Received 10 November 2023; accepted 3 December 2023. Date of publication 8 December 2023; date of current version 29 December 2023.

Digital Object Identifier 10.1109/OJCOMS.2023.3341002

Proactive and Intelligent Monitoring and Orchestration of Cloud-Native IP Multimedia Subsystem

RASEL CHOWDHURY [1], CHAMSEDDINE TALHI [1], HAKIMA OULD-SLIMANE [2], AND AZZAM MOURAD [3,4] (Senior Member, IEEE)

[1]Department of Software Engineering and Information Technology, École de technologie supérieure, Montreal, QC H3C 1K3, Canada
[2]Department of Mathematics and Computer Science, Université du Québec à Trois-Rivières, Trois-Rivières, QC G8Z 4M3, Canada
[3]Artificial Intelligence and Cyber Systems Research Center, Department of CSM, Lebanese American University, Beirut 1102 2801, Lebanon
[4]Division of Science, New York University, Abu Dhabi, UAE

CORRESPONDING AUTHOR: R. CHOWDHURY (e-mail: rasel.chowdhury.1@ens.etsmtl.ca)

ABSTRACT As the cloud moves from monolithic infrastructure to a self-isolated cloud native microservice environment, automation is becoming an important aspect for the management of the application life cycle. In this context, there are many tools available that can monitor these applications and raise alarms. However, automated orchestration is still in its early stages, and the available solutions are not capable of monitoring the whole system from application to hardware levels and performing automated operations within the system. Moreover, IP Multimedia Subsystem (IMS), which is the core part of the Telecom industry, has switched to a microservice environment. These IMS services are critical and need to be proactively monitored to provide automated orchestration operations. In this paper, we address the aforementioned problem by proposing a new scheme for monitoring the metrics from different sources and proactively and automatically performing orchestration using machine learning while improving the scalability of the cloud native Virtual IMS. Experiments carried out with a real cloud-native IMS running in a kubernetes cluster explore the relevance, efficiency and scalability of the proposed scheme.

INDEX TERMS Cloud-native, monitoring, orchestration, Kubernetes, machine learning, vIMS, network and service management.

NOMENCLATURE

3GPP	3rd Generation Partnership Project
C_N	Node CPU utilization
C_N^a	Assigned node CPU
C_P	Pod CPU Utilization
C_P^a	Assigned Pod CPU
CapEx	Capital expenditure
CLI	Command Line Interface
D_P	Pod disk utilization
DevOps	Development and operations
F_C	CPU resource requirement
F_L	Application latency
F_M	Memory requirement
IMS	IP Multimedia Subsystem
KHPA	Kubernetes Horizontal Pod Auto-Scaling
KPI	Key Performance Indicators
L_P	Application latency
LTE	Long-Term Evolution
M_N	Node memory utilization
M_N^a	Assigned node Memory
M_P	Pod memory utilization
M_P^a	Assigned Pod Memory
MAPE-K	Monitor, Analyze, Plan, Execute, Knowledge
NFV	Network Function Virtualization
OpEx	Operating Expenses
QoE	Quality of Experience
QoS	Quality of Service
R_N	Node resource
R_P	Pod resources
RBAC	Role-Based Access Control
RBF	Radial basis function
RN_P	Number of requests that a pod processes

SIP	Session Initialization Protocol
SLA	Service-level agreement
SSL	Secure Socket Layer
SVR	Support Vector Regression
T_P^{ex}	Execution time of SIP requests
TH_N	Node threshold
TH_P	Pod threshold
TLS	Transport Layer Security
UMTS	Universal Mobile Telecommunications Service
vIMS	Virtual IMS
VNF	Virtual Network Functions
VoLTE	Voice over LTE
x_i	Actual value

I. INTRODUCTION

RECENTLY, telecom industries have undergone a rapid evolution. As LTE and VoLTE are becoming the norm, the telecom industries must meet the challenges faced by 5G and beyond. The main target of evolution to 5G is to meet the main use cases: enhanced mobile broadband, massive ultra-reliable communication, and low latency in communication. Currently, there are around 8.4 billion mobile users and they generate an enormous amount of data, which is around 25 exabytes [1]. During a 6-year period, mobile data usage has increased exponentially. The telecom industries are moving towards cloud solutions in order to cope with the increasing number of traffic. Furthermore, the cloud provides a shorter time to deploy new services, a lower operational cost (OpEx) and a better customer experience (QoE), according to the requirements of 5G and 6G. According to [2] and [3], proactive and intelligent monitoring and orchestration will play an important role in the management of telcom services that operate in heterogeneous cloud infrastructures.

Internet Protocol Multimedia Subsystem (IMS) was originally designed to evolve to Universal Mobile Telecommunications Service (UMTS) networks to provide Internet Protocol multimedia to mobile users [4]. IMS core components are traditionally run on over-provisioned proprietary hardware, which usually has higher capital expenditure (CapEx) and Operational Expenses (OpEx). Furthermore, QoS and QoE will play a vital role in the 5G network [5], as 5G is required to achieve better mobile traffic, a large number of users, better handling of user traffic and reduced end-to-end latency [6]. In order to reduce CapEx and OpEx, and to ensure the QoS and QoE, the telecom industries are moving toward vendor-neutral software-based solutions by virtualizing the IMS core. Cloudifying the telecom solution means that the components need to be transformed from the proprietary hardware-based solutions into Cloud-Native applications. Cloud native is a recent technological trend as an upgrade from monolithic architecture to distributed cloud infrastructure. Monitoring and orchestration of these cloud-native applications has important impacts on the application life cycle. Orchestration also plays an important role in managing different behaviors, such as scheduling, load balancing, health checking, fault tolerance, etc., for all cloud-native microservice solutions.

Cloud native applications or architecture are container-based environments, which are used to develop applications built with services packaged in containers, deployed as microservices, and managed on elastic infrastructure through agile software development and IT operations (DevOps) processes that continuously deliver workflows of the application itself. Cloud-native applications allow DevOPs to continuously deliver and use the microservice and container architecture. The Cloud-Native approach provides resilience, operating system abstraction, collaborations, continuous delivery, independent and automated scalability. IBM projected that automation will be the next phase for the cloud environment and the optimized industry [7]. Moreover, there is a rise in the utilization of cloud native applications for the manufacturing, healthcare, and telecom sectors. IBM Global Research has stated that 40% organizations have the ability and strategy to manage a cloud ecosystem, but lack the management tools to automatically manage them [8].

Cloudifying the IMS means that its modules and entities will become modular software components running in vendor neutral clouds, servers, or virtual machines. There are different ways of virtualizing the IMS core, such as running the whole IMS core in one virtual machine, making the IMS modules VNF (Virtual Network Functions) using NFV (Network Function Virtualization), and containerizing the IMS modules as microservices. Microservices are becoming popular for cloud native solutions [9] in cloud industries, due to their independent decoupled application modules, which are composed of small processes that communicate with each other using language-agnostic APIs.

Microservices use containerization, as it solves different cloud application issues, like Application Dependency Hell problem, application portability problem, performance overhead problem, etc. Orchestration of the native cloud application manages the microservices running in the cluster for resource allocation, life cycle, and other operations based on the policies set by the administrator. The orchestrator has an important role in the cloud native deployment, as it is responsible for the microservice management decisions. Kubernetes [10], Red Hat Openshift [11], Docker swarm [12], Amazon EKS [13], etc. are the container orchestration platform. Still these Orchestration platform have some limitations. The authors [14] explained that there are many limitations to orchestration operations. Some of the limitations are as follows:

- There are no or limited validated performance model for orchestration which are required to maintain a certain level of service.
- Orchestration metrics are not standardized to determine the performance of the containerized application.
- There are few standards or policies for different orchestration operations using the metrics of the application and system.

- The policies available are very simple for orchestration operations. For example, threshold-based scalability, round-robin for load balancing, etc.

The two most popular container orchestration platforms are the Docker Swarm and Kubernetes, and the rest uses these technologies incorporated in their system. Docker Swarm is faster in terms of deployment time, but it does not provide automatic scaling. It does not have built-in monitoring mechanisms that are provided through a third-party application. Docker Swarm allows for simple TLS for security and access control-related work. Kubernetes allows for high availability, fault tolerance, and self-healing, and also provides automatic scaling and can replace faulty pods if required. Kubernetes has a basic built-in monitoring system and supports integration with third-party monitoring tools. Kubernetes has support for various security schemes, such as RBAC, TLS/SSL, Secret management, etc. The main disadvantage of Kubernetes is that the installation process is complex and requires separate CLI tools.

Among the container orchestration platform, Kubernetes is the most popular in terms of open-source community, application deployment, availability, fault tolerance, self-healing, etc. Kubernetes ecosystem consists of a set of nodes that run the containerized applications; pods are the smallest deployable units of computing of an application running in the nodes. Kuberenetes have different components such as Control Plane Components which makes the global decision about the cluster as well as performing cluster events; Node Components executes on every node that performs operations such as maintaining the pods and Kubernetes runtime environment; and Addons that use the Kubernetes resources in order to perform cluster-level operations such as DNS, Web UI, Container Resource Monitoring, etc. [15], [16]

Monitoring the application metrics running in a microservice environment is important for orchestration operation. There are different metrics that can be monitored such as CPU, Memory, Network traffic of Pods, and Nodes. These monitored metrics allow the Kubernetes cluster orchestrator to perform reactive operations like scalability, load-balancing, fault tolerance, health check, etc. based on some simple SLA policies. However, these operations are very basic; for example, scalability operations are performed on the basis of threshold using CPU and memory utilization. Orchestrators can actively monitor the system and can perform the operation reactively, but these operations policies are very simple and cannot perform the operation proactively for critical systems that require the operations to be executed in a timely manner and efficiently.

There are many metrics that the telecom industries monitor ranging from hardware resources to application to provide better QoS and QoE. According to [5], it is important to identify service-specific QoE and its effects on various QoS metrics based on the different services that the telcom operators provide. Current telecom operators can actively monitor the system using the monitoring tools available for native cloud solutions, but these monitoring tools do not prevent the system from crashes or abnormal behaviors since the events occur randomly. Currently, there are many monitoring tools, but there are no orchestration tools that can take into account the behavior of the system and take the necessary actions in real time. In the field of research, there have been few researches, but they are domain specific or they do not take into consideration all of the parameters for monitoring and orchestration; also there are no systems available to analyze the events so that the cloud native IMS can adapt in real time. Usually, the cloud native systems are managed by the operator who is in charge of the system, and the decisions are left to the operator based on their own experience. In this paper, we propose an orchestration framework that is capable of monitoring native cloud applications and automatically taking orchestration decisions. In our research, we have used the implementation of the session initialization protocol (SIP) of cloud native virtual IMS (vIMS) to evaluate and validate our architecture. The main contributions of this paper are as follows.

