Intelligent Scaling of Container-based Web Applications in Geographically Distributed Clouds

by

Yasser Aldwyan
ORCID: 0000-0002-4918-2509

A thesis submitted in total fulfillment for the degree of Doctor of Philosophy

in the
School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

April 2021
Intelligent Scaling of Container-based Web Applications in Geographically Distributed Clouds

by Yasser Aldwyan

Principal Supervisor: Prof. Richard O. Sinnott
Co-Supervisor: Dr. Glenn T. Jayaputera

Abstract

Cloud data centers are increasingly distributed around the globe. Recently, containerisation, a lightweight virtualization technology, has been rapidly adopted as an application packaging mechanism for efficient, consistent web application deployment and scaling within and across Cloud-based data centers. To leverage Cloud elasticity and scalability, containers commonly run on elastic, scalable clusters of virtual machines (VMs). Such global infrastructure and lightweight deployment capabilities offer a perfect choice for deploying latency-sensitive web applications in multiple locations to serve globally distributed users. However, managing container-based web applications, including containers and VMs, in widely dispersed data centers currently lacks intelligent deployment and elasticity capabilities from Cloud providers.

This thesis investigates several problems related to the lack of such capabilities. This includes problems of deployment such as where and how to deploy VM clusters as well as geo-replicated application containers across data centers to address potential outages while considering wide-area network latency issues. It also considers how to dynamically deploy clusters across data centers to handle potential spatial workload fluctuations with minimum costs. This in turn gives rise to elasticity problems for multi-cluster container-based web applications deployed to multiple data centers. These problems include how to rapidly scale overloaded clusters at the VM level through temporary inter-cluster resource utilisation to avoid Cloud VM provisioning delays. Ideally this should provide sufficient VM resources for the timely launching of new containers in response to sudden workload spikes and avoid costly resource over-provisioning. A further challenge is how to control elastic scaling for both containers and VMs while considering application-level metrics and potential variations in container processing capacity, due to performance interference in shared Cloud data centers. Key to this is the need to optimise performance, availability and costs in a flexible and intelligent manner.

This thesis aims to enhance the state-of-the-art in the deployment and elasticity of container-based web applications in geographically distributed Cloud environments, by tackling the above-mentioned problems using meta-heuristics and queuing theory. The thesis makes the following key contributions:

1. it provides an approach for latency-aware failover deployment of container-based web applications across multiple Cloud-based data centers to maintain performance with associated SLOs under normal conditions and in the presence of failures;
2. it provides an approach for dynamic elastic deployment of container-based clusters, both in terms of the quantity and placement across data centers whilst offering trade offs between cost and performance in the context of geographic web workload changes;

3. it offers a cost-efficient, rapid auto-elastic scaling approach for bursty multi-cluster container-based web applications deployed across data centers that scales containers in overloaded situations in a timely and cost-efficient fashion;

4. it presents a two-level elasticity controller algorithm that seamlessly auto-scales at both the container and VM levels based on application-level metrics and queueing-based performance models through estimating the container capacity needed without violating SLOs;

5. it supports dynamic, inter-data center latency aware container scheduling policies for cross-data center clusters that are able to optimise the overall performance, and

6. it presents extensive experiments using case studies based on the container technologies Docker, Docker-Swarm and Kubernetes on the Australia-wide distributed Cloud computing environment (NeCTAR) and international (commercial) cloud data centres.
Declaration of Authorship

I, Yasser Aldwyan, declare that this thesis titled, ‘Intelligent Scaling of Container-based Web Applications in Geographically Distributed Clouds’ and the work presented in it are my own. I confirm that:

- The thesis comprises only my original work towards the PhD except where indicated in the preface;
- due acknowledgement has been made in the text to all other material used; and
- the thesis is fewer than the maximum word limit in length, exclusive of tables, maps, bibliographies and appendices as approved by the Research Higher Degrees Committee.

Signed:  Yasser Aldwyan

Date:  April 2021
Preface

This thesis research has been carried out in the School of Computing and Information Systems at The University of Melbourne. Chapters 3 to 5 of the thesis are based on the following published/submitted for publication articles:


The work in the thesis also resulted in another publication.

Acknowledgements

Throughout my rewarding yet challenging PhD journey, I have received a great deal of support and assistance. I owe my deepest gratitude to all who have helped me to reach the end of this journey.

First and foremost I am deeply grateful to my supervisor, Professor Richard O. Sinnott, whose expertise was invaluable in developing my academic and non-academic skills. His insightful feedback pushed me to sharpen my thinking and brought my work to a higher level. Thank you for always being there to help me throughout.

I would also like to thank my co-supervisor Dr. Glenn T. Jayaputera and the chair of my PhD committee Associate Professor Atif Ahmed for their insightful comments and advice.

In addition, I would like to thank the Australian National Research Cloud (NeCTAR) for the free, generous usage of their resources throughout my PhD study.

Most importantly, I would also like to express my sincere gratitude to my father and mother for their wise counsel and sympathetic ear. They have always been there for me.

I would also like to offer my special thanks to my wife Arwa. Without her support, I could not have finished this long-term journey. I would also like to thank my children Sumaia, Ammar and Mariah who provided countless yet happy distractions to rest my mind outside of my research.

Finally, my appreciation goes out to the rest of my family and friends for their encouragement and support throughout my studies.

Yasser Aldwyan

Melbourne, Australia
To my deceased grandmother Hosa, my father, my mother, my wife and children Sumaia, Ammar and Mariah.
# Contents

1 Introduction 1

1.1 Managing Applications in Distributed Clouds through Containers 3

1.1.1 Container-as-a-Service based Cloud Platforms 4

1.1.2 Cloud-based Clustering of Elastic Containers 5

1.1.3 Exemplar Distributed Hybrid Cloud-based Container Management Architecture 6

1.2 Challenges in Managing Containerised Web Applications in Distributed Cloud Environments 8

1.3 Research Question and Problems 11

1.4 Aim and Scope 13

1.5 Evaluation Methodology 13

1.6 Thesis Contribution 14

1.7 Thesis Structure 15

2 Background and Literature Review 19

2.1 Introduction 19

2.2 Web Applications: Requirements and Challenges 21

2.3 Application-centric Perspectives on Distributed Clouds 23

2.4 Container Orchestration and Cluster Management Platforms 25

2.5 Workload Management Approaches 28

2.6 Placement Challenges 32

2.7 Distributed Cloud Container Deployment Models and Platforms 37

2.8 Cloud Container Elasticity 42

2.9 Summary and Discussion 49

3 Latency-aware Failover Deployment for Containerised Web Applications in Distributed Clouds 51

3.1 Introduction 52

3.2 Background and Related Work 56

3.3 Application and Container Cluster-based Infrastructure Models 61

3.4 Problem Formulation 63

3.5 Proposed Approach 66

3.5.1 Requirements and Assumptions 66

3.5.2 The Overall Algorithm 67

3.5.3 Response Time and Violation Models 68

3.5.4 Genetic Algorithm (GA) for DC Selection 70

3.5.5 Microservice Deployment 73

3.6 Performance Evaluation 75

3.6.1 Experiment Set 1: Evaluation of our GA 76

3.6.2 Experiment Set 2: Evaluation of the Proposed Approach in Real Cloud Contexts 82
3.7 Conclusions and Future Directions ........................................ 89

4 Elastic Deployment of Container-based Web Applications across Distributed Clouds 91
  4.1 Introduction ........................................................................ 92
  4.2 Related Work ........................................................................ 96
  4.3 Problem Formulation ............................................................... 101
    4.3.1 Assumptions for Container and Application Deployment Models ...... 101
    4.3.2 Problem Definition .......................................................... 102
  4.4 Proposed Geo-elastic Deployment Approach ......................... 104
    4.4.1 Requirements and Assumptions ........................................ 104
    4.4.2 Adaptation Triggers ........................................................ 105
    4.4.3 Geo-elastic Deployment Framework .................................. 106
    4.4.4 SLO-based Violation Model ............................................. 109
    4.4.5 Multi-cluster Geo-elastic Deployment Controller Algorithm .......... 110
    4.4.6 Cluster Replacement Method for Spatial Adaptation .................. 113
    4.4.7 Genetic Algorithm for Cluster Replacement .......................... 118
  4.5 Performance Evaluation ......................................................... 122
    4.5.1 General Settings ............................................................ 123
    4.5.2 Experiment Set 1: Evaluation of the Proposed Geo-elastic Deployment Approach ......................................................... 126
    4.5.3 Experiment Set 2: Evaluation of the Proposed Geo-elastic deployment Solution in a Real Cloud Context ......................................................... 131
    4.5.4 Experiment Set 3: Evaluation of the Proposed Cluster Replacement Algorithm ......................................................... 137
  4.6 Conclusions and Future Directions ......................................... 140

5 Cost-oriented Scaling of Container-based Web Applications in Distributed Clouds 143
  5.1 Introduction ........................................................................ 144
  5.2 Multi-Cluster Container Deployment Model: Assumptions and Challenges ... 148
  5.3 The proposed approach ......................................................... 150
    5.3.1 System Architecture ....................................................... 150
    5.3.2 Overall Workflow .......................................................... 153
    5.3.3 Service and Cluster Conditions ........................................ 154
    5.3.4 Local Cluster Elasticity Management at Container and VM Levels ...... 156
    5.3.5 Cluster Overload Handling Algorithm via Inter-Cluster Resource Utilisation ......................................................... 160
    5.3.6 Container Scheduling Policies ........................................... 161
  5.4 Performance Evaluation ......................................................... 163
    5.4.1 Experimental Set-up ........................................................ 163
    5.4.2 Application Benchmarking ............................................... 165
    5.4.3 Workload ....................................................................... 165
    5.4.4 Benchmarks ................................................................. 166
    5.4.5 Evaluation of the Proposed Elasticity Approach ....................... 167
    5.4.6 Web Scheduler Performance ............................................ 171
    5.4.7 Container Capacity Model Analysis .................................... 173
  5.5 Related Work ....................................................................... 176
  5.6 Conclusions and Future Directions ......................................... 178

6 Conclusions and Future Directions ............................................. 181
  6.1 Summary and Conclusions .................................................... 181
6.2 Future Directions ................................................................. 185

6.2.1 Dynamic Distributed Load Balancing for Multi-Cluster Models for Container Deployment ........................................... 185

6.2.2 Multi-objective Cloud Evaluation for Container Cluster Placement ................................................................. 186

6.2.3 Efficient Global-scale Data Management for Dynamic, Fast Container-based Web Application Deployments ...................................................... 186

6.2.4 Inter-Cluster Resource Management for Cost-Efficient Scaling ................................................................. 187

6.2.5 Dynamic Placement Policies for Multi-Cluster-aware Container Orchestration ...................................................... 187

6.2.6 Dynamic Configuration of Container Disruptions by Orchestration Platforms ...................................................... 188
List of Figures

1.1 Web Application Deployment in Distributed Clouds .......................... 2
1.2 Exemplar Cloud-based Container Cluster Architecture based on a Single Cloud Data Center .................................................. 6
1.3 Distributed Cloud-based Single-cluster Container Deployment Model . 7
1.4 Exemplar Multi-cluster Model of Distribute Cloud-based Container Deployment ........................................................................... 8
1.5 Thesis Organisation ....................................................................... 16

2.1 Web application requirements and challenges in Cloud computing ...... 22
2.2 Deployment models in distributed Clouds ....................................... 24
2.3 Workload management approaches .............................................. 28
2.4 A classification of container deployment in distributed Clouds based on VM clustering approaches ................................................. 38
2.5 Container deployment models in distributed Clouds. (DC: Data Center. Data centers at each model can belong to either multiple zones, regions or Clouds.) .................................................................................. 39

3.1 Geographical container clustering of web applications across two geographical areas in distributed clouds ................................. 53
3.2 Geo-redundant model of a microservice based web application ........ 62
3.3 Servicing requests under normal and failover conditions ................. 64
3.4 HAP microservice-based application deployment in distributed clouds 67
3.5 A chromosome with 2 selected genes (DC pairs) ............................ 71
3.6 Convergence curves for proposed GA using various parameter settings 78
3.7 Performance comparisons of different algorithms using 2 primary-backup DC pairs and various workloads and SLOs under normal and failover conditions .............................................................. 79
3.8 Performance comparison of distributed deployments using various numbers of DC pairs and the optimal single DC pair deployment ...... 81
3.9 Performance comparison of 3 deployments for read-intensive workloads generated from 5 different locations around Australia. First half of each 600-second deployment is under normal conditions, while the second one is after a failover. (SLO=40 ms) ........................................... 85
3.10 Performance comparison of 3 deployments for write-intensive workloads generated from 5 different locations around Australia. (SLO=40 ms). .. 86
3.11 The 95th percentile of response time before and after a failover of 3 deployments using various workloads generated from 5 different locations around Australia. For each deployment, approximately 32,400 requests were generated from each location (total 162,000 requests). (SLO=40 ms)

4.1 Container-based multi-cluster deployment of web applications across geographically distributed clouds .................................................. 93

4.2 A framework of enabling automated elastic deployment of container clusters in geographically distributed Clouds .............................. 105

4.3 Geographic workload changes over time ....................................... 124

4.4 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. .................. 128

4.5 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. ................. 130

4.6 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) .................. 132

4.7 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%). 138

5.1 Kubernetes-based Multi-cluster Web Application Deployment across Geographically Distributed Cloud Data Centers .......................... 146

5.2 Proposed Architecture of the Prototype System with 3 Kubernetes Clusters Deployed at Geographically Distributed Cloud Data Centers .... 151

5.3 Real-world Workload Pattern (FIFA World Cup 1998) .................. 165

5.4 Performance comparisons of different elasticity solutions during overload situations for multi-cluster container-based web applications. .... 168

5.5 The performance of the Melbourne cluster with different elasticity approaches in response to dynamic workloads. ......................... 169

5.6 The 95th percentile of response times using two schedulers during a cluster overload condition (i.e. the first 300s) and after the cluster returns back to the normal condition (i.e. the last 200s). ....................... 172

5.7 Comparisons of container capacity estimated by the proposed queuing-based capacity models vs the empirically measured container capacity at different times of the day based on SLOs. AEDT: Australian Eastern Daylight Time. .......................... 174
List of Tables

2.1 A summary of work related to placement research challenges. . . . . . . 34
2.2 Summary of Container Management Platforms in Distributed Clouds. . 41
2.3 Summary of Container and VM Elasticity Work in Cloud Computing . 48

3.1 Summary of related work. ......................................................... 59
3.2 Terms used in the Models ......................................................... 66
3.3 Estimate and standard error (SE) of the average fitness for GA under normal and failover conditions over 35 runs as well as the number of generations for all runs to convergence using different GA parameter settings. The higher the fitness the better. ............................................. 77
3.4 95% confidence interval (CI), estimate and standard error (SE) of the mean of $SBVR$ for different algorithms under normal and failover conditions over 35 runs. The lower the $SBVR$ the better. ....................... 80
3.5 95% confidence interval (CI), estimate and p-value of the difference in means of $SBVR$ between different algorithms under normal and failover conditions. Each algorithm ran 35 times. ....................... 81
3.6 95% Confidence Intervals (CI) and estimates of the mean and standard deviation of response times before and after failover of different deployments for different workload locations. Here the workload type is read-intensive and measurements are in milliseconds. (SLO=40ms). .............. 88
3.7 95% Confidence Intervals (CI) and estimates of the mean and standard deviation of the response times before and after failover of different deployments for different workload locations. Here the workload type is write-intensive and measurements are in milliseconds (SLO=40ms). . . 89

4.1 Terms Used in the Models ......................................................... 103
4.2 Terms Used in Cluster Replacement Algorithm .............................. 119
4.3 Workload locations over time periods for the different experiments. . 124
4.4 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 ......................................................... 136
4.5 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. ....................... 137
4.6 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.

4.7 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.

5.1 Terms Used in The Paper

5.2 95% confidence interval (CI), estimate and p-value of the difference in means of the 95th percentile of response times using two schedulers. P: Proposed web scheduler. K: Kubernetes default scheduler.

5.3 The difference in container capacity between the empirical measurements and models’ estimations over times of the day and the accuracy of the models’ estimations.
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AKS</td>
<td>Amazon Kubernetes Engine</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>CaaS</td>
<td>Container as a Service</td>
</tr>
<tr>
<td>CDN</td>
<td>Content Delivery Networks</td>
</tr>
<tr>
<td>CI</td>
<td>Confidence Interval</td>
</tr>
<tr>
<td>DC</td>
<td>Data Center</td>
</tr>
<tr>
<td>DC/OS</td>
<td>Distributed Cloud Operating System</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name System</td>
</tr>
<tr>
<td>ECA</td>
<td>Amazon ECS Cluster Auto-scaling</td>
</tr>
<tr>
<td>EC2</td>
<td>Amazon Elastic Compute Service</td>
</tr>
<tr>
<td>EKS</td>
<td>Amazon Elastic Kubernetes Service</td>
</tr>
<tr>
<td>ECS</td>
<td>Amazon Elastic Container Service</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithm</td>
</tr>
<tr>
<td>GKE</td>
<td>Google Kubernetes Engine</td>
</tr>
<tr>
<td>GLB</td>
<td>Geographical Load Balancing</td>
</tr>
<tr>
<td>HA</td>
<td>High Availability</td>
</tr>
<tr>
<td>HAP</td>
<td>High Availability and Performance</td>
</tr>
<tr>
<td>HPA</td>
<td>Kubernetes Horizontal Pod Autoscaler</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure as a Service</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>KCA</td>
<td>Kubernetes Cluster Autoscaler</td>
</tr>
<tr>
<td>K8s</td>
<td>Kubernetes</td>
</tr>
<tr>
<td>LLB</td>
<td>Local Load Balancing</td>
</tr>
<tr>
<td>NeCTAR</td>
<td>National eResearch Collaboration Tools and Resources Research Cloud</td>
</tr>
<tr>
<td>Acronym</td>
<td>Term</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PaaS</td>
<td>Platform as a Service</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
</tr>
<tr>
<td>SE</td>
<td>Standard Error</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>SLO</td>
<td>Service Level Objective</td>
</tr>
<tr>
<td>TPC-W</td>
<td>Transaction Processing</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
</tbody>
</table>

*W = 10000 transactions per second*
Chapter 1

Introduction

The Cloud computing paradigm offers IT resources and infrastructure as services based on a pay-as-you-go model. These resources are hosted in data centers where users can remotely provision and release them using a self-service, on-demand provisioning model [1]. Gartner defines Cloud computing as "a style of computing in which scalable and elastic IT-enabled capabilities are delivered as a service using internet technologies" [2]. Since its emergence, Cloud computing has put forward a vision of an economic, outsourced model of computing. It enables organisations to eliminate the up-front investments and maintenance expenses of establishing and maintaining their own infrastructure [3]. Instead of acquiring and maintaining large scale infrastructure to run their applications, organisations can deploy their applications on-demand in a pay-as-you-go manner to Cloud infrastructures at major data centres that give the impression of infinite scaling possibilities.

Cloud computing offers dynamic computing capacity on demand for hosted applications. This allows application providers to maintain performance and minimise costs by varying their resource needs as workload changes. Planning for computing capacity under dynamic workloads is a challenging task as estimating workload fluctuations can be difficult. This can be especially problematic for web applications as their workloads can vary drastically at given time periods [4, 5]. Any gap between the provisioned computing capacity and the given demand at any time point can result in either inefficient, costly over-provisioning of resources or in the inability to fulfil user-experienced quality of service (QoS) due to under-provisioning of sufficient resources in times of load/demand. The scalability, elasticity and self-service provisioning of
Cloud resources should enable application providers to handle this issue by offering capabilities to automatically expand or reduce the computing capacity at any time, based on workload fluctuations to deliver QoS at lower cost.

Traditionally, Cloud computing services have been broadly classified into three main service models: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS). IaaS enables cloud customers to provision infrastructural resources such as nodes/servers (usually in the form of virtual machines (VMs)), storage and network. PaaS provides a higher level of abstraction compared to IaaS that enables developers to easily deploy and manage their applications in the Cloud without the overheads of managing the infrastructure by providing a development framework including the necessary runtime environments as well as management services, e.g., for auto-scaling. SaaS, the highest level of abstraction, provides on-demand applications in the Cloud. IaaS is not only the foundation for PaaS and SaaS, but is also regarded as the backbone of Cloud computing [6]. Cloud computing resources have historically been based on virtualisation of hardware through technologies such as hypervisors [7].

Cloud data centers are increasingly distributed around the globe. This global Cloud setting offers opportunities for web application providers to deploy their applications at different locations (see Figure 1.1) [8]. This distributed deployment can bring key
benefits such as geographical scalability and performance improvement to globally dis-
tributed users by reducing network latency as well as offering a high degree of avail-
ability against individual Cloud data center outages. A distributed Cloud deployment
is a deployment model where application services/components are deployed across at
least two geographically-dispersed Cloud data centers. These data centers can belong
to a single Cloud (multi-zone or multi-region deployments) or indeed across multiple
Clouds (multi-Cloud deployments). The Clouds themselves can be heterogeneous and
numerous major Cloud providers now exist, e.g. Amazon Web Services (AWS) [9],
Azure [10], Google [11] and many others.

1.1 Managing Applications in Distributed Clouds through Con-
tainers

Recently, container technologies have been put forward as the go-to solution for Cloud-
based deployments [12, 13]. Containers provide a lightweight operating system (OS)
form of virtualization. They support process isolation and multi-tenancy at the OS
level and provide lightweight deployment units known as containers. Docker [14] and
rkt [15] are two example of container technologies. Containers are lightweight because
their images do not require an OS to run them while the traditional, hardware virtu-
alization, e.g., to create VMs, requires an entire OS for each VM instance. As a result,
containers are smaller and often application-specific. They offer numerous benefits in-
cluding: efficient memory usage and reduced start-up overheads [16]. For instance, the
start-up times of containers take only few seconds [17] while the start-up times of VMs
can take from a minute to more than 10 minutes [18]. Like the more traditional VM
approach, e.g., using snapshots or scripted build/deployments, containers can be used
as an application packaging mechanism for deploying applications over computing in-
frastructures. However they provide a more rapid and efficient deployment approach
[16]. It is possible to launch hundreds of application containers in only a few seconds.
Hence, containers are increasingly selected for Cloud-based applications that require
rapid scalability, compared to VM-based solutions [19]. Additionally, containerisation
enables the management of resources at a finer grained level and thus they can offer
Introduction

better utilisation of resources [20]. Furthermore, the microservice paradigm, which al-

ows to break up monolithic applications into lightweight, independently deployable

services [21], has recently gained popularity [22]. Using containerisation and microser-

vices can increase scalability, availability and portability of applications [21, 22]

1.1.1 Container-as-a-Service based Cloud Platforms

In IaaS Clouds, developers deploy their applications onto VMs and this gives them

complete flexibility over the application infrastructure. However, the deployment and

management APIs of such VM-based deployment models are more machine-centric.

That is, developers are required to configure VM images directly, e.g., choosing the

right-sized VM type to run their applications. This machine-centric step can be difficult

for many developers [16]. In PaaS Clouds, this step is abstracted away from develop-

ers and hence they can focus on their code and leave the overheads of managing the

underlying infrastructure, usually VMs, to the middleware layer. PaaS application de-

ployment models have a number of downsides however as they provide less flexibility

to developers. PaaS not only abstracts the management overhead of resources but also

some developer-related application details such as the programming languages and

framework used. This leaves developers with restricted software development capa-

bilities, e.g., they have to use specific languages, software versions and libraries, and

include overheads to the middleware. PaaS application developers are also unable to

control all application components [23]. This model is also platform-dependent and

can lead to vendor lock-in and portability issues [24].

Container-as-a-Service (CaaS) is a new container-based Cloud service model that has

the potential to address the above-mentioned issues with IaaS/PaaS. It provides a sim-

ple, clean Cloud-based abstraction providing considerable flexibility to developers [16,

25]. Container images provide an abstraction that can isolate application environments

and their components from the underlying deployment infrastructure [26]. As such,

CaaS platforms can provide a clean separation of concerns so application developers

can focus on their code and control all application details and components while the

middleware layer (or IT operations teams) focus on the deployment and management
1.1 Managing Applications in Distributed Clouds through Containers

of containers and their infrastructure without heed to what code or application runtime environments are used inside the containers [27, 28]. Running containers on top of VMs has been adopted widely by all public Cloud providers especially as they can add an extra layer of security [28, 29]. CaaS application developers have no restrictions on programming languages or software versions. The container-based model makes the deployment and management application-centric [16, 26].

1.1.2 Cloud-based Clustering of Elastic Containers

Containers usually run on clusters of Cloud-based nodes/servers to leverage computing resources beyond the limits of a single machine. On top of these server clusters, container orchestration and cluster management systems such as Kubernetes [30], Docker Swarm [31] and Apache Mesos [32] are used for deploying, scaling and managing container-based applications and supporting inter-container communication. Clusters of servers are configured with daemons (agents) that deploy containers based on the given container runtime, e.g., Docker [14]. Cluster nodes can be either configured as master nodes, which maintain the cluster state and manage the control plane components, e.g., schedulers and API servers, or they can be configured as worker nodes that run containerised applications.

For elastic and scalable solutions, container-based clusters are typically deployed over Cloud computing infrastructures as illustrated in Figure 1.2. IaaS services are commonly used to provide these infrastructures with the underpinning resources (VMs) for hosting and running the clusters. To manage the deployment and elasticity of containers and VMs in Clouds, organisations can either develop solutions using open source container management solutions, e.g., Docker Swarm or Kubernetes, or offload these management responsibilities to managed CaaS platforms, e.g., use of Amazon Elastic Container Service [33] or Azure Kubernetes Service [34]. For web applications, adopting this container-based approach can provide fast and fine-grained scaling capabilities suited to bursty and fine-grained web workload fluctuations. This can improve performance and allow to optimise resource utilisation and minimise cost [35, 36].
1.1.3 Exemplar Distributed Hybrid Cloud-based Container Management Architecture

As noted, containers offer many advantages for managing distributed Cloud-based applications as they provide a lightweight application packaging approach that decouples applications from the underlying infrastructure on which they run [26]. As a result they have been widely adopted for cross-Cloud platforms to manage applications across (heterogeneous) distributed Cloud environments [37–39]. Adopting the container approach brings a number of benefits: it improves application portability and Cloud interoperability [40]; it provides consistent, fast orchestration of application services across multiple Clouds, and it supports geo-scaling applications beyond a single Cloud-based data center [41, 42].

To provide the necessary level of abstraction to developers in distributed Clouds, application providers need to manage containers and VMs at multiple, distributed data centers. To facilitate this, a variety of distributed Cloud-based container management
platform solutions have been developed. Based on the clustering models of container deployments across distributed Clouds, these can be classified into single-cluster and multi-cluster solutions. In the single-cluster model, as illustrated in Figure 1.3, the goal is to deploy a single cluster of geo-distributed nodes from different Cloud data centers, e.g., Google Kubernetes Engine [43] adopts this model. In the multi-cluster model, as shown in Figure 1.4, container management solutions deploy multiple clusters at individual data centers, e.g., Rancher [44] adopts this approach. However, these solutions are limited in their capabilities, e.g., managing QoS for applications. As web applications have QoS and other requirements such as cost-effectiveness, the management of containers and clusters for web applications in distributed Clouds imposes a number of challenges that require application providers to address. These are discussed below.
1.2 Challenges in Managing Containerised Web Applications in Distributed Cloud Environments

In this section, we discuss several challenges in managing container-based web applications in distributed Cloud environments. In this thesis, we assume that containers run on clusters of VMs. This is by far the most common deployment model of running containers in Clouds. Container deployment in distributed Clouds involves consideration of the following aspects: placement of VM clusters (e.g. data center selection); cluster infrastructure provisioning and container placement across these clusters to run containerised applications. Cloud container elasticity involves adapting container processing capacity by scaling the number of containers and adapting the number of underpinning server nodes, typically through establishing a clustered infrastructure. This scaling of containers can be supported by scaling at the VM level and subsequently scaling the number of containers on those VMs, or depending on the system load, varying
the number and placement of containers on existing VMs. These give rise to a range of challenges.

Challenges in Container Deployment

Challenge 1. **Network latency for high availability (HA) cross-data center deployments.** Most web applications require both high availability and performance (HAP). Deploying containers across multiple Cloud data centers can overcome the issues of potential Cloud data center outages and achieve HA since containerised applications can be deployed at different Cloud locations. However, arbitrary, latency-unaware deployments can be a bottleneck in application performance due to the global distribution of end-users and impact of potential wide-area network latencies between users and data centers and/or between inter-data center communication latency. These latencies can impact the response times of user requests which affect the QoS experienced by end-users.

Challenge 2. **The lack of intelligent and elastic deployment capabilities for handling geographically varying workloads.** Multi-cluster container deployments can help application providers to leverage the capabilities of global scale Cloud resources to elastically geo-scale web applications and hence manage localised workload variations to achieve particular QoS. Ideally any intelligent solution should help to minimise cost. However, as it is the application provider’s responsibility to make the deployment and adaptation plans, these are unlikely to be optimal without tool support and algorithms. Manual decisions can be naive and inefficient and lead to cost and performance issues, e.g., excessive deployment of clusters and/or placing clusters in data centers far away from users at any given time. This can incur latency issues for user requests and affect the overall QoS experienced.

Challenges in Container Elasticity

Challenge 3. **The lack of cost-efficient inter-cluster resource management to rapidly scale containers in overload situations with minimal overall resource costs.** With
Introduction

multi-cluster containerised web applications, some clusters can become overloaded suddenly due to unexpected bursts in application load, e.g., user requests. Basic Cloud elasticity solutions are not able to manage such sudden overloads in a timely manner because they introduce launch latency issues for newly created pending containers. These containers cannot be launched in a timely manner before the often considerable time required to provision new VMs. This can negatively affect application performance and violate service level objectives (SLOs). Other solutions allow launching containers immediately in response to sudden increased traffic, however these are often at the cost of running additional, idle VMs for each cluster. Inter-cluster resource sharing should ideally optimise launch times to maintain QoS and SLOs with minimal underpinning resources.

