source address. Therefore, using this information, we can predict the load of each microservice more accurately. In the threats to validity section, it is mentioned that the system's workload may increase in a time window for all microservices. Therefore, all or a large number of them need to be scaled. However, the proposed method selects some of the most sensitive components for scaling, and scaling for other items will be postponed to the next time steps. This method is efficient when resources and time are limited. Otherwise, an approach should be proposed to recognize such conditions and take appropriate action.

8. Ethics approval and consent to participate

This article does not contain any studies with human participants or animals performed by any of the authors.

9. Availability of data and material

Raw data was generated for this research work. Derived data supporting the findings of this study are publicly available through a GitHub repository which is referenced in the paper.

10. Competing interests

All of the authors declare that they have no conflict of interest.

11. Funding

The authors did not receive support from any organization for the submitted work.

12. Authors' contributions

Matineh ZargarAzad: Conceptualization, Methodology, Software, Visualization and Original draft preparation.
Mehrdad Ashtiani: Conceptualization, Writing- Reviewing and Editing, Verification of the results, Supervision.

13. Acknowledgments

Not Applicable.

References

[1] S. K. Sarma, "Metaheuristic based auto-scaling for microservices in cloud environment: a new container-aware application scheduling," *International Journal of Pervasive Computing and Communications,* vol. 19, no. 1, pp. 74-76, 2023.

[2] N. Marie-Magdeline and T. Ahmed, "Proactive Autoscaling for Cloud-Native Applications using Machine Learnong," in *Proceedings of GLOBECOM 2020-2020 IEEE Global Communications Conference*, Taipei, Taiwan, pp.1-7, 2020.

[3] E. G. Radhika, G. Sudha Sadasivam and J. Fenila Naomi, "An efficient predictive technique to autoscale the resources for web applications in private cloud," in *Proceedings of 2018 Fourth International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB)*, IEEE, Chennai, India, pp.1-7., 2018.

[4] W. Iqbal, A. Erradi and A. Mahmood, "Dynamic workload patterns prediction for proactive auto-scaling of web applications," *Journal of Network and Computer Applications,* vol. 124, pp. 94-107, 2018.

[5] B. Liu, R. Buyya and A. Toosi, "A fuzzy-based auto-scaler for web applications in cloud computing environments," in *Proceedings of International Conference on Service-Oriented Computing*, 2018, Springer, Cham, pp.797-811, 2018.

[6] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam and A. Leon-Garcia, "Elascale: Autoscaling and monitoring as a service," *arXiv preprint arXiv:1711.03204,* 8 NOV 2017.

[7] "NIST, National Institute of Standards and Technology," [Online]. Available: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf. [Accessed April 2023].

[8] J. Benifa and D. Dejey, "Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment," *Mobile Networks and Applications,* vol. 24, no. 4, pp. 1348-1363, 2019.

[9] V. Messias, J. C. Estrella, R. Ehlers, M. J. Santana, R. C. Santana and S. Reiff-Marganiec, "Combining time series prediction models using genetic algorithm to autoscaling web applications hosted in the cloud infrastructure," *Neural Computing and Applications,* vol. 27, no. 8, pp. p.2383-2406, 2016.

[10] L. Wang, G. Von Laszewski, A. Younge, X. He, M. Kunze, J. Tao and C. Fu, "Cloud computing: a perspective study," *New Generation Computing,* vol. 28, no. 2, pp. 137-146, 2010.

[11] Q. Zhang, L. Cheng and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications,* vol. 1, no. 1, pp. 7-18, 2010.

[12] "Xen. Xen hypervisor," [Online]. Available: URL: http://www.xensource.com. [Accessed 2023 April].

[13] "KVM. Kernel Based Virtual Machine," [Online]. Available: https://www.linux-kvm.org/page/Main_Page. [Accessed April 2023].

[14] "ESX, V. VMWare ESX Server," [Online]. Available: https://www.vmware.com/products/esx. [Accessed April 2023].

[15] I. Odun-Ayo, M. Ananya, F. Agono and R. Goddy-Worlu, "Cloud computing architecture: A critical analysis," in *Proceedings of 2018 18th International Conference on Computational Science and Applications (ICCSA)*, IEEE, Australia, pp.1-4, 2018.

[16] J. Novak, S. Kasera and R. Stutsman, "Auto-Scaling Cloud-Based Memory-Intensive Applications," in *Proceedings of 2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, IEEE, Beijing, China, pp.229-237, 2020.

[17] G. YU and D. Christiana, "The Architectural Implications of Cloud Microservices," *IEEE Computer Architecture Letters,* vol. 17, no. 2, pp. 155-158, 2018.

[18] A. U. Gias, C. Giuliano and W. Murray, "ATOM: Model-driven autoscaling for microservices," in *Proceedings of 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS).*, IEEE, 07-10 July 2019, Dallas, TX, USA, 2019.

[19] C. K. Rudrabhatla, "A quantitative approach for estimating the scaling thresholds and step policies in a distributed microservice architecture," *IEEE Access,* vol. 8, pp. 180246 - 180254, 2020.

[20] M. Abdullah, W. Iqbal, A. Mahmood, F. Bukhari and A. Erradi, "Predictive autoscaling of microservices hosted in fog microdata center," *IEEE Systems Journal,* vol. 15, no. 1, pp. 1275-1286, 2020.

