Future Generation Computer Systems 101 (2019) 1081–1095

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

# 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, Melbourne, 3010 Victoria, Australia
[b] Faculty of Computer and Information Systems, Islamic University of Madinah, Saudi Arabia

## HIGHLIGHTS

- We present an approach for achieving availability and performance when deploying web applications in distributed Clouds.
- A genetic algorithm for data center (DC) selection that factors in proximity to users and inter-DC latencies is presented.
- This work focuses on the placement issue and improves end-to-end response times even in the presence of failures.
- We show how latency-aware application deployment can offer higher performance and stability before and after failures.
- We present results based on realistic Cloud-based experiments across the national research Cloud in Australia.

## ARTICLE INFO

*Article history:*
Received 17 February 2019
Received in revised form 27 May 2019
Accepted 16 July 2019
Available online 25 July 2019

*Keywords:*
Distributed Clouds
High availability
Performance
Container technologies
Cloud outages
Web applications
Distributed deployment

## 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 at most1.5 ms above the SLO compared to 11–32 ms using other approaches.



## 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 been highly attractive to many organizations that wish to deploy and subsequently have their applications outsourced and hosted in Cloud environments [1].

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 [2,3]. Cloud outages, even short-term ones, can have a considerable impact [4]. 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, [5] identified that approximately 11,000 dollars per minute was lost to one business due to such an outage.

* Corresponding author at: School of Computing and Information Systems, The University of Melbourne, Melbourne, 3010 Victoria, Australia.
*E-mail addresses:* yaldwyan@student.unimelb.edu.au (Y. Aldwyan), rsinnott@unimelb.edu.au (R.O. Sinnott).

https://doi.org/10.1016/j.future.2019.07.032
0167-739X/

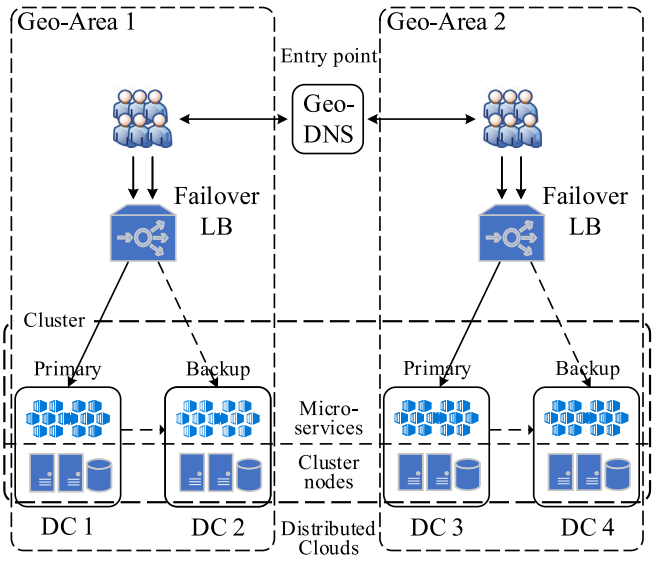



**Fig. 1.** Geographical clustering of web applications across two geo-graphical areas using container technologies in distributed Clouds.

To improve availability and other requirements such as performance and locality [6], Cloud providers have typically applied a distributed Cloud model and thus distributed their Cloud resources and services across multiple, geographically dispersed data centers (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 [7]. 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, minimizing network latency and so on [6–8]. Coady et al. [6] 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. 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 realize HAP for many modern web-based applications [4]. 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 optimizing 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 [9]. Such microservices allow to break up monolithic applications into lightweight, independently deployable services [10] that offer many advantages. Containerization 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 [1,11,12]. Microservices also help increase scalability, portability and availability of applications [10]. 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 [13]. In our work we utilize container technologies and adopt a microservice-based application architecture as shown in Fig. 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 deploying virtual servers in distributed Clouds and then deploying container based microservices on top of that infrastructure. Secondly, we present a server placement algorithm suited for optimizing 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 [14] on the NeCTAR Research Cloud. 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 min).

[1] SLOs are measurable key elements in service level agreements (SLAs).

The rest of this paper is organized as follows. Section 2 provides background and related work. We describe the application and container cluster-based infrastructure models in Section 3. In Section 4, we formulate the research problem. Section 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 6. Finally, Section 7 presents conclusions and future directions.

## 2. Background and related work

### 2.1. Background

#### 2.1.1. HAP application deployment in distributed clouds

Application deployment models have been dramatically changed by the Cloud computing paradigm [1] with many features now offered for HA solutions [6]. In the standard, centralized 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 [15].

To handle these issues, most mainstream Clouds have adopted a distributed model where Cloud resources and services are distributed across multiple, geographically dispersed DCs [6]. 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 [16,17].

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 [16,17], helps application providers improve availability and the QoS of their applications [1]. 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 [16,18]. 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 [18]), 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 back-up replicas. Placement is a key issue in geographically distributed Cloud environments and is largely left to application providers [3]. 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 [13]. 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 microservices at a given point in time.

### 2.2. 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 [19–22], migration [23,24], deployment [25], affinity rules and clustering [26] and at different failover levels: VM [19,20,25,26], container [22,24] and application [21].

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, 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 [27] and among regions [17], or using lightweight container solutions to reduce the size of snapshots that need to be transferred, compared to more heavyweight VM-based solutions [28,29]. Others [30] have proposed optimization techniques to dramatically minimize 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 paper) and use replication and/or caching to improve performance [31–33], however, most approaches simply ignore availability.

Some works [34,35] focus specifically on HAP. In [34], 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

**Table 1**
Summary of related work.

| Work | Solution | Objective | | | Computing environment | Failure level | Target application |
|---|---|---|---|---|---|---|---|
| | | HA | Performance | Performance after failover | | | |
| [19–22] | Replication | ✓ | – | – | Cloud | Within DC | General |
| [23,24] | Migration | ✓ | – | – | Cloud | Within DC | General |
| [25] | Deployment | ✓ | – | – | Cloud | Within DC | General |
| [26] | Affinity rules and clustering | ✓ | – | – | Cloud | Within DC | General |
| [17,27] | High-speed links among zones and regions | ✓ | ✓ | – | Distributed Cloud | Zone, region | General |
| [28,29] | Using container solutions to reduce size of snapshots | ✓ | ✓ | – | Distributed Cloud | Cloud | General |
| [30] | Techniques to eliminate redundant memory data and compress data | ✓ | ✓ | – | Distributed Cloud | Cloud | General |
| [31–33] | Replication/caching at the edge | – | ✓ | – | Edge | – | Web apps |
| [34] | Query caching at the edge | ✓ | ✓ | – | Edge | Server | Web apps |
| [35] | Replication at the edge | ✓ | ✓ | – | Edge | Network | Web apps |
| [36] | Application deployment | – | ✓ | – | Distributed Cloud | – | Web apps |
| [37] | Application deployment | – | ✓ | – | Distributed Cloud | – | Service-oriented apps |
| [38] | Application deployment | – | ✓ | – | Distributed Cloud | – | Virtual desktops |
| [39] | Application deployment | ✓ | – | – | Distributed Cloud | Node, link | Service-oriented apps |
| Present work | Application deployment | ✓ | ✓ | ✓ | Distributed Cloud | DC, zone, Cloud | Containerized Web apps |

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 [35], 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. [34,35] 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 [36–40]. In [36], the authors aim to improve performance, by minimizing user-to-DC and inter-DC latencies to achieve strong consistency for web applications. [37] takes into account performance and service dependencies for service-oriented applications, while the approach in [38] takes into consideration proximity to users only to improve response times using virtual desktops as the example application. [39] considers availability as well as resource utilization. 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. Table 1 shows a summary of related work.

