Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

Elastic Deployment of Container Clusters across Geographically Distributed Cloud Data Centers for Web Applications

Yasser Aldwyan[1,2] | Richard O. Sinnott[1] | Glenn T. Jayaputera[1]

[1]School of Computing and Information Systems, The University of Melbourne, VIC, 3010, Australia

[2]Faculty of Computer and Information Systems, Islamic University of Madinah, Madinah, Saudi Arabia

Correspondence
Yasser Aldwyan, School of Computing and Information Systems, The University of Melbourne, Australia.
Email: yaldwyan@student.unimelb.edu.au

Summary

Containers such as Docker provide a lightweight virtualization technology. They have gained popularity in developing, deploying and managing applications in and across Cloud platforms. Container management and orchestration platforms such as Kubernetes run application containers in virtual clusters that abstract the overheads in managing the underlying infrastructures to simplify the deployment of container solutions. These platforms are well suited for modern web applications that can give rise to geographic fluctuations in use based on the location of users. Such fluctuations often require dynamic global deployment solutions. A key issue is to decide how to adapt the number and placement of clusters to maintain performance, whilst incurring minimum operating and adaptation costs. Manual decisions are naive and can give rise to: over-provisioning and hence cost issues; improper placement and performance issues, and/or unnecessary relocations resulting in adaptation issues. Elastic deployment solutions are essential to support automated and intelligent adaptation of container clusters in geographically distributed Clouds. In this paper, we propose an approach that continuously makes elastic deployment plans aimed at optimising cost and performance, even during adaptation processes, to meet service level objectives (SLOs) at lower costs. Meta-heuristics are used for cluster placement and adjustment. We conduct experiments on the Australia-wide National eResearch Collaboration Tools and Resources Research Cloud using Docker and Kubernetes. Results show that with only a 0.5ms sacrifice in SLO for the 95th percentile of response times we are able to achieve up to 44.44% improvement (reduction) in cost compared to a naive over-provisioning deployment approach.

KEYWORDS:
Placement, Containers, Dynamic deployment, Multi-cluster, Docker, Kubernetes.

1 | INTRODUCTION

Cloud-based web applications often need to be intelligently deployed to specific geographical locations to improve end user experiences, e.g., regarding performance. Containers, a lightweight virtualization technology, have gained popularity for deploying applications efficiently in such distributed environments 1, 2, 3. They provide application packaging that allows consistent, portable deployment across multiple Clouds 4 as well as abstracting away many of the overheads when deploying and managing containers and infrastructures 4, 5. These infrastructures used for container deployment models in Clouds are usually clusters of virtual machines (VMs), where each cluster of VMs has a container cluster management system, e.g., Kuberenetes 6, that is used to deploy, manage and scale containers across Cloud resources. Container management platforms, such as Google Anthos 7 and Rancher 8, take all management responsibilities to automate the deployment and simplify the management of such clusters and containers across Clouds 1. Most of these platforms support multi-cluster deployment models, giving application providers capabilities to deploy/remove clusters in Clouds to fit their needs e.g., isolation, location or application scaling 7, 9, 3. The multi-cluster

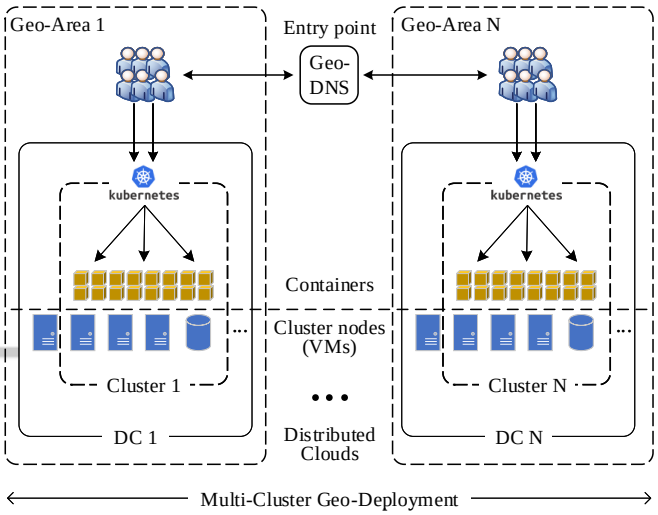


FIGURE 1 Container-based multi-cluster deployment of web applications across geographically distributed clouds

deployment models for distributed Clouds are well suited for modern web applications that can exhibit global fluctuations over time based on the user base demand. This requires intelligent dynamic global deployments of container clusters 10, including application containers, to data centers, e.g., they should be deployed in proximity to end users to maintain performance as shown in Figure 1.

However, due to the absence of automated elastic deployment across multiple distributed Clouds, adapting such deployments to handle spatial workload variations is a daunting task. A key issue is to decide when and how to adapt the deployments of clusters, in terms of their quantity and location to maintain performance and minimise operating and adaptation costs. In particular, we consider the geographical (spatial) distribution of workloads, i.e., the locations of an application can depend on the where user requests are concentrated and this should be used to determine the optimal data center(s) for hosting the application. Spatial fluctuations in user workloads can occur due to the variations in application popularity across countries and/or cities over time. For instance, tracing the web traffic of 6.5 million user check-ins to the social media platform Gowalla over a two-year period (2009 and 2010), showed obvious variations in monthly rate in user growth across many (global) locations 11. The application became popular in some areas due to local events, e.g., festivals. However the resources that deliver this platform may be deployed in distant Cloud data centers. Geographical distance typically increases network latency 11, 12, 13, which ultimately has negative impacts on businesses with poor user experience resulting in lost revenues 14 and on given service level objectives (SLOs) that have been set. Such increased latencies incur delays in the response times of user requests that are often unacceptable for zero-tolerance-to-delay web applications, e.g., e-commerce web sites. According to Forrester 15, 40% of customers leave online e-commerce sites if loading a page takes more than 3 seconds. Additionally, applications running on clusters in given areas may become less popular over time resulting in monetary costs that are incurred with no benefits to the users.

Although most container management platforms in distributed Clouds are relatively mature with many advanced features, e.g., automation and governance 2, they do not currently offer elastic deployment techniques to handle spatial workload variations. Instead, application providers typically decide (manually) to add/remove clusters across Clouds. Such manual adaptation decisions are naive and inefficient as they can lead to costly, over-provisioning issues (excessive deployment of clusters), performance issues (improper cluster placement) and/or adaptation issues e.g., performance degradation during adaptation and/or unnecessary cluster relocation. Much of the prior work on elastic container deployment problems in the Cloud does not consider spatial aspects of workload variations as they focus on local techniques within a single data center to handle fluctuations in workload volumes through localised auto-scaling of containers 16, 17, 18, 19, clusters 20, 21, 22, or both 23, 24. Other work proposes techniques to handle geo-workload variations 11, 25, however, they do not use containers and hence do not benefit from the benefits of container-based solutions, and they do not maintain application performance during adaptation.

Container-aware elastic deployment techniques for handling spatial workload fluctuations are essential to support automated deployment adaptation of container clusters in geo-distributed Cloud environments to efficiently maintain cost and performance during adaptation. They need to make intelligent decisions to add, relocate and/or remove clusters across data centers as required. Also, they need to consider latencies between data centers when making such adaptation decisions to maintain performance (response times) and SLOs during the adaptation.

In this paper we propose an elastic deployment approach for web applications using container solutions. We argue that container management platforms should support elastic deployment techniques to support web application Quality of Service (QoS) and support SLOs at lower costs. This work makes three key **contributions**. Firstly, we present an elastic deployment technique that automatically and continuously makes proper deployment plans to optimise the number and placement of clusters. The core idea is that sacrificing an acceptable level of performance can help to reduce operating cost. For cluster placement, genetic algorithms are used that consider proximity to users and cost of adaptation (i.e., number of relocated/new clusters and inter-data center latencies), while

a heuristic is introduced for adjusting cluster quantity. Secondly, we present a framework to demonstrate how container platforms can support automated elastic deployment of container clusters in geographically distributed Clouds. Thirdly, we carry out experiments using case studies based on Kubernetes on the Australia-wide and highly distributed NeCTAR Research Cloud. Results show that with only a 0.5ms sacrifice in the SLO for the 95th percentile of response times, our approach achieves 16.67% - 44.44% reduction in cost compared to static and over-provisioning deployment solutions.

The rest of the paper is organised as follows. In Section 2, we cover related work. Section 3 describes the application and the container deployment models that are adopted as well as provides the problem definition. Section 4 discusses the proposed solution. We evaluate the proposed approach in Section 5. Finally in Section 6 we provide conclusions and identify potential future research directions.

2 | RELATED WORK

Container Deployment in Distributed Clouds. Significant efforts have been made in container deployment across distributed Clouds to tackle different challenges such as automation, migration and multi-cluster management. Regarding automation, solutions like Kops 26 and Kubespray 27 automate the deployment of Kubernetes clusters across multiple Clouds. Orchestration solutions automate the deployment of containers across multi-zone clusters and across multi-region/multi-Cloud clusters, e.g., Nomad 28. Moreover, migration solutions can relocate container clusters across data centers, either via rescheduling 10 or live migration 29, 30. Container management platforms, like Rancher 8, Google Anthos 7 and OpenShift 31 make container clusters easier to deploy in distributed clouds 1. In addition to multi-cluster governance and visibility, they provide application providers the ability to easily adapt the deployment of container clusters across data centers through a unified user interface or API. However, as it is the application provider's responsibility to make the deployment and adaptation plans, and these are unlikely to be optimal. These plans require accurate workload analysis that correctly estimate workloads 32. Hence, automated elastic container deployment techniques are needed to fill in this gap to provide accurate workload estimation and adapt the deployment to changes to maintain application performance and cost requirements.

Elastic Container Deployment in Cloud Computing. The problem of elastic container deployment in Cloud computing has been studied intensively at different resource levels: container deployment 16, 17, 18, 19, cluster deployment 20, 21, 22 or both 23, 24. These approaches use horizontal methods 21, 23, 17, 22, vertical methods 18, 24 or hybrid approaches 19, 20, 16 depending on the elasticity dimensions. However these solutions lack the ability to include spatial aspects in their adaptation processes, which is essential to reduce network latency - a key performance factor for global web applications. Instead they focus on local, auto-scaling techniques, i.e., within a single data center, to support scalability, elasticity and utilisation of Cloud resources to handle variations in workloads in a cost-efficient manner for both application and Cloud providers. Geo-elastic container deployment techniques are complementary mechanisms to these local solutions, and needed to adapt application deployment in geo-distributed Cloud environments, exploiting the lightweight and portable nature of containers.

Spatial Workload Management. The problem of spatial workload management has been tackled in different computing environments, e.g., Edge and Fog computing, although they have different demands and associated scenarios.

Geographical load balancing is a common approach for managing spatial workloads. Domain Name System (DNS)-based geographical load balancing solutions, like AWS Route 53 33 and Azure traffic manager 34 can distribute load to different Cloud data centers based user geo-locations to reduce latency and other factors such as energy savings 35, 36. Centralised geographical load balancers gather all incoming requests and distribute them to an appropriate data center based on one or more factors, e.g., carbon footprint and energy costs 37. These centralised solutions add extra latency to every request and can limit the benefit of distributing application replicas. Decentralised agent-based geographical load balancing solutions avoid issues with centralised solutions since each data center running applications has an individual load balancer realised as an agent. Agents coordinate with each other in a decentralised manner to distribute load. In 38, authors propose a decentralised geographical load balancing solution suitable for Edge computing and Internet of Things (IoT) applications. They assume a multi-cluster architecture at the edge layer where each cluster consists of edge nodes and has an orchestrator used to manage workload distribution, either locally or globally across clusters. The aim is to optimise end-to-end latency of IoT applications in Edge infrastructures. Similarly, the authors in 39 present a decentralised geographical load balancing solution suitable for multi-Cloud web applications to manage short-term spatial workload variations. This approach, however, is not adequate when managing long-term spatial variations of web applications, e.g., with monthly/seasonal variations as user requests are usually distributed to predefined and static locations. New workloads can arrive from new areas that may be distant from those static locations and such distances can incur latency issues for user requests and thus affect the overall QoS.

Another approach to handle spatial workloads is to use a geographical load balancing solution to direct users to appropriate, possibly new data centers according to given factors, e.g., latency, with auto-scaling capability at each data center. SeaClouds 40 provides a platform for the seamless management of applications on multi-Cloud environments based on this approach. It uses a geographical load balancer to redirect requests to the application replica closer to the user and uses a policy, called follow-the-sun, to auto-scale resources for applications with possibility to move replicas closer to the user. In 41 a centralised geographical load balancing and adaptive resource provisioning solution is presented. The geographical load balancer in this solution acts as an entry point to the application and selects an appropriate data center for users according to regulation requirements and other factors, e.g., latency, with resources auto-scaled at each data center. Even though such an approach can handle long-term spatial workloads, it is not suitable to our needs as it only considers optimising latency for each user individually and this can lead to deployment of application replicas at excessive number of data centers close to users. This can be costly as each data center will run their own container cluster. Reducing the number of clusters would reduce the number of master nodes and thus reduce the operating costs. Therefore, a technique to adapt the deployment and placement of container clusters in distributed Clouds, according to accumulated workloads for all clusters, is required to achieve including optimising the overall latency with minimum costs.

