Simulation Modelling Practice and Theory 123 (2023) 102713

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

Modeling and optimizing micro-service based cloud elastic management system

Tariq Daradkeh *, Anjali Agarwal

Department of Electrical and Computer Engineering, Concordia University, Montreal H3G 1M8, QC, Canada

ARTICLE INFO

Keywords:
Elastic scaling
Micro-service
Optimization
Orchestration

ABSTRACT

Cloud elasticity is a comprehensive solution that takes into account all running applications and resources, including the cloud management system. Cloud resources and applications are constantly changing in terms of capacity and behavior, implying a dynamic change in the architecture and characteristics of the cloud management system. Following the micro-service architecture pattern, the new trend in application modeling is to decompose it into components and make them standalone cooperated modules. By customizing the application operation modules to match new tasks, this design provides the application with rapid adaptation agility to changes in requirements. This paper proposes a full-stack micro-service-based elastic cloud management system that elastically scales and manages cloud resources. The proposed model focuses on the elastic scaling performance of micro-service management modules by analyzing cloud management in three areas: interactions, end-to-end delay, and communication. In addition, optimizing the decoupling and orchestration scheduling of micro-service cloud management system components to achieve elastic management scaling. To the best of our knowledge, no work has investigated cloud management as a workload required for elastic resource provisioning. In comparison to a monolithic architecture, the proposed micro-service-based model achieves 45% to 65% enhancement in elastic scaling performance.

1. Introduction

Elastic approach [1] in cloud computing is one of the fundamental requirements of the cloud service model to meet the needs of customer hosting their applications in the cloud. Many cloud elastic models are created as one single integrated unit in a cloud management system alongside other modules such as monitoring and orchestration. Elasticity is enabled by a set of operational services that allow for rapid resource monitoring, scaling, and utilization without the need for human intervention. The objectives are to maximize resource utilization and application efficiency while minimizing costs. To describe elasticity, the authors of [1] develop Eq. (1), where scalability refers to resource scaling, optimization to efficiency, and automation to automatic operations. Scalability has one dimension of scaling that increases the resources in this case. However, we make a minor change to the definition by requiring it to increase and decrease resources. As a result, resizing the resources is a better description in this context. Furthermore, the time of scaling is an important factor that must be considered in the definition, where resource scaling must be completed within a specific time frame. Elasticity is thus defined as a timely, automatic, and optimized method of resizing resources.

$$Elasticity = \underbrace{scalability + automation + optimization}_{auto-scaling}. \tag{1}$$

* Corresponding author.
E-mail address: tariqghd@gmail.com (T. Daradkeh).

https://doi.org/10.1016/j.simpat.2022.102713
Received 26 July 2022; Received in revised form 17 December 2022; Accepted 19 December 2022
Available online 23 December 2022
1569-190X/

In elastic actions, optimization is a required operation that takes time to find the best solution, especially in a non-determinant environment, and is defined as a non-deterministic polynomial time (NP-hard) problem, in addition to orchestration and resource configuration time. That is, a cloud elastic system must be agile in its actions, comprehensive in its decisions, interoperable in its interaction, aware of resources and applications, granular in configuration, generic in architecture, adaptable to changes, self-monitoring and evaluation, autonomous in management, and efficient in usage. These are the necessary components of the elastic management model. However, combining all of these ingredients without a good recipe that describes how and when to use them is a recipe for failure. The micro-service pattern architecture [2] is the recipe that can handle all of this.

In contrast to monolithic architecture, which designs the application as one integrated unit with high code dependency, micro-service pattern architecture deploys an application as independent components. Micro-services enable application components to be interchangeable, customizable, modifiable, and modular, providing the system with agility to adapt to changes, (growth/shrinking) scaling, availability, reliability, failure resiliency, and dynamic development. Each component in a micro-service architecture is a self-contained, independent module that can perform one or more tasks and communicate with other applications via communication protocols. These components are dynamically distributed and auto-scalable in cloud service applications. The micro-service model, on the other hand, necessitates a good architecture design for its application components as well as communication channels between them via standard API.

As the number of components increases, so does the complexity of management and communication, as the cloud environment changes dynamically [3]. To handle such a complex system a non-linear solution to describe micro-service model behavior and functionality is required. Developing an application using a micro-service architecture improves code maintenance, re-factoring, re-usability, integration, and deployment, according to [4]. Because of the high volume of application demands and resource configuration cases, resources scaling based on micro-service architecture can be tedious and undeterministic, necessitating more measurements rather than using resources threshold values for CPU and RAM utilization.

Micro-services are being expanded for use in a variety of distributed and smart systems in large-scale environments, including cloud computing, grid computing, edge computing, smart grid monitoring and control, cloud application, and cloud management. Distributed systems can be managed and interacted with in a loosely coupled manner, making system components independent and replicative. As a scaling time sensitivity [5], this can impose a complex monitoring, management, communication, and synchronization model with an end to end delay.

In our previous work [6], we proposed a micro-service elastic cloud monitoring and management system in which self-monitoring and management components are automatically created, duplicated, consolidated, collocated, and de-located on the fly. The goal of this work is to model and optimize a generic holistic elastic cloud management system based on the architecture of the micro-service pattern. Furthermore, elastic components provisioning capabilities for cloud management modules at all operational levels are taken into account. The novel aspect of this work is the use of optimization on a micro-service pattern architecture model to achieve proper cloud management orchestration and elastic resource provisioning while accounting for cloud management overhead and workload diversity. Cloud management components are dynamically allocated and expanded in response to cloud resource scheduling procedures that take end-to-end delay, communication complexity, resource scaling, and component dependency into account. This work is in comparison to a traditional cloud management system, which has one centralized management module and traditional scheduling algorithms. Cloud management designed in accordance with the monolithic architecture managing different cloud layers of service oriented model has several limitations. This style of architecture does not have any correlation between application workload and cloud resource status. Moreover, the orchestration actions occur in fixed time intervals basis. This limits its adaptability for dynamic management orchestration and reduces the accuracy of elastic scaling.

This paper is structured as follows: The related work is presented in Section 2. Section 3 discusses micro-service elastic architecture. Section 4 presents a full stack micro-service cloud elastic model, and Section 5 discusses an optimization module. Sections 6 and 7 cover implementation and performance evaluation. Finally, in Section 8, the conclusion is presented.

2. Related work

Integrating various service component modules allows for the creation of a cloud elastic management system. The general cloud management model considers four common components, which include monitoring, cloud scheduling and scaling, system architecture (micro-services), and cloud scaling (orchestration) optimization. The end-to-end time delay of service modules, communication overhead, modules colocation, number of dependency chains, and redundancy are all integrating aspects of these components. The majority of the related work effort was concentrated on one or two of these module integration aspects, rather than all of them, as has been proposed in this paper. The research for each module is listed below.

2.1. Micro-service monitoring

Multiple monitoring frameworks are proposed in [6–12]. We implemented a micro-service self-monitoring and management system in our previous work [6] to manage cloud operation components and cloud management itself. The authors of [8] proposed a framework for monitoring micro-service application deployment across multiple cloud environments. The model is validated using the Book-Shop application on Amazon Web Services (AWS) and Microsoft Azure. The authors refined their model to account for multiple visualization types, multiple cloud providers, and multiple micro-service communication APIs. The proposed framework employs two modules: a monitoring agent that runs within a container or virtual machine, and a monitoring manager that is configured as an independent module across the cloud and collects data from monitoring agents. Monitoring fog computing resources

as a Support and Confidence Based (SCB) technique is proposed in [7], to track IoT data generators and processing resource allocation. The SCB algorithm operates in three agent layers: the fog server agent, which predicts fog device information, the fog leader agent, which assigns service to devices, and the fog devices agent, which collects/sends data from/to devices. Edge NOde Resource Management (ENORM), a framework for managing cloud resources in the edge layer, is proposed in [9] to handle resource distribution expansion in fog computing. Auto-scaling was used to improve application performance, and data transmission overhead was reduced to improve communication quality. The ENORM framework is divided into three levels: (1) The upper tier layer is the primary cloud datacenter layer, (2) the middle tier layer is the edge nodes distributed in fog computing, and (3) the bottom tier layer is the end devices connected to IoT services. This framework's auto-scaling is pre-defined by a set of rules, such as a static if-statement, which is a limitation in this model for dynamic changes. A packet sensitive monitoring method was described in [10], this method employs a Linux container sensor tool, the extended Berkeley Packet Filter (eBPF), to create a real-time full stack micro-service container-based monitoring tool. Micro-service end-to-end component latency is a metric in SLA. The proposed monitoring module is built on distributed software components that communicate via REST API over deployed VMs and containers, with management provided by OpenStack, Docker, and K8s. To test the model and evaluate its performance, a book store benchmark called "Bookinfo" is used. An elastic monitoring architecture for cloud network resources proposed in [11]. The authors of this paper investigated monitoring of physical and virtual cloud network resources at various scale levels from a technical and administrative standpoint. This work focuses on resource slicing, which is the sharing of cloud computing resources such as storage, computing power, and network. The proposed architecture is based on monitoring modules that are fully integrated with the components management system and follow orchestration actions such as adding or removing monitoring components. An elastic cloud network monitoring slicer architecture is proposed by [12]. It collects data logs from cloud network components uniformly and reflects the monitoring parts as needed. In order to be a part of the general monitoring module, logs generation, sampling, tracking, and storing are evaluated in our previous works [13–15] respectively. The authors of [13] investigated the accuracy of log data sources and the performance of communication messages. Dynamic sampler that employs the point estimator method is described in [14]. Finally, [15] discusses efficient log storage and replaying using PCA and Wavelet transform methods.

In summary, all of the presented work used a limited number of cloud monitoring functionality as independent monitoring tasks, which operate at different levels of cloud operation for cloud monitoring methods based on micro-service architecture. The limitation of all previous works was that they did not consider micro-service architecture modeling complexity and management as parameters to be monitored, as well as cloud service components, which we have addressed in our proposed work.

2.2. Cloud micro-service management and scaling

In [5,16–21], micro-service pattern architecture is used to build cloud management and scaling framework or module. In [16], authors proposed a new scaling algorithm in micro-services-based application architecture, and they used an agnostic approach to deploy the application in Google Kubernetes. The deployment model takes into account application characteristics and resource requirements, and distinct micro-service conditions are used to improve cloud micro-service based application response time in the auto-scaling paradigm. An auto-scaling micro-service cloud framework is presented in [17], in which the authors used a simulated agents-based cloud environment to test cloud scaling by investigating cloud instance replication and configuration. The authors built their framework on top of the Micro-service-based Cloud Application-level Dynamic Orchestrator (MiCADO) framework [18], which supports various scaling models and orchestration algorithms. Authors in [19] proposed a micro-service driven model that employs a pattern approach to integrate standard application architecture into the micro-service model. An automatic application deployment by generating application artifacts using the micro-service composition (MICO) method, which can simplify deployed architecture transformation, integration, and deployment. Kubernetes was used for container orchestration, self-healing, resource monitoring, auto-scaling, and automatic deployment by the authors. The MICO model is based on the concept of seamlessly transferring the application abstract model into deployed running components via pipes and filters, where pipes represent connection channels and filters represent processing units.