## 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 geo-area. As shown in Fig. 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

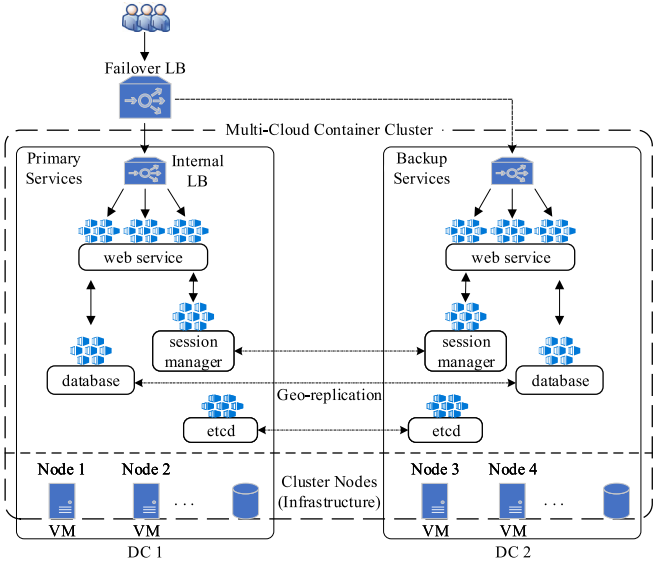



**Fig. 2.** Geo-redundant model of a microservice based web application across a pair of DCs in a geo-area.

(geo-DNS) service, similar to Amazon Route 53 [41], 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 Fig. 2. A full copy (or multiple copies)

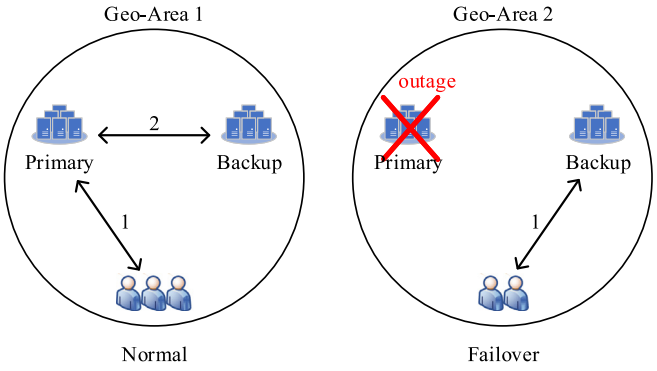



**Fig. 3.** Servicing requests under normal and failover conditions.

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 [42]. 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 do not have access to the underlying hypervisors for these Cloud. Each one of these servers should have a container engine (e.g. Docker Engine [43]) installed and configured as part of a container-based clustering system (e.g. Docker Swarm [44] or Kubernetes [13]). 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.

## 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 in Cloud context, 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 Fig. 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 optimization problem where the objective function aims to minimize 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, 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 2, given $N, C$, $\mathbf{U}$, $\mathbf{D}$, we want to select a set of DC pairs, $\mathbf{H} = \{(DC_1, DC_2), \ldots, (DC_j, DC_k)\}$, where $j \neq k \forall\ (DC_j, DC_k) \in \mathbf{H}$ and $|\mathbf{H}| = N$, to place a number of servers ($C$) at each DC in $\mathbf{H}$ such that the objective functions under normal (Eq. (1)) and failover (Eq. (2)) conditions are minimized, i.e. predefined SLOs are satisfied and the average user end-to-end response time is minimized even in the presence of outages.

$$\begin{aligned} &\text{minimize} \quad \sum_i \beta_i \cdot \min_{(DC_j, DC_k) \in \mathbf{H}} (nl^p_{ij} + il_{jk}) \quad \forall i \in \mathbf{U} \\ &\text{subject to} \quad \mathbf{H} \subset \mathbf{A},\ |\mathbf{H}| = N \end{aligned} \tag{1}$$

$$\begin{aligned} &\text{minimize} \quad \sum_i \beta_i \cdot \min_{(DC_j, DC_k) \in \mathbf{H}} nl^b_{ik} \quad \forall i \in \mathbf{U} \\ &\text{subject to} \quad \mathbf{H} \subset \mathbf{A},\ |\mathbf{H}| = N \end{aligned} \tag{2}$$

In Eq. (1), $\min_{(DC_j, DC_k) \in \mathbf{H}} (nl^p_{ij} + il_{jk})$ denotes that the user $i$ will be served by the primary-backup DC pair $(DC_j, DC_k)$ in $\mathbf{H}$ 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 $\min_{(DC_j, DC_k) \in \mathbf{H}} nl^b_{ik}$, in Eq. (2), denotes that the user $i$ will be served by the backup DC $k$ of $(DC_j, DC_k)$ in $\mathbf{H}$ 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 2, given a set of primary microservices, $\mathbf{M}^p = \{\mu^p_1, \ldots, \mu^p_x\}$, and their backups, $\mathbf{M}^b = \{\mu^b_1, \ldots, \mu^b_x\}$, and $\mathbf{H}$, we want to place $\mathbf{M}^p$ in cluster nodes in $DC_j$ and $\mathbf{M}^b$ in nodes in $DC_k$, $\forall\ (DC_j, DC_k) \in \mathbf{H}$. A service deployment configuration should be generated autonomously considering placement constraints. The resultant solution should subsequently be sent to the container cluster scheduler to actually deploy containers for microservices.

