Novel Cloud Load Distribution Management and Deployment Techniques

Adekunbi Adeyinka Adewojo

UNIVERSITY OF SALFORD

SCHOOL OF SCIENCE, ENGINEERING AND ENVIRONMENT

JULY 21, 2023

This document is toward the degree of Doctor of Philosophy

Declaration

I hereby declare that the work presented in this thesis is original and the outcome of a PhD research carried out by me.

I certify that, to the best of my knowledge, this thesis is consistent with copyright requirements. All views, techniques, and quotations from other studies have been fully acknowledged with references. This thesis has not previously been submitted for the award of any academic degree at the University of Salford or any other university.

Adekunbi Adeyinka Adewojo

Dedication

This thesis is dedicated to the memory of my father, **Mr Jacob Adegboyega Adewojo**.

Acknowledgement

I thank and praise you, God of my ancestors, for you have given me wisdom and strength. Daniel 2:23a NLT

I would like to thank my supervisor Prof. Julian M. Bass for his patience over the years, consistency, and support during this PhD journey.

I would like to express my deepest thanks to my husband, Olayinka Popoola for his patience, and encouragement, and for being a pillar of support during the course of my study. I am incredibly grateful to my daughter, Ilerioluwa Popoola, for being mummy's buddy during this journey, this endeavour would not have been possible without you and daddy. I am extremely grateful to my mum, Revd. Mrs Folashade Adewojo, for being a prayer warrior and a constant cheer giver on this journey, your prayers saw me through. I am grateful to my brothers for the ways you cheered me up, especially my brother, Olaoluwa Adewojo for proofreading, and critically evaluating my writings.

Words cannot express my gratitude to Mr Ola Jimoh-Akindele, you were like a second supervisor to me. Thank you for your suggestions, corrections, and words of encouragement. Many thanks to Prof. Samson Arekete for constantly asking about my progress, giving advice, proofreading, and critically evaluating my work.

I am also thankful to Bukola Adesina, Mrs Adesola Anidu, Dr Ibiye Iyalla, and Chioma Ujomu, you all made this journey possible by helping me academically when I felt deflated.

Lastly, I would like to acknowledge all those who supported me through this process; friends, colleagues, peer reviewers, and examiners who contributed to this study.

Contents

1 Introduction **1**

1.1 Introduction 1

1.2 Problem Context 4

1.3 Research Motivation 6

1.4 Research Aim, Question, and Objectives 7

1.5 Research Contribution 8

1.6 Thesis Organisation 10

2 Literature Review **13**

2.1 Introduction 13

2.2 Cloud Computing 13

2.2.1 Cloud Service Model 14

2.2.2 Cloud Deployment Models 15

2.3 Cloud-Hosted Applications Architectures 16

2.3.1 Monolithic 16

2.3.2 Microservices 17

2.3.3 Event-driven serverless 17

2.3.4 Cloud-based architecture 18

2.4 Design Patterns 19

2.4.1 Background on Design Patterns 20

2.4.2 Object-Oriented Design Patterns 21

2.5 Cloud Design and Architecture Patterns 23

2.5.1 Cloud Architecture Patterns 24

2.5.2 Formal Description of Design Patterns 30

2.6 Cloud Migration Process and Methodology 30

2.6.1 Cloud Migration Process 30

2.6.2 Cloud Migration Methods 32

2.6.3 Cloud Migration Tools 33

2.7 Common Challenges of Cloud-Hosted Web Applications 36

2.7.1 Flash Crowds 36

2.7.2 Resource Failures 36

2.8 Addressing the Challenges of Cloud-Hosted Web Applications . . 37

2.8.1 Workload Distribution and Resource Management in Single Cloud 37

2.8.2 Test Environments for Load Balancing Algorithms 41

2.8.3 Workload Distribution and Resource Management in Multi-Cloud Deployment 43

2.9 Summary 48

3 Research Design 50

3.1 Introduction 50

3.2 Research Process Model 51

3.3 Selection of Case Study Application 51

3.3.1 Enhanced Cloud Design Pattern 52

3.3.2 Novel Weight Assignment and Multi-Cloud Load Balancing Algorithm 53

3.4 Experimental Research Approach 53

3.4.1 Preliminaries 54

3.4.2 Enhanced Cloud Design Pattern 56

3.4.3 Novel Weight Assignment Load Balancing Algorithm . . 58

3.4.4 Multi-Cloud Load Balancing Algorithm 61

3.5 Summary 65

4 Enhanced Cloud Design Pattern Structure - Cloud Application Deployment and Architecture Pattern 67

4.1 Introduction 67

4.2 Enhanced Cloud Design Pattern Structure 68

4.3 Enhanced Multi-tenancy Patterns 74

4.3.1 Formal Description of Shared Component Pattern 75

4.3.2 Formal Description of Tenant-Isolated Component Pattern 76

4.3.3 Dedicated Component Pattern 77

4.4 Implementation of Multi-tenancy Patterns in Case Study Applications 78

4.4.1 Implementation of Shared Component Pattern in Case Studied BPM 78

4.4.2 Implementation of Tenant Isolated Pattern in Case Studied BPM 78

4.4.3 Implementation of Dedicated Component Pattern in Case Studied BPM 79

4.4.4 Implementation of Multi-tenancy Patterns in Containerized Cloud Hosted WordPress Application 79

4.5 Results 80

4.6 Summary 82

5 A Novel Weight-Assignment Load Balancing Algorithm for Cloud Applications 84

5.1 Introduction 84

5.2 Requirements and Assumptions of Proposed Load Balancing Algorithm 85

5.3 The Proposed Approach 86

5.3.1 Load Balancing Architecture 87

5.3.2 Overall Architecture 87

5.3.3 Design and Deployment Architecture 88

5.4 Proposed Load Balancing Service 92

5.4.1 Proposed Weighting Technique 92

5.4.2 Proposed Load Balancing Algorithm 94

5.5 Results and Performance Analysis 97

5.5.1 Performance under Resource Failure 97

5.5.2 Performance under Flash crowd 105

5.6 Summary 110

6 Multi-Cloud Load Distribution Algorithm and Architecture 112

6.1 Introduction 112

6.2 Application Requirements 113

6.2.1 Communication Protocol 114

6.3 Deployment and Overall Architecture 115

6.4 Proposed Multi-Cloud Load Distribution Service 117

6.4.1 Proposed Multi-Cloud Weighting Technique 117

6.4.2 Multi-Cloud Load Balancing Algorithm 117

6.4.3 Algorithm Implementation 119

6.5 Performance Analysis and Results 120

6.5.1 Benchmarks 121

6.5.2 Performance under Flash crowds 121

6.5.3 Performance under Resource Failure 124

6.6 Summary 130

7 Discussion 132

7.1 Introduction 132

7.2 Patterns 132

7.2.1 Enhanced Cloud Design Pattern Structure 133

7.2.2 Formal Description of Multi-tenancy Patterns 134

7.2.3 Novel Implementation and Validation of Multi-tenancy Pattern 135

7.3 Cloud Infrastructure 137

7.3.1 Experimental Environment for Novel Weighting Algorithm 138

7.3.2 Experimental Environment for Multi-Cloud Deployment . 139

7.4 Novel Weighting Algorithm 140

7.4.1 Key Server Metrics for Load Balancing Algorithm 141

7.4.2 Novel Weighting Algorithm for Load Balancing in Cloud . 142

7.4.3 Overall Design Architecture of Proposed Load Balancing Solution 143

7.4.4 Empirical Evaluation of Algorithms 144

7.4.5 Architectural Component Collaboration and Performance . 145

7.4.6 Extensible Feature of Weighting Algorithm 146

7.5 Multi-Cloud Deployments 147

7.5.1 Multi-Cloud Load Distribution Algorithm and Deployment Architecture 148

7.5.2 Design Approach for Three-Tier Multi-Cloud Web Application 149

7.5.3 Enhanced Multi-Cloud Architecture and Communication Protocol 150

7.6 Limitations of Research 151

7.7 Summary 154

8 Conclusions and Future Direction **156**

8.1 Thesis Summary 156

8.2 Key Research Contributions 158

8.3 Future Work 161

8.3.1 Generalisation to Multi-tier Applications and Serverless Deployments 161

8.3.2 Creation of Realistic Workload Model or Dataset for Web Applications 161

8.3.3 Metrics Measurement 162

References **162**

Appendix **178**

A Enhanced Multi-tenancy Patterns **179**

A.1 Shared Component Pattern 179

A.2 Tenant Isolated Component Pattern 183

A.3 Dedicated Component Pattern 189

B Approved Ethical Application Document **193**

B.1 Ethical Clearance 193

List of Tables

3.1 Capacity of VM used in Experiments. 58

3.2 VM Capacity of Proposed Multi-Cloud Environment 62

3.3 Average Latencies between data centres in milliseconds 63

3.4 Workload Parameter 64

4.1 Initial Structure of OOP and Cloud Design Patterns. 69

4.2 Initial and Enhanced Cloud Design Pattern Structure 70

4.3 OOP and Cloud Design Pattern Structure 72

4.4 Description of Enhanced Cloud Design Pattern Structure 73

List of Figures

1.1 Thesis Structure and Logical Dependencies. 11

2.1 Multi-tenancy Patterns 28

3.1 Research Process Model. 51

3.2 Experimental Architecture for Evaluation. 57

3.3 Single OpenStack Cloud Experimental Test-Bed. 59

3.4 Multi-Cloud Experimental Test-bed. 62

3.5 Workloads with Flash crowds range from 110% to 220%. 65

4.1 Z Representation of Shared Component Pattern (Adewojo, Bass, & Allison, 2015). 76

4.2 Z Representation of Tenant-Isolated Component Pattern (Adewojo, Bass, & Allison, 2015). 77

4.3 Z Representation of Dedicated Component Pattern (Adewojo, Bass, & Allison, 2015). 77

4.4 Average Response Time of the Multi-tenancy Patterns when creating Blog Posts. 81

4.5 Number of Request Errors when creating Blog Posts. 82

5.1 Proposed Load Balancing Architecture (Adewojo & Bass, 2022b) 88

5.2 Proposed Overall Layered Architecture in a Single Data Center . . 89

5.3 Visual Representation of Novel Load Balancing Algorithm. 97

5.5 Fully Utilised VM Nodes. 99

5.7 Distribution Values of Server Failures within Private Cloud. 103

5.8 Failed Request Chart — Error Rates 104

5.9 125% Increased Request Flash Crowds. 105

5.10 150% Increased Request Flash Crowds. 106

5.11 188% Increased Request Flash Crowds. 106

5.12 240% Increased Request Flash Crowds. 107

5.13 320% Increased Request Flash Crowds. 108

5.14 Failed Request Chart — Error Rates During Peak Flash Crowds . 109

6.1 Communication Protocol 114

6.2 Load Balancing Overall Architecture. 116

6.3 Cumulative Distribution Values of Flash Crowds – 140% increased flash crowd. 122

6.4 Cumulative Distribution Values of Flash Crowds – 190% increased flash crowd. 122

6.5 Cumulative Distribution Values of Flash Crowds – 240% increased flash crowd. 123

6.6 Cumulative Distribution Values of One Server Failure across Cloud Data Centres. 125

6.7 Cumulative Distribution Values of Two Server Failures in One Data Centre Each. 126

6.8 Cumulative Distribution Values of Two Server Failures in Two Data Centres Each. 128

6.9 Three Server Failures across Cloud Data Centres. 129

A.1 UML Diagram for Shared Component Pattern 181

A.2 UML Diagram for Tenant Isolated Component Pattern 186

A.3 UML Diagram for Dedicated Component Pattern 191

Abstract

Cloud computing provides scalable, flexible, and cost-effective computing resources. Cloud adoption has, however, introduced resource utilisation and service unavailability issues during the process of designing, deploying, and hosting cloud-native applications. These challenges can occur due to large, sudden, yet legitimate influxes of user requests, also known as flash crowds, and resource failures. Interactive web applications that experience sudden surges in user activity are particularly susceptible to these challenges. In this research, novel cloud algorithms and techniques were developed to address these challenges. An experimental approach was used to evaluate these novel cloud algorithms and techniques.