In [20], authors proposed an Aloe framework to elastically auto-scale a Software Defined Network (SDN) controller to support IoT deployments, where they attempt to find the best location to allocate the closest resources to the IoT devices. This reduces traffic flow, delays, and fault tolerance. The authors of [20] proposed their algorithm for determining SDN controller placement using a randomized strategy and testing the selection. With a large complex setup that requires an optimization method, this may not be a good idea. In [21], an SDN-based controller for cloud transport networks using micro-service architecture is developed to achieve automatic scaling, self-healing, and adaptive scaling for network connections. A Kubernetes cloud container management extension for micro-service elastic scaling architecture is presented in [5]. The authors introduced a hierarchical multi-level elastic Kubernetes (me-kube) that works based on queuing theory for proactive and reactive actions using reinforcement learning for scaling.

Micro-service architecture was used partially for cloud application management in all presented works, with application scaling considered only with limited orchestration. In SDN allocation, a randomization technique is used, which may cause resource allocation issues, and no optimization methods are used to find the best component allocation and arrangement. All of these issues are addressed in our proposed work by taking into account workload application and cloud resource allocation and types in a micro-service cloud management system. Furthermore, component allocation and their arrangement are optimized to reflect resource scaling.

2.3. Micro-service performance evaluation

Micro-service performance evaluation is tested in [3,4,11,22,23]. Increasing services components causes unpredictable service behaviors which require a new performance evaluation. Authors in [22] investigated service modeling by focusing on service constraints and performance satisfaction as a model. Authors used Markov Decision Process as a problem formulation and used a Q-learning algorithm to solve the model. The constraints used in the composite service model conditions are quality of service (QoS), total price, and throughput response time.

A performance aware micro-service web application deployment on cloud containers style with dynamic scaling model is proposed in [3] as a Robust resource Scaling (RScale). Authors used machine learning and probability theory Gaussian Process (GP) Regression to scale cloud resources considering an end to end delay of micro-service based application workflow performance. A trusted range for resources demand is predicted to setup containers based resource provisioning to respond quickly to changes and micro-service components flow conditions. Chameleon testbed [24] is used in the implementation using KVM as VM virtualization, and Docker for container virtualization. The orchestration of the container is managed by Kubernetes. Micro-service performance overhead is investigated in [23], where a performance modeling is proposed in micro-service based application workflow to evaluate service performance and predict it in a cloud environment. A heuristic scheduling arrangement algorithm is used to minimize micro-service components end to end delay while respecting budget cost. Authors in this work used mathematical and stochastic method to model each micro-service container's service, processing time, request, business processing, and transactions. Works in [23], and [10] considered performance evaluation by testing end-to-end delay of micro-service components. A micro-service framework Cloud Native, which demonstrates complex management for dynamic setup, is tested in [11]. The main goal of that test is to understand micro-service behaviors in cloud systems. A micro-service response model used to predict cloud auto-scaling requirements to reduce the delay is proposed in [4]. Authors in this work used stress testing to model the micro-service response time, and trace-driven to emulate performance and resource provisioning by forecasting the workload using different regression methods like Elastic Net (EN), Linear Regression (LR), Polynomial Regression (PR), Decision Tree Regression (DTR), and Random Decision Forest (RDF).

The evaluation methods that have been presented in the literature focused on the performance of micro-services deployed as applications in cloud containers. End-to-end delay, micro-service component flow conditions and redundancy, and application response time performance are the metrics used to evaluate the models. However, resource capacity accuracy and elastic scaling accuracy are not considered as metrics. In our work, we have considered elastic scaling accuracy as well as resource capacity accuracy in addition to the other metrics considered in the literature. To achieve elastic component scaling, service components allocation is optimized in two levels of components arrangement and allocation.

2.4. Cloud scaling optimization

Optimization is used in [25–31] targeting different objectives, which depend on the problem targets. Optimization models are implemented using heuristic math functions and algorithms. A cost stochastic optimization model for elastic micro-service deployment that considers workload, service performance (QoS), and operation service rate is proposed in [25]. Authors used the Lyapunov optimization framework to tradeoff between the service performance and cost. There are several sub-models for the whole cloud architecture that represents the workload as task requests, the computation resources as Pod, the stochastic operation as a queuing model, cost as Pod deployment in a physical server, and operation tasks as running replica. A final elastic deployment for container-based micro-services with a cost-efficient in edge computing is achieved with this model. An analytical model for cloud software defined network (SDN) controller allocation and optimization of IaaS cloud layer orchestration is proposed in [26] to minimize cloud service components faults. Authors focused on software serviced switches controllers arrangement to maintain cloud operation controllers with minimum components, communication overhead, and failures. The model follows an undirected graph for the SDN switches topology where nodes (vertices) represent dominant switches and connection (edges) represent the traffic flow. A cloud capacity scaling is proposed in [27], where scaling is measured based on workload prediction and cost optimization as a scaling policy. Authors used the scaling policies derivation tool (SPDT) in the micro-service architecture model and test five different scaling policies (Naive, Best Resource Pair, Only-Delta-Load, Always-Resize, and Resize when Beneficial) to evaluate cloud capacity scaling concerning cost and performance. A datacenter electrical cost aware for cloud workload scheduling method is proposed in [28], which considers distributed datacenter and electrical smart grid. Authors introduce a cost objective function that maintains the power cost of the smart grid and datacenter workload scheduling service burden. The proposed solution works by applying a dual decomposition method solving multi objective functions to optimize workload scheduling among distributed datacenters. The workload, datacenter power, and smart grid cost are modeled mathematically by formulating a general multi objectives cost function that is solved by Receding Horizon Control (RHC) algorithm. A power cost elasticity model is proposed in [29], where authors develop an algorithm based on a stochastic model for electricity price based on a rational factor φ that represents maximum to minimum electricity price. An optimization using dynamic programming for the Markov model of electricity changes is used to reduce power cost in datacenter scaling responding to workload and electricity cost changing. The cost function goal is to minimize the number of operation servers that must run to serve workload tasks during a certain time, where the time to finish the task is reduced once the electricity price is high. A heuristic scheduling algorithm (FLAVOUR) is proposed in [30] for edge computing that reduces service latency in micro-service architecture. The algorithm uses stochastic modeling of service time that is solved using the Lyapunov optimization method considering network performance and task completion. Authors in [31] proposed a queuing model with time series ARIMA prediction method to achieve elastic scaling of web applications model. Kubernetes is used for container management for elastic scaling to optimize the number of replicas.

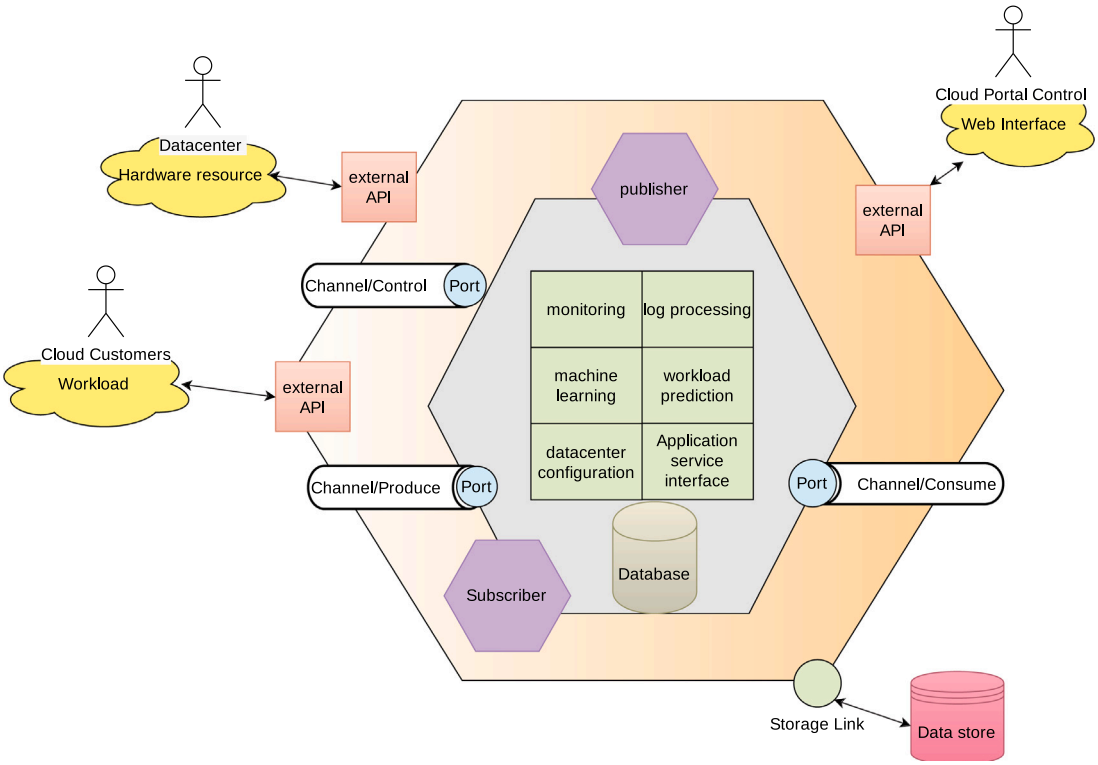

Fig. 1. Generic monolithic hexagonal service module.

All of the preceding approaches regard scaling optimization as resource scheduling and orchestration by controlling the number of running hosts and cost (to save energy and maintain service performance). The cost function and optimization objectives are combined in a single cloud operation layer. Meanwhile, we want to achieve elastic scaling of cloud computing resources and cloud management components that take into account cloud application workload and cloud datacenter scaling capabilities.

3. Micro-services pattern components architecture

The proposed model is implemented based on micro-services pattern [32]. The basic low-level component is a service module that has three communication channels and one or more core function units for processing/service operation. These channels are used for communication between system components. Each basic building block module communicates by reading (consuming), writing (producing), and synchronizing (controlling) between system components. The function unit ($processing/service$) operation defines the component tasks and functionalities. Six major functional services comprise the cloud elastic micro-service model: (1) monitoring (sampling, reading, reporting, serializing), (2) log processing (storing, replaying, analysis, reshaping, reporting), (3) machine learning (dataset preparation, training, clustering, classification, validation, model reporting), (4) workload prediction (applied pre-built analytical model and machine learning prediction models), (5) datacenter configuration (applied machine learning classifier, scheduling, optimization, scaling boundaries, scaling capacity), and (6) datacenter configuration (applied machine learning classifier, scheduling, optimization (web interfaces, external API interfaces, REST interfaces, data structure manager, application instances, service instances, management instances, database instances, messages parser, registrar–subscriber, publisher). The monolithic structure of the entire cloud management elastic model using hexagonal architecture is depicted in Fig. 1. In this model, all elastic management components operate as a single integrated service unit, with internal communication between them. External REST API is used to read and update the external world (datacenter resource status and control, workload demands, and management portal). The log and data are collected from distributed API agents and delivered to the management node (all to one communication style), and control is distributed from management to all actuator agents (one to all communication style).