A better approach for managing workload variations is to use deployment optimisation techniques to intelligently adapt the deployment of applications across distributed computing environments, when needed, to maintain application needs, e.g., QoS and cost. In 42, the authors present a solution to support the adaptive deployment of multi-component IoT applications to Fog infrastructure factoring in limited infrastructure capabilities, latency, and bandwidth to achieve QoS. This solution is not applicable to multi-replica web application deployment here as Cloud infrastructures provide scalable, unlimited resources and advanced data center networks 39.

In the context of distributed Cloud and web applications, solutions such as 11, 25 propose geo-elastic deployment techniques of multi-replica web applications to maintain performance to support SLOs at lower costs. These solutions, along with geographical load balancers, can manage long-term spatial workload variations as they can dynamically adapt the number and placement of web application replicas across geo-distributed data centers. Work in 11 assumes cross-data center eventual data consistency whereas the solution in 25 targets web applications requiring strong consistency between data centers. Furthermore, the solution in 25 considers the number of cross-data center application relocations as a cost of adaptation that should be minimised while 11 does not, hence this can lead to needless relocation of application. However, none of these approaches consider inter-data center latency as a cost of adaptation (i.e., latencies between data centers for the current deployment and data centers for a new deployment) when choosing new data centers for new deployment plans to help to maintain application performance. Such considerations would minimise the latency between source and destination data centers and thus reduce the geo-replication overheads needed for web applications that require to maintain the state (e.g., user sessions).

Containerisation provides a lightweight portable runtime to facilitate the elastic deployment of applications to distributed heterogeneous Cloud platforms 43, compared to heavyweight VM-based models used in 11, 25. Elastic deployment techniques using containers allow rapid adaptation of application deployment in geographically distributed clouds since they eliminate the significant latency incurred with VM-based models when provisioning new applications and new instances. Copying container images across data centers is also much faster than copying large-sized VM images. To reduce provisioning latency in VM-based models, pre-copying optimisation techniques were proposed in 11. However, they require determination of new potential future Cloud locations in advance and periodically copying VM images which limits the capabilities of deployment optimisation techniques to choose Cloud locations other than pre-selected ones when needed. Also, by using containers such deployment techniques can utilise more Cloud locations as they provide consistent, portable deployment of applications across data centers regardless of the underlying Cloud infrastructure. Container images provide an abstraction that can isolate the application environment from the underlying deployment infrastructure 4. Container-based solutions also provide a cross-Cloud overlay networking that facilitates the process of geo-replication and migration. Thus container-based solutions have many direct benefits for elastic deployment demands compared to historic Infrastructure-as-a-Service solutions.

Application Placement in Distributed Computing Environments. Placement solutions to tackle latency management have been studied in different distributed computing environments. A solution in 42 adaptively places IoT application components across Fog and Cloud infrastructure, while considering inter-fog node and fog-Cloud latency issues and other factors such as bandwidth constraints. Similarly, in integrated edge-Cloud environments, the authors in 44 propose a dynamic placement solution for IoT requests, aimed at reducing task latency times and system power consumption. Also, work in 38 presented an approach to place IoT requests across local

or remote cluster of edge nodes (or Cloud servers), considering inter-edge-node and edge-Cloud communications, queuing and processing delays, to achieve better response times.

In the distributed Cloud context, a body of work has been proposed considering proximity to users, e.g., 11, 45, 46, inter-data center latency issues, e.g., 47 or both 25, 12, 48. Work has explored place virtual desktops 45 and service-oriented solutions 48, 46, web solutions 25, 11, 12 or diverse 47 applications across data centers. Works have explored either a static 12, 47 or dynamic 25, 11, 45, 48 deployments using VMs 25, 11, 45, 48, 47, 46 or containers 12 to achieve inter-data center strong data consistency and performance 25. Other works have considered high availability and performance even after complete or partial Cloud outages 12 factoring in budget and performance issues 46 or performance issues alone, e.g., 11, 45, 48, 47.

Dynamic placement (replacement) solutions such as 25, 48 consider inter-data center latency issues to optimise performance at deployment times. None of them however consider inter-data center latency to optimise performance during the adaptation. Solutions that tackle real time deployment scenarios are thus needed. In this work, we consider eventual data consistency between data centers, inter-data center latency during deployment is not considered.

3 | PROBLEM FORMULATION

This section provides an overview of the assumptions made for container deployment and containerised web applications in distributed Cloud environments. A specification of the problem definition is also provided.

3.1 | Assumptions for Container and Application Deployment Models

As shown in Figure 1, we assume that multiple container clusters are deployed on top of Infrastructure-as-a-Service (IaaS) distributed Clouds. Each Cloud has a number of geographically distributed data centers. We assume one cluster per data center and that clusters are deployed at different geo-areas. Each cluster should ideally serve users within its given geo-area. A geo-location DNS (geo-DNS) service, e.g., Azure Traffic Manager 34 can be used to determine the traffic to appropriate clusters based on the user geo-location. This can be obtained using IP-to-Location mapping services such as IP2Location[†]. These can associate user requests to the nearest cluster.

[†]Identifying Geographical Location by IP Address https://www.ip2location.com

In this work, we assume each cluster runs a full copy of containerised web applications as this model enables an application to scale globally 3, which fits our needs. We assume each application service, e.g., web server and database, is auto-scaled to cope with dynamic local workload volumes. For the underlying web application data model, at least one full copy is assumed to be present at each data center running a cluster. While data consistency model between replicas within a data center can be supported, eventual consistency between inter-data center copies is required to improve scalability without performance loss.

A cluster infrastructure consists of a number of VMs that can be scaled elastically by provisioning/terminating VMs from a data center through the associated Cloud APIs. Each VM is assumed to have a container runtime, e.g., a Docker engine 49 installed that allows it join the cluster as a worker node to run containers. On top of the cluster, a container orchestration and cluster management platform is assumed, e.g Kubernetes. Kubernetes has management components such as an API server and scheduler that provide a control plane for the cluster. The control plane runs on one or more master nodes for availability.

We also assume a multi-cluster container management platform that runs on an individual VM and logically runs on top of the running clusters. This should include a set of management services, e.g., for migration and cross-cluster workload monitoring, required to add new clusters or relocate/migrate existing ones.

3.2 | Problem Definition

As stated, a geo-elastic container deployment technique for multi-cluster deployment of web applications in distributed Clouds is essential to maintain application QoS and SLOs at lower costs. Any solution needs to be able to modify the current state of the cluster deployment to a new, desired state, by adding, relocating and/or removing container clusters, whenever a geo-workload fluctuation leads to unacceptable SLO violations and/or one or more idle or redundant clusters gives rise to unnecessary costs. The key challenges for deployment modifications are how to decide how many clusters are required; where they should be placed, which existing clusters should be removed/replaced and the underlying capabilities needed for cross-data center migration and replication to reach a desired deployment state. This work focuses on cluster quantity adjustment and associated, dynamic cluster replacement strategies.

To adjust the number of clusters, a mechanism is needed to minimise the number of running clusters, where possible, to reduce the associated operating costs. For the cluster replacement problem, it can be formulated as an optimisation problem

where an objective function is used to reduce the violation rates and decrease costs. Maximising the function should help to achieve such aims. Using symbols in Table 1, the objective function can be represented as follows.

$$\begin{aligned} &\text{maximise} \quad f\,(current, cand, W_t) = \frac{R\,(current,\ cand, W_t)}{adapt\_cost\,(current,\ cand, W_t)} \\ &\text{subject to} \quad \mathbf{F} \subset \mathbf{A},\ |\mathbf{F}| = N \end{aligned} \tag{1}$$

where $R\,(current, cand, W_t)$ is the estimated amount of reduction in the violation rate when changing the deployment (*current*) to the candidate one (*cand*) and where $adapt\_cost\,(current, cand, W_t)$ is the estimated cost of that adaptation for a given workload (W_t) at time (t).

Formally, the cluster replacement problem can be defined as follows. Using the terms defined in Table 1 at a given a point in time (t), for N required clusters and workload W_t for a system with *current* clusters, we want to select a set of Cloud data centers, $\mathbf{F} = \{data\_center_1, data\_center_2, \ldots, data\_center_n\}$, where $|\mathbf{F}| = N$, to (re)place container clusters across data centers in $\mathbf{F}$ such that the objective function in Eq. 1 is maximised.

4 | PROPOSED GEO-ELASTIC DEPLOYMENT APPROACH

To handle the above-mentioned issues, we present a geo-elastic deployment approach. In this approach, we propose two mechanisms that work together as one technique to make appropriate elastic scaling decisions, when needed. The first mechanism is a geo-elastic deployment controller that determines how many clusters are required. It implements an SLO-based heuristic that attempts to establish an optimal size adjustment of the current deployment (*n_clusters*) among different possible adjustments to avoid/minimise SLO violations under cost and performance constraints. Within each possible adjustment, the controller uses the second mechanism, a cluster replacement method, to handle spatial aspects of elasticity, i.e., finding the best location for cluster deployments across geographically distributed Cloud data centers (*cand*). A candidate deployment plan with minimal cost of adaptation and acceptable SLO violation rate is selected as the new deployment plan. Moreover, we present a framework to support the automated elastic deployment of container clusters in geographically distributed Clouds. This factors in the geographical distance between users and data centers running application containers and between data centers themselves as key factors that affect network latency (performance). We refer to this as geo-elastic deployment.

TABLE 1 Terms Used in the Models

Term	Meaning
current	Current deployment of container clusters
n_clusters	Size of the current deployment
cand	New deployment candidate
$\mathbf{A}$	Set of available data centers
$\mathbf{D}$	Set of the data centers of the current deployment (*current*)
$\mathbf{F}$	Set of selected data centers of the candidate deployment (*cand*)
$\mathbf{K}$	Set of new and/or relocated clusters
W_t	Workloads collected at time t
N	Number of required clusters for the new deployment
S_j^t	Number of sessions in data center j at time t
V_j^t	Number of violated sessions in data center j at time t
nl_{ij}	Estimated round-trip-time (RTT) network latency between user i and data center j if that user is served by an application running a cluster in j
il_{jk}	Estimated inter-data center (RTT) network latency between data center j and data center k (migration)
U_{thr}	Upper bound of acceptable Session Based Violates Rates ($SBVR$)
G_{thr}	Gain threshold to be considered an improvement in $SBVR$ when $SBVR$ is below U_{thr}
τ	Consecutive periods for cool down
T	Time interval
P	Pause interval

4.1 | Requirements and Assumptions

The approach requires pre-agreed SLOs for user requests to be provided by application providers. Periodic collection of workloads (i.e., user requests) from running container clusters is also obtained. Each cluster is assumed to have a load balancer that distributes incoming user requests to appropriate application containers. Such requests are stored in log files. The request logs are assumed to have information about users including their IP addresses that can be mapped to geo-locations. The geo-locations of data centers are assumed to be known and available.

Moreover, the approach depends on knowing the network latency between users and data centers and between data centers themselves. Round-trip-time network latency data can be obtained using third party services, e.g., Ookla 50, or by latency estimators to estimate latencies between users and data centers, e.g., 51, or through empirical measurement. Additionally we assume that the processing time of requests is constant and that clusters have sufficient Cloud resources and negligible internal communication overheads within data centers due to the high-speed networking capabilities. Finally, we assume that homogeneous VMs (in terms of size and price) exist for clusters running at different Clouds.

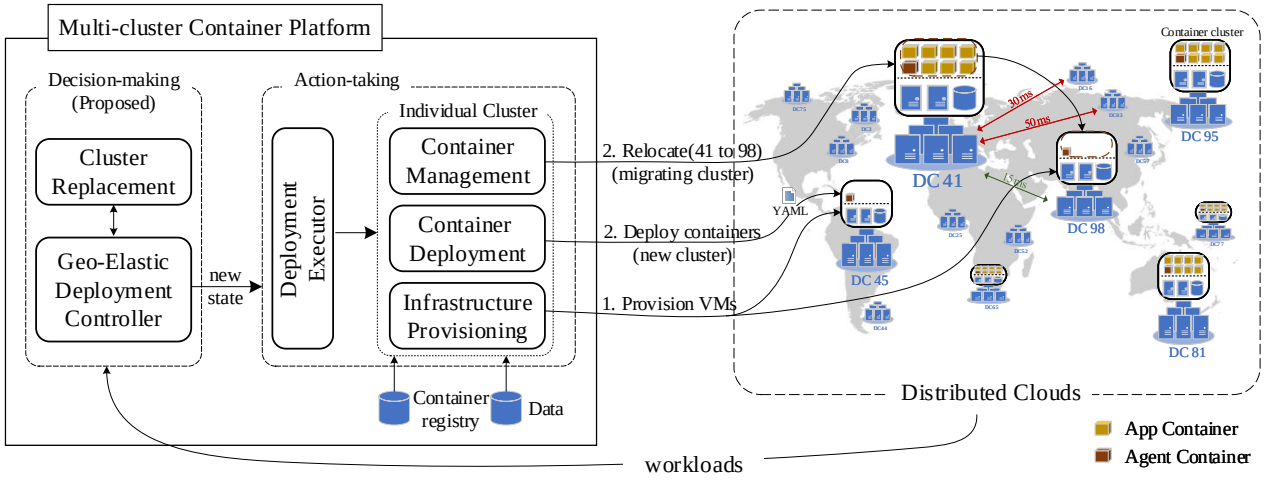


FIGURE 2 A framework of enabling automated elastic deployment of container clusters in geographically distributed Clouds

4.2 | Adaptation Triggers

In this section, we investigate the dynamic characteristics of web application geo-workloads and identify how to trigger the adaptation process to make appropriate elastic actions. We identify three cases.