The main contribution of this research is the creation of an experimentally characterised novel weight-assignment load balancing algorithm that combines five carefully selected server metrics to determine the server's capacity to efficiently distribute the workload of three-tier web applications among application servers. A novel decentralised multi-cloud architecture and algorithm were also developed to distribute the workload of three-tier web applications using geographical and improved load distribution techniques.

In this research, a private OpenStack cloud was configured followed by a bespoke cloud experimental testbed, and finally, a heterogeneous multi-cloud experimental testbed to evaluate the novel algorithms. The experimental evaluation validated the novel algorithms against the baseline load distribution algorithms and techniques. The algorithms were implemented as a software service and were tested using the workload of an open-source cloud-hosted E-Commerce application.

The experiments carefully measured response times, scalability, number of errors, and throughput during flash crowds and resource failure scenarios. Results showed that the novel load balancing algorithms and architecture are more resilient to fluctuating loads and resource failures than baseline algorithms. For example, the novel single cloud load balancing algorithm improved the average response times by 12.5% when compared to the baseline algorithm and by 22.3% when compared

to the round-robin algorithm in the flash crowds situation.

The main conclusion of this research is that cloud-native application developers can mitigate the adverse effects of flash crowds and resource failures by using dynamic load balancing algorithms that carefully combine selected server metrics that affect a specific class of applications. Furthermore, the success of implementing and coordinating heterogeneous cloud infrastructure demonstrates the usability of the experimental testbed to evaluate cloud algorithms.

Chapter 1

Introduction

1.1 Introduction

Cloud computing is the latest answer to scalable and reliable computing. Its services provide pay-as-you-go access to computing infrastructure and resources over the Internet thus making utility computing a reality (Armbrust et al., 2010; Buyya et al., 2018; Rimal & Maier, 2017; Varghese & Buyya, 2018). Cloud computing offers a flexible and economical way to rent pre-configured computing resources. This model of outsourcing computing resources offers organisations the opportunity to own software applications that can serve global users.

Consequently, the use of cloud applications continues to gain rapid adoption in businesses because of the benefits of using the cloud (Akintoye & Bagula, 2019; de Paula Junior et al., 2015; Kumar & Kumar, 2019; Odun-Ayo et al., 2017). These benefits include flexibility, low cost of ownership, the ability to reach users across the globe, and the availability of configurable options that suit customers' needs (Adewojo & Bass, 2022b; Buyya et al., 2018; "Cloud adoption to accelerate IT modernization — McKinsey", 2022; Elmroth, 2022). Additionally, scalability, on-demand access, and elasticity are key features that promote the rapid adoption

of cloud computing and cloud-deployed applications (de Paula Junior et al., 2015; Grozev & Buyya, 2014b; Qu et al., 2017). The effect of this rapid adoption is that cloud data centres are fast becoming the preferred deployment environment for software applications (Buyya et al., 2018; Grozev & Buyya, 2014b).

Cloud-native applications are generally offered as Software-as-a-Service (SaaS) to end users (Foster et al., 2008; Grozev & Buyya, 2014b). These applications run on the web and tend to be interactive and serve a general purpose (Fowler, 2002; Grozev & Buyya, 2014b). The three-tier architecture is a prevalent deployment architectural model for SaaS business applications (Fowler, 2002; Grozev & Buyya, 2014b). These applications are often deployed to the cloud, but the process of building, deploying, and hosting these applications contains challenges. The process requires a reusable, methodical, and efficient deployment approach. Design patterns ensure that technical risk and software debt are reduced when properly followed. Design patterns also determine whether or not the required quality attributes of the application will be exhibited (Bass et al., 2012; Fehling, Leymann, Retter, Schupeck, et al., 2014). Designing cloud-native applications requires a mastery of design principles, patterns, and architectural insights. However, there is still a long way to go before these aspects are fully understood (Vale et al., 2022). The use of design patterns is one such aspect. Design patterns begin to emerge as professionals encounter and overcome challenges associated with this architectural approach and share solutions to specific problems (Vale et al., 2022). Design patterns give software architects an array of techniques to overcome software design issues, thereby reducing the technical risk to their projects by not having to employ new and untested design approaches. Similar to design patterns, cloud design patterns are applicable to frequently occurring cloud design problems.

A recurring challenge associated with the implementation of the right cloud design pattern (while building cloud applications), is the difficulty in understanding how and where the cloud design pattern applies (Ochei et al., 2015). Despite research that has collated cloud patterns for describing the cloud, its properties, and how to deploy and use various cloud offerings (Fehling, Leymann, Retter, Schupeck, et al., 2014; Fowler, 2002; Homer et al., 2014; Wilder, 2012), the structure for describing

these patterns lacks uniformity and comprehensiveness. Research also indicates that there is not enough evidence to show that design patterns are inherently valid (Baltes & Ralph, 2022; Vale et al., 2022). For these reasons, it is not uncommon to find situations where a good fit design pattern for a cloud-native application is picked but teams are unable to implement them successfully (Jamshidi et al., 2018). Patterns should therefore be subjected to further evaluations just as any other type of theory (Riehle et al., 2021).

Aside from the challenges associated with building cloud applications, there is a problem with balancing the workload of cloud-native applications after they have been deployed to the cloud. This load-balancing challenge is commonly the result of flash crowds and resource failures (Qu et al., 2016; Qu et al., 2017). Flash crowds is a legitimate, rapid, and fluctuating user request surge that occurs because of the increase in users trying to access an application. Some adverse effects of not properly handling flash crowds and resource failures are unavailability of services and low network latency to end users (Buyya et al., 2018; Grozev & Buyya, 2014b). 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 because these applications still suffer some performance degradation as a result of the inability of the load balancer to effectively distribute the workload or because the auto-scaling strategy was too slow to scale out resources or it did not respond at the required time (S.-L. Chen et al., 2017; de Paula Junior et al., 2015; Grozev & Buyya, 2014b; Tychalas & Karatza, 2020). Also, research shows that there is a need for deployment strategies that can overcome service unavailability, vendor lock-in, low latency to end users, and improved regulatory compliance strategy (Buyya et al., 2018; Grozev & Buyya, 2014b).

This study will investigate design patterns for software applications with a view to improving the cataloguing of cloud design patterns. This study will also create an enhanced cloud pattern structure and apply the enhanced pattern structure to multi-tenancy patterns which is a key cloud architecture pattern for successfully deploying applications. Most importantly, this study will develop novel load

distribution techniques and combine these techniques alongside multiple cloud deployment strategies to mitigate the adverse effect of flash crowds and resource failures such as service unavailability, vendor lock-in issues, low latency to end users, regulatory compliance issues and other related issues of cloud-hosted web applications.

1.2 Problem Context

There are complexities involved in the process of designing and developing software applications, which is only further exacerbated in the design process of cloud-native applications. This is because cloud-native applications are designed and deployed to be inherently scalable, portable, available, reliable, and predictable. As a result of this inherent complexity, software architects find it difficult to implement the right design patterns when developing software applications. The incorrect implementation of design patterns can result in software failure, which leads to a host of negative cascade effects such as economic loss and a resultant impact on the quality of life of the software users. The failure of cloud-native applications is even more far-reaching because these applications usually serve users globally.

Conversely, research shows that the correct use of design patterns reduces the chances of software failures (Bass et al., 2012; Fowler, 2002), improves quality attributes of the application (Bass et al., 2012), increases fault detection rate (Ali et al., 2020), and improves the response time of web application and micro-services (Saboor et al., 2021). Therefore, it is expedient that solutions be provided to improve the chances of correctly using design patterns. This study investigates cloud design patterns and the difficulty associated with implementing the right cloud design pattern to cloud-native applications. This scrutiny is critical for building reliable and robust software applications. In addition, the structure of cloud design patterns will be improved to ease the process of designing cloud-native applications with the right cloud design pattern. The improved cloud pattern

structure will also be applied to a key architectural cloud design pattern to validate its use against common practices.

So, this study will compare and contrast the structures of cloud design patterns with widely cited pattern catalogue structures. The knowledge gained from the comparison will be used to create an improved cloud design pattern structure. Furthermore, this study will implement the multi-tenancy patterns in case-study applications to validate the use of multi-tenancy patterns. The implementation will showcase the use of a general cloud deployment strategy and the use of container technology to validate the ability to use the patterns in different environments. The implementation will also be experimentally evaluated, with its performance characterised through the use of response times and the number of errors.

In addition to these challenges, there is the problem of load-balancing the workload of cloud-native web applications. This problem arises because of the vulnerability of these applications to flash crowds, caused by viral interactions and posts (de Paula Junior et al., 2015; Grozev & Buyya, 2014b; Qu et al., 2017). Challenges to workload distribution in cloud-native web applications is also exacerbated by resource failures. These challenges (flash crowds and resource failures) pose potential economic damage and previous studies concluded that they constitute critical problems (Qu et al., 2016; Qu et al., 2017). Also, performance degradation of cloud-hosted web applications is one of the leading causes of economic loss in businesses (de Paula Junior et al., 2015), and it is a major cause of customer dissatisfaction when using web applications (Buyya et al., 2018; de Paula Junior et al., 2015).

This study will investigate and create solutions to combat the negative effects of flash crowds and resource failures, thereby alleviating performance degradation in cloud-hosted web applications. Research shows that an approach to address performance degradation is to use customised and targeted load-balancing techniques that take into account the key factors that affect the real-time behaviour of a server and cloud resource (S.-L. Chen et al., 2017; Z. Chen et al., 2018). The proposed solutions will include load distribution algorithms and cloud deployment strategies

for cloud-hosted web applications. The proposed algorithms will combine key server metrics for distributing three-tier web application workloads. This study will also create a novel multi-cloud algorithm and deployment strategies to mitigate flash crowds, resource failure, low network latency, and legislative issues. By adopting a multi-cloud deployment strategy, the over-provisioning of resources, vendor lock-ins, service availability, and customisation issues can be avoided. The proposed algorithms will be experimentally evaluated on real cloud infrastructures, and their performance will be compared with the performance of benchmarked algorithms, and performance metrics such as response times, throughput, and scalability will be used to characterise the performance of the proposed algorithms.

1.3 Research Motivation

Research has identified that faulty cloud-hosted web applications occur partly because of the lack of the use of right design patterns and principles (Ali et al., 2020; Fehling, Leymann, Rette, et al., 2014). It is therefore pertinent to investigate current cloud design patterns and principles with the view of proposing much better design patterns. Furthermore, studies (S.-L. Chen et al., 2017; Grozev & Buyya, 2014b; Tychalas & Karatza, 2020) show that the standard load balancing technique is not sufficient for most cloud-based applications. Hence, the reason for performance degradation in cloud-hosted applications. These studies corroborate the fact that there is a need for a dynamic and real-time capacity-focused load balancing algorithm and deployment strategies for cloud-hosted web applications.