Applying micro-services to this architecture is a difficult task because each component has a set of working tasks that may run in a single service module or in separate modules. The design is also influenced by the communication complexity and service latency between application components. The majority of the literature focuses on micro-service end-to-end delay for cloud-running applications, but not on cloud management and control applications that take into account elastic management and scaling, which is the focus of this paper. Our goal is to create a cloud management system that uses a generic and optimal micro-service pattern architecture, responds quickly to datacenter and workload changes, and achieves elastic scaling with a hierarchical communication style.

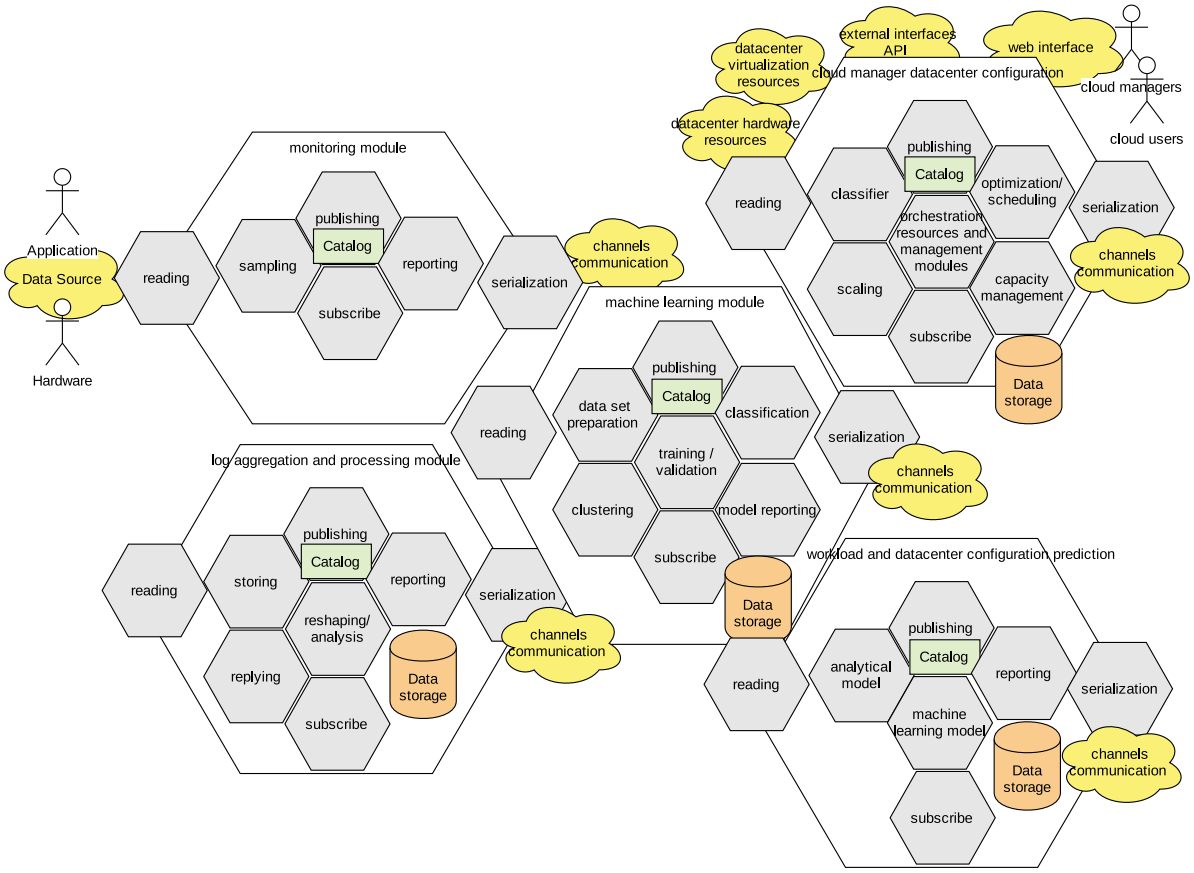

Fig. 2. Generic micro-service hexagonal service module.

The proposed micro-service hexagonal architecture, depicted in Fig. 2, decouples the elastic cloud management system into separate modules. The architecture depicts the five main modules, which can be assigned to one or more assembled tasks in one or multiple dependent components. The cloud external agent is represented as hardware, software, communication channels, external API interfaces, and cloud users and administrators. There are four generic services in each module of the system: (1) reading is the input gate with two lanes (data and control), (2) serialization is the output gate with a standard message format that uses JSON encoding by converting a data object into a JSON string. Data messages are also presented as objects and are serialized as an array of objects each time, (3) The registration agent that handles dynamism in modules is the subscriber. Each module includes a sign-up API for other service components, as well as a service class that describes its functionality and participation task, and (4) Publisher is the public interface for that module with the catalogue map for the system's running module. Publisher presents an addressing schema for the system that allows each module in the system to know all components and their location (in this case, location is the distance between communication components to access cost tuple {delay, bandwidth, transaction number, and communication direction}, and in turn is to publish the component node unique ID and service class). Our focus is on the services rather than security, so a simple authentication using a pre-shared key is used. We leave out the security section because it is a separate issue that employs attributes-based encryption and authentication. The remaining components are specialized components that run in a single generalized module as a single service or in conjunction with other services depending on decoupling conditions. In the monitoring module, for example, there are two specialized services, sampling and reporting, which can run in either a single general module or two separate modules. The same tasks are carried out for other specialized services. It does not, however, group components in the monitoring module with components in the aggregation module. A sampling and log storing decouple example, because it makes no sense to sample the log and store it at its source, which will require sampling again, and so on. The following section contains a comprehensive discussion of the micro-service model design pattern and operations.

4. Full stack cloud elastic micro-service modules

The design pattern used is following the software engineering concept "No Silver Bullet-Essence and Accident in Software Engineering" [33], which is defined "There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity" [33]. Cloud

management application architecture develops dynamically where its components are evolving and changing during time, and cloud customer applications are changing as well. A full stack of cloud elastic management model is proposed, where there are multiple running components that work in cooperation to serve cloud tenant application and its services in a way that reduces the cost and maintain the performance. The management modules are spread in all cloud service layers and communicate via communication channels. The proposed cloud management system follows the software engineering concept "No Silver Bullet-Essence and Accident", since it was not built in one step or in a single stage. Rather, it has evolved and grown over time by employing distinct operation modules for cloud services and resource management (orchestration and scheduling), and by measuring various types of cloud management system metrics and implementing micro-service architecture as a holistic platform architecture that takes into account all cloud layers. The proposed work focuses on the advantages of micro-services in handling complex systems and avoiding model limitation issues by applying optimization to find the tradeoff configuration setup.

The proposed micro-service hexagonal architecture, depicted in Fig. 2, decouples the elastic cloud management system into separate modules. The architecture depicts the five main modules, which can be assigned to one or more assembled tasks in one or multiple dependent components. The cloud external agent is represented as hardware, software, communication channels, external API interfaces, and cloud users and administrators. There are four generic services in each module of the system: (1) reading is the input gate with two lanes (data and control), (2) serialization is the output gate with a standard message format that uses JSON encoding by converting a data object into a JSON string. Data messages are also presented as objects and are serialized as an array of objects each time, (3) The registration agent that handles dynamism in modules is the subscriber. Each module includes a sign-up API for other service components, as well as a service class that describes its functionality and participation task, and (4) Publisher is the public interface for that module with the catalogue map for the system's running module. Publisher presents an addressing schema for the system that allows each module in the system to know all components and their location (in this case, location is the distance between communication components to access cost tuple {delay, bandwidth, transaction number, and communication direction}, and in turn is to publish the component node unique ID and service class). Our focus is on the services rather than security, so a simple authentication using a pre-shared key is used. We leave out the security section because it is a separate issue that employs attributes-based encryption and authentication. The remaining components are specialized components that run in a single generalized module as a single service or in conjunction with other services depending on decoupling conditions. In the monitoring module, for example, there are two specialized services, sampling and reporting, which can run in either a single general module or two separate modules. The same tasks are carried out for other specialized services. It does not, however, group components in the monitoring module with components in the aggregation module. A sampling and log storing decouple example, because it makes no sense to sample the log and store it at its source, which will require sampling again, and so on. The following section contains a comprehensive discussion of the micro-service model design pattern and operations.

The challenge in this dynamic management model is to make the best actions for cloud serving resources and cloud monitoring components in an elastic way reducing end to end delay of inter-communications between management modules, and reducing the cost by minimizing number of operation components. The requirements conditions will raise five challenging dilemmas in the model of low level components topology and decoupling scheme as following: (1) allocation of system modules (location means the cost of running modules with respect to others), (2) number of services per modules (granularity of module), (3) number of modules in one operation segment, (4) service supply chains (services dependency sequence), and (5) services segregation.

As a result of these five conditions and based on full stack design, a three-dimensional relational diagram is generated in Fig. 3 that depicts the mathematical model for the problem formulation. According to Fig. 2 there are five main service operation SO modules represented as a set $SO = \{so_1, so_2, so_3, so_4, so_5\}$. These service applications can run in one or multiple micro-service components $C = \{c_1, c_2, \ldots, c_n\}$, where n is an integer number. Services in one component might be decoupled and deployed on lower granular multiple components. The cooperated chain of end to end tasks between separate micro-services components that formulate one main operation service task is defined as a service operation module so_i, and the lower granular components are called service operation segment SOS. For example in the full-stack model, the monitoring module of cloud resources that collects the log using a sampling component is not at the same running monitoring component. The cooperation between the sampling component at cloud resources sides sending the log to the monitoring component of reporting is one complete service operation segment. There are three characteristic directions for the system. First, vertical relation defines the dependency between system components and their chaining (number of dependent components) to implement the full modules tasks. Second, the horizontal relation represents the system decoupling and the granularity level of each service component. And the third is the depth relation that represents the component redundancy and distribution. The surface layer S represents the full model service of the full stack cloud elastic model. The arrangements of each service layer are calculated based on a cost matrix tuple, which is a measurement of the component's location. The tuple matrix includes communication transaction number, end to end delay, number of components in the dependent chain, number of redundant components, configuration time condition, and capacity. These metrics are represented as a vector of Greek letters $\{A, B, X, \Delta, E, \Phi\}$.