- New monitoring and orchestration architecture for cloud native systems for Kubernetes cluster, which can perform operations by monitoring Pod, Node and Application;
- Formulation of the horizontal scalability problem as multi-variant support vector regression using different kernels for resource prediction;
- Resource prediction algorithms for the application specifically for SIP servers in order to achieve scalability operation;
- Realistic cloud-native SIP application as a use-case to test and evaluate the monitoring and orchestration architecture.

The rest of the paper is organized as follows: In Section II, we present the background on IMS, microservices, orchestration, monitoring, etc. Section III provides the related work in the research domain. In Section IV, we propose our orchestration architecture. Section V explains the formal definition of the problem. Section VI presents the data collection, the modeling of the machine learning algorithm, and the orchestration algorithm. In Section VII, we present our case study and performance analysis. Finally, the conclusion of this paper is drawn in Section VIII.

II. BACKGROUND

A. INTERNET PROTOCOL MULTIMEDIA SUBSYSTEM(IMS)

IMS [17] is a global, access-independent, and standard-based Internet Protocol (IP) connectivity and service control architecture that enables various types of multimedia services to end-users, using common Internet-based protocols. IMS is standardized by 3GPP and is used to provide different services like Image Share, IMS Security, Multimedia telephony, OMA Instant Messaging and Presence Service, Peer-to-peer video sharing, Push-to-talk, Real-time text, Session Initialization Protocol (SIP) extensions for the IP

Multimedia Subsystem, Text over IP, Video Share, Voice call continuity, Presence, etc. for users.

SIP [18] is a signalling transport protocol, which is one of the core functionalities of cloud-native IMS. SIP is used for signalling and controlling communication sessions for different services like instant messaging, voice of LTE, events notification, Internet conferencing, instant messaging, etc. The SIP protocol makes use of proxy SIP severs to interconnect remote SIP users and systems end-to-end. Additionally, SIP has been used to establish Quality of Service (QoS) between network operators and for different Internet architectures.

B. MICROSERVICES

Microservices are software development techniques that make an application a collection of loosely coupled independent services [19]. Microservices are implemented and operated as small independent systems, offering access to their internal logic and data through a well-defined network interface. In this architecture, the services are made into fine-grained modules and the protocols that are used to communicate are usually lightweight. Microservices allow faster delivery, improved scalability, and greater autonomy for development and software maintenance. Due to their nature and their benefits, microservices are gaining popularity. Container is a portable encapsulated environment where a bundle of OS, libraries, environment variable, and software are located. There are different architectures like [20] and [21] that used microservices and docker containers for application deployment.

C. MONITORING MICROSERVICES

Monitoring the microservices running in containers is necessary, as containers are buried under the physical host, which is invisible to traditional monitoring architecture. Due to the nature of containers, microservices create a big blind spot in the system. Monitoring the containers will allow different orchestration decisions, like diagnosing performance of the container, failure detection, malicious activities, network performance, etc.

There are lots of metrics available to monitor a container ecosystem; some of the most important metrics and their importance for orchestration are as follows:

- *Relative metrics* measure values based on data collected from the virtual file system, such as the utilization of the CPU / memory of the container. These are usually collected using tools such as docker stats or cAdvisor.
- *Absolute metrics* measure the cumulative activity of the overall system, such as the CPU/Memory consumption of the physical machine. These are collected using mpstat, top, etc.
- *Application metrics* measure the performance and activity of the application, such as the latency of the processes, the number of requests it is processing, etc. These metrics are collected if the application is exposing these metrics using custom tools.

D. ORCHESTRATION

Orchestration allows the cloud and application providers to define how to select, deploy, monitor, and dynamically control the configuration of multi-container packaged applications [14]. According to [14], the different types of microservice orchestration are:

- *Resource limit control* allows to reserve a specific amount of resources like CPU and memory for a container;
- *Scheduling* defines the policy used to place the amount of container on nodes at a given time instance;
- *Load Balancing* distributes the loads/tasks among multiple container instances;
- *Health checking* checking verifies if the container is capable of answering requests;
- *Fault tolerance* is implemented as replica control and/or high availability controller;
- *Auto-scaling* allows to automatically add or remove container or resources based on some conditions.

There are two main types of scalability, which are as follows:

- *Vertical scalability* increases or decreases the computing resources, such a CPU, memory, etc. Vertical elasticity is also known as resource resizing. The vertical elasticity is limited to the host machine capacity as it cannot provision more resources when all the host machine resources are already allocated to different containers.
- *Horizontal scalability* adds or removes instances of computing resources associated with an application. Horizontal elasticity is also known as replication of resources. One disadvantage of horizontal scalability is that it requires more support from the application to be decomposed into instances. Another disadvantage of horizontal scalability is that it requires some time to start another instance, as well as a period of cooling when scaling down.

E. KUBERNETES

Kubernetes [10] is an open source container orchestration platform for automating the deployment and management of applications. Kubernetes has become one of the most popular orchestration tools for cloud native applications. The main Kubernetes components that are used in this research are as the following:

- *Container* is a portable encapsulated environment where a bundle of OS, libraries, environment variable and software are located.
- *Pod* is the basic scheduling unit in which the containers are placed. A pod can consist of one or more containers running in parallel with each other.
- *Node*, sometimes called a Worker or a Minion, is a virtual machine or physical machine where pods can be deployed.
- *Cluster* is a collection of nodes which work together to run the containerized applications.

F. KUBERNETES HORIZONTAL POD AUTO-SCALING(KHPA)

The state of the Art Kubernetes Auto-Scaling is the KHPA Algorithm [22], a relative performance measured using threshold. The algorithm inputs the relative utilization and the number of active pods and outputs the number of pods needed to be deployed.

Relative performance measures are adapted by all container orchestration frameworks. But relative usage measures underestimate the required capacity, so it is not capable of determining the most appropriate resources required by the service level objectives.

III. LITERATURE REVIEW

In the domain of automated orchestration, there is much research being carried out. In this section we will mainly list out the research specific to monitoring and the orchestration on cloud native applications.

Monitoring the services running in a container is an important aspect for the management and orchestration of microservices. It allows the administrator to evaluate the microservice, such as creating or removing replicas based on the usage of the containers, detecting faults, and monitoring malicious activities in the containers. Available monitoring tools typically include system resources like CPU, memory and I/O usage which usually consume a lot of resources, and most of them have not taken into consideration the network traffic between containers.

The authors [23] implemented an open source lightweight container resource monitoring tool (PyMon), mainly designed for resource-constrained devices. The architecture is based on the host-based monitoring tool Monit for docker container inspection Moradi et al. [24] proposed an automated inter-containers network monitoring tool for measuring the network performance such as delays, jitters and packet loss between applications of the containers. It is a distributed and automated solution for observing network metrics in a container-based environment. The implementation of the tool allows dynamic monitoring as well as automatic service discovery and setup of the environment in case of scaling and migration without manual intervention on the containers. The monitoring system allows passive as well as active monitoring of the network traffic of containers.

According to Casalicchio et al. [14], in the state of autonomic orchestration, orchestration operations should include self-healing, self-optimization, self-protection along with orchestration functionality. The authors [25] discussed the orchestration problems and challenges. Reference [26], proposed a method to automatically form and monitor the Kubernetes Federation using the TOSCA standard. The main reason for choosing TOSCA as their solution was that they wanted to demonstrate multi-cloud solutions for mainstream cloud technology and platform.

Pelaez et al. [27] proposed a dynamic adaptation of policies using machine learning. The approach allows administrators to define their policies in terms of high-level goals and lets the system determine which actions to apply (and when to apply them) in order to guarantee that those goals are met. Support Vector Regression, Nearest-Neighbors Regression, Random Forest Regression, Ridge Regression, and Neural Network are all used for evaluation.

In [28], [29], the author presented a cloud-native implementation of telecom services as microservices using containers, as well as Kubernetes for auto-scaling. In their approach, they have integrated a predictive machine learning algorithm using linear regression and a moving average integrated with auto regression. In their autoscaling orchestration, they have predicted long-term forecasting based on the seasonal traffic to update resources using long-term data; and predicted real-time scaling decision based on the current data from the system to predict the number of containers. In their implementation, they used CPU, memory, etc. as features for the prediction algorithm.

The authors [30] proposed a reinforcement learning agent that can horizontally scale container instances based on the user's demands and also schedule the placement of containers based on the available resources. Reference [31] used K-means clustering along with the generic algorithm to address container placement in fog devices. Reference [32] used multi-objective particle swarm for container-based scheduling for IoT in cloud environments. Reference [33] quantified the impact of resource utilization and performance interference on the end-to-end tail latency of various requests from Web applications running in a microservice environment. In their approach, they used linear regression, support vector regression, decision tree, random forest, and deep neural network for comparison. Reference [34] showed how reinforcement learning can be used to place servers and allocate workloads in edge computing. Reference [35] proposed QoE-DEER a game-theoretic approach to solve edge resource allocation, which allocates resources to various Telcom IoT users to maximize the overall QoE of the ecosystem. They have performed several experiments to show that their approach is similar to the optimal baseline of QoE.

In [36], the authors explained why the selection of appropriate metrics is very important for container auto-scaling in Kubernetes, as auto-scaling can increase the performance of the overall system. In this paper, the authors discussed the correlation between absolute and relative usage measures and how resource allocation decisions can influence them. Reference [37] provided a comprehensive architecture-level view of Kubernetes and its autoscaler. The authors also provided a detailed analysis of KHPA using multiple scenarios.

Reference [38] introduced a multilevel monitoring framework and a method for dynamic thresholds for fine-grained autoscaling. Their approach continuously monitors the system and allocates resources needed to have efficient performance of the application. The fine-grained method is based on a set of adaptation rules using dynamic thresholds.

TABLE 1. Comparison other research.