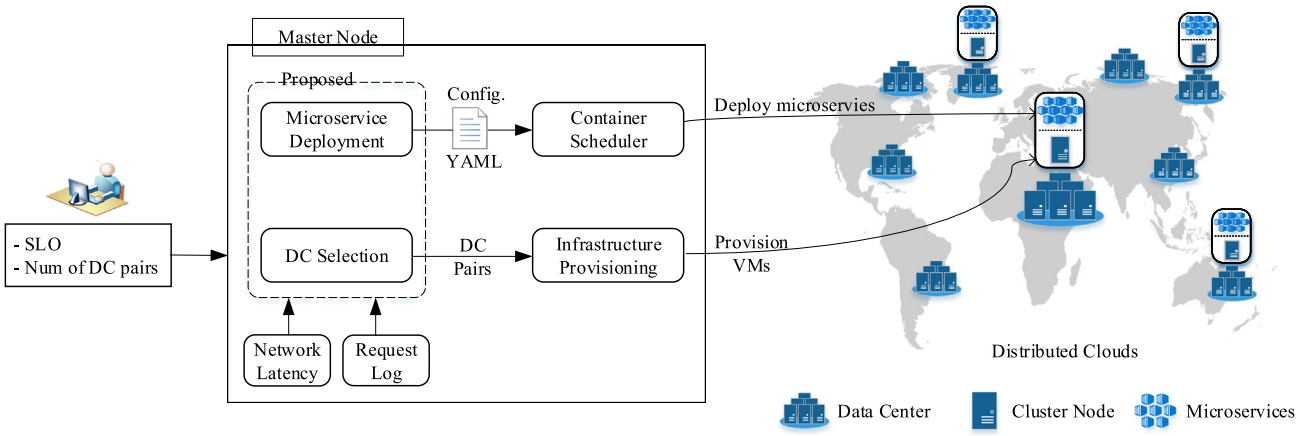



**Fig. 4.** HAP microservice-based application deployment in distributed Clouds.

**Table 2**
Terms used in the models.

| Term | Meaning |
|---|---|
| **D** | Set of available DCs |
| **A** | Set of available DC pairs |
| **H** | Set of the selected DC pairs for hosting microservices |
| **U** | Set of application users |
| **G** | Set of geo-areas, servicing different geographical user workloads |
| $\mathbf{M}^p$ | Set of microservices to be placed in a primary DC |
| $\mathbf{M}^b$ | Set of backup microservices to be placed in a backup DC |
| **W** | Set of write requests |
| **R** | Set of read requests |
| $C$ | Number of servers (cluster nodes) to be placed at each DC |
| $N$ | Number of required primary-backup DC pairs (geo-areas) |
| $g$ | Number of generations (iterations) |
| $p$ | Number of chromosomes (solutions) in the population |
| $r$ | Crossover rate |
| $m$ | Mutation rate |
| $\beta_i$ | Number of requests made by user $i$ |
| $V_a^c$ | Number of violated sessions at geo-area $a$ under condition $c$ |
| $S_a^c$ | Number of sessions at geo-area $a$ under condition $c$ |
| $nl_{ij}^p$ | Estimated (RTT) network latency between user $i$ and primary DC $j$ if that user is served by microservices in DC $j$ |
| $il_{jk}$ | Estimated inter-DC (RTT) network latency between primary DC $j$ and backup DC $k$ |
| $nl_{ik}^b$ | Estimated network latency (RTT) between user $i$ and backup DC $k$ if that user is served by microservices in DC $k$ |

## 5. Proposed approach

To address the challenges in deploying containerized application microservices in distributed Clouds, we propose an approach to optimize and autonomously generate the deployment while minimizing the total amount of estimated SLO violations under normal and failover conditions. The deployment configuration forms an HAP geo-cluster.

### 5.1. Requirements and assumptions

As illustrated in Fig. 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 [45], to predict unknown latencies or using third party services like Ookla [46]. 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.

### 5.2. The overall algorithm

Once an application administrator provides the requirements, as shown in Fig. 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 infrastructure provisioning and the microservice deployment components. This step is discussed in detail in Section 5.4.

**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. This step also uses labeling 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 labeled, 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 5.5.

### 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.

[2] www.ip2location.com.

*5.3.1. 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 2, the response time for write requests under normal condition can modeled as:

$$\alpha_w^n = nl_{ij}^p + \rho_w^p + \gamma_w \qquad w \in \mathbf{W},\ i \in \mathbf{U} \tag{3}$$

where $\alpha_w^n$ is the response time of the write request $w$ under normal condition $n$, $\rho_w^p$ 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:

$$\gamma_w = il_{jk} + \rho_w^b \qquad w \in \mathbf{W},\ (DC_j, DC_k) \in \mathbf{H} \tag{4}$$

where $\rho_w^b$ is the processing time of the write request $w$ at the backup DC $b$. Therefore, using Eqs. (3) and (4), the response time of a write request, $\alpha_w^n$, under normal conditions can be represented as:

$$\alpha_w^n = nl_{ij}^p + \rho_w^p + il_{jk} + \rho_w^b \qquad w \in \mathbf{W},\ i \in \mathbf{U},\ (DC_j, DC_k) \in \mathbf{H} \tag{5}$$

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) where geo-replication overheads can be omitted:

$$\alpha_r^n = nl_{ij}^p + \rho_r^p \qquad r \in \mathbf{R},\ i \in \mathbf{U} \tag{6}$$

where $\alpha_r^n$ is the response time of the read request $r$ under normal condition $n$ and $\rho_r^p$ is the time needed to process the read request $r$ at the primary DC $p$.

*5.3.2. 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 2, a response time of a request after a failover can be given as:

$$\alpha_t^f = nl_{ik}^b + \rho_t^b \qquad t \in \mathbf{R} \cup \mathbf{W},\ i \in \mathbf{U} \tag{7}$$

where $\alpha_t^f$ is the response time of the request $t$ after a failover $f$ and $\rho_t^b$ is the processing time of the request $t$ at the backup DC $b$.

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

*5.3.3. 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 2, the definition of the model is given as:

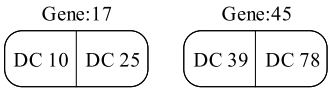



**Fig. 5.** A chromosome with 2 selected genes (DC pairs).

$$\begin{aligned} &\text{minimize} \quad f_{SBVR}(\mathbf{H}) = \sum_c \frac{\sum_a V_a^c}{\sum_a S_a^c} \qquad \forall a \in \mathbf{H}, c \in \{n, f\} \\ &\text{subject to} \quad \mathbf{H} \subset \mathbf{A},\ |\mathbf{H}| = N \end{aligned} \tag{8}$$

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.

*5.4. Genetic algorithm (GA) for DC selection*

In this section, 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.

*5.4.1. GA overview*

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 optimization solutions. GA has two advantages. Firstly, it is easy to use meta-heuristics to optimize multiple objective functions. In our context, we have two objective functions that need to be minimized 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 genetic representation used to encode the candidate solutions from the problem domain, and a fitness function used for ranking those solutions.

A GA initially 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.

*5.4.2. 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 Fig. 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:

$$fitness(\mathbf{H}) = 1 - f_{SBVR}(\mathbf{H}) \qquad \mathbf{H} \subset \mathbf{A},\ |\mathbf{H}| = N \tag{9}$$

A GA comprises various phases: initialization, selection, crossover and mutation. In the **initialization 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 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.

