SN Computer Science (2023) 4:270
https://doi.org/10.1007/s42979-023-01702-7

ORIGINAL RESEARCH

A Novel Weight-Assignment Load Balancing Algorithm for Cloud Applications

Adekunbi A. Adewojo[1] · **Julian M. Bass**[1]

Received: 1 September 2022 / Accepted: 23 January 2023

Abstract

Web applications commonly suffer from flash crowds and resource failure, resulting in performance degradation. Flash crowds are large, sudden, yet legitimate influxes of user requests that constitute a critical problem because of their potential economic damage. For cloud providers, resource estimation is challenging, while distributing workload and sustaining performance. To alleviate flash crowds and resource failure problems, we propose a novel weight assignment load balancing algorithm that combines five carefully selected server metrics to efficiently distribute the workload of three-tier web applications among virtual machines. We experimentally characterised, using a private cloud running OpenStack, the load distribution ability of our proposed novel algorithm, as well as a baseline algorithm and round-robin algorithm. We compared the performance of the three algorithms by simulating resource failures and flash crowds, while carefully measuring response times. Our experimental results show that our approach improves average response times by 12.5% when compared to the baseline algorithm and 22.3% when compared to the round-robin algorithm in the flash crowds' situation. In addition, average response time was improved by 20.7% when compared to the baseline algorithm and 21.4% when compared to the round-robin algorithm in resource failure situations. These experiments show that our novel algorithm is more resilient to fluctuating loads and resource failures than baseline algorithms.

Keywords Cloud computing · Load balancing · Weight assignment · Three-tier applications

Introduction

The use of cloud applications continues to gain rapid adoption in businesses because of the benefits of using the cloud [1–3]. These benefits include but are not limited to high availability, increased agility, flexibility, lower total cost of ownership, ability to reach users across the globe, and provision of configurable options that suit users' needs [4–6]. Cloud applications are commonly offered as Software as a Service (SaaS) to end users. Scalability, on-demand, and elasticity are key features of the cloud [3, 7, 8] that promote the rapid adoption of cloud and cloud-based applications.

The popularity of cloud has promoted the hosting of e-commerce and social networks such as Facebook, Twitter, and LinkedIn [3]. Posts on these networks can easily become viral, leading to large number of requests trying to access the application, thereby resulting in performance degradation [3]. This scenario is termed flash crowds. Flash crowd is a legitimate, rapid, and fluctuating user request surge, that occurs because of the increase in users trying to access an application. Therefore, the cloud's ability to scale, such that available resources can cater to this need, is highly essential. Apart from the flash crowd, it is possible for these applications to experience resource failures, leading to performance degradation.

Cloud providers typically use common load balancing and auto-scaling strategies to combat flash crowds and resource failure scenarios. However, research confirms that this approach does not suffice [3, 8–10]. This is because these applications still suffer some levels of performance degradation due to the inability of the load balancer to

This article is part of the topical collection "Advances on Cloud Computing and Services Science" guest edited by Donald F. Ferguson, Claus Pahl and Maarten van Steen.

✉ Adekunbi A. Adewojo
a.a.adewojo@edu.salford.ac.uk

Julian M. Bass
j.bass@salford.ac.uk

[1] School of Science, Engineering, and Environment, University of Salford, 43, The Crescent, M5 4WT Salford, UK

Published online: 17 March 2023

effectively distribute the workload, or because the auto-scaling strategy was too slow to scale out resources, or never responded at the required time.

To combat these problems in cloud applications, especially in three-tier applications, a targeted and improved load-balancing algorithm that works in collaboration with an auto scaler is essential. Unfortunately, there has been limited research on load-balancing algorithms for web-based three-tier business applications deployed on the cloud [7, 11]. There is even more limited research on experimentally evaluating load balancing/distribution techniques using real cloud infrastructures. Most studies conducted on the evaluation and development of load balancing techniques have focused on the use of simulation tools to evaluate these cloud models [8, 10, 12–15]. This is in contrast to the few evaluations done on real cloud infrastructure [16–18].

In this research, we introduce a novel weight assignment load balancing algorithm. This algorithm improves the performance of cloud-based three-tier web applications by alleviating the negative effect of flash crowds and resource failures. The algorithm's implementation architecture mitigates common limitations of general load balancing algorithms such as single point of failure, excessive re-routing, and slow sensing of uncertainties.

Compared to other research outputs, the proposed novel algorithm incurs less communication overhead than that of Qu et al. [7]. This is because the novel load balancing architecture does not use agents in its communication strategy. In addition, compared to Grozev et al. [8], it does not require an auto-scaling algorithm because it collaborates with existing and standard auto-scaling technique.

The newly introduced algorithm combines five carefully selected key server (Virtual Machine [VM]) metrics (thread count, network buffer, Central Processing Unit (CPU), Rapid Access Memory (RAM), and bandwidth utilisation) to properly distribute the workload of a web-based three-tier business application. This research implements our novel algorithm by combining the use of a software load balancer called HAProxy with our novel weight-assignment load distribution technique. The proposed solution follows the monitor-analyze-plan-execute loop architecture commonly adopted by cloud-based systems [7, 19]. This consistently distributes workload across available servers to maintain the agreed SLA response time without performance degradation despite random user request surges. We validated our algorithm by comparing it to a baseline load balancing algorithm by [8] and the round-robin algorithm. The proposed load balancing solution was evaluated on a private OpenStack cloud using a case study E-commerce application deployed on the private cloud.

This research will help cloud software developers and organisations who want to migrate their applications to the cloud or want to build a data-driven cloud-native application to become aware of improved load balancing algorithms and techniques.

The major contributions of this research are:

- A novel hybrid-dynamic load distribution algorithm that improves response time and scalability of web-based three-tier business applications; and
- An extended evaluation of the proposed algorithm evaluated in a private cloud test-bed

The rest of this paper is organised as follows: "Related Work" summarises related work and how it compares with this research. "Motivation" and "Use Case Scenarios" provides motivation for the research, present our research questions, and describes the use cases of the proposed solution. "Application Architecture and Requirements" describes the architecture of our chosen software application, assumptions, and requirements. "The Proposed Approach" presents our proposed approach and design of the load balancing service. "Proposed Load Balancing Algorithm" presents the novel load balancing algorithm. "Experimental Environment" and "Performance Analysis and Results" describes the experimental environment, evaluation, and present results of our algorithm and benchmark algorithms. "Conclusions and Future Work" concludes the research.

Related Work

Load balancing in the cloud is a method to optimally distribute workload so that the resources of a VM in a cloud computing environment are efficiently utilised. Since cloud computing services have become a vital part of companies, it is even more crucial to improve existing load balancing techniques and enhance cloud application performance because the number of resources are restricted.

Static and dynamic load balancing techniques represent the two broad categories of load distribution techniques [1, 14]. Static techniques are commonly used to distribute predictable loads. The use of static techniques does not require foreknowledge of the current state of the system. Dynamic techniques are used to distribute unpredictable loads. This technique considers the current state of the system before distributing loads.

Researchers have proposed various load balancing algorithms based on either the static or dynamic load balancing technique. Shafiq et al. in [12] proposed a dynamic load balancing algorithm for allocating resources on the Infrastructure as a Service (IaaS) cloud model. The load balancing algorithm was aimed at optimizing resources and improving load balancing of VMs. Their proposed algorithm focused mainly on addressing the priority of VMs, Quality

of Service (QOS) task parameters, and resource allocation. They used an efficient task scheduling technique to improve load balancing of VMs across IaaS by considering the priority of workload, SLA, and key metrics that determine performance. Results show that their algorithm resulted in an average of 78% resource utilisation compared to the existing dynamic load balancing algorithm which had a much lower resource utilisation rate.

Chen et al. in [9] proposed a load-balancing architecture and a dynamic load-balancing algorithm for cloud services. Their approach used a dynamic annexed balance method to solve the problem of uneven workload distribution on servers. Their approach considered both server processing power and computer load to create a load-balancing algorithm that can handle excessive computational requirements. Their result showed that their approach improved the mean response time of load-balancing digital applications deployed on the cloud.

Wang et al. in [16] experimentally compared weighted round-robin and probabilistic routing policies in load balancing multi-class workloads of applications with SLA differentiated across users. Based on their discovery of the relationship between the two policies, they presented and verified algorithms that support multi-class workloads for applications with SLA across different users.

In the bid to improve existing work on dynamic load-balancing for cloud applications, authors [8, 9, 18, 20, 21] identified specific server metrics that commonly affect the performance of both cloud applications and the servers they are running on. These identified metrics included the CPU, RAM, bandwidth utilisation, and network buffer, among others. Tychalas et al. in [10] proposed a dynamic probabilistic load balancing algorithm that uses the weighted round-robin algorithm. Their research combined computational power and the current utilisation of key server metrics to assign probabilities to each available resource. Their simulation result using Bag-of-Tasks jobs as workload showed that their algorithm performed better than the popular weighted round-robin method in terms of mean response time by 8.5% and the utilisation of remote fast resources by 25%.

Similar to the research by Tychalas et al. authors in [20, 21] proposed dynamic load balancing algorithms that combined key server metrics in determining the weights of servers. They highlighted that these metrics play important roles in determining the efficiency of a load balancer. Furthermore, because these algorithms are dynamic, a foreknowledge of the current utilisation of resources and available capacity is a good determiner of how much load a server can handle.