An approach to alleviating issues associated with a standard load balancing algorithm is to incorporate into the algorithm the key factors that affect the real-time load of a virtual machine (VM). This research identifies the following key server metrics, namely: CPU utilisation, memory utilisation, network bandwidth, number of threads running, and network buffers. These server metrics are considered to be the most relevant determinants of a VM's real-time capacity and load. Incorporating these factors will help a load balancer to make decisions based on the

current capacity and real-time load of a VM, and, therefore, better utilise available resources and improve performance.

The use of multiple clouds (multi-clouds) promises an improved quality of service (QoS), compliance with legislation, resilience, and cost-effectiveness for cloud-hosted applications. Multiple geographically distributed cloud sites can serve users worldwide with low network latency, thereby providing improved Quality of Experience (QoE). As a result, the question arises as to how these objectives can be achieved in conjunction, without adding significant overhead to the system (Grozev & Buyya, 2014a, 2014b). This research will improve the previously created load balancing algorithm to create a multi-cloud load distribution algorithm and deployment strategy that features a unique communication strategy. The use of the proposed unique communication strategy will ensure that the added overhead to the system is reduced while mitigating the challenges of cloud-hosted web applications.

1.4 Research Aim, Question, and Objectives

The aim of this research is to create novel cloud load distribution management and deployment techniques, which can alleviate the challenges of hosting web applications on cloud. More specifically, this research focuses on the following research question.

How do we ensure efficient deployment and hosting of web applications on cloud?
In this research, the following requirement constraints are imposed on the research question:

1. stringent reliability of cloud-native web application
2. minimised changes to existing applications that will be migrated to the cloud.
3. maintain quality of service in flash crowds and resource failures.

4. serve end users near geographical locations to experience better responsiveness and to adhere to legislative rules.

To achieve the aim of this research and answer the research question, the specific objectives of this research are as follows:

1. To critically investigate, analyse, and experiment with different approaches to application deployment, brokering techniques, resource management, and load distribution in the cloud.
2. To create a reliable experimental platform that can be used to evaluate the proposed cloud algorithms and techniques.
3. To develop a flexible and general application brokering architecture that can facilitate resource management and efficient workload distribution, especially for existing web applications that will be migrated to the cloud.
4. To develop novel load distribution and resource provisioning algorithms and workload management approach that can mitigate cloud-hosted application challenges.
5. To experimentally evaluate the novel cloud algorithms and techniques proposed.

1.5 Research Contribution

Following the previously defined research question and objectives, the contributions of this research are as follows:

1. A survey of the state-of-the-art cloud application deployment, resource management, and brokering techniques and architectures.

2. An enhanced cloud design pattern structure, enhanced multi-tenancy patterns, and a novel implementation of multi-tenancy on chosen web applications.

3. A bespoke private and multi-cloud experimental environments to evaluate cloud algorithms and techniques.

4. A novel hybrid-dynamic weight assignment load balancing algorithm to improve workload distribution of web applications and combat the negative effect of flash crowds and resource failures.

    - The identification and introduction of a key server metric that forms part of the novel algorithm's server metrics.
    - A design approach for deploying existing web applications to the cloud with minimal changes to the code base.
    - Experimental evaluation through extensive cloud infrastructure and workload model.

5. A novel decentralised multi-cloud load balancing architecture and algorithm that alleviates the challenges of vendor lock-in issues, legislative compliance issues, low network latency issues, and the negative effects of flash crowds and resource failures.

    - An improved architecture to properly distribute the workload of three-tier web applications across multiple clouds.
    - An improved communication protocol of multi-cloud load balancing system.

The following publications were also contributions of this study.

- Adewojo, A.A., Bass, J.M. A Novel Weight-Assignment Load Balancing Algorithm for Cloud Applications.*SN COMPUT. SCI. 4, 270 (2023)*. https://doi-org.salford.idm.oclc.org/10.1007/s42979-023-01702-7.

- Adewojo, A. A., & Bass, J. M., (2022). A novel weight-assignment load balancing algorithm for cloud applications. *CLOSER 2022: 12th International Conference on Cloud Computing and Services Science, 86-96.* [1]

- Adewojo, A. A., & Bass, J. M., (2022). Multi-cloud load distribution for three-tier applications. *CLOSER 2022: 12th International Conference on Cloud Computing and Services Science, 296-304.*

- Adewojo, A. A., & Bass, J. M. (2018). Evaluating the effect of multi-tenancy patterns in containerized cloud-hosted content management system. *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), 278–282.*

- Adewojo, A. A., Bass, J. M., & Allison, I. K. (2015). Enhanced cloud patterns: A case study of multi-tenancy patterns. *2015 International Conference on Information Society (i-Society), 53–58.*

- Adewojo, A. A., Bass, J. M., Allison, I. K., & Hui, K. (2015). Cloud deployment patterns: Migrating a database driven application to the cloud using design patterns. *Proceedings of the World Congress on Engineering and Computer Science, 1, 198–203*

1.6 Thesis Organisation

The structure of the thesis in terms of their logical dependencies is indicated in Figure 1.1.
The remaining chapters of this thesis are organised as follows:

- **Chapter 2** presents the related literature on design patterns, cloud migration process, methods and cloud deployment tools, strategies, and technologies.

[1] Awarded Best Student Paper.

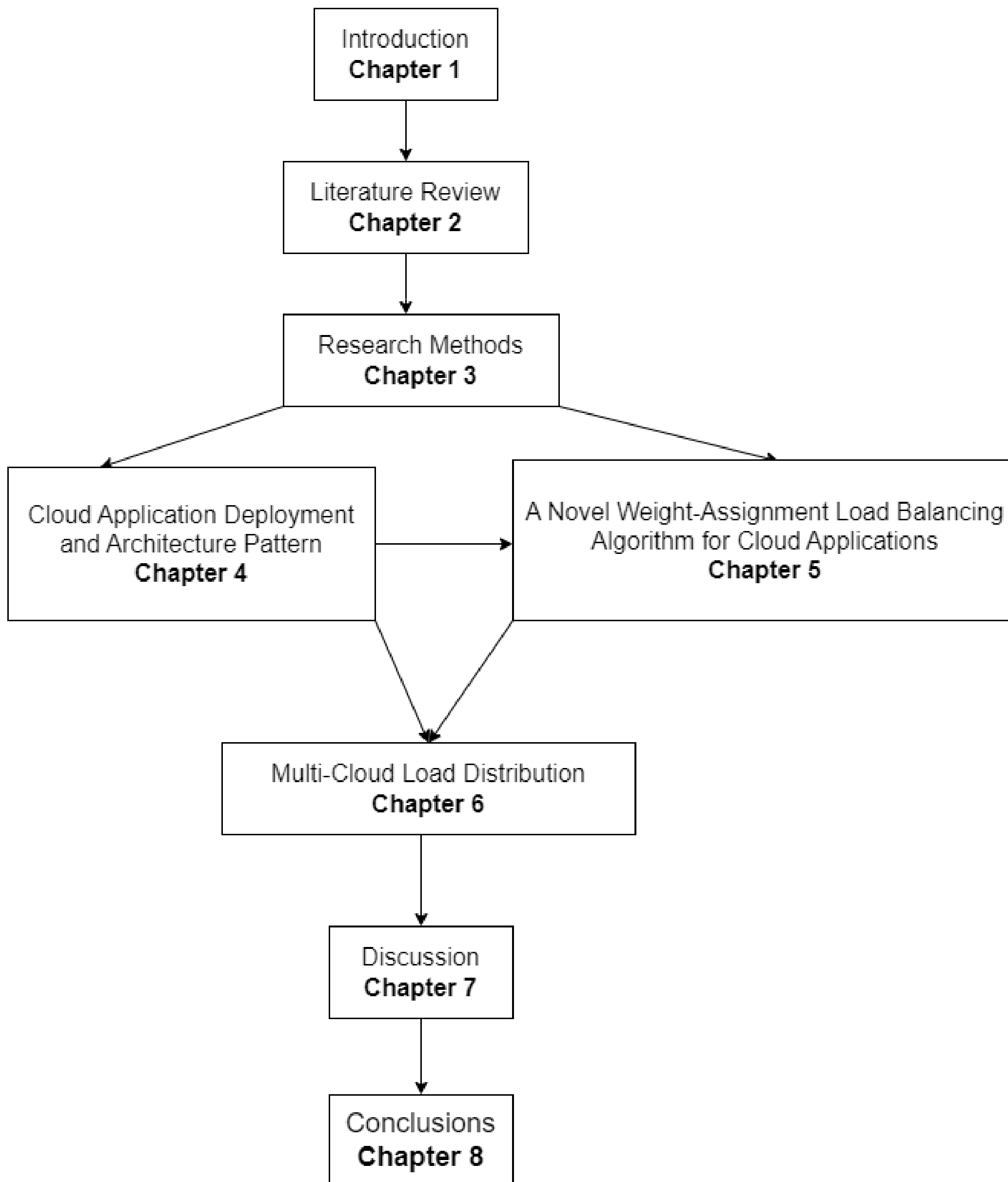


Figure 1.1: Thesis Structure and Logical Dependencies.

Furthermore, this chapter reviews resource management and brokering technologies for cloud-based applications and experimental cloud environments that are used to evaluate cloud-based algorithms.

- **Chapter 3** presents the research approach and methodologies that were used to conduct the research in this thesis.

- **Chapter 4** presents enhanced cloud design pattern structure, enhanced multi-tenancy patterns, and a novel implementation of multi-tenancy patterns (Adewojo, Bass, & Allison, 2015; Adewojo, Bass, Allison, & Hui, 2015; Adewojo & Bass, 2018).

- **Chapter 5** presents a novel approach to mitigating the challenges of cloud-deployed applications. It proposes a novel load distribution algorithm for three-tier web applications deployed on the cloud using a novel weighting technique and dynamic load balancing strategy. This algorithm improves resource provisioning and workload management for three-tier applications deployed on cloud (Adewojo & Bass, 2022b, 2023).

- **Chapter 6** presents a novel approach for deploying three-tier web applications across multiple clouds through a de-centralised deployment architecture. It further presents a novel multi-cloud load balancing algorithm for distributing the workload of three-tier web applications deployed across multiple clouds (Adewojo & Bass, 2022a).

- **Chapter 7** discusses the findings of the research, compares them to existing research work, and positions the contributions of the research.

- **Chapter 8** concludes the thesis, revisits and summarises the research contributions, and suggests the scope for future work.

Chapter 2

Literature Review

2.1 Introduction

This chapter presents an overview of existing literature on design patterns, cloud design patterns, deployment strategies for cloud-hosted applications, and techniques to mitigate the challenges of cloud-hosted web applications.

This chapter also presents knowledge from articles, reports, conference proceedings, books, and experimental research conducted during this research. Overall, this chapter contributes to the thesis by justifying the necessity of cloud patterns and novel load distribution management and deployment techniques.