#### 5.4.3. Computational complexity of proposed GA

Using the terms in Table 2, the running time of the selection phase requires $O((1-r)\cdot p)$ while the crossover phase runs in $O(r \cdot p)$. The running time of the mutation phase is in $O(m \cdot p)$. 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 $\mathbf{U}$ and this requires $(N-1)$ comparisons per user. Therefore, evaluating a chromosome requires $O(|\mathbf{U}| \cdot N)$ and hence the evaluation phase will be in $O(p \cdot |\mathbf{U}| \cdot N)$. Based on the running times of the above-mentioned phases, the efficiency of an entire iteration to create a new generation is given by $O(max\{(1-r)\cdot p,\ r \cdot p,\ m \cdot p,\ p \cdot |\mathbf{U}| \cdot N\}) = O(p \cdot |\mathbf{U}| \cdot N)$. The time complexity of the GA is given by $O(g \cdot p \cdot |\mathbf{U}| \cdot N)$

### 5.5. Microservice deployment

Once the GA produces a list of DC pairs ($\mathbf{H}$) and a number of required cluster nodes ($C$) are provisioned at each DC and labeled according to their geo-areas and DCs to which they belong, it is necessary to place primary microservices ($\mathbf{M}^p$) and secondary microservices ($\mathbf{M}^b$) in those nodes such that $\mathbf{M}^p$ and $\mathbf{M}^b$ 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 customize how the cluster scheduler places containers of microservices to meet their applications' requirements [47]. 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.

**Algorithm 1:** Microservice Deployment Generator

**Input** : List of selected DC pairs, $\mathbf{H}$, $\mathbf{M}^p = \{\mu_1^p, \ldots, \mu_x^p\}$, $\mathbf{M}^b = \{\mu_1^b, \ldots, \mu_x^b\}$
**Output**: Deployment Configuration (YAML file)

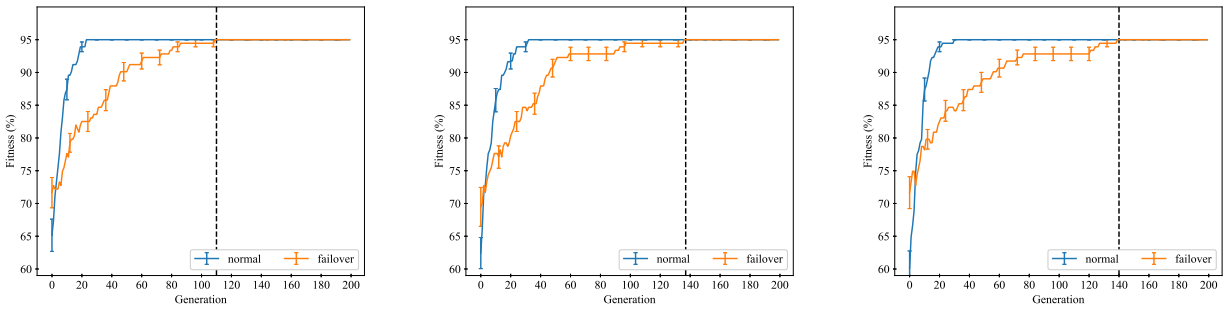
```
/* Define microservices and their backup in each
   geo-area                                          */
foreach g = (dc_j, dc_k) in H do
    /* Define microservices at primary DC j in
       geo-area g                                    */
    foreach μ^p in M^p do
        define μ^p;
        add placement constraints geo-area = g, dc = j,
        dc_type = primary;
    end
    /* Define redundant microservices at backup DC
       j in geo-area g                               */
    foreach μ^b in M^b do
        define microservice μ^b;
        add placement constraint geo-area=g, dc = k, dc_type
        = backup;
    end
end
```

To tackle this, we propose an algorithm for autonomously generating the deployment configuration (Algorithm 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 $\mathbf{H}$. At each iteration, it defines microservices in $\mathbf{M}^p$ 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 ($\mathbf{M}^b$) to be deployed in the backup DC ($k$).

## 6. Performance evaluation

We evaluate our approach by conducting experiments on the Australia-wide National eResearch Collaboration Tools and Resources (NeCTAR - www.nectar.org.au) research Cloud. 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.




(a) Crossover: 0.7, Mutation: 0.5 (best) (b) Crossover: 0.5, Mutation: 0.7 (c) Crossover: 0.7, Mutation: 0.7

**Fig. 6.** Convergence curves for proposed GA using various parameter settings. GA with each setting had 35 runs. Parameter setting for the figure on the left shows fastest rate of convergence. The higher the fitness, the better. The best fitness is 95% for both conditions.

### 6.1. 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 Section 6.3.2. 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.

### 6.2. 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 choose five different locations within Australia as primary sources of the workload: Brisbane, Canberra, Melbourne, Sydney and Tasmania. In addition to the spatial characteristic of the workload, we set a user session to include 10 requests and categorize 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.

### 6.3. 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 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 value (%) 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.

#### 6.3.1. 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.

**Table 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.

| GA parameters | | Generations to convergence (all runs) | Average fitness (%) | | | |
|---|---|---|---|---|---|---|
| Crossover | Mutation | | Normal | | Failover | |
| | | | Estimate | SE | Estimate | SE |
| 0.7 | 0.5 | 110 | 93.86 | 0.31 | 91.24 | 0.42 |
| 0.5 | 0.7 | 137 | 93.61 | 0.33 | 91.09 | 0.43 |
| 0.7 | 0.7 | 140 | 93.70 | 0.36 | 90.77 | 0.39 |
| 0.5 | 0.5 | 146 | 93.71 | 0.36 | 90.45 | 0.46 |
| 0.7 | 0.2 | 169 | 93.63 | 0.36 | 88.03 | 0.55 |