In addition to research that focusses on developing dynamic load balancers, researchers have also focused on incorporating techniques to improve the limitations of the load balancing architecture such as single point of failure [1, 8, 22], scalability, limitation in sensing uncertainties [17], excessive overhead and re-routing [17].

Zhang et al. in [17] introduced Hermes, a data centre load balancer that is resilient to uncertainties such as traffic dynamics, topology asymmetry, and failures. Hermes leverages comprehensive sensing to detect path conditions and reacts using timely yet cautious re-routing techniques. Hermes is a hardware-based load balancer that was implemented using switches. Evaluation using real testbed experiments and simulations show that Hermes can handle uncertainties under asymmetries with 10% and 20% improved performance compared to existing implementations.

Cruz et al. in [22] identified a major challenge in determining how to optimize the mapping of tasks to cluster nodes and cores through increased locality and load balancing. To solve this problem, they proposed an EagerMap algorithm to determine task mappings, which is based on greedy heuristics to match application communication patterns to hardware hierarchies. This technique also considers task load when mapping tasks. They argue that their solution influences communication performance and load balancing in parallel architectures because EagerMap helps to evenly distribute load among clusters and grids. They also claim that their algorithm design alleviates the single point of failure problem in load balancing.

Grozev et al. in [8] introduced an approach for deploying three-tier applications across multiple clouds so that it can satisfy key non-functional requirements. Their research proposed a dynamic and adaptive resource provisioning and load distribution algorithm to improve the load balancing of workload in a multi-cloud setting using heuristics. Their algorithm uses heuristics to optimize overall cost and response delays without violating essential legislative and regulatory requirements. Their simulation results show that their approach improved popular load-balancing algorithms for multi-cloud in terms of availability, regulatory compliance, and quality of experience (QoE) with acceptable sacrifice in cost and latency.

On one hand, dynamic load-balancing techniques are being used frequently to create load-balancing algorithms for cloud applications, and the result from these studies are promising. On the other hand, the use of cloud simulation to experimentally evaluate these cloud models are increasingly used by researchers [8, 17, 23–26]. Furthermore, evaluation of the above researches including these [14, 15, 27] were all done using simulation tools and environment. Also, there is little research [18] on evaluating cloud computing models by completely using real cloud infrastructures. In general, a simplified real infrastructure experiment is performed to complement the simulated experiments as recorded in [8, 16, 17]. Cloud simulation is a common and suitable alternative to using real cloud infrastructure. Cloud simulators are

software that can reproduce the behaviour of cloud systems with a high degree of precision. Cloud simulation employ models to represent and experiment with cloud characteristics and behaviours.

Fakhfakh et al. [25] argues that experimentation in a real cloud environment is a difficult problem with both high financial cost and required time. Again, experiments are not repeatable because several variables are not under the control of the tester. In summation, the use of a cloud simulation framework is preferred because it offers cheaper and faster means for testing new cloud policies and algorithms. Calheiros et al. in [28] claim that cloud simulation provides an avenue to simplify the process of quantifying the performance of scheduling and allocation policy on cloud infrastructure. They argue that because the cloud supports a variety of internet-based applications that require different configuration and deployment requirements, there is a need for a modelling framework that enables seamless modelling, simulation, and experimentation of cloud properties. Many other researchers [14, 15, 23, 24] also posit that the advantages of using simulation include flexibility to switch between different models of cloud providers, repeatability, and self-contained platforms for experiments.

On the contrary, cloud simulation might not be cheaper in terms of the realism of produced results. In addition, the capability of cloud simulator tools is not exhaustive because each one is usually designed to address a specific process of the cloud, for the cloud is usually a combination of several complex components [23]. To successfully use a cloud simulation tool, comprehensive documentation of the tool is required. Likewise, users must be conversant with the programming language required to use the tool. When users are not familiar with the programming language required, learning a new language requires some effort resulting in a loss of time. The availability of cloud simulation tools is limited, not all cloud simulation tools are open source, and this defeats the claim that using real infrastructure requires financial cost [23, 25]. Cloud tools require regular updates to include new and emerging features in the cloud, this becomes a problem if the tool is not regularly updated [24]. In summary, simulation experiments rely heavily on parameters to be accurate, and so the challenge remains - how to choose an accurate parameter. As a result, if the parameters are not right, an incorrect simulation result is inevitable.

In this paper, we present a dynamic load-balancing algorithm that combines carefully selected key server metrics specific to three-tier web applications, to determine the weight of a server. The proposed algorithm and architecture improve performance degradation due to the negative effect of flash crowds and resource failures, limitations such as single point of failure, reduces excessive re-routing using HAproxy, and quick sensing of uncertainties using selected key server metrics. It complements and works cooperatively with auto-scaling mechanisms in cloud data centres to achieve its objectives. While previous work focused on a few server metrics, this research focused on server metrics that directly impact three-tier cloud-deployed applications and the algorithm can be extended to function in a multi-cloud environment. Moreover, due to the stated disadvantages of using cloud simulation tools, this research uses real cloud infrastructure to experiment with the proposed algorithm. This research has access to a private cloud and researchers are conversant with using the cloud infrastructure, so this prevents the challenge of time and cost. We argue that running our experiments on real cloud infrastructure presents real behaviours of resources used and therefore produced more realistic results. This research validates a baseline algorithm and compared the baseline algorithm with round-robin and our novel algorithm. Also, it extends previous research by [4] to accommodate extensive experiments on flash crowds and resource failure situations in cloud-deployed applications.

Motivation

A good load-balancing algorithm should reduce response time, increase throughput, and improve the utilisation of resources while it enhances system performance at a lower cost. Furthermore, a suitable load balancing technique should consider different metrics to make it relevant for applications whose size and needs may increase and which may need to use more resources [1]. A load balancer becomes ineffective when it does not match the traffic patterns and instantiates additional resources accordingly. Consequently, the result of an ineffective load balancer is an increased load on existing servers and increased application latency.

Research [8–10] confirms that standard load-balancing techniques will not be sufficient for many applications deployed on the cloud. Cloud-deployed web applications popularly called SaaS are interactive web applications whose usage fluctuates, so there is a need to monitor and efficiently balance the workload of these applications to ensure consistent performance. These web-based applications commonly suffer from flash crowds and sometimes resource failures; because of these challenges, we posit that SaaS will benefit from dynamic load balancers.

One approach to resolve the challenges associated with standard load balancing techniques and algorithm is to include in the algorithm the key factors that affect the real-time behaviours of a server and cloud resource. The following: CPU utilisation, memory utilisation, network bandwidth, number of threads running, and network buffers were carefully chosen as the major server metrics responsible for determining a VM's real-time capacity and load, in

this research. The use of the above server metrics in a load balancing algorithm will enable a load balancer to distribute the load based on the current capacity of a server, thereby leading to better utilisation of resources.

Our proposed algorithm enhances the previous load balancing algorithm proposed by Grozev et al. [8]. The proposed algorithm combines dynamic load balancing techniques with key server metrics to calculate the weight of a VM. Each VM is assigned a weight based on its current utilisation and capacity. Therefore, the probability of using a VM changes dynamically during runtime every time its current state is evaluated. This concept is represented in our novel load balancing algorithm, depicted in 1.

Use Case Scenarios

The proposed load balancing algorithm will combat the following issues in a three-tier cloud-deployed application:

Flash Crowds

Flash crowds are rapid, fluctuating, exponential, and legitimate web request surges that occur due to increased user requests in web applications [7, 29, 30]. Flash crowds occur without prior notice and may become difficult to manage. A common solution to flash crowd is the use of autoscaling services in combination with load balancers [8, 31]. Commercial cloud providers commonly use autoscaling services to launch new VMs after an application already experienced increased user requests for a specific period of time, a time usually set by users. The services will continue to monitor user requests and will often terminate some VMs and services after a steady stream of user requests has been established. In this research, our proposed algorithm can complement the role of an autoscaler in cloud deployment. The proposed algorithm will effectively balance application workload across VMs and maintain the consistent performance that is in line with predefined service level agreement (SLA) before new resources are provisioned during flash crowds situations.

Resource Failures

Resource failure in cloud happens when a component of a cloud environment fails to function as it is intended to or does not start up when required. Authors in [32, 33] identified hardware, virtual machines and application failures as the three common resource failures in a cloud environment. Authors in [34] argued that faults lead to partial failures in cloud. These authors classified failures in terms of fault namely; network faults, physical faults, process faults, and service expiry faults. The failure of any cloud resource can happen abruptly, thereby, resulting in degradation of performance of cloud services or even a total loss of service. One of the common methods to mitigate the effect of resource failure is to use timely intervention of autoscalers and load balancers to mitigate the effect of resource failure. Meanwhile, an autoscaler requires time to launch new resources, the time between the launch and full functionality of resources usually results in performance degradation if current user requests are not properly managed [7]. Therefore, our proposed algorithm will aid in maintaining acceptable service performance while the autoscaler provisions new resources. Our proposed algorithm combines dynamic and adaptive methods to effectively distribute workload by re-assigning weights to available servers.