Implementation	Orchestration	Algorithm	Metrics			
			Pod	Node	Application	Latency
[22]	Scalability	Threshold	Partial	No	No	No
[43]	Cluster scheduling, Cluster Fault Tolerance	Random Forrest	No	Yes	Yes	No
[44]	Fault Tolerance	Fuzzy Logic	No	Yes	No	No
[28] [29]	Scalability	Linear Regression	Partial	No	No	Yes
[33]	Scalability	Linear Regression, Decision Tree, etc	Partial	Partial	No	No
[36]	Scalability	Rule Based	Partial	No	No	No
[38] [38]	Scalability	Rule Based	Yes	No	Yes	Yes
[39]	Scalability	Threshold Rule Based	Partial	No	Partial	No
[40]	Scalability	Heuristic	Partial	No	Yes	No
Our framework	Scalability	Support Vector Regression	Yes	Yes	Yes	Yes

Balla et al. proposed [39] an adaptive autoscaler, Libra, which automatically determines the optimal resource settings for the Kubernetes pod and manages the horizontal scaling process. Libra uses a canary deployment of the application to find the threshold of CPU based on the number of request its handling. On the basis of their results, their autoscaler provides better results than the Kubernetes native autoscaler. Reference [40] used heuristic-based for autoscalability, while the authors in [41] proposed an intent-based orchestrator for 5G application that dynamically relocates applications to fulfill the requirements of end-user services.

Table 1 shows the comparison of our work with other research and the added values in addition to these implementations. KHPA [22] is the native Kubernetes threshold-based scalability algorithm using equation (7) that takes the CPU utilization of the pod over a period of 30 seconds to increase or decrease the number of pods, but its implementation does not take into account the memory utilization, CPU and memory of the node, as well as the application parameters such as latency and the number of requests the application is processing. References [28], [29] used linear regression using metrics from the CPU and memory utilization, as well as the latency of pods for scalability. In their experimentation they have shown that their implementation works better than KHPA in terms of time for scalability, but their approach has not considered monitoring metrics from the node and application to tackle the scalability of the application. References [33], [36] quantified and explained the different metrics requirement for scalability; in their approach, they have utilized pod metrics, but their approach has not taken into account node and application metrics. Reference [33] used linear regression, support vector regression, decision tree, random forest, and deep neural network to implement and compare their approach, whereas [36] used threshold rule based approach. References [38], [39] are able to monitor the utilization of pod resources and application and have used a threshold rule-based approach to perform scalability operation, but have not anticipated the utilization of node resources nor the latency of the process. All monitoring and orchestration frameworks implemented discussed do not take into account resource utilization of nodes, which is a very important metric to monitor, as it is the space holder of the pods. Furthermore, these implementations have not used the metrics available from the application, such as the number of process requests that it is processing, which can provide better insight into the decision about scalability. Our implementation has taken into account all the different types of metrics (relative, absolute, and application) to make orchestration decisions for scalability.

IV. ARCHITECTURE

Figure 1 shows a high-level design of the Cloud-Native IMS automation which is based on the MAPE-K architecture [44] for Cloud Native Application orchestration. The architecture is designed to collect metrics from applications, pods, and nodes. Then, based on the behavior of the pod, it can take orchestration decisions like scalability, faults detection and recoveries, anomalies detection and recovery, etc. for the application it is monitoring. The *Monitor* collects metrics of pods along with application metrics, as well as metrics of the node where the pod is running, then transmits the values to the database for future use and to the *Enforcer*. *ML-Engine* creates different models for orchestration based on the data stored in the *Database* and sends the models to the *Enforcer*. *Enforcer* takes the orchestration decisions based on the current metrics provided by the Monitor using the models received by the *ML Engine* and forwards the decision to the *Orchestrator*. Based on the orchestration decision of the *Enforcer*, *Orchestrator* performs the operation in the cluster. All modules run on a native Kubernetes cluster. The details of each module are as follows:

Figure 2 shows in depth how the Monitor is implemented and able to collect metrics from different sources. The Monitor is implemented using Python using the Kubernetes Python API [45] and Prometheus API [46]. Monitor queries

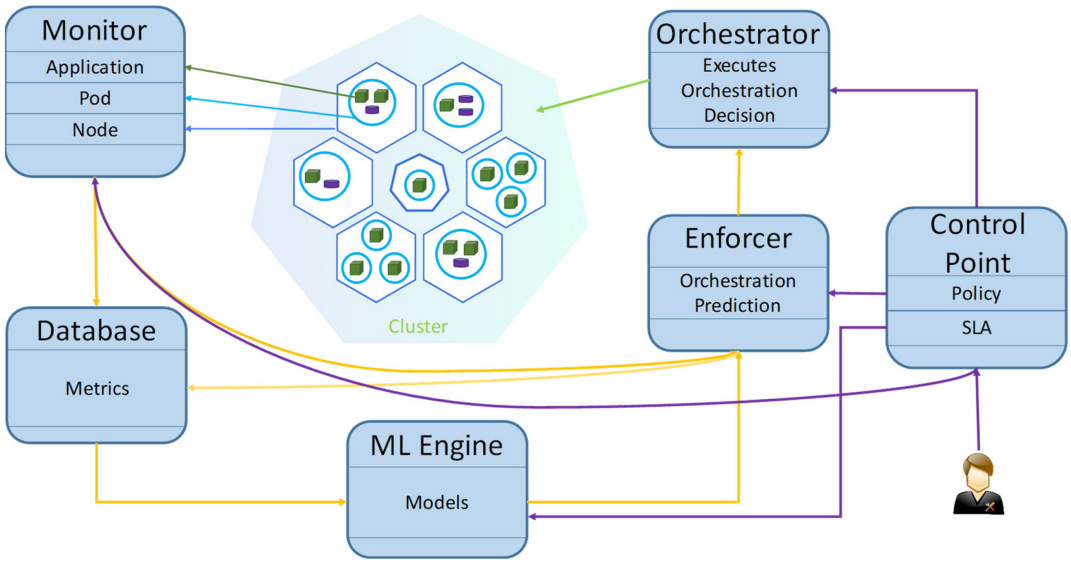


FIGURE 1. Proposed Architecture.

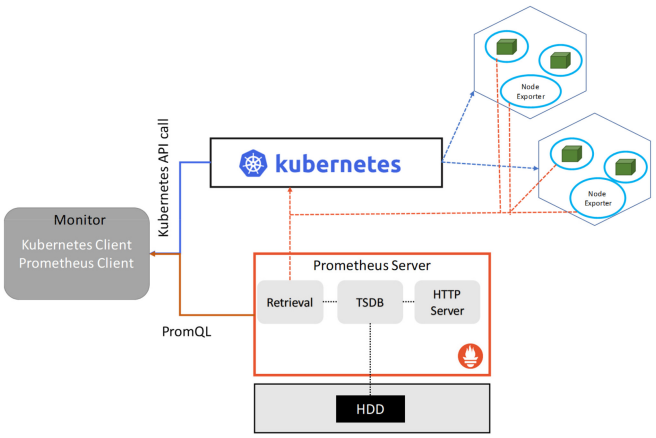


FIGURE 2. Monitor.

the Kubernetes cluster using the Kubernetes API to get the information regarding the clusters, like the list of nodes, pods, deployments, namespaces, etc. Then, the Monitor selects a specific deployment and queries the cluster using the Prometheus API to find the metrics of the system. For relative metrics or Pod Metrics, it selects a deployment and queries the cluster to get the resources, like resource limit and resource utilization, of CPU, Memory, Disk and Network. For node or absolute metrics, it collects the resource utilization as well as the configuration of the node, like the number of cores available for the node memory, by using multiple Prometheus queries on the cluster. Application metrics are specific application values generated by the application itself, which are also collected with Prometheus queries. Finally, metrics collected from Pods, Nodes and Applications are aggregated based on certain criteria and are transmitted to the Database for storage as well as to the Enforcer for decision.

A. ML-ENGINE AND ENFORCER

ML-Engine and Enforcer are the brain of the system. ML-Engine retrieves the metrics from the database and creates models periodically based on different algorithms, and then sends them to the Enforcer.

ML-engine generates multiple Machine Learning Models using supervised or unsupervised learning for each type of orchestration operations. In our system there are two types of Model for the same operations. One of them generates models for long-term decisions and later one for the pro-active decisions. Long-term models use time-based historical data to generate models for orchestration operations. However, short-term models are generated using metrics from all the collection points for making proactive orchestration decisions. Once a model is generated using the machine learning algorithm, it is being transferred to the Enforcer to update that specific model to make the decisions. The Enforcer will provide an efficient resource management, load balancing of the traffic, as well as anomaly detection along with recovery solutions.

The Enforcer uses the models generated by the ML-Engine to predict the type of operations needed to be performed on the Kubernetes cluster. For long-term predictions, it will use the long-term models generated by the ML-Engine and take the orchestration decision based on seasonal time events. For proactive predictions, the short-term model and SLA policies are taken, provided by the Cluster Manager. When the Enforcer makes the decision, it passes the decision to the Orchestrator as well as to the Database if the decision is made using a supervised learning algorithm.

Figure 3 shows the internal operation of the ML-Engine and the interconnection to the enforcer. Modules are implemented using Python to utilize enriched machine learning libraries, such as Keras [47] for the deep learning algorithm

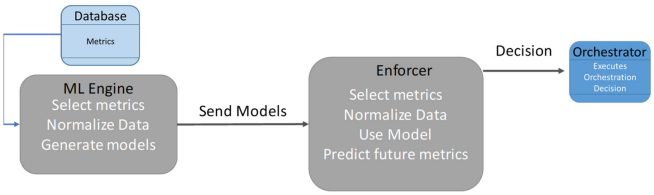


FIGURE 3. ML-Engine and Enforcer.

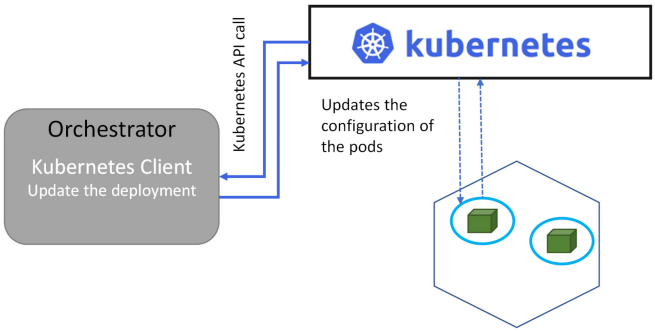


FIGURE 4. Orchestrator.

along with Sci-kit [48] for decisions. When the ML-Engine pulls the data, it normalizes them so that the model provides more accurate results. For short-term models, ML-Engine periodically pulls the specific dataset from the Database, selects specific metrics, and retrains the data to create a new model using supervised or unsupervised Machine Learning algorithms. For long-term predictions, the ML-Engine uses data over a period of days, e.g., a month or a few months to train the model. Once the models are generated, it sends them to the Enforcer. Once the Enforcer receives the aggregated data from the Monitor, it normalizes the data and selects the model for the proactive orchestration decisions. For long-term predictions, the Enforcer periodically uses the long-term models to make the orchestration decisions. Details of the machine learning algorithm and the instances that predict the future are discussed in V. The decision for the orchestration is sent to the Orchestrator along with the deployment name or the pod id.