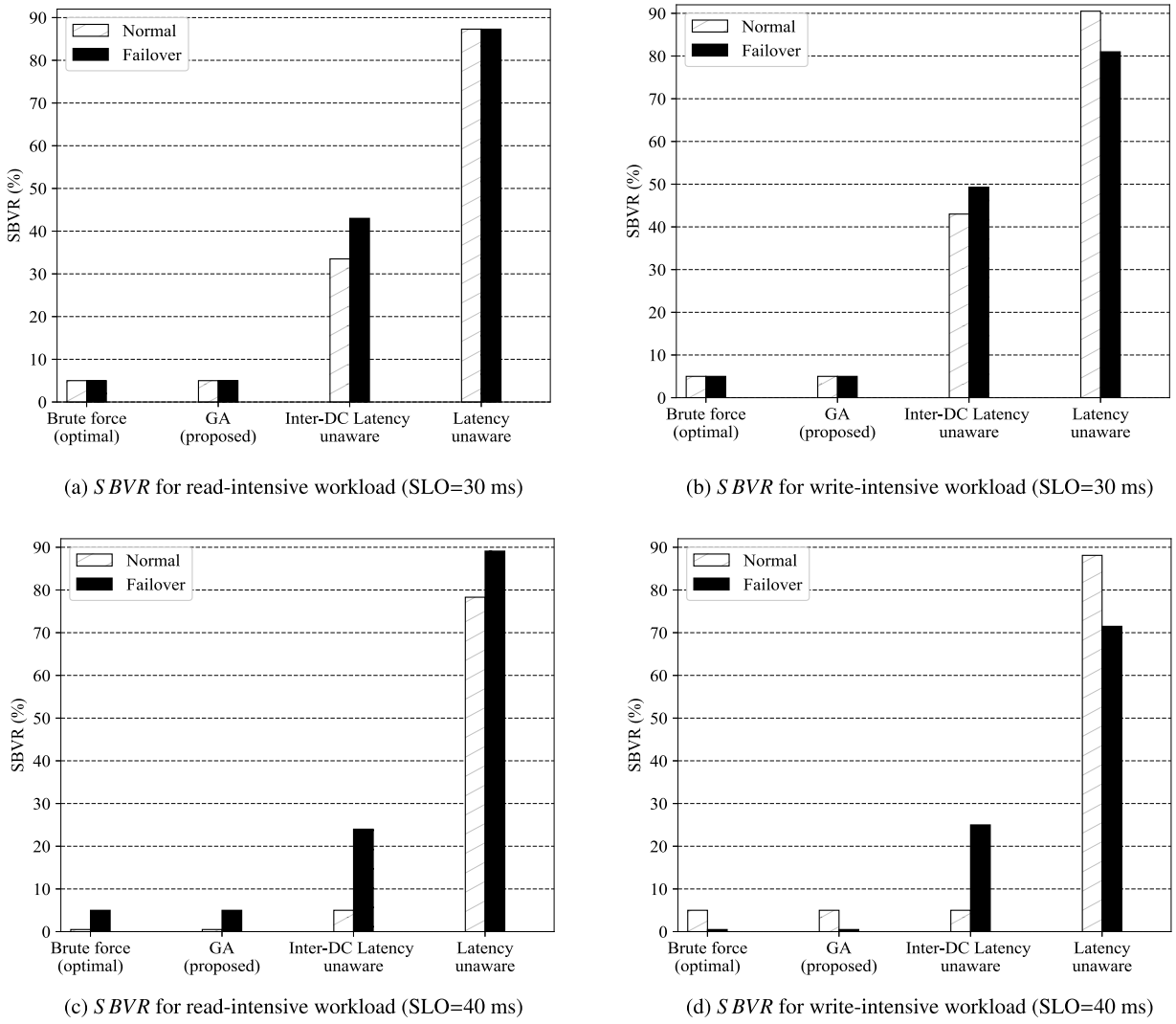
#### 6.3.2. GA parameter tuning

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 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 upper bound of 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 Fig. 6 for the best three parameter settings shows that the chosen parameter setting (Fig. 6a) is the best one for our problem.

#### 6.3.3. 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 30 ms 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 40 ms.




(a) *S BVR* for read-intensive workload (SLO=30 ms)

(b) *S BVR* for write-intensive workload (SLO=30 ms)

(c) *S BVR* for read-intensive workload (SLO=40 ms)

(d) *S BVR* for write-intensive workload (SLO=40 ms)

**Fig. 7.** Performance comparisons of different algorithms using 2 primary-backup DC pairs and various workloads and SLOs under normal and failover conditions.

The results presented in Fig. 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, while taking into account both proximity to users and inter-DC latencies in the GA as well as the optimal algorithm indicates substantial improvements in *SBVR*. From the results, it is evident that network latency consideration plays a crucial rule in improving performance in all cases.

**Table 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.

| Algorithm | SBVR (%) | | | | | |
|---|---|---|---|---|---|---|
| | Normal | | | Failover | | |
| | 95% CI | Estimate | SE | 95% CI | Estimate | SE |
| GA | (5.42, 5.91) | 5.67 | 0.12 | (7.44, 9.16) | 8.30 | 0.43 |
| ILU | (50.17, 71.26) | 60.72 | 5.19 | (51.69, 76.29) | 63.99 | 6.05 |
| LU | (91.11, 100.16) | 95.64 | 2.23 | (74.31, 94.05) | 84.18 | 4.86 |

### 6.3.4. 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 min, whilst it takes the brute force algorithm about 26 h to find the optimal solution. Furthermore, the GA solution has the same *SBVR* as the optimal one as shown in Fig. 7.

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 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 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* 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.

**Table 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.

| Difference in mean | SBVR (%) | | | | | |
|---|---|---|---|---|---|---|
| | Normal | | | Failover | | |
| | 95% CI | Estimate | P-value | 95% CI | Estimate | P-value |
| ILU - GA | (44.50, 65.60) | 55.05 | 0.00 | (43.36, 68.02) | 55.69 | 0.00 |
| LU - GA | (85.44, 94.50) | 89.97 | 0.00 | (65.97, 85.79) | 75.88 | 0.00 |
| LU - ILU | (23.55, 46.29) | 34.92 | 0.00 | (4.68, 35.69) | 20.19 | 0.01 |

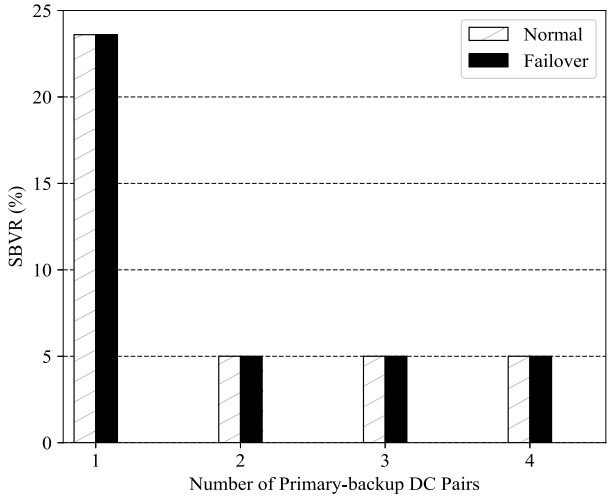



**Fig. 8.** Performance comparison of distributed deployments using various numbers of DC pairs and the optimal single DC pair deployment.

#### 6.3.5. 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 30 ms 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 Fig. 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.

### 6.4. 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 proposed approach can improve performance, reduce pre-agreed SLO violations and maintain performance variability before and after a failover for geographically distributed users whilst realizing 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.

#### 6.4.1. 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 s. For each run, the first half of a run is under normal conditions 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 300th second), we deliberately 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.

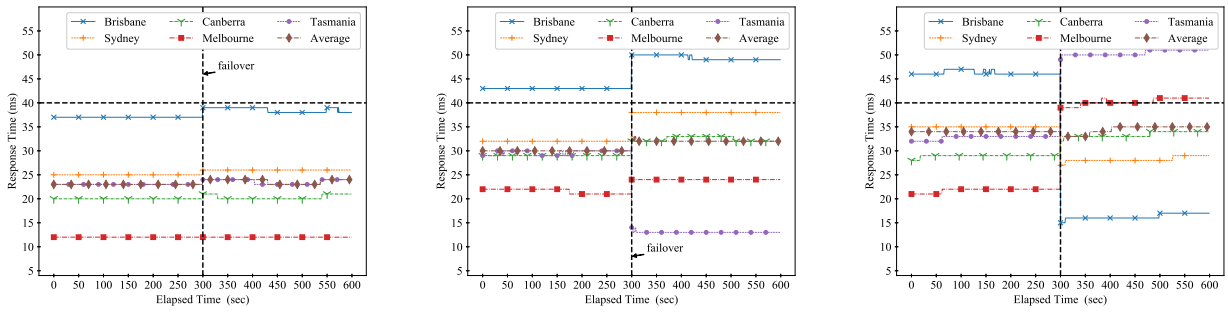
With regards to workloads, to simulate geo-distributed users, we generate different workloads from five different locations across Australia. At each location, we provision a VM (4 VCPUs and 16 GB of RAM) from a Cloud DC located at that location. For each experiment, we simulate 825 simultaneous users (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.