2.2 Cloud Computing

Cloud computing provides scalable resource provisioning, IT infrastructure, development platforms, data storage and software applications over the Internet on a pay-as-you-use manner (Armbrust et al., 2010; Rimal & Maier, 2017; Varghese & Buyya, 2018; Walraven et al., 2012). The National Institute of Standards

and Technology (NIST) defines the cloud as "a model for enabling ubiquitous, convenient, and on-demand access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" (Mell, Grance, et al., 2011). In addition, Armbrust *et al.* (Armbrust et al., 2010) specified that the definition of cloud computing includes both the applications delivered as a service and the hardware and systems software in the cloud data centres that provide those services. Although there exist many definitions of cloud computing, the common agreed concepts and characteristics of this trending model according to Buyya *et al.* (Buyya et al., 2010) are on-demand self-service, pay-per-use, broad network access, elastic capacity, and virtualised resources (Buyya et al., 2010; Mell, Grance, et al., 2011).

2.2.1 Cloud Service Model

Cloud service models describe the style for which cloud providers offer IT resources. According to the NIST definition of cloud, the cloud is composed of three service models.

1. Software as a Service (SaaS). SaaS is a cloud service model that offers running and complete applications to customers as a service (Fehling, Leymann, Retter, Schupeck, et al., 2014; Mell, Grance, et al., 2011). These applications are accessible from different client devices such as a web browser, a thin client, or a program interface. Furthermore, the ability to manage the underlying infrastructure (network, servers, operating systems, and storage) or application capabilities is solely the responsibility of the cloud providers such as Salesforce, and Cloud-based WordPress. SaaS eliminates the need for personal computing infrastructure for application deployment.

2. Platform as a Service (SaaS). PaaS is a cloud service model that offers the complete execution environment and deployment platform for a specific type of applications (Fehling, Leymann, Retter, Schupeck, et al., 2014; J.-F.

Zhao & Zhou, 2014). Customers are given the ability to deploy on to cloud infrastructures their bespoke or acquired applications using programming tools that are provided by the cloud provider. Similar to SaaS, consumers do not manage or control the underlying cloud infrastructures but have control over the deployed applications and configuration settings for the application hosting environment (Mell, Grance, et al., 2011). Google App Engine is an example of a PaaS.

3. Infrastructure as a Service (IaaS). IaaS is a cloud service model that allows consumers to provision processing, storage, network, and other fundamental computing resources. Examples of IaaS providers include Amazon Web Services (AWS), Windows Azure, and IBM Cloud.

2.2.2 Cloud Deployment Models

Cloud deployment models represent the cloud environments that host the computing resources and the group of consumers that can access these resources. According to the NIST definition of cloud computing, there are four cloud deployment models as discussed below.

1. Public: A public cloud offers its cloud services and infrastructure to everyone (Fehling, Leymann, Retter, Schupeck, et al., 2014; Mell, Grance, et al., 2011).

2. Private: A private cloud is exclusively created for a specific organisation, hence it is available only to accredited members of that organisation (Fehling, Leymann, Retter, Schupeck, et al., 2014; Mell, Grance, et al., 2011).

3. Community: A community cloud offers its services exclusively to a specific community of consumers from different organisations that have shared concerns and trust each other (Fehling, Leymann, Retter, Schupeck, et al., 2014; Mell, Grance, et al., 2011).

4. Hybrid: A hybrid cloud combines two or more discrete cloud infrastructures (private, public, or community) that remain distinct entities but are linked by proprietary or standardised technologies for data and application portability. An example of proprietary technology is cloud bursting which balances the load between clouds. It is the usage of external cloud resources when local ones are insufficient (Mell, Grance, et al., 2011).

   The multi-cloud is a type of hybrid cloud that combines multiple independent cloud infrastructures through a client or service (Buyya et al., 2018; Grozev & Buyya, 2014b). As discussed previously, multi-clouds facilitate the development and deployment of responsive cloud-hosted web applications. A novel implementation of multi-cloud as a deployment strategy for managing web applications' workload and combating resource failures and flash crowds will be discussed in 2.8.3.

2.3 Cloud-Hosted Applications Architectures

There are commonly four architectural styles that cloud-native applications follow ("Four Architecture Choices for Application Development in the Digital Age: Which application architecture model is best for you in the cloud era?", 2020). They are Monolithic, Microservices, Event-driven serverless, and cloud-based architecture.

2.3.1 Monolithic

Monolithic architecture encapsulates several tightly coupled functions and functionality into a single application (De Lauretis, 2019; Ponce et al., 2019). Monolithic applications are designed to handle multiple related tasks and so their modules cannot be executed independently. This architecture favours a quick deployment of newly developed applications (easier to develop, deploy, and manage). However, this architecture no longer meets the needs of scalability and elasticity which are

key properties of cloud and rapid development cycles and a key property of cloud deployed applications. Common issues that are associated with this architectural style are a large code base, the inability to maintain code, and the low productivity of developers (Ponce et al., 2019).

2.3.2 Microservices

In a microservices architecture, the application is structured as a collection of services. It is possible to write each service in a different programming language and test it separately. They can be deployed independently and are organized around business capabilities ("Four Architecture Choices for Application Development in the Digital Age: Which application architecture model is best for you in the cloud era?", 2020). Microservices promise agility, autonomy, scalability, and reusability (Vale et al., 2022). They are used primarily for large-scale and cloud-native commercial applications.

2.3.3 Event-driven serverless

An event-driven architecture (EDA) consists of decoupled systems that respond to events. Events are used to trigger and communicate between decoupled services in an event-driven architecture. While EDA has existed for a long time, it now has more relevance in the cloud because the architecture style provides a significant increase in agility, cost savings, and operational benefits ("Four Architecture Choices for Application Development in the Digital Age: Which application architecture model is best for you in the cloud era?", 2020).

2.3.4 Cloud-based architecture

A cloud-based architecture is an abridged architecture that improves a monolithic architecture to suit the cloud. The cloud-based architecture is best suited for building and deploying modern web applications in the cloud. This architecture typically involves load balancers, web servers, application servers, and databases. It can benefit from cloud features such as resource elasticity, software-defined networking, auto-provisioning, high availability, and scalability ("Four Architecture Choices for Application Development in the Digital Age: Which application architecture model is best for you in the cloud era?", 2020).

The decision on which architecture model is best for an application depends on the type of application and non-functional requirements. However, the deployment of applications that follows any of these architectural styles often follows the tiered architectural style. The three-tier deployment architecture is a prevalent deployment architecture for cloud-hosted web applications (Grozev & Buyya, 2014b).

Three-tier architecture is an enterprise software architecture and a common architectural pattern for deploying web applications (Fowler, 2002; IBM Cloud Education, 2020). The three-tier architecture is the most popular implementation of a multi-tier architecture (Amazon, 2021b). This architecture supports loose coupling, scalability, reliability, efficient load balancing, and so on (Archiveddocs, 2022b; Fowler, 2002; Grozev & Buyya, 2014b; "Rockford Lhotka - Should all apps be n-tier?", 2020). It features three physical deployment tiers of a software application's logical layers (code organization). Therefore, a three-tier web application is an interactive application that can be accessed over the Internet, with three or more layers deployed across three physical tiers/machines (Fowler, 2002; Ramirez, 2000). The tiers are commonly referred to as Client Tier, Business/App Tier, and Data Tier.

- Client Tier: This tier commonly hosts the presentation layer of the web application. The presentation layer is the user interface of the web application.

- Business/App Tier: The business logic/domain layer, service layer, and sometimes an application facade are deployed on this tier. This tier commonly has the application server installed on it. The business logic is responsible for the manipulation of application data, the application of business rules and policies, and ensuring data consistency and validity.
- Data Tier: The data tier hosts the application that stores and manages the information processed by the software application.

As a result of the variety of options offered by cloud computing, it is a popular choice for hosting and deploying software applications(Buyya et al., 2018; Grozev & Buyya, 2014b). Several studies have shown that businesses are rapidly moving their applications to the cloud because of its perceived benefits (Buyya et al., 2018; Fehling, Leymann, Retter, Schupeck, et al., 2014; Grozev & Buyya, 2014b; Odun-Ayo et al., 2017). There are, however, a number of factors to consider when migrating or deploying applications to the cloud. These considerations are particularly critical because they influence the quality and efficiency of cloud-hosted applications. Some of these factors include software design patterns, cloud architecture patterns, cloud deployment strategies, and cloud resource management techniques.

Also, having introduced cloud service models and cloud deployment types, it is imperative to describe design patterns that software architects and developers can use to build cloud-native applications. When developing cloud-native applications, the use of cloud design patterns is a key step toward ensuring a successful deployment.

2.4 Design Patterns

A design pattern is a general, reusable solution to a commonly occurring problem. Design patterns solve recurring problems inherent in a particular design (Gamma et al., 1994). To communicate and express how these design patterns are used, groups of related patterns are collated to form a pattern language. Patterns are largely

abstract and independent of the programming language or runtime infrastructure used (Ali et al., 2020; Fehling, Leymann, Retter, Schupeck, et al., 2014). However, some design pattern catalogues include code samples or snippets that show how to implement the pattern on the desired technology (Wilder, 2012).

2.4.1 Background on Design Patterns

Architectural and design patterns have long been used to provide known solutions to many common problems facing distributed systems (Bass et al., 2012; Gamma et al., 1994). The concept of cloud pattern, which had its origin from a building's architectural pattern, was proposed by Christopher Alexander in the early 1970s (Alexander, 1977; Alexander et al., 1979). The knowledge of architecture and its associated best practices which were captured in a pattern format, influenced several other practices in the early days including tailoring and designing as well as computing. By the late 1980s, Kent Beck and Ward Cunningham (Gamma et al., 1994) adopted this concept of pattern, and began experimenting with the idea of patterns to programming.

Their experimentation culminated in a summarised system of five patterns which they successfully used to design windows-based user interfaces, and presented a detailed pattern language for object-oriented programs at the Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) conference in 1987 (Cunningham & Beck, 1989). Improving on the work done by Kent *et al.* (Cunningham & Beck, 1989), Gamma *et al.* (Gamma et al., 1994) created design patterns for object-oriented (OOP) software (Gamma et al., 1994) which led to the modern-day popularity of patterns in computing. Many more studies have improved patterns in computing (Beck, 1997; Fowler, 2002; Freeman et al., 2008; Hansen, 1995; Hohpe & Woolf, 2004; Wilder, 2012).

Patterns are not a finished design product, but rather templates or structured techniques to solve a problem and can be transformed into usable products. In cloud computing, patterns can be transformed into source codes or infrastructure designs.

They can also be duplicated to produce the expected outcome. Their use ranges from evaluating cloud offerings, building cloud-ready applications, and building custom cloud offerings to evaluate application landscapes for cloud-readiness (Fehling, Leymann, Retter, Schupeck, et al., 2014).

Recently, researchers have focused on distributed models of software application and development of which the cloud is an example (Adewojo, Bass, Allison, & Hui, 2015). Cloud computing services centralise deployment, maintenance and the evolution of software applications while catering for a worldwide user audience (Adewojo, Bass, & Allison, 2015; Adewojo, Bass, Allison, & Hui, 2015). These advantages of the cloud have attracted many organisations to adopt cloud computing and its hosting capabilities.

Many companies have moved or are in the process of migrating their on-premise applications and IT infrastructure to the cloud as a result of the popularity of cloud computing (Adewojo & Bass, 2018; Buyya et al., 2018; Odun-Ayo et al., 2017). However, migrating or developing cloud native applications is potentially complex. In addition, effective cloud migration, and/or design of a cloud solution, requires considering issues that may not become visible until later in the implementation. Cloud design patterns help to prevent such issues by providing general solutions and directions that are not tied to a specific development language or problem. Cloud patterns can speed up the development process by providing tested, proven development paradigms and thereby assist organisations to overcome challenges associated with hosting a cloud application. In addition, the use of cloud patterns ensures that reliable, scalable, and secure applications are built or migrated to the cloud because a cloud pattern describes the problem it addresses as well as offers considerations for applying the pattern.