Application Architecture and Requirements

This work primarily focuses on three-tier web applications. This architecture is an enterprise software architecture and a common architectural pattern for web applications that supports loose coupling, scalability, reliability, efficient load balancing, etc. [35–37]. It features three physical deployment tiers of logical layers (code organisation) of a software application. Therefore, a three-tier web application has three or more layers deployed across three physical tiers [35, 38]. These layers are referred to as

- Presentation Layer: This represents the user interface, and is commonly referred to as the presentation or client tier.
- Business/Domain Layer: This layer contains the business logic and sometimes an application facade. The business logic is responsible for the manipulation of application data; application of business rules and policies; and ensuring data consistency and validity. This is commonly called the App Tier and is deployed on the application server.
- Data Layer: This layer abstracts the logic to manage the persistent data. It is commonly called the data tier.

The main reason for deploying software layers across multiple tiers is to ensure a balance between performance, scalability, fault tolerance, and security [36]. The presentation layer often executes at the users' end and is thus not a focus of this research work. The business and data layer of a three-tier web application is executed in the back-end server and can be scaled because they are deployed across different tiers.

The business layer commonly consists of one or more Application Servers (AS) which are hosted in separate VMs in Infrastructure as a Service (IaaS) cloud environments, thus supporting scalability. In addition, performance can be

achieved because one of the goals of the cloud is the speed of network communication. Therefore, network boundaries (tiers) are not an issue, so each tier can have multiple running servers that communicate seamlessly while they accommodate user workload. Load balancing also becomes easier because the application and database functionality is separated, so the load balancer can be focused on a specific tier. In terms of fault tolerance, each tier is isolated, so any tier can be modified without affecting the other tiers. Adequate security policies can also be enforced within each tier without affecting users. Horizontally scaling the business layer by adding application servers VMs will improve the ability of a load balancer to distribute incoming requests. However, this is not always the case if the application's business layer is designed to keep session data. For this research, we focus on applications whose domain layer does not keep session data in memory, these types of applications are called stateless applications.

The data layer also consists of one or more database servers. However, this layer often becomes a performance choke point because of the requirements for transactional access and atomicity [8, 39]. Some techniques such as replication, caching, and sharding are recommended to ease the scaling of the data layer [8, 35]. These techniques are application specific, so will be impossible to include them in a generalised framework that targets any three-tier web applications. In short, the right balance for applying the above technique is domain-inherent. Therefore, this research work does not cover application-specific data deployment. Our approach assumes data has been deployed appropriately and will focus on distributing workload effectively across the app tier or application server layer.

The Proposed Approach

As discussed in "Application Architecture and Requirements" about the focus of this research on stateless application, a key principle behind stateless applications and our proposed approach is session continuity. Session continuity is an important factor in distributing requests across any application server. "Session continuity ensures that end-user sessions, established over any access networks, will not lose connection or any internal state even when different servers process the user requests [4]". Session continuity is the principle behind the improved performance of stateless applications. Stateless applications do not keep track of requests made previously or store them on servers, so user requests can be distributed across any application server without any difficulty. Also, stateless applications can be easily scaled because they do not require the persistence of end-user Internet Protocol (IP).

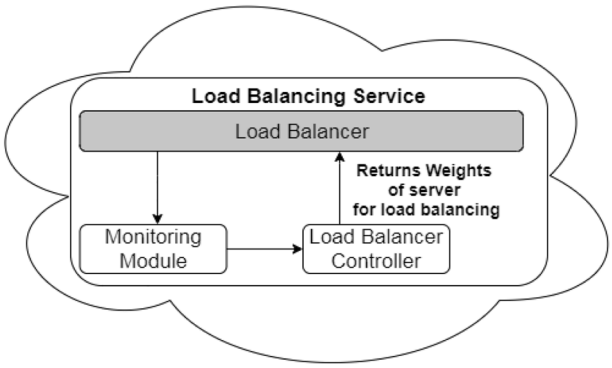


Fig. 1 Proposed load balancing architecture [4]

Load Balancing Architecture

Figure 1 depicts the design architecture of our proposed load-balancing service. The load-balancing algorithm (load balancer controller) forms part of our load-balancing service, including a load-balancer software (HAProxy), and a monitoring module. The load balancer controller returns the weight of all participating servers to the load balancer software at a specified time that will be discussed later. The monitoring module monitors all servers for scaling/de-scaling purposes, the health of servers, and the frequency of incoming user requests.

The communication strategy used in the above design is the message-based communication strategy. This strategy improves decoupling, flexibility, and maintainability [40]. This included serialisation of data when required, particularly when transferring data to calculate the weights of the server.

The load balancer and other components were deployed on the same VM in the same data centre with the test web application. The co-location of these components helps to achieve fast detection of flash crowds and perform quick adaptations.

Overall Architecture

Figure 2 depicts the overall architecture of the proposed load-balancing system and software application. The conventional three-tier web application was extended to include an extra layer of components. This extra layer represents the proposed load balancing service and an auto-scaler. This extra layer of component enables the migration of existing three-tier application to the cloud with little or no modification to the code base.

The workflow of this architecture is as follows: users will interact with the presentation layer and requests will be sent to an entry point that consists of VMs that host our load balancing service. The load balancing service will send user requests to the application server. These application servers are hosted on separate VMs, and they host the domain

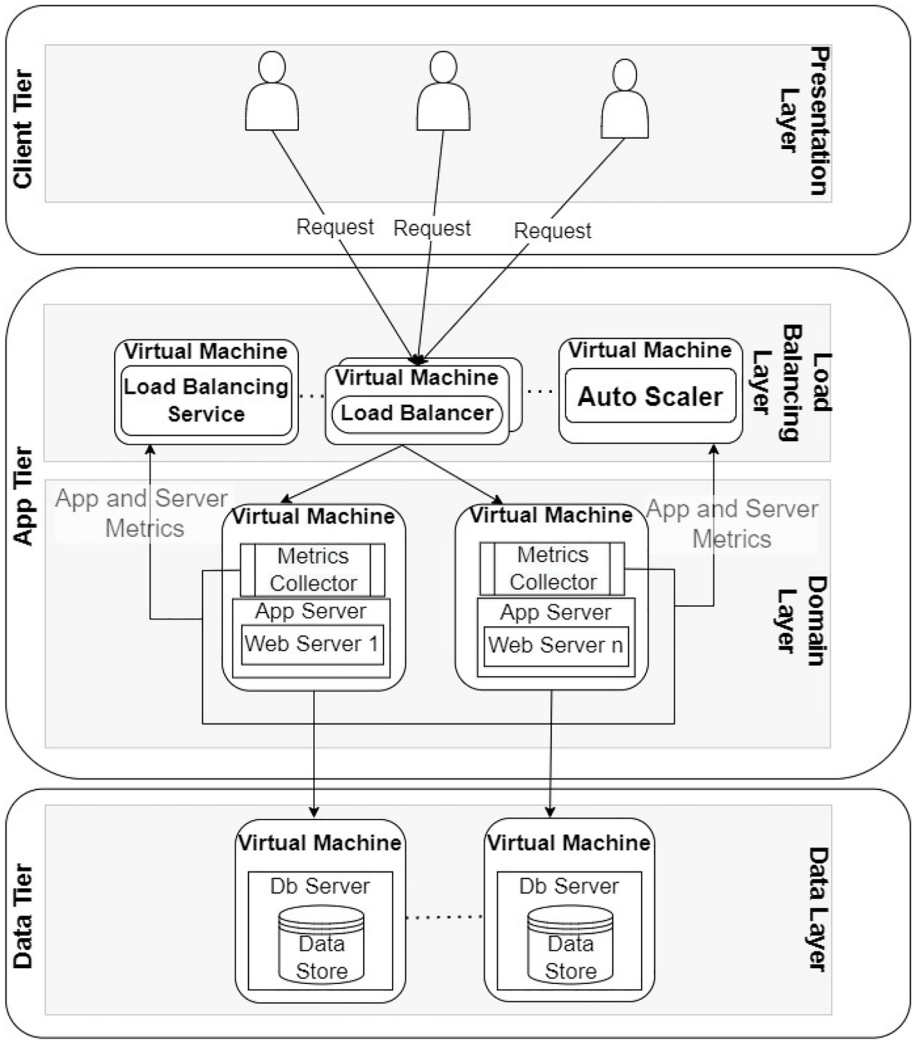


Fig. 2 Proposed overall layered architecture in a single data center

layer. The domain layer then interacts with the data layer to manipulate required data before sending back responses to the user. The data and domain layers consist of multiple servers installed across VMs. The process is repeated every time requests come in, and the load balancing service distributes the workload accordingly.

Design and Deployment Architecture

An efficient and bespoke load-balancing algorithm should include different metrics that are relevant to the application it supports [1]. Therefore, the design of the proposed load balancing algorithm will address issues and limitations that were discussed in "Related Work" by incorporating techniques and carefully selected metrics that are important to load-balance an interactive three-tier web application. These issues spotlight areas of improvement in a load-balancing algorithm and architecture. Below we discuss these issues and techniques to mitigate them as used in the proposed design.