The cost of the communication transactions A is found by the sum of exchange messages at all surface layers $\sum_{i=1}^{5} \alpha s_i$ and number of communication messages between different surface layers αS, where αs_i represents number of communication at same layer i. Communications at one surface layer αs_i is defined by total number of messages exchanged between all service running components list C_i with number N_i at layer i. Based on the Fig. 3, at surface layer i communication links transactions CK_i of layer i are connected with the graph edges Ed_i, where the vertices represent the component nodes list C_i. Each component has number of direct links with other components at the same surface level represented in the graph as edges ec_{ji}, number of edges of component j in surface i. Each direct link is responsible for sending messages to other components, which are counted as messages exchanged in an acyclic order (this means no loops in sending messages, CK_i also are directed links) with edge component sending operator

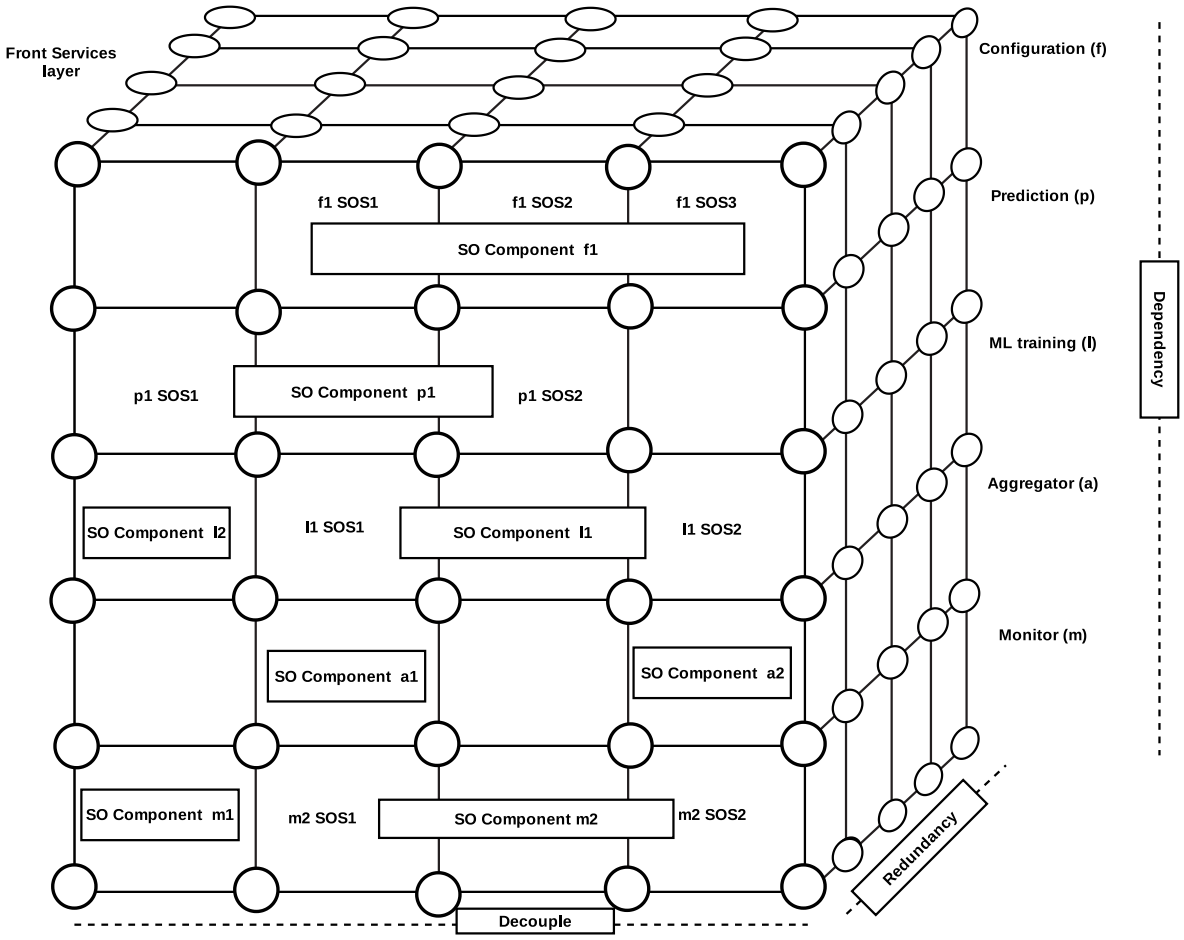

Fig. 3. Full stack operational segments and components relations.

ω_{ij}. According to that $\alpha c_{ij} = \omega_{ij} \times ec_{ij} \times ck_{ij}$, where each node has ck_{ij} communication links multiply by sending factor ω_{ij} multiply by edges ec_{ji} (note that one edge might contain multiple communication links). The total cost of messages exchanges A and its relations formulae are defined in Eq. (2). Now the total number of communication between surfaces αS are found by counting the total number of cross edges M that are stored in CE list between layers. Where the cross edge is the summation of links between all layers $CE = \sum_{i=1}^{5} \sum_{j=1}^{5} ce_{s_i,s_j}$, where $i \neq j$. Then $ce_{s_i,s_j} = \{\sum_{k=1}^{N_i} ck_{ik}\}_{s_i \rightarrow s_j}$ the directed links from layer s_i to layer s_j and k is the index of the component at layer i.

$$
\begin{aligned}
A &= \sum_{i=1}^{5} \alpha s_i + \alpha S, \\
where\ \alpha s_i &= \sum_{j=1}^{N_i} \alpha c_{ij}, \\
\alpha S &= \sum_{i=1}^{M} ce_i \times ec_i \times \omega_i.
\end{aligned}
\tag{2}
$$

End to end delay B, dependency chain X, redundancy Δ, configuration time E, and capacity Φ metrics are related. The dependency is driven from the graph as predecessor and successor relation, where each component node (predecessor) is dependent on an input from other component node (successor) that constructs operation service unit (os_i). Redundancy is needed for performance and availability, the sibling with the same active services are considered redundant. Redundant components might construct multiple flows, which follow a dependency path with aggregation hubs as intermediate join component. Multiple dependency paths might also be created to serve one service component, i.e. to finish the machine learning training model. One to seven components might be created in dependent order. However, with redundancy enabled it may go to m components. End to end delay is found by accumulating processing and communication time cost in the longest service dependent path of the full stack task.

Redundancy measure configuration and processing time, communication affinity and collocation, and resources capacity are used to measure the best performance. The configuration time E is the time of each individual service component needed to communicate and process the service requests that is assigned. It is measured by time of component communication transactions received rck_{ij} multiply by communication time ct_{ij} to receive a message, multiply by processing time pt. It assumes a constant processing time factor for all operations. $E = rck_{ij} \times ct_{ij} \times pt$ when capacity is enough at that location $\Phi_l > \phi_{req}$, where ϕ_{req} is the required capacity size. Capacity plays an important role in the resources allocation and system shape, because if there is no room for a new component needed as its successor it might cost to create the new successor component far away and increase the communication cost and the configuration cost, or it might cause to reallocate the whole service chain to a new location. In this work, the assumption is that the datacenter resources (network, storage, computation and memory) are homologous and allocation happens if all resources are available in that location l. Then $\Phi_l = \sum_{i=1}^{L} \phi_i$ and $\phi_i = \{CPU, RAM, Disk, NetSwitch\}_i$.

Dependency and redundancy are formulated as following. Dependency $X = \sum_{p=1}^{P} Path_p$ where $Path_p = \sum_{i=1}^{N} path_{os_i}$, $path_{os_i} = \forall c_{ij} \in os_i \cap \{c_{ij}\ read\ c_{i(j-1)} \cap c_{ij} \subseteq \delta_i\}$. Redundancy $\Delta = \sum_{i}^{N} \delta_i$, where $\delta_i = \forall\{\Phi_l > \Phi_{req}\} \cap E_i \geq E_{th}$, where E_{th} is a time threshold values defined by SLA agreement or required service maximum time to handle management time, $E_{th} = Max\{service\ time,\ SLAtime\}$. End to end delay is the longest dependent path of operation service $B = Max\{X_{os}\}$.

To review the full stack architecture relationship, consider one example of cloud resources scaling in response to workload demands. The model addressed elastic orchestration and scheduling of micro-service cloud management components for the management system as well as cloud resource orchestration and scheduling. Monitoring components will run to measure cloud resources and application demands, as shown in Fig. 3. The monitoring components are distributed across cloud datacenter resources and run at all cloud layers (IaaS, PaaS, AaaS), collecting logs and metrics about hardware, software, platforms, and applications. Reading data sources, sampling logs, and updating cloud service management components are all tasks that the monitoring component can perform. As a result, when new hardware is turned on and used in production, a new set of monitoring components is created and installed at that location. The number of monitoring components in the new servers at each cloud service layer is determined by micro-service metrics, communication complexity, dependency chain, and component performance. By investigating how the monitoring component will be created and deployed, the cloud management system will fork new components (as a clone from the basic management module) and run the services for each cloud layer. The system will begin communicating and synchronizing messages, as well as building the dependencies chain, determining the delay for each component and how many data sources it can read in order to achieve performance conditions. Each monitoring component will be a data producer and must send updates to consumer components in the next cloud management layer, such as the machine learning service layer, that uses the log. However, because the machine learning service layer handles a variety of operations such as system scheduling, optimization, and orchestrator, each service component requires different types of logging. ML is required for resource scheduling that needs IaaS resources logs and workload demands as a low-level resource request (CPU, RAM, Disk). However, for application components scheduling and prediction, ML requires application traces such as running time, number of requests, and number of threads for the system. Log aggregation is important for reducing model communication complexity by segregating need logs for specific components or aggregating common logs. The log summary will be fed into the necessary data consumer of machine learning and predictor modules, which will be used in system management to determine the best datacenter configuration and micro-service cloud management component arrangement.

The full stack cloud management elastic model is implemented based on the aforementioned relations that are used to model and optimize management operation and cost. The following sections show the self-adaptive management scheme, the optimization and the orchestration.

5. Optimization module

The elastic management and provisioning depends on optimizing the provisioned resources to best match cloud workload demand, and optimizing allocation of the cloud management component over cloud service. Cloud management components and cloud running applications are creating the overall cloud workload demands. Prediction of that demand will help to configure cloud datacenter resources prior of time, and find the best match resources configuration with the demand. Fig. 4 depicts the architecture for finding optimal configuration for cloud management in an operation segment. Finding optimal configuration for all segments in the cloud management modules is a tedious task. It requires cooperation, communication and processing between all cloud management components to satisfy and respect operation conditions in the SLA and minimize the cost. The ultimate goal of this part is to achieve the best actions that reduce the end to end delay B keeping a good performance configuration time E with minimum cost and resources capacity Φ. But as in all real life problems, there is a contradiction between end to end delay and performance with respect to collocation, capacity and communication.