Case 1: Geographical growth of workloads. Applications may gain more popularity in particular areas. In this case, geo-workloads on current clusters running application containers can violate SLOs because of the potentially large geographical distances between users and the data centers clusters and the associated network latencies that can arise. This situation should trigger the geo-elastic deployment approach to *geo-expand* the current deployment.

Case 2: Geographical shrinkage of workloads. This case is the opposite of the previous one. Applications at some point in time may lose popularity in some regions causing clusters running in data centers within those regions to become under-utilised. This can cause unnecessary expenses to be incurred due to the over-provisioned deployment of clusters. This condition should be a trigger to *geo-shrink* the current deployment.

Case 3: Geographic shift in workloads. In this case, the popularity of applications can shift between regions, hence clusters already running in data centers may need to be partially redeployed to other ones. This case should be used as a trigger to *relocate* some of the running clusters at new data centers to meet the geo-area needs of users at that time.

4.3 | Geo-elastic Deployment Framework

As shown in Figure 2, the framework's components are divided into two main categories: decision-making and action-taking. This work mainly focuses on the decision-making components. Components in such multi-cluster platforms communicate with container cluster platforms running in data centers through agents. An agent, which can be deployed as a containerised service, receives commands from components and executes them on the local cluster platform. Agents can also communicate with other agents running in other cluster platforms in different data centers, when needed, to provide inter-cluster management services such as container relocation and data replication between clusters.

The functionality of the framework's components and how they interact are discussed below.

Decision-making Components

Decision-making components such as a geo-elastic deployment controller and cluster replacement component are responsible for making elastic decisions in terms of the quantity and placement of container clusters as required, e.g., to periodically assess and produce new, desired states of the deployment to meet evolving web application requirements. Once a new state of deployment is determined, it is passed as a deployment plan to the deployment executor component.

Geo-Elastic Deployment Controller. The geo-elastic deployment controller component is responsible for deciding on the optimal quantity of container clusters based on current geo-workloads and pre-agreed SLOs. It needs to strike a balance between acceptable SLO violation rates and the least possible number of container clusters distributed geographically across different Cloud data centers. An SLO-based violation model to estimate SLO violation rates of a given deployment is discussed in section 4.4.

This component implements a decision-making algorithm for the geo-elastic deployment controller mechanism, which

is discussed in more detail in section 4.5. The geo-scaling decisions include: geo-expanding, geo-shrinking and geo-relocation. Within the algorithm, the cluster replacement component is called to determine the optimal placement of clusters for any new, deployment plan produced as part of a given scaling decision.

Cluster Replacement. The cluster replacement component aims to automatically handle the spatial aspect of the adaptation process by finding the optimal placement of container clusters for any potential deployment plan. It implements a meta-heuristic using genetic algorithms for dynamic cluster placement as discussed in Sections 4.6 and 4.7.

Action-taking Components

Action-taking components consist of components that take appropriate actions based on any new deployment plan obtained from the geo-elastic deployment controller. Each container cluster in a given deployment plan has a set of possible conditions for a given cluster: *new*, *migrating* and *leave-as-is*. A container cluster with a *new* condition requires creation of a new cluster while a cluster with a *migrating* condition indicates that the cluster already exists however it needs to be relocated to another data center. A cluster with a *leave-as-is* condition implies the cluster is already running at a data center and should remain there.

Deployment Executor. This component is responsible for taking elastic actions to change the current state of the deployment to a new, desired state. Specifically, it determines the condition of container clusters involved in any proposed deployment plan, and subsequently makes appropriate actions for each container cluster based on its current condition and the intended future state.

Only clusters with *new* and/or *migrating* conditions require actions to be taken. Both conditions initially require container clusters to be prepared for a given selected data center. Following this, clusters with a *new* condition, require creation of application containers to be deployed, while for migrating clusters, containers need to be relocated to the remote (selected) data centers. The required implementations of those actions should be abstracted in the individual cluster management components as discussed below.

Individual Cluster Management Components. Individual cluster management has three components: cluster infrastructure provisioning; container deployment and container management. Each component is responsible for providing different elastic actions.

Cluster Infrastructure Provisioning. This component automates the process of providing the infrastructure-related actions that are responsible for making a container platform ready at a new, selected data center. It implements all infrastructure automation capabilities required for this action including provisioning VMs through Cloud APIs to create a new cluster as well as installing required container-related software (e.g., Docker and Kubernetes) on the cluster nodes. This minimises risks related to human errors and expedites the deployment process. This automation should abstract the different implementations required to make the proposed geo-elastic deployment feature suitable for multi-Cloud environments. Multi-Cloud libraries, e.g., Apache Libcloud[‡] and jclouds[§] are examples of technologies that can be used to manage Cloud resources from different Cloud providers using a unified API. For Kubernetes, automation tools like Kops 26 or Kubespray 27 can be used here.

Container Deployment. This component supports the deployment of new application containers at container clusters with a *new* condition set. It abstracts the implementation details needed to perform the container deployment including dealing with the related data. To achieve this, this component sends a deployment request (e.g., in the form of a YAML or JSON configuration file) to a given agent. The agent then geo-replicates the application data from the nearest data center that is already running a cluster or from the data storage located in a data center running the multi-cluster container management platform. Following this, the agent passes the deployment file to the local container cluster platform (e.g., Kubernetes) via the cluster APIs. Then the local container platform pulls the required container images from a nominated container image registry.

Container Management. This component provides a migration action for existing clusters to relocate containers and their data from a source cluster to a destination cluster at a new, selected data center. It implements all techniques and services required to migrate running clusters for clusters with the *migrating* condition set. It needs to provide agents in the source and destination clusters with the required inter-cluster management services to complete the migration process. Solutions for such services have been proposed in several other works, e.g., 30, 52, 53.

4.4 | SLO-based Violation Model

In this section, we introduce a SLO-based violation model to estimate the violation rate of an application deployment at a given time, t, based on pre-agreed SLOs. This is used as a metric, **Session Based Violation Rate (SBVR)**, to evaluate the

[‡] Apache Libcloud https://libcloud.apache.org
[§] Apache jclouds https://jclouds.apache.org

performance of a given deployment. Using terms defined in Table 1, the violation model is defined as follows:

$$f_{SBVR}(\mathbf{X}, W_t) = \frac{\sum V_j^t}{\sum S_j^t} \qquad \forall j \in \mathbf{X} \tag{2}$$

At a given point in time, t, and given a deployment (i.e., the container clusters to be deployed at data centers in $\mathbf{X}$) as well as the current user geo-workloads, W_t, collected from running clusters at time t, then the SBVR of the deployment can be calculated as the total number of violated user sessions, $\sum V_j^t$, divided by the total number of user sessions, $\sum S_j^t$, where each j in $\mathbf{X}$ represents a data center running or potentially running a cluster. In this work, a user session consists of a set of successive requests. These are considered to be violated if the average response time of the requests is beyond the defined SLOs. Since we assume the processing times of requests are constant, an SLO refers to the acceptable network latency, plus the (processing) constant.

4.5 | Multi-cluster Geo-elastic Deployment Controller Algorithm

In this section, we present a decision-making algorithm (Algorithm 1) for automatically controlling the size of multi-cluster deployment according to the geo-dynamics of workloads. The algorithm aims to provide a balance between performance and cost. When adjusting the size of the deployment, it relies on a cluster replacement method, which will be discussed in the following sections, to modify the actual location of the clusters.

As discussed, there are three types of elastic decisions: geo-shrinking, geo-relocation and geo-expanding, and each one is a form of adaptation trigger. Selecting the right decision requires detecting changes in workloads. Using the terms defined in Table 1, the algorithm uses the SBVR of a deployment as a performance indicator to detect changes in workloads and as the basis for making appropriate decisions. Once the current workloads are obtained, the SLO-based violation model presented in the previous section is used to calculate the violation rates, SBVR, for both current and any candidate deployments. If the SBVR of the current deployment is beyond a pre-defined upper bound of acceptable SBVR, U_{thr}, then geo-relocation or geo-expanding decisions should be made to reduce the network latency and thus reduced the SBVR (below U_{thr}). On the other hand, if the SBVR of the current deployment has remained under U_{thr}, there are two possible courses of action. The first is to try geo-shrinking the deployment to reduce the cost. The second one, which should be used if the resultant candidate deployment is not able to maintain an acceptable violation rate, is to consider geo-relocation decisions and accept them if the new, candidate deployment is more likely to improve the SBVR beyond a predefined gain threshold, G_{thr}.

In more detail, the input of the algorithm is the initial (*current*) deployment of the container clusters. The genetic algorithm-based cluster replacement algorithm in Section 4.7 is used throughout as part of the *replace_clusters* function. The *redeploy_clusters* function should pass the accepted, candidate deployment to the deployment executor introduced previously. It should be noted that the intervals between consecutive decisions should be determined carefully to avoid unnecessarily loading the system with many status request updates. To address this we introduce a pre-configured parameter, pause time (P), to enforce these intervals.

After the initialisation steps, the algorithm runs a control loop. At every time interval, T, if a decision was made in the last iteration, $T-1$, then we pause the algorithm for P (Lines 5-8) to ensure that the system is not perpetually asking for update information. Then, the current user workloads, W_t, are collected from the container clusters comprising the current deployment (*current*). It will try to geo-shrink the deployment if the SBVR of the current deployment has remained under a given threshold (U_{thr}) for a consecutive number of periods (τ as shown in Lines 10-18). It then gets the potential, candidate deployments, *cand*, by decreasing the number of clusters by one and then replacing the clusters, when needed, through the cluster replacement method. If the violation rate of the new, candidate deployment is below the threshold, U_{thr}, then the algorithm will call the *redeploy_clusters* function to pass the new deployment plan to the deployment executor to take the appropriate actions and update the current deployment and terminate the execution of the current iteration of the loop whilst waiting for a new interval, $T+1$.

Following this the algorithm continues to explore geo-relocation decisions (Lines 19-33). When a geo-shrinking decision is not made or the SBVR of the current deployment is beyond the threshold, U_{thr}. It obtains the candidate deployment by replacing clusters only. This decision can be made when one of the two following conditions is satisfied. Firstly when the candidate deployment can help to reduce the unacceptable SBVR of the current deployment to be under the threshold, U_{thr} (Lines 21-24). Secondly when the SBVR of the current deployment is acceptable and the candidate deployment can improve the SBVR for a value that is greater than the gain threshold, U_{thr} (Lines 25-28). Lastly, a geo-expanding decision is made when the SBVRs of the current deployment as well as the candidate one, produced in the previous step,remain greater than the threshold, U_{thr} (Lines 34-40).

Algorithm 1: Decision-making Algorithm for the Multi-cluster Geo-elastic Deployment Controller

Input : *init_deploy*

current = *init_deploy*;
n_clusters = *size(init_deploy)*;
paused = *False*;
for *every T* **do**
  if *paused is True* **then**
    pause (*P*);
    paused = *False*;
  end
  /* Get clusters workloads at time *t* */
  $W_t = get\_workloads(current)$;
  /* Geo-shrinking decision */
  if $f_{SBVR}(current, W_t) < U_{thr}$ *for* τ **then**
    cand = *replace_clusters(n_clusters-1, current,* W_t*)*;
    if $f_{SBVR}(cand, W_t) < U_{thr}$ **then**
      /* To deployment executor */
      redeploy_clusters(cand);
      current = *cand*; *paused* = *True*;
      n_clusters − −;
      continue;
    end
  end
  /* Geo-relocation decision/same size */
  cand = *replace_clusters(n_clusters, current,* W_t*)*;
  relocate = *False*;
  if $f_{SBVR}(current, W_t) \geq U_{thr}$ *and*
  $f_{SBVR}(cand, W_t) < U_{thr}$ **then**
    relocate = True;
  end
  if $f_{SBVR}(current, W_t) < U_{thr}$ *and*
  $(f_{SBVR}(cand, W_t) - f_{SBVR}(current, W_t)) \geq G_{thr}$ **then**
    relocate = True;
  end
  if *relocate is True* **then**
    redeploy_clusters(cand);
    current = *cand*; *paused* = *True*;
    continue;
  end
  /* Geo-expanding decision */
  if $f_{SBVR}(cand, W_t) > U_{thr}$ *and*
  $f_{SBVR}(current, W_t) > U_{thr}$ **then**
    cand = *replace_clusters(n_clusters+1, current,* W_t*)*;
    redeploy_clusters(cand);
    current = *cand*; *paused* = *True*;
    n_clusters + +;
  end
end

4.6 | Cluster Replacement Method for Spatial Adaptation

The cluster replacement method, which is the second proposed mechanism of our geo-elastic deployment solution handles the spatial aspect of adaptation to improve the performance, geo-scalability and cost-effectiveness. This is an optimisation problem as discussed in section 3. It requires the following challenges to be addressed. One challenge is to determine how to estimate the improvement in performance as well as the cost of adaptation of a candidate deployment plan to be used by the objective function. Another challenge is to design algorithms to rapidly establish near-optimal solutions (i.e., finding potential data centers for candidate deployments) to support dynamic and near-real time scaling decisions.

Cost of Adaptation

The cost of adaptation refers to the potential cost of changing the current deployment to a new one. To improve web application demands for cost-effectiveness and performance, this cost needs to aim at minimising the operational cost as well as the adaptation time (i.e., time to complete the adaptation process). While the former can be reduced by minimising the number of clusters that are deployed and/or relocated to new cloud data centers, the latter can be reduced by minimising the total inter-data center network latencies. These latencies can occur between a data center running a multi-cluster platform or data centers of a current deployment, **D**, and selected data centers put forward for a candidate deployment, **F**.