2.4.2 Object-Oriented Design Patterns

In a bid to create a structured approach to solving recurring problems in object-oriented software design, Gamma *et al.* (Gamma et al., 1994) proposed a pat-

tern language for object-oriented systems. Object-oriented design is one of the most popular application design methodologies (Bernstein & Newcomer, 2009; Fowler, 2002; Gahlyan & Narayan Singh, 2018; Kurmangali et al., 2022; Lasso & Kazanzides, 2020). It is a revolutionary way of thinking that focuses on modelling things as objects rather than functions (Tupper, 2011).

Object-oriented systems are viewed as a collection of objects whose state is distributed among the objects with each object handling its state data. The design focus is on identifying problems as a set of objects, so that the development of a software application can be broken down into as small as possible identifiable problems. Gamma *et al.* (Gamma et al., 1994) used a systematic approach to catalogue twenty-three different object-oriented design patterns that can solve recurrent problems in object-oriented systems.

The catalogue is divided into creational patterns (patterns for creating objects), structural patterns (patterns for composing objects into larger structures), and behavioural patterns (patterns for interactions between objects). These patterns are as follows: Strategy, Decorator, Factory Method, Observer, Chain of Responsibility, Singleton, Flyweight, Adapter, Facade, Template, Builder, Iterator, Composite, Command, Mediator, State, Proxy, Abstract Factory, Bridge, Interpreter, Memento, Prototype, and Visitor. Since then, these patterns have been widely and consistently used to design object-oriented systems (Ali et al., 2020; Haq et al., 2017; Kurmangali et al., 2022; M. Zhang & Wang, 2000). Furthermore, most programming languages used in creating software and web applications are object-oriented languages, so the principles found in these patterns are still being employed in developing web applications. While the pattern catalogue is widely accepted, it is old, and software development, particularly web application development, has evolved. Although some patterns have become part of the core lexicon of object-oriented programming and some have formed the bedrock for new patterns in software development, the original design patterns alone do not suffice for cloud-hosted applications. They are more subtle compared to current design patterns for web or cloud hosted applications. However, the rigour of description, comparisons, and discussion of these patterns is valuable for recent design pattern catalogues.

A similar study to that by Gamma *et al.* catalogued seven different patterns for object-oriented analysis (OOA) and object-oriented design (OOD) (Coad, 1992). Patterns were illustrated with examples and guidelines for their use. Martin (Martin, n.d.) further improved this research by investigating the effect of applying and not applying the right design patterns to object-oriented design. They discussed the positive and negative effects of their investigations and created a guideline on architecting software applications when using design patterns. They also summarised the major symptoms of rotting designs and how design patterns can help alleviate these problems. Examples of these symptoms are rigidity, fragility, immobility, and viscosity.

Although object-oriented design patterns are primarily applicable to object-oriented systems, this research relates the principles behind object-oriented design patterns to cloud-native applications. Cloud-hosted applications have key properties such as scalability, multi-tenancy, flexibility, manageability, and portability (Adewojo, Bass, & Allison, 2015). Object-oriented design patterns can be used as a tool to improve these properties, especially when migrating existing software applications to the cloud.

For instance, the strategy pattern is useful in multi-tenant applications. Using the strategy pattern, the concept of multi-tenancy can be improved so that the same functionality can be shared among multiple tenants with different looks and outputs. For scalability, observer patterns are useful. The reason for this is that agents are commonly used to monitor cloud applications. As a result of the observer pattern, I am able to understand how an agent signals to other agents the change of their state as well as the need to obtain additional resources automatically.

2.5 Cloud Design and Architecture Patterns

The previous section established that design patterns are applicable to distributed systems, and that the cloud is a distributed system (Adewojo, Bass, Allison, &

Hui, 2015; Bass et al., 2012; Fehling, Leymann, Retter, Schupeck, et al., 2014). Therefore, a consistent cloud pattern and pattern language are essential. This section explores cloud design and architectural patterns as a step in identifying and solving recurring issues that affect the deployment of cloud-hosted applications. Cloud patterns offer a verifiable and reusable cloud structuring solution to recurring cloud problems (Adewojo, Bass, Allison, & Hui, 2015). A cloud pattern is a well-defined format for describing a suitable solution to a cloud-related problem (Fehling, Leymann, Rette, et al., 2014; Fehling, Leymann, Retter, Schupeck, et al., 2014). These patterns are abstract, technology-neutral, and independent of the used programming language or runtime environment.

Cloud design patterns is a generalised pattern catalogue that focuses on all properties of cloud computing. They are the best practise for building cloud services (Fehling, Leymann, Retter, Schupeck, et al., 2014). Cloud architectural patterns focus on the compositions of architectural elements that can be used to build cloud-native applications that exhibit essential cloud properties such as rapid elasticity, resource pooling, on-demand self-service, and so on. These patterns guide architects, software developers, and others on different types of cloud offerings and how to build applications on them (Fehling, Leymann, Retter, Schupeck, et al., 2014). Additionally, software architects place significant weight on cloud patterns because they dictate whether the system's quality characteristics will be manifested (Bass et al., 2012; Fehling, Leymann, Retter, Schupeck, et al., 2014).

2.5.1 Cloud Architecture Patterns

Architectural patterns are compositions of architectural elements that provide packaged strategies for solving recurring problems in a system (Bass et al., 2012). In cloud computing, cloud architecture patterns describe how applications should be designed to benefit from a cloud environment (Fehling, Leymann, Retter, Schupeck, et al., 2014). Furthermore, they describe how these applications can be offered as configurable cloud services.

Homer *et al.* (Homer et al., 2014) presented (i) twenty-four patterns that are useful in developing cloud-hosted applications, (ii) two primers and eight guidance topics that provide basic information and industry-standard practice techniques for developing cloud-hosted applications, and, (iii) ten sample applications that illustrate how to implement the design patterns using features of Windows Azure. The sample code (written in C#) for these sample applications was included in the pattern. This makes it easy for architects who intend to use similar cloud patterns to convert the code to other web programming languages (such as Java and Python) for use on other cloud platforms. Software architects tend to benefit from this pattern format, which is uncommon and usually comprehensive.

Erl *et al.* (Erl et al., 2015) present a catalogue of over 100 cloud design patterns for developing, maintaining, and evolving cloud-deployed applications. The cloud patterns, which are divided into eight groups, cover several aspects of cloud computing, including scaling and elasticity, reliability and resilience, data management, and network security and management. For example, patterns such as shared resources, workload distribution and dynamic scalability (which are listed under the "sharing, scaling and elasticity" category) are used for workload management and overall optimisation of the cloud environment. The major strength of Erl *et al.*'s catalogue of cloud patterns is its extensive coverage of techniques for handling the security challenges of cloud-hosted applications. It describes various strategies covering areas such as hypervisor attack vectors, threat mitigation, and mobile device management.

Jamshidi *et al.* (Jamshidi et al., 2014) presented a catalogue of fine-grained service-based cloud architecture migration patterns that target multi-cloud settings which are specified with architectural notations. The key patterns reflect the different construction principles for cloud architecture: re-deployment, cloudification, relocation, refactoring, rebinding, replacement, and modernisation. These patterns are described as migration strategies, decision making, and best practices for cloud migration. Therefore, these patterns are different from cloud patterns such as those shown in the following studies (Fehling, Leymann, Retter, Schupeck, et al., 2014; Mendonca, 2014; Wilder, 2012). And, so they may not be applied at runtime during

the design and deployment of cloud applications. Some other documentation of cloud deployment patterns can be found in the following studies (Musser, 2012; Strauch et al., 2012; Varia, 2010, 2011).

A collection of over seventy-five patterns for building and managing cloud-native applications were provided by Fehling *et al.* (Fehling, Leymann, Retter, Schupeck, et al., 2014). The "known uses" of the implementation of each pattern is provided with examples of cloud providers offering products that exhibit the properties described in the pattern. This helps us to better understand the core properties of each pattern.

According to Fehling *et al.* (Fehling, Leymann, Retter, Schupeck, et al., 2014), cloud architecture patterns consist of fundamental application architectural patterns, cloud application component patterns, cloud integration patterns, and multi-tenancy patterns. Fundamental application architecture patterns cover the architectural principles found in most cloud-native applications. It is an umbrella for the remaining three categories of patterns covered within the cloud architecture pattern. These patterns specify how to design and build individual components of cloud-native applications so that the overall application can be built on top of an elastic platform. Multi-tenancy patterns describe how cloud applications and individual components can be shared by multiple customers called tenants at different levels of the application stack. Cloud integration patterns describe a process for integrating multiple cloud environments or cloud components as well as on-premise data centres and applications both inside and outside the cloud. These patterns can also be classified as cloud deployment patterns because they embody decisions about how elements of the cloud application will be assigned.

The above research shows different ways to represent cloud design patterns. A consistent and elaborate pattern catalogue structure, however, will benefit these cloud patterns by aiding software architects in developing cloud-native applications with less development effort. Therefore, this research will address the gaps found in the description of cloud patterns by Fehling *et al.* (Fehling, Leymann, Retter, Schupeck, et al., 2014). In addition, this research will create an enhanced and

systematically-improved cloud pattern structure. It will contribute to a consistent and elaborate pattern catalogue structure. This updated structure will include a formal description and code snippets for cloud patterns. The enhanced pattern structure will be applied to the multi-tenancy patterns and practical implementation of this pattern will be created on case study applications.

Multi-tenancy Patterns

One of the essential cloud properties is resource sharing (Fehling, Leymann, Retter, Schupeck, et al., 2014; Mell, Grance, et al., 2011). Resource sharing at different levels of the cloud is a common practice, and multi-tenancy is the architectural framework that enables resource sharing. At the infrastructure layer, multi-tenancy describes the extent to which cloud resources can be shared while guaranteeing isolation among components (tenants) using these resources (Adewojo, Bass, & Allison, 2015).