The two main system metrics targeted are to reduce the cost and maintain good performance, which are required to minimize all other system metrics, like number of communication transactions A, end to end delay B, number of dependency chain X, number

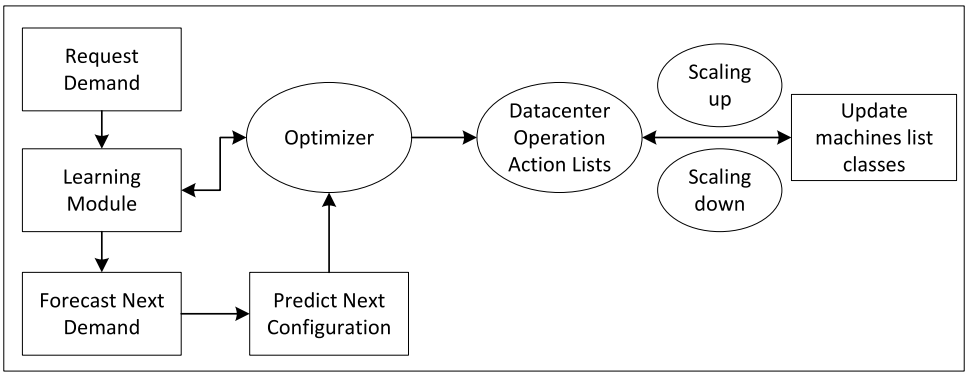

Fig. 4. Predictor and optimizer architecture.

of redundant components Δ, configuration time condition E and capacity Φ as individual targets, as Eq. (3) depicts.

$$
\begin{aligned}
Min\ A &= \sum_{i=1}^{5}\sum_{j=1}^{N_i} \alpha c_{ij} + \sum_{i=1}^{M} ce_i \times ec_i \times \omega_i,\\
Min\ B &= Max\{X_{os}\},\\
Min\ X &= \sum_{p=1}^{P}\sum_{i=1}^{N} path_{os_{pi}},\\
&where\\
path_{os_i} &= \forall c_{ij} \in os_i \ \cap\ \{c_{ij}\ read\ c_{i(j-1)} \ \cap\ c_{ij} \subseteq \delta_i\},\\
\delta_i &= \forall\{\Phi_l > \Phi_{req}\} \cap E_i \geq E_{th}\\
Min\ \Delta &= \sum_{i}^{N} \forall\{\Phi_i > \Phi_{req}\} \cap E_i \geq E_{th},\\
Min\ E &= rck_{ij} \times ct_{ij} \times pt,\\
Min\ \Phi_l &= \sum_{i=1}^{L}\{CPU, RAM, Desk, NetSwitch\}_i.
\end{aligned}
\tag{3}
$$

Operational metrics are derived in the context of cost functions and constraints formulation as shown in Eq. (3). The relation equations are developed according to the full stack operational segments metrics relation and dependency as explained in Section 4, by specifying the number of communication A, which is defined as the number of communication channels between service components at the same layer $\sum_{i=1}^{M} ce_i \times ec_i \times \omega_i$ multiplied by the number of cross communication channels between different layers $\sum_{i=1}^{5}\sum_{j=1}^{N_i} \alpha c_{ij}$. Reduced system communication complexity and overhead, as well as increased log and data flow between cloud management and component entities, will result from lowering this factor. As shown in Fig. 2, each component has read/serialization (writing) queues for data and log transactions and publisher/subscriber queues for management communication that are affected by the number of messages. End-to-end delay B is a typical outcome of dependency chaining for one operational service (os) component (one service like workload prediction). To make decisions, the operational service requires specific input, which may be passed through multiple stages, such as logs from the application layer and front end users being collected, correlated, aggregated, and reported to the prediction module. This operation will generate a path of dependent service modules that will form the chain $X = \sum_{p=1}^{P}\sum_{i=1}^{N} path_{os_{pi}}$. The path is defined by all the components $\forall cij$ belonging to an operation service $\in os_i$ and having a predecessor component to read from it $c_{i(j-1)}$ and belonging to a set of redundant components $c_{ij} \subseteq \delta_i$, with resources capacity condition $\Phi_l > \Phi_{req}$ and configuration time $E_i \geq E_{th}$. The number of redundant components Δ is evaluated in terms of performance and capacity while taking into account configuration time $\sum_{i}^{N} \forall\{\Phi_i > \Phi_{req}\} \cap E_i \geq E_{th}$. Configuration time E is the time it takes each individual component ct_{ij} to communicate and process a service request rck_{ij} in relation to uniform time pt. Finally, the competent operation's resource capacity is determined by adding the used resources of each participating computing node that runs components $\sum_{i=1}^{L}\{CPU, RAM, Desk, NetSwitch\}_i$.

Formulating one cost function to describe the problem is not visible, therefore we propose an algorithm that works by separating optimization goals with adaptive constraints to achieve the best accepted tradeoff solution in a reasonable time. In this work the cost function needed is to focus on two important factors: (1) minimize overall communication A, and (2) performance response time $R_t = E_{os}$ which indirectly will reduce end to end delay B. These two points contradict where reducing number of communications intuitively is achieved by reducing number of system components, whereas, to achieve good performance we must increase redundancy Δ and capacity Φ. Algorithm 1 depicts the optimization procedure.

Algorithm 1 begins by locating the operation services as sets of components stored in an OS array list. Path vectors will be constructed and stored in the $Path$ array list based on the dependency of (consumer/producer). Because it must respect SLA condition system performance, resources capacity Φ_{os} and communication configuration time E_{os} will be validated for each operation service and the resources capacity must be larger than requested and communication configuration time must be lower than predefined threshold value. Next, two minimization functions will be run: the first $FindMin(Delta)$ function will look up on system configuration and redundant components in order to find the exact required resources and check to consolidate over provisioned components by returning a list of components that can be consolidated; this task will achieve elastic conditions for application and cloud resources. The second function, $FindMin(A)$ will minimize communication channels by using an aggregation component or by finding alternative paths by iterating over all dependent components and looking for predecessor components with redundant successors. This ensures that only critical paths exist and that communication channels are not duplicated. We will ensure that no resources are over provisioning. The FindMin(A) function is repeatedly called until the next lower value for the communication configuration time A is achieved. The list of communication channels will then be tested in order to consolidate the components.

Algorithm 1 Optimization Algorithm

Require: $Catalog, Demands$ /* Catalog holds all system information and components location, Demands requests resources D.*/
  $OS \leftarrow FindOperationServices()$
  $Path \leftarrow Findpath(OS)$
  while $E_{os} < E_{th}$ && $\Phi_{os} > \Phi_{reg}$ **do**
    $ConsolidationList \leftarrow FindMin(\Delta)$ /* reduce redundancy with capacity Φ and performance E constraint */
    $CK \leftarrow FindMin(A)$ /* minimizing communication links return set of removed links CK */
    for `item` **in** CK **do**
      if $item.\alpha$ has alternative successor links **then**
        $remove(item.ck)$ /* test the communication links, if not required then remove them.*/
      end if
    end for
  end while
  for `item` **in** $ConsolidationList$ **do** /* check if the removed component will impact end to end delay β, and the critical path has alternative way*/
    if $item.\alpha \notin X_{mid}$ && $item.\chi \subseteq ConnectionSuccessor()$ **then**
      $remove(item)$
    end if
  end for

As Algorithm 1 shows the optimization function $FindMin()$ is a generic function that calls IBM Cplex java API library [34], which solves the optimization problem with constraint using linear integer programming. The cost functions and constraint are defined as in Eq. (3), for both redundancy Δ and communication A.

6. Implementation and performance evaluation

A java based application is developed to simulate the model using multi-threading and inter processes communication. The model is implemented as a discrete time event application that is driven by the system actions in each time unit. One Dell workstation machine is used to run the simulation with 4 cores of Intel Xeon 3.6 GHz processor, and 32G of RAM. Eclipse version "Eclipse IDE for Enterprise Java Developers. Version: 2019–06 (4.12.0) Build id: 20190614-1200 OS: Windows 10, v.10.0, x86_64/win32 Java version: 1.8.0_291" are used as an IDE with MAVEN package management, and excel sheet for results analysis and drawing. The simulation includes two types of elastic cloud management models, 1) monolithic architecture, and (2) micro-service architecture. A multiple aspects comparison between these two models are conducted with results shown in Section 7. Performance evaluation is validated and tested using java JUnit 5 [35], an API to measure code time and space complexity. The execution time of the algorithm diverges from $O(n \times \log(n))$ in the worst case scenario to $O(\log(n))$ in the best case scenario. Workload demands are synthesized using normal random variables with different mean and variance parameters.

7. Experiment results

Figs. 5 and 6 depict the communication overhead in both the monolithic and micro-service elastic management models. In Fig. 5 the communication transactions number is measured in comparison to the number of components. In the system there are two types of messages, the management messages and the orchestration messages. The management messages are used for handling all system communication required for scheduling. The orchestration messages are the actions messages in the scheduling task like control the VMM to create new VM, or Docker to create new container. Orchestration messages, in both models, (like reduce container size or create new one) are much less than management tasks messages (like collecting logs or synchronizing the operations), because cloud management system keeps track of resources and application status by communicating between modules all the time, to choose the best orchestration actions. On the other hand, orchestration messages are sent only when the best actions are made with a grace time between each change to keep the cloud stable and avoid oscillations, as was mentioned in Section 3. In monolithic model the communication number increases gradually by increasing number of resources and running component increases in a

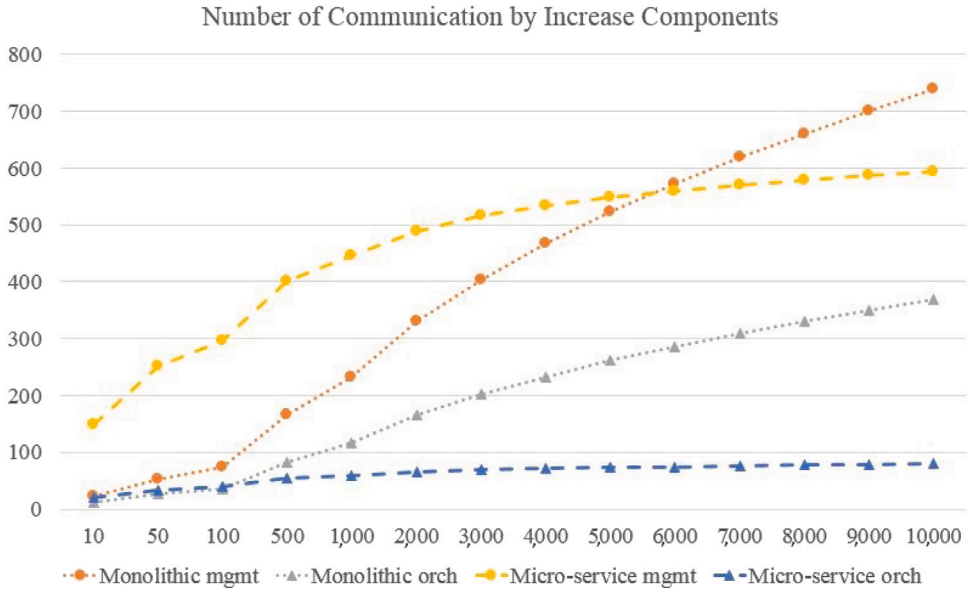

Fig. 5. Communication with increasing number of components.