- Scalability: Scalability in load balancing is the ability of a load balancer to continue to distribute workload across any finite number of servers. The proposed load-balancing architecture will scale across any number of servers because it uses a proven load-balancing system - HAProxy 2.4.2-1 [41]. HAProxy is an open source, very fast and reliable reverse proxy high-performing TCP/HTTP load balancer [41]. It is recommended for most websites and is particularly suited for high-traffic websites. It is commonly used by major websites and powers a significant portion of the world's most visited sites [41]. In addition, our experiments featured the removal and addition of a varied number of heterogeneous VMs to test scalability. The solution proved its ability to distribute the load across a finite number of nodes on the server.
- Fault Tolerance: Cloud fault tolerance is the capability of the infrastructure to support uninterrupted functionality of deployed applications despite failures of components [42, 43]. In the cloud, fault tolerance approaches include

the use of various system models that depend on network topology, proactive approaches such as the use of a fault detection system, reactive approaches such as the use of VM placement model, and other approaches such as the use of machine learning and meta-heuristic approach in algorithms [34]. The ability of a load balancer to continue to deliver services despite failure [44] of a cloud component is highly desirable in guaranteeing the availability and reliability of cloud-deployed applications. The proposed architecture created multiple load-balancing front-ends and standby application servers as failover systems. This technique also includes self-healing, job migration, and replication policies of HAProxy as an additional fault tolerance technique in the proposed architecture. The proposed approach utilised HAProxy's high availability keep-alive technology to regularly monitor servers and services for a fast job migration. More so, the use of floating IPs that can be moved between load balancers to allow continued availability of service at optimum performance level.

- Reduced Overhead and Latency: Performance overhead is the extra time taken in performing an assigned functionality by a cloud component. A high-performance overhead will lead to an increased communication cost and noticeable performance degradation [45] in running applications. In a load balancer, performance overhead is influenced by several factors, including the load balancing algorithm [14, 44]. Our proposed load balancing algorithm runs efficiently with negligible performance overhead of less than 2% when compared to other load balancing algorithms. In addition, the proposed load-balancing architecture exemplifies a decentralised approach to collecting and updating server metrics. This centralised approach is a recommendation by authors in [14] to reduce performance overhead and latency. Also, in the proposed architecture, each participating VM is preinstalled with glances [46] agents, a cross-platform monitoring tool for monitoring system resources and utilisation. Glances agent uses the RESTful Application Programming Interface (API) of glances to capture server metrics that are sent to a time series database called InfluxDB. The use of the RESTful API improves performance when data is collated.
- Server Metrics: Server metrics play an important role in determining the efficiency of a load-balancing algorithm. Chen et al. [9] encouraged the use of varied but application-specific server metrics in load-balancing algorithms. The proposed algorithm combines carefully selected key server metrics in its formulation. These server metrics are specific to three-tier web applications and are also recommended by various authors [8, 10, 16, 21]. We argue that besides popular and common metrics of CPU, Memory, and network bandwidth utilisation, the thread count of a processor is a vital server metric that is often overlooked. Thread count determines how efficiently data can be transmitted in and out of a system, and provides a summary of concurrent requests within a server. The combination of these metrics indicates the true state of the server's current workload; thereby providing adequate information for the load balancer to perform efficiently.

Proposed Load Balancing Algorithm

This section describes our novel weight assignment technique and load balancing algorithm.

Proposed Weighting Technique

The proposed VM weighting technique for our novel load balancing algorithm is an improvement on the research work by [9] and [18]. The weighting technique combined four server metrics to calculate the weight of a VM and an additional server metric as a key determining factor in the load-balancing algorithm. These server metrics are represented as follows: M_k represents Memory utilisation, B_k represents Bandwidth utilisation, C_k represents CPU utilisation, T_k represents Thread Count and NB_k represents Transmission Control Protocol (TCP) network buffer/queues.

The weight of a VM represented as $W(X_k)$ requires first calculating the real-time load, $Lr(X_k)$, as shown in eq. (1). The real-time load uses the following server metrics: M_k, B_k, C_k, T_k. These metrics are retrieved in percentages from the monitoring tool and therefore converted to integer values by dividing each of them by 100.

$$\begin{aligned} Lr(X_k) = (e_1 * (C_k/100)) + (e_2 * (M_k/100)) \\ +(e_3 * (B_k/100)) + (e_4 * (T_k/100)) \end{aligned} \tag{1}$$

To compensate for the presence of bias and to reflect the influence each metric has on a VM, weight factors were used for all the metrics, as shown in eq. (1). e_1, e_2, e_3, e_4 represents the weights of CPU, RAM, bandwidth, and thread count respectively. The sum of these weights is 1. The weight factor values were carefully chosen experimental proven values. Weight factor values were fitted in experiments, and chosen values were the best-fit values representing the unique influences of each server metric of an application server for a three-tier web application. The chosen values for each weight factor are $0.5 for CPU$, $0.3 for RAM$, $0.15 for bandwidth$, $and 0.05 for threadcount$ for thread count. CPU utilisation has the biggest weight because our chosen class of application becomes processor intensive when a lot of data is being passed. Memory utilisation has the next biggest weight because its influence on a VM can quickly lead to a non-responsive server when the utilisation is high.

Network buffer was not included in eq. (1), so no weight was assigned to it. The reason network buffer was not included in the equation is because this metric stores packets temporarily and can measure the ratio of network utilisation to availability of a network, consequently its influence on a server is constant. In this experiment, the network buffer was used in the load balancing algorithm to regularly monitor network availability, mostly when the real-time load of a server is being calculated because a larger buffer size reduces the potential for flow control to occur.

We define a threshold for load comparison as the average value of all participating application server VM load, as shown in eq. (2). n represents the number of all participating application server VMs.

$$Lr_{th} = \frac{\sum Lr(X_k)}{n} \tag{2}$$

To further improve and analyse the weight calculator, some modalities are set out as follows: if $Lr(X_k) \leq Lr_{th}$, this means the application server's load is relatively small, so the weight assigned to the server can be increased. If it is the opposite, the assigned weight should be decreased because the server's load was high. To define and quantify these changes, a modification parameter δ is defined as shown in eq. (3).

$$\delta = \frac{Lr(X_k)}{\sum Lr(X_k)} \tag{3}$$

Following these calculations, the real-time load can now be calculated and a load comparison of servers can also be done using the aforementioned equations. The weight calculator will return the lowest real-time load value for the least utilised VM, but HAProxy, our chosen load balancer, functions in a reverse manner. This means the load balancer expects or appropriates the biggest weight value for the least utilised VM. To make the weight calculator function in line with HAProxy's policy on weight, the inverse of the real-time load, $Lr(X_k)$, in eq. (4) is computed. This makes the lowest real-time load value to be assigned the biggest value.

Lastly, HAProxy's weight policy is bounded for real integer values, this means that the supplied weight must be a whole number; otherwise, it will not be consistent with the original intention. The proposed novel algorithm rounds up any decimal value that is greater than eight to make the value a whole number. The weight calculator is represented in eq. (4).

$$W(X_k) = \begin{cases} (\frac{1}{Lr(X_k)} + \delta), Lr(X_k) \leq Lr_{th} \\ (\frac{1}{Lr(X_k)} - \delta), Lr(X_k) \geq Lr_{th} \end{cases} \tag{4}$$

Proposed Load Balancing Algorithm

We present the proposed novel load balancing algorithm in Algorithm 1. This algorithm is an abstraction of the overall process flow of the weighting and load-balancing logic. It is a hybrid-dynamic load balancing algorithm that computes every two seconds the utilisation of all participating servers. After computing the utilisation of each server VM, it then assigns a weight to each server during runtime using our proposed weighting technique, this process also changes the load balancer's configuration, in other words, the load balancer is notified of the changes. The load balancer then automatically adjusts the amount of load distributed to each server based on the weight of each server.

The input parameters for the algorithm are as follows:

- Thc—CPU threshold;
- Thr—RAM threshold;
- $Thbw$—Bandwidth threshold;
- Th_tr—Thread count threshold;
- VM_{as}—list of currently deployed application server VMs;

The algorithm first receives and sets the overall threshold for the above input parameters. Experimentally and research-approved threshold values of 80%, 80%, 80%, 85% for CPU, RAM, Bandwidth, and Thread count respectively are set. Experimentally, we performed profiling tests on a medium application server VM using a synthetic workload generated from real web application requests to corroborate the threshold values. A pre-defined SLA that states that 90% of requests should be handled within one second, as recommended by [7] was used to determine the average utilisation percentage.

In line 7 of the algorithm, the function **TCPBufferOverloaded()** represents an extracted logic for checking TCP network buffer NB_k. This function utilises the netstat command to check for the presence of a TCP socket whose ratio of the buffer sizes, Recv-Q and Send-Q values, is greater than 0.9. The value of 0.9 is a research-verified value by [8]. A ratio value greater than 0.9 means the network buffer is overloaded and requests will not arrive promptly at the VM.