Using the terms in Table 1, the cost of an adaptation function that can be used as the denominator of the objective function in Eq 1 is defined as:

$$adapt\_cost = 1 + |\mathbf{K}| + (1 - \frac{1}{1 + \sum_k \min_{DC_j \in \mathbf{D}} il_{jk}}) \quad \forall k \in \mathbf{K}, \mathbf{K} \subset \mathbf{F} \tag{3}$$

where $|\mathbf{K}|$ is the number of new and/or relocated clusters, DC_j is a data center j and $1 - \frac{1}{1+\sum_k \min_{DC_j \in \mathbf{D}} il_{jk}}$ is the weight of the total inter-data center latencies needed to make clusters in **K** ready. It should be noted that one is added to avoid division by zero in our objective function when there is no cost of adaptation. Since the aim is to minimise the cost of adaptation and maximise the reduction in the violation rate, a candidate deployment that helps to realise this aim should be selected by the cluster replacement algorithms. These algorithms use the objective function to propose new data centers in **F** to be geographically near the currently running ones (to reduce the overheads of inter-data center latencies during adaptation to speed up the adaptation process) as well as near new

geo-workloads (to reduce user-to-data center latency after new deployment takes place).

Inter-data center latency consideration during adaptation help to reduce the geo-replication overheads and thus its benefits can be realised in two ways. One obvious benefit is that it speeds up the adaptation process since it reduces the network latencies by expediting the relocation of containers and/or data. Another one is that it helps to maintain performance (e.g., response times) during the adaptation by lowering the overheads involved in geo-replicating the state (e.g., current user sessions) until the adaptation finishes and DNS records are updated. For example, the green and red lines represent the amount of inter-data center latencies between some data centers in Fig 2, where it is clear that adding 15ms to the response time as an overhead is better than adding 30-50ms.

Violation Rate Improvement

One way to improve the performance is to let the cluster replacement method select a candidate deployment that can produce the maximum reduction in SBVR. Selecting a deployment in this way can be costly due to the high possibility of over-provisioning. In other words, reaching the maximum reduction amount in SBVR can result in provisioning more clusters than required.

Another way is to find a balance between performance and cost when adapting deployments. In some situations, the operational cost can be reduced by minimising the number of running clusters albeit with an acceptable sacrifice in performance. This may cause an increase in the SBVR but this increase may not go beyond the upper bound of acceptable SBVR, U_{thr}. However it may also give rise to increased network latency with no increase in the SBVR since the response times of requests cause an increased delay, yet still be under the threshold of the defined SLOs. To achieve this, we propose two cases for the reduction function, R, that is used in the objective function in Eq 1 and represented as follows.

$$R = \begin{cases} U_{thr} - f_{SBVR}(cand, W_t) \\ \quad if\ f_{SBVR}(current, W_t) \geq U_{thr}\ or\ |cand| < |current| \\ \\ f_{SBVR}(current, W_t) - f_{SBVR}(cand, W_t) \\ \quad otherwise \end{cases} \tag{4}$$

Choosing which case to be used in the reduction function depends on the SBVR of the current deployment or the suggested elastic decision made by the controller. The first case is chosen if at least one of the following conditions are satisfied. The first condition is met when the SBVR of the current deployment, *current*, is above the acceptable SBVR, U_{thr}. In this condition, the first case requires a candidate deployment that reduces the SBVR below the U_{thr} threshold and maximises the reduction amount in SBVR from the U_{thr} threshold. The second condition of the first case is satisfied when the suggested elastic decision is to geo-shrink the deployment, $|cand| < |current|$. In this condition, any candidate deployment with smaller size and the maximum reduction amount in SBVR from the U_{thr} threshold will be chosen. In this second condition, the cost is reduced by possibly increasing the SBVR, however it should still be under the upper bound for acceptable SBVR, U_{thr}. If none of the above-mentioned conditions are met, the second case of the reduction function will be selected. This case aims to choose a candidate deployment that can maximise the reduction amount in SBVR from its current status.

It is noted that considering both cases is necessary in Eq 4 since ignoring one of them can lead to improper elastic decisions being made under certain conditions. Consider the following example. Using Eq. 1 and 3 and assuming the inter-data center network latency is constant at 1, the objective function can be presented as $f = \frac{R}{1+|\mathbf{K}|}$. Now assume at some time the violation rate SBVR of the current deployment is 30% and the upper bound of acceptable SBVR U_{thr} is 10%. Also assume that there are two candidate deployments, *cand1* with SBVR: 15% and K: 1 and *cand2* with SBVR: 8% and K: 2. If the second case is only considered, then f for *cand1* and *cand2* will be 7.5 and 7.3 respectively. Since the aim is to maximise the function, *cand1* will be returned by the cluster replacement algorithm (Algorithm 1). That is, in Algorithm 1, the decision is geo-expanding as the if-condition in Line 34 is met.

On the other hand, if we consider the two cases for R, then f for *cand1* and *cand2* will be -2.5 and 0.6 respectively. Therefore, the right candidate deployment *cand2* will be selected and returned. In this case, in Algorithm 1 the if-condition in Line 21 is met and the decision is geo-relocate. Hence, considering both cases for R improves the reduction in SBVR to be under U_{thr} without the need to increase the number of clusters while the one-case-only approach fails to choose the right candidate deployment and hence increases the cost as it requires an increased number of clusters.

Hardness of The Problem

Since the cluster replacement problem considers the cost of network latencies between users and data centers as well as the cost of adaptation, it falls into the class of Mobile Facility Location Problems 54 that are hard to solve 55, 56. These problems are a form of problem of moving each facility from one location to another and assigning each client to some facility such that the total costs of moving facilities and client assignments are minimised. They also generalise the NP-hard k-median problem 54, which given a set of points, involves

identifying k centers such that the total distances of the points to their closest centers are minimised. Our problem can be shown to be NP-hard by restriction, which is a method of showing that an already-known NP-hard problem is a special case of the target problem. We prove the hardness of our problem by showing that the NP-hard k-median problem is a special case of this problem.

Proof. Suppose at some point in time, t, a given reduction in a potentially remote workload occurs (i.e. Case 2 in Section 4.2) and the predefined, constant U_{thr} is set to 1. This causes a geo-shrinking decision of the current deployment, $current$, to be triggered and hence the number of required clusters, N, for the new candidate deployment, $cand$, becomes $N = |current| - 1$. If we let il_{jk} and $\mathbf{K}$ be zero in Equation 3 since no new/existing cluster can be deployed or relocated. In this case, the cost of adaptation is $adapt\_cost = 1$, which is constant. Since $|cand| < |current|$ is satisfied in Eq 4 and the U_{thr} is constant, then $R = -f_{SBVR}(cand, W_t)$. By applying the last two steps to $adapt\_cost$ and R in Eq 1 and transforming the maximisation problem to a minimisation one, the problem can be represented as minimise $f_{SBVR}(cand, W_t)$. Given a fixed amount of estimated SLO violations between any user, i, and data center, j, and given a set of available data centers, $\mathbf{A}$ where N data centers are to be chosen from $\mathbf{A}$ for deploying clusters for users in, W_t. Minimising the total amount of SLO violations is the k-median problem. Hence the cluster replacement problem $\in$ NP-hard.

As our problem is NP-hard, we need algorithms that can approximate global optimisation by finding good solutions in polynomial time.

Cluster Replacement Algorithm

To address this optimisation problem, we present an approach based on genetic algorithms. This meta-heuristic suits our problem for several reasons. First, it is used for approximating global optimisation for many problems as it generally finds good global solutions and has the power to evade local optima 57. It has also been used to solve a variety of related problems in diverse contexts as it is able to make a good trade-off between the quality of solutions and the completion time since it solves problems efficiently 12, 25, 48, 58, 59. Theoretically, by considering the complexity of the algorithm and assuming all inputs are of the same size n, then the worst-case complexity is $O(n^4)$, which is polynomial. Generally, in practice the number of required clusters, N, is relatively small for cost and administrative reasons and thus we can say the upper bound of the running time of the algorithm is $O(n^3)$. Empirically, this approach provides good solutions to our problem within timeframes that are acceptable in our context. As the aim of this algorithm is to maintain SLO violation rates below a pre-defined upper bound U_{thr}, genetic algorithms are able to achieve this for all runs requiring only 19 iterations.

Additionally, meta-heuristics, like genetic algorithms, provide ease of use and flexibility to add new selection criteria to given objective functions, e.g. data center failure rate to improve availability. Finally, even though algorithms based on genetic algorithm can be used to solve large-sized problems, they can suffer from long computational complexity. However such meta-heuristic can be easily parallelised as it implicitly supports parallelisation techniques 60. Furthermore, the resources required to realise parallel versions of genetic algorithms should not be an issue in our context because Cloud computing promises scalable, powerful compute-optimised resources. However, for our problem we found that proposed normal, non-parallelised versions of this algorithm was effective as will be discussed in section 5.

TABLE 2 Terms Used in Cluster Replacement Algorithm

Term	Meaning
r	Crossover rate
m	Mutation rate
g	Number of generations (iterations)
p	Number of individuals (feasible solutions) in the population
g_{max}	Maximum number of generations
cands	Candidate deployments
$cand_{best}$	Best candidate deployment (individual) with maximum fitness

4.7 | Genetic Algorithm for Cluster Replacement

A genetic Algorithm provides a meta-heuristic approach based on the realisation of natural iterative improvements in population genetics. It relies on a set of bio-inspired operators, e.g., selection, crossover and mutation to iteratively modify populations and thus evolve successive populations. Using a genetic algorithm to design an algorithm requires a genetic representation used to encode candidate solutions based on the problem domain and a fitness function to evaluate the quality of each solution.

For the genetic representation in this work, a chromosome (or an individual) can be represented by a non-duplicated set of data centers (i.e., a current/candidate deployment with **D/F**) where an individual gene of the individual is represented by a data center. Duplicated genes in an individual are forbidden and the order of the genes are unimportant. For increased efficiency, genes are enumerated to rapidly detect duplicate genes.

For the fitness function, the objective function in Eq. 1 is used to select an individual (i.e., a candidate deployment,

cand) with the highest fitness. A fitness function is presented as:

$$fitness(current, cand, W_t) = max\ f(current, cand, W_t) \quad (5)$$

A genetic algorithm-based cluster replacement algorithm is shown in Algorithm 2. Using the symbols defined in Tables 1 and 2, the algorithm works as follows. The inputs of the algorithm are the number of required data centers of a candidate deployment (i.e., number of required genes to form an individual), the current deployment, *current*, as well as accumulated workloads collected from running clusters at time t, W_t. The algorithm initialises a population, P, by randomly generating p candidate deployments using available data centers in **A** (Line 2). Duplicate individuals are not allowed in a population. The algorithm then evaluates each candidate deployment in the population, P, and then sorts candidate deployments and the number of generations, g (Lines 3-6).

The algorithm then iterates to produce new successive generations by applying the bio-inspired operators: selection, crossover and mutation (Lines 7-23). At each iteration, it creates an empty new generation, P_{new} (Line 8). Then it applies a selection operator by selecting $(1-r)\cdot p$ candidate deployments from the start of the population, P, and adds them to the new population, P_{new}, (Lines 9-10). The crossover operator is then applied (Lines 11-13). This operator selects $\frac{r\cdot p}{2}$ pairs of candidate deployments from the beginning of the population, P. For each pair, it produces two children (two new candidate deployments) by randomly selecting a data center from each pair of members and then swaps the two selected data centers. It then adds all children to the new population, P_{new}.

The algorithm then applies the mutation operator (Lines 14-15) by randomly choosing $m\cdot p$ candidate deployments from the new population, P_{new}, and replaces a randomly selected data center from each chosen deployment with a randomly selected data center from **A**. It updates the population, P, and then evaluates and sorts candidates in descending order (Lines 16-18). If the fitness of the best candidate deployment is less than the fitness of the first candidate deployment in the current population, P, it updates the best candidate deployment with the first candidate deployment in the current population (Lines 19-21). Then, the number of generations, g, is incremented by 1 (Line 22). The algorithm terminates when the number of generations, g, reaches the maximum number of generations, g_{max}. The algorithm returns the best candidate deployment, $cand_{best}$, that has the maximum fitness (Line 24).

In terms of the time complexity of the genetic algorithm, the approach requires calculating the running time of an iteration (generation) using the terms given in Tables 1 and 2.