Multi-tenancy is also a type of software architecture and an architectural pattern for Software as a Service (SaaS) that enables resource sharing, so that tenants (customers) can share the same instance of software efficiently (Wilder, 2012). Multi-tenancy patterns describe how cloud applications and individual components can be shared by multiple users on different levels of the application stack. Multi-tenancy pattern allows tenants to share more than just the application, including capital and operational expenses of using the cloud. Furthermore, multi-tenancy pattern enables SaaS to become configurable to suit tenants' needs. These benefits of multi-tenancy make it one of the essential cloud properties and SaaS' default deployment architectural style. Figure 2.1 shows the advantages and disadvantages of multi-tenancy patterns in a pictorial view.
There are several ways to implement multi-tenancy in a software application (Krebs et al., 2013; Odun-Ayo et al., 2017; Strauch et al., 2012). This is because multi-tenancy can be implemented at different layers of the cloud stack: application layer, middleware layer, and data layer (Fehling, Leymann, Retter, Schupeck, et al., 2014; Wilder, 2012). Also, cloud providers implement multi-tenancy at

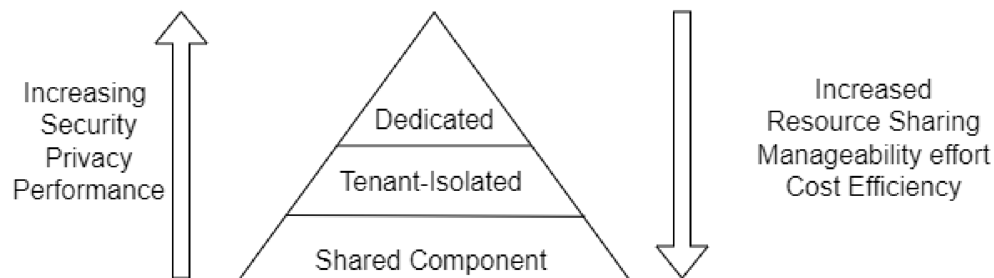


Figure 2.1: Multi-tenancy Patterns

the PaaS level so that they can provide customised versions of the same service to multiple customers thereby exploiting economies of scale. This means it is important to consider the components of the applications that will share resources when developing a cloud-native application. It is equally important to consider the cloud infrastructure that can and will be shared with other applications or other cloud instances. Therefore, during the design of cloud-native applications, three levels/degrees of multi-tenancy have to be considered for applications to fully benefit from the cloud's resource-sharing features (Fehling, Leymann, Retter, Schupeck, et al., 2014). The three degrees of multi-tenancy patterns are: shared components, tenant-isolated components, and dedicated components. These patterns apply to different components and application needs. The degree of isolation between tenants enabled by multi-tenancy is the differentiating factor among the degrees of multi-tenancy patterns. Isolation is defined in terms of performance, data volume and security and is one of the essential factors for choosing the correct multi-tenancy pattern.

- **Shared Component Pattern**
  The first level/degree of the multi-tenancy pattern is the shared component pattern. It is the basic minimum requirement for resource sharing in a SaaS application. It minimizes resource usage. However, this is at the risk of data and processes isolation (Adewojo, Bass, Allison, & Hui, 2015; Ochei et al., 2015). It is used to provide functionality to different tenants without maintaining a notion of tenants itself. As a result of this mode of setup, tenants can influence each other while this functionality is being accessed

(Adewojo, Bass, & Allison, 2015; Fehling, Leymann, Retter, Schupeck, et al., 2014). Shared component pattern enables the highest form of sharing. It results in lower operational costs by reducing the cost per tenant, and simpler management of resources. In the case of SaaS, tenants of the software application share the same instance of the application. Examples of SaaS that use the shared component pattern are Salesforce CRM system, and NOASS web service (Fehling, Leymann, Rette, et al., 2014; Fehling, Leymann, Retter, Schupeck, et al., 2014). Although the shared component pattern enables the highest form of sharing, this model of resource sharing can become difficult to provide guaranteed performance, privacy, and security. Hence, it is not usually suitable for organisations or applications with critical needs, legislative requirements, and those whose computing infrastructure majorly relies on cloud.

- **Tenant-Isolated Pattern**
  The second level/degree of the multi-tenancy pattern is the Tenant-Isolated component pattern. This level exemplifies the ability to balance cloud resource sharing and privacy with intermediate security level. Tenant-Isolated component pattern is a compromised implementation between shared and dedicated component pattern (Adewojo, Bass, & Allison, 2015). Resources are commonly shared with intermediate level of performance, privacy, security, and resource overhead. Tenant-isolated pattern provides a middle ground for resource sharing while performance, privacy, and security can be guaranteed.

- **Dedicated Component Pattern**
  The dedicated component pattern is the third and highest level/degree of the multi-tenancy pattern. This pattern provides exclusive access to application components that provide critical functionality (Adewojo, Bass, & Allison, 2015). Security, privacy, and isolation are maximized when using this pattern. However, this is at a higher cost in terms of finance, and use of hardware.

2.5.2 Formal Description of Design Patterns

There are research works that advocate the use of formal language to describe design patterns. Formal language reduces ambiguity when specifying software requirements and patterns. It improves accuracy and efficiency in the implementation of software applications, especially for software architects. The purpose of formal language theory is to bring order to complex system anarchy (Révész, 1991). Formal languages are characterised by predefined rules, such as formal notations in mathematics, logic, and computer science (Brookshear, 1989; Révész, 1991).

In this research, the Z language will be used to formally describe the multi-tenancy pattern. Z language will be used because it is a state-based language. Its domain of use is not limited, and it is fairly easy to understand. Z is old and stable. According to Lana *et al.* (Lana et al., 2019) the use of Z language dominated the space of formal language between the years 2011 and 2015. This is around the time that this research began. Furthermore, Saratha *et al.* (Saratha et al., 2017) considers the Z language as a domain-neutral and consistent language.

2.6 Cloud Migration Process and Methodology

Apart from the use of design patterns to ensure a successful deployment of cloud-native applications, researchers have worked on methodologies to migrate existing applications to the cloud. The sections below review researchers' efforts in migrating existing applications to the cloud.

2.6.1 Cloud Migration Process

Researchers have proposed different processes to migrate software systems to the cloud. These processes aim to migrate either the whole application to the

cloud at once, or part of the application that is cloud-ready, and then focus on some transformation process of the remaining part before migration. Gunka *et al.* (Gunka et al., 2013) used an evolutionary algorithm to describe and implement this process based on the on-going work of MODAClouds EU project (Ardagna et al., 2012). Model-based techniques were used to support the steps of the transitioning process. They created a baseline for the development of appropriate deployment architectures and the selection of suitable cloud providers.

Zhao *et al.* (J.-F. Zhao & Zhou, 2014) analysed the research achievements and application status of legacy system migration to the cloud; based on this analysis, they came up with five strategies of migrating legacy system to the cloud based on the cloud service model. These strategies are Migrate to IaaS, Migrate to PaaS, Replace by SaaS, Revise based on SaaS, and Re-engineer to SaaS.

Cretella *et al.*(Cretella & Di Martino, 2014) proposed an overview of migration approaches based on cloud patterns. These migration approaches are the use of cloud patterns, the use of a model-driven engineering concept, and semantic modelling.

Zhang *et al.* (W. B. Zhang et al., 2009) proposed a generic process to guide software developers on how to migrate legacy systems to the cloud. The process includes the following steps: legacy system representation, redesign of the architecture model with identified services, model-driven architecture transformation, web service generation, invocation of legacies functionalities, selection of suitable cloud platform and provision of cloud web service to the end users. This process will produce a new web service application that can maintain the workflow of the legacy system but provided as a SaaS on cloud. The process was applied to a case study application but no concrete result was presented.

2.6.2 Cloud Migration Methods

There are several proposed cloud migration methods. Some proven methods are discussed as follows: Venugopal *et al.* (Venugopal et al., 2011) proposed a methodology that used a connection-oriented framework that smoothly migrates and tunes a web service based enterprise application to the cloud. It is a non-invasive approach that observes the performance of a web service based system and appropriately reconfigures this system to achieve higher performance. It is aimed at mass migrations of enterprise applications to the cloud. However, it targets specific cloud platforms.

To simplify the process of migrating applications to cloud, Ward *et al.* (Ward et al., 2010) developed a workload migration framework called Darwin. This framework is designed to promote migration to virtualised environments in a simplified method. In addition to the framework concept, Cardoso *et al.* (Cardoso et al., 2014) presented six steps for the framework workload definition. Total Cost of Ownership calculation, definition of criteria and multi-criteria method application were used to migrate real world applications to the cloud.

Peddigari *et al.* (Peddigari, 2011) proposed that the framework and factory patterns can be used in a systematic approach to migrate applications to the cloud. Based on this claim, they proposed a unified cloud migration framework to migrate applications to the cloud. To support this, Cai *et al.* (Cai et al., 2015) proposed a pattern-based code transformation approach to migrate applications to the cloud. This approach is supported by three key elements: namely, patterns, rules, templates, and a process that systematically applies these elements.

The above studies show that researchers claim that cloud migration methods can help migrate software applications to the cloud. However, drawing from the interactions between the literature on cloud migration process and methods, the conclusion of this research is that the proposed cloud migration methods should be combined with migration processes amongst others to provide a complete migration strategy.

2.6.3 Cloud Migration Tools

Andrikopoulos *et al.* (Andrikopoulos et al., 2013) developed a migration decision support system that helps to select an appropriate cloud provider based on cost minimization. They created a web-based system that combines state-of-the-art approaches to offer matches and cost calculation of migrating an existing application to the cloud. This system uses a set of requirements and a knowledge base of properties of existing cloud vendors to support migration to the cloud.

Khajeh-Hosseini *et al.* (Khajeh-Hosseini et al., 2011) proposed two tools to support decision-making, particularly in terms of the benefits and risks of migrating to the cloud and to determine resource usage patterns during migration to the cloud. The tools are modelling tools that estimate the costs of using an IaaS cloud and a spreadsheet outlining the benefits and risks of using IaaS.

Meng *et al.* (Meng et al., 2011) proposed a migration tool named Application Migration Solution (AMS). This tool will be used for converting legacy applications into web applications that can be migrated to the cloud. To convert a legacy system into a web application, this tool uses GUI recognition and reconstruction technology. However, this tool does not focus on the service layer. As a result, I conclude that a more comprehensive and customised tool is required to transform the application to suit the cloud.

The previous research work on cloud migration tools, methods, and processes only claimed successful cases of migration and translation of cloud processes. However, they have failed to showcase applications that demonstrate cloud properties, which is a defect of the previous research work. As a result, the benefits of cloud properties such as multi-tenancy, and flexibility that results in improved resource sharing, reliability, and improved ability to handle cloud challenges could not be fully realised in successfully migrated applications. To address this defect, this research study recommends a focus on ensuring that the applications being studied and migrated exemplify cloud properties.

In addition to migration tools, strategies, and processes, researchers proposed some architectural concepts to aid a successful migration of existing applications to cloud. Zhang et. al (W. B. Zhang et al., 2009) presents a cloud migration approach that uses key architectural principles like virtualisation of infrastructure, service orientation for reusable services, and extensible provisioning and subscription. Zhang's approach focuses on developing reusable architectures for cloud platforms as a framework for migrating existing applications to the cloud.

Johannes *et al.* (Wettinger et al., 2017) systematically classified and characterised available deployment approaches independently from the underlying technology used. They used this process to identify and characterise two different types of deployment approaches: namely, application and middleware-oriented approaches. They developed deployment plans based on the two identified approaches for three applications with significantly different deployment requirements. They successfully migrated the case study applications using their identified approaches. Their results show better reusability, portability, and flexibility of middleware-oriented plans when compared to application-oriented deployment plans.

A cloud-agnostic decision support system was introduced by Ardagna et. al (Ardagna et al., 2012). Ardagna *et al.* (Ardagna et al., 2012) used the model-driven approach to create this decision support system. The system was created to help solution architects to determine which cloud system to adopt for hosting different components of a software application. The decision support system is based on parameters such as costs, risks, and analyses of non-functional characteristics of each alternative provider.