The algorithm regularly retrieves the utilisation values of our chosen VM metrics. After the retrieval, in line 2, the algorithm loops through available server VMs, and compare each utilisation metric of the current server against the set threshold. The algorithm then computes the weight using our weighting technique in eq.(4), and carries out the TCP network buffer check. A weight is then assigned to each server VM whose network buffer is not overloaded as depicted in line 9. If the server VM is fully utilised 100%, a weight of 0 is assigned to the VM as shown in line 11. A fully utilised VM with a weight of zero means, the VM will not accept any more incoming requests until the utilisation rate is lower or equal to the set threshold.

In line 14, requests will be distributed by the load balancer according to the weight of each server VM in a round-robin manner. This process distributes server load proportionally based on a VM's real-time capacity.

Algorithm 1: Novel Load Balancing Algorithm.

Input: s_i, Thc, Thr, $Thbw$, Th_tr, VM_{as}

```
RetrieveAllocateToInputThresholdValues ();
for each VM, vm_i ∈ VM_as do
    Utl_cpu ← CPU utilisation of vm_i;
    Utl_ram ← RAM utilisation of vm_i;
    Utl_bw ← Bandwidth utilisation of vm_i;
    UtlThreadCount ← Threadcount of vm_i;
    if (Utl_cpu ¡ Thc & Utl_ram ¡ Thr & Utl_bw ¡ Thbw &
     Utl_ThreadCount ¡ Th_tr & !TCPBufferOverloaded ()) then
        W(X_k) ← CalculateWeightbyutilisation (Utl_cpu, Utl_ram,
         Utl_bw, UtlThreadCount, VM_as, vm_i);
        assignweighttoVM (vm_i, W(X_k));
    else
        assignweighttoVM (vm_i, 0);
    end
end
HAProxyAssignRequest (s_i, VM)
```

A visual representation of the proposed algorithm and the weighting technique is depicted in Fig. 3.

Experimental Environment

We depict our experimental setup in Fig. 4. The proposed load balancing algorithm was evaluated on a private cloud infrastructure running OpenStack, the train version. The experimental testbed consists of seventeen heterogeneous VMs with characteristics as shown in Table 1.

Apache Jmeter, an open source load testing tool for measuring and analysing the performance of services, was deployed and run on a separate stand-alone machine. HAProxy was deployed alongside our load-balancing solution on two VM instances. Eight medium instances of VMs were launched as application servers, two extra medium

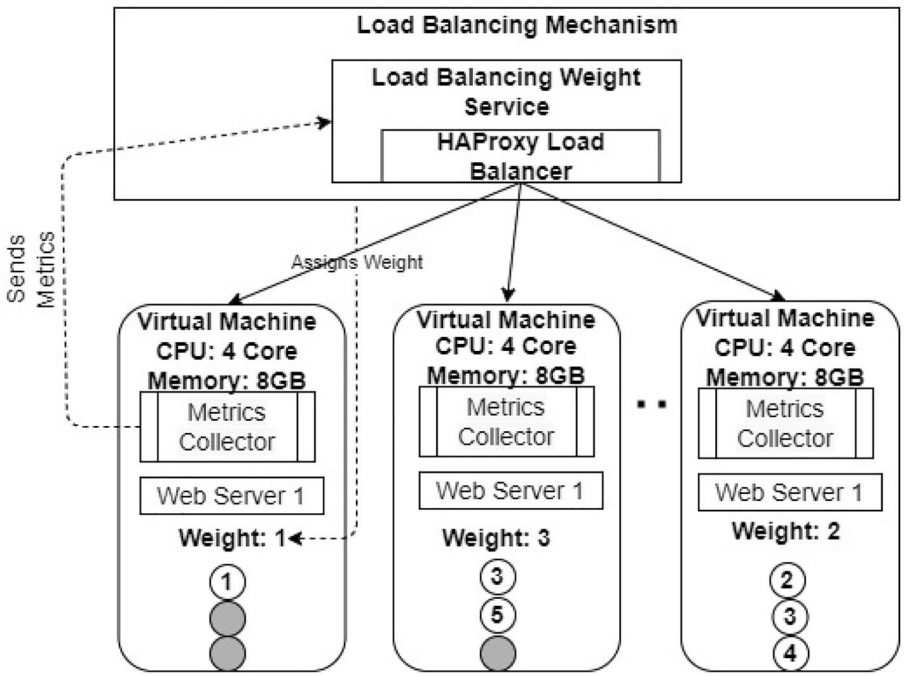


Fig. 3 Visual representation of the load-balancing algorithm

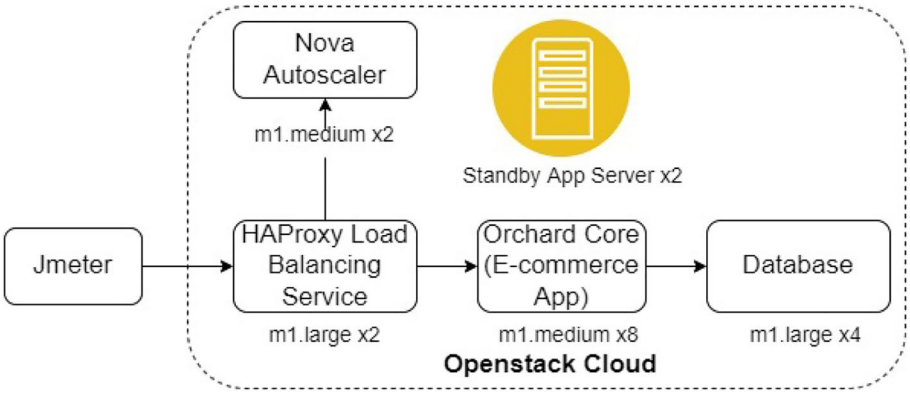


Fig. 4 Experimental test-bed

Table 1 VM capacity used in experiments

Characteristics	m1.medium	m1.large
VCPUs	4	8
Memory Size	4GB	8GB
Storage Size	40GB	80GB

instances of application server VM are in standby mode, and four large VM instances were launched as database servers. Each application server had installed on them, technologies for hosting our case study applications, and a part of the load-balancing module that was responsible for polling utilisation values.

Case Study Application

The case study application used in this research is a stateless, three-tier, open source multi-tenant E-Commerce application. It is a pre-built application that used Orchard Core framework coupled with elastic search for its search functionalities. The application is a data-driven application that is backed by MySQL database. The case study application was deployed on each participating application server. The database was also deployed and replicated on all participating database servers.

Workload

Apache Jmeter was used to simulate user requests that were sent to the case study application. Apache Jmeter was hosted on a standalone machine because of the need to simulate realistic requests in terms of location and time. Our evaluations seek to characterise prominent load balancing performance parameters as identified by researchers. It will also compare the difference in performance between our novel load balancing algorithm and the baseline load balancing algorithm by Grozev et al. [8]. We generated workload from Apache Jmeter by simulating loads of various scenarios using a predefined model that will be discussed below.

Firstly, we profiled one of our application instances to determine the number of requests that can be successfully handled within one second. Secondly, we profiled our application instance again to determine the number of requests that can be handled with an SLA constraint of 90% within one second as recommended by [7].

An average combination of the two profiles showed that our application server instance can handle between 90 and 92 requests per second. Based on this profiling, we created our workload to emulate a Poisson distribution because of the nature of user request arrival as discussed by [47]. We used Jmeter to send requests based on the Poisson workload model to our deployed application. We repeated each experiment three times and took the average of the repeated tests in all of our evaluations.

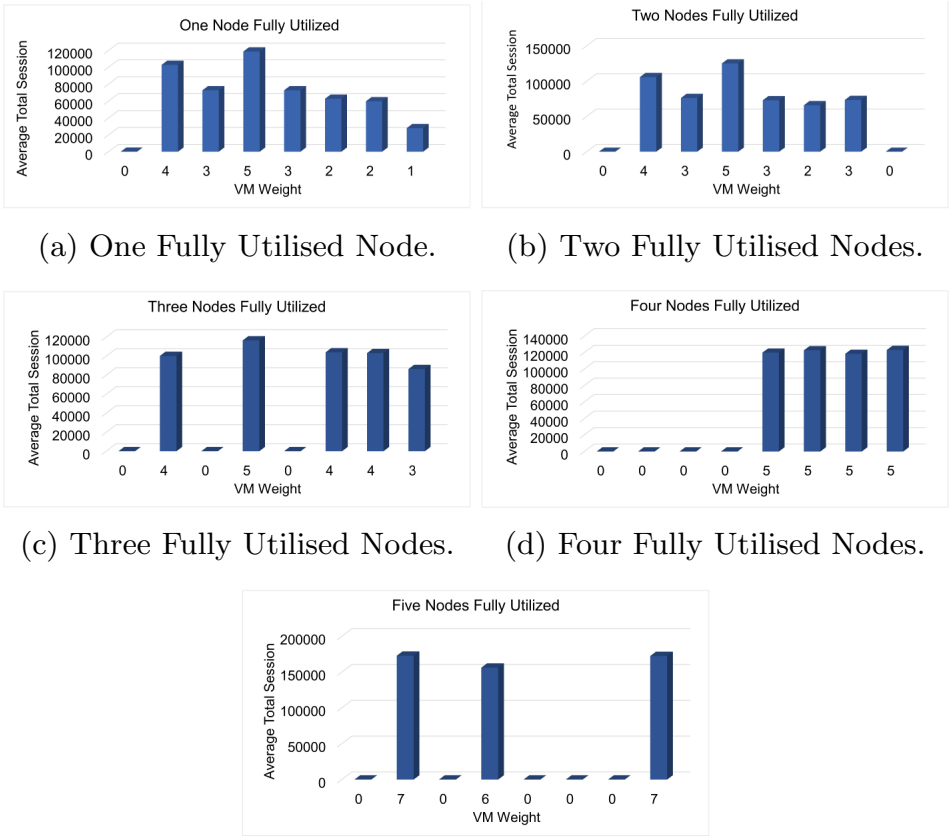
Baseline Application and Evaluation

To evaluate our proposed load balancing solution; we test the performance of our algorithm, and we compare our algorithm with a baseline algorithm by [8] and a standard load balancing algorithm - round-robin algorithm. Our evaluation sought to characterise our algorithm using response times, a prominent load-balancing performance parameter [14, 44].