Algorithm 2: Genetic Algorithm for Cluster Replacement

Input : N, *current*, W_t

1 Set the number of data centers (genes) of a candidate deployment to N;
`/* Initialise population */`
2 $P \leftarrow$ Generate p candidate deployments (cands) randomly using data centers in **A**;
`/* Evaluate */`
3 For each *cand* in P, compute $fitness(cand)$ using Eq. 5;
4 Sort cands in P by their (descending) fitness;
5 $cand_{best} \leftarrow$ first *cand* in P;
6 $g = 1$;
7 **while** $g < g_{max}$ **do**
`/* Create a new generation P_new */`
8 $P_{new} \leftarrow \emptyset$
`/* Apply selection */`
9 Select $(1-r)\cdot p$ cands from the start of P;
10 Add selected cands to P_{new};
`/* Apply Crossover */`
11 Select $\frac{r\cdot p}{2}$ pairs of cands from the start of P;
12 For each pair, $\langle c1, c2\rangle$, produce two children (new cands) by swapping a randomly selected data center from $c1$ with a randomly selected one from $c2$;
13 Add all children to P_{new};
`/* Apply mutation */`
14 Choose $m\cdot p$ cands randomly from P_{new};
15 For each chosen *cand*, replace a randomly selected data center of *cand* with a randomly selected data center from **A**;
`/* Update */`
16 $P \leftarrow P_{new}$;
`/* Evaluate */`
17 For each *cand* in P, compute $fitness(cand)$ using Eq. 5;
18 Sort cands in P by their (descending) fitness;
19 **if** $fitness(cand_{best}) < fitness(first\ cand\ in\ P)$ **then**
20 $cand_{best} \leftarrow$ first *cand* in P;
21 **end**
22 Increment g by 1;
23 **end**
24 Return $cand_{best}$ that has the maximum fitness;

The selection, crossover and mutation operators at each iteration require $O((1-r)\cdot p)$, $O(r\cdot p)$ and $O(m\cdot p)$,

respectively. With regards to the evaluation phase, computing the fitness of each individual in a population, P, requires finding the best data center (gene) with the least network latency overheads in for each W_t and this requires $(N-1)$ comparisons. As a result, evaluating an individual requires $O(|W_t| \cdot N)$ thus evaluating all individuals in the population requires $O(p \cdot |W_t| \cdot N)$. Also, sorting individual of a population requires $O(p \log p)$. Hence, the evaluation phase runs in $O(max\{p \cdot |W_t| \cdot N, p \log p\}) = O(p \cdot |W_t| \cdot N)$. As a consequence, the running time of an iteration is given by $O(max\{(1-r) \cdot p, r \cdot p, m \cdot p, p \cdot |W_t| \cdot N\}) = O(p \cdot |W_t| \cdot N)$. The time complexity of genetic algorithm is therefore given by $O(g \cdot p \cdot |W_t| \cdot N)$.

5 | PERFORMANCE EVALUATION

To evaluate the proposed geo-elastic deployment solution, experiments were carried out on the Australia-wide National eResearch Collaboration Tools and Resources (NeCTAR - www.nectar.org.au) research Cloud. Experiments are classified into three sets. In the first set, we study the behaviour of our geo-elastic deployment approach towards geo-workload variations to maintain performance and cost using the SLO-based violation rates, SBVR, and the number of container clusters as performance and cost metrics respectively. We also examine the cost of adaptation.

In the second set of experiments, we evaluate the proposed geo-elastic deployment approach on the NeCTAR Cloud. This set considers end-to-end response times to requests after adapting deployments according to geo-workload changes, as well as the number of container clusters as the key evaluation metrics. In the third set of experiments, we evaluate the effectiveness of the proposed cluster replacement algorithm, genetic algorithm, using the metrics, SBVR and the execution time.

To clarify, the operating costs of a given deployment in the Cloud are proportional to the total number of container clusters from multiple perspectives. First, each container cluster incurs a management fee, e.g. Kubernetes clusters in Google and Amazon Web Services costs $0.10 per cluster per hour. A deployment of 3 clusters can reduce the total cost of management fees by 50% compared to one with 6 clusters. Secondly, more container clusters used require more VMs to run master and worker nodes. For master nodes, highly available clusters require at least 2 master nodes per cluster (i.e. 2 VMs). A deployment of 3 clusters requires 6 VMs while a single cluster requires only 2 VMs. For worker nodes, increasing the number of clusters can reduce the chance of condensing containers to fewer VMs. For instance, assume each worker node runs on a VM that consists of 4 CPUs and 2 user workloads from different locations require 2 and 6 CPUs respectively. If containers handle these two loads run in 2 separate clusters, the worker nodes for clusters will require 1 and 2 VMs respectively, i.e. a total of 3 VMs. However, if they can run in one cluster and ensuring that the Cloud location is carefully selected to optimise network latency for both user workloads, their containers will be condensed into only two 2 VMs. Hence this deployment reduces the cost by 33% compared to the one with two clusters.

In all experiments, we refer to the SLO as the pre-agreed upper level of a response time for a user request, which includes the network latency between a user and a data center, the processing time of the request and communication overheads when interacting with the data center. This represents the threshold that application/platform providers should preserve for their response times to achieve a satisfactory user experience. Common settings among all experiment sets are explained in the next section. Individual settings are described separately within each experiment set.

5.1 | General Settings

Cloud Data Centers and Compute Resources Settings

We ran extensive experiments on the NeCTAR Research Cloud. The NeCTAR Cloud is a geographically distributed Cloud comprising 19 availability zones (data centers in our context) distributed around Australia. For cluster replacement problem, since this number of available data centers is relatively small, we added another 41 data centers (60 in total) scattered across the globe including AWS Singapore, London, Cape Town, Stockholm and numerous others. This expands the search space and thus allows to accurately evaluate the performance of the search algorithms in finding optimal solutions from an expanded and realistic global Cloud search space. To make sure that the candidate deployments with SLO violation rates were below a pre-defined upper bound for acceptable violation rates, we selected global data centers specifically to be highly distributed and hence likely to incur network latencies. Therefore, selecting one (or more) of these data centers will increase the estimated violation rate and hence increase the likelihood that it will go beyond the upper bound of the acceptable violation rate. This will make the search algorithms explore more solutions and ideally demonstrate that they select NeCTAR data centers for their optimal deployment locations.

With regards to compute resources, every VM instance used in all experiments was assigned the following resources: 4 virtual CPUs and 16 GB of RAM running Ubuntu 18.04 LTS (Bionic) as the operating system.

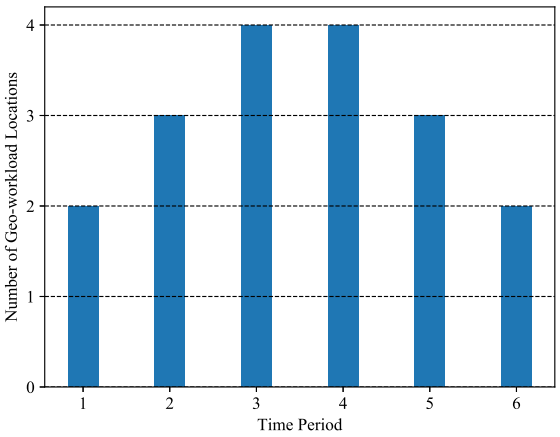


FIGURE 3 Geographic workload changes over time

TABLE 3 Workload locations over time periods for the different experiments.

Time Period	Workload Locations
1	Canberra, Melbourne
2	Canberra, Melbourne, Brisbane
3	Canberra, Melbourne, Brisbane, Sydney
4	Canberra, Melbourne, Sydney, Tasmania
5	Melbourne, Sydney, Tasmania
6	Melbourne, Tasmania

Geo-workload Variation Scenarios

Five different locations around Australia were chosen to be sources of workloads: Brisbane, Canberra, Melbourne, Sydney and Tasmania. To simulate real-world geo-workload change scenarios over time, we synthesised a series of consecutive geo-workload variation scenarios as shown in Figure 3 and Table 3. This was used to generate spatial workload variations in user growth across different locations, e.g. cities, across Australia. We targeted workloads within Australia because, as discussed earlier, our experiments were carried on the Australia-wide NeCTAR Research Cloud. Also, we refer to time periods as blocks of time where in the beginning of each block of time, a geo-workload variation scenario occurs.

In experiments sets 1 and 2, we used these scenarios to reflect spatial changes in workloads over time to understand how the proposed and baseline approaches react to those changes. Additionally, in experiment set 2 we set the length of each period to 1,200s. Changes in geo-workloads were carefully chosen to allow geo-elastic deployment approaches to make three possible elastic decisions: geo-shrinking, geo-relocation and geo-expanding. It is noted that to infer an elastic decision at any period in these experiments, we calculate the difference between the size of the deployment at that period and the one beforehand. The size of deployments over periods are shown in Figures 4c, 4d and 6b. If the difference is greater than 0, then the decision is geo-expanding. If it is less than 0, then it is geo-shrinking. Otherwise, the decision is geo-relocation or no change.

The Proposed Geo-elastic Deployment and Baselines Settings

In these experiments, the default settings of the geo-elastic deployment controller's parameters T, P, τ, U_{thr}, G_{thr} were set to 2 min, 20 min, 1 period, 10% and 3% respectively. For genetic algorithm parameters, we set r, m, p and g_{max} to 0.7, 0.2, 60 and 120 respectively. Tuning genetic algorithm parameters for our problem is discussed in section 5.4.2. We refer to our approach as **Geo-elastic (proposed)**.

Regarding baselines, we have two baseline approaches: over-provisioning and static deployment. Over-provisioning approach is a latency-sensitive, cost-unaware geo-elastic deployment solution that only considers proximity to users. It adapts deployments with the aim of improving the performance by selecting data centers for clusters such that the network latency between application users and data centers hosting container clusters regardless is minimised. It is independent of the cost of the adaptation. This approach is similar to work described in 11 as it only considers network latency between users and data centers and completely ignores inter-DC latency. Furthermore work in 25 only considers inter-DC latency and ignores inter-data center latency as the cost of adaptation. We refer to this approach as **Over-provisioning**. Nevertheless, the parallels of using over-provisioning for the baseline is consistent with these other works.

The static deployment approach, as its name suggests, is a non-elastic approach used for the number and location of clusters, i.e., it does not include the spatial aspect of adaptation. It also only supports local elasticity. In all experiments, the deployment size of this approach was set to two container clusters and the clusters were statically located at two data centers. We refer to this approach as **Static**. This approach has no cost of adaptation since it is static.

Network Latency Data

To estimate unknown network latencies between users and data centers, we use an approach based on our previous work 12 that relies on the distance between them. The approach simply relies on a correlation between network latencies and geographic distances between data centers. We empirically measured the round trip time among all data centers using the `ping` utility to obtain network latencies between data centers.

To calculate the geographical distances we used the Harversine formula. We use this approach because the correlation coefficient is found to be strongly positive (0.97).

5.2 | Experiment Set 1: Evaluation of the Proposed Geo-elastic Deployment Approach

The aim of this experiment set is to investigate the effect of our geo-elastic deployment approach on maintaining the performance as well as the cost of deployment and adaptation, when adapting cluster deployments to handle geo-workload changes. In this set, the delay, caused by the processing time of a user request as well as all inner-data center communication overheads, is constant and fixed to 10 ms. We run experiments 32 times (except for the static case) .

5.2.1 | User Setting

To simulate realistic workloads (user request logs), W_t, to be used as input to geo-elastic deployment approaches, we first model 300 user sessions at each workload location and set the number of requests per session to 10. Then, we obtain realistic geo-locations of users for each workload location by using Twitter data collected from each workload location, extracting the geo-locations of tweeters and then assigning those geo-locations to the users in our model. When generating user request log files for container clusters, we inject the geo-location of a user instead of the IP address at each request record. The user geo-locations and number of requests for each user are key information needed from the workloads. Finally, user workloads at each time period of geo-workload variation scenarios as shown in Figure 3 and Table 3 are generated depending on the number and locations of geo-workloads over that period.

5.2.2 | Experimental Procedure

In the first experiment, we set SLO to 20 ms and the current deployment to the initial deployment using two clusters. Then, we run experiments for each stochastic approach 32 times independently on a VM. At each run, the geo-elastic deployment controller iterates 6 times to simulate the number of time periods shown in Figure 3 and Table 3. At each period, the controller retrieves the workloads over that period. Then, the controller either makes an elastic decision and produces a new deployment plan or leaves the current deployment as it is. Then, we evaluate the new/current deployment and record the SBVR for the number of clusters. We also record the cost of any adaptation (the number of relocated/new clusters and the total inter-data center latency) if there is a change in the deployment at that period that is put forward.