Challenge 4. Failure to consider high-level application-related metrics to efficiently scale at both container and VM levels. Basic elasticity approaches that rely solely on resource-level metrics such as CPU utilisation, are inadequate for container-based application elasticity as they are unaware of the actual processing capacity of given containers, e.g., the number of requests which the given web application can process without violating QoS. Due to the high potential for performance interference of VMs sharing the same physical server in multi-tenant, shared Cloud infrastructure [8, 45], the processing capabilities of containers can vary over time. This impacts on the accuracy in estimating not only the number of containers but also the number of VMs required for the cluster. This can lead to under- or over-provisioning of resources. This in turn gives rise to issues with application performance, cost and/or resource utilisation issues. Hence, performance models need to consider such variations to accurately estimate container capacity. Furthermore, elasticity techniques that consider such models and other application-related metrics, e.g., request arrival rates, are required to efficiently adapt the number of containers and potentially the number of VMs needed to run these containers.
1.3 Research Question and Problems

The focus of this research is to explore the management of containers and VMs of containerised web applications in distributed clouds with particular focus on deployment and elasticity aspects. This gives rise to the following research question:

How to efficiently deploy container-based web applications in geographically distributed Cloud computing environments and elastically scale applications to cope with dynamic workload whilst considering QoS requirements such as:

• Network latency: deployment of applications should consider the proximity to potentially globally distributed users and any associated inter-data center latencies that may arise;

• High availability: end users should be continually served even if unexpected situations arise, e.g., partial or complete (isolated) Cloud outages take place;

• Cost efficiency: deploying applications in Cloud environments requires continuous cost considerations to avoid expensive and potentially unnecessary over-provisioning;

• Service level objective (SLO\(^1\)) recovery: SLO violations should be detected as early as possible and addressed in a timely fashion to ensure minimal impact to end users.

To address these issues, we explore the following research problems.

Research Problem 1. How can we intelligently deploy containers and VM clusters across Cloud data centers to provide HA and address network latency issues to achieve HAP against pre-agreed SLOs. This in turn gives rise to several related questions:

• How should Cloud data centers used for hosting containerised web applications be selected considering factors such as the proximity to user workloads to minimise network latency between users and data centers, even in the presence of partial or complete data center outages?

---

\(^1\)SLOs are measurable elements that form part of service level agreements (SLAs) that form the basis for contracts between the service provider and the end user.
• How should different Cloud data centers used for hosting backup replicas of containerised web applications be selected so that they are far enough from primary replicas to achieve high availability whilst minimising inter-data center latency overheads?

Research Problem 2. How can we support automatic adaptation of multi-cluster container deployments across distributed Cloud data centers. Such adaptation should make intelligent deployment decisions to handle spatial workload fluctuations, whilst maintaining performance and SLOs at lower costs. This gives rise to the following challenges that need to be addressed:

• at any point in time, how many clusters are required and where should they be placed, e.g., should existing clusters should be removed/replaced?

• estimating the cost of adaptation for potential deployments and accordingly choosing the one with the minimum costs, whilst not overly sacrificing the QoS?

Research Problem 3. How to scale an overloaded cluster of VMs in a timely manner for multi-cluster container-based web applications through temporary inter-cluster resource utilisation to avoid (inevitable) Cloud VM provisioning delays. This should provide enough resources for rapidly launching new containers in response to sudden workload bursts at some clusters in a cost-effective way. This in turn gives rise to a number of related challenges.

• How to coordinate clusters so that they can offer/request resources from each other to provide a global shared computing capacity that can intelligently handle overload situations with minimum costs?

• How to auto-scale clusters at both container and VM levels while considering application-level metrics and potential variations in container processing capacity to optimise performance and costs.

• How to dynamically schedule containers across multiple data centres whilst considering inter-data center network latency and web application needs?
1.4 Aim and Scope

The aim of this research is to address the research problems discussed in the previous section and propose appropriate approaches, algorithms and mechanisms for deploying and elastically scaling containers and VMs hosting containerised web applications in geographically distributed clouds under dynamic workloads. The ultimate aim is to improve end-to-end response time, availability, reduce cost and ensure the delivered QoS experienced by end users of web applications deployed to the Cloud meets SLOs.

This thesis focuses on the deployment and elasticity aspects of container-based web applications in distributed-Cloud container management platforms. We also consider (dynamic) container scheduling within/across data centers to make container-aware solutions. Such consideration avoids container disruptions when taking elasticity actions at the VM level, e.g., scaling in the VM cluster, as well as to fit web application needs. In undertaking this we make several assumptions. Firstly we assume that container orchestration systems such as Kubernetes [30] are deployed and available on Cloud platforms. Our scope is limited to container platforms in IaaS Clouds. That is, we consider IaaS clouds as a source of compute (VMs), storage and networking resources that can be utilised to deliver the underlying application infrastructure. We also focus on global-scale web applications and use of web servers. The challenges of scaling and distributing data across servers and data centers are beyond the scope of this thesis. Finally we assume that data that might be associated with web applications is already distributed across the Cloud data centers.

1.5 Evaluation Methodology

We conduct experiments on the Australia-wide National eResearch Collaboration Tools and Resources (NeCTAR) Research Cloud [46]. The NeCTAR Cloud is based on multiple geo-spatially distributed data centers across Australia. We also utilise resources from Amazon and Google Clouds for the global evaluations. With regards to container technologies, we used Docker as the container runtime and Kubernetes [30] and Docker
Swarm [31] as container orchestration and management platforms. For web application benchmarking, we use the real-world transactional web benchmark (TPC-W) [47] as well as an exemplar sock shop application that is realised as a microservice-based e-commerce application [48].

1.6 Thesis Contribution

This thesis covers many areas and makes several contributions. These include:

1. An approach for generating HAP container-based cluster deployment plans to maintain web application performance with associated SLOs under normal conditions and in situations where failures have occurred. This includes:
   - models to estimate SLO-based violation and response times before and after failures have occurred;
   - a cluster node (server) placement algorithm based on genetic algorithms for selecting primary-backup pairs of data centers where each pair runs microservices (at a primary data center) and their backups (at a backup data center) and serves users at a different areas/locations;
   - HA-aware container-based microservice deployment algorithms for generating the deployment configuration of primary and backup microservice container replicas across given servers/nodes in specific data centers, and
   - extensive experiments using case studies based on Docker and Docker-Swarm on Australia-wide distributed Cloud computing environments to demonstrate the ability of the approach to maintain application performance before and after failovers.

2. An elastic, dynamic multi-cluster container deployment approach for web applications that trades off cost and performance to maintain QoS and SLOs as well as maintaining performance during adaptation times.
   - An elastic deployment technique that dynamically makes intelligent deployment plans to optimise the number of clusters and their placement. Genetic
algorithms are applied for cluster placement and a heuristic is introduced for adjusting the quantity of clusters.

• A framework to enable automated elastic multi-cluster container deployments.

• Extensive experiments on mainstream distributed Cloud computing environments using Docker and Kubernetes to demonstrate how sacrificing minor, acceptable performance levels can result in significant cost reductions.

3. A cost-oriented container-based elasticity approach to rapidly handle sudden overload situations for bursty multi-cluster containerised web applications. This includes:

• an architectural framework that extends the capabilities of Kubernetes clusters to support inter-cluster resource management;

• queuing-based performance models to estimate container capacity to process requests without violating SLOs;

• two-level cluster elasticity techniques that can horizontally autoscale containers and VMs based on SLO-based container capacity and high-level metrics;

• dynamic container scheduling policies that consider inter-data center placement demands to fit web application needs, and

• extensive experiments on mainstream distributed Cloud computing environments across Australia and Europe using the container technologies Docker and Kubernetes.

1.7 Thesis Structure

The thesis structure is depicted in Figure 1.5. Chapter 2 presents the necessary background and the related literature. Chapters 3 and 4 target the challenges in container deployment for web applications in distributed Clouds. Chapter 5 focuses on container elasticity for web applications in Clouds. Specifically:
• Chapter 2 presents the necessary background on web applications in Clouds, distributed Clouds and container management platforms. It also presents related research on web application management, container deployment and elasticity in distributed Cloud environments.

• Chapter 3 proposes an approach that deploys clusters and containerised microservices across Cloud data centers for web applications requiring both high availability and performance whilst supporting SLOs under normal and failover conditions. This chapter is derived from:

- Chapter 4 describes an approach to provide elastic multi-cluster deployments in distributed Clouds to elastically scale containerised web applications to address spatial workload variation of web applications whilst maintaining performance and SLOs at lower costs. This chapter is derived from:


- Chapter 5 presents a cost-efficient elastic scaling approach for bursty multi-cluster containerised web applications that handles cluster overloads in a timely manner using inter-cluster resource utilisation. The approach optimises the size of clusters at both container and VM levels whilst meeting performance, SLOs and overall resource cost. This chapter is derived from:


- Chapter 6 summarises the key findings of the thesis and provides an overview of further research challenges that could form the basis for potential future work.
Chapter 2

Background and Literature Review

This chapter covers the related research context representing the current state of the art in managing applications in distributed Cloud environments. It provides the research background on Cloud-based web applications, distributed Clouds and container management platforms. It also covers research areas such as workload management, container placement, deployment and automatic scaling and management of container-based applications.

2.1 Introduction

Cloud computing has become the go-to deployment option for application providers as it promises unlimited, on-demand computational capabilities in a cost-effective manner. Clouds are now globally distributed and application developers/providers can now distribute their applications across multiple Cloud locations and use different Cloud providers and infrastructures. This distribution possibility can provide better performance, higher availability and meet a range of objectives as might be specified in service level agreements (SLAs).

Containers provide a lightweight virtualisation environment. They have recently become a de facto approach in managing applications in distributed Clouds. They can be used to support the development, deployment and scaling of applications across the Cloud. However, a number of challenges remain open in managing container-based applications in distributed Clouds. The management of container deployment and elasticity aspects for web applications in distributed Clouds is one area that has numerous open research challenges that need to be addressed.
In this thesis, we assume that containers are run on clusters of Cloud-based VMs - this is a widely adopted container deployment model. Thus, container deployment in the distributed Cloud context is an activity involving the following tasks: placement (i.e., the selection of the appropriate Cloud data centres for hosting container-based applications); provisioning of resources across the selected data centers to provide the underpinning container infrastructure (i.e., creation of VM clusters) and the container deployment scheduling and placement across those resources. In this context, elasticity involves adapting the processing capacity to support dynamic workloads at run-time so that the applications acquire enough resources to accomplish their tasks and achieve quality of service (QoS) at the lowest possible cost (from an application provider perspective). This adaptation process can take place at the container level, at the VM level or both. The more resources that are used results in increased overall costs.

In this chapter, we identify Cloud-based web application challenges and investigate deployment models of distributed Clouds. We review the current state of the art in managing applications in distributed Clouds and other related models using container-based solutions.

The rest of the chapter is organised as follows. In Section 2.2 we explore research into web application requirements and identify related research challenges in the context of the Cloud. We then provide a definition of distributed Clouds and discuss distributed-Cloud deployment models in Section 2.3. Section 2.4 presents an overview of several popular container orchestration platforms. In Section 2.5, we identify and discuss existing workload management approaches. Following this, we discuss current research in container placement in Section 2.6. Section 2.7 classifies container deployments in distributed Clouds into models based on clustering approaches and discusses several projects that fit each model. In Section 2.8 we discuss container elasticity in Clouds. Finally, Section 2.9 provides a summary and discussion of the research that underpins the contributions made in this thesis.
2.2 Web Applications: Requirements and Challenges

Requirements

A key goal of organisations relying on web applications is to offer responsiveness and availability, as even slight degradation in performance, availability or reliability can have a significant impact on business goals [49]. In addition to QoS, cost considerations are often important to avoid unnecessary and costly over-provisioning of compute resources to increase economic benefits of organisations and application providers [50]. As such, performance, availability, reliability, and cost are key requirements to many modern web applications. Other requirements such as security and operational excellence are crucial; however, they are beyond the scope of this thesis.

Web applications typically include user-facing web interfaces where their workloads are request-based and their associated response time is a key performance metric. Most web applications, especially those used in e-commerce sites, aim to achieve minimal delayed responses as such delays are likely to lead to lost revenue [51]. For instance, according to Forrester [52], 40% of customers leave online e-commerce sites when loading a page takes more than 3 seconds. Similarly, unavailability of applications for a single one hour can cost millions of dollars in lost revenue [49]. Maintaining such QoS requirements helps application providers avoid/minimise violations of service level objectives (SLOs) expressed as part of service level agreements (SLA) which can incur penalty costs for such violations.

Challenges

As shown in Figure 2.1, we identify a number of challenges that can affect web applications. Those include network latency, Cloud related challenges (e.g., Cloud outages, resource failures and data center overloads) and web workload behaviour. The performance can be impacted by the physical limitations and latency that can arise in wide-area, Internet-scale networks. This can introduce delays in response times directly affecting user experiences [53]. In term of Cloud-related challenges, Cloud outages and
resource failures (e.g., VM crashes) can obviously affect application availability. Even though availability is considered as a key characteristic in Cloud computing, it is, at the same time, one of its main challenges [54]. Cloud data centers may also experience overloading that can affect application requirements. Some Clouds may exceed the capacity limitations of their infrastructure. Such violations can cause delays in the processing times of user requests leading to user-experienced performance issues [54]. In severe situations, this can result in the unavailability of applications [54].

Another challenge that affects application requirements is the dynamic and potentially bursty workload behaviour experienced by web applications. The global distribution of the user base is a core characteristic of modern web applications [8], i.e., they can be invoked from anywhere. This can introduce performance issues associated with the geographical distance between users and data centers running applications. This can further increase network latency. The dynamic nature of web workload has two dimensions: geographic (spatial) and volume workload variations. Spatial workload variations can affect performance over time as new workloads can stem from potentially remote locations. Moreover, fluctuations in workload volumes can lead to performance
or cost issues. High loads can cause over-utilisation of resources which degrades performance while low workloads can introduce cost issues and lead to over-provisioning of excessive resources [5]. Workload variation can have a temporal factors that can be divided into long-term and short-term challenges, e.g., flash crowds. Long-term workload fluctuations can occur based on daily, monthly or seasonal rates, e.g. local events. In such situations, it may be possible to predict the load and plan accordingly. Flash crowds, on the other hand, are bursty and highly variable within short timescales. These can lead to inaccurate resource predictions and web applications under severe load with little time to plan to mitigate against the load [5].

2.3 Application-centric Perspectives on Distributed Clouds

Definition

Whilst “distributed Cloud” terminology has been used in the literature, e.g., [55–57], it has no clear definition [58]. Gartner provides a clear definition of distributed Cloud. It defines a distributed Cloud as “the distribution of public Cloud services to different physical locations, while the operation, governance, updates and evolution of the services are the responsibility of the originating public Cloud providers” [59]. According to StackPath [60], a distributed Cloud is defined as “an execution environment where application components are placed at geographically-dispersed locations chosen to meet the requirements of specific applications. We extend these definitions of a distributed Cloud to potentially go beyond a single Cloud. Specifically, we define a distributed Cloud as “a Cloud model that offers a choice of multiple geographically-distributed data centers, which belong to either a single Cloud or multiple Clouds upon which applications may be deployed”.

Deployment Models of Distributed Clouds

Distributed Clouds can be classified into homogeneous and/or heterogeneous models. In homogeneous distributed Cloud models, application providers (or brokers) use
Distributed Cloud deployments have a number of benefits. They can reduce network latency [56, 65], offer higher availability [66] and support global-scale applications [67]. Single-Cloud deployment models can also provide a variety of built-in, Cloud-dependent services and tools, e.g., floating IP addresses, geo-replication capabilities, cross-zone load balancing. These encourage and facilitate the adoption of multi-zone/region deployments to help improve application availability as well as QoS [68]. In terms of...
availability, multi-zone deployments can overcome individual zone failures whereas multi-region deployments provide a higher level of availability as they can tackle region-wide outages, e.g. Azure Cloud provides a multi-region deployment option of its Azure SQL database service [69]. Multi-region deployments can help improve performance as they allow applications to be geographically distributed. However, such single-Cloud deployment models can suffer from Cloud outages and lead to vendor lock-in issues. Multi-Cloud deployment models overcome Cloud outages and avoid vendor lock-in and hence enhance availability and flexibility. They can also improve performance as they provide application providers with more locations to choose to deploy their applications, e.g. to deploy the applications closer to the end users. Additionally, they can be cost-effective as they allow application providers to move to cheaper Clouds. Adopting this deployment model comes at a cost however, as it introduces Cloud interoperability and application portability issues. It can also increases the management complexity of applications and the underpinning resources offered by different Cloud providers [40, 68, 70].

2.4 Container Orchestration and Cluster Management Platforms

Containers have been widely adopted by organisations to deploy different types of modern applications including web applications, internet of things-based applications through to big data applications to Cloud infrastructures [71]. This has given rise to the development of container orchestration and cluster management platforms. Nowadays, such platforms play a central role in managing container-based applications and resources when deployed across large-scale clusters. They automate the deployment and scaling of containers across clusters of nodes and facilitate inter-container communication across the clusters. Clusters themselves can consist of physical or virtual machines. In this section, we discuss three popular container orchestration platforms: Kubernetes [30], Docker Swarm [31] and Apache Mesos [32].
Kubernetes

Kubernetes [30] is an open-source container orchestration and cluster management platform for automating the deployment, management and scaling of containerised applications. It was originally developed by Google and has become a de facto standard for container orchestration. Kubernetes introduces the notion of a pod. A pod is the smallest deployment unit in Kubernetes. It consists of one or more containers that are tightly coupled, share resources, and run on the same machine. A single pod is meant to run a single instance of an application. Hence, scaling application can occur by adding more pods. Kubernetes supports different types of containerised applications including long-running web applications through to batch jobs. It supports a decentralised scheduler architecture which can have multiple replicas of distributed schedulers where each replica handles a subset of deployment requests to achieve scalability. This allows custom schedulers and supports multi-zone deployments. Kubernetes also provides self-healing features to cope with container and node failures. It offers health checks and can restart containers that fail or kill the ones that do not respond as well as reschedule containers at healthy nodes in the presence of node failures. Manual and automatic (elastic) scaling are supported by Kubernetes. Kubernetes natively supports Docker [14] and rkt [15] runtimes. It can also support other container technologies such as containerd [72].

Docker Swarm

Docker Swarm [31] is an open-source native orchestration and cluster manager developed specifically for Docker containers. It provides a simple, lightweight and easy to use container management solution. Regular Docker commands can be used to deploy application services across clusters of nodes. An application is represented as a service. A task is the atomic unit of scheduling and a container is the instantiation of a task. Application services can have one or more tasks (i.e. containers with the same image). Swarm is intended to run long-running applications like web applications. As with Kubernetes, the Docker Swarm scheduler is decentralised. In term of availability,
a scheduling strategy (e.g., spread) can be used to spread multiple containers of a single service across clusters of nodes to improve availability. However, Swarm does not detect or handle service or node failures at runtime. However, cluster elasticity can be scaled manually.

**Apache Mesos**

Apache Mesos [32] is an open-source cluster management platform for containerised and non-containerised applications. It was developed at the University of California, Berkeley. It supports a two-level offer-base scheduling architecture. Mesos is responsible for the bottom level while the other level is handled by one or more application frameworks. Examples of application frameworks include Aurora [73] and Marathon [74] for long-running services, Hadoop and Spark for big data processing and Jenkins and Chronos for batch processing. Mesos manages and offers resources to application frameworks. These frameworks can accept resource offers and schedule their applications to allocated resources based on their specific needs. Thus, Mesos gives more flexibility to application developers as it allows them to utilise different application frameworks. Two well know frameworks, which act as container orchestrators on top of Mesos cluster are Aurora [73], which was originally developed by Twitter, and Marathon [74]. They provide applications with more availability as they handle node failures by rescheduling containers on healthy nodes. Even though these frameworks are very similar, Marathon is considered to be simpler to set up and easier to use [75].

It is noted that these container orchestration platforms are not Cloud-aware. They need solutions/platforms to automate their deployment in Clouds as well as deploy/scale/manage any associated cluster infrastructure, i.e., they do not specify or manage Cloud-platform-level needs or deployment processes [76]. Many platforms have been introduced by Cloud providers, open-source communities, or external third-party service providers to bridge this gap. Some Cloud-based platforms are discussed in Section 2.7.
2.5 Workload Management Approaches

Significant work has explored workload management problems using a variety of approaches and tackling different challenges based on meeting diverse objectives in different environments, e.g., Cloud, Edge and Fog computing. In this section, we present a classification taxonomy of workload management approaches. As shown in Figure 2.3, these approaches comprise: local and global load balancing, auto-elasticity, Cloud bursting, admission control and resource borrowing. We discuss each approach and provide solutions that fit each classification. It is noted that some solutions use more than one approach to manage workloads to meet their needs.

Load Balancing

A common approach for managing workload is through load balancing. This distributes workloads across different application replicas/compute resources based on some logic to achieve specific goals. Load balancing solutions can be local (i.e. replicas/resources are in one location) or geographical (i.e. replicas/resources are in multiple
distinct geographical locations). For local load balancing (LLB), most public Clouds provide local load balancing solutions to balance loads across a number of virtual machines running application replicas on the same Cloud to improve application performance and availability. Azure [77] and Google load balancers [78] are two such examples. Other works such as [79] aim at improving resources utilisation and minimising the makespan of tasks for IaaS Clouds. Other works such as [80] balance data distributions in Cloud data centers to improve the performance of data-intensive applications, e.g., data mining applications.

Geographical Load Balancing

Geographical load balancing (GLB) has become popular for tackling the problem of spatial workload management. It exploits the geographical distribution of resources in different distributed computing environments. A common type of GLB is the domain name system (DNS)-based GLB. DNS-based GLB solutions such as Google Cloud DNS [81] and Azure Traffic Manager [82], distribute workloads across different Cloud data centers according to the geo-locations of users to reduce latency. GLB also facilitates fault isolation and increased resilience to improve availability and other factors, e.g., energy savings [83]. In [84], the authors propose a content-aware DNS-based load balancing algorithm with a cache mechanism for geographically distributed web servers to store the most used content at each location to improve application performance. Another type of GLB is a centralised GLB where all incoming requests are gathered in one location running the GLB and then distributed to appropriate data centers according to one or more factors, e.g., energy costs and carbon footprint [85, 86] or regulatory requirements and latency [87]. This approach can however add extra latency to every request thus limiting the benefits of distributing applications to multiple geographical locations.

Another type of GLB is decentralised agent-based GLB. This approach can overcome issues introduced by a centralised GLB as different replicas (agents) of GLB can be located at different locations and they can decide whether to distribute traffic locally and/or coordinate with each other to distribute extra loads between different locations
to maintain application needs or at least mitigate the impact of extra loads on SLOs to meet application specific QoS. In [88], the authors propose a decentralised GLB solution that manages short-term spatial workload variations for web applications in multi-Cloud environments. In the Edge computing area, the authors in [89] provide a decentralised GLB solution for internet of things-type applications. They assume multiple clusters of edge nodes where each cluster has an orchestrator/coordinator that can distributed workloads locally across local nodes or globally across other remote clusters. This approach is designed to handle short-term spatial workload variations.

**Auto-Elasticity**

Auto-elasticity, or auto-scaling, is another workload management approach. It allows automatically expanding and shrinking the amount of compute resources needed for applications based on dynamic workloads. It aims to avoid over- and under-provisioning situations to maintain QoS, achieve performance, minimise cost and improve the overall resource utilisation. Over-provisioning can lead to extra economic cost and resource wastage while under-provisioning can cause performance and availability issues and lead to violation of SLAs. Some solutions take scaling decisions reactively based on pre-defined (static) thresholds [90], dynamic thresholds [35] or after detecting high load situations [91, 92]. Other solutions proactively predict future workloads and adapt the number of resources in advance [93–101]. Overall, this approach can help manage workload volume variations within data centers. Container elasticity approaches will be discussed in more details in Section 2.8.

**Cloud Bursting in Hybrid Cloud**

Another workload management approach is Cloud bursting in hybrid Cloud environments. Cloud bursting is the process of expanding compute resources in private Clouds by bursting into and using public Clouds when private resources are not sufficient to manage current workloads due to either spikes in workloads or infrastructure issues, e.g., partial resource failures. This approach can be cost-effective as it exploits local
resources. Many solutions adopting this approach have been proposed to manage applications and resources in hybrid Cloud environments [102–109].

Admission Control

Another approach to manage workload is to enforce admission control to guarantee specific goals, e.g., performance [110–114], high availability [115] or regulatory compliance [87]. Work such as [111, 114] put forward adaptive admission control mechanisms that admit incoming requests adaptively based on the current workload and compute capacity. They are aimed at preventing or at least mitigating the impact of application overloads due to flash crowds or resource failures. Such overloads can lead to performance degradation in applications or indeed server failures resulting in violation of SLOs and QoS. Work in [87] uses admission control to select appropriate data centers for users according to regulatory requirements.

Resource Borrowing

Resource borrowing is another workload management approach. The core idea behind this approach is to enable overloaded services/components to manage overloads by immediately borrowing idle resources from other idle services/components or resources that can be temporarily shut down [45, 116–118]. In [116], the authors present a solution for managing overload periods. They assume data centers are multi-tiered, where resources can be borrowed from other underutilised tiers until additional computing resources becomes ready to use by those bottleneck tiers. In [117], a self-adaptation programming paradigm is presented for managing unexpected high loads by dynamically deactivating features to allow core application services to borrow resources. Similarly, in [45] an integrated solution is proposed for managing high loads as well as tackling the energy consumption in container-based data centers. The approach is based on temporarily shutting down non-essential application containers to free their resources to overloaded services to add more containers and hence reduce the energy consumption of data centers.
Geo-Distributed Deployment

Another approach for managing workload is through geo-distributed deployment. This is defined as the process of geographically distributing the deployment of applications/resources in distributed computing environments. Examples of this approach include Fog computing [119, 120], content delivery networks [121–123] and Cloud platforms [8, 124–126]. These approaches are used to manage workloads, considering factors such as network latency [8, 119–126] and availability [127]. Solutions based on this approach can be static [121–125] or dynamic [8, 119, 120, 126]. While the former can only handle geographical workload distribution only, the latter can handle both geographical workload distribution and workload variation over time. In this approach, placement plays a central role to achieve/maintain application goals as discussed in the following section.

2.6 Placement Challenges

As Cloud data centers are increasingly geographically distributed, application providers have the opportunity to deploy their application replicas at multiple locations to meet their own specific goals. However, they still need to cope with the problem of when and where to place application replicas and the information required to make such decisions. Placement can be a challenging task and improper placement can impact on the intended benefits that might arise from the distributed deployment of applications. Considerations to be used when making placement decisions may often vary, e.g., based on availability or cost. In the following section, we identify several challenges for the placement of web applications.

Challenges

Latency Management. As network latency is a key performance factor for web applications, deploying application replicas at multiple data centers close to users can minimise the latency and thus improve application performance. This however requires
application providers to identify the location of users and data centers when selecting data centers to deploy application replicas.

**High Availability and Performance.** Application providers can use multiple Clouds to overcome potential (individual) Cloud outages and thus improve application availability. Multi-Cloud failover deployment plans can be utilised in this regard. This can in turn require consideration of geo-replication overheads which may incur extra network latency and negatively affect application performance. Therefore, in addition to proximity to users, application providers need to be aware of locations of data centers when selecting the most suitable data centers for placing primary and backup replicas to minimise latency as much as possible. This includes consideration of the proximity to users from data centers running backup replicas to maintain application performance even under partial failover conditions.

**Operating Cost.** Application deployment at multiple Cloud locations can be costly. Limiting the number of data centers for hosting applications is thus likely to be essential. Application providers need to determine the appropriate number of locations used to trade-off between cost, availability and the overall performance.

**Dynamic Deployment.** Application providers may need to adapt the deployment of applications across data centers for a variety of reasons. One reason can be to maintain performance based on the evolving location of new users, e.g., due to changes in application popularity over time. Economic benefits can be another reason, e.g., application providers may wish to utilise cheaper Clouds or remove idle application replicas from some data centers. This may also be for other reasons, e.g. regulatory reasons.

Application providers may need to consider the cost of adaptations from different perspectives. For example, it may be necessary to minimise the number of cross-data center application relocations as this process can be difficult and costly, e.g. in terms of moving application data. Application providers also need to be aware of the locations of data centers when selecting source and destination data centers and their evolving pricing costs. Such awareness can help reduce the impact of network latency between data centers during any adaptation processes for web applications especially those where geo-replication needs to maintain the state of the application.
Table 2.1: A summary of work related to placement research challenges.

<table>
<thead>
<tr>
<th>Work</th>
<th>Objective</th>
<th>Algorithm</th>
<th>Computing Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>[122]</td>
<td>Minimise total latency</td>
<td>Heuristics</td>
<td>CDN</td>
</tr>
<tr>
<td>[126]</td>
<td>Minimise total latency</td>
<td>Dynamic programming</td>
<td>CDN</td>
</tr>
<tr>
<td>[129]</td>
<td>Minimise communication cost</td>
<td>Dynamic programming</td>
<td>CDN</td>
</tr>
<tr>
<td>[130]</td>
<td>Minimise total replication cost</td>
<td>Heuristics</td>
<td>CDN</td>
</tr>
<tr>
<td>[119]</td>
<td>Minimise service latency</td>
<td>Heuristics</td>
<td>Fog</td>
</tr>
<tr>
<td>[120]</td>
<td>Minimise service latency</td>
<td>Heuristics</td>
<td>Fog</td>
</tr>
<tr>
<td>[131]</td>
<td>Minimise task latency/power consumption</td>
<td>Meta-heuristics (PSW)</td>
<td>Edge-Cloud</td>
</tr>
<tr>
<td>[132]</td>
<td>Minimise latency</td>
<td>Greedy algorithm</td>
<td>Cloud</td>
</tr>
<tr>
<td>[8]</td>
<td>Minimise total latency/operating cost</td>
<td>Iterative greedy algorithm</td>
<td>Cloud</td>
</tr>
<tr>
<td>[126]</td>
<td>Minimise overall latency/strong data consistency overhead</td>
<td>Meta-heuristics (GA)</td>
<td>Cloud</td>
</tr>
<tr>
<td>[127]</td>
<td>Improve availability/resource utilisation</td>
<td>Meta-heuristics (GA)</td>
<td>Cloud</td>
</tr>
<tr>
<td>[133]</td>
<td>Improve availability/resource utilisation</td>
<td>Subgraph isomorphism detection</td>
<td>Cloud</td>
</tr>
<tr>
<td>[134]</td>
<td>Minimise overall inter-data center latency</td>
<td>approximation algorithms</td>
<td>Cloud</td>
</tr>
<tr>
<td>[135]</td>
<td>Minimise traffic of services/operating cost</td>
<td>Meta-heuristics (GA)</td>
<td>Cloud</td>
</tr>
<tr>
<td>[136]</td>
<td>Global-scale data placement</td>
<td>Integer Linear Programming</td>
<td>Cloud</td>
</tr>
<tr>
<td>[137]</td>
<td>Global-scale data placement</td>
<td>Iterative optimisation</td>
<td>Cloud</td>
</tr>
</tbody>
</table>

Global-scale Data Management. Some applications may require managing data in a globally distributed manner. This can be challenging as the overheads of coordination to keep data replicas consistent can be high due to wide-area network latency issues. This is worse in case of applications requiring strong cross-data center consistency, e.g. in the case of distributed databases. Application providers need to place their applications and data in appropriate data centers to minimise these challenges. It is noted that we do not consider data consistency challenges in this thesis.