The use of deployment technologies such as containers has also gained attention in the research community. Docker containers can be used to migrate applications without major modifications to the cloud. As cloud architectural platforms evolve and the need to reduce the amount of modification to existing applications arises, maximising cost efficiency of application deployments with less performance trade-offs often emerge. The advent of containers provides the opportunity to seamlessly deploy applications on cloud while significant software modification

and development effort is saved (Lahmann et al., 2018). Containerisation is a DevOps technology that offers containers that constitute the bundling of an application and its dependencies as a package that can run on many computing platforms (Muddinagiri et al., 2019). Docker is a widely popular containerisation platform and technology for creating containers. Kubernetes is the frontrunner in container orchestration tools. These software often work together seamlessly to facilitate the successful implementation of containerization both locally and on the cloud. Containers are popular for running huge volumes of applications, applications with microservice architectures (Govind & González–Vélez, 2021; Ha et al., 2017), migrating existing applications to the cloud as it is (Adewojo, Bass, & Allison, 2015; Hanafy et al., 2017), and for continuous deployment practices (Muddinagiri et al., 2019; Rajavaram et al., 2019). Additionally, research indicates that deployment of applications through containers is an efficient and effective way to improve cloud deployment (Hanafy et al., 2017). However, if the applications being deployed are not cloud-ready, they will not fully benefit from the features of cloud deployment (Lahmann et al., 2018).

The aforementioned research studies focus on the process and methods of migrating applications to the cloud. Research shows some efforts in using tools, processes, and methodologies for cloud migration. However, there is a gap in addressing common problems of hosting the software applications that are being migrated. This research will not only review the migration process and methods but also provide solutions to recurring problems of cloud-hosted applications through the use of design patterns and deployment strategies.

2.7 Common Challenges of Cloud-Hosted Web Applications

2.7.1 Flash Crowds

An increase in web user requests causes flash crowds, which are characterised by rapid, fluctuating, exponential, and legitimate increases in web request volume (Ari et al., 2003; Le et al., 2007; Qu et al., 2017). Managing flash crowds can be challenging because they arise without warning. Autoscaling services and load balancers are commonly used to address flash crowds (Amazon, 2021b; Grozev & Buyya, 2014b). Commercial cloud providers commonly use autoscaling services to launch new VMs after an application has experienced increased user requests for a specific period of time: a time usually set by the user. 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.

2.7.2 Resource Failures

Cloud resource failure occurs when any of the components of a cloud environment fails to function as it is intended to. According to (Prathiba & Sowvarnica, 2017; Priyadarsini & Arockiam, 2013), the three most common resource failures in any cloud environment are hardware, virtual machines and application failures. Also, authors in (Kumari & Kaur, 2021) 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. Cloud resource failures can occur suddenly, resulting in degradation of service performance or total loss of service. The most common method to mitigate the effects of resource failure is to use a timely intervention of autoscalers and load balancers. Meanwhile, an auto-scaler 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 (Qu et al., 2017).

2.8 Addressing the Challenges of Cloud-Hosted Web Applications

The following sections will discuss the use of single and multi-cloud deployment strategies and load balancing techniques as tools to address the challenges of cloud-hosted web applications.

2.8.1 Workload Distribution and Resource Management in Single Cloud

Efficient workload distribution and resource management are essential to addressing the challenges of cloud-hosted web applications. The successful deployment and hosting of cloud applications is also a function of how the resources of the cloud are managed and how the applications are set up to accommodate the workload that are passed to them. Managing workload distribution in cloud applications generally relies on load distribution techniques. This technique, which is popularly known as load balancing, has evolved over the years from serving servers farms of applications to being used in the cloud. Load balancing in the cloud is a method for optimally distributing workload so that the resources of a VM in a cloud computing environment are efficiently utilised. Cloud applications that serve global users usually use one form of a load-balancing tool. Since cloud computing services have become a vital part of companies, it has become even more essential to improve existing load-balancing techniques and enhance cloud application performance, for the number of resources are restricted.

Significant efforts have been made in developing and categorising load distribution techniques for the cloud. There are two categories of load-balancing techniques:

static and dynamic techniques (Beaulah Soundarabai et al., 2012; Kumar & Kumar, 2019; Zomaya & Teh, 2001). Static techniques are typically used to distribute predictable load requests. The use of static techniques does not require foreknowledge of the current state of the system. Dynamic techniques are used to distribute unpredictable load requests. This technique considers the current state of the system before distributing loads.

Based on the two categories of load-balancing techniques, researchers have proposed different load-balancing algorithms that are aimed at improving workload distribution of cloud-deployed applications. Static load balancing techniques include random allocation, threshold-based allocation, round-robin, and central manager allocation (Beaulah Soundarabai et al., 2012). They are well-established techniques but are used sparingly because of their limited applicability.

Shafiq *et al.* in (Shafiq et al., 2021) proposed a dynamic load balancing algorithm for allocating resources on the IaaS cloud model and addresses the VM isolation issue in the cloud through efficient task scheduling procedures. Their proposed algorithm focuses on addressing the priority of VMs, Quality of Service (QOS) task parameters, and resource allocation. The proposed algorithm improves load-balancing of VMs across IaaS by considering the priority of workload, SLA, and key metrics which determine performance. Results show that their algorithm resulted in an average of 78% resource utilisation compared to the existing dynamic load balancing algorithm with lower resource utilization rate.

Chen *et al.* in (S.-L. Chen et al., 2017) proposed a load-balancing architecture and a dynamic load-balancing algorithm for cloud services. The load-balancing algorithm can be used for both virtual web servers and physical servers. Their technique used a dynamic annexed balance method to solve the problem of uneven workload distribution on servers. Their approach considered both server processing power and compute 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 (Wang & Casale, 2014) 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 routing policies, they presented and verified algorithms that support multi-class workload for applications with SLA across different users.

In the bid to improve existing work on dynamic load balancing for cloud applications, authors identified specific server metrics that commonly affect the performance of both cloud applications and the servers they are running on. These identified metrics include CPU, RAM, bandwidth, Buffer, and so on. Tychalas *et al.* (Tychalas & Karatza, 2020) 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 utilization of remote fast resources by 25%.

Similar to the research by Tychalas *et al.*, authors in (Devi & Uthariaraj, 2016; Sahu et al., 2013) proposed dynamic load algorithms that combined key server metrics in determining the weights of servers. They both considered specific server metrics such as CPU, RAM, and bandwidth in determining the dynamic weight of a particular physical server or VM. They highlighted that these metrics play important roles in determining the efficiency of a load balancer. In addition, 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.

Besides research that focuses on developing dynamic load balancers, researchers have also focused on incorporating techniques to improve load balancing imitations such as single point of failure (Cruz et al., 2019; Grozev & Buyya, 2014b; Kumar & Kumar, 2019), scalability, limitation in sensing uncertainties (H. Zhang et al.,

2017), excessive overhead, and re-routing (H. Zhang et al., 2017).

Zhang *et al.* in (H. Zhang et al., 2017) in a bid to reduce load balancing limitations introduced Hermes. Hermes is a data center 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 20% improved performance compared to similar implementations.

Cruz *et al.* in (Cruz et al., 2019) identified a major challenge in determining how to optimise 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 (Grozev & Buyya, 2014b) 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 optimise 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 QoE with acceptable sacrifice in cost and latency.

After analysing the above literature on dynamic load balancing for cloud-hosted applications, this research study identified a few gaps. One of the gaps is the

need for load balancing algorithms and architecture that reactively and proactively handle uncertainties such as traffic dynamics and resource failures. While some of the literature addressed these issues, more research is needed to develop load balancing algorithms that can handle these challenges in a scalable and efficient manner. Also, existing research focuses on popular metrics such as CPU, RAM, and bandwidth, while ignoring other metrics and important factors such as thread count, reliability, and fault tolerance. This research will consider these additional factors to improve the overall performance and efficiency of cloud-hosted applications.

A dynamic load-balancing algorithm based on carefully selected server metrics specific to the chosen class of application will be developed in this research. The proposed algorithm and architecture will improve limitations such as single point of failure, reduction of excessive re-routing by using HAproxy, and quick sensing of uncertainties using selected key server metrics. It will also mitigate the negative effect of flash crowds and resource failures. The proposed load-balancing algorithm will complement and work cooperatively with auto-scaling mechanisms in cloud data centres to achieve its objectives. While previous work focused on a few server metrics, this research will focus on server metrics that directly impact the chosen class of cloud-deployed applications and the algorithm can be extended to function in a multi-cloud environment.

2.8.2 Test Environments for Load Balancing Algorithms

On one hand, dynamic load-balancing techniques are frequently being used to create load-balancing algorithms for cloud applications, and the results from these researches are promising. On the other hand, the use of cloud simulation to experimentally evaluate these cloud algorithms are increasingly used by researchers (Bambrik, 2020; Byrne et al., 2017; Fakhfakh et al., 2017; Grozev & Buyya, 2014b; Makaratzis et al., 2018; H. Zhang et al., 2017). Furthermore, evaluation of the above researches including these (Elgedawy, 2015; Hellemans et al., 2019; Zomaya & Teh, 2001) were all done using simulation tools and environment. As well, there is little research (Z. Chen et al., 2018) on evaluating cloud computing

algorithms by completely using real cloud infrastructures. Commonly, a simplified real infrastructure experiment is done to complement simulated experiments. This is recorded in (Grozev & Buyya, 2014b; Wang & Casale, 2014; H. Zhang et al., 2017). 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 use models to represent and experiment with cloud characteristics and behaviours.

On the contrary, cloud simulation might not be cheaper in terms of the realism of the results that are produced. 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 (Bambrik, 2020). 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 which results 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 (Bambrik, 2020; Fakhfakh et al., 2017). 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 (Byrne et al., 2017). In summary, simulation experiments rely heavily on parameters for accuracy, so the challenge remains of how to choose an accurate parameter. As a result, if the parameters are not right, an incorrect simulation result is inevitable.

Because of the stated disadvantages of using cloud simulation tools, this research will use real cloud infrastructure to experiment with the proposed algorithm. This research has access to a private cloud and the researchers are conversant with using the cloud infrastructure, so this prevents the challenge of time and cost. Running experiments on real cloud infrastructure is encouraged because it presents the real behaviours of resources used and therefore produces realistic results.

2.8.3 Workload Distribution and Resource Management in Multi-Cloud Deployment

On one hand, the deployment of cloud applications on single cloud is popular and cloud data centres are fast becoming the preferred deployment environment. On the other hand, the use of single cloud poses many challenges such as vendor lock-in, availability issues, less competitive cloud offerings, legislation compliance issues, and so on (Grozev & Buyya, 2014b; J. Zhao et al., 2020). Multi-cloud avoids over-provisioning of resources, vendor lock-in, unavailability, and customisation issues. Mainly because of these stated advantages, multi-cloud deployment has become increasingly popular (Grozev & Buyya, 2014b). If properly implemented, the multi-cloud deployment model makes it a good fit for overcoming performance degradation because of flash crowds and resource failure. Therefore, the deployment of web applications across multiple clouds is also one of the ways to mitigate the challenges of hosting applications on a single cloud.

Multi-cloud open-source libraries for different languages such as JClouds, Apache-LibCloud, SimpleCloud, and Nuvem (Cloud, 2017; A. S. Foundation, n.d.; T. A. S. Foundation, 2019, 2021; JClouds, 2017) are some research efforts towards the successful deployment of applications across multiple clouds. These libraries provide unified API for the management of cloud resources so that software developers do not have to write codes for specific cloud vendor's APIs. These libraries can be instrumental in developing new cross-cloud brokering components. However, these libraries do not provide application brokering and resource management utilities. Aside from multi-cloud libraries, services are alternative options. Services such as Scalr and Kaavo (ProgrammableWeb, 2011-2019; Scalr, 2011-2021) provide unified interfaces, APIs, and tools for managing multiple clouds. However, it is the user's responsibility to implement an appropriate resource management and scheduling approach.