#### 6.4.2. Container cluster settings and sample application

We use Docker Swarm [44] 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 [14], which models an online bookstore. Initially, the architecture of TPC-W application is monolithic, hence we transform the application into microservices. 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 [48] as a microservice for coordination across a cluster of machines. These microservices are illustrated in Fig. 2.

We containerize 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.




(a) GA (proposed) (b) Inter-DC latency unaware (c) Latency unaware

**Fig. 9.** Performance comparison of 3 deployments for read-intensive workloads generated from 5 different locations around Australia. First half of each 600-s deployment is under normal conditions, while the second one is after a failover. (SLO = 40 ms).

**Table 6**
95% Confidence Intervals (CI) and estimates of the mean and standard deviation (StDev) 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 = 40 ms).

| Workload location | GA (proposed) | | | ILU | | | LU | | |
|---|---|---|---|---|---|---|---|---|---|
| | 95% CI | Estimate | StDev | 95% CI | Estimate | StDev | 95% CI | Estimate | StDev |
| Brisbane | (37.69, 37.83) | 37.76 | 0.83 | (45.94, 46.46) | 46.2 | 3.22 | (30.07, 32.48) | 31.28 | 14.99 |
| Sydney | (25.46, 25.54) | 25.50 | 0.50 | (34.76, 35.24) | 35.00 | 3.00 | (31.33, 31.88) | 31.61 | 3.41 |
| Canberra | (20.12, 20.18) | 20.15 | 0.36 | (30.58, 30.87) | 30.73 | 1.76 | (30.98, 31.34) | 31.16 | 2.26 |
| Melbourne | (12.00, 12.00) | 12.00 | 0.00 | (22.69, 22.89) | 22.79 | 1.26 | (30.29, 31.78) | 31.03 | 9.27 |
| Tasmania | (23.24, 23.31) | 23.28 | 0.45 | (20.67, 22.01) | 21.34 | 8.34 | (40.90, 42.32) | 41.61 | 8.83 |

### 6.4.3. Results and discussions

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

Figs. 9 and 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.

Figs. 9a and 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 Figs. 9b and 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.

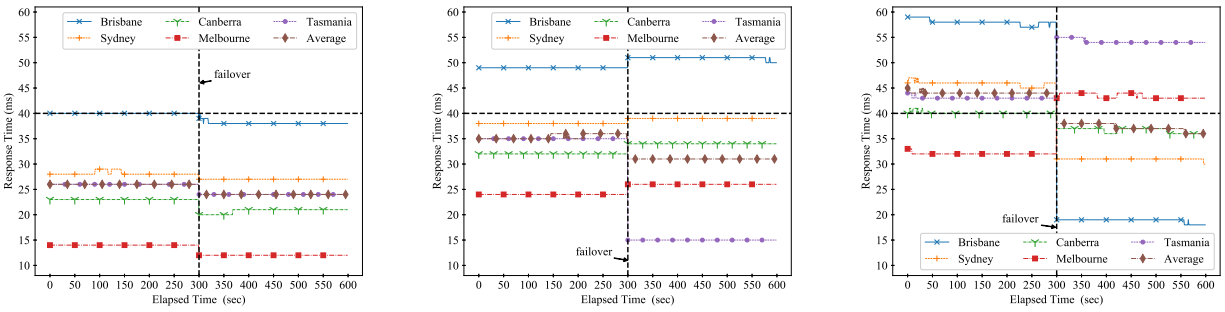
Moreover, with the LU approach, the results presented in Figs. 9c and 10c indicate that response times for most locations are higher and go beyond the SLO for write-intensive workloads while some locations have a degradation in performance for read-intensive workloads. This is because most requests during 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.

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.*

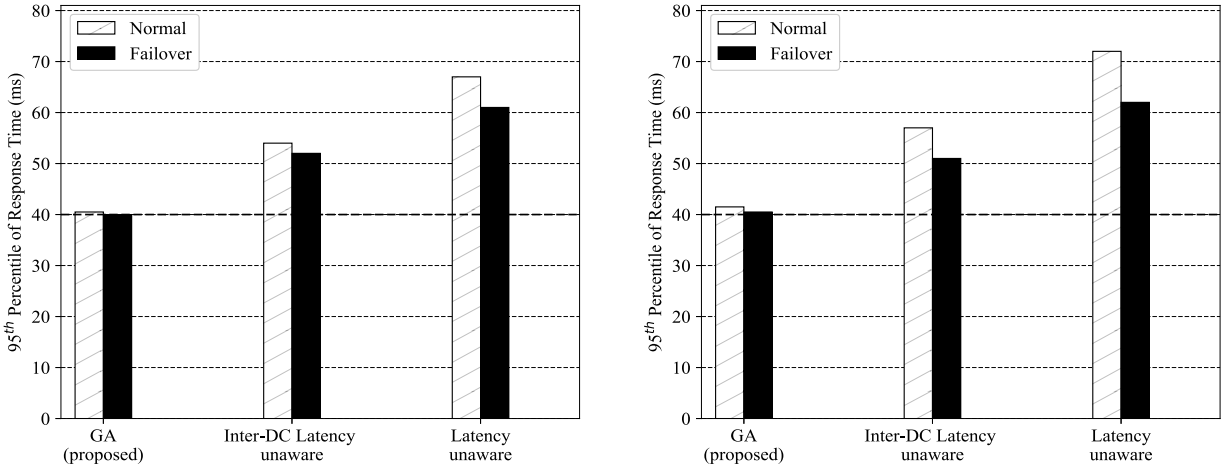
In this section, we show how our approach is able to meet SLOs or minimize SLO violations even in the presence of Cloud outages. Fig. 11 displays the 95th percentile for the response time under normal and after failover of the three approaches using read and write-intensive workloads. For the read workload, as shown in Fig. 11a, our GA shows only 0.5 ms violation in response times based on the pre-agreed SLO (40 ms) 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 ms and 12 ms under normal and failover conditions respectively. Furthermore, the LU deployment is worse still and shows 27 ms and 21 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, Fig. 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 Fig. 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 ms above the pre-agreed SLO while it is up to at least 11 ms (and up to 32 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




(a) GA (proposed) (b) Inter-DC latency unaware (c) Latency unaware

**Fig. 10.** Performance comparison of 3 deployments for write-intensive workloads generated from 5 different locations around Australia. (SLO = 40 ms).