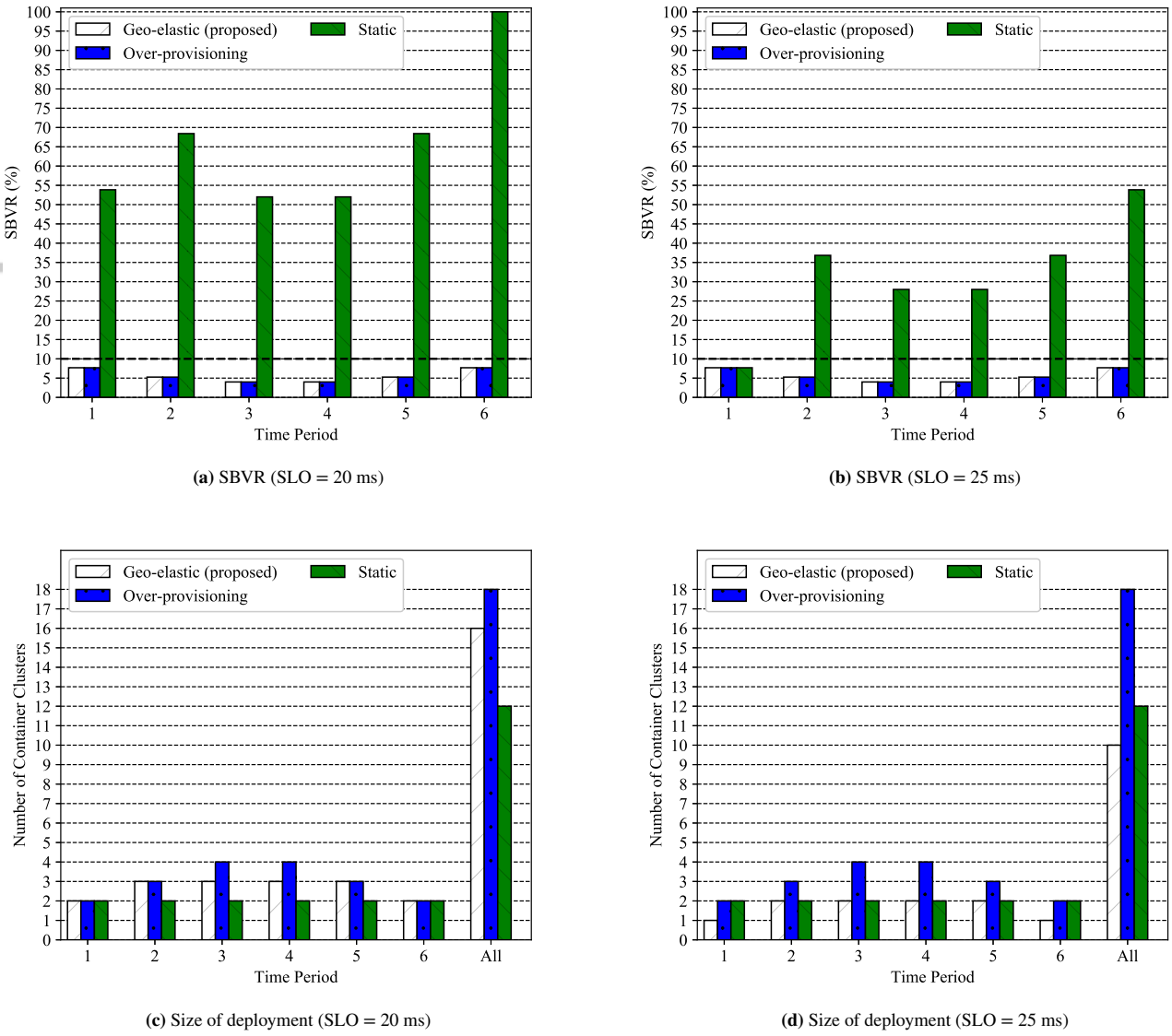
For the static deployment approach, we run the experiment once since it is a deterministic approach. We simply evaluate the single, static deployment against workloads over all periods. In the second experiment, we set SLO to 25 ms and then repeat the same steps followed in the first experiment.

5.2.3 | The Impact of Geo-elastic Deployment on Performance and Cost

In this section, we present the impact of considering geo-elastic deployment solutions as well as the importance for those solutions on both performance and cost when adapting deployments of container clusters with geo-workload variations across geo-distributed data centers. Figures 4a and 4b indicate that all geo-elastic deployment approaches (geo-elastic and over-provisioning) undoubtedly show very low SLO violation rates, SBVR, at all periods for both SLO settings (20 and 25 ms). They adapt deployments against workload changes that result in successfully maintaining SBVR below the pre-defined upper bound of acceptable SBVR ($U_{thr} = 10\%$) in all cases. On the other hand, the static approach unsurprisingly incurs significantly higher SBVR and exceeds 10% at all periods for both SLO settings except for period 1 when SLO is 25 ms.

Furthermore, while the geo-elastic and over-provisioning approaches have the same SBVR level in all cases, our approach as shown in Figures 4c and 4d, successfully reduce the cost in most cases, especially when SLO is relaxed further (SLO = 25 ms). For SLO with 20 ms, our approach shrinks the size of deployment by one cluster at periods 3 and 4 when compared to the over-provisioning approach. Moreover Figure 4d shows that relaxing SLO by only 5 ms (i.e., from 20 to 25) is exploited to reduce the number of running clusters by one at every period, compared to the 20-ms SLO setting. Overall, our approach shows a 37.5% improvement in cost when the SLO is relaxed to 25 ms as the total number of clusters for all periods decreases from 16 to 10 clusters. As these results show, the over-provisioning approach is very costly since it requires a total of 18 clusters for all periods, even after relaxing the SLO to maintain the system performance.

The reason behind this improvement is that the cost-effectiveness in our approach, as discussed in section 4.6, balance performance and cost (where possible) by sacrificing an acceptable amount of performance to reduce the cost, i.e., they tolerate acceptable increases in user-to-data center network latencies to select data centers that are moderately distant from users with a lower number of clusters. This sacrifice need not result in response times that violate the defined SLOs and upper level threshold for the SBVR. On the other hand, the over-provisioning approach only considers improving performance by maximising the reduction in SBVR resulting in



(a) SBVR (SLO = 20 ms)

(b) SBVR (SLO = 25 ms)

(c) Size of deployment (SLO = 20 ms)

(d) Size of deployment (SLO = 25 ms)

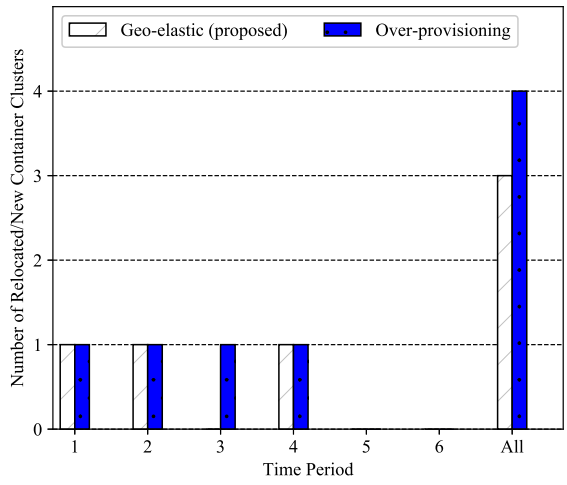
FIGURE 4 Performance and cost comparison with 3 different approaches for multi-cluster deployment against geographic workload changes over 6 time periods using various SLOs. The lower the SBVR and size of deployment the better. Each approach with different SLOs ran 32 times.

provisioning more clusters at different data centers close to geo-distributed users.

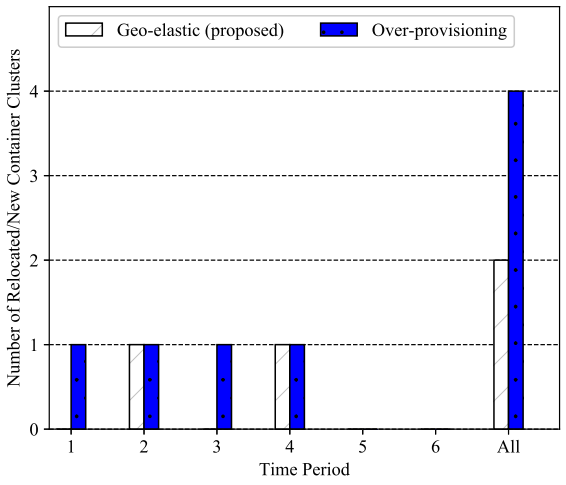
From these results, it is evident that geo-elastic deployment with cost and latency awareness plays a crucial rule in improving performance under cost constraints.

5.2.4 | The Cost of Adaptation

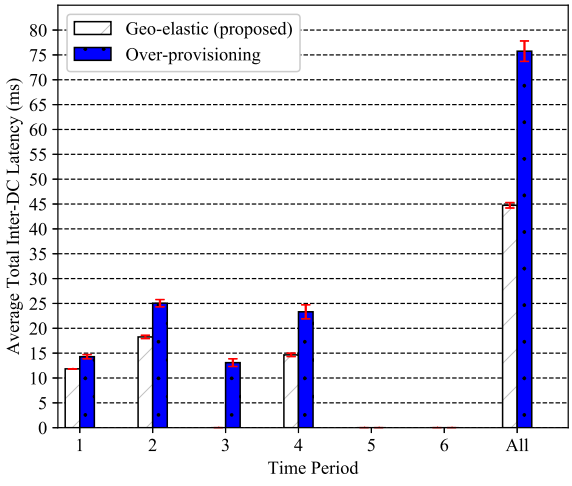
In this section, we evaluate the impact of the cost of adaptation on geo-elastic deployment approaches. As discussed, we refer to the cost here as the number of relocated/new container clusters combined with the total inter-data center network latency when a geo-elastic deployment solution adapts a given deployment. As illustrated in Figure 5, our geo-elastic approach avoids the cost of adaptation at period 3 and at periods, 1 and 3, when the SLO set to 20 and 25 ms respectively. Over-provisioning solutions, on the other hand, incur costs at periods, 1, 2, 3 and 4, for both SLO settings. It should be noted that all approaches have no cost of adaptation at periods, 5 and 6, because the elastic decision is for geo-shrinking or no change required at those periods. The static deployment solution is ignored since it does not have an adaptation ability.



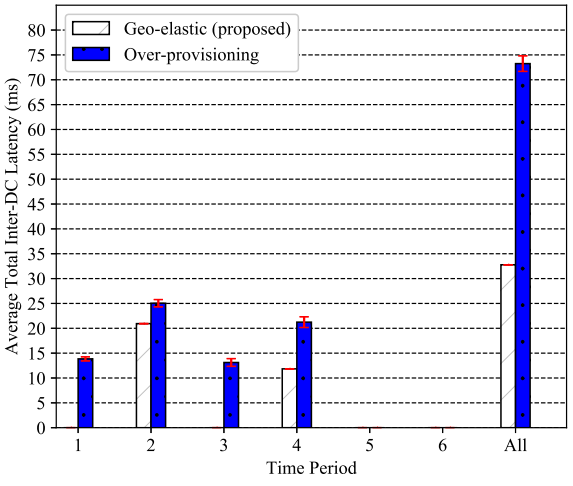
(a) Number of Relocated/New Container Clusters (SLO = 20 ms)



(b) Number of Relocated/New Container Clusters (SLO = 25 ms)



(c) Average total inter-data center (DC) latency (SLO = 20 ms)



(d) Average total inter-data center (DC) latency (SLO = 25 ms)

FIGURE 5 Cost of adaptation of 2 different geo-elastic deployment approaches against geographic workload changes over 6 time periods using various SLOs. Each approach has different SLOs and is run 32 times.

Overall, in terms of the number of relocated/new clusters, our approach reduces the cost of relocating running clusters or adding new clusters by 25% for all periods of SLO with 20 ms, compared to the over-provisioning approach as shown in Figure 5a. When the SLO is relaxed as shown in Figure 5b, our solution takes advantage of this relaxation and decreases the cost by 50%, compared to the over-provisioning case. With regards to the total inter-data center network latency, as shown in Figures 5c and 5d, our approach obviously has less inter-data center latencies than those of over-provisioning in all cases. Thus our approach is capable of speeding up the adaptation of cluster deployments as well as maintaining better response times during adaptation. We can conclude that such cost of adaptation considerations can help geo-elastic deployment solutions to minimise the number of relocated/new clusters as well as the total inter-data center network latencies.

5.3 | Experiment Set 2: Evaluation of the Proposed Geo-elastic deployment Solution in a Real Cloud Context

The aim of this experiment set is to evaluate the deployments made by the geo-elastic deployment approaches in real distributed-Cloud contexts using the NeCTAR Cloud to cope with geo-workload variations. Workloads here are realistically generated from different locations. We show that our

geo-elastic deployment approach is capable of maintaining performance and meeting SLOs (or at least minimising SLO violations) at lower operational costs. A key web application metric, end-to-end response times, and the number of operating container clusters are used as evaluation metrics for performance and cost.

The assumptions considered regarding processing times of requests as well as internal-data center communication delays are relaxed here because these times and delays vary in real Cloud experiments. Since our geo-elastic deployment approach considers performance related to network latencies (to improve geo-scalability), we consider mitigating the impact of this variation and avoid any possible overload issues by taking the two following steps. First, we generate workloads for experiments in a moderated way. Second, we provision more resources for each cluster to allow the cluster platform (Kubernetes) auto-scale the containers when needed (i.e., providing enough resources as local elasticity out of scope).

5.3.1 | Experimental Set-up

SLO and the Geo-elastic Deployment Settings

We set the SLO to 25 ms. As with standard benchmarks, we use the 95th percentile of response time as a benchmark where it should be within 25 ms. We set the upper bound of acceptable SBVR, U_{thr} to 5%.

Realistic Workload Generation and Request Routing

To generate realistic workloads from the five locations discussed in Section 5.1, we use a workload generator running on a VM at each location and a single workload manager running on a separate VM. Each workload generator consists of Locust, an open-source modern load testing framework and an agent. The agent runs as a daemon and waits for commands from the manager to activate/deactivate the generation of user requests. It also checks the health of the target cluster using heartbeats and records the response times of requests. We set the number of concurrent user sessions at each generator to 80 users. Each user session consists of 10 requests: 7 read and 3 write requests.

The workload manager simulates geo-workload changes over different time periods (see Figure 3 and Table 3) and provides a request routing service similar to Geo-DNS services. At the beginning of each period, the manager selects the required workload generator at that period and sends appropriate commands to agents to activate workload generators or deactivate unnecessary workload generators related to previous period (if required).

The request routing service routes traffic for workload generators at each period (first case) and reroutes traffic when adapting deployments requires changes in the IP addresses of clusters (second case). In the first case, when a workload generator is activated, it asks the request routing service for an IP of a cluster (similar to Geo-DNS lookup for IPs). The request routing service responds with an IP of a cluster in the current deployment at that time with the least network latency. In the second case, once a new deployment of clusters takes place, the request routing service is updated with new IPs of clusters, if any. Then, if any agent of running workload generators does not receive any response from the target cluster, it asks the request routing service for a new IP of the cluster. The request routing service then responds with a new IP (i.e., redirecting traffic).