State-of-the-art Placement

Placement has been studied in Cloud and other distributed computing environments like content delivery networks (CDN), Fog and Edge computing with different assumptions, objectives and scenarios. In CDN, a body of work has been proposed to address replica placement problems based on the network of geographically distributed servers used to deliver content, e.g., files, videos, or streaming videos. In this context, placement solutions usually assume that the topologies of the network is known and the challenge is placement of content across the network while considering different factors, e.g., performance [122, 128], communication [129] and total replication costs [130].
In [122], the authors assume unlimited capacity of resources at each potential location and their optimisation problem is modelled as a k-median problem. They propose a heuristic to place replicas at locations such that the overall network latency between users and replicas is minimised.

In Fog computing, work such as [119] propose a heuristic to adaptively place IoT application components across Fog and Cloud infrastructure. Their objective is to minimise the service delivery time for IoT applications. They factor in inter-fog node and fog-Cloud latency as well as bandwidth and infrastructure processing constraints. In [120], the authors present a dynamic placement heuristic for container-based IoT application components across geographically distributed Fog nodes. They consider the mobility of end-users and current Fog node processing capabilities to minimise the overall service latency. Similarly, in integrated Edge-Cloud environments, work such as [131] present a particle swarm optimisation algorithm to dynamically place IoT application components, with the aim of minimising the total system energy consumption of the edge-Cloud infrastructure as well as the overall task processing latency. However, solutions in these environments usually assume limited infrastructure capabilities and power constraints [138, 139]. In Cloud computing and the context of this thesis, these issues are not paramount in comparison to features such as high availability and scalability.

In the Cloud context, work such as [132] explored how to dynamically place highly latency-sensitive VMs running virtual desktops across Cloud data centers close to end users. A greedy heuristic approach was proposed to solve placement challenges. The goal was to minimise network latency to improve the performance of latency sensitive virtual desktops. A black box fingerprinting technique for a VM’s network traffic was proposed to rank the latency sensitivity of virtual desktops to decide which ones to replace.

The authors in [8], proposed a dynamic placement approach for web application replicas across geographically distributed Cloud data centers. They assumed that pre-configured geographic bins were provided by application providers. These bins could be in any geographic area, e.g., cities, countries etc. Their goal was to minimise the
overall network latency. To tackle this they proposed an iterative greedy clustering algorithm. After mapping users to appropriate bins based on their locations, the algorithm iteratively tried to map each bin to the best possible (closest) data center at each iteration. This continued until all data centers in the list of iterations had sufficient threshold-based traffic. This solution also optimised the number of locations to run applications to reduce the operating costs. However, they ignored the cost of the adaptations.

In [126], the authors introduced an SLO-aware placement optimisation solution for web applications across distributed Cloud data centers. Their objective was to minimise user-to-data center and inter-data center network latencies to reduce the overheads of cross-data center data consistency and thus improve the performance of applications. They proposed a genetic algorithm to dynamically place application and associated data replicas to avoid/minimise SLO violations. They also considered minimising the number of application relocations as a cost of adaptation. However this work did not consider inter-data center latency challenges that arise during the adaptation process.

In [127], the authors proposed an availability-aware placement method for applications in hybrid Cloud environments. They used duplicates (i.e. active backup copies of applications) to improve application availability against node or link failures in the Cloud. Their objective was to improve availability of applications as well as maximise the total number of placed applications to improve the resource utilisation. To achieve this they proposed a genetic algorithm to solve the placement problem. In work such as [133], a new fast and scalable algorithm based on detection of subgraph isomorphism was introduced. This improved application availability and resource utilisation. This work was valid for large-scale environments and it could be used for application models such as MapReduce and 3-Tier. However, the work did not consider application performance after failover conditions.

In [134], the authors present a holistic approach to optimise the selection of data centers for server placement. For data center selection, their objective was to minimise the overall network latency between selected data centers. They proposed 2-approximation algorithms to solve this issue. They assume a data center network was structured as a
Based on this, they proposed a server placement approach for virtual machines (VMs) within each selected data center with the aim to reduce communication between VMs.

In [135], the authors investigate how to place services across geographically distributed Cloud data centers to provide interdependent services as part of a service-oriented application. They attempt to minimise the overall traffic including the traffic between services as well as the number of Cloud locations to reduce the operating cost. They propose a genetic algorithm to address this problem.

Regarding data placement, the authors in [136] propose a dynamic data placement approach across multiple Clouds. The objective was to provide a balance between operating costs and latency issues, whilst considering fault tolerance, SLOs and consistency requirements. They address the problem using an approach based on integer linear programming. Similarly, in [137] the authors present a dynamic placement solution applicable for global-scale data across geographically distributed Cloud data centers. An iterative optimisation algorithm was proposed to address placement issue.

A summary of the related work discussed in this section is shown in Table 2.1.

### 2.7 Distributed Cloud Container Deployment Models and Platforms

Container technologies have gained popularity in developing, deploying, and managing application in distributed Cloud environments [37–39]. They provide a simple, yet powerful, abstraction that provides a clean decoupling of applications from the infrastructure on which they run. This decoupling provides several key benefits: it supports consistent and portable deployment of applications across (heterogeneous) Clouds [26]; it allows developers to focus on code without worrying about machine-centric management typically required for VM-based application deployment and it allows for cross-Cloud management platforms to focus on the deployment and management across Clouds, regardless of the applications within containers [16, 26].
Numerous container management projects from industry and open source communities have been developed. Even though they aim to address Cloud middleware platform requirements and facilitate deployment processes, they have taken different approaches to achieve this goal, especially in terms of the architectural models that they realise. In this section, we first provide a classification of container deployment models based on clustered server models in distributed Cloud environments and then we discuss several container management projects that fit each deployment model.

**Container Deployment Models**

As shown in Figure 2.4, container deployment in distributed Clouds can be classified into single-cluster and multi-cluster models. Multi-cluster deployment models can be further classified into independent and federated (centralised) models.

**Single-Cluster Deployment Models**

The single-cluster deployment model is the most basic deployment model. Each deployment consists of a single logical cluster of geographically distributed VMs that are managed by a single manager as shown in Figure 2.5c. Cluster nodes are deployed at two or more Cloud data centers. Clusters can support multi-zone, multi-region or multi-Cloud deployments depending on the platform implementation. This model is easy to deploy. To improve the availability, a cluster manager can be replicated on multiple servers at different Cloud locations. However, a cluster failure can happen in this model and introduce availability issues [140]. A scheduler is usually aware of
the locations of nodes and provides different scheduling capabilities. For example, in multi-zone deployments, the spreading behaviour of the application containers can be extended across zones to improve availability of applications against isolated zone failures.

However for global-scale application deployments, this model has some limitations. One is that all management overheads are handled by a single manager which can introduce scalability issues. Another limitation is that schedulers that support multi-location spreading behaviour, require a homogeneous deployment of nodes (i.e., the same number and size of nodes) at each location. This can lead to costly, over-provisioning situations as application workloads can vary at different locations. Application providers can overcome these limitations by deploying multiple clusters based on this model across multiple locations. However, this would increase the management complexity for application providers as they need to take responsibility to manage, deploy and scale clusters separately.
Federated Multi-Cluster Models

In a federated multi-cluster deployment, a number of clusters can be run and managed independently, but they can also be managed through a single API by a centralised manager as shown in Figure 2.5a. This federation layer provides cross-cluster functionalities, e.g., cross-cluster scheduling, service discovery and application deployment. It can easily and automatically apply configurations, synchronise the application deployment as well as roll out application updates across clusters. This can be beneficial to large-scale application deployments. For example it can provide multi-cluster governance and offer a single unified view of the system. With regards to availability, this model supports high availability. Thus, in the presence of federation manager or individual cluster failures, others clusters can work independently since each cluster has independent management capabilities including an independent scheduler. However, this model also has some issues as it reduces cross-cluster isolation. Thus any issue or errors in the federation manager can affect all clusters and cause multi-cluster outages.

Independent Multi-Cluster Models

The independent multi-cluster deployment model is similar to the federated model since a deployment has multiple clusters, but it does not have a federation layer as shown in Figure 2.5b. In this model, each cluster is completely independent. Thus, this model improves high availability as it provides complete cluster isolation. This model also provides centralised observability and governance across multiple clusters. However this model can increase the management complexity, which is left to application providers, i.e., they are responsible for (manual) deployment and software updates across clusters. Despite this, platforms adopting this model aim to eliminate this complexity while managing each cluster separately.

Distributed-Cloud Container Platforms

Single-Cluster Platforms. Platforms that support single-cluster deployment models include Cloud-provider-managed Kubernetes Platforms such as Google Kubernetes
Table 2.2: Summary of Container Management Platforms in Distributed Clouds.

<table>
<thead>
<tr>
<th>Container Platform</th>
<th>Cluster Deployment Model</th>
<th>Orchestrator</th>
<th>Cloud Deployment Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Managed Kubernetes Platforms [34, 43, 141]</td>
<td>Single-Cluster</td>
<td>Kubernetes</td>
<td>Multi-zone</td>
</tr>
<tr>
<td>ECS [33]</td>
<td>Single-Cluster</td>
<td>AWS</td>
<td>Multi-zone</td>
</tr>
<tr>
<td>DC/OS [142]</td>
<td>Single-Cluster &amp; Independent Multi-Cluster</td>
<td>Mesos</td>
<td>Multi-Cloud</td>
</tr>
<tr>
<td>Knative [143]</td>
<td>Single-Cluster</td>
<td>Kubernetes</td>
<td>Multi-Cloud</td>
</tr>
<tr>
<td>Nomad [144]</td>
<td>Single-Cluster &amp; Federated Multi-Cluster</td>
<td>Nomad</td>
<td>Multi-Cloud</td>
</tr>
<tr>
<td>KubeFed [145]</td>
<td>Federated Multi-Cluster</td>
<td>Kubernetes</td>
<td>Multi-Cloud</td>
</tr>
<tr>
<td>Rancher [44]</td>
<td>Independent Multi-Cluster</td>
<td>Kubernetes</td>
<td>Multi-Cloud</td>
</tr>
<tr>
<td>Anthos [146]</td>
<td>Independent Multi-Cluster</td>
<td>Kubernetes</td>
<td>Multi-Cloud</td>
</tr>
<tr>
<td>OpenShift [147]</td>
<td>Independent Multi-Cluster</td>
<td>Kubernetes</td>
<td>Multi-Cloud</td>
</tr>
</tbody>
</table>

Engine (GKE) [43], Azure Kubernetes Services (AKS) [34] and Amazon Kubernetes Service (AKS) [141]. These platforms support multi-zone deployments that are managed by Cloud providers. Another single-Cloud platform is Amazon Elastic Container Service (ECS) [33]. ECS has two methods for managing the cluster infrastructure: Fargate and EC2 [33]. While the former is managed by the Cloud provider, the latter is managed by application providers. Such solutions can however lead to vendor lock-in.

To avoid this issue, platforms that adopt multi-Cloud deployment models can be used. One example of this is the Distributed Cloud Operating System (DC/OS) Platform [142]. DC/OS is an open-source, distributed system for the Mesos platform. It manages multiple machines that can belong to different Clouds from a single interface. It deploys containerised and legacy applications onto those machines and provides resource management, inter-machine networking and service discovery. Another single-cluster, multi-Cloud platform is Knative [143], a Kubernetes-based open-source platform for deploying and managing serverless workloads. Knative extends Kubernetes capabilities by providing a set of middleware components that are needed to deploy and scale containerised applications anywhere. Nomad [144] is another open-source platform that is designed to support single-cluster models across multiple Clouds. Knative provides a simple, lightweight, and easy-to-operate platform for both containerised and non-containerised applications.

Federated Multi-cluster Platforms. Kubernetes Cluster Federation [145] also known as
KubeFed, is an open-source platform that supports the federated multi-cluster model for Kubernetes clusters based on multi-Cloud deployment models. It runs in one cluster (e.g. a host cluster) which is used to run the KubeFed control plane and expose the KubeFed API. Other Kubernetes clusters can join a host cluster to be become members of the federation. However, KubeFed is still evolving and not yet mature. Recently, Nomad [144] has been put forward. It also supports multi-cluster federated capabilities for Nomad clusters in multi-Cloud settings.

**Independent Multi-cluster Platforms.** Rancher [44] is an open-source solution that supports the independent model of multi-cluster deployments in multi-Cloud environments. It simplifies the deployment and management of Kubernetes clusters across data centers, while each cluster manages its containers independently through Kubernetes. It provides a unified single view multi-cluster application management solution. Google Anthos [146] is another multi-cluster, multi-Cloud Kubernetes management platform. It provides the ability to manage Kubernetes clusters across multiple Clouds and facilitates containerised application deployment across clusters. This can limit the possibility of vendor lock-in as it relies on open source solutions to manage clusters and avoid Cloud-specific management services. The Red Hat OpenShift platform [147] is another container platform that supports the independent multi-cluster model. It provides visibility and control for Kubernetes clusters and applications in multi-Cloud deployment settings. A summary of the aforementioned container management platforms is shown in Table 2.2.

### 2.8 Cloud Container Elasticity

Elasticity is a key characteristic of Cloud computing. It provides application providers with capabilities to dynamically allocate compute resources to their applications based on evolving workloads. This in turn requires solutions to control the degree of elasticity to help prevent under- or over- provisioning of resources and thus achieve QoS at lower costs. Containers in Cloud computing usually run on elastic, scalable clusters of VMs that provide scalable infrastructure amongst other aspects such as isolation and
security [13, 28]. Container-based resource models in Clouds can be divided into two dimensions: container-based and VM-based models.

The problem of controlling container elasticity in Cloud environments has been addressed at either one level (container level or VM level) or both levels. Such solutions can be classified into two types based on whether they manage the elasticity of the underlying infrastructure (i.e., VMs) or not. In the following section, we discuss the state-of-the-art research related to each classification type.

**State-of-the-art in Elasticity**

**Container-level Elasticity**

Solutions in this class provide elasticity mechanisms at the container level without managing the underpinning infrastructure. They adapt the processing capacity for containerised applications by adding/removing containers (e.g., through horizontal scaling approaches) [35, 148–153] or through resizing containers - often referred to as vertical scaling [154, 155]. Most of these solutions are based on a predefined threshold-based model for auto-scaling [35]. Kubernetes Horizontal Pod Autoscaler (HPA) [148] and service auto-scaling (SA) [149] with the Amazon Elastic Container Service (ECS) [33] are examples of this model. HPA is a Kubernetes-based container-level elasticity controller that reactively adjusts the number of containers for an application service to match the average CPU utilisation of containers to a predefined CPU utilisation threshold. Similar to HPA, SA is another horizontal scaling solution that provides HPA for ECS.

Such solutions typically use elasticity rules with fixed thresholds. This can give rise to performance issues. Apart from workload intensity, the processing capabilities of containers can vary, e.g. due to noisy neighbour interference between containers on the same over-utilised VM [16] or due to the whole data center being overloaded [45]. Such static threshold-based solutions are more likely to fail when handling ad hoc situations. To overcome this issue, solutions need to consider the current status of the execution environment when making scaling decisions. In [35], the authors present a dynamic
A threshold-based elasticity mechanism that is able to dynamically configure thresholds at run-time by considering changes in container processing capabilities. The solution is based on two-level monitoring: application and container monitoring.

The authors in [150] propose a container elasticity controller that adapts the number of containers for queue-based micro-services according to the container processing capacity and workload intensity at fine-grained time scales, e.g., over the last minute. Each microservice has a request queue and its container instances listen to and pull requests to process from the queue. Moreover, to handle unexpected spikes in workload that are not reflected in the observed workload, they propose a mechanism for providing extra containers to offer additional capacity. This capacity is auto-scaled based on the ability to handle the current workload.

For vertical scaling, Kubernetes Vertical Pod Autoscaler [154] provides a vertical container-level auto-elasticity solution that automatically resizes containers by adding/removing more resources (e.g., CPU and memory) to help right-size applications. It allows rescheduling of containers at other nodes that have available resources as and when required. It is able to scale-up containers that are resource-hungry and scale-down containers that are over-requesting resources based on their usage over time. In [155], the authors propose a vertical container elasticity controller using reactive threshold-based rules to automatically adapt the size of resources needed for Docker containers.

In [151], the authors present a proactive horizontal auto-scaling solution for web applications based on the Kubernetes platform. They aim to increase the availability of applications by reducing the number of rejected requests based on a slight increase in resource utilisation. They use various AI-based forecasting methods for predicting workloads and the most reliable one is selected. They propose a simple management parameter for controlling the desired resource utilisation of applications.

In work such as [152], a horizontal elasticity controller is proposed for Docker-based web applications. It uses a hybrid approach based on the ARIMA technique to predict future workloads whilst fixed threshold-based rules are applied for more reactive scenarios. It scales out using the hybrid approach but is able to scale in reactively.
In [153], the authors present a container-level elasticity solution using both horizontal and vertical scaling methods to cope with heterogeneous workloads associated with different container-based health services. A threshold-based reactive approach is used depending on various quality of experience metrics, e.g., response time, availability and average request rate.

**VM-based Elasticity**

In this classification, works focus on providing the infrastructure for container-based applications in Clouds with auto-elasticity features, e.g., adapting the number of VMs (horizontal scaling) or resizing VMs (vertical scaling) based on particular resource demands. Such approaches usually make VM-level elasticity decisions based on monitoring the utilisation of VMs and/or unschedulable (pending) containers that cannot be launched due to clusters being overloaded. These approaches aim to avoid disruptions to running containers and ensure scalability at minimum costs. Solutions need to be tightly coupled with the container level scaling [156–158] or have (re)scheduling capabilities, e.g., replacing running containers on under-utilised VMs before terminating unnecessary VMs to downscale the cluster whilst ensuring given quality of service (i.e., container-aware VM-level elasticity is offered) [159–162].

**Two-level Elasticity**

In [156], the authors propose a container and VM level elasticity mechanism to auto-scale containers for multi-tier web applications both horizontally and vertically. They use discrete-time feedback controllers to periodically estimate the amount of cluster resources needed to run sufficient containers to maintain predefined SLOs and response times with minimum cost. Similarly, in [157] the authors propose a horizontal and vertical auto-scaling solution for both containers and VMs using Docker-based containers. Their objective is to reduce the costs by reducing the number of (leased) VMs.

In work such as [158], the authors propose a two-level vertical elasticity approach to automatically adapt both the size of Docker containers and VMs to cope with dynamic
workloads. Their aim is to improve application performance. They adopt a reactive approach using predefined thresholds.

**Container-aware VM-level Elasticity**

The Kubernetes Cluster Autoscaler (KCA) [159] and the ECS Cluster Auto-scaling (ECA) [160] components provide container-aware VM-level auto-elasticity solutions, albeit with different levels of container awareness. ECA auto-scales the number of VMs based on the average cluster resources (CPU or memory) that has been reserved or utilised for containers together with a spare capacity reservation. However, it is not aware of pending containers when making elasticity decisions. On the other hand, KCA is more aware of containers and expands the cluster when pending containers are non-schedulable for a given period, e.g., due to insufficient resources. It also shrinks the cluster by removing underutilised VMs and hence provides the ability to automatically replace running, movable containers to other existing nodes/servers. It is noted that ECA is also capable of replacing containers, however this requires manual intervention.

Moreover, ECA also allows reserving additional (spare) capacity. Both KCA and ECA have the ability to quickly scale the capacity of their clusters when needed from other clusters as required for serverless container platforms. Such features can accommodate sudden, unexpected workloads. Both KCA and ECA, support a distributed-Cloud single-cluster architectural model that aims to improve availability against isolated zone failures.

In work such as [161], the authors propose a cost-efficient VM-level cluster elasticity mechanism with container (re)scheduling capabilities to meet the resource demand of containerised applications by shrinking the size of the cluster when possible. This runs on Kubernetes clusters and aims to optimise the initial placement of pending containers and the relocation of movable, running containers to best fit the cluster infrastructure with as few VMs as possible before making scaling decisions. Similar to KCA, the proposed solution scales out clusters by adding a new VM, when the scheduler fails to provide enough resources for pending containers from the current cluster. It is also able to scale in the cluster so that all pending containers are successfully scheduled,
by removing any idle, auto-scaled nodes (i.e. VMs with no running containers). An extension of this work presented in [162] has the improvement in considering the heterogeneity of cluster resources, e.g., different VM sizes and types, to support various types of Cloud workloads and thus satisfy cost and QoS requirements.

A summary of the container and VM elasticity work discussed in this section is shown in Table 2.3.
Table 2.3: Summary of Container and VM Elasticity Work in Cloud Computing

<table>
<thead>
<tr>
<th>work</th>
<th>Objective</th>
<th>Scope</th>
<th>Level</th>
<th>Method</th>
<th>Approach</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>[35, 148, 149]</td>
<td>Performance and cost</td>
<td>One level</td>
<td>Container</td>
<td>Horizontal scaling</td>
<td>Reactive</td>
<td>General</td>
</tr>
<tr>
<td>[150]</td>
<td>Performance and cost</td>
<td>One level</td>
<td>Container</td>
<td>Horizontal scaling</td>
<td>Reactive</td>
<td>Queue-based microservices</td>
</tr>
<tr>
<td>[151]</td>
<td>Resource utilisation</td>
<td>One level</td>
<td>Container</td>
<td>Vertical scaling</td>
<td>Reactive</td>
<td>General</td>
</tr>
<tr>
<td>[152]</td>
<td>Resource utilisation, cost</td>
<td>One level</td>
<td>Container</td>
<td>Vertical scaling</td>
<td>Reactive</td>
<td>Docker-based</td>
</tr>
<tr>
<td>[153]</td>
<td>Application availability</td>
<td>One level</td>
<td>Container</td>
<td>Horizontal scaling</td>
<td>Proactive</td>
<td>Web apps</td>
</tr>
<tr>
<td>[154]</td>
<td>Performance and cost</td>
<td>One level</td>
<td>Container</td>
<td>Horizontal scaling</td>
<td>Hybrid</td>
<td>Docker-based Web apps</td>
</tr>
<tr>
<td>[155]</td>
<td>Performance and cost</td>
<td>One level</td>
<td>Container</td>
<td>Both</td>
<td>Reactive</td>
<td>Health services</td>
</tr>
<tr>
<td>[156]</td>
<td>Performance and cost</td>
<td>Two level</td>
<td>Container and VM</td>
<td>Both</td>
<td>Reactive</td>
<td>Web apps</td>
</tr>
<tr>
<td>[157]</td>
<td>Cost</td>
<td>Two level</td>
<td>Container and VM</td>
<td>Both</td>
<td>Reactive</td>
<td>Web apps</td>
</tr>
<tr>
<td>[158]</td>
<td>Performance</td>
<td>Two level</td>
<td>Container and VM</td>
<td>Vertical scaling</td>
<td>Reactive</td>
<td>General</td>
</tr>
<tr>
<td>[159–162]</td>
<td>Resource utilisation, cost</td>
<td>One level</td>
<td>VM</td>
<td>Horizontal scaling</td>
<td>Reactive</td>
<td>General</td>
</tr>
</tbody>
</table>
2.9 Summary and Discussion

In this chapter, we have reviewed the state of the art research in application management in distributed Clouds with particular focus on workload management and container placement. We examined research and solutions focused on container deployment and elasticity in the Cloud. In particular, we focused on their abilities to address potentially global web-based challenges to maintain scalability and performance goals. In this thesis, our aim is to enhance the state-of-the-art by addressing challenges and/or improving existing solutions for managing web applications in distributed Clouds based on intelligent deployment and use of containers.

We classified container deployment in distributed Clouds into different models based on the underpinning VM cluster models across data centers that they depend upon and presented several platforms representative of each model. Even though these solutions provide some capabilities for application providers to manage resources needed for container-based applications, they can be limited in managing application QoS and the intelligent, cost effective decisions that are required.

As discussed web applications can require both high availability and performance to improve QoS and support SLOs. In Chapter 3) we tackle the problem of network latencies (i.e. proximity to users in global, multi-Cloud settings with a potential global distribution of users. We show how it is possible to deploy containerised web application to achieve high availability and performance based on a single-cluster container deployment model.

In Chapter 4 we also address the problem of long-term spatial workload variations and the impact on containerised web application deployment so that intelligent, elastic deployment of container clusters across geo-distributed data centers can be supported to maintain web application QoS and support SLOs with minimum deployment and adaptation costs. This is based on an independent multi-cluster container deployment model.
With regards to container elasticity in Clouds, solutions were classified into two main approaches: container-level elasticity and container- and VM-level elasticity solutions. Existing approaches fail to consider application-related metrics, e.g., container processing capacity without violating SLOs when estimating cluster capacity. In Chapter 5 we propose a container elasticity solution that considers high-level metrics to autoscale not only containers but also the VMs needed for containerised web applications. Since we assume multi-cluster containerised web applications are deployed across geographically distributed Cloud data centers, our solution mainly provides cost-aware auto-elastic scaling using inter-cluster resource utilisation to handle sudden overload situations.

In the following chapter, we introduce the proposed approach for providing latency-aware failover strategies for container-based web application deployments in distributed Clouds.
Chapter 3

Latency-aware Failover Deployment for Containerised Web Applications in Distributed Clouds

The previous chapter covered the state of the art in deploying web applications across multiple Clouds and use of containers for such deployments. One benefit is that it can provide failover deployments that can cope with potential Cloud outages and hence improve application availability. However, as discussed in Section 2.6, such deployments need to consider inter-data center network latencies, proximity to users and attempt to reduce the impact of geo-replication overheads on application performance. To address these issues, there is a need to address placement problems to support latency-aware failover deployments through containers and microservice solutions, however this gives rise to two sub-problems: creating and placing cluster nodes (VMs) and the associated placement of containerised microservices and their replicas across those nodes. This is the focus of this chapter.

Specifically, in this chapter we propose an approach to tackle placement problems for container deployments in distributed Clouds. The goal is to maintain web application performance with associated service level objectives (SLOs) under normal conditions and in situations where partial or complete failures have occurred. This addresses Research Problem 1 presented in Section 1.3.

We propose a user session-based model to estimate SLO violation rates. Following this we present a cluster node (server) placement algorithm based on genetic algorithms for selecting primary-backup pairs of data centers where each pair runs microservices at a primary data center and their backups at a backup data center. These are chosen to serve users at different areas/locations. Building on this, we propose high availability (HA)-aware microservice deployment algorithms for generating the deployment configuration of primary and backup microservices across given node clusters in distributed data centers. Through extensive experiments on the NeCTAR Research Cloud, we demonstrate the effectiveness of the approach and the extent that it tackles Research Problem 1.

Latency-aware Failover Strategies for Containerized Web Applications in Distributed Clouds

Yasser Aldwyan\(^{a,b}\), Richard O. Sinnott\(^a\)

\(^a\)School of Computing and Information Systems, The University of Melbourne, Australia
\(^b\)Faculty of Computer and Information Systems, Islamic University of Madinah, Saudi Arabia
yaldwyan@student.unimelb.edu.au, rsinnott@unimelb.edu.au

Abstract

Despite advances in Cloud computing, ensuring high availability (HA) remains a challenge due to varying loads and the potential for Cloud outages. Deploying applications in distributed Clouds can help overcome this challenge by geo-replicating applications across multiple Cloud data centers (DCs). However, this distributed deployment can be a performance bottleneck due to network latencies between users and DCs as well as inter-DC latencies incurred during the geo-replication process. For most web applications, both HA and Performance (HAP) are essential and need to meet pre-agreed Service Level Objectives (SLOs). Efficiently placing and managing primary and backup replicas of applications in distributed Clouds to achieve HAP is a challenging task. Existing solutions consider either HA or performance but not both. In this paper we propose an approach for automating the process of providing a latency-aware failover strategy through a server placement algorithm leveraging genetic algorithms that factor in the proximity of users and inter-DC latencies. To facilitate the distributed deployment of applications and avoid the overheads of Clouds, we utilize container technologies. To evaluate our proposed approach, we conduct experiments on the Australia-wide National eResearch Collaboration Tools and Resources (NeCTAR - www.nectar.org.au) Research Cloud. Our results show at least a 23.3% and 22.6% improvement in response times under normal and failover conditions respectively compared to traditional, latency-unaware approaches. Also, the 95th percentile of response times in our approach are 1.5 ms above the SLO compared to 11–32 ms using other approaches.

3.1 Introduction

Cloud computing has become a vital computing model underpinning information technology. It provides numerous valuable characteristics such as scalability, on-demand, self-service resource provisioning and cost-effectiveness. A substantial number of Cloud services now exist and offer hardware resources (e.g. compute and storage) as well as software resources and applications. These features and services have
3.1 Introduction

Figure 3.1: Geographical container clustering of web applications across two geographical areas in distributed clouds

been highly attractive to many organisations that wish to deploy and subsequently have their applications outsourced and hosted in Cloud environments [68].

However, high availability (HA) still remains a challenge for Cloud providers, even for the most reliable providers, due to the continued occurrence of different forms of Cloud outages [54, 163]. Cloud outages, even short-term ones, can have a considerable impact [49]. A side-effect of these outages on applications is system downtime which can affect business continuity, a loss of reputation and associated revenue. As one example, [164] identified that approximately 11,000 dollars per minute was lost to one business due to such an outage.

To improve availability and other requirements such as performance and locality [56], Cloud providers have typically applied a distributed Cloud model and thus distributed their Cloud resources and services across multiple, geographically dispersed data centres (DCs). These provide a variety of built-in HA services and tools, e.g. floating IP addresses and geo-replication capabilities. These have encouraged application and service providers to distribute and replicate their applications and services across different zones and/or regions to help meet application requirements regarding availability and performance.
Multi-cloud distributed application deployment offers a promising HA mechanism to achieve increased availability of Cloud-based applications [165]. This can be done by implementing multi-Cloud failover strategies where primary and backup replicas are deployed to different DCs, potentially from different Cloud providers. In this scenario, when a primary replica of an application becomes unavailable, e.g. due to a Cloud outage, a backup replica which is deployed in another Cloud provider takes over - ideally immediately.

This distributed Cloud model can have a number of other benefits, e.g. reducing Internet traffic, reducing vendor lock-in, minimising network latency and so on [53, 56, 165]. Coady et al. [56] argue that a distributed Cloud can also help provide instant failover capabilities. In this case, both replicas should be near users and not far from each other to reduce replication time under normal conditions and thereby reduce delays in response times, even after particular failures take place. Normal conditions in the context of this thesis are meant to be healthy application deployment conditions where there are no outages in data centers running application replicas (see Figure 3.3). Where to replicate applications to achieve HAP in large-scale, spatially distributed Cloud environments where failures, particularly Cloud outages, can arise has not been resolved however. This is the focus of this paper.