Performance Analysis and Results

This section discusses the analysis and performance comparison of our novel algorithm and the benchmarks. In this paper, we extend the experiments performed in our previous research [4] to include more resource failures and significantly increased flash crowds.

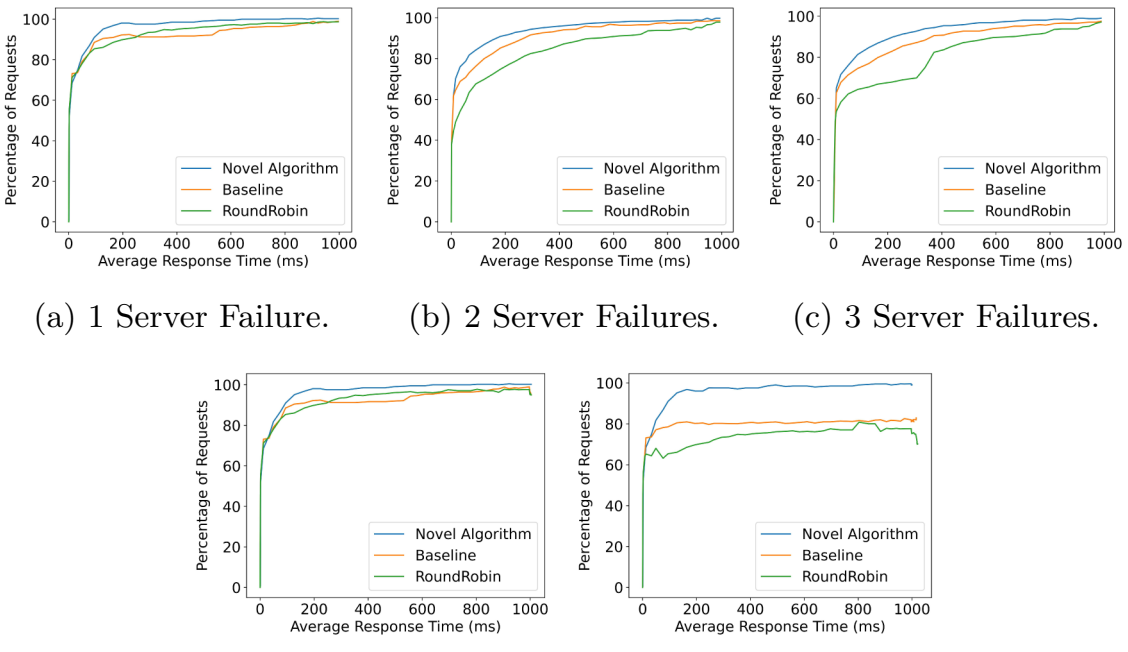
Fig. 5 Fully utilised VM nodes



(a) One Fully Utilised Node. (b) Two Fully Utilised Nodes.

(c) Three Fully Utilised Nodes. (d) Four Fully Utilised Nodes.

(e) Five Fully Utilised Nodes.



(a) 1 Server Failure. (b) 2 Server Failures. (c) 3 Server Failures.

(d) 4 Server Failures. (e) 5 Server Failures.

Fig. 6 Distribution values of server failures

Performance Under Resource Failures

To test the algorithms' performance, we first validated our novel algorithm by testing its ability to distribute workload across VM. The baseline algorithm by [8] was also tested to ascertain it produces similar results recorded by the researchers. We simulated varying numbers (1–5) of over-utilised VM as depicted in Fig. 5. The aim of this experiment was to test if the proposed novel algorithm will assign the correct weight to the VM, and if it will redirect workload to other VMs while correctly assigning appropriate weight to these working VMs. The proposed novel algorithm uniformly distributed workload among available VMs within the stipulated time demanded by the SLA.

The aim of simulating resource failure is to test the level of fault tolerance and scalability of the load balancing system in this research. To replicate resource failure situations, especially hardware failure, some VMs were stopped and removed from the available pool of participating VMs. VMs were removed at 300ms time point at every experiment. VMs were gradually added back to the pool after five seconds to emulate recovery from failure. In total, five different server failure scenarios were created.

The performance of the three (novel load balancing algorithm, baseline algorithm, and round-robin algorithm) algorithms were compared in all the experiments. The round-robin algorithm performed worst, as depicted in Fig. 6. The number of requests the round-robin algorithm handled when it encountered between one and two server failures were 10% less than the stipulated SLA requirement. The performance of the round-robin algorithm depleted when server failure was more than four. The algorithm handled less than 80% of requests within one second. This shows that randomly distributing workload, even uniformly, cannot suffice for our chosen class of application. This type of load balancing might be sufficient for applications that are not critical or resource intensive.

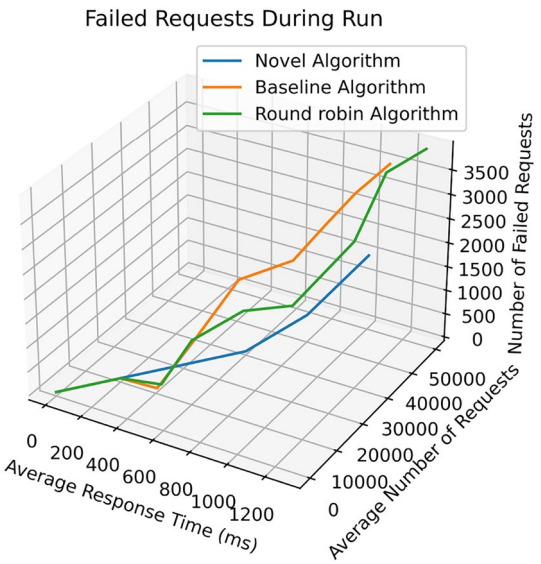


Fig. 7 Failed request chart — error rates

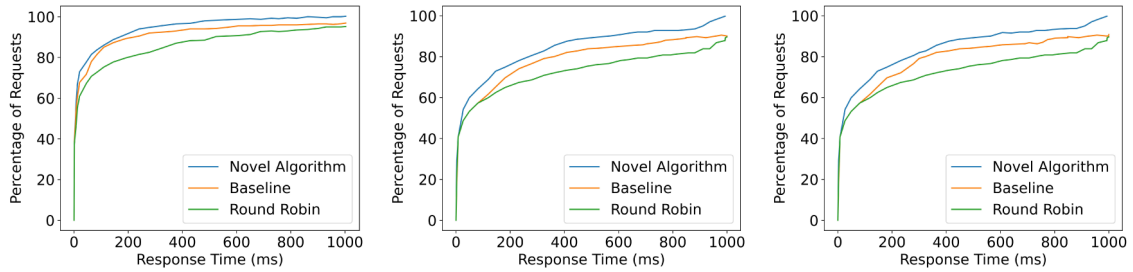
The baseline algorithm by [8] performed better than the round-robin algorithm, as shown in Fig. 6. When server failures were not more than three, the baseline algorithm could handle approximately 90% of requests, albeit at a much higher response time compared to the novel algorithm. When server failures increased to five, the algorithm suffered from performance degradation. The response times were high, with values between 1.2 and 1.6 s. The percentage of requests handled declined to less than 80% of requests. Also, we noted that at the peak of server failures, the algorithm became unresponsive resulting into errors as shown in Fig. 7.

The proposed novel load balancing algorithm distributed requests within the stipulated SLA as shown in Fig. 6. It did not violate the SLA at any point of the server failure. Response times were constant and improved compared to the two other algorithms. During all server failures, the novel algorithm attended to over 90% of requests between 0 ms and 998ms. The result showed that response times of our novel algorithm falls within an acceptable range and were less than the response times of the two benchmark algorithm. This proves that the proposed novel algorithm functions best for our chosen class of applications.