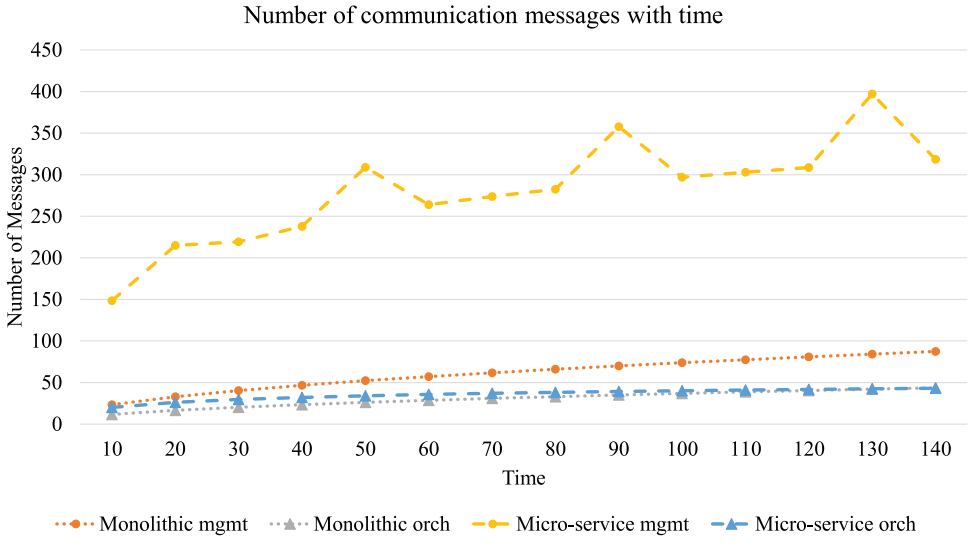

Fig. 6. Communication with time and gradually increasing number of components.

linear fashion, and that is because the communication follows (one to all) and (all to one) style. But using micro-service model the communication follows hierarchical style that aggregates the common messages in communication. By increasing number of components in micro-service model, it shows a logarithmic increase in communication number, which is much better than a linear increase ($log(n) \ll n$). Micro-service orchestration has lower number of communication messages, because the locality of management (closest management modules to resources actuators) does not need to get the task to be scheduled from central management node to make the actions, like in the monolithic architecture. Fig. 6 depicts the communication number with respect to time (unit in seconds). It is clear that some spikes in the micro-service model happen frequently with time. This is because the update messages in the publish/subscribe model (increase/decrease number of components) propagate once change occurs, and follow clusters paths as in tree spreading from leaf to root with elimination if the other node does not need the information in other modules. It is good to use monolithic model with small number of components but micro-service shows lower communication overhead with huge number of components.

Figs. 7 and 8 describe the scaling accuracy for both elastic models using two types of synthesized workload. In this comparison the same scaling and scheduling algorithm are used. Both models show a good elastic scaling with some variations, and provisioned values are close to demands. But the difference is that with micro-service the system is aware more about the orchestration options,

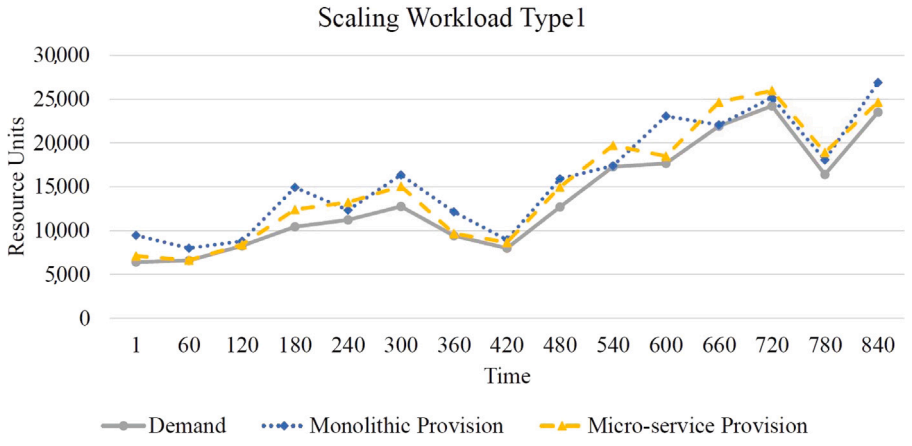

Fig. 7. Elastic scaling accuracy with workload type 1 $\mu = 2000$, $\sigma = 1500$.

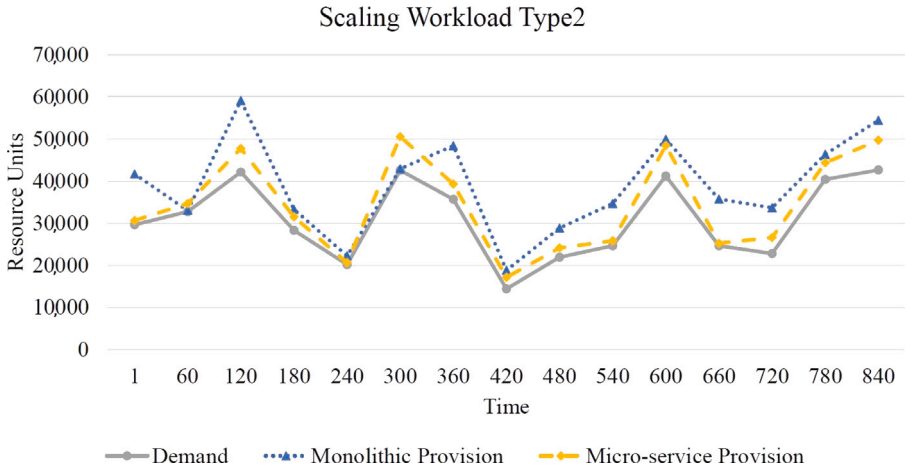

Fig. 8. Elastic scaling accuracy with workload type 2 $\mu = 4000$, $\sigma = 2500$.

Table 1
Micro-service to monolithic elastic scaling ratio.

Workload types	Monolithic accumulative provisioning	Micro-service accumulative provisioning	Ratio%
Type 1	119,187.188	52,591.619	44.125
Type 2	32,766.391	21,709.332	66.254

and micro-management orchestration nodes are closer to resources (work in localization zones) that make the actions faster to be applied. Each management operation service group can decide the orchestration actions locally if the actions can handle the demands. If not, the operation group will cooperate with others by escalating the requirement to its higher level (ancestors), which has higher data population about the system. It allows the system to offer the required resources using the closest operation service groups giving the accurate value with lower latency in time. However in monolithic it must wait until the master management node collects the information from all running nodes, and then make the decision. For both types of workload micro-service model shows a better elastic scaling than the monolithic model, due to better orchestration and agility in handling the changes.

To evaluate elastic scaling, we define a scaling factor metric SF by dividing provisioned resources Pr on workload demands WL. The best elastic accuracy of the system is by provisioning resources close to demand as represented by the scaling factor $SF = \frac{Pr}{WL}$, and as shown in Figs. 9 and 10. In these figures, the micro-service model achieved more accurate scaling, resulting in less over-provisioning resources. Another metric used to compare elastic scaling between monolithic and micro-service architecture models is to find the ratio for the accumulative provisioned resources of micro-service to monolithic, as shown in Table 1, which shows that the micro-service model saves 44% to 66% of over-provisioned resources.

End to end delay in the micro-service model is the time of longest dependence path between system components communication. Fig. 11 depicts a longer time delay in micro-service communication, because of hierarchical communication path and longer dependency chains, unlike in monolithic where a direct communication exists that has a constant time. In micro-service a protection

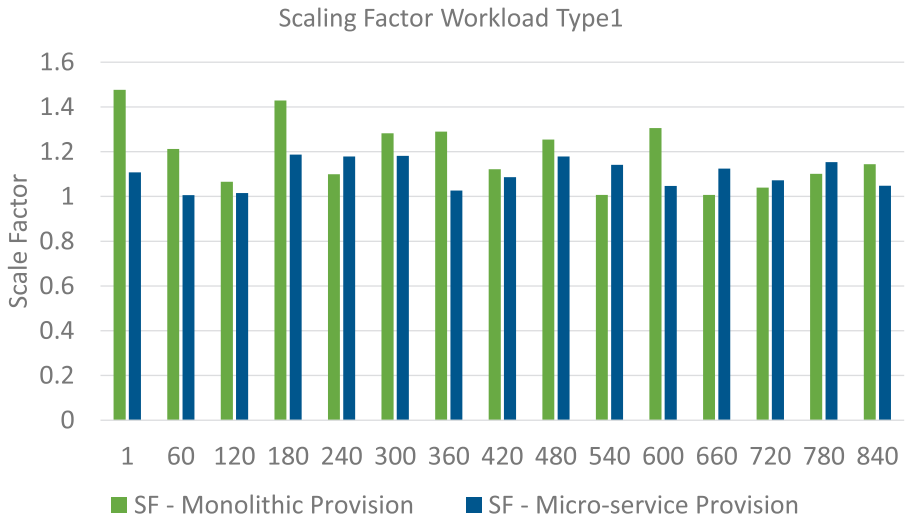

Fig. 9. Elastic scaling accuracy for workload type 1.

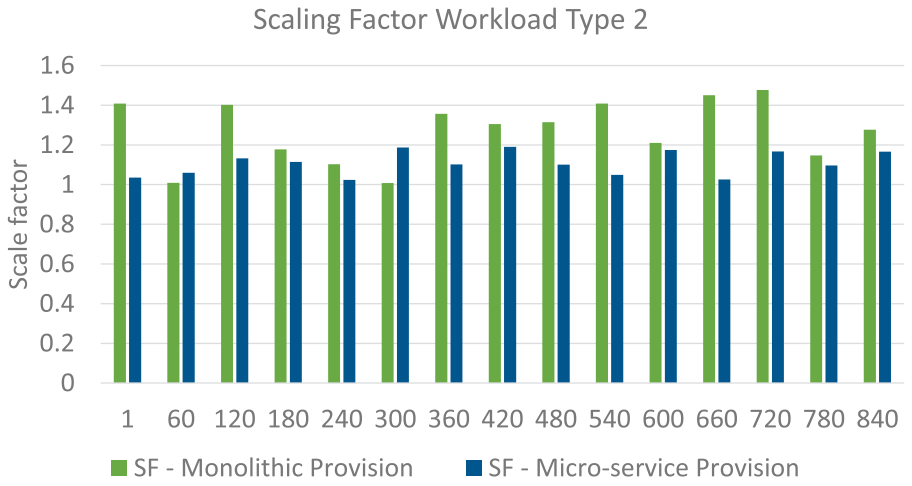

Fig. 10. Elastic scaling accuracy for workload type 2.