There are many challenges that need to be addressed to realise HAP for many modern web-based applications [49]. Firstly, with the physical limitations of wide-area network latencies across the Internet and the geo-distribution of users of web applications, deploying primary and backup replicas of applications in distributed Clouds without considering the locations of workloads (i.e. users) and Cloud DCs can increase network latencies. This can have a negative impact on the overall performance and response times experienced by end users. Large user-to-DC and/or inter-DC latencies can incur large delays in response times of user requests which affect the overall quality of service (QoS) experienced by users. Manually selecting appropriate DCs for server placement is naive and unlikely to be the optimal placement. There is a need for placement algorithms that take into account multiple factors to help automate failover plans and achieve optimal HAP. Secondly, the management of distributed, heterogeneous Cloud resources, as well as the application components running on top of these
resources can be a difficult task, especially factoring in geo-replication of user sessions and associated data across multiple Clouds. Virtual cross-Cloud clustering and private networks are typically necessary to automate and facilitate this task.

The motivation behind this paper is to address the above-mentioned challenges to help application providers migrate their web applications to the Cloud whilst factoring in and optimising the deployment with specific focus on availability and performance. We aim to improve the responsiveness of applications for geographically distributed users under normal conditions as well as in the presence of Cloud outages. This can help application providers meet customer SLOs\(^1\), even in the presence of partial or complete Cloud outages. The SLO here is referred to as the desirable response time to user requests. Our work targets session-based web applications and is limited to DC, availability zone and cloud outages in terms of failure level, i.e. we do not consider partial outages.

To address the challenges resulting from distribution and heterogeneity, container technologies and use of microservices are adopted to support lightweight virtualization and application management solutions [166]. Such microservices allow to break up monolithic applications into lightweight, independently deployable services [21] that offer many advantages. Containerisation solutions such as Docker provide a solution for application packaging to overcome, or at least mitigate, application portability and Cloud interoperability issues in distributed Cloud environments [40, 68, 167]. Microservices also help increase scalability, portability and availability of applications [21]. Additionally, to facilitate multi-Cloud failover capabilities and inter-DC communication complexity and give application providers more control when deploying applications across heterogeneous distributed Clouds, container technologies now support cluster management and orchestration services such as Kubernetes [30]. In our work we utilise container technologies and adopt a microservice-based application architecture as shown in Figure 3.1.

The key contributions of this paper are as follows. Firstly, we propose an approach for autonomously generating HAP deployment plans of web applications starting from

\(^1\)SLOs are measurable key elements in service level agreements (SLAs).
deploying servers in distributed clouds and then deploying container based microservices on top of that infrastructure. Secondly, we present a server placement algorithm suited for optimising DC selection. This leverages genetic algorithms (GAs) that factor in the proximity of users and inter-DC latencies. Chosen DCs will run servers as part of container based clusters to form the application infrastructure. Finally, we conduct experiments using case studies based on a real-world transactional web benchmark (TPC-W) application [47] on the Australia-wide NeCTAR Research Cloud [46]. The NeCTAR Research Cloud is itself based on multiple geo-spatially distributed DCs (availability zones). Our results show at least 23.3% and 22.6% improvement in the response time under normal and failover conditions respectively as compared to more traditional, latency-unaware approaches. Also, the 95th percentile of response time in our approach is at most 1.5 ms away from the pre-agreed SLO while it is at least 11 ms (and up to 32 ms) with the other approaches. Furthermore, our proposed solution can produce deployment plans, which are the same as the ones produced by the optimal algorithm, in near real-time (approximately 4 minutes).

The rest of this paper is organised as follows. Section 3.2 provides background and related work. We describe the application and container cluster-based infrastructure models in Section 3.3. In Section 3.4, we formulate the research problem. Section 3.5 discusses our proposed solution in detail, including the response time and violation models and proposed algorithms for DC selection and microservice deployment. Following this, we evaluate the proposed solution in Section 3.6. Finally, Section 3.7 presents a summary.

### 3.2 Background and Related Work

#### Background

##### HAP Application Deployment in Distributed Clouds

Application deployment models have been dramatically changed by the Cloud computing paradigm [68] with many features now offered for HA solutions [56]. In the
standard, centralised deployment model, applications are typically deployed in a single Cloud data center (DC). Even though this kind of deployment can tolerate hardware failure by replicating applications in different physical hosts within a given DC, the model cannot handle complete (or potentially partial) DC outages. This deployment model also has an impact on application performance (i.e. response times) when users are geographically far away from the Cloud DC due to the physical limitations of wide-area network latencies at the Internet scale [65].

To handle these issues, most mainstream Clouds have adopted a distributed model where Cloud resources and services are distributed across multiple, geographically dispersed DCs [56]. These can be classified as homogeneous or heterogeneous models. In the homogeneous distributed cloud model, all cloud DCs belong to a single Cloud provider, e.g. Amazon (AWS), while the heterogeneous model uses multiple DCs from different Cloud providers. A distributed Cloud typically partitions its DCs across geographical zones, also known as availability zones. These are typically distributed across regions around the globe [63, 64].

In this context, application deployment needs to consider distributed placement strategies for deployment of applications. This emerging model, which has been encouraged by many Cloud providers [63, 64], helps application providers improve availability and the QoS of their applications [68]. This deployment model can be classified into two sub-models: single-Cloud and multi-Cloud. In the single-Cloud distributed deployment model, applications can be deployed across multiple zones in a homogeneous distributed Cloud – also known as multi-zone deployment. To help achieve HA in this deployment model, a number of built-in, Cloud dependent HA services and features have been offered by Cloud providers such as floating IP addresses, cross-zone load balancing mechanisms, geo-migration, geo-replication and so on [63, 69]. This model can handle DC and zone outages by applying multi-zone failover strategies, where primary and backup replicas are deployed in different zones. When a primary replica of a web application becomes unavailable, e.g. due to a zone outage, a backup replica deployed in a different zone, takes over immediately. However, this HA solution fails when a region outage occurs since primary and backup replicas are located in the same region.
Multi-region failover solutions, where primary and backup replicas are deployed in different regions within the same Cloud can tackle region-wide failures (e.g. failover in an Azure SQL database service [69]), however they have a number of limitations. They can affect the performance of stateful and time response-critical applications (e.g. session-based web applications) thereby requiring replication of sessions and/or data between primary and backup replicas with associated communication overheads. It can be the case that after a failover, large delays in response times occur due to backup replicas being far away from the user workloads that may be associated with the primary DC at a given point in time. The above solutions are unable to handle major outages and can lead to vendor lock-in issues.

Multi-cloud failover solutions can overcome cloud outages and avoid vendor lock-in issues, however, they have the same performance issues as multi-region ones unless the location of users and DCs are considered when placing applications and their backup replicas. Placement is a key issue in geographically distributed cloud environments and is largely left to application providers [54]. As mentioned, container solutions can help support multi-cloud deployments where the inter-communication between application components running over heterogeneous cloud resources can be abstracted from the underlying infrastructure at the operating system level, e.g. by providing a cluster of nodes as part of a cross-cloud cluster service. Such a cluster requires container orchestration support for deploying microservices across nodes and require overlay private networks and other HA features such as monitoring and auto-scaling [30]. However, HAP microservice deployment across geo-clustered nodes produces other placement problems that have not yet been satisfactorily addressed: where exactly to optimally deploy the cluster at a given point in time.

Related Work

Low Geo-replication and Failover Overhead. The problem of availability in Cloud computing has been extensively studied. However, much of the work has focused on providing HA solutions for failures within a DC such as VM and hardware failures.
Solutions include replication [168–171], migration [172, 173], deployment [174], affinity rules and clustering [175] and at different failover levels: VM [168, 169, 174, 175], container [171, 173] and application [170].

HA solutions in distributed cloud contexts can help overcome failures beyond the ones within DCs such as cloud outages. However, such solutions come at a cost of increased geo-replication overheads due to inter-DC network latency. This affects latency-sensitive that can impact user-facing web applications where response times are often critical.
There have been few efforts aimed at reducing this overhead. Some approaches include high-speed links among zones [176] and among regions [64], or using lightweight container solutions to reduce the size of snapshots that need to be transferred, compared to more heavyweight VM-based solutions [177, 178]. Others [179] have proposed optimisation techniques to dramatically minimise the data size to be transferred by eliminating redundant memory and disk data and using data compression techniques. However, none of these approaches consider the geo-locations of DCs where replicated applications should be deployed to reduce the geo-replication overheads. In this work, we address the placement issue of distributed application deployment to reduce inter-DC network latencies and thus improve availability and performance.

**HAP Web Applications for Distributed Client Bases.** Proximity to users is often a key performance factor for web applications. Various solutions have been built to host web applications at the edge (a geo-area in this work) and use replication and/or caching to improve performance [180–182], however, most approaches simply ignore availability. Some works [183, 184] focus specifically on HAP. In [183], the authors use query caching at the edge to locally process (only) read requests to improve performance while write requests need to be remotely processed at an origin server to ensure consistency. For availability in the presence of an edge server failure, users of failed edge servers are redirected to a healthy one, while in case of origin server failure, core services are replicated across multiple servers. Unlike our work however, they do not consider performance after a failover since the users of the failed edge can be served by another one that is geographically far away. In this work, we replicate application components and data across DCs from different clouds at each edge to improve performance even in the presence of cloud outages. In this model, failover components and data are located in DCs that are near users at that point in time.

In [184], the authors propose a replication service at the edge using application-specific distributed objects which act as a middleware to coordinate data access with each other to maintain consistency. To improve HAP, their approach relaxes the consistency within distributed objects and allows all requests to be processed locally to handle short term, e.g. up to one-minute, network failures. [183, 184] aim to achieve HAP, however they
only focus on reducing delays caused by databases. In contrast, our approach aims to improve the end-to-end response times of the whole web application to achieve HAP.

**Application Deployment in Distributed Clouds.** There have been several efforts aimed at addressing application deployment in distributed Cloud contexts [126, 127, 132, 134, 135]. In [126], the authors aim to improve performance, by minimising user-to-DC and inter-DC latencies to achieve strong consistency for web applications. [135] takes into account performance and service dependencies for service-oriented applications, while the approach in [132] takes into consideration proximity to users only to improve response times using virtual desktops as the example application. [127] considers availability as well as resource utilisation. However, none of these approaches consider both performance and availability or adopt lightweight and flexible container-based deployment models. In this work, we adopt a microservice-based architectural design using containers as a compute deployment model and consider HAP when deploying web applications in distributed clouds.

### 3.3 Application and Container Cluster-based Infrastructure Models

As noted, in this work we focus on session-based web applications that require replication of sessions and data between replicas distributed within a geographical area (geo-area). As shown in Figure 3.1, we assume each pair of cloud DCs form a geo-area where its usage/users shape its boundary and requests are served by application replicas deployed in either DC. We also assume a geo-location DNS (geo-DNS) service, similar to Amazon Route 53 [185], is used as an entry point to route traffic to appropriate cloud resources in a geo-area such that response times to users’ requests should have the least latencies from that geo-area compared to other geo-areas. A failover load balancer (FLB) is used within each geo-area and responsible for several aspects: forwarding requests to primary replicas located in one DC pair member (a primary DC) under normal conditions and to check the health of resources (or the whole DC);
detecting failures or cloud outages, and forwarding requests to backup replicas in other DCs (a backup DC) in the presence of failure or outages.

Moreover, we assume web applications are based on container-based microservices. A web application can consist of a number of microservices, e.g. a web server and a database, and each microservice can be scaled by adding/removing containers. We also assume that a primary-backup model of microservices requiring data replication (e.g. databases and session manager) and primary microservices are deployed in primary DCs and failover DCs as shown in Figure 3.2. A full copy (or multiple copies) of data is assumed to be present at both primary and backup DCs in each geo-area. This is similar to the approach adopted by Facebook [186]. In this model, replication takes place at the application level and is handled by the application and multi-master share-nothing databases, i.e. all database queries issued by a database server in a geo-area can be processed by database nodes co-located in the same geo-area. Eventual consistency of data between geo-areas is assumed.

For the application infrastructure, we consider container-based clustering systems on top of Infrastructure-as-a-Service (IaaS) distributed clouds. These clouds can be public and/or private and each one consists of a variety of DCs from different geo-locations
and allow application providers to provision/terminate virtualized servers (VMs), through APIs however application providers don’t have access to the underlying hypervisors for these cloud. Each one of these servers should have a container engine (e.g. Docker Engine [14]) installed and configured as part of a container-based clustering system (e.g. Docker Swarm [31] or Kubernetes [30]). Finally we assume that the cluster is Cloud-independent, i.e. the cluster manager can add nodes from different Clouds and its scheduler can orchestrate containers across nodes running in different Clouds. This includes supporting auto-scaling of the infrastructure to meet workload fluctuations.

3.4 Problem Formulation

A key challenge for any application deployment that supports failover capability and geo-replication whilst maintaining latency-based SLOs to achieve HAP in distributed clouds is to decide where and by whom application replicas can be placed, and subsequently which techniques to apply to maintain the consistency of replicas. In this work, we focus on the placement problem and not on the replication techniques that may be used. Since we adopt microservices and container solutions, the placement problem itself can be divided into two sub-problems: that of placing servers (i.e. cluster nodes required for the application infrastructure) and that of placing microservices. Server placement problem, also known as DC selection problem, focuses on finding the best DCs to place a server which can host microservices and thus form geographical clusters (geo-clusters) needed for the application infrastructure. Microservice (or service) placement is concerned with finding the optimal servers for placing microservices and their backups, e.g. for redundancy. Each microservice and its backup should be placed in servers deployed in different DCs, however this results in a deployment configuration that raises placement constraints on the cluster scheduler.

The ultimate goal of this work is improve end-to-end response times to meet SLOs under normal conditions and in the presence of outages. This is greatly impacted by network latencies. As illustrated in Figure 3.3, response times of user requests depend on: user-to-primary-DC, inter-DC and user-to-backup-DC latencies. The first two items
affect response times under normal conditions while the third item impacts response times after a failure.

With the increasing number and geo-distribution of DCs, the DC selection problem can be formulated as an optimisation problem where the objective function aims to minimise the overall network latencies between users and DCs and among DCs for requests under normal and failover conditions. With a distributed diverse user base, our approach considers that a primary-backup pair of DCs can serve a cluster of users within a given geo-area. As mentioned in section 3.3, replicas within a geo-area should be kept consistent. The benefit of this approach is to improve response times by reducing geo-replication overheads while HA requirements are considered. For example, replicating user sessions across a set of replicas near users is more efficient than replicating them across replicas in distributed geo-areas. A predefined number of DC pairs can be used to determine the number of geo-areas. Each user will be served by a DC pair with minimal network latency and thus the geo-distribution of users of each geo-area will determine its size.

In detail, the DC selection problem is defined as follows. Using the terms in Table 3.2, given $N, C, U, D$, we want to select a set of DC pairs, $H = \{(DC_1, DC_2), \ldots, (DC_j, DC_k)\}$, where $j \neq k \forall (DC_j, DC_k) \in H$ and $|H| = N$, to place a number of servers ($C$) at each DC in $H$ such that the objective functions under normal (Eq. 3.1) and failover (Eq. 3.2) conditions are minimised, i.e. predefined SLOs are satisfied and the
average user end-to-end response time is minimised even in the presence of outages.

\[
\text{minimise} \quad \sum_i \beta_i \cdot \min_{(DC_j,DC_k) \in H} (nl_{ij}^p + il_{jk}) \quad \forall \ i \in U
\]

subject to \( H \subset A, \ |H| = N \)

(3.1)

\[
\text{minimise} \quad \sum_i \beta_i \cdot \min_{(DC_j,DC_k) \in H} nl_{ik}^b \quad \forall \ i \in U
\]

subject to \( H \subset A, \ |H| = N \)

(3.2)

In Eq. 3.1, \( \min_{(DC_j,DC_k) \in H} (nl_{ij}^p + il_{jk}) \) denotes that the user \( i \) will be served by the primary-backup DC pair \( ((DC_j,DC_k)) \) in \( H \) with the least round-trip-time (RTT) network latency between the user \( i \) and DC \( j \) and between DC \( j \) and DC \( k \) when the user request requires replication, e.g. committing a transaction or updating the user session (i.e. write/update operations). Otherwise, \( il_{ik} \) will be omitted for read operations. Similarly, after a failover \( \min_{(DC_j,DC_k) \in H} nl_{ik}^b \) in Eq. 3.2, denotes that the user \( i \) will be served by the backup DC \( k \) of \( (DC_j,DC_k) \) in \( H \) with the least RTT network latency between the user \( i \) and the DC \( k \). The DC selection problem is NP-hard and can only be solved through heuristics to reduce computation complexity.

Microservice placement takes place after placing cluster nodes in chosen DCs. To address this problem, we need to schedule primary microservices and their backups within each geo-area such that primary and backup microservices are non-dependent at the DC level. That is, using terms in Table 3.2, given a set of primary microservices, \( M^p = \{ \mu^p_1, \ldots, \mu^p_x \} \), and their backups, \( M^b = \{ \mu^b_1, \ldots, \mu^b_x \} \), and \( H \), we want to place \( M^p \) in cluster nodes in \( DC_j \) and \( M^b \) in nodes in \( DC_k, \forall (DC_j,DC_k) \in H \). A service deployment configuration should be generated autonomously considering placement constraints. The resultant solution should subsequently be sent to the cluster manager scheduler to actually deploy microservices.
### Table 3.2: Terms used in the Models

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(D)</td>
<td>Set of available DCs</td>
</tr>
<tr>
<td>(A)</td>
<td>Set of available DC pairs</td>
</tr>
<tr>
<td>(H)</td>
<td>Set of the selected DC pairs for hosting microservices</td>
</tr>
<tr>
<td>(U)</td>
<td>Set of application users</td>
</tr>
<tr>
<td>(G)</td>
<td>Set of geo-areas, servicing different geographical user workloads</td>
</tr>
<tr>
<td>(M^p)</td>
<td>Set of microservices to be placed in a primary DC</td>
</tr>
<tr>
<td>(M^b)</td>
<td>Set of backup microservices to be placed in a backup DC</td>
</tr>
<tr>
<td>(W)</td>
<td>Set of write requests</td>
</tr>
<tr>
<td>(R)</td>
<td>Set of read requests</td>
</tr>
<tr>
<td>(C)</td>
<td>Number of servers (i.e. cluster nodes) to be placed at each DC</td>
</tr>
<tr>
<td>(N)</td>
<td>Number of required primary-backup DC pairs (i.e. geo-areas)</td>
</tr>
<tr>
<td>(g)</td>
<td>Number of generations (i.e. iterations)</td>
</tr>
<tr>
<td>(p)</td>
<td>Number of chromosomes (solutions) in the population</td>
</tr>
<tr>
<td>(r)</td>
<td>Crossover rate</td>
</tr>
<tr>
<td>(m)</td>
<td>Mutation rate</td>
</tr>
<tr>
<td>(\beta_i)</td>
<td>Number of requests made by user (i)</td>
</tr>
<tr>
<td>(V^c_a)</td>
<td>Number of violated sessions at geo-area (a) under condition (c)</td>
</tr>
<tr>
<td>(S^c_a)</td>
<td>Number of sessions at geo-area (a) under condition (c)</td>
</tr>
<tr>
<td>(nl^p_{i,j})</td>
<td>Estimated (RTT) network latency between user (i) and primary DC (j) if that user is served by microservices in DC (j)</td>
</tr>
<tr>
<td>(il_{j,k})</td>
<td>Estimated inter-DC (RTT) network latency between primary DC (j) and backup DC (k)</td>
</tr>
<tr>
<td>(nl^b_{i,k})</td>
<td>Estimated network latency (RTT) between user (i) and backup DC (k) if that user is served by microservices in DC (k)</td>
</tr>
</tbody>
</table>

### 3.5 Proposed Approach

To address the challenges in deploying microservice-based applications in distributed Clouds, we propose an approach to optimise and autonomously generate the deployment while minimising the total amount of estimated SLO violations under normal and failover conditions. The deployment configuration forms an HAP geographical cluster (geo-cluster).

#### 3.5.1 Requirements and Assumptions

As illustrated in Figure 3.4, the approach requires application administrators to provide SLOs for user requests as well as the required number of DC pairs. The approach needs access to trace logs of user requests which can be collected from either a load balancer,
which distributes requests across web servers, or from each web server individually. These request logs contain information about clients, particularly their IP addresses. Such IP addresses can help identify geo-locations of users by using IP-to-Geolocation mapping services such as IP2Location\(^2\). Additionally, the approach requires network latency data between DCs and between users and DCs. Because of the difficulty of obtaining all RTT network latencies between each user and DC, network latency estimators can be used such as the one proposed in [187], to predict unknown latencies or using third party services like Ookla [188]. Information about the location of DCs is also assumed to be provided. Finally, we assume that sufficient Cloud resources are provisioned and that intra-DC communications are neglected since DCs have high-speed (internal) networking capabilities.

3.5.2 The Overall Algorithm

Once an application administrator provides the requirements, as shown in Figure 3.4 the procedure for placing cluster nodes to form a geo-cluster and then deploying microservices across those nodes involves the following steps:

**Step 1. DC Selection.** This initial step in our approach aims to select appropriate DCs for cluster node placement. This step uses a genetic algorithm (GA) to produce a list of selected primary-backup DC pairs which will be sent to the node provisioning and the microservice deployment components. This step is discussed in detail in section 3.5.4.

\(^2\)www.ip2location.com
Step 2. Infrastructure Provisioning. This step is responsible for provisioning the requested servers (VMs) at each chosen DC and joining them to a geo-cluster as cluster nodes as discussed in section 3.3. This step also uses labelling techniques for applying metadata to each cluster node. Metadata describes aspects of a node such as its geo-area and DC type (primary or backup). This information is used by a cluster scheduler to deploy microservices in the right nodes.

Step 3. Microservice Deployment. Once the cluster infrastructure is ready and its nodes are labelled, this step adds placement constraints on microservices and their backups, to ensure that each microservice is scheduled on a node in the right geo-area/DC to meet availability and performance requirements. This step generates the deployment configuration, usually in the form of YAML file, and sends it to the cluster scheduler for deploying microservices across nodes. This step is discussed in more detail in section 3.5.5.

3.5.3 Response Time and Violation Models

In this section, we consider response time models of user requests under normal condition and after a failover occurs as well as SLO-based violations. As discussed, in our model we consider read and write requests of web applications.

Response Times under Normal Conditions

For a typical web application, the response time of a user request can be affected by three main factors: network latency between a user and a primary DC running the application; the processing time for the request and any geo-replication overheads. The geo-replication time is only considered during write requests to keep data replicas consistent within a given geo-area. Geo-replication overheads between geo-areas is not considered here since an eventual consistency model is assumed. Thus, using the terms given in Table 3.2, the response time for write requests under normal condition can modelled as:

\[ \alpha_{w}^{p} = n_{i,j}^{p} + \rho_{w}^{p} + \gamma_{w} \quad w \in W, \; i \in U \]  

(3.3)
where $\alpha^w_n$ is the response time of the write request $w$ under normal condition $n$, $\rho^p_w$ is the time needed to process the request $w$ at the primary DC $p$ and $\gamma_w$ is the geo-replication overhead of that request. $\gamma_w$ depends on the inter-DC network latency between primary and backup DCs and the associated processing time of the request $w$ at the backup DC. $\gamma_w$ can be represented as:

$$\gamma_w = il_{jk} + \rho^b_w \quad w \in W, (DC_j, DC_k) \in H$$

(3.4)

where $\rho^b_w$ is the processing time of the write request $w$ at the backup DC $b$. Therefore, using equations 3.3 and 3.4, the response time of a write request, $\alpha^w_i$, under normal conditions can be represented as:

$$\alpha^w_n = nl_{ij} + \rho^p_w + il_{jk} + \rho^b_w \quad w \in W, i \in U, (DC_j, DC_k) \in H$$

(3.5)

Read requests under normal conditions need only be processed at the primary DC since database replication issues are not a factor. Therefore, a response time for read requests under normal conditions can be derived from Eq. 3.3 where geo-replication overheads can be omitted:

$$\alpha^r_n = nl_{ij} + \rho^p_r \quad r \in R, i \in U$$

(3.6)

where $\alpha^r_n$ is the response time of the read request $r$ under normal condition $n$.

### Response Times after Failovers

In the presence of outages, redundant microservices running in a backup DC take over and handle all subsequent requests. In such a failover condition, the response time of all requests (both read and write) depends on the network latency between users and backup DCs and the time needed by the backup services to process requests. Using the terms in Table 3.2, a response time for a request after a failover can be given as:
\[ \alpha_t^f = n l_{i_k}^b + \rho_t^p \quad t \in R \cup W, \ i \in U \]  

(3.7)

where \( \alpha_t^f \) is the response time of the request \( t \) after a failover \( f \).

Since intra-DC communication overheads can be neglected, improving end-to-end response times and satisfying SLO requirements is primarily dependent on network latencies.

### Violation Model

We propose a user session based model to estimate SLO-based violation rates incurred by each application deployment plan under normal and failover conditions. Specifically, we present a **Session-Based Violation Rate (SBVR)** metric, based on the total number of violated sessions divided by the total number of sessions in geo-areas running primary-backup pairs of application microservices. Using the terms given in Table 3.2, the definition of the model is given as:

\[
\text{minimise} \quad f_{SBVR}(H) = \sum_c \sum_a \frac{V_c^a}{S_a^c} \quad \forall a \in H, c \in \{n, f\} \\
\text{subject to} \quad H \subset A, \ |H| = N
\]  

(3.8)

In this model, we calculate SBVR under two conditions (\( c \)): normal (\( n \)) and failover (\( f \)). Each user session consists of a set of read and/or write requests. A session is considered to be violated when the average response time exceeds predefined SLOs.

### 3.5.4 Genetic Algorithm (GA) for DC Selection

To tackle this, we present a placement algorithm based on genetic algorithms (GA) to solve the DC selection problem. Its goal is to select a set of DC pairs for node placement from a number of geographically distributed DCs to improve end-to-end response times under normal and failover conditions. The GA takes into account two factors: proximity to users and inter-DC network latencies.
3.5 Proposed Approach

A GA is a meta-heuristic based on the process of natural selection. It is commonly used to generate solutions to a variety of problem domains including optimisation solutions. GA has two advantages. Firstly, it is easy to use meta-heuristics to optimise multiple objective functions. In our context, we have two objective functions that need to be minimised since we have two conditions: normal and failover. Secondly, the results produced by the GA are satisfactory and robust in our context as we show in our experiments.

Each candidate solution in a GA is encoded using a data structure known as a chromosome comprised of a set of genes. A typical GA requires two main things: a representation used to encode the candidate solutions from the problem domain, and a fitness function used for ranking those solutions.

A GA typically generates a random population of chromosomes and then iteratively modifies the population using a set of bio-inspired operators, e.g. selection, crossover and mutation. It then uses these to evolve over successive generations. At each iteration (i.e. generation), the GA forms the next generation by probabilistically selecting chromosomes according to their fitness (selection phase) and by adding new chromosomes. These new chromosomes are formed by choosing pairs of the most fit chromosomes from the current population to be parents and then applying a crossover operator on those parents to produce the children in the next generation (crossover phase). Single genes can be mutated resulting in the generation of chromosomes (mutation phase). This process is iterated until a termination condition, e.g. a sufficiently fit solution is discovered or a fixed number of iterations has been reached with no further improvements observed.
Proposed GA in Detail

For the genetic representation of the GA we consider all available DCs ($D$) and generate all possible primary-backup DC pairs ($A$) where each resultant DC pair in $A$ is represented as a gene. A chromosome is encoded as a non-duplicated array of genes, as illustrated in Figure 3.5. The number of genes in a chromosome (i.e. the size of the array) is equal to the number of required primary-backup DC pairs ($N$) provided by the application administrator. Duplicated genes are not allowed in a chromosome, and similarly the order of genes within a chromosome is not important. We enumerate all genes to make the process of detecting duplicate genes more efficient. All chromosomes in the population need to be unique. Additionally, the fitness function of the GA is considered as the complement of the function ($f_{SBVR}$) in the violation model, which can be defined as:

$$fitness(H) = 1 - f_{SBVR}(H) \quad H \subset A, \ |H| = N$$

(3.9)

A GA comprises various phases: initialisation, selection, crossover and mutation. In the initialisation phase, the initial population is randomly generated. In an initial chromosome, each gene is generated stochastically and must be unique in the chromosome. Following this, each chromosome in the population is evaluated and subsequent phases occur after each iteration. For the selection phase, the probability of selecting each chromosome in the population is calculated by dividing its fitness value by the total amount of fitness values of all chromosomes in the current population. The chromosomes are ordered in a descending order according to their selection probability. A fraction of chromosomes with higher selection probability will be selected to be members of the next generation.

In the crossover phase, a fraction of the fittest chromosomes in the current population are selected as parents. Each pair of parents are randomly selected from the resulting fraction and random swaps of their genes are used to produce two offspring. If any one of the resulting children has duplicated genes, mutations will be performed to
3.5 Proposed Approach

prevent duplication. Following this, all offspring will be added to the population of the next generation. In the **mutation phase**, a portion of the population of the next generation is randomly selected. For each chosen chromosome, a gene is randomly selected and mutated by replacing it with a new randomly selected gene. The resultant chromosome is then added to the next population. The mutated gene has to be unique in the chromosome, so this step is repeated until all genes are unique. Following this, all resultant chromosomes in the current iteration are evaluated (**evaluation phase**). The algorithm stops when the fitness value reaches a predefined threshold or it remains the same for a given number of iterations.

**Computational Complexity of Proposed GA**

Using the terms in Table 3.2, the running time of the selection phase requires $O \left( (1 - r) \cdot p \right)$ while the crossover phase runs in $O \left( r \cdot p \right)$. The running time of the mutation phase is in $O \left( m \cdot p \right)$. For the evaluation phase, $p$ chromosomes are evaluated by computing their fitness. Computing the fitness of a chromosome requires finding the best gene (DC pair) in that chromosome for each user in $U$ and this requires $(N - 1)$ comparisons per user. Therefore, evaluating a chromosome requires $O \left( |U| \cdot N \right)$ and hence the evaluation phase will be in $O \left( p \cdot |U| \cdot N \right)$. Based on the running times of the above-mentioned phases, the efficiency of an entire iteration to create a new generation is given by $O \left( \max \left\{ (1 - r) \cdot p, r \cdot p, m \cdot p, p \cdot |U| \cdot N \right\} \right) = O \left( p \cdot |U| \cdot N \right)$. The time complexity of the GA is given by $O \left( g \cdot p \cdot |U| \cdot N \right)$.