B. ORCHESTRATOR

The Orchestrator enforces the orchestration decisions on the Kubernetes cluster. It has the necessary functionalities to execute the appropriate commands to perform the operations like elastic scalability, restarting the application, load-balancing, etc. This module updates the Kubernetes cluster with the decision provided by the Enforcer based on the best possible orchestration solutions.

The internal mechanism of the Orchestrator is shown in 4. The module is implemented with Python using the Kubernetes API and machine learning libraries [47], [48]. The orchestration decision is received from the Enforcer which contains the deployment name, as well as the pod id. The Orchestrator uses the Kubernetes API to retrieve the deployment configuration or the pod information from the cluster. If the orchestration decision is to update the deployment configuration, it modifies the deployment configuration object and pushes it to the cluster, which updates the deployment in the cluster. If the orchestration decision is to modify the pod, it pushes the updates to the cluster using the Kubernetes API. Therefore, the cluster is updated with the latest decision based on the current situation of the cluster.

C. CONTROL POINT

Control Point is an interface for the application administrator to assign SLA policies to microservices. This module is the only way for the administrator to access the system to intervene and update the policies. Control Point allows the administrator to set different parameters from service-level features. Features include the data collection frequency, new machine learning algorithms for the orchestration, maximum resources a microservice can have, maximum number of pods to be assigned to the microservices, the different strategies for recovering from faults, etc.

D. DATABASE

The Database is required for storing the data generated by the system in the time series database. Whenever the Monitor sends the aggregated metrics from the system, the database stores the data using a time-series schema. When the data are requested by the ML-Engine, it converts the database into a suitable format and sends the dataset back to the ML-Engine. If the data requested by the ML-engine are for the proactive models, it sends the dataset for a specific interval of time; otherwise, it sends the whole dataset.

V. MULTI-OBJECTIVE OPTIMIZATION FOR SCALABILITY

In this section, we present a formal definition of the scalability problem and explain the formulation as a multiobjective optimization problem.

A. PROBLEM DEFINITION

In our Kubernetes cluster, resource utilization plays an important role in horizontal scalability. A pod has different resources varying from hardware to the application metrics. [49], [50], [51] illustrated different approaches for resource optimization based on the type of workload as well as the window size to predict resource utilization. The resources that impact the scalability of a pod are composed of C_P, M_P, RN_P and L_P, which we denote as R_P=(C_P, M_P, RN_P, L_P). In addition, a pod has a threshold of the resources indicated as TH_P. These threshold limits are the CPU utilization assigned C_P^a and the assigned memory utilization M_P^a. A cluster Node consists of multiple Pod running, so the resource utilization of a node is $R_N = \sum_{i=1}^{n} [C_{P_i}, M_{P_i}, D_{P_i}]$, where n is the number of Pods. The threshold of a node, Th_N, includes C_N^a and M_N^a.

$R_P \propto RN_P$ as the pod has to process the SIP requests in parallel. So, $L_P \propto [RN_P, R_P]$ increases the latency of that instance. Also $L_P \propto \frac{1}{TH_P}$ since there will be less resources to process requests, the latency will increase. Once a pod reaches these thresholds, the pod needs to be scaled up. A replica in the Kubernetes cluster is an instance of the Application running in a pod, and once a replica reaches the threshold, the cluster needs to add a new replica to perform the operations efficiently.

To perform horizontal scalability, we need to formulate the resource consumption of each pod independently as the pod will be having different workloads based on the number of SIP requests it is processing. Based on the workload and resource consumption of a pod at time t, the pod might need to create a replication or remove it. As an example, the conditions below show simple scalability actions based on the resources of the pod at time t.

$$\begin{aligned} R_{P_i} &< TH_{P_i}, \quad Do\ Nothing \\ L_{P_i} &< 200ms, \quad Do\ Nothing \\ R_{P_i} &> TH_{P_i}, \quad Scale\ Up \\ \forall P_i(t), i : 1 \rightarrow n, if\ L_{P_i} &> 200ms, \quad Scale\ Up \\ R_{P_i} &> TH_N, \quad Scale\ Up \\ R_{P_i} &< 0.2\%, \quad Scale\ Down \\ L_{P_i} &< 10ms, \quad Scale\ Down \end{aligned}$$

In order to predict resource utilization at a given time t, Multivariant Support Vector Regression (SVR) is one of the suitable solutions. The goal of SVR is to calculate a function that is capable of minimizing the overestimation and underestimation of data based on the ε insensitive loss function, penalizing predictions that are farther from the desired output. The target function can be formulated according to [52] as

$$\begin{aligned} f(x) &= < w, x > + b \\ &= \sum_{j=1}^{M} (w_j \times x_j) + b \end{aligned} \tag{1}$$

$$\text{where } w, x \varepsilon \mathbb{R}^M, \text{ b} \varepsilon \mathbb{R}$$

where x is the input variable and w is the weight. Using optimization and applying slack variables and Lagrange multipliers equation as shown in [52] and [53], equation (1) can be re-written as

$$\begin{aligned} w &= \sum_{i=1}^{n} (\alpha_i - \alpha_i^*) \gamma(x_i) \\ f(x) &= \sum_{i=1}^{n} \left(\alpha - \alpha^*\right) K < x_i, x > + b \end{aligned} \tag{2}$$

where $K < x_i, x >= \varphi(x_i)\varphi(x)$ is called the kernel functions. The kernel functions used in this research for formulation of our cost functions are

- Linear: $K < x_i, x >= x_i^T x$
- Polynomial: $K < x_i, x >= (\gamma x_i^T x + r)^d, \gamma > 0$
- Radial basis function (RBF) : $K < x_i, x >= \exp(\gamma \|x_i - x\|^2), \gamma > 0$

The efficiency of scalability depends on the prediction of resources required for Pod based on the current resource cost function.

- F_C is the CPU resource required to run the application at time t for processing the workload. It is equal to the number of SIP requests processed along with the utilization of other resources. By doing so, we maximize the CPU utilization of each pod.

$$F_C = \left(\frac{RN_P \times T_P^{ex} \times C_P \times C_P^a}{C_N^a} + C_N \right) \tag{3}$$

- F_M is the memory required for running the application and processing the workload based on the number of sip requests, as well as other dependent resource utilizations.

$$F_M = \left(RN_P \times T_P^{ex} \times \left(\frac{M_P^a}{M_N^a} \times M_P + D_P \right) \right) \tag{4}$$

- F_L calculates the application latency based on the application workload it is processing along with other resource consumption that affects the application latency.

$$F_L = \left(RN_P \times \left[\frac{C_P}{C_N} + \frac{M_P}{M_N} + \frac{T_P^{ex}}{TH_P} + D_P \right] + L_P \right) \tag{5}$$

Based on [52], [53] our objective function is

$$maximize \begin{cases} -\frac{1}{2} \sum_{i,j=1}^{l} (\alpha_i - \alpha_i^*)\left(\alpha_j - \alpha_j^*\right) k < x_i, x_j >, \\ -\epsilon \sum_{i,j=1}^{l} (\alpha_i - \alpha_i^*) + \sum_{i=1}^{j} y_i (\alpha_i - \alpha_i^*) \end{cases} \tag{6}$$

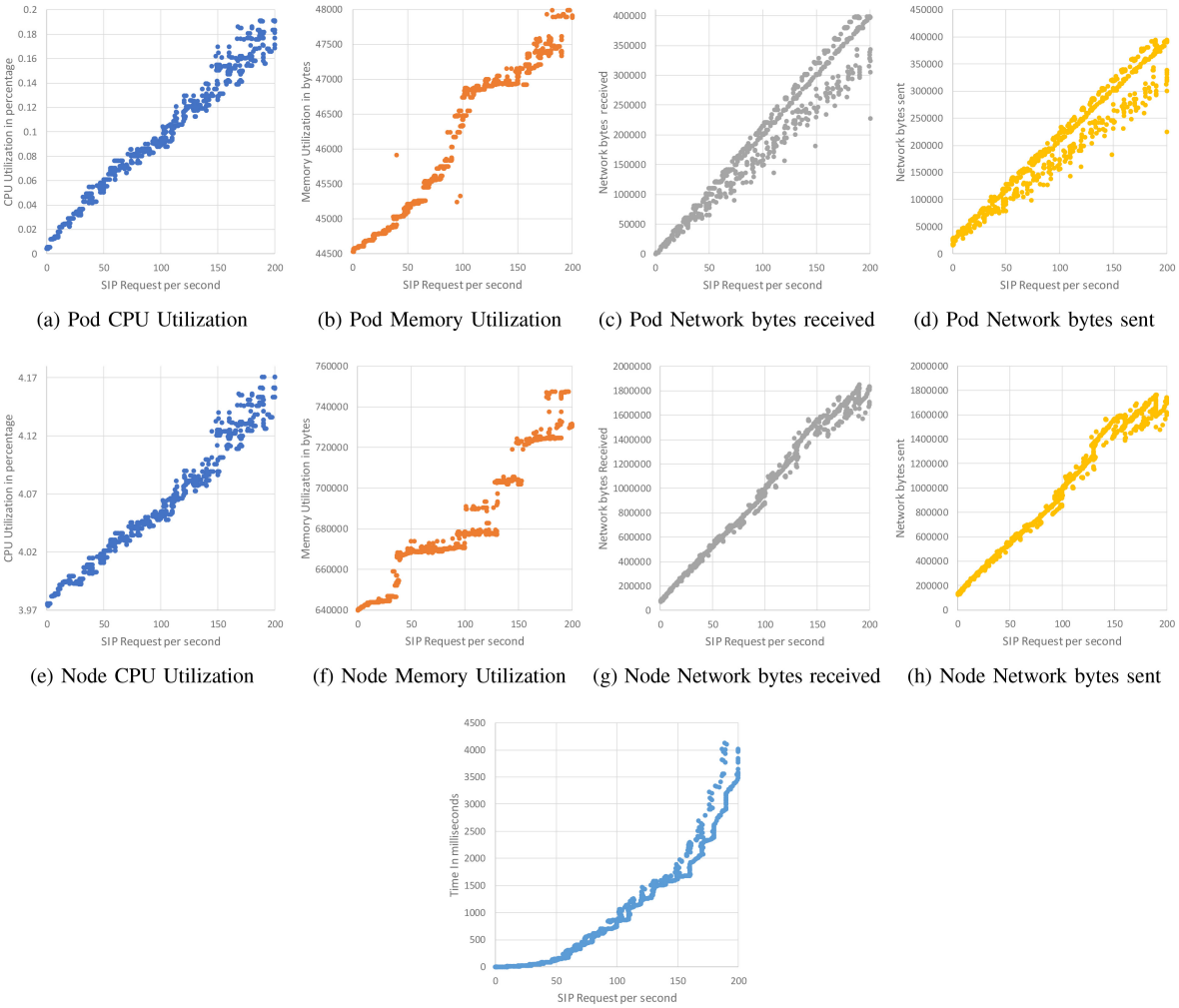
Subject to $\sum_{i=1}^{l} (\alpha_i - \alpha_i^*) = 0$ and $\alpha_i, \alpha_i^* \varepsilon [0, C]$, where $x_i =$ F_C or F_M or F_L.