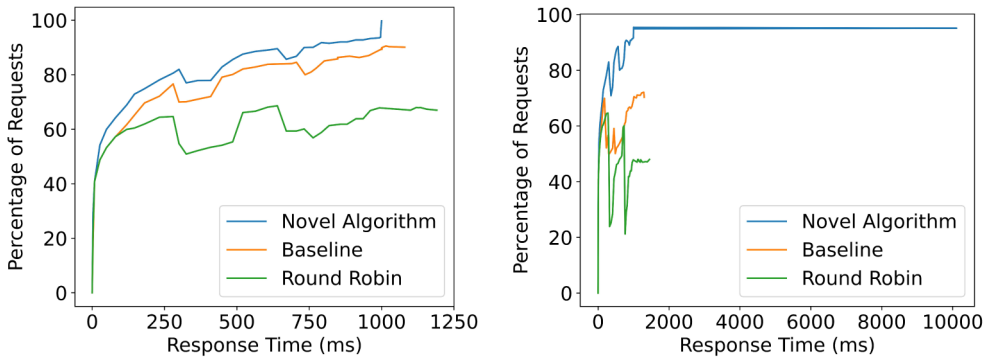
To explore boundaries where our approach starts to break down, the ratio of the number of server failures to available servers was experimented with. The result showed that to achieve the SLA recommendation of 90% of requests to be handled in one second, using our workload model with the number of requests between 1000 and 50,000 requests, a minimum of three VMs should be available and running. Figure 7 showed that the novel algorithm had less than 1500 failed requests when there were five server failures, unlike round-robin and the baseline algorithm that had over 3000 failed requests with response times more than 1 s.

Performance Under Flash Crowds

To test the algorithms' performances under flash crowds scenario, we simulated flash crowds by updating our request workload to exponentially increase for a period of 300ms every five seconds within one minute. The percentage of flash crowds ranged between 110% and 320% of the normal workload. The three algorithms consistently distributed workload across available VMs. The three algorithm all handled 90% or more requests within one second, as depicted in Fig. 8a. When the flash crowd reached 150%, their response



(a) 125% Increased Request Flash Crowds. (b) 150% Increased Request Flash Crowds. (c) 188% Increased Request Flash Crowds.



(d) 240% Increased Request Flash Crowds. (e) 320% Increased Request Flash Crowds.

Fig. 8 Cumulative distribution of flash crowd

times varied but they all maintained the SLA constraint of attending to 90% of requests within one second.

Flash crowds of 188% as shown in Fig. 8c revealed that the baseline and round-robin algorithm's performance are starting to get depleted but with failed requests of 8% of the overall requests.

Flash crowds of 240% revealed that the round-robin algorithm could only handle 50% of requests at approximately 300ms when the first flash crowd happen, and 56% of requests when flash crowds occurred a second time. However, Fig. 8d showed that the round-robin algorithm took 1190ms before it could meet 67% of requests, thereby violating the SLA constraint. The baseline algorithm as shown in Fig. 8d did better than the round-robin algorithm. The baseline algorithm handled 70% of requests at around 300ms when the first flash crowd hit, and then handled 81% of requests when the second flash crowd happened. The baseline algorithm handled a total of 89% of requests at 999.987ms. The baseline algorithm could not handle 90% of requests within one second. The novel algorithm as shown in Fig. 8d handled 77% of requests at 300ms when the first flash crowd hit, and 85% of requests when the second flash crowd occurred. The novel algorithm still

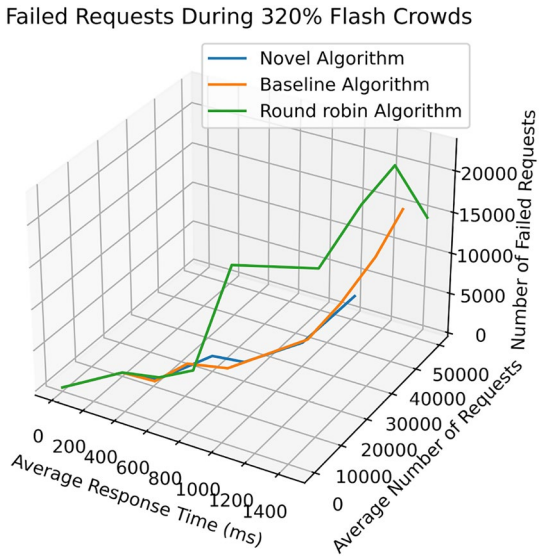


Fig. 9 Failed request chart – error rates during peak flash crowds

maintained the handling of a minimum of 90% of requests within one second, even though the response time was 999.999ms longer than the previous occurrence of flash

crowds; with a response time between 960.905ms and 992.798ms.

A flash crowd of 320% above the normal is shown in Fig. 8e. Figure 8e shows a different calibration of the response time because it took longer for the algorithms to attend to requests. Round-robin algorithm handled 23% and 21% of requests when flash crowds of 320% occurred. The overall average response time of the round-robin algorithm was 450ms longer than the other two algorithms. The baseline algorithm handled 50% and 56% of requests during flash crowds of 320%. The baseline algorithm could only handle 70% of requests within one second before the algorithm started to timeout. The novel algorithm handled 70% and 80% of requests when flash crowds of 320% occurred. It took a longer time for the novel algorithm to attend to 90% of requests, this means a flash crowd of 320% above normal workload will break the three algorithms if no extra VM is provisioned. The ability of the algorithm to consistently distribute workload during the varying scenarios confirms and validates our choice of carefully selected metrics.

Figure 9 showed the amount of failed request at the peak of flash crowds - 320% flash crowds. The graph showed that the novel algorithm recorded less than 2000 failed requests at the peak of the flash crowds compared to the round-robin which had almost 19000 failed requests and the baseline algorithm which had 14560 failed requests.

To test the effect of the auto scaler when there is a flash crowd and how the proposed load balancing service will work with an autoscaler, the auto scaler was programmed to start when all the available VMs utilisation exceeded 89% usage. In addition, two extra VMs were kept as standby VMs, these two VMs will be used by the autoscaler to launch extra VMs when needed.

Research Limitation

The scope and limitations of this research are discussed using the model of experimental research limitations. This approach is characterised by evaluating the research's process through the lens of internal and external validity and threats to validity. Identifying and defining the variables of interest, including how to measure them in a reliable and valid manner, is the first step in evaluating the validity of experimental research's data [48, 49].

This research identified potential variables and their measures through the extensive literature on similar research work. This research then carefully selected widely cited variables and their units of measurement from research works such as [7–9, 50, 51]. Identified variables are but not limited to CPU, RAM, Network buffer, bandwidth, response time, throughput, latency, and so on.

A research design has strong internal validity if the observed relationship is not related to extraneous variables such as differences in subjects, location, or other related factors [48, 49, 52]. To create a strong internal validity for this research, the appropriateness of the variables were considered before deciding to use them.

The load balancing algorithm apply to three-tier web applications; therefore, the behaviour of the algorithm is uncertain when used in other cloud services. Also, the number and size of user requests used in the experimental evaluations were within the limits of a private cloud. Therefore, the results of this study should not be generalised to extremely large public clouds.

The use of a real cloud infrastructure nullifies the effects of location and setting on experimental scenarios. The reason for this is that real infrastructure corresponds to a natural context for using the cloud.

Conclusions and Future Work

Web applications commonly suffer from flash crowds and resource failures, which degrade their performance. We have created a novel weight assignment load balancing algorithm and architecture for cloud-based three-tier web applications. This new algorithm utilises five carefully selected server metrics to determine and assign weight to server VMs by analysing their utilisation values to determine a VM's real-time load. The server metrics are as follows: thread count, CPU, RAM, network buffers, and network bandwidth utilisation. Our novel load balancing algorithm addresses the challenges faced by three-tier web applications deployed on cloud.

To tackle single point of failure, reliability, and scalability, a highly available deployment architecture and a standardised load balancer software were also used. Additionally, the proposed load balancing algorithm and service were deployed in the same data centre. As a result, rapid adaptation to changes in the environment, reduced communication overhead, and faster network capabilities were achieved.

Using a case study e-commerce web application, the load balancing solution and benchmark algorithms were evaluated on an OpenStack private cloud. Firstly, the novel algorithm was validated on the test bed by evaluating its ability to distribute workload evenly. Secondly, the novel algorithm's performance was compared with a baseline load balancing algorithm by [8] and round-robin algorithm. During the workload simulation, we experimentally measured and compared the response times of the case study application.

When compared to the baseline algorithm and round-robin algorithm, the novel algorithm proposed improved the overall average response times by 20.7% and 21.4% respectively, in resource failure situations. In flash crowd

situations, the novel algorithm improved response times by 12.5% and 22.3% respectively, over the baseline algorithm and round-robin algorithm. These results show that the novel algorithm can adapt to flash crowds and resource failures without performance degradation.

The experiments will be extended in the future to evaluate other types of resource failures and the algorithms' performance. Additionally, we will investigate serverless deployment across both single cloud and multi-cloud environment.

Data availability This is a future work recommendation, so there is no available data yet.

Declarations

Conflict of interest Adekunbi Adewojo declares that she has no conflict of interest. Julian Bass declares that he has no conflict of interest.

Ethical approval This article does not contain any studies with human or animal participants performed by the authors.