3.5.5 Microservice Deployment

Once the GA produces a list of DC pairs ($H$) and a number of required cluster nodes ($C$) are provisioned at each DC and labelled according to their geo-areas and DCs to which they belong, it is necessary to place primary microservices ($M^p$) and secondary microservices ($M^b$) in those nodes such that $M^p$ and $M^b$ of each geo-area are independent at the DC level. This is achieved by applying placement constraints to microservices against the labels (metadata) of nodes when configuring the deployment of those microservices. Placement constraints allow the user/application provider to customise
Algorithm 3.1: Microservice Deployment Generator

Input : List of selected DC pairs, $H$, $M_p = \{ \mu_{p1}, \ldots, \mu_{px} \}$, $M_b = \{ \mu_{b1}, \ldots, \mu_{bx} \}$
Output: Deployment Configuration (YAML file)

/* Define microservices and their backup in each geo-area */

1 foreach $g = (dc_j, dc_k)$ in $H$ do

   /* Define microservices at primary DC $j$ in geo-area $g$ */
   2 foreach $\mu^p$ in $M_p$ do
      3 define $\mu^p$;
      4 add placement constraints geo-area = $g$, dc = $j$, dc_type = primary;
   5 end

   /* Define redundant microservices at backup DC $j$ in geo-area $g$ */
   6 foreach $\mu^b$ in $M_b$ do
      7 define microservice $\mu^b$;
      8 add placement constraints geo-area = $g$, dc = $k$, dc_type = backup;
   9 end
10 end

how the cluster scheduler places containers of microservices to meet their applications’ requirements [71]. As noted, the deployment configuration sent to the scheduler is typically left to the application provider and can be error-prone, especially when the number of microservices and/or required DC pairs are large.

To tackle this, we propose an algorithm for autonomously generating the deployment configuration (Algorithm 3.1). The input of the algorithm are the selected DC pairs, where each pair represents a geo-area as well as the primary and backup microservices. We assume that the application providers provide all requirements for each microservice to run e.g. the container image, the data and the amount of resources (e.g. CPU and memory) . The algorithm works by iterating over all DC pairs in $H$. At each iteration, it defines microservices in $M_p$ and sets placement constraints on them to inform the scheduler on where to deploy them across a subset of cluster nodes belonging to the primary DC ($j$) in the current geo-area ($g$). The same procedure is also performed for redundant microservices ($M_b$) to be deployed in the backup DC ($k$).
3.6 Performance Evaluation

We evaluate our approach by conducting experiments on the Australia-wide National eResearch Collaboration Tools and Resources (NeCTAR) research cloud [46]. We consider two sets of experiments. In the first set, we evaluate failover deployment plans generated by the GA with three other approaches used as baselines using the proposed metric, \( SBVR \). We show the effectiveness of our GA as well as the distributed deployments of applications. In the second set of experiments, we evaluate deployments of the approaches on the NeCTAR cloud and use the end-to-end response time as the evaluation metric.

Settings include our GA and baselines as well as Cloud DCs and workloads are described in the next two sub-sections. The different settings are explained within each experiment set.

GA and Baseline Settings

We refer to our GA approach as GA (proposed) and set crossover and mutation rates to 70% and 50% respectively with the population size set to 75. Analysis of the GA parameter tuning for our problem is discussed in detail in the first experiment set. Moreover, we consider three baseline algorithms: Brute force (optimal), which generates all possible solutions in the solution space and traverses the search space to find the optimal solution; Inter-DC latency unaware (ILU), which only considers proximity to users for the primary DC while the backup DC selection of DC pairs occurs randomly, and Latency unaware (LU), which is unaware of both the proximity to users, and where the primary and backup DC pair selections are selected randomly.

Cloud DCs and Workload Settings

NeCTAR is a distributed IaaS cloud consisting of 19 availability zones distributed across Australia. Each zone can be considered as a DC. For the workload, we select five DCs at
different locations within Australia from which the workload is generated. Their locations are Brisbane, Canberra, Melbourne, Sydney and Tasmania based on the NeCTAR availability zones. In addition to the spatial characteristic of the workload, we set a user session to include 10 requests and categorise workloads into read-intensive and write-intensive workloads. In read-intensive workloads, a user session consists of 7 read and 3 write requests while it has 3 read and 7 write requests in case of write-intensive ones.

3.6.1 Experiment Set 1: Evaluation of our GA

In all experiments in this set, we fix the delay caused by the processing time and all needed internal-DC communications to process a user request within a DC, to $15\,\text{ms}$. This is considered constant in our approach. Also, we consider the system performance ($SBVR$) under the two deployment conditions: normal and failover. In some cases for GA, we use the fitness (%) as the performance evaluation metric. In this experiment set, as discussed we run experiments for each stochastic algorithm and for each GA parameter setting 35 times.

Network Latency and User Settings

In order to collect network latency data, we measure the average round trip time (RTT) between all DCs using the `ping` utility based on multiple pings at different times. Using the resultant data and the geographical distances between these DCs, which are calculated using the Haversine formula, a linear regression model is fitted with a strong positive correlation (coefficient is 0.97). This model is then used to predict the network latencies between users and DCs based on the distances between them. Regarding the users, to generate realistic user request logs, we use Twitter data stored at the above-mentioned locations and extract the geo-locations of users (tweeters) to create the workloads.
3.6 Performance Evaluation

TABLE 3.3: Estimate and standard error (SE) of the average fitness for GA under normal and failover conditions over 35 runs as well as the number of generations for all runs to convergence using different GA parameter settings. The higher the fitness the better.

<table>
<thead>
<tr>
<th>GA Parameters</th>
<th>Generations to Convergence (all runs)</th>
<th>Average Fitness (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Normal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Failover</td>
</tr>
<tr>
<td></td>
<td>Crossover</td>
<td>Mutation</td>
</tr>
<tr>
<td>0.7</td>
<td>0.5</td>
<td>110</td>
</tr>
<tr>
<td>0.5</td>
<td>0.7</td>
<td>137</td>
</tr>
<tr>
<td>0.7</td>
<td>0.7</td>
<td>140</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
<td>146</td>
</tr>
<tr>
<td>0.7</td>
<td>0.2</td>
<td>169</td>
</tr>
</tbody>
</table>

GA Parameter Tuning

GA has two parameters, crossover and mutation that need to be tuned for our application deployment problem. To find the best parameter setting for the problem, we fix the problem-specific parameter settings and run 16 experiments on our GA with different GA parameter settings. These settings are combinations of different values of the crossover and mutation rates. The crossover rate values are 0.2, 0.5, 0.7 and 0.9, while the values of the mutation rate are 0.1, 0.2, 0.5 and 0.7.

Table 3.3 lists the GA parameter settings which are able to converge on the problem over the 35 runs. For each parameter setting, the table shows the number of generations needed for all runs to converge on the problem as well the estimate and standard error of the average fitness under normal and failover conditions over all runs. As seen, it is evident that tuning the crossover and mutation rates to 0.7 and 0.5 respectively shows the fastest rate of convergence and hence better performance. Also, convergence curves shown in Figure 3.6 for the best three parameter settings shows that the chosen parameter setting (Figure 3.6a) is the best one for our problem.

Impact of Network Latency Consideration

It is important to take into account network latencies and consider both the proximity to users and the inter-DC latency when deploying and geo-replicating applications across DCs. In the first experiment, we set the number of required DC pairs to 2 and specify
SLO to 30ms for all request types. We model 1500 users (300 users per location). We set the workload type to read-intensive to generate appropriate requests for user sessions. The GA algorithm and other algorithms are run independently. In the second experiment, we change the workload type to write-intensive and then repeat the steps followed in the read scenario. The third and fourth experiments are similar to the two previous ones, however, the SLO is set to 40ms. As we validate our proposal within an Australia context and the delay caused by the processing time and the internal-DC communications is fixed to 15ms, we set SLOs to small values (i.e. 30ms and 40ms). In real-world applications, SLO values are usually 1 second or more; however, users can
be anywhere in the world and the internal-DC delays of requests can vary.

The results presented in Figure 3.7 indicate that for both workload types and deployment conditions, (unsurprisingly) ignoring network latency results in solutions that incur significantly higher SBVR compared to solutions that support partial or total awareness of network latencies. Considering only user-to-primary-DC latency in the ILU algorithm shows noticeable improvements in SBVR for all cases, since it takes into account both proximity to users and inter-DC latencies. In the GA as well as the optimal algorithm, a substantial improvement is shown. From the results, it is evident that network latency consideration plays a crucial role in improving performance in all
Effectiveness of GA

In terms of execution time, we perform an empirical analysis of the GA. We compare the GA with brute force algorithms. We run each algorithm in a virtual machine instance on NeCTAR with 4 virtual CPUs (vCPUs) and 16 GB of RAM. We set the number of DC pairs to 3. The execution time of the GA is approximately 4 minutes, whilst it takes the brute force algorithm about 26 hours to find the optimal solution. Furthermore, the GA solution has the same SBVR as the optimal one as shown in Figure 3.7. It is noted that the execution time of a GA is the placement time required to generate application deployment plans in distributed Clouds, i.e. it is the time for Step 1. DC Selection discussed in Section 3.5.2). SBVR is subsequently used to evaluate the system performance after deploying the applications, i.e. at run time.

As the GA and the other baseline algorithms (ILU and LU) are stochastic algorithms, we run each one 35 times and then perform statistical analysis. Table 3.4 shows the performance of each algorithm over the 35 runs. The estimates of SBVR for the GA under normal and failover conditions shows very low violation rates. Also, the very narrow confidence interval (CI) and the low values of the standard errors for GA show more precision and stability when compared to the other algorithms.

Moreover, Table 3.5 shows the 95% CI, estimate and p-value of the t-test of the difference in the means of SBVR between the algorithms. Our GA shows improvement in SBVR by at least 55.05% and 75.88% when compared to ILU and LU respectively. Since all values in all intervals are positive and all p-values of differences in SBVR...
3.6 Performance Evaluation

TABLE 3.5: 95% confidence interval (CI), estimate and p-value of the difference in means of SBVR between different algorithms under normal and failover conditions. Each algorithm ran 35 times.

<table>
<thead>
<tr>
<th>Difference in Mean</th>
<th>SBVR (%)</th>
<th>95% CI</th>
<th>Estimate</th>
<th>P-Value</th>
<th>Failover</th>
<th>95% CI</th>
<th>Estimate</th>
<th>P-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILU - GA</td>
<td>(44.50, 65.60)</td>
<td>55.05</td>
<td>0.00</td>
<td></td>
<td>(43.36, 68.02)</td>
<td>55.69</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>LU - GA</td>
<td>(85.44, 94.50)</td>
<td>89.97</td>
<td>0.00</td>
<td></td>
<td>(65.97, 85.79)</td>
<td>75.88</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>LU - ILU</td>
<td>(23.55, 46.29)</td>
<td>34.92</td>
<td>0.00</td>
<td></td>
<td>(4.68, 35.69)</td>
<td>20.19</td>
<td>0.01</td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 3.8: Performance comparison of distributed deployments using various numbers of DC pairs and the optimal single DC pair deployment

means between GA and the other algorithms are 0.00, we can conclude that there are highly statistically significant differences in performance between the GA and other algorithms and thus our GA performs significantly better.

Benefit of Distributed Application Deployments

It is important to consider whether it is worth increasing the distribution of the applications where possible. In this case, we set the SLO to 30ms and run the GA multiple times with different numbers of required DC pairs. We then compare the results with the optimal single DC-pair deployment.
As Figure 3.8 shows, distributing applications across multiple DC pairs is obviously beneficial. Using two DC pairs can reduce the amount of SLO violations to three-quarters of that when using the optimal single DC pair deployment. However, no performance gain is achieved when the number of DC pairs exceeds two.

3.6.2 Experiment Set 2: Evaluation of the Proposed Approach in Real Cloud Contexts

The goal of this experiment set is to evaluate the failover deployment plans generated by the GA in real Cloud context (i.e. the NeCTAR Cloud). We show that the approach can improve performance, reduce pre-agreed SLO violations and maintain performance variability before and after a failover for geographically distributed users whilst realising HAP. Since our work addresses placement issue considering network latencies between users and DCs as well as inter-DC latencies, performance variability here is referred to as the variations between response times of user requests processed by application microservices running in primary DCs (normal condition) and response times when redundant microservices in backup DCs take over (after a failover).

In this experiment set, the assumptions considered are relaxed since the processing time of requests and inner-DC communication costs vary in real Cloud experiments. However, we generate workloads that can be used to evaluate our approach whilst the actual impact of this variation are moderated. We also only discuss GA, ILU and LU approaches, i.e. the brute force one is ignored because our GA and the brute force algorithms produce the same deployment plans.

Experimental Set-up and Spatial Workload Generation

In all experiments, we consider deployment plans generated by GA and other approaches where the number of required DC pairs is set to two and the SLO is specified to 40 ms. We run all experiments during weekdays in a period between 10 a.m. and 12 p.m. to reduce the variations in network traffic and loads in DCs between runs as much as possible. We run experiments for each approach six times. In three runs, we set the
workload type to read-intensive and consider the average result. We do the same in the
other three ones; however, write-intensive workload is set.

Each run lasts for 600 seconds. For each run, the first half of a run is under normal con-
ditions where response times are affected by geo-replication overheads that depend on
inter-DC network latencies. In the middle of the run (i.e. the 300\textsuperscript{th} second), we delib-
erately cause an outage in primary DCs by shutting down all VMs running as cluster
nodes. Then, redundant microservices in backup DCs take over at each geo-area. In
the second half of the experiment, all incoming requests are subsequently forwarded to
the backup DCs.

With regards to workloads, to simulate geo-distributed users, we generate different
workloads from five different locations across Australia. At each location, we provision
a VM (4 vCPUs and 16 GB of RAM) from a Cloud DC located at that location. For each
experiment, we simulate 825 simultaneous users (165 concurrent users per location)
to generate loads. Approximately, 162,000 requests are generated at each run (32,400
requests from each location). Half of those requests are processed before a failover
while the other half are handled after the failover.

**Container Cluster Settings and Sample Application**

We use Docker Swarm [31] as a container-based cluster management system. For our
system infrastructure, we provision a VM to be a manager node for the cluster. Four
VMs are provisioned at each geo-area and join the cluster (i.e. two cluster nodes are in
the primary DC while the other two are in the backup DC). Each VM instance consists
of four vCPUs with 16 GB of RAM. They run Ubuntu 16.04 and have Docker version
18.03 installed.

For application benchmarking, we use a real-world transactional web benchmark (TPC-
W) application [47], which models an online bookstore. Initially, the architecture of
TPC-W application is monolithic, hence we transform the application into microser-
VICES. Consequently, we have three microservices: a web server, a session manager and
a database. We also use etcd (v3.3.9), an open-source distributed key-value store [189]
as a microservice for coordination across a cluster of machines. These microservices are illustrated in Figure 3.2.

We containerise each microservice using Docker. Tomcat 8.5 is used as a stateless web server microservice while a session manager microservice using Couchbase database version 5.5.0 is used to geo-replicate sessions. For the application database, Percona XtraDB Cluster 5.7, an open source, highly available and robust MySQL clustering solution, is used to facilitate the data replication process between cluster nodes distributed across pairs of DCs.

Results and Discussions

The effect of network latency awareness on HAP for geo-distributed users.

Figures 3.9 and 3.10 show the average response times to requests coming from 5 different workload locations of the deployments of the three approaches for read-intensive and write-intensive workloads respectively. It should be noted that deployments during normal conditions (i.e. before failover) have inter-DC latencies of 30% for their requests for read-intensive workloads, while inter-DC latencies are required for 70% of requests for write-intensive deployments.

Figures 3.9a and 3.10a show that our GA, which considers user-to-DC and inter-DC latencies, noticeably improves the response times for all geo-distributed users (i.e. the five locations) before and after a failover for both read and write workload types and thus the goal to make response times that do not exceed the pre-agreed SLO (i.e. 40\(ms\)) is obviously successful. On the other hand, as shown in Figures 3.9b and 3.10b, response times of ILU deployments, where user-to-primary-DC network latencies are only considered, show improvement in performance for all workload locations and types, except the case of the Brisbane workload. Response times to Brisbane workload are high in both conditions and workload types and have violated the SLO since the random selection of backup DCs make the Brisbane workload (spatially) distant from its original DC and thus it incurs significant delays in response times.
3.6 Performance Evaluation

Moreover, with the LU approach, the results presented in Figures 3.9c and 3.10c indicate that response times for most locations are higher and go beyond the SLO for write-intensive workloads while some locations have a degradation in performance for read-intensive workloads. This is because most requests during write-intensive workload deployments require inter-DC latencies for geo-replications. Furthermore, even though the ILU and LU approaches for some locations show performance improvements after failover (e.g. Tasmania in ILU and Brisbane in LU), they fail to improve the performance under normal conditions for all locations.
can conclude that network latency consideration helps applications improve HAP for performance under normal and failover conditions to 25.7% and 22.6% respectively. We other approaches. Also, for write-intensive workloads, our GA has improved the performance by 

Figure 3.10: Performance comparison of 3 deployments for write-intensive workloads generated from 5 different locations around Australia. (SLO=40 ms).

Overall, our GA shows a 23.3% and 25% improvement in response time for read-intensive workloads under normal and failover conditions respectively compared to other approaches. Also, for write-intensive workloads, our GA has improved the performance under normal and failover conditions to 25.7% and 22.6% respectively. We can conclude that network latency consideration helps applications improve HAP for geographically distributed users.
Mitigating the impact of HA overhead on SLOs.

In this section, we show how our approach is able to meet SLOs or minimise SLO violations even in the presence of Cloud outages. Figure 3.11 displays the 95th percentile for the response time under normal and after failover of the three approaches using read and write-intensive workloads. For the read workload, as shown in Figure 3.11a, our GA shows only 0.5\text{ms} violation in response times based on the pre-agreed SLO (40ms) under normal conditions while the SLO is met after failover. On the other hand, SLO violations in ILU and LU approaches are obviously high. The ILU approach has violated the SLO by 14\text{ms} and 12\text{ms} under normal and failover conditions respectively. Furthermore, the LU deployment is worse still and shows 27\text{ms} and 21\text{ms} SLO violations for the 95th percentile of response time before and after failover respectively. It is noted that deployments under the failover condition have less response times than those under the normal condition since failover condition does not have any inter-DC latency overheads.

For write-intensive workloads, Figure 3.11b shows that the 95th percentile of response times for all approaches exhibits more delays in response times in the case of normal conditions compared to the ones for read-intensive workloads in Figure 3.11a. This is because the number of user requests requiring inter-DC latency in write-intensive workload is higher than those of read-intensive one. Overall, in all cases, the 95th percentile of response time in our GA is at most 1.5\text{ms} above the pre-agreed SLO while it is up to at least 11\text{ms} (and up to 32\text{ms}) with the other approaches. We can conclude that considering user-to-DC and inter-DC network latencies can mitigate the overheads caused by HA to help meet SLOs, or at least reduce SLO violations as much as possible, under normal conditions and in the presence of outages.

Maintaining Stability in Application Performance.

In this section, we discuss how our approach can help maintain stability in performance for applications under normal and after failover conditions. We use performance variability before and after failover as an evaluation metric of how stable the
(A) Read-intensive workload

(B) Write-intensive workload

Figure 3.11: The 95th percentile of response time before and after a failover of 3 deployments using various workloads generated from 5 different locations around Australia. For each deployment, approximately 32,400 requests were generated from each location (total 162,000 requests). (SLO=40 ms)

Table 3.6: 95% Confidence Intervals (CI) and estimates of the mean and standard deviation of response times before and after failover of different deployments for different workload locations. Here the workload type is read-intensive and measurements are in milliseconds. (SLO=40ms).

<table>
<thead>
<tr>
<th>Workload Location</th>
<th>GA (proposed)</th>
<th>ILU</th>
<th>LU</th>
</tr>
</thead>
<tbody>
<tr>
<td>95% CI Estimate</td>
<td>Estimate</td>
<td>StDev</td>
<td>95% CI Estimate</td>
</tr>
<tr>
<td>Brisbane</td>
<td>(37.69, 37.83)</td>
<td>37.76</td>
<td>0.83</td>
</tr>
<tr>
<td>Sydney</td>
<td>(25.46, 25.54)</td>
<td>25.50</td>
<td>0.50</td>
</tr>
<tr>
<td>Canberra</td>
<td>(20.12, 20.18)</td>
<td>20.15</td>
<td>0.36</td>
</tr>
<tr>
<td>Melbourne</td>
<td>(12.00, 12.00)</td>
<td>12.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Tasmania</td>
<td>(23.24, 23.31)</td>
<td>23.28</td>
<td>0.45</td>
</tr>
</tbody>
</table>

Application performance is in both conditions. The greater the performance variation, the less the stable the performance. We consider the response times before failover and after failover of each approach to provide the 95% Confidence Intervals (CIs) and estimate the mean and standard deviation of response times for the different workload locations.

Table 3.6 and Table 3.7 list the 95% CIs and the estimates of the mean and standard deviation of the response times for read-intensive and write-intensive workloads respectively. In our GA, the 95% CIs for all locations contain values which are less than the pre-agreed SLO (40 ms) for both workload types. Additionally, the GA has lower standard deviations for all workload locations and types, i.e. the standard deviations
3.7 Conclusions and Future Directions

We have introduced a new approach to help web application providers deploy their applications with HAP requirements in distributed clouds. The approach aims to improve the responsiveness of applications to ensure that they meet SLOs under normal conditions and after failover as well as tackle placement issues. The work utilizes container technologies and a microservice-based application architecture. The approach

<table>
<thead>
<tr>
<th>Workload Location</th>
<th>GA (proposed)</th>
<th>ILU</th>
<th>LU</th>
</tr>
</thead>
<tbody>
<tr>
<td>95% CI</td>
<td>Estimate</td>
<td>StDev</td>
<td>95% CI</td>
</tr>
<tr>
<td>Brisbane</td>
<td>(38.95, 39.11)</td>
<td>39.03</td>
<td>0.99</td>
</tr>
<tr>
<td>Sydney</td>
<td>(27.53, 27.63)</td>
<td>27.58</td>
<td>0.63</td>
</tr>
<tr>
<td>Canberra</td>
<td>(21.79, 21.90)</td>
<td>21.89</td>
<td>1.15</td>
</tr>
<tr>
<td>Melbourne</td>
<td>(12.92, 13.08)</td>
<td>13.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Tasmania</td>
<td>(24.92, 25.08)</td>
<td>25.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

of response times are at most 0.83\textit{ms} (i.e. Brisbane workload) for read workloads and 1.15\textit{ms} or less for the write workload.

On the other hand, the ILU and LU approaches show more variability in performance and fail to make the values of the 95\% CIs for all locations below the SLO. In the case of ILU deployment, the standard deviations of different locations have increased by up to 8.34\textit{ms} and 10.01\textit{ms} (in the case of Tasmania) for read and write workloads respectively. The values of the 95\% CIs for the Brisbane workload for both workload types start from 45.94\textit{ms}.

Moreover, the LU approach shows much more instability. Standard deviations of response times for most locations are substantially higher. The standard deviation of response times for Brisbane, Melbourne and Tasmania are 14.99\textit{ms}, 9.27\textit{ms} and 8.83\textit{ms} respectively for read workload and 19.60\textit{ms}, 7.47\textit{ms} and 5.59\textit{ms} respectively for the write one. We can conclude that our GA shows less variability in performance and thus offers higher performance stability for applications before and after failover.

3.7 Conclusions and Future Directions

We have introduced a new approach to help web application providers deploy their applications with HAP requirements in distributed clouds. The approach aims to improve the responsiveness of applications to ensure that they meet SLOs under normal conditions and after failover as well as tackle placement issues. The work utilizes container technologies and a microservice-based application architecture. The approach
autonomously generates latency-aware failover capabilities by providing deployment plans for microservices and their redundant placement across multiple cloud DCs, with the goal of minimizing the amount of SLO violations.

In our approach, we proposed a user session-based model to estimate SLO violation rates. We presented a genetic algorithm for the DC selection problem that factors in the proximity to users and inter-DC latencies. We also introduced an algorithm for autonomously generating the deployment configuration of the associated microservices. To demonstrate the efficacy of our approach, we conducted experiments on the NeC-TAR Research Cloud using the TPC-W application.

Our future work will focus on addressing issues that can influence HAP of web applications and SLOs during runtime. One problem is the lack of consideration of the dynamic characteristics and geo-distribution of workloads and geo-location of application replicas when scaling the system. This issue can impact the performance and/or availability of solutions. We intend to solve this issue by adopting geo-scaling techniques which can help determine where to scale before deciding how many resources are actually needed.

Another issue for the future is how to handle flash crowds and stochastic volatility in application replicas with high loads in diverse Cloud locations. This can affect HAP in locations and thus impact SLOs. To handle this problem, we are considering geo-load balancing techniques that are aware of the geo-location of application replicas in other DCs. This information can be used to decide where to route user requests such that the amount of SLO violations are minimized.
Chapter 4

Elastic Deployment of Container-based Web Applications across Distributed Clouds

The previous chapter explored approaches for container deployment problems in distributed Clouds to help maintain web application availability and performance through container technologies. It considered proximity to potentially globally distributed users and factored in any associated inter-data center latencies that might arise to generate latency-aware failover deployment plans. This is only part of the problem however. Other latency-related container deployment problems that need to be considered are the lack of intelligent, elastic, global deployment capabilities of container clusters across geographically distributed Cloud data centers to address spatial workloads of web applications and their subsequent management. As discussed in Section 2.6, such capabilities are essential to maintain web application performance and adapt the deployment of clusters based on proximity to users whenever such spatial workload changes occur.

This chapter presents an approach that focuses on making intelligent adaptation of multi-cluster deployments in distributed Clouds to manage such spatial workload fluctuations. It explores the trade off in cost and performance to maintain QoS and SLOs including during the adaptation process. This chapter addresses the Research Problem 2 discussed in Section 1.3.

Specifically in this chapter we propose a framework to enable automated multi-cluster deployments. We present an elastic deployment technique that dynamically makes intelligent deployment plans to optimise the number and placement of clusters whilst considering proximity to users and the cost of adaptation. To tackle the cluster placement problem, we explore approaches based on genetic algorithms. For adjusting the quantity of clusters, a heuristic based solution is proposed. To evaluate the approach, we conduct extensive experiments using case studies based on Docker and Kubernetes-based clusters on the NeCTAR Research Cloud.

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

Yasser Aldwyan\textsuperscript{a,b}, Richard O. Sinnott\textsuperscript{a}, Glenn T. Jayaputera\textsuperscript{a}

\textsuperscript{a}School of Computing and Information Systems, The University of Melbourne, Australia
\textsuperscript{b}Faculty of Computer and Information Systems, Islamic University of Madinah, Saudi Arabia
yaldwyan@student.unimelb.edu.au, rsinnott@unimelb.edu.au, glenn.jayaputera@unimelb.edu.au

Abstract

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 95\textsuperscript{th} 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.

4.1 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 [37–39]. They provide application packaging that allows consistent, portable deployment across multiple Clouds [26] as well as abstracting away many of the overheads when deploying and managing containers and infrastructures [16, 26]. 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., Kubernetes [30], that is used to deploy, manage and scale containers across Cloud resources. Container management platforms, such as Google Anthos [146] and Rancher [44], take all management responsibilities to automate the deployment and simplify the management of such clusters and containers across Clouds [37]. 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 [39, 146, 190]. The multi-cluster 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 [191], including application containers, to data centers, e.g., they should be deployed in proximity to end users to maintain performance as shown in Figure 4.1.
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 [8]. 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 [8, 192, 193], which ultimately has negative impacts on businesses with poor user experience resulting in lost revenues [49] 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 [52], 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 [38], 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 [151, 153, 155, 194], clusters [162, 195, 196], or both [158, 197]. Other work proposes techniques to handle geo-workload variations [8, 126], 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 4.2, we cover related work. Section 4.3 describes the application and the container deployment models that are adopted as well as provides the problem definition. Section 4.4 discusses the proposed solution. We evaluate the proposed approach in Section 4.5. Finally in Section 4.6 we provide conclusions and identify potential future research directions.

4.2 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 [198] and Kubespray [199] 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 [144]. Moreover, migration solutions can relocate container clusters across data centers, either via rescheduling [191] or live migration [200, 201]. Container management platforms, like Rancher [44], Google Anthos [146] and OpenShift [147] make container clusters easier to deploy in distributed clouds [37]. 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 [53]. 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 [151, 153, 155, 194], cluster deployment [162, 195, 196] or
4.2 Related Work

both [158, 197]. These approaches use horizontal methods [151, 162, 196, 197], vertical methods [155, 158] or hybrid approaches [153, 194, 195] 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 [185] and Azure traffic manager [82] can distribute load to different Cloud data centers based user geo-locations to reduce latency and other factors such as energy savings [83, 85]. 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 [86]. 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 [89], 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 [88] 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 [202] 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 [87] 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 [119], 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 [88].

In the context of distributed Cloud and web applications, solutions such as [8, 126] 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 [8] assumes cross-data center eventual data consistency whereas the solution in [126] targets web applications requiring strong consistency between data centers. Furthermore, the solution in [126] considers the number of cross-data center application relocations as a cost of adaptation that should be minimised while [8] 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 [7], compared to heavyweight VM-based models used in [8, 126]. 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 [8]. However, they require determination of new potential
future Cloud locations in advance and periodically copying VM images which lim-
its the capabilities of deployment optimisation techniques to choose Cloud locations
other than pre-selected ones when needed. Also, by using containers such deploy-
ment techniques can utilise more Cloud locations as they provide consistent, portable
deployment of applications across data centers regardless of the underlying Cloud in-
frastructure. Container images provide an abstraction that can isolate the application
environment from the underlying deployment infrastructure [26]. Container-based so-
lutions also provide a cross-Cloud overlay networking that facilitates the process of
geo-replication and migration. Thus container-based solutions have many direct ben-
efits for elastic deployment demands compared to historic Infrastructure-as-a-Service
solutions.

**Application Placement in Distributed Computing Environments.** Placement solu-
tions to tackle latency management have been studied in different distributed comput-
ing environments. A solution in [119] 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 [131] propose a dynamic placement solution
for IoT requests, aimed at reducing task latency times and system power consumption.
Also, work in [89] 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 prox-
imity to users, e.g., [8, 132, 203], inter-data center latency issues, e.g., [134] or both [126,
135, 192]. Work has explored place virtual desktops [132] and service-oriented solutions
[135, 203], web solutions [8, 126, 192] or diverse [134] applications across data centers.
Works have explored either a static [134, 192] or dynamic [8, 126, 132, 135] deploy-
ments using VMs [8, 126, 132, 134, 135, 203] or containers [192] to achieve inter-data
center strong data consistency and performance [126]. Other works have considered
high availability and performance even after complete or partial Cloud outages [192]
factoring in budget and performance issues [203] or performance issues alone, e.g., [8,
132, 134, 135].
Dynamic placement (replacement) solutions such as [126, 135] 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.