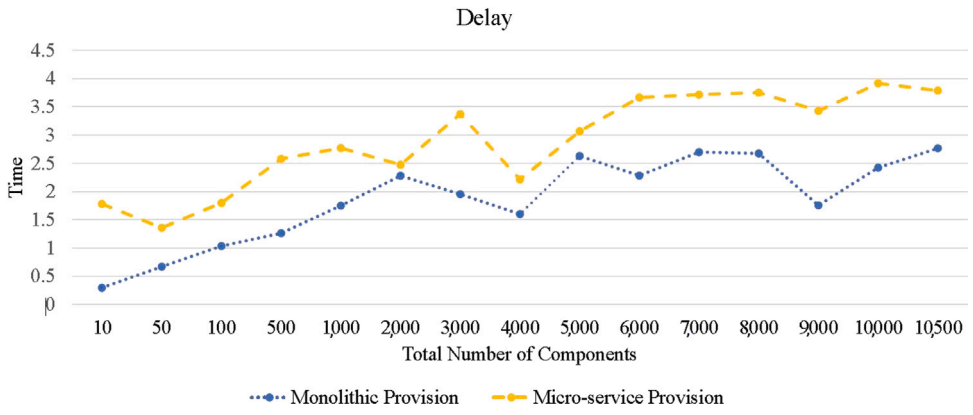

Fig. 11. End to end delay.

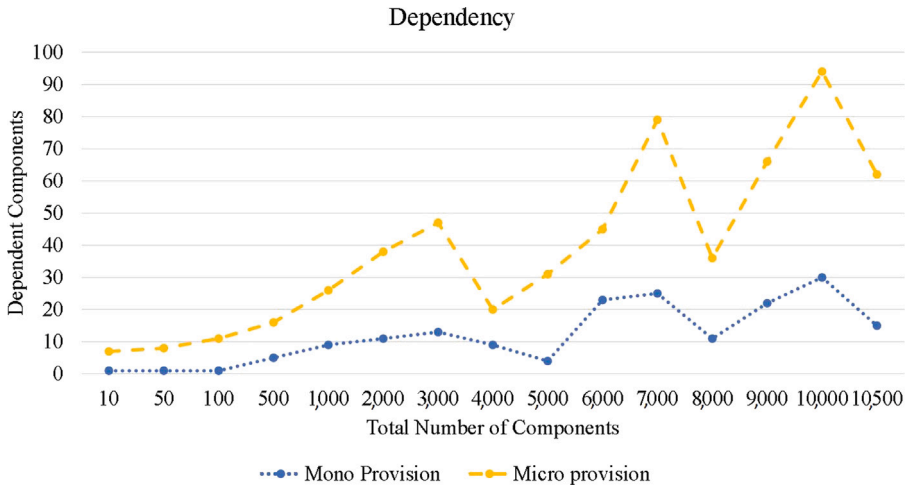

Fig. 12. Number of components in communication dependency.

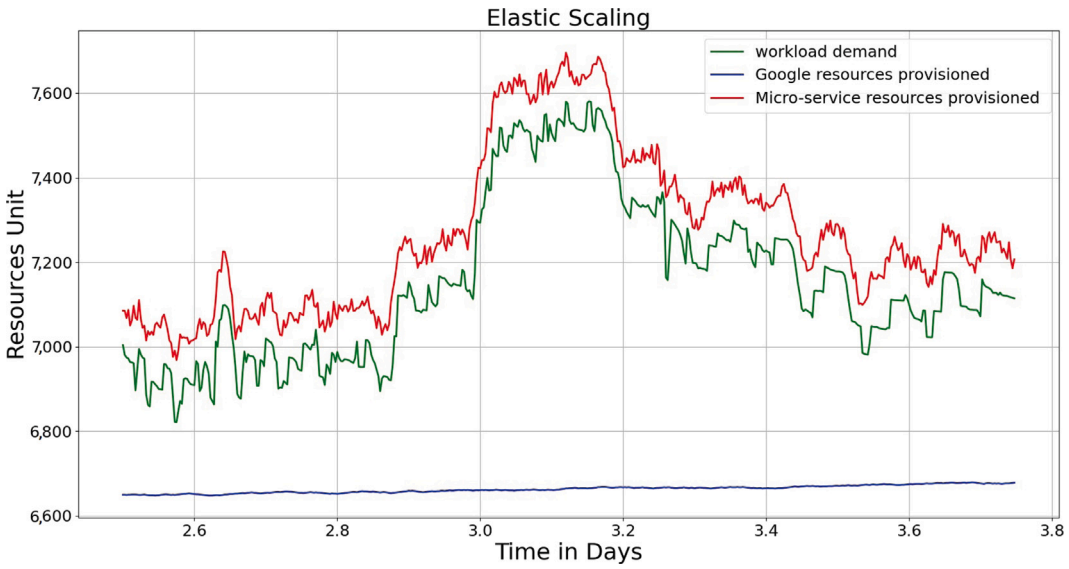

Fig. 13. Elastic resources scaling.

from long delay is applied if number of component increases to guarantee it not to exceed the configuration time in any orchestration, which is 5 seconds hard coded value in the system simulation, where the system will re-balance its paths by selecting a new aggregator node to reduce longest path communication. Also, the monolithic model has a noticeable delay time, which is close to micro-service model value. Micro-service model protects itself from higher delay by realigning dependent components and reducing dependency chain series. If the delay exceeds the maximum time a new aggregation for the dependent components takes place to reduce their number. As in Fig. 12 the value of chain dependency might be reduced even with increasing number of running components. It is the key to keep the end to end delay under the accepted ratio. In the monolithic model the delay comes from waiting of master node to communicate with all in charge nodes to make the actions, and the dependency occurs from same reason.

7.1. System comparison with Google traces

Running Google cloud datacenter traces as a real workload input is used to evaluate the proposed system. The orchestration and configuration of system resources is emulated as component scaling, which is compared to Google cloud management machine scaling. Google datacenter traces were also studied in our previous work [36], where Google cloud datacenter traces were analyzed with a focus on workload and datacenter configuration aspects. Workload is submitted as jobs, and each job contains a patchwork of tasks that make up the workload demands. Google datacenter resources are configured by adding or removing physical machines

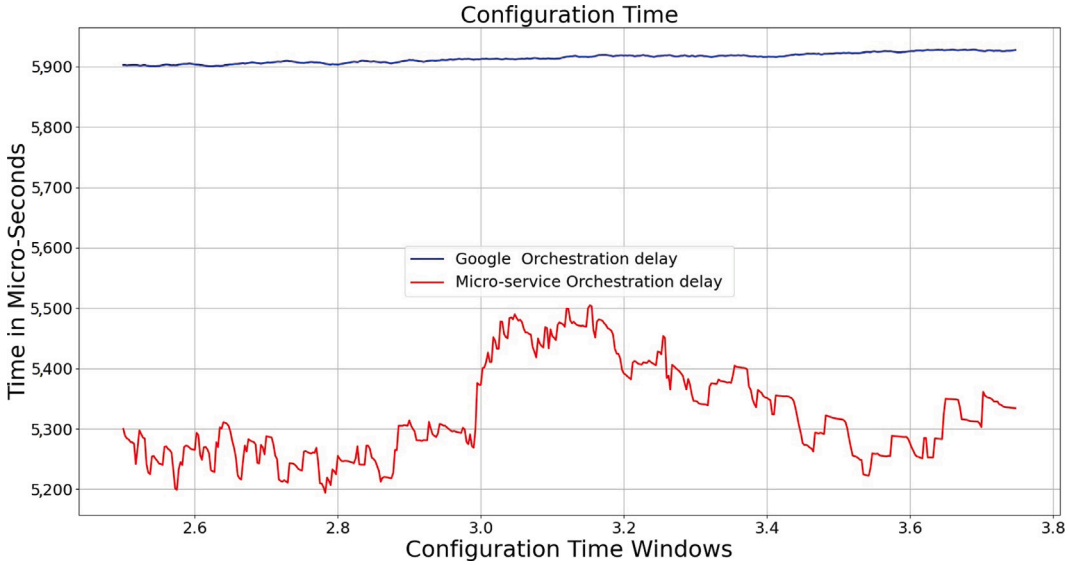

Fig. 14. Configuration time.

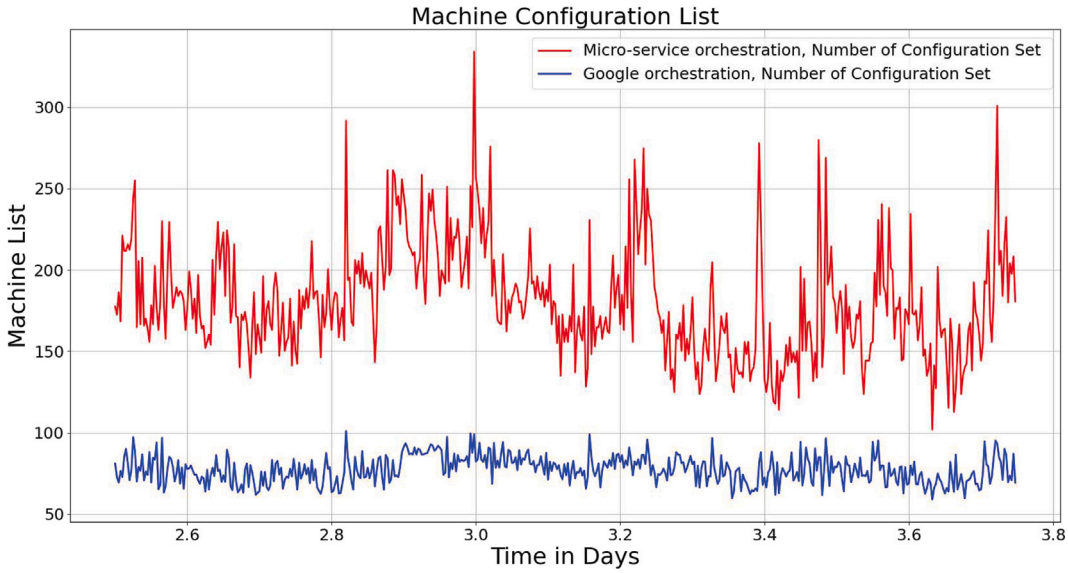

Fig. 15. Number of components/machine list in configuration time window.

to meet demand, as determined by the Google scheduling and orchestration management system. Furthermore, in [37], we mathematically modeled Google workload and applied machine learning for prediction, which is used in our system for datacenter resource provisioning and configuration. Experiments on elastic scaling, datacenter configuration set, and configuration time were carried out. A behavior-based approach is used to compare the micro-service management system and the Google management system. The time required for micro-service resource scaling, datacenter resource orchestration, and configuration in response to a workload submission window, are comparing to Google's actual action for the same workload time window.

Fig. 13 shows that the micro-service model achieved more accurate elastic scaling matching the demands while respecting service level agreement (SLA) provisioning resources that are equal to or larger than the demands. Meanwhile, Google Orchestrator violates the SLA by under-provisioning resources and gradually increasing them. Connecting this scaling behavior in Google orchestrator, it is clear that the configuration time for cloud resources is long when compared to the micro-service architecture model, as shown in Fig. 14. The reason for this is that Google has a centralized orchestrator that responds to demand by polling resources at fixed average time intervals. However, in a micro-service architecture, the system has dynamic monitoring and a hierarchical orchestration setup. This is also reflected in the Google datacenter configuration machine list, which occurs in a gradual manner. However, as illustrated in Fig. 15, micro-service architecture has more aggressive resource configuration to reflect workload demands.