B. PREDICTION ALGORITHM

The Algorithm 1 is the prediction algorithm which is designed based on the definitions of the problem and the objective function. The algorithm requires the current resource metrics of a pod, then evaluates the equation (2) with functions (3), (4), and (5) using different kernel functions to predict the future resources required. After the prediction of the resources is done, it makes the decision for the pod to scale up or down or to keep it the same.

VI. IMPLEMENTATION

The first step in automated scalability is to collect data from the observation system. Collected data will be used to observe the behavior of the system and select the appropriate algorithms for the pro-active automated scalability. In this section, we explain the data collected from the system using the monitor, representation of the data, and selection of the machine learning algorithm based on the system behavior and implementation of the orchestration algorithm.



(a) Pod CPU Utilization (b) Pod Memory Utilization (c) Pod Network bytes received (d) Pod Network bytes sent

(e) Node CPU Utilization (f) Node Memory Utilization (g) Node Network bytes received (h) Node Network bytes sent

(i) Latency

FIGURE 5. Representation of metrics with respect to the traffic load.

A. DATA COLLECTION AND REPRESENTATION

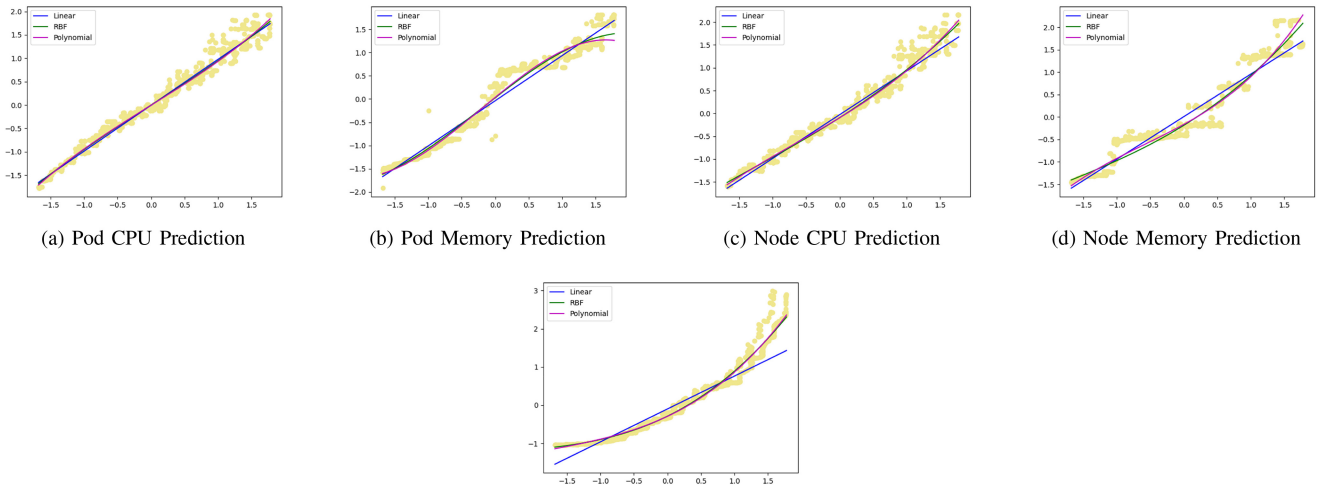
Figure 5 illustrates the resource metrics collected from the system that vary with respect to the number of SIP requests generated by the SIP traffic generator. Figure 5(a) shows the CPU utilization of the SIP traffic handler instance running in the pod, and Figure 5(e) shows the utilization of the node cpu that hosts the pod. Figures 5(b) and 5(f) show the memory utilization of the pod and node, respectively. The network utilization, divided into bytes sent and received by the SIP instance in the pod, as well as the node, are shown in Figures 5(c), 5(d), 5(g) and 5(h) in the given order. The latency of the SIP instance is illustrated in Figure 5(i).

From Figures 5(a) to 5(i), it is clear that the CPU utilization and network utilization are increasing gradually with the number of SIP requests sent to the SIP server. Whereas the latency for processing the SIP requests is increasing exponentially as the SIP requests are increased. Memory utilization increases gradually in the pod, but it increases step by step in the node as the resources are provisioned by Kubernetes and internal processing of the nodes.

B. MACHINE LEARNING ALGORITHM FOR PREDICTION

To predict future resource utilization based on current metrics collected from the system, we have used a Machine Learning technique. Machine learning is capable of predicting future resources based on the correlation of metrics shown in Figure 5. As shown in Figure 5, the behavior of the system follows a regression model instead of using classification, since the metrics are not labeled to decide the number of replicas required proactively. We have selected Support Vector Regression (SVR) as the machine learning algorithm to predict the resource utilization for CPU, memory, and latency. SVR is a type of Support Vector Machine for regression which is used to predict values based on the subset of the training data to build the prediction model.

Concerning the orchestration decision for scalability, we have selected the resource utilization of the pod and the node where the pod is currently running along with the



(a) Pod CPU Prediction (b) Pod Memory Prediction (c) Node CPU Prediction (d) Node Memory Prediction

(e) Latency Prediction

FIGURE 6. Prediction Algorithms.

application metrics. The dataset used to train the model is derived from the metrics collected from running different traffic loads as shown in Figure 5. The dataset is split into 70% and 30% for training and testing the algorithms, respectively. To train the prediction algorithms, we first normalize the data using the normalizer. We trained the system to predict five different resource utilizations, namely pod CPU utilization, pod memory consumption, node CPU utilization, node memory consumption, and the latency of the pod based on the number of SIP requests it is processing. We have tested the model with three different SVR algorithms; linear SVR, Polynomial SVR and Radial Basis Function (RBF) kernel.

The visual representation for training the different SVR algorithms is shown in Figure 6 for training 70% of the dataset. From the training, we have the following estimations:

- The Pod CPU utilization (Figure 6(a)): we see that the three types of SVR are suitable for prediction;
- The Pod Memory consumption (Figure 6(b)): linear is not suitable for prediction where as RBF or Polynomial are preferable;
- The Node CPU consumption (Figure 6(c)): polynomial and RBF shows better results for prediction;
- The Node Memory prediction (Figure 6(d)): RBF seems to be more suitable than polynomial and linear;
- The latency (Figure 6(e)): polynomial and RBF algorithm are better in predicting the latency of the system.

To select the best algorithm to predict resource consumption, we tested the system with the remaining 30% of the dataset. The R2 score of the algorithms for predicting CPU, memory, and latency is presented in Table 2. From the results of Table 2, it is clear that

- For Pod CPU, RBF SVR is the best among them;
- For Pod Memory, RBF SVR is the most suitable;
- For Node CPU, RBF SVR has the highest R2 score;

TABLE 2. R2 scores of the SVR algorithms.

Metrics	Linear SVR	Polynomial SVR	RBF SVR
Pod CPU	0.984019	0.98417	0.98449
Pod Memory	0.95508	0.96700	0.96721
Node CPU	0.97499	0.98232	0.98261
Node Memory	0.92144	0.93957	0.93871
Latency	0.87845	0.98061	0.98009

- For Node Memory, polynomial SVR is the best;
- For latency, polynomial SVR shows the best R2 score.

C. ORCHESTRATION ALGORITHM

Algorithm 1 and 2 show the orchestration algorithms that predict the future values of resource consumption and make the orchestration decisions for scalability, which are to increase or decrease the number of pods of SIP server instances. instances.

The Algorithm 2 is the main algorithm which receives the metrics from the Monitor in the form of a list that is collected over that time. The first step of this algorithm is to get the number of SIP requests from the Metrics List and add it at the end of the SIPList, which holds the previous numbers of SIP requests. Then it finds the trend of the SIP request the pod is processing, as either increasing, decreasing, or keeping stable based on the SIPList values. After that, the average of the SIP request is calculated and added to the metrics list. By averaging the SIP requests, it eliminates abnormal values of the system. Finally, SIPTrend is found using the SIPList that formulates whether the traffic is increasing, decreasing, or keeping stable. Based on SIPTrend, it increases SIP requests accordingly, as shown in the algorithm, and then calls the Algorithm 1 for the scalability decision.

The Algorithm 1 decides whether to scale up or down the pod based on the prediction of future metrics. The algorithm requires the Metrics list as well as the Machine Learning

Algorithm 1 Predictor

Require: MetricsList, Resource Models, Limits
RequirePodCPU = evaluate equation (2) with x=F_C
RequiredPodMemory = evaluate equation (2) with x=F_M
RequiredNodeCPU = evaluate equation (2) with x= MetricsList
RequiredNodeMemory = evaluate equation (2) with x= MetricsList
RequiredLatency = evaluate equation (2) with x= MetricsList
if RequiredPodCPU > P_{ac} OR RequiredPodMemory > M_P^a **then**
Send IncreasePod to Orchestrator
else if RequiredNodeCPU > C_N^a OR RequiredNodeMemory > M_N^a **then**
Send IncreasePod to Orchestrator
else if FutureLatency > UpperLimits **then**
Send IncreasePod to Orchestrator
else if RequirePodCPU < LowerPodCPULimit OR RequiredPodMemory < LowerPodMemoryLimit OR RequiredLatency < LowerLatencyLimit **then**
Send DecreasePod to Orchestrator
else
Do Nothing
end if

Algorithm 2 Enforcer

Require: MetricsList
SIP ← *SIP requests* from *Metrics*
Insert SIP in SIPList
SIPTrend ← GetSIPTrend(SIPList)
SIP ← Average(SIPList)
if SIPTrend = Increasing **then**
SIP ← SIP + 20
Update SIP request in MetricsList ← SIP
CALL Algorithm 1 with MetricsList
else if SIPTrend = Decreasing **then**
SIP ← SIP + 10
Update SIP request in MetricsList ← SIP
CALL Algorithm 1 with MetricsList
else if SIPTrend = Stable **then**
Do Nothing
end if

models which are generated by the ML Engine. At the beginning of the algorithm, it predicts the future values of the resources by using the ML models with the values sent by the Algorithm 2. Based on the predicted value if

- greater than the allocated resources, it increases the number of pod;
- lower than Lower threshold, it decreases the number of pod;
- not greater than the allocated resources or lower than lower threshold, it does nothing.