[21] M. Abdullah, W. Iqbal, J. Berral, J. Polo and D. Carrera, "Burst-aware predictive autoscaling for containerized microservices," *IEEE Transactions on Services Computing,* vol. 15, no. 3, pp. 1448-1460, 2020.

[22] R. Bouabdallah, S. Lajmi and K. Ghedira, "Use of reactive and proactive elasticity to adjust resources provisioning in the cloud provider," in *Proceedings of 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, IEEE, 12-14 December 2016, Sydney, NSW, Australia, 2016.

[23] A. Qassem and M. Lamees, "Proactive Random-Forest Autoscaler for Microservice Resource Allocation," *IEEE Access,* vol. 11, pp. 2570-2585, 2023.

[24] I. Prachitmutita, W. Aittinonmongkol, N. Pojjanasuksakul, M. Supattatham and P. Padungweang, "Auto-scaling microservices on IaaS under SLA with cost-effective framework," in *Proceedings of 2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)*, IEEE, Xiamen, China, 2018.

[25] E. G. Radhika, G. Sudha Sadasivam and J. Fenila Naomi, "A RNN-LSTM based Predictive Autoscaling Approach on Private Cloud," *International Journal of Scientific Research in Computer Science, Engineering and Information Technology,* vol. 3, no. 3, 2018.

[26] M. S. Aslanpour, M. Ghobaei-Arani and A. N. Toosi, "Auto-scaling web applications in clouds: A cost-aware approach," *Journal of Network and Computer Applications,* vol. 95, pp. 26-41, 2017.

[27] S. Srirama, M. Adhikari and S. Paul, "Application deployment using containers with auto-scaling for microservices in cloud environment," *Journal of Network and Computer Applications,* vol. 160, p. 102629, 2020.

[28] A. Khaleq and I. Ra, "Agnostic approach for microservices autoscaling in cloud applications," in *Proceedings of 2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, IEEE, 05-07 December 2019, Las Vegas, NV, USA, pp. 1411-1415, 2019.

[29] f. Zhang, X. Tang, X. Li, S. U. Khan and Z. Li, "Quantifying cloud elasticity with container-based autoscaling," *Future Generation Computer Systems,* vol. 98, pp. 672-681, 2019.

[30] A. Shahidinejad, M. Ghobaei-Arani and M. Masdari, "Resource provisioning using workload clustering in cloud computing environment: a hybrid approach," *Journal of Cluster Computing,* vol. 24, no. 1, pp. 319-342, 2021.

[31] G. Yu, P. Chen and Z. Zheng., "Microscaler: Automatic scaling for microservices with an online learning approach," in *Proceedings of 2019 IEEE International Conference on Web Services (ICWS)*, IEEE, Milan, Italy, pp. 68-75, 2019.

[32] B. Zhu, "Decision method for research and application based on preference relation," Southeast University, Nanjing, 2014.

[33] Y. Li and Y. Xia., "Auto-scaling web applications in hybrid cloud based on docker," in *Proceedings of 2016 5th International Conference on Computer Science and Network Technology (ICCSNT)*, IEEE, Changchun, China, pp.75-79, 2016.

[34] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk and Iyer R. K., "FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices," in *Proceedings of The 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, USENIX Association, 2020 NOV.

[35] R. Krzysztof, F. Paweł, Ś. Jacek, Z. Przemyslaw, B. Przemyslaw, K. Jarek, N. Paweł Krzysztof , S. Beata, W. Piotr, H. Steven and W. John, "Autopilot: Workload Autoscaling at Google Scale," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ACM, Heraklion Greece, pp. 1-16, April-2020.

[36] L. Qian, L. Bin, M. Pietro, I. Ramesh, T. Charlie, K. Michael and K. Christos, "RAMBO: Resource Allocation for Microservices Using Bayesian Optimization," *IEEE Computer Architecture Letters,* vol. 20, no. 1, p. 46–49, 2021.

[37] J. Park, B. Choi, C. Lee and D. Han, "GRAF: A graph neural network based proactive resource allocation framework for SLO-oriented microservices," in *Proceedings of the 17th International Conference on emerging Networking Experiments and Technologies*, ACM, Virtual Event Germany, pp. 154-167, December-2021.

[38] X. Nguyen, H. S. ZhuLiu, and M. Liu,, "Graph-PHPA: Graph-based Proactive Horizontal Pod Autoscaling for Microservices using LSTM-GNN," in *Proceedings of 2022 IEEE 11th International Conference on Cloud Networking (CloudNet), pp. 237-241*, IEEE, Paris, France, November-2022.

[39] J. Li, J. Q. Wang and J. H. Hu, "Multi-criteria decision-making method based on dominance degree and BWM with probabilistic hesitant fuzzy information," *International Journal of Machine Learning and Cybernetics,* vol. 10, pp. 1671-1685, 2019.

[40] [Online]. Available: https://github.com/matin96/Thesis-automicro.git.

[41] "Online Storefront of Domain Model of Acmezon," [Online]. Available: https://gist.github.com/kbastani/4f1e5fe25088209dcc26. [Accessed May 2023].

[42] N. Roy, A. Dubey and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Proceedings of 2011 IEEE 4th International Conference on Cloud Computing*, IEEE, Washington, DC, USA, Jul-2011.