Sample Application and Container Cluster Platforms

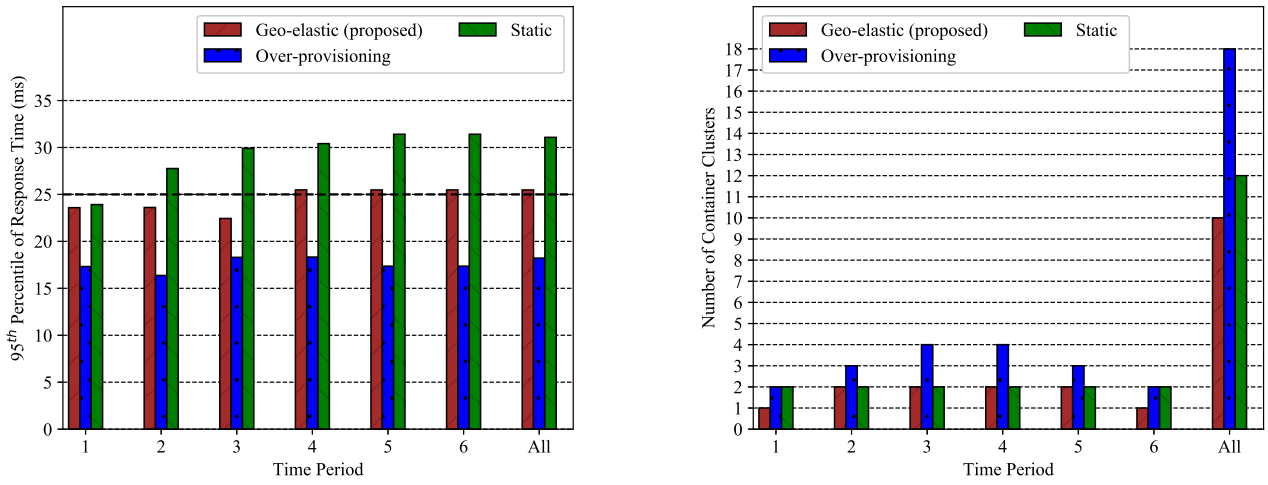
For web application benchmarking, we use a real-world transactional web e-Commerce benchmark (TPC-W) application 61, which simulates business-oriented activities of an online bookstore. A Java implementation of TPC-W is used where the main application components are containerised individually using Docker. Those components include Tomcat (v8.5) as a web server, Couchbase database (v5.5.0) as a user session manager and MySQL database (v5.7) as the application database. A copy of the application data exists at each data center prior to any load on the system.

Whenever a cluster is required at a data center, 3 VMs are provisioned at that data center as cluster nodes. For each node we use Docker (v18.06.2) 49 as a container Runtime and Kubernetes (v1.16.3) 6 as a cluster platform with HAProxy load balancer (v1.9) as an ingress controller for Kubernetes. One node is a master while the others are workers. The initial deployment of application services has 3 web servers, 1 session manager and 1 database deployed as containers.

Multi-cluster Container Platform

The multi-cluster container platform runs on a VM. It has the components for the geo-elastic deployment framework implemented in Python. Since this work focuses on decision-making components, only the necessary aspects of the action-components are implemented to complete the evaluation. Therefore, when adapting a deployment, if the provisioned cluster is with a *new* condition, then a deployment configuration (YAML file) is passed to the master, which pulls container images from the image registry.

If the cluster is with a *migrating* condition then several steps are necessary. First, a new master VM at the destination data center joins the running cluster at the source data center as another master replica (to share the current cluster state). Then, the two new destination workers join the cluster



(a) 95th percentile of response time

(b) Size of deployment

FIGURE 6 Performance and cost comparison of 3 different approaches for multi-cluster deployment running on the Australia-wide NeCTAR Cloud and reacting against geographic workload variations over 6 time periods. At each period, workloads are generated from different locations around Australia to simulate geographic changes over time. Each period lasts for 20 min (1200s). Each experiments was run 3 times. (SLO = 25 ms).

as worker nodes. The Kubernetes cluster is then instructed to drain the two source workers to evict all running pods (containers). In this case, all evicted pods will be scheduled to run at the new workers at the destination data center. Finally all nodes at the source are removed and hence the cluster relocation is complete.

5.3.2 | Experimental Procedure

We run experiments for each approach three times. For all experiments, we fix the locations of two clusters for the initial deployment. At each run, once the clusters of the initial deployment are ready at the pre-determined data centers, their IPs are registered at the request routing service. Then, the workload manager iterates over the 6 periods of time. At each period, the following steps are repeated. First, the workload manager selects workload generators to start generating requests for that period. Next, the geo-elastic deployment approach collects log files. It makes scaling decisions and actions if needed to adapt the current deployment. Whether the deployment has been adapted or not, the request routing service is updated with IPs and the number of clusters for that period is recorded. Once the request routing service is updated, the workload manager instructs running workload generators to start recording response times for the length of the period (1200 seconds with a resolution of one second). For the static deployment approach, the workload manager, at each period, selects workload generators and records the response times immediately.

5.3.3 | Results and Discussions

Maintaining Performance at Lower Cost

Figure 6 displays the 95th percentile of response time as well as the size of the deployment of the three approaches. In terms of performance, as Figure 6a shows, the 95th of response time of the over-provisioning for every period unsurprisingly satisfies the SLO, 25 ms. For our approach, the 95th percentiles of response time, for periods, 1, 2 and 3, meet the SLO while for the other periods, 4, 5 and 6, they exhibit very mild violations (i.e., only 0.5 ms above the SLO). The static approach shows higher SLO violations by at least 5 ms for most periods. It should be noted that the 95th percentile of response time of the static approach for the first period meets the SLO because geo-workloads at the first period were close to the clusters of the static deployment approach by chance.

With regard to the cost, Figure 6b shows that the improvement in performance for over-provisioning comes at high cost for all periods. Our approach, when compared to the over-provisioning solution, shows a noticeably lower cost at every period. This gives a justification for our approach, i.e., we sacrifice small amounts of performance to reduce costs. In more detail, we improve the cost-effectiveness by reducing the size of deployment by 1 at periods (1, 2, 5 and 6) and by 2 at periods (3 and 4) compared to the over-provisioning approach. In

TABLE 4 The difference in response time and cost for different approaches for all time periods. Each approach runs 3 times. Geo: Geo-elastic (proposed). OP: Over-provisioning

	Difference in Response time Mean in ms			Difference in Cost	
	95% CI	Estimate	P-value	*n_clusters*	%
Geo - Static	(-3.24, -3.07)	-3.16	0.00	-2	-16.67
OP - Static	(-5.95, -5.81)	-5.89	0.00	6	50.00
Geo - OP	(2.68, 2.77)	2.72	0.00	-8	-44.44

other words, costs are reduced by 33% at periods (2 and 5) and by 50% at periods (1, 3, 4 and 6).

Table 4 shows the results comparing the three approaches using t-test. This indicates that over-provisioning and our approach, on average improve the response time by 5.89 ms and 3.16 ms respectively compared to the static approach. While this performance improvement requires an over-provisioning solution that increases the cost by 50%, our approach reduces the cost by approximately 16%. Moreover, compared to the over-provisioning approach, our approach incurs only minor delays in response time, on average only 2.71 ms whilst reducing the cost by 44.44%. As a consequence, it is evident that our approach has the ability to preserve performance and minimise SLO violations with at greatly reduced cost.

5.4 | Experiment Set 3: Evaluation of the Proposed Cluster Replacement Algorithm

In this experiment set, we evaluate the genetic algorithm-based cluster replacement algorithm. Experiments here are classified into two groups. In the first group, the aim of the experiments is to tune the genetic algorithm parameters for our problem. In the second group, we show the benefits of network latency awareness when (re)placing clusters as well as examine the performance of genetic algorithm. We use SBVR and the execution time as key evaluation metrics.

For all experiments here, we set problem-specific parameters as introduced in the next section.

5.4.1 | Problem-specific Parameter Settings

We set SLO and processing time to 20 ms and 10 ms, respectively. For the workload, W_t, we choose five locations to generate workload and set the number of user sessions to 240 per location, i.e., we have a total of 1200 users. The size and SBVR of the current deployment is 3 and 76.40% respectively. The number of required clusters for the new deployment N is set to 4. As mentioned before, there are 60 potential data centers available, hence we have a total of 487,635 candidate deployments (feasible solutions).

TABLE 5 Estimate and standard error (SE) of the average fitness and SBVR for genetic algorithm (GA) over 32 runs as well as the number of generations for all runs to converge using different GA parameter settings. The higher the fitness the better. The lower the $SBVR$ the better.

GA parameters		Iterations to Convergence (all runs)	Average Fitness		Average SBVR (%)	
Crossver	Mutation		Estimate	SE	Estimate	SE
0.7	0.2	54	-0.89	0.02	7.39	0.09
0.5	0.1	61	-0.9	0.02	7.46	0.09
0.7	0.1	61	-0.87	0.02	7.33	0.09
0.7	0.7	68	-0.85	0.02	7.22	0.08
0.5	0.5	88	-0.88	0.02	7.32	0.09

5.4.2 | Group 1: Genetic Algorithm Parameter Tuning

Two key parameters of the genetic algorithm are the crossover rate, r, and the mutation rate, m. These need to be tuned for the cluster replacement problem. Therefore, we run 16 experiments with the genetic algorithm where each experiment has a different parameter setting. To obtain these 16 parameter settings, we combine different values of the two parameters. The values for r and m are set to 0.2, 0.5, 0.7 and 0.9 and to 0.1, 0.2, 0.5, 0.7 respectively.

Tables 5 shows the best five parameter settings for the genetic algorithm ordered based on their speed rate (i.e., number of iterations) to converge on the problem over the 32 runs. From the results, it is evident that the fastest rate of convergence occurs when setting r and m to 0.7 and 0.2 respectively. The convergence curve using these parameter settings is indicated in Figure 7a.

5.4.3 | Group 2: Effectiveness of Genetic Algorithm

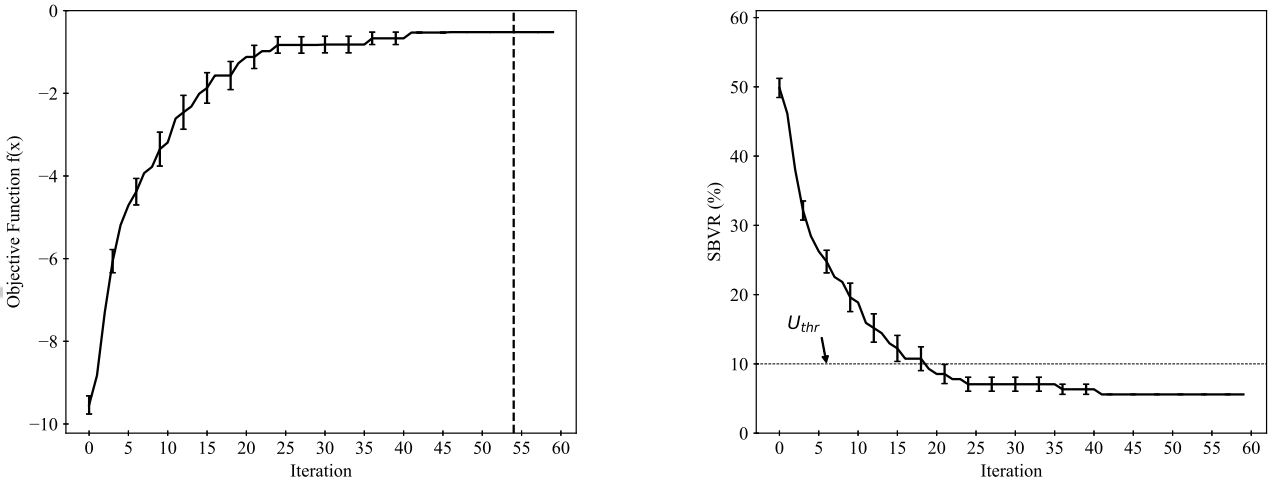
Proposed Algorithms and Baselines Settings

Genetic algorithm parameter settings were discussed in section 5.1. Regarding the baselines, we have two baseline algorithms: Brute force algorithm, which examines all possible solutions in the solution space and latency unaware algorithm, which does not consider network latency and randomly selects data centers for cluster (re)placement.

For stochastic algorithms genetic algorithm and latency unaware, we run experiments on each algorithm 32 times. We also run the deterministic algorithm, brute force, once. The algorithms run independently on the VMs.

Impact of Network Latency Consideration

Table 6 indicates that the latency unaware algorithm shows a very high violation rate in SBVR, 90.41%, while our genetic



(a) Convergence curve

(b) SBVR improvement

FIGURE 7 Convergence curve and SBVR improvement for proposed genetic algorithm. The algorithm is run 32 times and converges at the 54th iteration. The higher the objective function value, the better (the optimal value is -0.52). The lower the SBVR the better (the optimal SBVR value is 5.6%).

TABLE 6 95% confidence interval (CI), estimate and standard error (SE) of the mean of the objective function value and $SBVR$ for different algorithms run 32 times. The higher the objective function value, the better. The lower the $SBVR$ the better. The optimal SBVR value is 5.6%. GA: Genetic Algorithm. LU: Latency Unaware.

Algorithm	Objective function value			SBVR (%)		
	95% CI	Estimate	SE	95% CI	Estimate	SE
GA	(-0.96, -0.82)	-0.89	0.04	(7.03, 7.75)	7.39	0.18
LU	(-19.22, -17.21)	-18.22	0.51	(85.46, 95.36)	90.41	2.53

TABLE 7 95% confidence interval (CI), estimate and p-value of the difference in means of the objective function value and $SBVR$ between different algorithms. Each algorithm ran 32 times. GA: Genetic Algorithm. LU: Latency Unaware.