(a) Read-intensive workload (b) Write-intensive workload

**Fig. 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 7**
95% Confidence Intervals (CI) and estimates of the mean and standard deviation (StDev) 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 = 40 ms).

| Workload location | GA (proposed) | | | ILU | | | LU | | |
|---|---|---|---|---|---|---|---|---|---|
| | 95% CI | Estimate | StDev | 95% CI | Estimate | StDev | 95% CI | Estimate | StDev |
| Brisbane | (38.95, 39.11) | 39.03 | 0.99 | (49.89, 50.05) | 49.97 | 0.98 | (36.87, 40.01) | 38.44 | 19.60 |
| Sydney | (27.53, 27.63) | 27.58 | 0.63 | (38.46, 38.54) | 38.50 | 0.50 | (37.84, 39.04) | 38.44 | 7.47 |
| Canberra | (21.79, 21.98) | 21.89 | 1.15 | (32.92, 33.08) | 33.00 | 1.00 | (38.23, 38.50) | 38.36 | 1.74 |
| Melbourne | (12.92, 13.08) | 13.00 | 1.00 | (24.92, 25.08) | 25.00 | 1.00 | (37.267, 38.18) | 37.72 | 5.71 |
| Tasmania | (24.92, 25.08) | 25.00 | 1.00 | (24.20, 25.80) | 25.00 | 10.01 | (48.17, 49.06) | 48.61 | 5.59 |

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 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.

Tables 6 and 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 of response times are at most 0.83 ms (i.e. Brisbane workload) for read workloads and 1.15 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 ms and 10.01 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 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 ms, 9.27 ms and

8.83 ms respectively for read workload and 19.60 ms, 7.47 ms and 5.59 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.

## 7. Conclusions and future directions

We have introduced a new approach to autonomously deploy containerized microservice-based web 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 NeCTAR 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 of applications in heavily-loaded locations and thus impact SLOs. To handle this problem, we are considering geo-elastic and load balancing techniques that are aware of the geo-location of idle, availabile computing capacity in other DCs and exploiting the ability of containers to deploy application replicas with speed at scale to cope with sudden workload fluctuations. This location information can be used to decide where to deploy new containers and route user requests so that the response times can be improved rapidly, by reducing network latencies and having enough capacity, and thus the amount of SLO violations are minimized.

## Declaration of competing interest

There are no conflicts of interest with this work.

## References


[1] R. Ranjan, The Cloud interoperability challenge, IEEE Cloud Comput. 1 (2) (2014) 20–24, http://dx.doi.org/10.1109/MCC.2014.41.

[2] P.T. Endo, M. Rodrigues, G.E. Gonçalves, J. Kelner, D.H. Sadok, C. Curescu, High availability in Clouds: systematic review and research challenges, J. Cloud Comput. 5 (1) (2016) 16.

[3] M. Nabi, M. Toeroe, F. Khendek, Availability in the Cloud: State of the art, J. Netw. Comput. Appl. 60 (2016) 54–67.

[4] E. Nygren, R.K. Sitaraman, J. Sun, The akamai network: A platform for high-performance internet applications, SIGOPS Oper. Syst. Rev. 44 (3) (2010) 2–19.

[5] Emerson Network Power, Understanding the cost of data center downtime: An analysis of the financial impact on infrastructure vulnerability. URL https://www.anixter.com/content/dam/Suppliers/Liebert/White%20Paper/Downtime/%20-%20data-center-uptime_24661-R05-11.pdf.

[6] Y. Coady, O. Hohlfeld, J. Kempf, R. McGeer, S. Schmid, Distributed Cloud computing: Applications, status quo, and challenges, ACM SIGCOMM Comput. Commun. Rev. 45 (2) (2015) 38–43.

[7] R. Moreno-Vozmediano, R. Montero, E. Huedo, I. Llorente, Orchestrating the deployment of high availability services on multi-zone and multi-Cloud scenarios, J. Grid Comput. (2017) 1–15.

[8] F. Nawab, D. Agrawal, A. El Abbadi, The challenges of global-scale data management, in: Proceedings of the 2016 International Conference on Management of Data, in: SIGMOD '16, ACM, New York, NY, USA, 2016, pp. 2223–2227, http://dx.doi.org/10.1145/2882903.2912571, URL http://doi.acm.org/101145/28829032912571.

[9] C. Pahl, A. Brogi, J. Soldani, P. Jamshidi, Cloud container technologies: a state-of-the-art review, IEEE Trans. Cloud Comput. (2017) 1, http://dx.doi.org/10.1109/TCC.2017.2702586.

[10] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, M. Villari, Open issues in scheduling microservices in the Cloud, IEEE Cloud Comput. 3 (5) (2016) 81–88, http://dx.doi.org/10.1109/MCC.2016.112.

[11] C. Pahl, B. Lee, Containers and clusters for edge Cloud architectures – a technology review, in: 2015 3rd International Conference on Future Internet of Things and Cloud, 2015, pp. 379–386, http://dx.doi.org/10.1109/FiCloud.2015.35.

[12] B.D. Martino, G. Cretella, A. Esposito, Advances in applications portability and services interoperability among multiple Clouds, IEEE Cloud Comput. 2 (2) (2015) 22–28, http://dx.doi.org/10.1109/MCC.2015.38.

[13] Kubernetes, Kubernetes: an Open-source System for Automated Container Deployment, Scaling, and Management (2018). URL https://kubernetes.io.

[14] Transaction Processing Performance Council, Tpc-w: a transactional web e-commerce benchmark (2018). URL http://www.tpc.org/tpcw.

[15] D.S. Linthicum, Approaching Cloud computing performance, IEEE Cloud Comput. 5 (2) (2018) 33–36, http://dx.doi.org/10.1109/MCC.2018.022171665.

[16] J. Barr, A. Narin, J. Varia, Building Fault-Tolerant Applications on aws (2011). URL https://media.amazonwebservices.com/AWS_Building_Fault_Tolerant_Applications.pdf.

[17] T. Keane, Introducing Azure Availability Zones for resiliency and high availability (2017). URL https://azure.microsoft.com/en-au/blog/introducing-azure-availability-zones-for-resilien-and-high-availability.

[18] Microsoft Azure, Overview: Active geo-replication and auto-failover groups (2018). URL https://azure.microsoft.com/en-au/blog/introducing-azure-availability-zones-for-resilien-and-high-availability.

[19] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, A. Warfield, Remus: High availability via asynchronous virtual machine replication, in: 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08), USENIX Association, San Francisco, CA, 2008, URL https://www.usenix.org/conference/nsdi-08/remus-high-availability-asynchronous-virtual-machine-replication.

[20] U.F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, A. Warfield, Remusdb: Transparent high availability for database systems, VLDB J. Int. J. Very Large DataBases 22 (1) (2013) 29–45.

[21] A. Kanso, Y. Lemieux, Achieving high availability at the application level in the Cloud, in: 2013 IEEE Sixth International Conference on Cloud Computing, 2013, pp. 778–785, http://dx.doi.org/10.1109/CLOUD.2013.24.