References

1. Kumar P, Kumar R. Issues and challenges of load balancing techniques in cloud computing: a survey. ACM Comput Surv (CSUR). 2019;51(6):1–35.
2. Akintoye SB, Bagula A. Improving quality-of-service in cloud/fog computing through efficient resource allocation. Sensors. 2019;19(6):1267. https://doi.org/10.3390/s19061267.
3. de Paula Junior U, Drummond LM, de Oliveira D, Frota Y, Barbosa VC. Handling flash-crowd events to improve the performance of web applications. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, pp. 769–774 (2015)
4. Adewojo A, Bass J. A novel weight-assignment load balancing algorithm for cloud applications. In: CLOSER 2022: 12th International Conference on Cloud Computing and Services Science (2022). Scitepress
5. Cloud Adoption to Accelerate IT Modernization | McKinsey. https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/cloud-adoption-to-accelerate-it-modernization. Accessed 2022-07-20
6. Elmroth E. "15 Years of Cloud Control" (2022). https://closer.scitevents.org/PreviousInvitedSpeakers.aspx. Accessed 2022-04-27
7. Qu C, Calheiros RN, Buyya R. Mitigating impact of short-term overload on multi-cloud web applications through geographical load balancing. Concurr Comput Pract Exp. 2017;29(12):4126.
8. Grozev N, Buyya R. Multi-cloud provisioning and load distribution for three-tier applications. ACM Trans Auton Adapt Syst. 2014;9(3):13–11321. https://doi.org/10.1145/2662112.
9. Chen S-L, Chen Y-Y, Kuo S-H. Clb: A novel load balancing architecture and algorithm for cloud services. Comput Electr Eng. 2017;58:154–60.
10. Tychalas D, Karatza H. An advanced weighted round robin scheduling algorithm. In: 24th Pan-Hellenic Conference on Informatics, pp. 188–191 (2020)
11. Buyya R, Srirama SN, Casale G, Calheiros R, Simmhan Y, Varghese B, Gelenbe E, Javadi B, Vaquero LM, Netto MA, et al. A manifesto for future generation cloud computing: research directions for the next decade. ACM Comput Surv (CSUR). 2018;51(5):1–38.
12. Shafiq DA, Jhanjhi NZ, Abdullah A, Alzain MA. A load balancing algorithm for the data centres to optimize cloud computing applications. IEEE Access 9, 41731–41744 (2021)
13. Kang S, Veeravalli B, Mi Aung K.M. Scheduling multiple divisible loads in a multi-cloud system. In: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, pp. 371–378 (2014). https://doi.org/10.1109/UCC.2014.47
14. Zomaya AY, Teh Y-H. Observations on using genetic algorithms for dynamic load-balancing. IEEE Trans Parallel Distrib Syst. 2001;12(9):899–911.
15. Hellemans T, Bodas T, Van Houdt B. Performance analysis of workload dependent load balancing policies. Proc ACM Measurement Anal Comput Syst. 2019;3(2):1–35.
16. Wang W, Casale G. Evaluating weighted round robin load balancing for cloud web services. In: 2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp. 393–400 (2014). IEEE
17. Zhang H, Zhang J, Bai W, Chen K, Chowdhury M. Resilient datacenter load balancing in the wild. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, pp. 253–266 (2017)
18. Chen Z, Zhang H, Yan J, Zhang Y. Implementation and research of load balancing service on cloud computing platform in ipv6 network environment. In: Proceedings of the 2nd International Conference on Telecommunications and Communication Engineering, pp. 220–224 (2018)
19. Zeng J, Plale B. Multi-tenant fair share in nosql data stores. In: 2014 IEEE International Conference on Cluster Computing (CLUSTER), pp. 176–184 (2014). IEEE
20. Devi DC, Uthariaraj VR. Load balancing in cloud computing environment using improved weighted round robin algorithm for nonpreemptive dependent tasks. Sci World J. 2016;2016:3896065. https://doi.org/10.1155/2016/3896065
21. Sahu Y, Pateriya RK, Gupta RK. Cloud server optimization with load balancing and green computing techniques using dynamic compare and balance algorithm. In: 2013 5th International Conference and Computational Intelligence and Communication Networks, pp. 527–531 (2013). https://doi.org/10.1109/CICN.2013.114
22. Cruz EH, Diener M, Pilla LL, Navaux PO. Eagermap: a task mapping algorithm to improve communication and load balancing in clusters of multicore systems. ACM Trans Parallel Comput (TOPC). 2019;5(4):1–24.
23. Bambrik I. A survey on cloud computing simulation and modeling. SN Comput Sci. 2020;1(5):1–34.
24. Byrne J, Svorobej S, Giannoutakis K.M, Tzovaras D, Byrne PJ, Östberg P-O, Gourinovitch A, Lynn T. A review of cloud computing simulation platforms and related environments. In: International Conference on Cloud Computing and Services Science. 2017;2: 679–691. https://doi.org/10.5220/0006373006790691.

25. Fakhfakh F, Kacem HH, Kacem AH. Simulation tools for cloud computing: a survey and comparative study. In: 2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS). 2017; 221–226. https://doi.org/10.1109/ICIS.2017.7959997
26. Makaratzis AT, Giannoutakis KM, Tzovaras D. Energy modeling in cloud simulation frameworks. Future Gener Comput Syst. 2018;79:715–25. https://doi.org/10.1016/j.future.2017.06.016.
27. Elgedawy I. Sultan: A composite data consistency approach for saas multi-cloud deployment. In: 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC). 2015;122–131. https://doi.org/10.1109/UCC.2015.28
28. Calheiros RN, Ranjan R, Beloglazov A, De Rose CA, Buyya R. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software Pract Exp. 2011;41(1):23–50.
29. Ari I, Hong B, Miller E.L, Brandt SA, Long DD. Managing flash crowds on the internet. In: 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. IEEE 2003; 246–249.
30. Le Q, Zhanikeev M, Tanaka Y. Methods of distinguishing flash crowds from spoofed dos attacks. In: 2007 Next Generation Internet Networks. IEEE. 2007;167–173
31. Amazon: AWS Serverless Multi-Tier Architectures with Amazon API Gateway and AWS Lambda AWS Whitepaper. https://docs.aws.amazon.com/whitepapers/latest/serverless-multi-tier-architectures-api-gateway-lambda/three-tier-architecture-overview.html Accessed 2021-01-01
32. Priyadarsini RJ, Arockiam L. Failure management in cloud: An overview. International Journal of Advanced Research in Computer and Communication Engineering. 2013;2(10):2278–1021.
33. Prathiba S, Sowvarnica S. Survey of failures and fault tolerance in cloud. In: 2017 2nd International Conference on Computing and Communications Technologies (ICCCT). 2017. pp. 169–172. IEEE
34. Kumari P, Kaur P. A survey of fault tolerance in cloud computing. J King Saud Univ Comput Inf Sci. 2021;33(10):1159–76. https://doi.org/10.1016/j.jksuci.2018.09.021.
35. Fowler M. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley Longman Publishing Co. Inc; 2002.
36. Rockford Lhotka - Should All Apps Be N-tier? https://web.archive.org/web/20200802111420/http://www.lhotka.net:80/weblog/ShouldAllAppsBeNtier.aspx Accessed 2022-08-15
37. Archiveddocs: Chapter 19: Physical Tiers and Deployment. https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658120(v=pandp.10) Accessed 2022-08-15
38. Ramirez AO. Three-tier architecture. Linux J. **2000**(75es) (2000)
39. Brewer E. Cap twelve years later: How the "rules'' have changed. Computer. 2012;45(2):23–9.
40. Archiveddocs: Chapter 18: Communication and Messaging. https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658118(v=pandp.10) Accessed 2022-08-16
41. HAProxy: HAProxy Technologies - The World's Fastest and Most Widely Use Load Balancing Solution. https://haproxy.com/ Accessed 2021-01-01
42. Bala A, Chana I. Fault tolerance-challenges, techniques and implementation in cloud computing. Int J Comput Sci Issues (IJCSI). 2012;9(1):288.
43. What Is Fault Tolerance? Definition & FAQs. https://www-stage.avinetworks.com/glossary/fault-tolerance/ Accessed 2022-08-18
44. Shah J.M, Kotecha K, Pandya S, Choksi D, Joshi N. Load balancing in cloud computing: Methodological survey on different types of algorithm. In: 2017 International Conference on Trends in Electronics and Informatics (ICEI). 2017. pp. 100–107. IEEE
45. Xu F, Liu F, Jin H, Vasilakos AV. Managing performance overhead of virtual machines in cloud computing: a survey, state of the art, and future directions. Proc IEEE. 2013;102(1):11–31.
46. Hennion N. Glances an Eye on Your System. A Top/htop Alternative for GNU/Linux, BSD, Mac OS and Windows Operating Systems. https://glances.readthedocs.io/en/latest/ Accessed 2021-01-01
47. Chlebus E, Brazier J. Nonstationary poisson modeling of web browsing session arrivals. Inf Process Lett. 2007;102(5):187–90.
48. Yin RK. Case study research: design and methods. New York: Sage publications; 2014.
49. Experimental design research approaches. Perspectives. 2014. Cham: Springer.
50. Fehling C, Leymann F, Retter R, Schupeck W, Arbitter P. Cloud computing patterns. In: Fundamentals to design build, and manage cloud applications. 2014. Springer, London. https://doi.org/10.1007/978-3-7091-1568-8
51. Grozev N, Buyya R. Performance modelling and simulation of three-tier applications in cloud and multi-cloud environments. Comput J. 2015;58(1):1–22.
52. Bhandari P. Internal Validity in Research | Definition, Threats & Examples (2020). https://www.scribbr.com/methodology/internal-validity/ Accessed 2022-10-24