TABLE 3. System specification.

System Name	CPU	RAM	Operating System	Kubernetes Version
Master	8 Cores	8 GB	Ubuntu Desktop 18.04.4	1.18.2
Slave 1	4 Cores	4 GB	Ubuntu Core 18.04.1	1.17.0
Slave 2	4 Cores	6 GB	Ubuntu Core 18.04.1	1.17.0
Slave 3	4 Cores	4 GB	Ubuntu Core 18.04.1	1.17.0

VII. EXPERIMENTATION

To evaluate our framework, we have used vIMS SIP as our case study. Telecom companies are moving towards cloud solutions in order to cope with the increasing traffic and number of subscribers. Also, the cloud will provide shorter time for deploying new services, lower OpEx, and better customer experience, according to requirements of 5G and beyond.

A. TESTBED

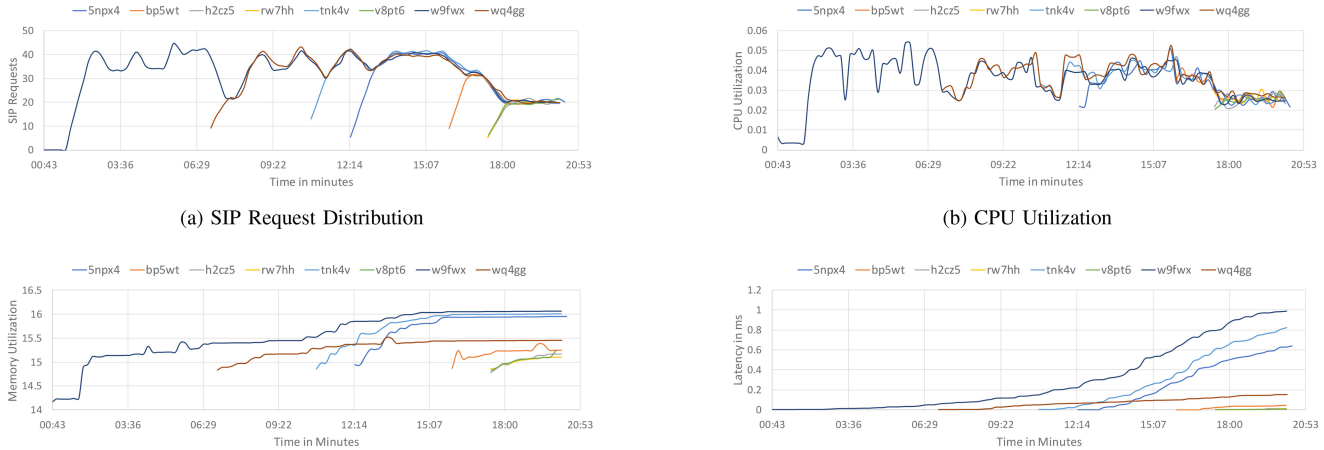
To test our proposed framework for automated monitoring and orchestration, we have used the implementation of the cloud native function (CNF) of SIP microservices running in a Kubernetes cluster. Our Kubernetes cluster contains one master and three nodes. The details of the configuration are shown in Table 3.

To simulate SIP traffic, we have used a SIP traffic generator called SIPp [54], which is a free Open Source test tool for the SIP protocol. The tool allows for different scenarios such as the number of SIP requests per second, duration of the SIP request, simultaneous concurrent SIP requests, etc. This tool is used to simulate real IMS traffic to test and validate the monitoring part of the research. The details of test scenarios are discussed in Section VII-D.

B. METRICS COLLECTED

In our research we have collected different types of metrics ranging from the application-based specific to physical system resource consumption which hosts the Kubernetes cluster. We ran the system for two hours and collected twenty five different types of metrics running the SIPp traffic generator from 1 SIP request to 200 SIP requests per second in real time at an interval of 2 seconds. The SIP requests are gradually increased over a period of 120 minutes, so that there are multiple numbers of instances for the same number of requests. The details of the categories and types of metrics are as follows. For application-specific metrics, we have collected:

- *SIP request code* is the SIP request sent by the SIPp traffic generator, which includes INVITE, REGISTER, ACK, etc.



(a) SIP Request Distribution

(b) CPU Utilization

(c) Memory Utilization

(d) Latency

FIGURE 7. Experimentation.

- *Number of SIP request value* which is the current number of SIP the system is processing;
- *SIP responses* which are sent to the SIPp traffic generator as response codes, which range from 2xx to 6xx;
- *SIP latency* is the latency of the application instance for processing the sip requests.

Pod metrics include resource consumption and information about the application that is running in the pod. The types of metrics are:

- *Name space* is the name-space where the deployment is running;
- *Deployment name* is the deployment identifier of the application;
- *Pod name* is the name of the application instance which is a unique identifier assigned to application replica by the Kubernetes;
- *Assigned CPU* is the number of cores limit assigned to the pod;
- *Assigned Memory* is the memory limit assigned to the pod;
- *CPU utilization* is the CPU utilization of the pod at that current moment;
- *Memory utilization* is the memory utilization of the pod;
- *Network usage* is the network utilization of the pod, which includes the incoming and outgoing traffic.

Node specific metrics are the data collected on the physical machines which is hosting the node, the metrics are:

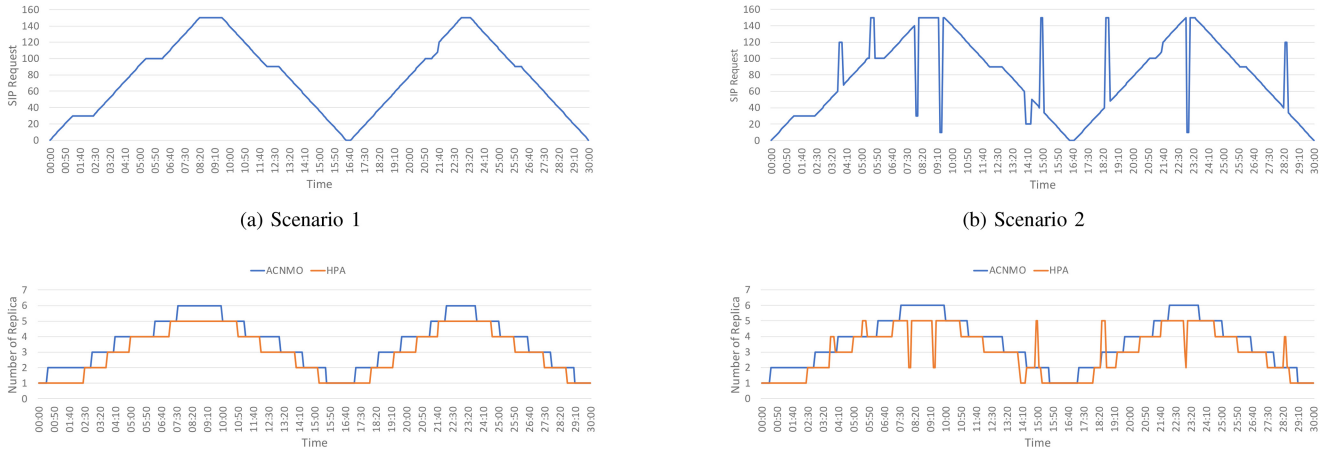
- *Node name* is the name assigned to the node;
- *Allocated CPU* is the CPU core available for the node;
- *Allocated memory* is the memory available for the node;
- *CPU utilization* is the CPU utilization of the node;
- *Memory utilization* is the memory utilization of the node;
- *Network utilization* is the network utilization of the node, which includes the incoming and outgoing traffic on the network.

C. PERFORMANCE OF OUR FRAMEWORK

In order to test the end-to-end functionality of the framework, we performed different types of experimentations. We tested our implementation with SIP traffic generator to see how the system reacts proactively. Figure 7 shows the reaction of the system when the SIP requests are increased or decreased over a 20-minute period and the number of replicas created by the system. Over the period of time, seven total replicas were created by the system and the deploying replicas takes around 4.4 seconds. Also, the traffics are divided equally among the replicas, which are provided by the Kubernetes native traffic load balancer.

In Figure 7(a) shows the distribution of SIP requests by the number of replicas. When a replica processes more than 45 requests, it increases the number of replicas and distributes the new traffic between them along with the time at which the replicas are created. From the beginning of the experimentation to the time of 6 minutes 40 seconds, the system has one replica for processing the sip requests; during this period, we have increased and decreased the number of sip requests from the SIPP. At around 5 minutes of the experimentation, we started to gradually increase the number of requests sent to the server, and our framework is able to determine that the sip requests are increasing, so it created another replica, and the Kubernetes load balancer distributed the traffic between the two replicas. After that, we reduce the requests sent to the server for a short period of time and again increase the requests sent to the system. So, our framework proactively created a replica in around 10 minutes and 12 minutes. Consequently, we have kept the number of sip requests sent to the server save over a period of three minutes and started increasing the number of sip requests. Hereafter, our framework also increases two more replicas based on the number of processes each replica is processing.

Figures 7(b) and 7(c) show the utilization of replica CPU and memory resources, respectively. At around 6 minutes 40 seconds the second replica is created, at 7 minutes of



(a) Scenario 1

(b) Scenario 2

(c) Scenario 1 test

(d) Scenario 2 test

FIGURE 8. Test Scenarios.

the experimentation the third replica is being deployed, and at around 7 minutes, 10 minutes, 12 minutes, 16 minutes, and 19 minutes the fourth, fifth, and sixth replicas are being created, respectively. In Figures 7(b) at around 2, 4, and 5 minutes, CPU utilization decreases as the number of sip requests is reduced for a short period of time. The utilization of CPU resources never exceeds 0. 55%, and whenever the utilization of CPUs exceeds 0.05, the system increases the number of replicas. From Figure 7(b), it is clear that the memory utilization of the replicas increases gradually with the number of requests it receives. The latency of each replica is shown in Figure 7(d). As shown in the figure the latency of the application is also gradually increase due to the number of sip requests each replica is receiving. The latency of the first replica is always the highest, as it has to process more SIP requests than other replicas.