4.3 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.

4.3.1 Assumptions for Container and Application Deployment Models

As shown in Figure 4.1, 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 [82] 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\(^1\). These can associate user requests to the nearest cluster.

In this work, we assume each cluster runs a full copy of containerised web applications as this model enables an application to scale globally [39], 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

\(^1\)Identifying Geographical Location by IP Address https://www.ip2location.com
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 [14] 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.

4.3.2 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.
4.3 Problem Formulation

Table 4.1: Terms Used in the Models

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>current</td>
<td>Current deployment of container clusters</td>
</tr>
<tr>
<td>n_clusters</td>
<td>Size of the current deployment</td>
</tr>
<tr>
<td>cand</td>
<td>New deployment candidate</td>
</tr>
<tr>
<td>A</td>
<td>Set of available data centers</td>
</tr>
<tr>
<td>D</td>
<td>Set of the data centers of the current deployment (current)</td>
</tr>
<tr>
<td>F</td>
<td>Set of selected data centers of the candidate deployment (cand)</td>
</tr>
<tr>
<td>K</td>
<td>Set of new and/or relocated clusters</td>
</tr>
<tr>
<td>W_t</td>
<td>Workloads collected at time t</td>
</tr>
<tr>
<td>N</td>
<td>Number of required clusters for the new deployment</td>
</tr>
<tr>
<td>S_j</td>
<td>Number of sessions in data center j at time t</td>
</tr>
<tr>
<td>V_j</td>
<td>Number of violated sessions in data center j at time t</td>
</tr>
<tr>
<td>n_{i,j}</td>
<td>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</td>
</tr>
<tr>
<td>il_{j,k}</td>
<td>Estimated inter-data center (RTT) network latency between data center j and data center k (migration)</td>
</tr>
<tr>
<td>U_{thr}</td>
<td>Upper bound of acceptable Session Based Violates Rates (SBVR)</td>
</tr>
<tr>
<td>G_{thr}</td>
<td>Gain threshold to be considered an improvement in SBVR when SBVR is below U_{thr}</td>
</tr>
<tr>
<td>τ</td>
<td>Consecutive periods for cool down</td>
</tr>
<tr>
<td>T</td>
<td>Time interval</td>
</tr>
<tr>
<td>P</td>
<td>Pause interval</td>
</tr>
</tbody>
</table>

\[
\text{maximise} \quad f (\text{current}, \text{cand}, W_t) = \frac{R (\text{current}, \text{cand}, W_t)}{\text{adapt} \_ \text{cost} (\text{current}, \text{cand}, W_t)}
\]

subject to \( F \subset A, \ |F| = N \)  

(4.1)

where \( R (\text{current}, \text{cand}, W_t) \) is the estimated amount of reduction in the violation rate when changing the deployment (current) to the candidate one (cand) and where \( \text{adapt} \_ \text{cost} (\text{current}, \text{cand}, W_t) \) 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 4.1 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, \( F = \{ \text{data\_center}_1, \text{data\_center}_2, \ldots, \text{data\_center}_n \} \), where \( |F| = N \), to (re)place container clusters across data centers in \( F \) such that the objective function in Eq. 4.1 is maximised.
4.4 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 \((n_{\text{clusters}})\) 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 \((\text{cand})\). 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.

4.4.1 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
4.4 Proposed Geo-elastic Deployment Approach

FIGURE 4.2: A framework of enabling automated elastic deployment of container clusters in geographically distributed Clouds

data can be obtained using third party services, e.g., Ookla [188], or by latency estimators to estimate latencies between users and data centers, e.g., [187], 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.

4.4.2 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 underutilised. 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.

4.4.3 Geo-elastic Deployment Framework

As shown in Figure 4.2, 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 4.4.4.

This component implements a decision-making algorithm for the geo-elastic deployment controller mechanism, which is discussed in more detail in section 4.4.5. 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 4.4.6 and 4.4.7.

**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\(^2\) and jclouds\(^3\) are examples of technologies that can be used to manage Cloud resources.

\(^2\)Apache Libcloud https://libcloud.apache.org
\(^3\)Apache jclouds https://jclouds.apache.org
from different Cloud providers using a unified API. For Kubernetes, automation tools like Kops [198] or Kubespray [199] 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 georeplicates 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., [177, 201, 204].

### 4.4.4 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 performance of a given deployment. Using terms defined in Table 4.1, the violation model is defined as follows:

\[
 f_{SBVR}(X, W_i) = \frac{\sum j V_j^i}{\sum j S_j^i} \quad \forall j \in X
\]  

(4.2)
At a given point in time, \( t \), and given a deployment (i.e., the container clusters to be deployed at data centers in \( X \)) 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 \( X \) 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.

### 4.4.5 Multi-cluster Geo-elastic Deployment Controller Algorithm

In this section, we present a decision-making algorithm (Algorithm 4.1) 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 4.1, 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 resulting 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 4.4.7 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 ($\tau$ 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.
Algorithm 4.1: Decision-making Algorithm for the Multi-cluster Geo-elastic Deployment Controller

**Input**: \( \text{init_deploy} \)

1. \( \text{current} = \text{init_deploy}; \)
2. \( n\_\text{clusters} = \text{size}(\text{init_deploy}); \)
3. \( \text{paused} = \text{False}; \)
4. **for every** \( T \) **do**
5.   **if** \( \text{paused} \) **is** True **then**
6.     **pause** \( (P) \);
7.     \( \text{paused} = \text{False}; \)
8. **end**
9. /* Get clusters workloads at time \( t \) */
10. \( W_t = \text{get\_workloads}(\text{current}); \)
11. /* Geo-shrinking decision */
12. **if** \( f_{SBVR}(\text{current}, W_t) < U_{thr} \text{ for } \tau \text{ then} \)
13.     \( \text{cand} = \text{replace\_clusters}(n\_\text{clusters}-1, \text{current}, W_t); \)
14.     **if** \( f_{SBVR}(\text{cand}, W_t) < U_{thr} \text{ then} \)
15.         /* To deployment executor */
16.         \( \text{redeploy\_clusters}(\text{cand}); \)
17.         \( \text{current} = \text{cand}; \ \text{paused} = \text{True}; \)
18.         \( n\_\text{clusters} -= 1; \)
19.         **continue;**
20. **end**
21. **end**
22. /* Geo-relocation decision/same size */
23. \( \text{cand} = \text{replace\_clusters}(n\_\text{clusters}, \text{current}, W_t); \)
24. \( \text{relocate} = \text{False}; \)
25. **if** \( f_{SBVR}(\text{current}, W_t) \geq U_{thr} \text{ and} \)
26.     \( f_{SBVR}(\text{cand}, W_t) < U_{thr} \text{ then} \)
27.     \( \text{relocate} = \text{True}; \)
28. **end**
29. **if** \( f_{SBVR}(\text{current}, W_t) < U_{thr} \text{ and} \)
30.     \( (f_{SBVR}(\text{cand}, W_t) - f_{SBVR}(\text{current}, W_t)) \geq G_{thr} \text{ then} \)
31.     \( \text{relocate} = \text{True}; \)
32. **end**
33. **if** \( \text{relocate} \) **is** True **then**
34.     \( \text{redeploy\_clusters}(\text{cand}); \)
35.     \( \text{current} = \text{cand}; \ \text{paused} = \text{True}; \)
36.     **continue;**
37. **end**
38. /* Geo-expanding decision */
39. **if** \( f_{SBVR}(\text{cand}, W_t) > U_{thr} \text{ and} \)
40.     \( f_{SBVR}(\text{current}, W_t) > U_{thr} \text{ then} \)
41.     \( \text{cand} = \text{replace\_clusters}(n\_\text{clusters}+1, \text{current}, W_t); \)
42.     \( \text{redeploy\_clusters}(\text{cand}); \)
43.     \( \text{current} = \text{cand}; \ \text{paused} = \text{True}; \)
44.     \( n\_\text{clusters} += 1; \)
45. **end**
46. **end**
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).

### 4.4.6 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 4.3. 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 4.1, the cost of an adaptation function that can be used as the
denominator of the objective function in Eq 4.1 is defined as:

$$adapt\_cost = 1 + |K| + (1 - \frac{1}{1 + \sum_{DC_j \in D} \min \{il_{jk}\}})$$

$$\forall k \in K, K \subset F$$

where $|K|$ is the number of new and/or relocated clusters, $DC_j$ is a data center $j$ and
$1 - \frac{1}{1 + \sum_{DC_j \in D} \min \{il_{jk}\}}$ 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 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 4.2, 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 4.1 and represented as follows.

\[
R = \begin{cases} 
U_{thr} - f_{SBVR}(cand, W_t) & \text{if } f_{SBVR}(current, W_t) \geq U_{thr} \\
 & \text{or } |cand| < |current| \\
- f_{SBVR}(cand, W_t) & \text{otherwise} \\
\end{cases} 
\]

(4.4)

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 4.4 since ignoring one of them can lead to improper elastic decisions being made under certain conditions. Consider the following example. Using Eq. 4.1 and 4.3 and assuming the inter-data center network latency is constant at 1, the objective function can be presented as $f = \frac{R}{1+|K|}$. 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 4.1). That is, in Algorithm 4.1, 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 4.1 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 [205] that are hard to solve [206, 207]. 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 [205], 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 4.4.2) and the predefined, constant $U_{th}$ 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 $K$ be zero in Eq 4.3 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 4.4 and the $U_{th}$ is constant, then $R = -f_{SBVR}(cand, W_t)$. By applying the last two steps to adapt\_cost and $R$ in Eq 4.1 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, $A$ where $N$ data centers are to be chosen from $A$ 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 [208]. 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 [126, 127, 135, 192, 209]. 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 [210]. 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 4.5.

4.4.7 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. 4.1 is used to select an individual (i.e., a candidate deployment, $\text{cand}$) with the highest fitness. A fitness function is presented as:

$$fitness (\text{current}, \text{cand}, W_t) = \max f (\text{current}, \text{cand}, W_t)$$

(4.5)

A genetic algorithm-based cluster replacement algorithm is shown in Algorithm 4.2. Using the symbols defined in Tables 4.1 and 4.2, 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, $\text{current}$, 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).
Algorithm 4.2: Genetic Algorithm for Cluster Replacement

**Input**: \( N, \text{current}, W_t \)

1. Set the number of data centers (genes) of a candidate deployment to \( N \);

2. /* Initialise population */

3. \( P \leftarrow \text{Generate} \ p \ \text{candidate deployments (cands) randomly using data centers in A;} \)

4. /* Evaluate */

5. For each cand in \( P \), compute fitness(cand) using Eq. 4.5;

6. Sort cands in \( P \) by their (descending) fitness;

7. \( \text{cand}_{\text{best}} \leftarrow \text{first cand in } P; \)

8. \( g = 1; \)

9. while \( g < g_{\text{max}} \) do

   /* Create a new generation \( P_{\text{new}} \) */

   10. \( P_{\text{new}} \leftarrow \emptyset; \)

   /* Apply selection */

   11. Select \((1 - r) \cdot p\) cands from the start of \( P; \)

   12. Add selected cands to \( P_{\text{new}}; \)

   /* Apply Crossover */

   13. Select \( \frac{r \cdot p}{2} \) pairs of cands from the start of \( P; \)

   14. 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 \( c2; \)

   15. Add all children to \( P_{\text{new}}; \)

   /* Apply mutation */

   16. Choose \( m \cdot p \) cands randomly from \( P_{\text{new}}; \)

   17. For each chosen cand, replace a randomly selected data center of cand with a randomly selected data center from A;

   /* Update */

   18. \( P \leftarrow P_{\text{new}}; \)

   /* Evaluate */

   19. For each cand in \( P \), compute fitness(cand) using Eq. 4.5;

   20. Sort cands in \( P \) by their (descending) fitness;

   21. if fitness(cand\text{best}) < fitness(first cand in \( P \)) then

   22. \( \text{cand}_{\text{best}} \leftarrow \text{first cand in } P; \)

   end

23. Increment \( g \) by 1;

24. Return \( \text{cand}_{\text{best}} \) that has the maximum fitness;

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, \( P_{\text{new}} \) (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, \( P_{\text{new}} \) (Lines 9-10). The crossover operator is then applied (Lines 11-13). This operator selects \( \frac{r \cdot p}{2} \) pairs of candidate deployments
4.4 Proposed Geo-elastic Deployment Approach

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, \( P_{\text{new}} \).

The algorithm then applies the mutation operator (Lines 14-15) by randomly choosing \( m \cdot p \) candidate deployments from the new population, \( P_{\text{new}} \), 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, \( g_{\text{max}} \). The algorithm returns the best candidate deployment, \( \text{cand}_{\text{best}} \), 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 4.1 and 4.2. 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 \( O(g \cdot p \cdot |W_t| \cdot N) \).
4.5 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 contain-
ers 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 cen-
ter, 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. Com-
mon settings among all experiment sets are explained in the next section. Individual
settings are described separately within each experiment set.

4.5.1 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 ac-
curately 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 candi-
date 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 dis-
tributed 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.

**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 and due to the lack of real geographic workload variations within Australia, we synthesised a series of consecutive geo-workload variation scenarios as shown in Figure 4.3 and Table 4.3. 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. We empirically found this length to be suitable to validate the overall performance of the proposed technique. 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 4.4c, 4.4d and 4.6b. 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 \( T, P, \tau, U_{thr}, G_{thr} \) were set to 2 min, 20 min, 1 period, 10% and 3% respectively. These parameters are application-specific and they are either predetermined by the application administrator depending on the type of the application or pre-agreed based on SLOs in service level agreements. 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 4.5.4. 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 [8] as it only considers network latency between users and data centers and completely ignores inter-DC latency. Furthermore, work in [126] 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 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 those 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 [192] 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).

**4.5.2 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).

**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 4.3 and Table 4.3 are generated depending on the number and locations of geo-workloads over that period.

**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 4.3 and Table 4.3. 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.
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

![Graph A](image1.png)
![Graph B](image2.png)
![Graph C](image3.png)
![Graph D](image4.png)

**Figure 4.4:** 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.

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.

**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 4.4a and 4.4b 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 ($U_{th} = 10\%$) 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 4.4c and 4.4d, 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 4.4d 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 4.4.6, 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 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 role in improving performance under cost constraints.

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

![Diagram](image-url)

**Figure 4.5:** 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.
4.5 Performance Evaluation

Figure 4.5, 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.

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 4.5a. When the SLO is relaxed as shown in Figure 4.5b, 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 4.5c and 4.5d, 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.

4.5.3 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).

**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%.
4.5 Performance Evaluation

Realistic Workload Generation and Request Routing

To generate realistic workloads from the five locations discussed in Section 4.5.1, 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 4.3 and Table 4.3) 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 [47], 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) [14] as a container Runtime and Kubernetes (v1.16.3) [30] 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 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.

**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.

**Results and Discussions**

**Maintaining Performance at Lower Cost**

Figure 4.6 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 4.6a 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 4.6b 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 other words, costs are reduced by 33% at periods (2 and 5) and by 50% at periods (1, 3, 4 and 6).

Table 4.4 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.

<table>
<thead>
<tr>
<th></th>
<th>Difference in Response time Mean in ms</th>
<th>Difference in Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>95% CI</td>
<td>Estimate</td>
</tr>
<tr>
<td>Geo - Static</td>
<td>(-3.24, -3.07)</td>
<td>-3.16</td>
</tr>
<tr>
<td>OP - Static</td>
<td>(-5.95, -5.81)</td>
<td>-5.89</td>
</tr>
<tr>
<td>Geo - OP</td>
<td>(2.68, 2.77)</td>
<td>2.72</td>
</tr>
</tbody>
</table>
4.5 Performance Evaluation

<table>
<thead>
<tr>
<th>GA parameters</th>
<th>Iterations to Convergence (all runs)</th>
<th>Average Fitness</th>
<th>Average SBVR (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Estimate</td>
<td>SE</td>
</tr>
<tr>
<td>Crossver</td>
<td>Mutation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.7</td>
<td>0.2</td>
<td>-0.89</td>
<td>0.02</td>
</tr>
<tr>
<td>0.5</td>
<td>0.1</td>
<td>-0.9</td>
<td>0.02</td>
</tr>
<tr>
<td>0.7</td>
<td>0.1</td>
<td>-0.87</td>
<td>0.02</td>
</tr>
<tr>
<td>0.7</td>
<td>0.7</td>
<td>-0.85</td>
<td>0.02</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
<td>-0.88</td>
<td>0.02</td>
</tr>
</tbody>
</table>

4.5.4 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.

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).
FIGURE 4.7: 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%).

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 4.5 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 4.7a.
4.5 Performance Evaluation

Table 4.6: 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.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Objective function value</th>
<th>SBVR (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>95% CI</td>
<td>Estimate</td>
</tr>
<tr>
<td>GA</td>
<td>(-0.96, -0.82)</td>
<td>-0.89</td>
</tr>
<tr>
<td>LU</td>
<td>(-19.22, -17.21)</td>
<td>-18.22</td>
</tr>
</tbody>
</table>

Group 2: Effectiveness of Genetic Algorithm

Proposed Algorithms and Baselines Settings

Genetic algorithm parameter settings were discussed in section 4.5.1. 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 4.6 indicates that the latency unaware algorithm shows a very high violation rate in SBVR, 90.41%, while our genetic algorithm, which takes into account network latency, has very low SBVR violation rates, 7.39%. As shown in Table 4.7, 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.
TABLE 4.7: 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.

<table>
<thead>
<tr>
<th>Difference in Mean</th>
<th>Objective function value</th>
<th>SBVR (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GA - LU</td>
<td>95% CI</td>
<td>Estimate</td>
</tr>
<tr>
<td>(16.29, 18.37)</td>
<td>17.33</td>
<td>0.00</td>
</tr>
</tbody>
</table>

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 4.7a shows, the genetic algorithm converges at the 54th iteration.

With respect to accuracy, as shown in Tables 4.6 and 4.7, the distance for our genetic algorithm to the optimal value is only 1.79%. Moreover, as indicated in Figure 4.7b, 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.

4.6 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.
Chapter 5

Cost-oriented Scaling of Container-based Web Applications in Distributed Clouds

The previous chapters have addressed problems of container orchestration to manage the deployment of containerised web applications in distributed Cloud environments. They consider network latencies and spatial aspects of workload behaviour of web applications to maintain QoS, SLOs whilst meeting cost requirements. This chapter focuses on problems of container elasticity for multi-cluster container-based web applications deployed across distributed data centers with focus on timely handling of sudden local workload spikes and avoiding cluster overloads. Solutions that realise multi-cluster deployments are essential to provide cost-effective and rapid scaling of overloaded clusters leveraging global computing capacity now available through major international Cloud providers. Solutions need to consider cluster auto-scaling at both the container and VM levels based not only on the current application demand but also the current performance of distributed and heterogeneous Cloud resources. This is the focus of this chapter.

Specifically this chapter presents a cost-aware container-based elasticity approach for bursty multi-cluster containerised web applications in geographically distributed Clouds. We show how it is possible to scale suddenly overloaded cluster and avoid the Cloud overheads when provisioning new VMs via inter-cluster resource utilisation and auto-scaling of clusters. The aim here is to address the Research Problems 3 discussed in Section 1.3.

To support this we present an architectural framework that supports inter-cluster resource management. This utilises queueing-based performance models to accurately estimate container capacity; two-level cluster elasticity techniques for horizontal autoscaling of both containers and VMs and offering dynamic container scheduling policies that consider inter-data center placement demands. We demonstrate the effectiveness of the approach through extensive experiments on the NeCTAR Research Cloud and the Amazon Cloud.

Cost-oriented Scaling of Container-based Web Applications in Geographically Distributed Cloud Data Centers

Yasser Aldwyan\textsuperscript{a,b}, Richard O. Sinnott\textsuperscript{a}, Glenn T. Jayaputera\textsuperscript{a}

\textsuperscript{a}School of Computing and Information Systems, The University of Melbourne, Australia
\textsuperscript{b}Faculty of Computer and Information Systems, Islamic University of Madinah, Saudi Arabia
yaldwyan@student.unimelb.edu.au, rsinnott@unimelb.edu.au, glenn.jayaputera@unimelb.edu.au

Abstract

Container management platforms such as Kubernetes have been rapidly adopted by Cloud communities for deploying and elastically scaling containerised web applications on demand across clusters of virtual machines (VMs) based on workload changes. Although such clusters are able to adapt and provide resources required for container scaling, they can be still overloaded and run out of resources, e.g., due to sudden application workload spikes. In such scenarios, new containers cannot be launched in a timely manner, i.e., before the often considerable time required to provision new VMs. This can hinder the scaling out of containers resulting in performance/availability issues. Multi-cluster container-based application deployment across geographically distributed data centers is increasingly needed to improve end user experiences. In this model rather than over-provisioning each Cloud-based cluster individually to optimise container launch latency and handle overload situations, it is cost-efficient to temporarily utilise idle, already-running resources from other under-utilised clusters as workloads may vary in different locations. In this paper we propose a cost-efficient elastic scaling approach for such multi-cluster models that handles cluster overloads in a timely manner, both during decision making and provisioning times using inter-cluster resource utilisation. Our approach includes a two-level cluster elasticity technique that auto-scales both containers and VMs, using queuing theory-based container capacity models based on service level objectives (SLOs) and high-level metrics. This factors in inter-data center latency-aware container scheduling policies to dynamically prioritise the actual placement of overloaded containers. The work is evaluated using the Australian National Research Cloud (NeCTAR) and Amazon Clouds. Results show that our approach is able to maintain performance and SLOs in overloaded clusters using a lower number of resources and hence achieve a more cost-effective solution.

5.1 Introduction

Cloud-based web application developers and infrastructure providers have rapidly adopted container technologies such as Docker [14]. These offer a lightweight virtualization technology along with container management/orchestration support offered
by solutions such as Kubernetes [30]. Such technology can efficiently deploy and elastically scale container-based applications on demand across clusters of virtual machines (VMs) as workload (dynamically) changes. To benefit from Cloud scalability and elasticity features, container solutions need to adapt both the number of containers and potentially also the number of VMs they run on. However, in some cases clusters can be still overloaded and run out of resources impacting on the creation and deployment of new containers. While container start-up time takes only a few seconds, e.g. for Docker it is around one second [211], VM provisioning in Clouds can take considerably more time. An experimental study on VM start-up time using multiple Clouds found that the creation of a VM can range from about 50 seconds to over 900 seconds [18]. Such delays, especially in overload situations, give rise to launch latency for pending containers as a VM needs to be provisioned before containers can be launched. This can result in application performance issues, availability issues, and service level objective (SLO) violations.

In web applications, a common characteristic of workloads is sudden burstiness, i.e., unexpectedly drastic increase in user request rates within a few seconds [5]. This is often known as flash crowds. In such scenarios, clusters can become overloaded and need to scale containers immediately. Basic Cloud-based elasticity approaches based on creation of VMs cannot tackle manage such sudden overload situations. Furthermore, web workloads can be highly variable and demand fine-grained timescales which may lead to inaccurate scaling decisions and/or give rise to inefficient Cloud elastic scaling [5]. Hence, merely depending on basic Cloud elasticity solutions for dynamic workload characteristics is more likely to be inaccurate and maintaining acceptable levels of performance and SLOs all of the time will be difficult or impossible.

As Cloud data centers have been increasingly distributed around the world, global multi-cluster containerised application deployment models are becoming more commonplace. Improving user experience can require each cluster to serve nearby users as depicted in Figure 5.1. In this deployment model, we argue that for managing overloaded clusters in a cost-effective and timely manner, it is better to temporarily utilise idle, already-running VMs from other underutilised clusters deployed in different data centers to provide overloaded clusters with enough resources in a timely fashion and
thus quickly reduce launch latency for new pending containers. This approach is especially suited for web applications in overload situations since web burstiness is unlikely to occur simultaneously at two globally distributed clusters due to time and event differences [88]. Such an approach would optimise the overall deployment of potentially costly Cloud resources while maintaining application performance and availability to avoid/minimise violations of SLOs.

A common approach to deal with such overload situations is to over-provision resources at each Cloud-based cluster individually. This approach avoids such situations from arising as it constantly provides each cluster with extra capacity to enable a container platform to launch containers immediately even in the presence of sudden increased traffic. However this approach is at the cost of running more VMs than applications require most of the time for each cluster. Another approach is inter-cluster resource utilisation. This allows utilisation of resources from multi-tenant, shared clusters in the Cloud to deploy pending containers that cannot be launched in an overloaded cluster. For instance, an overloaded Kubernetes cluster can quickly run pending containers at an AWS Fargate shared cluster in an Amazon Cloud data center [212]. This is exposed as a virtual node with a given compute capacity based on use of Virtual Kubelet technology [213]. To improve isolation and security for containers in such
shared clusters, Cloud providers have also put forward lightweight isolation technologies such as AWS Firecracker [214] and Google gVisor [215]. This approach provides faster deployment with improved (reduced) cost-efficiency as additional resources are only used when they are actually needed, e.g., during flash crowds. However, these new lightweight isolation technologies can still introduce performance degradation for applications running in those shared infrastructures [216]. Furthermore, these approaches handle overload situations for each cluster individually and only within the same data center. Solutions that can benefit from multi-cluster deployments across data centers to handle overload situations is a key contribution of this paper.

In this paper, we propose a cost-effective container-based elasticity approach for multi-cluster containerised web applications deployed across geographically distributed Cloud data centers. Our approach aims to quickly and temporarily manage overload situations through inter-cluster resource utilisation to minimise launch latency of containers to timely handle flash crowds using the minimum number of resources while providing elasticity techniques offering two key benefits. Firstly, it gives them sufficient time to decide whether to provision new VMs locally in a more accurate manner within a decision-making period, to mitigate the impact of potential burstiness. This time period is configurable and can be used to fit each web application’s needs individually based on its own workloads. Secondly, it factors in inevitable VM provisioning delays when scaling out at the VM level. During such periods, the approach uses scheduling policies to dynamically move containers with less load to remote nodes (i.e., nodes running remotely at different data centers) to free up local resources. This considers the impact of inter-data center network latency on the overall application performance. Once a new local VM joins the cluster, remote containers will be replaced with a new local one. This optimises the utilisation of resources in the Clouds while timely handling flash crowds associated with web applications to constantly maintain performance and SLOs with minimum operating costs.

The key contributions of the paper are as follows:

- we present a solution that enhances state-of-the-art auto-elasticity in handling sudden overload situations for multi-cluster container-based web applications in
distributed Clouds;

• we present an architectural framework that extends the capabilities of Kubernetes clusters to support inter-cluster resource management;

• we show how queuing-based performance models can be used for estimating container capacity to process requests without violating SLOs;

• we present two-level cluster elasticity techniques that can horizontally autoscale containers and VMs. This in turn relies on SLO-based container capacity and high-level metrics;

• we show how dynamic container scheduling policies that consider inter-data center latency can be deployed to fit web application needs, and

• we perform extensive experiments on the NeCTAR Research Cloud [46] and the Amazon Cloud across Australia and Europe using both Docker and Kubernetes.

The rest of the paper is organised as follows. In Section 5.2, we discuss the assumptions and challenges associated with multi-cluster container-based web application deployment models. In Section 5.3 we present the proposed approach in detail. We evaluate the approach in Section 5.4. In Section 5.5 we compare the approach with related work. Finally in Section 5.6 we draw conclusions and identify potential future research directions.

5.2 Multi-Cluster Container Deployment Model: Assumptions and Challenges

In this section, we first discuss the assumptions of container deployment in distributed Cloud environments based on a multi-cluster model. Based on these assumptions, we then discuss challenges that need to be tackled to help manage application QoS and support SLOs that minimise cost for Cloud-based web applications through containerisation.
5.2 Assumptions and Challenges

Assumptions

In this paper we assume there exists a containerised web application and data that can potentially be fully replicated across multiple geographically distributed Cloud data centers. We assume that inter-data center data consistency challenges can be relaxed, i.e. there is no shared state. At each data center, the web application (micro)services are deployed across a cluster of VMs. The size of cluster can be adapted by adding/removing VMs through existing Cloud APIs. Additionally, we assume that Kubernetes is used as the container orchestration and cluster management platform. Each VM is also assumed to use Docker [14] as the container run time. Moreover, we assume Prometheus [217] is in place and used as a monitoring system for containers and for the underlying cluster resources. With regards to application workloads, user traffic is routed to the closest cluster using geo-location based DNS services, e.g., using a geo-location routing policy such as Amazon Route 53 [185]. This helps to reduce the impact of network latency on application response time to user requests. Finally we assume that each cluster has its own workload pattern and that each cluster is able to serve different (nearby) users at different times.

Challenges

As discussed, we wish to provide a cost-efficient approach that efficiently utilises multiple data centers to dynamically scale resources in a timely manner in response to bursty traffic increases. This gives rise to a number of related challenges.

Firstly, cluster scaling techniques need to efficiently adapt the number of containers and potentially the number of VMs to maintain application performance whilst aiming to minimise costs. As each cluster runs in a multi-tenant, shared Cloud data center and serves different users, such adaptations need to consider the current status of the shared execution environment and localised workloads. Due to the highly potential for application overloads and performance interference of VMs sharing the same physical server in the Cloud [8, 45], the processing capabilities of containers can vary over
time. This requires performance models that consider such variations to accurately estimate container capacity. The accuracy of this estimation as well as application-related metrics, e.g., user request arrival rates, can be used to infer the required number of containers and VMs needed to run these containers. Adapting the number of VMs can impact on running containers. The challenge is to optimise performance based on distributed usage whilst aiming to minimise the use of the Cloud resources and hence to minimise the cost incurred.

Inter-cluster resource management should handle cluster overload situations. This requires addressing the following issues: how to coordinate clusters so that they can offer/request resources from each other; how to choose a Cloud location that can have existing but idle resources that may be used for overload situations while factoring in inter-data center latencies and their impact on application performance; how to dynamically and selectively schedule container deployments across multiple data centres whilst considering inter-data center network latency and web application needs.

5.3 The proposed approach

In the section, we propose an approach that aims to address the above-mentioned challenges.

5.3.1 System Architecture

Figure 5.2 depicts the architecture of the system designed to meet the aforementioned challenges. The proposed components in the system are primarily divided into two modules: a multi-cluster manager and a local cluster manager. The multi-cluster manager, which runs as a single instance on a separate VM, is responsible for deploying clusters in the Cloud as well as coordinating between clusters to enable inter-cluster resource utilisation. The local cluster manager also runs as a distinct instance at each cluster. This component horizontally autoscales containers and, if required, VM clusters based on local dynamic workloads. The components of the local cluster manager
5.3 The proposed approach

Proposed Components

System Container
Application Container

Existing Components

Proposed Components

Resource Management
Web Scheduler
VM Provisioner
Agent
Web Scheduler
VM Provisioner
Agent

Cluster 1 (AU-Melbourne)
Master Node
Worker Nodes
IaaS Cloud Infrastructure
Cluster 2 (UK-London)
Master Node
Worker Nodes
IaaS Cloud Infrastructure
Cluster 3 (US-East)

Resource Offer/Request

Figure 5.2: Proposed Architecture of the Prototype System with 3 Kubernetes Clusters Deployed at Geographically Distributed Cloud Data Centers

run in multiple containers on the master node of the cluster and extend the capabilities of Kubernetes.