[22] H.V. Netto, L.C. Lung, M. Correia, A.F. Luiz, L.M.S. de Souza, State machine replication in containers managed by kubernetes, J. Syst. Archit. 73 (2017) 53–59.

[23] H. Liu, H. Jin, X. Liao, L. Hu, C. Yu, Live migration of virtual machine based on full system trace and replay, in: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing, ACM, 2009, pp. 101–110.

[24] S. Nadgowda, S. Suneja, N. Bila, C. Isci, Voyager: Complete container state migration, in: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), 2017, pp. 2137–2142, http://dx.doi.org/10.1109/ICDCS.2017.91.

[25] J.Z. Li, Q. Lu, L. Zhu, L. Bass, X. Xu, S. Sakr, P.L. Bannerman, A. Liu, Improving availability of Cloud-based applications through deployment choices, in: Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on, IEEE, 2013, pp. 43–50.

[26] A. Gulati, A. Holler, M. Ji, G. Shanmuganathan, C. Waldspurger, X. Zhu, Vmware distributed resource management: Design, implementation, and lessons learned, VMware Techn. J. 1 (1) (2012) 45–64.

[27] Amazon, Regions and availability zones. URL https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html.

[28] A. Reber, Container migration around the world. URL https://rhelblog.redhat.com/2017/10/12/container-migration-around-the-world/.

[29] C. Horn, How can process snapshot/restore help save your day? (2017). URL https://www.redhat.com/en/blog/how-can-process-snapshotrestore-help-save-your-day.

[30] T. Wood, K.K. Ramakrishnan, P. Shenoy, J.V. der Merwe, J. Hwang, G. Liu, L. Chaufournier, Cloudnet: Dynamic pooling of Cloud resources by live WAN migration of virtual machines, IEEE/ACM Trans. Netw. 23 (5) (2015) 1568–1583, http://dx.doi.org/10.1109/TNET.2014.2343945.

[31] M. Rabinovich, Z. Xiao, A. Aggarwal, Computing on the edge: A platform for replicating internet applications, in: Web Content Caching and Distribution, Springer, 2004, pp. 57–77.
[32] K. Amiri, S. Park, R. Tewari, S. Padmanabhan, Dbproxy: A dynamic data cache for web applications, in: Null, IEEE, 2003, p. 821.
[33] S. Sivasubramanian, G. Pierre, M. Van Steen, G. Alonso, Globecbc: Content-Blind Result Caching for Dynamic Web Applications, Tech. rep., Technical Report IR-CS-022, Vrije Universiteit, Amsterdam, The Netherlands, 2006.
[34] L. Rilling, S. Sivasubramanian, G. Pierre, High availability and scalability support for web applications, in: Applications and the Internet, 2007. SAINT 2007. International Symposium on, IEEE, 2007, p. 5.
[35] L. Gao, M. Dahlin, A. Nayate, J. Zheng, A. Iyengar, Application specific data replication for edge services, in: Proceedings of the 12th International Conference on World Wide Web, ACM, 2003, pp. 449–460.
[36] C. Qu, R.N. Calheiros, R. Buyya, SLO-Aware deployment of web applications requiring strong consistency using multiple Clouds, in: Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on, IEEE, 2015, pp. 860–868.
[37] J. Zhu, Z. Zheng, Y. Zhou, M.R. Lyu, Scaling service-oriented applications into geo-distributed Clouds, in: Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on, IEEE, 2013, pp. 335–340.
[38] T. Guo, V. Gopalakrishnan, K. Ramakrishnan, P. Shenoy, A. Venkataramani, S. Lee, Vmshadow: Optimizing the performance of virtual desktops in distributed Clouds, in: Proceedings of the 4th Annual Symposium on Cloud Computing, ACM, 2013, p. 42.
[39] R. Mennes, B. Spinnewyn, S. Latré, J.F. Botero, GRECO: A distributed genetic algorithm for reliable application placement in hybrid Clouds, in: 2016 5th IEEE International Conference on Cloud Networking (Cloudnet), IEEE, 2016, pp. 14–20.
[40] M. Alicherry, T. Lakshman, Network aware resource allocation in distributed Clouds, in: Infocom, 2012 Proceedings IEEE, IEEE, 2012, pp. 963–971.
[41] Amazon, Amazon Route 53: a scalable and highly available Domain Name System (dns) (2018). URL https://aws.amazon.com/route53.
[42] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H.C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al., Scaling memcache at facebook, in: Nsdi, Vol. 13, 2013, pp. 385–398.
[43] Docker Inc., Docker Engine: Open-source Containerization Technology (2018). URL https://www.docker.com/products/docker-engine.
[44] Docker Inc., Docker Swarm: Native Cluster Management and Orchestration for Docker containers (2018). URL https://docs.docker.com/engine/swarm.
[45] M. Szymaniak, G. Pierre, M. van Steen, Scalable cooperative latency estimation, in: Proceedings. Tenth International Conference on Parallel and Distributed Systems, 2004. ICPADS 2004, 2004, pp. 367–376, http://dx.doi.org/10.1109/ICPADS.2004.1316116.
[46] Ookla, The Definitive Source for Global Internet Metrics (2018). URL https://www.ookla.com/speedtest-intelligence.
[47] M.A. Rodriguez, R. Buyya, Container-based Cluster Orchestration Systems: A Taxonomy and Future Directions (2018). arXiv preprint arXiv:1807.06193.
[48] Red Hat Inc., Etcd: A distributed, reliable key-value store for the most critical data of a distributed system. (2018). URL https://coreos.com/etcd.

**Yasser Aldwyan** is a Ph.D. candidate at the University of Melbourne, Australia and a lecturer at Islamic University of Madinah in Saudi Arabia. He worked as a researcher at the Saudi National Center for Electronics, Communication and Photonics in King Abdulaziz City for Science and Technology. He has a B.Sc. in Computer Science from Taibah University, Saudi Arabia and a M.Sc. in Computer Science from the University of Melbourne. His research interests are in Cloud computing, containerization, placement and elasticity.

**Richard O. Sinnott** is the Director of eResearch at the University of Melbourne and Professor of Applied Computing Systems. In these roles he is responsible for all aspects of eResearch (research-oriented IT development) at the University. He has been lead software engineer/architect on an extensive portfolio of national and international projects worth over $400m, with specific focus on those research domains requiring finer-grained access control (security). Prior to coming to Melbourne, Richard was the Technical Director of the UK National e-Science Centre; Director of e-Science at the University of Glasgow; Deputy Director (Technical) for the Bioinformatics Research Centre also at the University of Glasgow, and for a while the Technical Director of the National Centre for e-Social Science. He has a Ph.D. in Computing Science, an M.Sc. in Software Engineering and a B.Sc. in Theoretical Physics (Hons). He has over 350 peer-reviewed publications across a range of computing and application-specific domains. He teaches High Performance Computing and Cloud Computing at the University of Melbourne.