To summarize this experimentation, the system is able to distribute the SIP traffic based on the CPU utilization of the replicas while keeping the system stable over a period of time. Better evaluation of the system can be deduced when we experiment the framework using different scenarios, which is explained in the next sections.

D. TEST SCENARIOS

We tested the framework using two scenarios; The first scenario has regular SIP traffic, while the second scenario has abnormal traffic behavior. We have compared our implementation with the native kubernetes horizontal pod autoscaling algorithm to show the difference between the two implementations and as well as the benefit of using our framework for performing autoscaling. In these experiments, we intend to show that our implementation is capable of ensuring the scalability of the system in any of the scenarios. The details of the scenarios are as follows;

1) SCENARIO 1

In this scenario, SIP traffic gradually increases from 0 requests to 150 requests per second, which illustrates the normal behavior of the system under observation. Figure 8(a) is the first scenario in which SIP traffic increases and decreases over time. In this case, the time period is set to 30 minutes, where the traffic is increased to 150 SIP requests per second over a period of 8 minutes, and then it gradually decreases to 0 SIP requests over a period of 6 minutes in total of around 17 minutes, and then the same process is performed increasing and decreasing for the remaining 15 minutes. This scenario is intended to simulate the ideal case in which network traffic increases and decreases over a period of time.

2) SCENARIO 2

In the second scenario, SIP traffic is similar to Scenario 1, but with a random increase or decrease in traffic for short periods. This scenario simulates abnormal traffic, which is usually caused by multiple short-term SIP requests. Figure 8 (b) shows that SIP requests are increasing or decreasing, but at some point in time the traffic increases or decreases for a short period of time.At 4 minutes, 5 minutes, 15 minutes, 18 minutes, and 28 minutes, we have increased the number of SIP requests abruptly for a very short period to simulate a sudden spike in the system. Approximately around 7 minutes, 9 minutes, 14 minutes, and 23 minutes we have decreased the number of SIP requests for a short period. In summary, as shown in the figure, there are small or large increases in traffic, as well as small or large drops in SIP traffic. This scenario tests the orchestration algorithm to determine whether it can detect random increases or decreases in traffic and acts accordingly, but without increasing or decreasing the pod.

E. EVALUATION

In order to evaluate the implementation of the framework, we have experimented with the two scenarios discussed. We have performed the experiment with the native KHPA and compared with our implementation.

KHPA uses a mathematical equation, as shown in equation (7). The number of replicas required is calculated by multiplying the number of current replicas of the pod and then summation of all CPU utilization of pods for that deployment and dividing it with the threshold of the CPU value.

$$number\ of\ replica = ceil\left[current\ Replicas \times \left(\frac{\sum pod\ CPU\ Metric}{Threshold\ CPU}\right)\right] \quad (7)$$

We have tested our implementation with KHPA using the two scenarios keeping the same input settings and with the traffic distribution evenly distributed among all replicas. The threshold limit for the CPU of a replica is set at 0.04% and the memory at 100 mb. In the experimentation, we performed several tests that include the time taken to deploy a pod along with the time required to initialize the pod to accept traffic. We have not taken into consideration the latency of the system with regard to the traffic request distribution. These experiments aim to determine how automated orchestration scales up and down based on traffic load. Also, they allow us to see how our implementation performs with the native Kubernetes scalability features. The experimental results of the scenarios are shown in Figure 8 (c) and Figure 8 (d), where HPA is the native Kubernetes auto-scaling algorithm (KHPA) and ACNMO is our auto-scaling algorithm using machine learning.

Figure 8(c) shows the scalability operations of the Kubernetes cluster for our framework and KHPA with respect to Scenario 1 in Figure 8 (a). Furthermore, Figure 8(c) provides a comparison between them with respect to the performance of the two implementations. At 2:30 minutes, 4 minutes, 5 minutes, 7 minutes, 18 minutes, 19:10 minutes, 20 minutes, and 21 minutes, KHPA increases the number of replicas based on the CPU utilization threshold of the system. However, our framework takes into account CPU, memory, and latency to predict whether or not a new replica is required. So, our implementation increases the number of replicas by 30 seconds, 3 minutes, 4:10 minutes, 6:30 minutes, 7:30 minutes, 17 minutes, 18:20 minutes, 19:30 minutes, 21 minutes, and 22 minutes based on the resource utilization of the system. Regarding the decrease of replicas, KHPA is eager to delete replicas which are happening at 10:50 minutes, 12 minutes, 14 minutes, 15:30 minutes, 24:30 minutes, 25:50 minutes, 27:30 minutes, and 29 minutes, whereas our implementation the deletion of replica occurs at 10 minutes, 11 minutes, 13:20 minutes, 14:10 minutes, 15:50 minutes, 24 minutes, 15 minutes, 26:40 minutes, 28 minutes, and 29:10 minutes. As shown in Figure 8(c), both our framework and the KHPA systems are able to increase and decrease the number of replicas based on the load of the system and the traffic load. Regarding the implementation of KHPA, the system reacts to resource utilization based on CPU utilization and increases the number of replicas when it reaches the threshold. According to our implementation, the system can scale proactively before reaching maximum resource utilization and gradually decreases replicas.

A better comparison for both implementations is evaluated when we introduce them to Scenario 2 (Figure 8 (b)). As illustrated in the figure, KHPA scales up and down with traffic, as it calculates the number of replicas required using a simple equation, but our framework keeps the number of replicas stable. KHPA increases and decreases replica number in 4 minutes, 5 minutes, 15 minutes, 18 minutes, and 28 minutes, but our system did not increase or decrease. At 7 minutes, 9 minutes, 14 minutes and 23 minutes, KHPA decreases and increases replicas within a very short amount of time as it calculates the number of replicas required using a simple equation, but our system is able to predict and keep the Kubernetes cluster stable. When the system abruptly changes KHPA, it does not perform well, whereas our framework is able to compensate for spikes and sinks in traffic loads and can maintain system stability throughout the experimentation periods.

In summary, both systems perform well when the traffic is gradually increasing and decreasing, but our framework is able to predict the future resource consumption and to make the decision to scale up before replica reaches the threshold. When the system has spikes and sinks of traffic load, KHPA does not maintain system stability, which will eventually add latency to the system. Our framework is able to foresee the spikes and sinks of the traffic and can maintain system stability, which eventually will have less latency to the system.

VIII. CONCLUSION

In this article, we proposed an architectural framework for automated orchestration for Cloud-Native applications, specifically for virtual IMS. The proposed architecture is designed to fill the gap of the limitations of the current work, which are discussed in Table 1. Our proposed framework provides efficient elastic scalability for the virtual IMS in a microservice environment.

We also implemented the proposed architecture in the Kubernetes cluster that runs the vIMS as well as a simple machine learning algorithm to predict future resource utilization. We introduced an orchestration algorithm that proactively makes the orchestration decision for scalability. We evaluated our implementation with Kubernetes native horizontal pod autoscaler and have shown that our framework is able to perform better than Kubernetes native autoscaler.

For future work, we are planning to add more orchestration operations, such as anomaly detection, load balancing, etc., using neural networks and deep learning, so that the architecture can automatically and efficiently handle different types of operations that will provide better proactive orchestration of the cloud-native IMS.

REFERENCES

[1] (Ericsson, Stockholm, Sweden). *Ericsson Mobility Report Q4 2018*. (2019). Accessed: Jan. 10, 2022. [Online]. Available: https://www.ericsson.com/assets/local/mobility-report/documents/2019/emr-q4-update-2018.pdf

[2] V. Ziegler, H. Viswanathan, H. Flinck, M. Hoffmann, V. Räisänen, and K. Hätönen, "6G architecture to connect the worlds," *IEEE Access*, vol. 8, pp. 173508–173520, 2020.

[3] Y. Xiao, G. Shi, Y. Li, W. Saad, and H. V. Poor, "Toward self-learning edge intelligence in 6G," *IEEE Commun. Mag.*, vol. 58, no. 12, pp. 34–40, Dec. 2020.

[4] "Ip-multimedia subsystem," 3GPP, Sophia Antipolis, France, Rep. 23.228, 2022. [Online]. Available: http://www.3gpp.org/technologies/keywords-acronyms/109-ims

[5] N. Banović-Ćurguz and D. Ilišević, "Mapping of QoS/QoE in 5G networks," in *Proc. 42nd Int. Conv. Inf. Commun. Technol., Electron. Microelectron. (MIPRO)*, 2019, pp. 404–408.

[6] D. Jiang and G. Liu, *An Overview of 5G Requirements*. Cham, Switzerland, Springer , 2017, [Online]. Available: https://doi.org/10.1007/978-3-319-34208-5_1

[7] (IBM Technol. Corp., Armonk, NY, USA). *Cloud in 2020: The Year of Edge, Automation and Industry-Specific Clouds*. (2019). [Online]. Available: https://www.ibm.com/blogs/cloud-computing/2019/12/12/cloud-2020-trends/

[8] (IBM Technol. Corp., Armonk, NY, USA). *Survey: Most Companies Use Multicloud, But Far Less Have Tools for Management*. (2019). [Online]. Available: https://www.ibm.com/blogs/cloud-computing/2018/10/19/survey-multicloud-management-tools/

[9] (Metaswitch Telecommun. Co., London, U.K.). *Virtualization and Containerization of the Mobile Network*. (2022). Accessed: Mar. 15, 2022. [Online]. Available: https://www.metaswitch.com/knowledge-center/white-papers/virtualization-and-containerization-of-the-mobile-network

[10] Kuberenetes. *Production-Grade Container Orchestration*. (2022). [Online]. Available: https://kubernetes.io/

[11] (RedHat Softw. Co., Raleigh, NC, USA). *Red Hat Openshift Container Platform*. (2022). [Online]. Available: https://www.openshift.com/products/container-platform

[12] Docker. *Docker Swarm*. (2022). [Online]. Available: https://docs.docker.com/engine/swarm/

[13] (Amazon Web Ser. Cloud Comput. Co., Seattle, WA, USA). *Amazon Elastic Container Service*. (2022). [Online]. Available: https://aws.amazon.com/ecs/

[14] E. Casalicchio, *Container Orchestration: A Survey*. Cham, Switzerland, Springer, 2019.