Multi-Cluster Manager

The multi-cluster manager has two components: a cluster deployment component and an inter-cluster resource management component. The cluster deployment component automates the process of getting clusters up and running across data centers, e.g. VM provisioning and container-related software installation once an administrator specifies the locations for the clusters. In our previous work [218] we propose an
approach that intelligently and continuously adapts the number of clusters whilst optimising their locations based on spatial workload variations. Furthermore, the inter-cluster resource management component coordinates between clusters to allow inter-cluster resource utilisation, e.g., when one cluster becomes suddenly overloaded and requires more resources as quickly as possible. Handling such overload situations is discussed in Section 5.3.5. Moreover, this component determines which data centers should have extra capacity to reduce the total number of idle resources needed for the overall deployment and hence reduce the operating costs. This component also considers the inter-data center network latency when selecting a data center that offers idle resources so that the overall network latencies between the selected data center and other data centers running different clusters are minimised to avoid application performance degradation.

**Local Cluster Manager**

A local cluster manager is used at each cluster. The local cluster manager consists of two sub-modules: an *elasticity controller* module and a *resource management* module. The elasticity controller continuously controls cluster scaling horizontally at two levels: the number of containers and the number of VMs upon which they run, as will be discussed in Section 5.3.4. Additionally, the resource management module is responsible for performing scaling actions based on decisions made by the elasticity controller. It has three components: an *agent*, a *web scheduler* and a *VM provisioner*. The agent manages local Cloud resources for a cluster and communicates with the inter-cluster resource management to offer/request resources. It runs as a control loop and waits for scaling requests from the elasticity controller to take appropriate actions. The web scheduler is a custom scheduler realising the scheduling policies of web applications. It works alongside the Kubernetes default scheduler. To use the web scheduler for scheduling containers, services need to specify the web scheduler in their deployment configurations e.g., `schedulerName:web-scheduler`. Finally, the VM provisioner is responsible for provisioning/releasing VMs using Cloud APIs as well as installing any required software, e.g., Docker and Kubernetes.
5.3 The proposed approach

5.3.2 Overall Workflow

At each cluster, the elasticity controller as discussed in Section 5.3.4, periodically collects required metrics from Prometheus to make elastic decisions as/when needed. At each cycle, it assesses requirements for container elasticity for (micro-)services and consistently adapts the number of resources required to run all containers at that given moment. A cluster is considered overloaded when it requires more resources than it actually has at the moment, e.g. there are pending (unschedulable) containers required to scale out some services. In this case before making a VM-level scale-out decision to provision new VMs locally, the elasticity controller first notifies the agent to handle the overload situation through inter-cluster resource utilisation mechanisms. The agent triggers an overload handling algorithm, discussed in Section 5.3.5, to send a resource request to the inter-cluster resource management to rapidly and temporarily utilise idle VMs from local or other Cloud clusters to be able to launch containers immediately. These VMs are added to the cluster as remote nodes. Contemporaneously the web scheduler selects an appropriate scheduling policy that dynamically and selectively places containers for overloaded services locally and/or remotely, as discussed in Section 5.3.6.

After a predefined overload interval, the elasticity controller checks whether the additional resources used to handle the overload are still needed or not. If they are still needed, it determines how many new VMs are required locally at that moment and then it sends a scale-out request to the agent which in turns informs the VM provisioner to provision the required number of VMs locally and to wait until they are ready. Otherwise, the cluster no longer needs the resources and local nodes are sufficient. In both cases before removing the remote nodes from the cluster and releasing them to the inter-cluster resource management so they can be used by other clusters as/when needed, containers running on those nodes will be replaced at local cluster nodes to avoid container disruption. Apart from overload situations, the cluster is considered over-provisioned when the elasticity controller at some given time finds one or more VMs idle and hence they are no longer needed. It then tries to scale in the VM cluster one VM at a time and notifies the agent. If the agent receives a scale-in request,
TABLE 5.1: Terms Used in The Paper

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>Time cycle</td>
</tr>
<tr>
<td>$scaleOutSvcs$</td>
<td>List of services to be scaled out</td>
</tr>
<tr>
<td>$reqSvcsCap$</td>
<td>Total required resource requests for service containers at some time point</td>
</tr>
<tr>
<td>$reqClusterCap$</td>
<td>Total allocated capacity for running containers and required capacity for pending containers in a given cluster</td>
</tr>
<tr>
<td>$desClusterCap$</td>
<td>Desired cluster capacity</td>
</tr>
<tr>
<td>$desNumVMs$</td>
<td>Desired number of VMs in a given cluster</td>
</tr>
<tr>
<td>$desClusterCPU$</td>
<td>Desired CPU capacity of a given cluster</td>
</tr>
<tr>
<td>$desClusterMem$</td>
<td>Desired memory capacity of a given cluster</td>
</tr>
<tr>
<td>$vmCpuCap$</td>
<td>Total CPU capacity of a given VM</td>
</tr>
<tr>
<td>$vmMemCap$</td>
<td>Total RAM capacity of a given VM</td>
</tr>
<tr>
<td>$actNumVMs$</td>
<td>Actual number of VMs in a given cluster</td>
</tr>
<tr>
<td>$ovldElapsed$</td>
<td>Total elapsed time since overload started</td>
</tr>
<tr>
<td>$OvldDetectIntvl$</td>
<td>Predefined overload interval required before deciding if a cluster is overloaded</td>
</tr>
<tr>
<td>$VmScaleInIntvl$</td>
<td>Predefined interval before VM scale-in occurs</td>
</tr>
<tr>
<td>$OvldIntvl$</td>
<td>Predefined overload interval to decide whether to scale-out VMs</td>
</tr>
<tr>
<td>$nLocalVMs$</td>
<td>Number of local nodes in the cluster</td>
</tr>
<tr>
<td>$nRemoteVMs$</td>
<td>Number of remote nodes in the cluster</td>
</tr>
<tr>
<td>$SvcScaleOutIntvl$</td>
<td>Predefined interval before service scale-out occurs</td>
</tr>
<tr>
<td>$SvcScaleInIntvl$</td>
<td>Predefined interval before service scale-in occurs</td>
</tr>
<tr>
<td>$SvcOvldIntvl$</td>
<td>Predefined interval before deciding if a service is overloaded</td>
</tr>
<tr>
<td>$mc$</td>
<td>A microservice $m$ at cluster $c$</td>
</tr>
<tr>
<td>$r_{mc}$</td>
<td>Mean response time of a container of service $m$ at cluster $c$</td>
</tr>
<tr>
<td>$s_{mc}$</td>
<td>Mean service time of a container of service $m$ at cluster $c$</td>
</tr>
<tr>
<td>$\lambda_{mc}$</td>
<td>Incoming load on a container hosting service $m$ at cluster $c$</td>
</tr>
<tr>
<td>$\lambda_{max}$</td>
<td>Maximum input load of a container hosting service $m$ at cluster $c$ without violating SLO</td>
</tr>
<tr>
<td>$\alpha_{mc}(y)$</td>
<td>The $y$th percentile of response time of service $m$ at cluster $c$</td>
</tr>
<tr>
<td>$\lambda_{total}$</td>
<td>Total incoming load of service $m$ at cluster $c$</td>
</tr>
</tbody>
</table>

it first deletes the excess (idle) nodes from the cluster locally. It also sends a resource offer to the inter-cluster resource management component which determines whether to keep the idle VM up and running as additional capacity to be used by other clusters when needed. If it is not needed, the agent informs the VM provisioner to release the resource, i.e. delete the VM via an associated Cloud API.

5.3.3 Service and Cluster Conditions

Each application service running in a cluster has a particular condition (ServiceCondition) that at any time can be either normal or overloaded. In the former case, the load of the service is normal and meets SLOs while in the latter case the overload can negatively affect performance and violate SLOs and hence require fast container scaling to handle such overloads. Similarly, a cluster also has a condition (ClusterCondition),
5.3 The proposed approach

Algorithm 5.1: Two-level Cluster Elasticity Controller Algorithm

\begin{algorithm}
\begin{verbatim}
1 every T do
2   collect metrics from Prometheus;
3   // Container-level elasticity
4   update reqClusterCap;
5   services ← get all application services in the cluster;
6   reqSvcsCap = 0;
7   foreach m ∈ services do
8       ∆ ← autoscale m using Algorithm 5.3;
9       c ← get container resource request of m;
10      reqSvcsCap += ∆ · c;
11      if ∆ > 0 then // Container scale-out
12          let Kubernetes launch ∆ new containers of m;
13      else if ∆ < 0 then // Container Scale-in
14          let Kubernetes delete ∆ containers of m;
15   // Compute desired cluster capacity
16   desClusterCap = reqClusterCap + reqSvcsCap;
17   desNumVMs = max(\lfloor \frac{desClusterCPU}{vmCPUcap} \rfloor, \lfloor \frac{desClusterMem}{vmMemCap} \rfloor);
18   R = desNumVMs - actNumVMs;
19   // VM-level elasticity
20   if ClusterCondition is normal then
21      if R > 0 for OvldDetectIntvl then // Overload is detected
22          set clusterCondition to overloaded;
23          set clusterPhase to overloadDetected;
24          let the agent handle the overload using Algorithm 5.4 and pass R;
25      else if R < 0 for VmScaleInIntvl then // Over-provisioned situation
26          try to scale-in VM using Algorithm 5.2;
27      else if clusterPhase = overloadHandled and ovldElapsed ≤ OvldIntvl and no VM is currently provisioning then // Determine number of VMs to provision locally
28          n = desNumVMs - nLocalVMs;
29          if n > 0 then
30              let agent provision n VMs locally;
31              let agent continue handling the overload using Algorithm 5.4 and pass n;
\end{verbatim}
\end{algorithm}

which at any point in time can be either be normal or overloaded. Normal clusters can launch all pending containers while overloaded ones cannot. The overloaded condition has two phases: overloadDetected where an overload situation in the cluster is detected but not yet handled, and overloadHandled where the overload is being handled through inter-cluster resource utilisation and use of remote resources.
Algorithm 5.2: VM-level Scale In

```
1 nodes ← getAllSchedulableNodes();
2 sort nodes descending based on total free, allocatable CPU, RAM resources;
3 foreach node in nodes do
   4   if All containers in node are movable and they can fit in other nodes then
   5       make node unschedulable;
   6       evict containers in node;
   7       let the agent remove node from the cluster;
   8       break;
```

5.3.4 Local Cluster Elasticity Management at Container and VM Levels

This section presents algorithms and performance models that are used to control elasticity for a cluster at both container and VM levels based on high-level metrics and awareness of variations in container processing capacity over time. This includes a two-level cluster elasticity controller algorithm; a container elasticity technique; and queuing-based performance models to estimate the container capacity based on given SLOs.

Two-level Cluster Elasticity Controller Algorithm

Algorithm 5.1 shows how elasticity for a cluster is consistently controlled at the container and VM level. At each time cycle, it collects low-level and high-level application-related metrics, e.g., request arrival and service times, using Prometheus. Depending on this, it updates the total amount of already allocated capacity for running containers and the required capacity for pending containers in the cluster to get the current state of the required cluster capacity before auto-scaling of containers occurs. Following this it autoscales containers for each service in the cluster using a technique presented in the following section and simultaneously sums the total required resource requests for the auto-scaled services.

The desired cluster capacity to run all containers is then computed. This includes the total amount of both CPU and RAM resources. Based on the CPU and RAM capacities per VM, the desired number of VMs for the cluster is calculated. This is the maximum
number of VMs that are required to provide the cluster with sufficient CPU and RAM resources respectively. The difference between the desired and actual number of VMs \( R \) as well as the cluster condition and phase at that time is used to determine whether an elastic decision at the VM level is needed or not.

When the cluster condition is normal there are two possible cases. The first case happens only when more resources are needed (i.e., \( R > 0 \)) and the overload since the start time elapses beyond a predefined overload detection interval. In this case an overload situation is detected and the agent is required to handle the situation via utilising \( R \) idle resources from other clusters (see Algorithm 5.4). The second case occurs only when one or more resources is not needed (i.e., \( R < 0 \)), i.e., an over-provisioned situation has arisen. In this case the VM cluster may be scaled down as shown in Algorithm 5.2. This algorithm makes a scale down decision and notifies the agent about the decision if the containers running on potentially non-necessary nodes can be replaced and run at other nodes. We refer to containers that can be replaced at other nodes as movable containers. These are stateless web server containers, i.e., we do not tackle state-based issues. Note that this approach favours a VM that has the most free resources to reduce the number of containers that are disrupted.

When an overload situation is handled, the overload time elapses beyond the overload interval and no further VMs are currently provisioned locally, it is necessary to decide whether there is a need to provision VMs locally and if so, how many. The number of new VMs to provision locally \( n \) is calculated based on the difference between the desired number of resources at that moment and the current number of local resources. If this is greater than 0, the decision is that one or more VMs need to be provisioned locally and the agent will be informed accordingly. Otherwise, local nodes are sufficient. Note that if the difference \( n \) is greater than 0 but less than the number of remote nodes, the number of new VMs to provision is less than the one when the overload was detected. This implies that the load has decreased but it is still more than the one when the cluster was last normal.
Container Elasticity Techniques

Container elasticity techniques aim to adapt the number of containers and detect an overload condition of a service \( m \) at a cluster \( c \) based on the current total workload of the service \( \lambda_{\text{total}}^{mc} \) and the estimated container processing capacity \( \lambda_{\text{max}}^{mc} \), which will be discussed in the next section. To adapt the number of containers we first calculate the desired number of containers \( \text{cont}_{mc}^{\text{des}} \) to handle the current total request rate of a given service \( \lambda_{mc}^{\text{total}} \) using Eq 5.1.

\[
\text{cont}_{mc}^{\text{des}} = \left\lceil \frac{\lambda_{mc}^{\text{total}}}{\lambda_{mc}^{\text{max}}} \right\rceil
\]  

Algorithm 5.3 shows how an elastic decision is made if there is a difference between the desired and actual (current) number of containers of the service. If the difference \( \Delta \) is positive for a predefined period, then this returns a scale-out decision with \( \Delta \) as a number of new additional containers. Also, using an approach presented in [88], we consider a service as overloaded when the incoming workload \( \lambda_{mc}^{\text{total}} \) is greater than \( \text{cont}_{mc}^{\text{act}} \cdot \lambda_{mc}^{\text{max}} + \sqrt{\text{cont}_{mc}^{\text{act}} \cdot \lambda_{mc}^{\text{max}}} \) for a pre-configured interval. The reason behind adding \( \sqrt{\text{cont}_{mc}^{\text{act}} \cdot \lambda_{mc}^{\text{max}}} \) is to lower the amount of false positives that may arise with highly oscillating workloads and overload situations caused by far higher loads [88]. On the other hand, if the difference \( \Delta \) is negative for some pre-defined time, then this returns a scale-in decision with \( \Delta \) as the number of containers to be deleted. Otherwise, the service remains as is.

SLO-based Queuing Models for Estimating Container Capacity

To estimate the container processing capacity of a service \( m \) at a given cluster \( c \) based on pre-agreed SLOs, it is necessary to determine the maximum input load \( \lambda_{mc}^{\text{max}} \) on an individual container of the service \( m \) at cluster \( c \) that can be processed per second without violating SLOs. Hence, we model a container based on an M/M/1 queuing model, assuming the arrival rate is Poisson and the service time is exponentially distributed. Table 5.1 is used to define the relevant terms. Using Little’s Law and since the mean
5.3 The proposed approach

**Algorithm 5.3: Container Elasticity Algorithm**

**Input:** Service \( m \)

1. \( \text{cont}^{\text{act}}_m \leftarrow \text{getCurrentNumConts}(m); \)
2. \( \text{cont}^{\text{des}}_m \leftarrow \text{getDesiredNumConts}(m) \) using Eq 5.1;
3. \( \Delta = \text{cont}^{\text{des}}_m - \text{cont}^{\text{act}}_m \);
4. if \( \Delta > 0 \) for \( \text{SvcScaleOutIntvl} \) then \(/ / \text{Scale out} \)
   5. \hspace{1em} if \( \lambda^{\text{total}}_m > (\text{cont}^{\text{act}}_m \cdot \lambda^{\text{max}}_m + \sqrt{\text{cont}^{\text{act}}_m \cdot \lambda^{\text{max}}_m}) \) for \( \text{SvcOvldIntvl} \) then
   6. \hspace{2em} mark \( m \) as overloaded;
5. \hspace{1em} return \( \Delta \);
6. else if \( \Delta < 0 \) for \( \text{SvcScaleInIntvl} \) then \(/ / \text{Scale in} \)
   7. \hspace{1em} mark \( m \) as normal;
   8. \hspace{1em} return \( \Delta \);
7. /* No change, otherwise */
8. return 0;

Service time \( \bar{s} \) is modelled as \( \bar{s} = \frac{1}{\mu} \) where \( \mu \) is the mean service rate (i.e. requests per second), the mean response time of a container of service \( m \) at cluster \( c \), \( r_{mc} \), can be represented as

\[
r_{mc} = \frac{\bar{s}_{mc}}{1 - \lambda_{mc} \cdot \bar{s}_{mc}} \quad (5.2)
\]

If SLO\(^{v}_{mc}\) is defined as the \( y \)th percentile of response times of service \( m \) at cluster \( c \), \( \alpha_{mc}(y) \), then it has to satisfy \( \alpha_{mc}(y) \leq v \). For instance, if \( v = 2 \) seconds and \( y = 95 \), then the 95th percentile of response time has to satisfy \( \alpha_{mc}(95) \leq 2 \). Based on Eq 5.2 and the formula \( \alpha_{mc}(y) = r_{mc} \cdot \ln \left( \frac{100}{100-y} \right) \) [219], the upper bound on the maximum input load \( \lambda^{\text{max}}_{mc} \) can be represented as:

\[
\lambda^{\text{max}}_{mc} \leq \left[ \frac{1}{\bar{s}_{mc}} - \frac{\ln \left( \frac{100}{100-y} \right)}{v} \right] \quad (5.3)
\]

If SLO\(^{v}_{mc}\) is defined as the response time \( r_{mc} \) and this has to be no more than \( v \) (i.e. \( r_{mc} \leq v \)), then by using Eq 5.2 we obtain the upper bound on the maximum input load \( \lambda^{\text{max}}_{mc} \) which is given as:

\[
\lambda^{\text{max}}_{mc} \leq \left[ \frac{1}{\bar{s}_{mc}} - \frac{1}{v} \right] \quad (5.4)
\]
Algorithm 5.4: Cluster Overload Handling Algorithm via Inter-Cluster Resource Utilisation

**Input**: \( N \) (Number of remote nodes to request or number of VMs are provisioning locally)

1. if \( \text{clusterPhase} = \text{overloadDetected} \) then
   1.1. \( \text{remoteNodes} \leftarrow \) send a resource request (\( N \) VMs) to inter-cluster resource management;
   1.2. join \( \text{remoteNodes} \) to the cluster;
   1.3. /* Make web scheduler aware of remote nodes */
   1.4. set \( \text{clusterPhase} \) to \( \text{overloadedHandled} \);
2. else if \( \text{clusterPhase} = \text{overloadHandled} \) then
   2.1. if \( N > 0 \) then
   2.2. | wait until new local VMs are provisioned and joined the cluster;
   2.3. make \( \text{remoteNodes} \) unschedulable;
   2.4. evict containers in \( \text{remoteNodes} \);
   2.5. delete \( \text{remoteNodes} \) from the cluster;
   2.6. set \( \text{ClusterCondition} \) to normal;
   2.7. hand in \( \text{remoteNodes} \) to inter-cluster resource management;

5.3.5 Cluster Overload Handling Algorithm via Inter-Cluster Resource Utilisation

As stated, the cluster overload handling algorithm shown in Algorithm 5.4, is triggered by the agent in two cases. In the first case an overload is detected in the cluster. The input \( N \) is the number of remote nodes to request from the inter-cluster resource manager. The algorithm joins them to the cluster and finally sets the cluster status to \( \text{overloadedHandled} \). In the second case the overload situation is already handled. The decision whether to provision VMs locally or not is made and it is necessary to finish the process of handling the overload. In this case the input \( N \) refers to the decision. If it is greater than 0 then new VMs are to be provisioned and the algorithm waits until new VMs are provisioned and join the cluster as new local nodes. Before deleting remote nodes from the cluster, they are marked as unschedulable and containers running on them are evicted and replaced at the new local nodes. At this stage the cluster is back to normal. Finally the remote nodes become idle and are returned back to the inter-cluster resource management system.
5.3 The proposed approach

Algorithm 5.5: Container Scheduling Policy Selector

Input: Pending Container \( cont \)

1. if \( ClusterCondition \) is normal then
   1.1 schedule \( cont \) using policy in Algorithm 5.6;
2. else // The cluster is overloaded otherwise
   2.1 schedule \( cont \) using policy in Algorithm 5.7;

5.3.6 Container Scheduling Policies

Algorithm 5.6: Container Scheduling Policy (Cluster condition is normal)

Input: Pending Container \( cont \)

1. \( svcCond \leftarrow \text{getServiceCondition}(cont) \);
2. if \( svcCond \) is overloaded then
   2.1 schedule(\( cont \));
3. else if all pending containers belong to normal services then // Prioritise overloaded services
   3.1 schedule(\( cont \));
4. Function schedule(\( Container cont \)): // Best-fit bin packing
   4.1 \( nodes \leftarrow \text{getSchedulableNodes}() \);
   4.2 /* Prefer nodes with most requested resources */
   4.3 sort \( nodes \) descending based on total requested resources;
   4.4 foreach node in \( nodes \) do
   4.5   if node has enough CPU and RAM to run \( cont \) then
   4.6     bind(node, \( cont \));
   4.7     break;

This section presents two container scheduling policies that aim to maintain web application performance during both normal and overloaded cluster conditions whilst mitigating the impact of container disruptions. Depending on the cluster condition at the time of scheduling, a given policy is selected by the web scheduler to place a pending container in one of the cluster nodes as described in Algorithm 5.5.

The policy in Algorithm 5.6 is selected when the cluster condition is normal. This policy aims to prioritise launching pending containers for overloaded services as they can negatively affect web application performance. This considers the best-fit bin packing heuristic for the scheduling problem. This heuristic places pending containers (i.e.
Algorithm 5.7: Selective Container Scheduling Policy (Cluster condition is overloaded)

Input: Pending Container \( cont \)

\[
\begin{align*}
\text{svcCond} & \leftarrow \text{getServiceCondition}(cont); \\
\text{if} \ \text{svcCond is overloaded} \text{ then} \\
& \quad \text{nodes} \leftarrow \text{getSchedulableLocalNodes}(); \\
& \quad \text{sort} \ \text{nodes ascending based on total requested resources}; \\
& \quad \text{scheduled} = \text{false}; \\
& \quad \text{foreach node in nodes do} \\
& \quad \quad \text{freeCPU, RAM} \leftarrow \text{getFreeNodeRes}(node); \\
& \quad \quad \text{if free CPU, RAM are enough to run cont then} \\
& \quad \quad \quad \text{bind}(\text{node, cont}); \\
& \quad \quad \quad \text{scheduled} = \text{true}; \\
& \quad \quad \quad \text{break}; \\
& \quad \quad \text{if node has movable containers then} \\
& \quad \quad \quad \text{movConts} \leftarrow \text{movNormalSvcConts}(\text{node}); \\
& \quad \quad \quad \text{sort movConts ascending based on load/resource utilisation}; \\
& \quad \quad \quad \text{foreach cont in movConts do} \\
& \quad \quad \quad \quad \text{add cont to evictConts list}; \\
& \quad \quad \quad \quad \text{update free CPU, RAM}; \\
& \quad \quad \quad \quad \text{if free CPU, RAM are enough for cont then} \\
& \quad \quad \quad \quad \quad \text{evict containers in evictConts}; \\
& \quad \quad \quad \quad \quad \text{bind}(\text{node, cont}); \\
& \quad \quad \quad \quad \quad \text{scheduled} = \text{true}; \\
& \quad \quad \quad \quad \quad \text{break}; \\
& \quad \quad \quad \text{if scheduled then} \\
& \quad \quad \quad \quad \text{break}; \\
& \quad \quad \text{if not scheduled then} \\
& \quad \quad \quad \text{schedule_remotely}(cont); \\
& \quad \text{else if clusterPhase is overloadHandled then} // Service is normal \\
& \quad \quad \quad \text{schedule_remotely}(cont); \\
& \text{Function schedule_remotely (Container cont):} \\
& \quad \text{rNodes} \leftarrow \text{getRemoteNodes}(); \\
& \quad \text{foreach node in rNodes do} \\
& \quad \quad \text{if node has enough free cpu, mem for cont then} \\
& \quad \quad \quad \text{bind}(\text{node, cont}); \\
& \quad \quad \quad \text{break}; 
\end{align*}
\]

items) inside the VM (i.e. bin) that has most requested resources to optimise the resource utilisation. This will result in minimising the number of VMs required by the cluster to run application containers, thereby reducing the operating cost.
When the cluster is overloaded, the policy in Algorithm 5.7 is chosen. This aims to selectively place pending containers for overloaded services at local cluster nodes while placing the ones for normal services at remote, temporary nodes once they join the cluster (i.e. this is the overloadHandled phase). If a pending container is for an overloaded service and there are not enough resources (CPU, RAM), a number of running containers with the least load/resource utilisation for normal services at local nodes will be evicted to provide enough resources to accommodate the pending container. To minimise the number of evicted containers, the policy favours nodes with the least requested resources. Evicted containers will subsequently be placed at remote nodes. If the policy fails to schedule the overloaded pending container locally, it will schedule the container remotely. If the pending container is for a normal service, it will be scheduled at a remote node. This policy minimises the impact of inter-data center latency on response times of overloaded services to optimise the overall application performance and minimise SLO violations.

5.4 Performance Evaluation

We have conducted an experimental evaluation of the proposed elasticity approach on the Australian National eResearch Collaboration Tools and Resources Research Cloud (NeCTAR) [46] and the Amazon Cloud. The evaluation focuses on (1) the effectiveness of inter-cluster resource utilisation, (2) the benefits of two-level cluster elasticity, (3) the effectiveness of inter-data center latency aware scheduling, and (4) the effectiveness of different queuing-based container capacity models. In the next section we describe the experimental set-up that is shared between experiments while individual settings are explained within each experiment section below.

5.4.1 Experimental Set-up

Cloud Data Centers and Compute Resources. Our experiments use resources from the Melbourne Data Center availability zone of the NeCTAR Cloud as well as resources
from the London data center availability zone of the Amazon Cloud. A network latency between these locations is on average between 280-320ms. We utilise two types of VM instances in the experiments: medium and large. A medium instance consists of 2 virtual CPUs with 8GB of RAM. A large one has 4 virtual CPUs with 16GB of RAM. All instances run Ubuntu 18.04 LTS (Bionic) as the operating system. For medium instances, we use \texttt{m1.medium} and \texttt{t3.large} types in the NeCTAR and Amazon Clouds respectively. For large instances, we use \texttt{m1.large} type in the NeCTAR Cloud. Also, VM provisioning times, including times for container-related software installation and for joining a cluster, for an autoscaled node once a VM scale-out decision is made are on average 300s.

\textbf{Multi-Cluster Deployment.} For the purpose of experimental evaluation we use one running cluster in a data center to be targeted as the overloaded cluster with idle resources in another data center that can be used by the multi-cluster manager. Hence, we run the cluster in the Melbourne data center (i.e. Melbourne cluster) while the idle, running resources are running in the London data center. Moreover, as we consider multi-cluster deployments, we assume that another cluster is running in the London data center that is used for evaluating the overall cost of multi-cluster deployment. At each cluster node, we use Docker (v18.06.2) and Kubernetes (v1.16.3). Master nodes run on large instances while worker nodes run on medium instances.

\textbf{Cluster and Application Monitoring.} At the Melbourne cluster, we have two Prometheus instances (v2.17). One instance is used to monitor and collect cluster-related metrics, e.g. resource utilisation, while the other is used to collect high-level application-related metrics, e.g. arrival request rates and service times.

\textbf{The Proposed Elasticity Controller Setting.} Using terms in Table 5.1, we set the following parameters $T$, $OvldDetectIntvl$, $VmScaleInIntvl$, $OvldIntvl$, $SvcScaleOutIntvl$, $SvcScaleInIntvl$ and $SvcOvldIntvl$ to 3s, 20s, 30s, 80s, 10s, 15s and 6s. It is noted that the settings of these parameters are application-specific and assumed to be tuned by the application administrators. In our experiments, we ran a range of experiments with different values to obtain the optimal parameter settings to suit the application benchmarking.
5.4 Performance Evaluation

5.4.2 Application Benchmarking

We use a Sock Shop microservice-based web application [48]. This consists of 13 microservices: front-end, order, payment, user, catalogue, cart, shipping, queue master, rebbit-mq, catalogue-db, order-db, user-db, cart-db. We use a “one-container-per-Pod” model, which is the most common Kubernetes use case. Thus, container and pod terms can be used interchangeably. We mark pods that are not related to database services as movable. Also, we specify resource requests and limits of CPU and memory resources of all pods to 100m and 300MB respectively\(^1\).

5.4.3 Workload

To validate the applicability of our approach against real-world situations, we use a real-world FIFA World Cup 1998 dataset [220]. For the workload pattern in our experiments, we used a 18-hour trace starting from 6am on the 7 July to 12am on the 8 July, 1998. As shown in Figure 5.3, this pattern fits our needs as it represents flash crowds.

\(^1\)One CPU/Core is equivalent to 1000m (millicpu).
This requires both scale-out and scale-in decisions for elasticity approaches. As the request arrival rate is collected per minute in the dataset, we use it in our experiment as the request rate. Hence, the length of each run in our experiments is 18 min (1080s). We scale the number of requests to the maximum that fits the number of Cloud resources available in our experiments.

To generate user requests we use Locust, an open-source modern, distributed load testing framework [221]. We run Locust load tests distributed across 6 large VM instances to avoid any potential performance issues on the client side when generating a substantial number of concurrent requests per second. We use one Locust master in a separate instance and 20 Locust workers in the other 5 instances (i.e. one worker per CPU).