8. Conclusion

A full stack micro-service elastic cloud management system is proposed in this work. The goal is to achieve a generic cloud management system with best cloud resources management and elastic provisioning. The proposed ecosystem is developed based on micro-service pattern architecture that has a dynamic decoupling and consolidation based on cloud configuration and workload characteristics demand. A Java application is developed to simulate the system behavior and monitor its performance using discrete time events. Group of metrics including communication overhead, end to end delay, application performance, scaling, and components dependency are evaluated. The proposed micro-service model shows an excellent performance when the system size is increased. Our future work is to implement the model in real cloud environment.

Data availability

No data was used for the research described in the article.

References

[1] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, P. Merle, Elasticity in cloud computing: State of the art and research challenges, IEEE Trans. Serv. Comput. 11 (2) (2018) 430–447, http://dx.doi.org/10.1109/TSC.2017.2711009.
[2] Z. Avidan, H. Otharsson, Accelerating the Digital Journey from Legacy Systems to Modern Microservices, CreateSpace Independent Publishing Platform, 2018, URL https://books.google.ca/books?id=Bb5IvAEACAAJ.
[3] P. Kang, P. Lama, Robust resource scaling of containerized microservices with probabilistic machine learning, in: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing, UCC, 2020, pp. 122–131, http://dx.doi.org/10.1109/UCC48980.2020.00031.
[4] M. Abdullah, W. Iqbal, A. Erradi, F. Bukhari, Learning predictive autoscaling policies for cloud-hosted microservices using trace-driven modeling, in: 2019 IEEE International Conference on Cloud Computing Technology and Science, CloudCom, 2019, pp. 119–126, http://dx.doi.org/10.1109/CloudCom.2019.00028.
[5] F. Rossi, V. Cardellini, F.L. Presti, Hierarchical scaling of microservices in Kubernetes, in: 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS, 2020, pp. 28–37, http://dx.doi.org/10.1109/ACSOS49614.2020.00023.
[6] T. Daradkeh, A. Agarwal, Adaptive micro-service based cloud monitoring and resource orchestration, in: 2022 13th International Conference on Information and Communication Systems, ICICS, 2022, pp. 127–132.
[7] S.K. Battula, S. Garg, J. Montgomery, B. Kang, An efficient resource monitoring service for Fog computing environments, IEEE Trans. Serv. Comput. 13 (4) (2020) 709–722, http://dx.doi.org/10.1109/TSC.2019.2962682.
[8] A. Noor, D.N. Jha, K. Mitra, P.P. Jayaraman, A. Souza, R. Ranjan, S. Dustdar, A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments, in: 2019 IEEE 12th International Conference on Cloud Computing, CLOUD, 2019, pp. 156–163, http://dx.doi.org/10.1109/CLOUD.2019.00035.
[9] N. Wang, B. Varghese, M. Matthaiou, D.S. Nikolopoulos, ENORM: A framework for edge node resource management, IEEE Trans. Serv. Comput. 13 (6) (2020) 1086–1099, http://dx.doi.org/10.1109/TSC.2017.2753775.
[10] T. Shiraishi, M. Noro, R. Kondo, Y. Takano, N. Oguchi, Real-time monitoring system for container networks in the era of microservices, in: 2020 21st Asia-Pacific Network Operations and Management Symposium, APNOMS, 2020, pp. 161–166, http://dx.doi.org/10.23919/APNOMS50412.2020.9237055.
[11] N. Marie-Magdelaine, T. Ahmed, G. Astruc-Amato, Demonstration of an observability framework for cloud native microservices, in: 2019 IFIP/IEEE Symposium on Integrated Network and Service Management, IM, 2019, pp. 722–724.
[12] A. Beltrami, P.D. Maciel, F. Tusa, C. Cesila, C. Rothenberg, R. Pasquini, F.L. Verdi, Design and implementation of an elastic monitoring architecture for cloud network slices, in: NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium, 2020, pp. 1–7, http://dx.doi.org/10.1109/NOMS47738.2020.9110415.
[13] T. Daradkeh, A. Agarwal, N. Goely, M. Zaman, Real time metering of cloud resource reading accurate data source using optimal message serialization and format, in: 2018 IEEE 11th International Conference on Cloud Computing, CLOUD, 2018, pp. 476–483, http://dx.doi.org/10.1109/CLOUD.2018.00067.
[14] T. Daradkeh, A. Agarwal, N. Goel, A. Kozlowski, Point estimator log tracker for cloud monitoring, in: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops, INFOCOM WKSHPS, 2019, pp. 62–67, http://dx.doi.org/10.1109/INFCOMW.2019.8845149.
[15] T. Daradkeh, A. Agarwal, N. Goel, J. Kozlowski, Elastic cloud logs traces, storing and replaying for deep machine learning, Procedia Comput. Sci. 171 (2020) 101–109, http://dx.doi.org/10.1016/j.procs.2020.04.011, Third International Conference on Computing and Network Communications (CoCoNet'19) URL https://www.sciencedirect.com/science/article/pii/S1877050920309741.
[16] A. Abdel Khaleq, I. Ra, Agnostic approach for microservices autoscaling in cloud applications, in: 2019 International Conference on Computational Science and Computational Intelligence, CSCI, 2019, pp. 1411–1415, http://dx.doi.org/10.1109/CSCI49370.2019.00264.
[17] A. Anagnostou, S.J.E. Taylor, N. Tijjani Abubakar, T. Kiss, J. DesLauriers, G. Gesmier, G. Terstyanszky, P. Kacsuk, J. Kovacs, Towards a deadline-based simulation experimentation framework using micro-services auto-scaling approach, in: 2019 Winter Simulation Conference, WSC, 2019, pp. 2749–2758, http://dx.doi.org/10.1109/WSC40007.2019.9004882.
[18] Cloud Resource Orchestration; on-click app deployment, MiCADOscale, 2022, URL https://micado-scale.eu/.
[19] V. Yussupov, U. Breitenbucher, C. Krieger, F. Leymann, J. Soldani, M. Wurster, Pattern-based modelling, integration, and deployment of microservice architectures, in: 2020 IEEE 24th International Enterprise Distributed Object Computing Conference, EDOC, 2020, pp. 40–50, http://dx.doi.org/10.1109/EDOC49727.2020.00015.
[20] S. Chattopadhyay, S. Chatterjee, S. Nandi, S. Chakraborty, Aloe: An elastic auto-scaled and self-stabilized orchestration framework for IoT applications, in: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, 2019, pp. 802–810, http://dx.doi.org/10.1109/INFOCOM.2019.8737656.
[21] Cloud-native SDN Controller Based on Micro-Services for Transport Networks, Zenodo, 2020, http://dx.doi.org/10.1109/NetSoft48620.2020.9165377.
[22] L. Ren, W. Wang, H. Xu, A reinforcement learning method for constraint-satisfied services composition, IEEE Trans. Serv. Comput. 13 (5) (2020) 786–800, http://dx.doi.org/10.1109/TSC.2017.2727050.
[23] L. Bao, C. Wu, X. Bu, N. Ren, M. Shen, Performance modeling and workflow scheduling of microservice-based applications in clouds, IEEE Trans. Parallel Distrib. Syst. 30 (9) (2019) 2114–2129, http://dx.doi.org/10.1109/TPDS.2019.2901467.
[24] Chameleon testbed, 2022, URL https://micado-scale.eu/.
[25] P. Zhao, P. Wang, X. Yang, J. Lin, Towards cost-efficient edge intelligent computing with elastic deployment of container-based microservices, IEEE Access 8 (2020) 102947–102957, http://dx.doi.org/10.1109/ACCESS.2020.2998767.
[26] J. Xie, D. Guo, X. Zhu, B. Ren, H. Chen, Minimal fault-tolerant coverage of controllers in IaaS datacenters, IEEE Trans. Serv. Comput. 13 (6) (2020) 1128–1141, http://dx.doi.org/10.1109/TSC.2017.2753260.

[27] Y.M. Ramirez, V. Podolskiy, M. Gerndt, Capacity-driven scaling schedules derivation for coordinated elasticity of containers and virtual machines, in: 2019 IEEE International Conference on Autonomic Computing, ICAC, 2019, pp. 177–186, http://dx.doi.org/10.1109/ICAC.2019.00029.
[28] H. Hu, Y. Wen, L. Yin, L. Qiu, D. Niyato, Coordinating workload scheduling of geo-distributed data centers and electricity generation of smart grid, IEEE Trans. Serv. Comput. 13 (6) (2020) 1007–1020, http://dx.doi.org/10.1109/TSC.2017.2773617.
[29] W. Zhang, Y. Wen, L.L. Lai, F. Liu, R. Fan, Electricity cost minimization for interruptible workload in datacenter servers, IEEE Trans. Serv. Comput. 13 (6) (2020) 1059–1071, http://dx.doi.org/10.1109/TSC.2017.2759206.
[30] A. Samanta, Y. Li, F. Esposito, Battle of microservices: Towards latency-optimal heuristic scheduling for edge computing, in: 2019 IEEE Conference on Network Softwarization, NetSoft, 2019, pp. 223–227, http://dx.doi.org/10.1109/NETSOFT.2019.8806674.
[31] Z. He, Novel container cloud elastic scaling strategy based on Kubernetes, in: 2020 IEEE 5th Information Technology and Mechatronics Engineering Conference, ITOEC, 2020, pp. 1400–1404, http://dx.doi.org/10.1109/ITOEC49072.2020.9141552.
[32] C. Richardson, Microservices Patterns: With Examples in Java, Manning, 2018, URL https://books.google.ca/books?id=UeK1swEACAAJ.
[33] F.P. Brooks, The Mythical Man-Month – Essays on Software-Engineering, Addison-Wesley, 1975.
[34] IBM, "ILOG CPLEX optimization, 2022, URL https://www.ibm.com/docs/en/icos.
[35] B. Stefan, B. Sam, L. Johannes, M. Matthias, P. Marc, d.R. Juliette, C. Stein, Junit 5 user guide, 2022, URL https://junit.org/junit5/docs/current/user-guide/.
[36] T. Daradkeh, A. Agarwal, N. Goel, J. Kozlowski, Google traces analysis for deep machine learning cloud elastic model, in: 2019 International Conference on Smart Applications, Communications and Networking, SmartNets, 2019, pp. 1–6, http://dx.doi.org/10.1109/SmartNets48225.2019.9069765.
[37] T. Daradkeh, A. Agarwal, M. Zaman, R.M. S, Analytical modeling and prediction of cloud workload, in: 2021 IEEE International Conference on Communications Workshops, ICC Workshops, 2021, pp. 1–6, http://dx.doi.org/10.1109/ICCWorkshops50388.2021.9473619.