Difference in Mean	Objective function value			SBVR (%)		
	95% CI	Estimate	P-value	95% CI	Estimate	P-value
GA - LU	(16.29, 18.37)	17.33	0.00	(-88.16, -77.88)	-83.02	0.00

algorithm, which takes into account network latency, has very low SBVR violation rates, 7.39%. As shown in Table 7, the genetic algorithm improves the performance since it reduces SBVR by at least 83.02%. In other words, compared to latency unaware algorithm, the genetic algorithm achieves 91.83% improvement in SBVR. It can therefore clearly be concluded that network latency considerations reduce SLO violation rates and hence improve performance.

The Performance of Genetic Algorithm

In this section, we examine the performance of the proposed genetic algorithm from two aspects: speed and accuracy. For speed, we use two metrics: the execution time (in seconds) as well as rate of convergence (number of iterations to converge for all runs). Regarding the accuracy, we evaluate the accuracy of the algorithm by measuring the average SBVR and determining how far this average differs over iterations from the optimal SBVR. The optimal SBVR value, which is obtained from the brute force algorithm, is 5.6%.

With regards to execution time, the execution time of brute force algorithm is 27,859s (approximately 8 hours). The estimate and 95% confidence interval (CI) of the average execution time over the 32 runs for the genetic algorithm are 347.6s (about 6 min) and (347.11s, 348.08s) respectively. Compared to brute force algorithm, the genetic algorithm is dramatically faster. Regarding the rate of convergence, as Figure 7a shows, the genetic algorithm converges at the 54th iteration.

With respect to accuracy, as shown in Tables 6 and 7, the distance for our genetic algorithm to the optimal value is only 1.79%. Moreover, as indicated in Figure 7b, the genetic algorithm obviously meets the upper bound of acceptable violation rate U_{thr} for all runs at the 18th iteration. This shows that it provides acceptable level of accuracy. We can conclude that the genetic algorithm achieves good quality of solutions.

6 | CONCLUSIONS AND FUTURE DIRECTIONS

We have proposed a geo-elastic container deployment approach for multi-cluster deployment that leverages the capabilities of distributed, potentially global scale Cloud environments to elastically and intelligently scale web applications. The approach enables container platforms to automatically adapt the deployment of container clusters based on geographically diverse workload variations. The aim is to maintain system performance to meet/support SLOs even during the adaptation process, whilst minimising operational costs. For cluster replacement, a genetic algorithm, considering proximity to users and the cost of adaptation, i.e., the number of relocated/new clusters and inter-data center latencies, was explored. A heuristic for cluster quantity adjustment was presented. We also presented a framework to show how automated elastic multi-cluster deployment is enabled. To evaluate our approach we carried out extensive experiments on the NeCTAR Research Cloud using Kubernetes clusters and TPC-W web application and demonstrated optimal deployment solutions that minimise cost and meet performance demands.

Our future work will focus on cross-cluster resource management as Cloud-based elasticity solutions are not suited to handle unexpected, large scale and bursty overloads due to the overheads of provisioning VMs. This overhead usually lasts for a few minutes before the cluster node is ready to run new containers. During this time, users requests may be dropped or they may experience delays in response times. To handle this problem, we intend to propose a cross-cluster resource management mechanisms that allows overloaded clusters to borrow already-running (idle) VMs from other clusters with normal or reduced loads in different Cloud locations. This mechanism is more suited to handle sudden spikes in loads due to the warm-started VMs.

References

1. Gartner . Competitive Landscape: Container Management Software, 2019. .
2. IBM . IBM Cloud Pak for Multicloud Management. .
3. Ambasssador . Understanding Multi-Cluster Kubernetes. .
4. Burns B, Grant B, Oppenheimer D, Brewer E, Wilkes J. Borg, Omega, and Kubernetes. *Queue* 2016. doi: 10.1145/2898442.2898444
5. Leung A, Spyker A, Bozarth T. Titus: Introducing Containers to the Netflix Cloud. *Commun. ACM* 2018; 61(2): 38–45. doi: 10.1145/3152529
6. Kubernetes . Kubernetes: an Open-source System for Automated Container Deployment, Scaling, and Management of containerized applications. 2020.
7. Google Cloud . Anthos: A modern application management platform. .
8. Rancher Labs . Rancher: One Platform for Kubernetes Management. .
9. D2IQ . Using DC/OS to manage multiple clusters. .
10. Kratzke N. About the Complexity to Transfer Cloud Applications at Runtime and How Container Platforms Can Contribute? BT - Cloud Computing and Service Science. In: Ferguson D, Muñoz VM, Cardoso J, Helfert M, Pahl C. , eds. *Communications in Computer and Information Science*. 864. Springer International Publishing; 2018; Cham: 19–45
11. Guo T, Shenoy P. Providing Geo-Elasticity in Geographically Distributed Clouds. *ACM Trans. Internet Technol.* 2018; 18(3): 38:1–38:27.
12. Aldwyan Y, Sinnott RO. Latency-aware failover strategies for containerized web applications in distributed clouds. *Future Generation Computer Systems* 2019; 101: 1081–1095.
13. Nikolay G, Rajkumar B. Inter-Cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience*; 44(3): 369–390.
14. Nygren E, Sitaraman RK, Sun J. The Akamai Network: A Platform for High-performance Internet Applications. *SIGOPS Oper. Syst. Rev.* 2010; 44(3): 2–19.
15. Consulting F. eCommerce Web Site Performance Today: An Updated Look At Consumer Reaction To A Poor Online Shopping Experience. 2009.
16. Santos G, Paulino H, Vardasca T. QoE-aware auto-scaling of heterogeneous containerized services (and its application to health services). In: ; 2020
17. Toka L, Dobreff G, Fodor B, Sonkoly B. Adaptive AI-based auto-scaling for Kubernetes. In: ; 2020: 599–608
18. Al-Dhuraibi Y, Paraiso F, Djarallah N, Merle P. Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER. *IEEE International Conference on Cloud Computing, CLOUD* 2017; 2017-June: 472–479. doi: 10.1109/CLOUD.2017.67
19. Rossi F, Nardelli M, Cardellini V. Horizontal and vertical scaling of container-based applications using reinforcement learning. In: ; 2019
20. Herrera J, Molto G. Toward Bio-Inspired Auto-Scaling Algorithms: An Elasticity Approach for Container Orchestration Platforms. *IEEE Access* 2020. doi: 10.1109/ACCESS.2020.2980852
21. Zhong Z, Buyya R. A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources. *ACM*

Transactions on Internet Technology 2020. doi: 10.1145/3378447

22. Alfonso dC, Calatrava A, Moltó G. Container-based virtual elastic clusters. *Journal of Systems and Software* 2017; 127: 1–11. doi: 10.1016/J.JSS.2017.01.007
23. Srirama SN, Adhikari M, Paul S. Application deployment using containers with auto-scaling for microservices in cloud environment. *Journal of Network and Computer Applications* 2020; 160(March): 102629. doi: 10.1016/j.jnca.2020.102629
24. Al-Dhuraibi Y, Zalila F, Djarallah N, Merle P. Coordinating vertical elasticity of both containers and virtual machines. *CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science* 2018; 2018-Janua(Closer 2018): 322–329. doi: 10.5220/0006652403220329
25. Qu C, Calheiros RN, Buyya R. SLO-aware deployment of web applications requiring strong consistency using multiple clouds. In: ; 2015: 860–868.
26. Kubernetes . kops: Kubernetes Operations. .
27. Kubernetes . Kubespray: Deploy a Production Ready Kubernetes Cluster. .
28. Harshicop . Nomad: a simple, flexible, and production-grade workload orchestrator. .
29. OpenVZ . CRIU: Checkpoint/Restore In Userspace. .
30. Govindaraj K, Artemenko A. Container Live Migration for Latency Critical Industrial Applications on Edge Computing. In: . 1. ; 2018: 83–90
31. Red Hat Inc. . Red Hat OpenShift Container Platform. .
32. Nawab F, Agrawal D, El Abbadi A. The Challenges of Global-scale Data Management. In: No. 1 in SIGMOD '16. ACM; 2016; New York, NY, USA: 2223–2227
33. Amazon . Amazon Route 53: a scalable and highly available Domain Name System (DNS). 2018.
34. Microsoft Azure . Azure Traffic Manager. .
35. Liu Z, Lin M, Wierman A, Low S, Andrew LL. Greening geographical load balancing. *IEEE/ACM Transactions on Networking* 2015. doi: 10.1109/TNET.2014.2308295
36. Zhang Y, Wang Y, Wang X. GreenWare: Greening cloud-scale data centers to maximize the use of renewable energy. In: ; 2011
37. Toosi AN, Buyya R. A Fuzzy Logic-Based Controller for Cost and Energy Efficient Load Balancing in Geo-distributed Data Centers. In: ; 2015
38. Azimi S, Pahl C, Shirvani MH. Particle Swarm Optimization for Performance Management in Multi-cluster IoT Edge Architectures.. In: ; 2020: 328–337.
39. Qu C, Calheiros RN, Buyya R. Mitigating impact of short-term overload on multi-cloud web applications through geographical load balancing. *Concurrency Computation* 2017; 29(12): 1–15. doi: 10.1002/cpe.4126
40. Brogi A, Carrasco J, Cubo J, et al. SeaClouds: An open reference architecture for multi-cloud governance. In: ; 2016
41. Grozev N, Buyya R. Multi-cloud provisioning and load distribution for three-tier applications. *ACM Transactions on Autonomous and Adaptive Systems* 2014. doi: 10.1145/2662112
42. Brogi A, Forti S. QoS-aware deployment of IoT applications through the fog. *IEEE Internet of Things Journal* 2017. doi: 10.1109/JIOT.2017.2701408
43. Pahl C. Containerization and the PaaS Cloud. *IEEE Cloud Computing* 2015; 2(3): 24–31. doi: 10.1109/MCC.2015.51
44. Fang J, Ma A. IoT Application Modules Placement and Dynamic Task Processing in Edge-Cloud Computing. *IEEE Internet of Things Journal* 2020: 1. doi: 10.1109/JIOT.2020.3007751
45. Guo T, Gopalakrishnan V, Ramakrishnan KK, Shenoy P, Venkataramani A, Lee S. VMShadow: Optimizing the performance of latency-sensitive virtual desktops in distributed clouds. *Proceedings of the 5th ACM Multimedia Systems Conference, MMSys 2014* 2014: 103–114. doi: 10.1145/2557642.2557646
46. Shi T, Ma H, Chen G, Hartmann S. Location-Aware and Budget-Constrained Service Deployment for Composite Applications in Multi-Cloud Environment. *IEEE Transactions on Parallel and Distributed Systems* 2020; 31(8): 1954–1969. doi: 10.1109/TPDS.2020.2981306
47. Alicherry M, Lakshman TV. Network aware resource allocation in distributed clouds. In: ; 2012
48. Zhu J, Zheng Z, Zhou Y, Lyu MR. Scaling service-oriented applications into geo-distributed clouds. In: IEEE. ; 2013: 335–340
49. Docker Inc. . Docker Engine: Open-source Containerization Technology. 2018.
50. Ookla . The Definitive Source for Global Internet Metrics. 2018.
51. Szymaniak M, Pierre G, Van Steen M. Scalable cooperative latency estimation. In: ; 2004
52. Ha J, Park J, Han S, Kim M. Live Migration of Virtual Machines and Containers over Wide Area Networks with Distributed Mobility Management. In: MobiQuitous '18. ACM; 2018; New York, NY, USA: 264–273
53. Reber A. Container Migration Around The World. .
54. Ahmadian S, Friggstad Z, Swamy C. Local-Search Based Approximation Algorithms for Mobile Facility Location Problems. In: SODA '13. Society for Industrial and Applied Mathematics; 2013; USA: 1607–1621.
55. Friggstad Z, Salavatipour MR. Minimizing movement in mobile facility location problems. *ACM Transactions on Algorithms* 2011. doi: 10.1145/1978782.1978783

56. Demaine ED. Minimizing Movement. : 258–267.
57. Lavine BK. 3.20 - Feature Selection: Introduction. In: Brown SD, Tauler R, Walczak BBTCC., eds. *Comprehensive Chemometrics*Oxford: Elsevier. 2009 (pp. 601–607)
58. Mennes R, Spinnewyn B, Latré S, Botero JF. GRECO: A distributed genetic algorithm for reliable application placement in hybrid clouds. In: IEEE. ; 2016: 14–20.
59. Shi T, Ma H, Chen G. A Genetic-Based Approach to Location-Aware Cloud Service Brokering in Multi-Cloud Environment. In: ; 2019: 146–153
60. Dawn Thompson J. 3 - Statistical Alignment Approaches. In: Dawn Thompson J. , ed. *Statistics for Bioinformatics*Elsevier. 2016 (pp. 43–51)
61. Transaction Processing Performance Council . TPC-W: a transactional web e-commerce benchmark. 2018.

Author/s:
Aldwyan, Y;Sinnott, RO;Jayaputera, GT

Title:
Elastic deployment of container clusters across geographically distributed cloud data centers for web applications

Date:
2021-11-10

Citation:
Aldwyan, Y., Sinnott, R. O. & Jayaputera, G. T. (2021). Elastic deployment of container clusters across geographically distributed cloud data centers for web applications. CONCURRENCY AND COMPUTATION-PRACTICE & EXPERIENCE, 33 (21), https://doi.org/10.1002/cpe.6436.

Persistent Link:
http://hdl.handle.net/11343/298638