[15] Kubernetes. *Kubernetes Components*. (2023). [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/

[16] Kubernetes. *Kubernetes Architecture*. (2023). [Online]. Available: https://kubernetes.io/docs/concepts/architecture/

[17] "IP multimedia subsystem (IMS); stage 2 (Release 11), Version 11.10.0," 3GPP, Sophia Antipolis, France, Rep. TS-23.228, 2013.

[18] J. Rosenberg, "SIP: Session initiation protocol," Netw. Working Group, RFC 3261, 2002.

[19] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Softw.*, vol. 35, no. 3, pp. 24–35, May/Jun. 2018.

[20] X. Wan, X. Guan, T. Wang, G. Bai, and B.-Y. Choi, "Application deployment using microservice and docker containers: Framework and optimization," *J. Netw. Comput. Appl.*, vol. 119, pp. 97–109, Oct. 2018.

[21] R. Wang, M. Imran, and K. Saleem, "A microservice recommendation mechanism based on mobile architecture," *J. Netw. Comput. Appl.*, vol. 152, Feb. 2020, Art. no. 102510.

[22] Kubernetes. *Horizontal Pod Autoscaler*. (2022). [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[23] M. Großmann and C. Klug, "Monitoring container services at the network edge," in *Proc. 29th Int. Teletraffic Congr. (ITC)*, 2017, pp. 130–133.

[24] F. Moradi, C. Flinta, A. Johnsson, and C. Meirosu, "ConMon: An automated container based network performance monitoring system," in *Proc. IFIP/IEEE Symp. Integr. Netw. Service Manag. (IM)*, 2017, pp. 54–62.

[25] E. Casalicchio, "Autonomic orchestration of containers: Problem definition and research challenges," in *Proc. 10th EAI Int. Conf. Perform. Eval. Methodol. Tools. EAI*, 2017, pp. 287–290.

[26] D. Kim, H. Muhammad, E. Kim, S. Helal, and C. Lee, "TOSCA-based and federation-aware cloud orchestration for Kubernetes container platform," *Appl. Sci.*, vol. 9, no. 1, p. 191, 2019.

[27] A. Pelaez, A. Quiroz, and M. Parashar, "Dynamic adaptation of policies using machine learning," in *Proc. 16th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGrid)*, 2016, pp. 501–510.

[28] D.-H. Luong, H.-T. Thieu, A. Outtagarts, and Y. Ghamri-Doudane, "Cloudification and autoscaling orchestration for container-based mobile networks toward 5G: Experimentation, challenges and perspectives," in *Proc. IEEE 87th Veh. Technol. Conf. (VTC Spring)*, 2018, pp. 1–7.

[29] D.-H. Luong, H.-T. Thieu, A. Outtagarts, and Y. Ghamri-Doudane, "Predictive autoscaling orchestration for cloud-native telecom microservices," in *Proc. IEEE 5G World Forum (5GWF)*, 2018, pp. 153–158.

[30] H. Sami, A. Mourad, H. Otrok, and J. Bentahar, "FScaler: Automatic resource scaling of containers in fog clusters using reinforcement learning," in *Proc. Int. Wireless Commun. Mobile Comput. (IWCMC)*, 2020, pp. 1824–1829.

[31] P. Farhat, S. Arisdakessian, O. A. Wahab, A. Mourad, and H. Ould-Slimane, "Machine learning based container placement in on-demand clustered fogs," in *Proc. Int. Wireless Commun. Mobile Comput. (IWCMC)*, 2022, pp. 1250–1255.

[32] M. Adhikari and S. N. Srirama, "Multi-objective accelerated particle swarm optimization with a container-based scheduling for Internet-of-Things in cloud environment," *J. Netw. Comput. Appl.*, vol. 137, pp. 35–61, Jul. 2019.

[33] J. Rahman and P. Lama, "Predicting the end-to-end tail latency of containerized microservices in the cloud," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, 2019, pp. 200–210.

[34] A. Mazloomi, H. Sami, J. Bentahar, H. Otrok, and A. Mourad, "Reinforcement learning framework for server placement and workload allocation in multiaccess edge computing," *IEEE Internet Things J.*, pp. 1–1, vol. 10, no. 2, pp. 1376–1390, Jan. 2023.

[35] S. Li, J. Huang, J. Hu, and B. Cheng, "QoE-DEER: A QoE-aware decentralized resource allocation scheme for edge computing," *IEEE Trans. Cogn. Commun. Netw.*, vol. 8, no. 2, pp. 1059–1073, Jun. 2022.

[36] E. Casalicchio, "A study on performance measures for auto-scaling CPU-intensive containerized applications," *Clust. Comput.*, vol. 22, no.3, pp. 995–1006, 2019.

[37] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in Kubernetes for elastic container orchestration," *Sensors*, vol. 20, no. 16, p. 4621, 2020.

[38] S. Taherizadeh, V. Stankovski, and J. Cho, "Dynamic multi-level auto-scaling rules for containerized applications," *Comput. J.*, vol. 62, no. 2, pp. 174–197, Feb. 2019.

[39] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of Kubernetes pods," in *Proc. IEEE/IFIP Netw. Oper. Manag. Symp.*, 2020, pp. 1–5.

[40] S. N. Srirama, M. Adhikari, and S. Paul, "Application deployment using containers with auto-scaling for microservices in cloud environment," *J. Netw. Comput. Appl.*, vol. 160, Jun. 2020, Art. no. 102629.

[41] S. Barrachina-Muñoz, J. Baranda, M. Payaró, and J. Mangues-Bafalluy, "Intent-based orchestration for application relocation in a 5G cloud-native platform," in *Proc. IEEE Conf. Netw. Funct. Virtual. Softw. Defined Netw. (NFV-SDN)*, 2022, pp. 94–95.

[42] C. Harrison, C. R. Kirkpatrick, and I. Dutra, "Bioinformatics computational cluster batch task profiling with machine learning for failure prediction," 2018, *arXiv:1812.09537*.

[43] N. Naik, "Applying computational intelligence for enhancing the dependability of multi-cloud systems using docker swarm," in *Proc. IEEE Symp. Ser. Comput. Intell. (SSCI)*, 2016, pp. 1–7.

[44] Y. Brun et al., "Engineering self-adaptive systems through feedback loops," in *Software Engineering for Self-Adaptive Systems*. Berlin, Germany, Springer, 2009, pp. 48–70.

[45] Kubernetes. *Kubernetes Api*. (2022). [Online]. Available: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.11/

[46] Prometheus. "Prometheus http Api." 2022. [Online]. Available: https://prometheus.io/docs/prometheus/latest/querying/api/

[47] Keras. "Keras: The python deep learning library." 2022. [Online]. Available: https://keras.io/

[48] Scikit-learn. *Scikit-Learn Machine Learning in Python*. (2022). [Online]. Available: https://scikit-learn.org
[49] S. Benmakrelouf, N. Kara, H. Tout, R. Rabipour, and C. Edstrom, "Resource needs prediction in virtualized systems: Generic proactive and self-adaptive solution," *J. Netw. Comput. Appl.*, vol. 148, Dec. 2019, Art. no. 102443.
[50] H. Tout, C. Talhi, N. Kara, and A. Mourad, "Selective mobile cloud offloading to augment multi-persona performance and viability," *IEEE Trans. Cloud Comput.*, vol. 7, no. 2, pp. 314–328, Apr.–Jun. 2019.
[51] H. Tout, C. Talhi, N. Kara, and A. Mourad, "Smart mobile computation offloading: Centralized selective and multi-objective approach," *Expert Syst. Appl.*, vol. 80, pp. 1–13, Sep. 2017.
[52] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statist. Comput.*, vol. 14, no. 3, pp. 199–222, 2004.
[53] M. Awad and R. Khanna, *Support Vector Regression*. Berkeley, CA, USA: Apress, 2015, pp. 67–80. [Online]. Available: https://doi.org/10.1007/978-1-4302-5990-9_4
[54] SIPp. "Open source SIP traffic generator." 2022. [Online]. Available: http://sipp.sourceforge.net/

RASEL CHOWDHURY received the M.Sc. degree in information technology engineering from the École de Technologie Supérieure, University of Québec, Montreal, QC, Canada, in 2018, where he is currently pursuing the Ph.D. degree in software engineering.

His research interests include the optimization and management of cloud infrastructure services, cloud native orchestration using MLOPs, cyber security, and the security and privacy of IoT, IoE, and Android.

CHAMSEDDINE TALHI received the Ph.D. degree in computer science from Laval University, Québec City, QC, Canada, in 2007.

He is currently a Full Professor with the Department of Software Engineering and IT, École de Technologie Supérieure, University of Québec, Montreal, QC, Canada. He is leading a research group investigating efficient security mechanisms for smartphones, the Internet of Things, and edge and cloud infrastructures. His current research interests include cloud native telco services management and security, DevOps security, and federated learning for mobile cloud and IoT.

HAKIMA OULD-SLIMANE received the Ph.D. degree in computer science from Laval University, Québec City, QC, Canada, in 2011.

She is currently a Professor with the Department of Mathematics and Computer Science, Université de Québec à Trois-Rivières, Trois-Rivières, QC, Canada. Her research interests include information security, cyber resilience, homomorphic encryption, federated learning, preserving data privacy in smart environments, machine learning-based intrusion detection, access control, optimization of security mechanisms, and security of social networks.

AZZAM MOURAD (Senior Member, IEEE) received the M.Sc. degree in CS from Laval University, Québec City, QC, Canada, in 2003, and the Ph.D. degree in ECE from Concordia University, Montreal, QC, Canada, in 2008.

He is currently a Professor of Computer Science and the Founding Director of the Cyber Security Systems and Applied AI Research Center, Lebanese American University, Beirut, Lebanon; a Visiting Professor of Computer Science with New York University Abu Dhabi, Abu Dhabi, UAE; and an affiliate Professor with the Software Engineering and IT Department, École de Technologie Supérieure, Montreal. His research interests include cyber security, federated machine learning, networks, and service optimization and management targeting IoT and IoV, cloud, fog, and edge computing, and vehicular and mobile networks. He was the General Chair of IWCMC2020, the General Co-Chair of WiMob2016, and the track chair, a TPC member, and a reviewer for several prestigious journals and conferences. He has served/serves as an Associate Editor for IEEE TRANSACTIONS ON SERVICES COMPUTING, IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT, IEEE NETWORK, IEEE OPEN JOURNAL OF THE COMMUNICATIONS SOCIETY, IET QUANTUM COMMUNICATION, and IEEE COMMUNICATIONS LETTERS.