5.4.4 Benchmarks

To evaluate our elasticity approach, we compare it with the following benchmarks:

- **Kubernetes Autoscaling (K8s-Autoscaling)**: This benchmark has been widely used for autoscaling Kubernetes clusters in many Cloud providers, e.g. Amazon, Google, Azure, etc. At the container level, Horizontal Pod Autoscaler (HPA) [154] is used. It periodically adapts the number of pod replicas of a service to match the observed average CPU utilisation of all containers to a pre-defined target. At the VM level, Cluster Autoscaler (CA) [148] is used. Since CA is Cloud-provider dependent, we implemented the CA algorithm to be used in our self-managed clusters in the NeCTAR Cloud. CA scales out the cluster if any newly created container remains pending for more than a predefined launch latency time (default is 30s). It also scales in the cluster when some nodes are idle for some time (default is 5 min) and running pods in these excess nodes can be replaced elsewhere. For HPA we run experiments with different parameter settings to make it more responsive to bursty workloads. Therefore, we set the target cpu percentage use, the sync period, the cpu initialization period, the initial readiness delay, the downscale stabilization for HPA to 65%, 5s, 10s, 10s and 15s respectively. For CA we only configure scale-down intervals
5.4 Performance Evaluation

before removing excess VMs to 30s to suit the length of our experiments while scanning interval and latency periods from the time a pod is tagged as unschedulable to the time a scale-out request is issued. These are set to 10s and 30s respectively. Note that the two autoscaling techniques run separately and are not aware of each other.

- **Over-provisioning**: The second benchmark periodically checks that each cluster has extra capacity, e.g. one or more idle VMs are running, to be able to handle sudden workload bursts. It periodically autoscales the number of VMs to ensure that the number of provisioned resources is always greater than that of the current resource demand by a predefined number (or percentage). In our experiments we set this number to 1. Also, we set $T$ (time cycle), VM scale-in interval and VM scale-out interval to 10s, 30s and 30s respectively. For autoscaling at the container level, we use the proposed container elasticity technique. However, as with the previous benchmark and real-world Cloud use case scenarios, elasticity techniques for containers and VMs are unaware of each other.

To evaluate our dynamic scheduling policies of our web scheduler, we compare it with the following scheduler:

- **Kubernetes Default Scheduler (K8s-default-scheduler)**: This is the default scheduler of Kubernetes, i.e., kube-scheduler. It is used in the experiments with the default configuration.

5.4.5 Evaluation of the Proposed Elasticity Approach

In the first set of experiments, we tested our elasticity approach and the other benchmarks under flash crowds using the workload depicted in Figure 5.3. There are 3 experiments, i.e. one per approach. We run each experiment 5 times and the length of each run is 18min. At each run, we have two clusters: Melbourne and London clusters. The initial deployment size of each cluster for our approach and Kubernetes Autoscaling approach is 3 VMs while for over-provisioning approach it is 4 VMs. Additionally, in our approach we have 1 idle, running VM in London data center available to handle
Cost-oriented Container-based Scaling in Distributed Clouds

Therefore, the total number of VMs for proposed, Kubernetes and over-provisioning offered an idle VM resource to the multi-cluster manager, as discussed in Section 5.3.2. Therefore, the total number of VMs for proposed, Kubernetes and over-provisioning approaches are 7, 6 and 8 respectively. Moreover, the SLO is defined as the 95th percentile of response times is less than 2s ($\alpha_{mc}(95) \leq 2s$). Also, the scrape interval for evaluation metrics is set to 5s.

At each run, we first deploy application microservices. There are 4 pod replicas for the front-end service and 1 pod replica for the remaining services. Then we warm up the application with 800 requests per second for 2min. Next, the workload generator starts

---

**Figure 5.4:** Performance comparisons of different elasticity solutions during overload situations for multi-cluster container-based web applications.
5.4 Performance Evaluation

![Graphs](image)

(A) The 95th percentile of response times. (The SLO here is the 95th percentile of response times is less than 2 seconds.)

(B) Number of containers. (Solid lines: running containers. Dotted lines: both running and pending ones.)

(C) Average response times.

(D) Number of VMs.

**Figure 5.5:** The performance of the Melbourne cluster with different elasticity approaches in response to dynamic workloads.

generating user requests using Locust and continuously modify request rates based on the workload pattern.

**The Effectiveness of Inter-cluster Resource Utilisation**

Figure 5.4 presents the performance comparisons between elasticity approaches. As Figure 5.4a shows, our approach, when compared to Kubernetes autoscaling approach, was able to significantly optimise the launch latency for containers during overload situations by 95%. The 95% confidence interval of the average launch latency for our
approach and the Kubernetes autoscaling approach are (8.74s, 12.56s) and (227.05s, 267.59s) respectively. The Kubernetes approach introduces a substantial increase in launch latency due to the inevitable VM provisioning delays required before pending containers can be launched. Our approach timely manages the launch latency using inter-cluster resource utilisation.

On the other side, the over-provisioning approach (unsurprisingly) introduces zero latency as it constantly promises to have an extra reserved capacity to eliminate any potential increase in the latency. However this comes at a high cost. As shown in Figure 5.4b, compared to the Kubernetes approach, the over-provisioning approach increases the operating cost by 20% while the cost in our approach increased by only 14%. In other words our approach reduces the cost by 6% comparing the over-provisioning approach because our approach, as shown in Figure 5.4c, reduces the number of VMs for the overall deployment that are used as extra capacity. For a larger size of multi-cluster deployments this cost reduction can be significantly high. For instance if we have 8 clusters and 1 VM as extra capacity for every 4 clusters. Our approach would reduce the cost by 75% compared the over-provisioning approach. This is because the over-provisioning approach requires 8 idle VMs (one per cluster) while 2 VMs are required in our approach.

Additionally, as depicted in Figure 5.4d both our approach and the over-provisioning approach are able to improve performance and avoid the pre-agreed SLO violation as they maintain the 95\textsuperscript{th} percentile of response times to be below 2 seconds. As a consequence, it is evident that our approach, by using inter-cluster resources utilisation, is able to optimise the overall operating cost while maintaining performance to avoid/minimise violations of SLOs.

**Benefits of Two-level Cluster Elasticity**

Results presented in Figure 5.5 show the performance of the Melbourne cluster using different elasticity solutions. Figure 5.5a shows the ability of the proposed and the over-provisioning approaches to maintain the performance and support SLOs while the Kubernetes autoscaling approach fails to do that for almost 70% of the time. Also,
the two approaches, as shown in Figure 5.5c, illustrate their ability to quickly react to sudden increases in workloads and thus dramatically drop the response times. This fast reaction is obvious in Figure 5.5b as the number of containers increases drastically around the 50th second of the run time while the Kubernetes autoscaling approach reacts later (at the 120th second) but also far more gradually. This is because the Kubernetes approach considers only resource-level metric, CPU utilisation, which is not enough for latency-sensitive bursty applications, while the other approaches rely on high-level, application-related metrics.

Moreover, the Kubernetes autoscaling fails to accurately adapt not only the number of containers but also the number of VMs. When the workload starts to decrease after the middle of the run time (at the 500th second), the Kubernetes approach continues to increase both the number of containers and VMs as shown in Figure 5.5b and Figure 5.5d. However, the number of containers for both the proposed and over-provisioning approaches are able to follow the workload demand. Similar to the Kubernetes approach, the over-provisioning approach fails to match the demand at the VM level. While the workload decreases starting from the middle of the run time, the over-provisioning approach increases the number of VMs to 5.

In comparison, our approach is able to follow the demand at both container and VM levels, as it consistently adapts both the number of containers and VMs based on the increase or decrease in the workload demand. This shows its ability to optimise the cluster resources while providing enough resources to run containers to maintain the application performance and support SLOs. This allows to minimise the cost while maintaining application performance.

5.4.6 Web Scheduler Performance

In the second experiment set we performed tests to evaluate the performance of the proposed scheduler. In particular we examine its ability using our proposed scheduling policies to improve/maintain the application performance during cluster overload
At each run we use the Melbourne cluster as the target cluster with a size of 4 VMs and an idle running VM is in the London data center. We set the minimum number of pod replicas of each service to 3 to increase the number of moveable pods for normal services. We start each run by generating loads that cause a cluster to be overloaded and hence the front-end service as experienced by end users becomes overloaded. We start monitoring and recording evaluation metrics with a resolution of 5s once a remote
node joins the Melbourne cluster to handle the overload situation whilst the VM is provisioned locally. We continue recording after a local VM joins the cluster for 200s.

Figure 5.6 and Table 5.2 indicate the impact of the two schedulers on the application performance using the 95\textsuperscript{th} percentile of the response times. The proposed scheduler obviously shows better application performance during overload situations (approximately up to the 300\textsuperscript{th} second). This improvement in performance occurs because the scheduler is aware of the inter-data center network latency and hence is able to dynamically place normally loaded containers at remote nodes to replace overloaded local containers dealing with a substantial number of user requests. This dynamic placement aims to exploit local resources where possible to process more requests to minimise the number of requests that will be processed remotely and hence to decrease the overall network latency that can negatively affect the overall response times. On the other hand, the Kubernetes default scheduler incurs higher response times as it not aware of the inter-data center latency when scheduling containers. This causes a large number of requests to be processed remotely. It is evident that schedulers that dynamically place containers whilst considering inter-data center latency for cross-data center clusters can help optimise the overall application performance.

5.4.7 Container Capacity Model Analysis

This section demonstrates the effectiveness of the proposed queuing-based container processing capacity models as well as the importance of considering high-level application-related metrics, e.g., container capacity when making scaling decisions. To evaluate the models, we use empirical measurements to obtain the actual processing capacity of a single container hosting a microservice over specific hours of the day and then compare the measurements to the model estimations.

We deploy microservices realising a Sock Shop application on the Melbourne cluster. As we target the front-end microservice, we deploy a single front-end container. We also run a workload generator in the same data center. For the first model (Eq. 5.3) we define the SLO as the 95\textsuperscript{th} percentile of response times less than or equal 1 second (i.e.
Figure 5.7: Comparisons of container capacity estimated by the proposed queuing-based capacity models vs the empirically measured container capacity at different times of the day based on SLOs. AEDT: Australian Eastern Daylight Time.

We present two experiments. At each experiment run, we warm up the application for 10 minutes and then run the experiment for 24 hours. At each hour, the Locust workload generator generates the maximum number of requests without violating the SLOs. We collect and record the request rates from Locust with a resolution of 3 seconds (i.e. 1200 times per hour) and then use their average to represent the actual container capacity over that hour. At the same time, we collect and record the application service times from the application-related Prometheus instance with the same level of temporal resolution. We use their average as input to the queuing model to estimate the container capacity at that hour. We repeated the experiment 3 times and get the average of the actual and estimated container capacity for each hour. In the second experiment, we examine the second queuing model (Eq. 5.4). We repeat the same procedure undertaken in the first experiment with the other defined SLO.
TABLE 5.3: The difference in container capacity between the empirical measurements and models’ estimations over times of the day and the accuracy of the models’ estimations.

<table>
<thead>
<tr>
<th>The Model</th>
<th>Differences in Container Capacity (Measured - Estimated) (Requests/Sec)</th>
<th>Accuracy (%)</th>
<th>Range</th>
<th>95% CI of Difference in Mean</th>
<th>Range</th>
<th>95% CI of Accuracy Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Model (Eq. 5.3)</td>
<td>2.65 - 2.95 (2.78, 2.83)</td>
<td>98.36 - 98.6</td>
<td>(98.48, 98.53)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Second Model (Eq. 5.4)</td>
<td>0.67 - 1.09 (0.78, 0.85)</td>
<td>99.38 - 99.64</td>
<td>(99.54, 99.58)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Effectiveness of the Proposed Queuing Models

Figure 5.7 and Table 5.3 show the estimated container processing capacity using the models compared to the observed capacity that is empirically measured over different times of the day. For the first model, as shown in Figure 5.7a, the difference between the observed and estimated container capacity ranges between 2.65-2.95 requests per second. The 95% Confidence Interval (CI) of the mean of that difference over all hours varies between 2.78-2.83 requests per second. This shows that the difference is very small. In terms of accuracy, over all hours the accuracy of the model’s estimation ranges between 98.36% - 98.6% and the 95% CI of the average of the accuracy is between 98.48%-98.53%. Similarly, as shown in Figure 5.7b, the second model exhibits very small differences between the observed and estimated container capacities and highly accurate estimations. The capacity differences fall within a range between 0.67-1.09 requests per second. The 95% Confidence Interval (CI) of the mean of these differences over all hours varies between 0.78- 0.85 requests per second. Moreover, the accuracy of the estimations over all hours ranges from 99.38% - 99.64%. The 95% CI of the average of the accuracy is between 99.54%- 99.58%. From the results, it is evident that both models are able to estimate container capacity effectively.

The Significance of High-level Metrics

Figure 5.7 indicates the variation in the maximum container capacity at different times of day. As shown, the measured container capacity over time varies over times showing up to 7.7% and 6.8% difference in the capacity over time in Figure 5.7a and Figure 5.7b respectively. For example, in Figure 5.7a the highest capacity (193.12 requests per second) was at 6am and the lowest capacity found at 4pm. Even though we fix the amount
of compute resources allocated to the container, the container capacity varies over time. This indicates that considering only low-level resource metrics, e.g., CPU utilisation, to make scaling decisions is inadequate in the Cloud because of the existence of other Cloud-related factors that can cause variations, e.g., performance interference between co-located and unrelated VMs on a given physical server [8]. Such factors emphasise the need for performance models that consider high-level application metrics for application containers, particularly for latency-sensitive applications, e.g., web applications. As such, considering container capacity and other high-level metrics can significantly help to improve application performance and support SLOs.

5.5 Related Work

**Overload Management in the Cloud.** Significant effort has been devoted in academia and industry to address the problem of overload management in Cloud using a variety of approaches and targeting different objectives. However, they differ from our work in terms of the approach taken or intended goal.

One common approach is Cloud-bursting through hybrid Clouds. Cloud bursting is the process of expanding compute resources in private Clouds by bursting into public Clouds when private resources are not sufficient to manage overloads, e.g., due to increases in workloads or resource failures [105–108]. This approach is cost-effective as it exploits local resources. However, it still suffers from the problem of VM provisioning delays in the Clouds and hence they cannot timely manage sudden workload bursts as often occurs with web applications. Our approach aims to handle such workload bursts in a timely manner whilst considering costs as well.

Resource borrowing is another overload management approach. It handles overloaded services/components by borrowing idle resources from other servers that are idle or can be temporarily shut down in a data center [45, 116, 118]. In [116], the authors present a solution for managing overload periods, assuming data centers are multi-tiered, by borrowing resources from other underutilised tiers. Similarly, in the container context [45] propose an integrated solution for managing high loads as well
as energy consumption in container-based data centers. Their approach temporarily shuts down non-essential application containers to lend resources to overloaded services to provision more containers while considering energy consumption. Different to them, our approach tackles the problem of managing overloads from an application provider’s view rather than a Cloud infrastructure perspective.

Another approach for overload management based on multi-cluster container-based applications is inter-cluster resource utilisation. Some solutions allow utilising resources from shared, public Cloud clusters, e.g., AWS Fargate [212] and Azure Container Instances [222]. However as stated, they can introduce performance interference issues impacting the application performance and SLOs. Similarly, Netflix Titus [16] provides an open-source container management platform in the Cloud. This was developed by Netflix and is targeted to private clusters however it does support different workload types at these clusters. However, all these solutions do not suit our needs as they assume clusters are running in the same data center. Different to these solutions, our solution assumes multi-cluster container-based web applications deployed across multiple geographically distributed data centers.

Over-provisioning is another widely used overload management approach. This approach requires each cluster to over-provision more resources individually to have extra reserved computing capacity, e.g., using AWS Capacity Reservation [223]. The size of the additional capacity can be static [224], e.g. 2 additional VMs or cores per cluster, or dynamic [225], e.g. 10% of cluster resources. Solutions adopting this approach can maintain application performance even in the presence of flash crowds. However, this can be very costly for multi-cluster deployments. Different to these solutions, our solution aims to provide global, shared capacity for all clusters to maintain application performance while minimising overall resource cost.

**Container-based Application Elasticity in Clouds.** Considerable work has explored controlling container-based application elasticity in the Cloud. However, none of them are table to our needs.
The Kubernetes Horizontal Pod Autoscaler [154] and service autoscaling through the Amazon Elastic Container Service (ECS) [33] are used to autoscale containers for application services based on resource-level metrics, e.g. the average CPU utilisation of containers. Other elasticity solutions like [35, 150] consider high-level metrics and variations in container capacity in the Cloud to accurately make elasticity decisions. However, such solutions only consider elasticity at the container level. On the other hand, other solutions manage elasticity at the VM-level only [148, 160–162]. Different to these solution, our work aims to adapt both the number of containers and VMs to efficiently optimise the Cloud resources needed/used and thereby reduce the overall costs.

A two-level vertical elasticity technique is presented in [158] that is able to resize both Docker containers as well as VMs to cope with dynamic workloads to maintain application performance. Similarly, [156] presents a vertical and horizontal elasticity mechanism for both container and VM levels for multi-tier web applications. However, resizing resources cannot benefit from inter-cluster resource utilisation and will never solve global spatial variations in load. Different to their solutions we present a two-level horizontal elasticity solution where adding/removing resources is applicable to inter-cluster resource management at a global scale.

### 5.6 Conclusions and Future Directions

In this paper, we have presented a cost-aware container-based elasticity approach for bursty multi-cluster containerised web applications. This is deployed in geographically distributed and heterogeneous Cloud computing environments. Instead of having individual extra compute capacity at each cluster to handle sudden workload bursts, our approach intelligently optimises the utilisation of resources between clusters running in different locations to provide a global shared computing capacity that is ready for use to handle overload situations in some location. We have shown that our approach is able to optimise launch latency in cluster overload situations to constantly maintain application performance and SLOs with minimum operating costs.
In the future, we will investigate the development of a model that can dynamically configure the proper size of the global shared capacity based on workloads for large-scale cluster deployments. We also intend to explore optimisation techniques that can dynamically configure the required number of running containers based on current workloads whilst minimising disruption when making Cluster elasticity decisions at the VM level. This would maintain not only application availability but also application performance.

Future work will also consider the varying pricing models in using commercial clouds such as Amazon and the challenges and opportunities that this offers.
Chapter 6
Conclusions and Future Directions

In this chapter, we summarise the research presented in this thesis and highlight the main contributions. We also outline some open research challenges that could form the basis for potential future work and research directions.

6.1 Summary and Conclusions

The Cloud computing paradigm provides computing resources to users with the ability to dynamically and elastically allocate computing and storage capacity to users, and hence their applications in an on-demand, scalable and pay-as-you-use manner. Application providers can benefit from capabilities to cope with dynamic workloads to achieve scalability and performance. With the global Cloud environments now available, where data centers have been increasingly geographically distributed, web application providers can leverage such capabilities by deploying their applications across multiple Cloud locations to gain advantages such as reducing network latency, increasing availability and the potential to access and use cheaper Cloud resources.

Recently, containers, a lightweight virtualisation environment for packaging applications, have become one of the leading approaches for developing, delivering, and subsequently managing applications in Cloud environments. They provide a number of benefits. They are portable, allowing fast deployment of applications across multiple Clouds and are able to provide consistent management of applications regardless of
the infrastructure on which they run. Containers are usually run on clusters of Cloud-based VMs to provide elastically scaling capabilities. However, managing container-based web applications in distributed Cloud environments introduces numerous challenges that have formed the basis for the research undertaken in this thesis. These form the basis for the contributions made.

Specifically in this thesis, we have proposed several complementary approaches to tackle a range of challenges in managing containers and clusters used for containerised web applications in globally distributed Clouds. A core goal was to deliver QoS to end-users and support SLOs with minimum costs for applications running on multiple distributed Clouds.

Chapter 1 presented the background motivation and the thesis organisation. It identified the over-arching research questions tackled in this thesis with regards to Cloud-based deployment of container solutions and their subsequent management. In particular we identified three main research problems: 1) where to geo-replicate container-based web applications across distributed Cloud data centers to address the issues of potential data center outages and network latency to achieve high availability and performance whilst supporting SLOs; 2) how to dynamically deploy container-based clusters across data centers to manage potential spatial workload variations and hence maintain performance, SLOs and cost-efficiency demands; and 3) how to timely scale overloaded clusters at the VM level for multi-cluster container-based web applications in a cost-effective way to avoid Cloud VM provisioning delays and/or costly over-provisioning of resources. This also include how to auto-scale clusters at both the container and VM levels to optimise performance and provide better utilisation of resources.

In this chapter we consider the extent that these research problems have been addressed.

Chapter 2 reviewed the current state of the art in managing web applications in distributed Cloud environments and other related computing paradigms. We focused especially on web application requirements and identified challenges related to these
requirements with specific regard to Cloud environments. The chapter covered the current state of the art in workload management, container placement and deployment as well as related work focused on elasticity in Clouds.

Chapter 3 tackled the problem of network latencies and the impact on the global distribution of Cloud-based web applications as experienced by end users based on application performance. It explored the deployment of container clusters across distributed Cloud data centers and support for high availability (HA). An approach was presented to deploy containerised microservice-based web applications with the aim of improving HA as well as application responsiveness under normal conditions and importantly, after failures have occurred. The aim was to ensure acceptable QoS experienced by end-users. The approach was based on a single-cluster container deployment model that focused on the placement issues for container cluster nodes across distributed Cloud data centers as well as for container-based microservices across those nodes. Through generating latency-aware failover capabilities and providing deployment plans, it was shown how cluster nodes and container-based application microservices could be optimally deployed. The approach considered a SLO-based Violation model and response time models under normal and failover conditions. A cluster node placement algorithm suited for optimising data center selection was presented. This leveraged genetic algorithms that factored in the proximity of users and inter-data center latencies. The chapter also introduced an algorithm for generating the deployment configuration of primary and backup microservices. To demonstrate the effectiveness of the approach, numerous experiments were conducted on the Australia-wide NeC-TAR Research Cloud. It was shown that by considering user-to-data center and inter-data center network latencies, it was possible to mitigate geo-replication overheads caused by HA to help meet SLOs and reduce SLO violations as much as possible, under both normal conditions and in the presence of outages.

Chapter 4 considered the problem of managing spatial workload variations of web applications using container solutions. A geo-elastic container deployment approach for multi-cluster deployment was presented that leveraged the capabilities of distributed, potentially global scale Cloud environments to elastically scale web applications factoring in the geographical distribution of user load and Cloud resource availability as
well as associated (financial) costs. The approach supported intelligent deployment adaptation decisions to add, relocate and/or remove clusters across data centers. The core idea was that by sacrificing a degree of performance a reduction in operating costs could be achieved, whilst ensuring acceptable levels of user-oriented quality of service. Importantly, the approach maintained application performance even during the adaptation processes. For cluster replacement, genetic algorithms and simulated annealing were applied that considered the proximity to users and the cost of adaptation, i.e., the number of relocated/new clusters and inter-data center latency issues. A heuristic for cluster quantity adjustment was presented. The chapter also presented a framework to show how automated elastic multi-cluster deployments could be enabled based on container platforms. Extensive experiments were conducted on the NeCTAR Research Cloud using Docker and Kubernetes, where it was shown that adapting deployments with minor, but acceptable sacrifices in performance, could result in significant cost reductions.

Chapter 5 addressed the problem of container-based elasticity in the Clouds. In this chapter a cost-aware container-based elasticity approach for bursty multi-cluster containerised web applications deployed in geographically distributed and heterogeneous Cloud environments was presented. To handle sudden workload bursts, the chapter showed how it was possible to intelligently optimise the utilisation of resources between clusters running in different locations to provide a global shared computing capacity that could handle overload situation whilst factoring in VM provisioning delays. The chapter also presented an architectural framework to support inter-cluster resource management; queuing-based performance models to estimate container processing capacity without violating SLOs; two-level cluster elasticity techniques that were able to horizontally auto-scale containers and VMs based on SLO-based container capacity and high-level metrics, and dynamic container scheduling policies that took into account inter-data center placement to fit web application needs. It was shown that the approach was able to optimise launch latency for pending containers in overloaded situations whilst maintaining application performance and service level objectives (SLOs) whilst minimising operating costs. The chapter demonstrated the approach by conducting extensive experiments on the NeCTAR Research Cloud and the Amazon Cloud.
across Australia and Europe using Docker and Kubernetes. It was shown that the approach was able to maintain acceptable levels of performance and SLOs in overloaded clusters using a minimal amount of resources and thereby achieve a more cost-effective solution.

6.2 Future Directions

This thesis tackled several challenges in managing web applications in distributed Cloud environments using container-based approaches. However, as container technologies have only recently gained popularity for building and managing applications in Clouds, there are still several open challenges that need to be addressed. In this section we discuss some activities that could be worth exploring in the future.

6.2.1 Dynamic Distributed Load Balancing for Multi-Cluster Models for Container Deployment

In this thesis, we proposed a geo-elastic distributed deployment solution for container clusters in distributed Cloud environments to manage spatial workload variations. A geographical load balancing (GLB) approach can also be explored that is suitable for managing shorter-term, dynamic workloads that might arise with multi-cluster models. GLB solutions could potentially be implemented within a cross-Cloud container management layer to seamlessly and dynamically distribute user traffic across container clusters to optimise the utilisation of resources for all clusters and avoid some clusters being overloaded at given timescales. For the federated model of multi-cluster deployment, centralised GLB and admission control solutions can be adopted while decentralised GLB solutions may be more suited to independent multi-cluster container deployment models in distributed Clouds. A hybrid combination of centralised and decentralised approaches may also be worthy of further exploration.
6.2.2 Multi-objective Cloud Evaluation for Container Cluster Placement

In this thesis, we proposed placement algorithms that consider network latency and distributed deployment of container clusters across geographically distributed Cloud data centers to achieve higher availability and performance. We also considered elastic deployments to cope with spatial workload variations to maintain performance whilst factoring in operating costs. These solutions can be further enhanced by exploring multi-objective Cloud evaluation models that consider other performance, cost, and availability aspects, e.g., VM pricing, storage costs, network congestion, resource failure rates, container processing capacity, data center overload rates. Such advanced models can be used for example to rank data centers after placement algorithms produce a set of potential data centers for deployment of applications. These can potentially address finer-grained performance criteria even within given geographical areas. Such models could provide better application performance, availability and cost efficiency.

6.2.3 Efficient Global-scale Data Management for Dynamic, Fast Container-based Web Application Deployments

In this thesis, we presented dynamic deployment solutions for container-based web applications across distributed data centers. These solutions assume that the application data is already distributed across the Cloud data centers. However, the challenges of dynamic scaling and distributing data across data centers to support dynamic, fast redeployment of containerised web applications in more efficient manner needs to be further investigated. Existing solutions have considered replicating/caching data at the edge [180–183]. They focus on reducing delays caused by databases to improve application performance. However, such data-oriented solutions can be costly and unable to dynamically optimise the number and placement of data replicas. Dynamic placement solutions for lightweight application containers are not readily applicable to address data placement issues due to the potential large size of the data. New dynamic data placement solutions are essential to dynamically optimise the number and location of data replicas to be in Cloud locations to serve multiple applications and their replicas running in different data centers. Such solutions also need to consider the potential
global distribution and fast mobility of web application containers. It has been widely accepted that big data is not moved, rather the compute is brought to the data. This has implications on where clusters can/should be deployed, i.e. where the data is. This requires the elastic scaling solution to factor in data location too.

### 6.2.4 Inter-Cluster Resource Management for Cost-Efficient Scaling

A more proactive model for cluster infrastructure elasticity for containers in the Cloud could improve performance and overcome the overheads and delays associated with creation of VMs. This might use forecasting techniques to predict future workloads, estimate resource consumption for containers and then accordingly, take appropriate cluster scaling actions. For multi-cluster models of container deployments, a simple elasticity approach for each cluster requiring a scale-out action might be to provision more resources in advance for each container cluster. This approach however can lead to costly, over-provisioning deployment of resources as workload behaviour of web applications may be difficult to predict accurately and in the worse case it may be completely unpredictable. Inter-cluster resource management methods for proactive cluster elasticity solutions may allow sharing idle, already-purchased and/or reserved instances between clusters to help mitigate the impact of inaccurate resource estimation. For instance, clusters that are predicted to be overloaded can make a reservation to utilise resources in advance from clusters that are anticipated to be idle. Clusters with actual workloads that match their predicted workload need only provision their required on-demand instances. Clusters with workloads that are in excess of predicted workloads may need to utilise resources from other clusters during VM launch time. Such methods could potentially reduce unnecessary operating cost.

### 6.2.5 Dynamic Placement Policies for Multi-Cluster-aware Container Orchestration

Dynamic placement policies supporting multi-cluster aware container orchestration are another area that could be explored in future extensions to this work. Existing solutions
assume a pre-defined set of containers across federated clusters. Policies can either dis-tribute containers across clusters evenly or based on some (static) weighted distribution. However dynamic placement of some container replicas across clusters to reach a pre-defined static state of distribution may result in issues, e.g. some clusters might be out of capacity at the time of the initial placement. Further extensions to the work may investigate new orchestration policies that factor in dynamic weighted distribution of container replicas across clusters to cope with workload variations of clusters at run-time. They should intelligently place containers and dynamically replace containers at idle clusters and avoid clusters that are currently running out of capacity. In a similar vein, Cloud platform pricing models are increasingly complex and policies that lend themselves to minimising cost is another extension. For example, if the deployment of a given container or cluster at a site will result in some usage threshold being exceeded and hence higher costs incurred, then other deployment plans need to be established.

6.2.6 Dynamic Configuration of Container Disruptions by Orchestration Platforms

Some container management platforms, e.g. Kubernetes, offer container deployment configuration parameters for applications to ensure availability in the presence of potentially disruptive actions. For instance, removing nodes from a cluster for scale-in operations can cause disruption of containers running on targeted nodes that will be stopped and replaced at other nodes. The parameters for container disruptions with Kubernetes are usually manually configured, and subsequently control the number or percentage of container replicas of an application at a given time, e.g. they can be scaled up or down. For web applications in the presence of dynamic workloads, it would be interesting to explore optimisation techniques that can dynamically configure such parameters based on current workloads to maintain not only application availability, but also application performance in the presence of cost constraints. For instance, at the time of performing a disruptive cluster action, load-agnostic parameter settings that reduce the number of containers could cause performance degradation if the load on the application is high at that time.
Bibliography


189


[27] Google. CONTAINERS AT GOOGLE: A better way to develop and deploy applications. URL: https://cloud.google.com/containers.


[85] Yanwei Zhang, Yefu Wang, and Xiaorui Wang. “GreenWare: Greening cloud-scale data centers to maximize the use of renewable energy”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2011.


[190] D2IQ. *Using DC/OS to manage multiple clusters*.


[215] Google. gVisor: an application kernel for containers that provides efficient defense-in-depth anywhere. URL: https://gvisor.dev.


[221] Locust: An open source load testing tool. URL: https://locust.io.