Apart from multi-cloud libraries and services, some research projects such as MODAClouds (Ardagna et al., 2012), mOSAIC (Petcu et al., 2011) and STRATOS (Pawluk et al., 2012) also facilitate multi-cloud application deployment. However,

these projects do not allow for the inclusion of geographical locations of cloud data centres. Thus, it is often not possible to implement legislation-aware application brokering during workload distribution. In addition, multi-cloud libraries, services, and projects tend to manage resource allocation and software component deployment. They do not facilitate workload distribution among cloud resources.

Aside from the approaches to deploy applications across multi-cloud, the distribution of workload across multiple cloud is another area of great importance. Workload distribution across multi-cloud requires the use of proven and reliable load distribution techniques. There have been various pieces of research aimed at distributing workloads ranging from popular cloud services to bespoke research services: cloud services such as Amazon Web Service (AWS) Route 53 (Amazon, 2021a), and AWS Elastic Load Balancer (ELB) (Amazon, 2021c); Azure load balancer (Azure, 2021b) and Azure autoscale; overload management (Qu et al., 2017), and geographical load balancing (Grozev & Buyya, 2014b).

The commonly adopted industry-based and quickest approach is to use a load balancer to manage workload, especially when there is an overload caused by request spikes (flash crowds) or resource failure. This process is termed autoscaling and relies on dynamic provisioning and de-provisioning of resources to mitigate resource scarcity or complete resource failure, leading to service disruption or complete resource failure. Cloud services such as ELB load balancer (Amazon, 2021c) can distribute requests to servers in single or multiple data centres using standard load balancing techniques and to a set threshold. However, this service can only distribute incoming requests to AWS regions and not to third-party data centres. Likewise, Azure load balancer (Azure, 2021b) and autoscale (Azure, 2021a) can distribute incoming user requests among servers and data centres owned by Azure alone. These approaches focus on predicting future workloads and provisioning enough resources in advance to accommodate increased workload. The downside of these approaches is that they eventually over-provision resources in most cases (Qu et al., 2016; Qu et al., 2017).

Research approaches such as those found in (de Paula Junior et al., 2015; Gandhi

et al., 2014) reactively provision resources after they detect increased incoming requests or when a set threshold has been met. Furthermore, a similar approach (Qu et al., 2016) proposed the use of spot instances and over-provision of application instances to combat terminations of spot instances and improve workload distribution. However, because resource failures and flash crowds are often unpredictable, it takes the auto-scaler considerable time to provision new resources. Also, it is even more difficult to consistently and evenly distribute the load, regardless of an overload or resource failure. Therefore, I argue that it is beneficial to support and improve an auto-scaler to be able to handle situations such as overload and resource failure more effectively. The approach used in this research can bridge this gap for multi-cloud web-based three-tier applications by enhancing the role of an auto-scaler.

Researchers (Javadi et al., 2012; Niu et al., 2015) have also used the concept of cloud burst (Ali-Eldin et al., 2014); "the ability to dynamically provision cloud resources to accelerate execution or handle flash crowds when a local facility is saturated," to combat overload and manage increasing user requests. Unlike these approaches, this research does not just focus on provisioning and de-provisioning but rather aims to consistently distribute the workload of cloud deployed web-based three-tier applications across multi-cloud using a dynamic-hybrid load balancing technique.

Divisible load theory (DLT) has also been used by researchers for scheduling large datasets in distributed systems, including cloud computing. DLT is a paradigm for load scheduling in distributed systems. This theory divides large computational tasks into manageable and smaller pieces that can be distributed across multiple processors. Taking into account system dependency constraints like communication delays and processor characteristics, it exploits data parallelism in computational loads. This approach improves the utilization of computational resources and enhances the efficiency of job execution in cloud environments. (Abdullah & Othman, 2013; Bharadwaj et al., 2003; Kazemi et al., 2023). However, because DLT is more suited for load that are arbitrarily divisible and contains huge data, this theory does not fit into E-commerce dataset.

Grozev *et al.* (Grozev & Buyya, 2014b) proposed adaptive, geographical, dynamic, and reactive resource provisioning and load distribution algorithms to improve response delays without violating legislative and regulatory requirements. This approach dispatches users to cloud data centres using the concept of an entry point of an application framework and a centralised solution. The approach of this research study is different from Grozev's (Grozev & Buyya, 2014b) research approach because it uses a decentralised architecture, an improved load distribution algorithm, and a real multi-cloud test-bed for the experiments. Also, Grozev's research approach is reactive, which may not timely intervene when there are challenges. On the contrary, the approach used in this research is both reactive and proactive. This novel approach improves the timely intervention of flash crowds and resource failures in cloud-hosted applications and gives better control of the experimental environment.

Amazon 53 (Amazon, 2021a) connects user requests to infrastructure running in AWS—such as Amazon EC2 instances, ELB load balancers, or Amazon S3 buckets—and can also be used to route users to infrastructure outside of AWS. However, Amazon 53 uses Amazon Route 53 (Amazon, 2021a) a DNS resolution system to implement geographic load balancing, but this process makes it impossible to react timely to overload situations because of the time taken to populate DNS settings across layered DNS servers.

Qu and Calherios (Qu et al., 2017) adopts a decentralised architecture composed of individual load balancing agents to handle overloads that occur within a data centre by distributing excess incoming requests to cloud data centres with unused capacities. Their approach is composed of individual load balancing agents that communicates using the broadcast protocol to balance extra load. They aimed to complement the role of an auto-scaler, reduce over-provisioning in data centres, and detect short-term overload situations caused by flash crowds and resource failure through the use of geographical load balancing and admission control so that performance degradation is minimized. The approach used in this research to implement workload distribution, resource management, and deployment issues using the multi-cloud deployment pattern is different from the approach used by

Qu *et al.* (Qu et al., 2017). Although a decentralised architecture will be adopted as implemented by (Qu et al., 2017), load balancing agents will not be used so that network broadcast will be limited. This is because a limit to the amount of network broadcast is a significant improvement to previous research studies.

The proposed novel algorithm and deployment technique will complement and improve the role of auto-scalers for three-tier web-based applications deployed across multi-cloud. The monitor-analyse-plan-execute loop architecture often used by cloud-based systems (Qu et al., 2017) will be used to monitor server functions and algorithm activation. Furthermore, the solution employs a peer-to-peer client-server communication protocol to avoid the overhead incurred by the broadcast protocol used in similar research (Qu et al., 2017). In addition, the proposed solution's framework exemplifies a highly available cloud deployment architecture and will be experimentally evaluated on a test-bed that consists of three heterogeneous cloud data centres. Response times and throughput will be used predominantly as metrics to evaluate performance.

The review of relevant cloud application studies identified some gaps, which this current research seeks to fill. One of such gap is the lack of attention given to the description of cloud design patterns. As cloud computing becomes increasingly popular, robust and reliable cloud applications are required to deliver business solutions. Thus, cloud-native software developers and architects need to be able to effectively use cloud design patterns in building cloud-native applications. This will ensure that these applications can withstand the challenges of cloud-hosted applications. This research will fill this gap by creating an enhanced cloud-design patterns structure, applying the enhanced pattern structure to multi-tenancy patterns, and experimentally evaluating a novel implementation of multi-tenancy patterns.

Arguably, the most important gap identified is the lack of focus on comprehensive server metrics and factors of load balancing the workload of cloud-hosted three-tier web applications when flash crowds and resource failures occur. In this research, novel load balancing algorithms and architectures will be created to mitigate the adverse effects of flash crowds and resource failures amidst common

challenges of cloud-hosted applications in single- and multi-cloud. Furthermore, the novel algorithms and architecture that will be created can handle multi-cloud environments that satisfy key non-functional requirements such as availability, and regulatory compliance with improved application performance.

Simulations have also been used extensively in the experimental evaluation of cloud algorithms. However, simulation results can be wrong due to inaccurate parameterisation. This research fills this gap by evaluating proposed cloud algorithms and benchmarking them using real cloud infrastructures.

2.9 Summary

This chapter reviewed relevant literature that is related to this research study. As a result of the reviewed literature, gaps in the process of deploying and hosting web applications in the cloud were revealed. Research shows that design patterns are the foundation of a well-developed software application. An application will perform as it should if it was properly developed. This research posits that a successful application deployment is hinged on a successfully developed application. Currently, there are no consistent structures for describing cloud patterns and designing cloud-native applications. This research will create an enhanced cloud design pattern by adopting and improving the pattern structure from a widely accepted pattern catalogue on OOP. This step will improve the clarity of implementing cloud design patterns when building cloud-native applications and thus form a step in solving the challenges of deploying and hosting web applications on cloud.

After deploying an application on cloud, research shows that performance degradation in cloud-hosted web applications is commonly due to flash crowds and resource failures. A review of the literature revealed several methodologies that researchers have explored in providing solutions which include migration process, methods, tools, brokering methods and resource distribution techniques. Despite a number of researches in this area, there is still a lack of bespoke distribution techniques for

web applications deployed on cloud. This research will create novel techniques and approaches that include load balancing algorithms and deployment architecture and patterns. The novel cloud algorithms and deployment strategies will alleviate the effects of recurring problems associated with cloud-hosted applications.

Chapter 3

Research Design

3.1 Introduction

This chapter describes the experimental research approach adopted in this research. This approach addresses the research question in Chapter 1 by applying the quantitative experiments research strategy to all phases of the research. In this chapter, the research process model, the research strategy, the experimental setup, the data collection methods, the evaluation measures, and the data analysis methods used to evaluate the proposed cloud algorithms and techniques are discussed.

The organisation of the chapter is as follows: Section 3.2 describes the research process that was used to achieve the research goal. Section 3.3 describes the case study applications that were used to implement the proposed solutions. Section 3.4 describes the experimental setup for evaluating the proposed cloud algorithms and techniques. Section 3.5 summarises the chapter and discusses the motivation behind the adoption of the research approach.

3.2 Research Process Model

The research process model combined three stages: an exploratory study, selection and modification of the case study applications, and experimental evaluations as depicted in Figure 3.1.

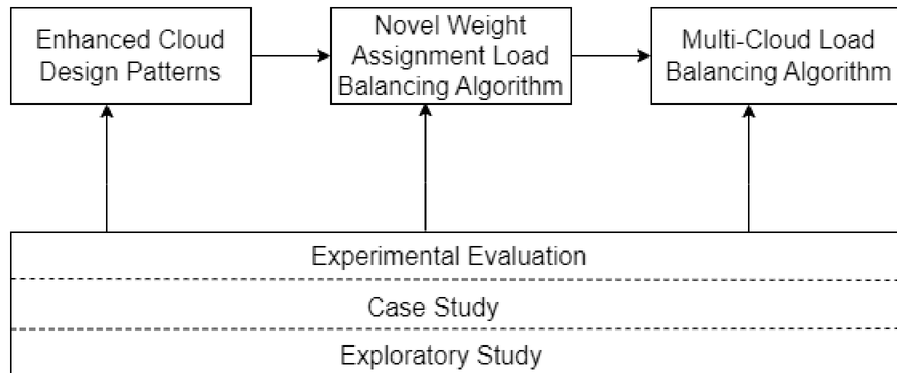


Figure 3.1: Research Process Model.

The first stage of the research process was discussed in Chapter 2. This stage explored related research materials and outputs that focus on developing and hosting cloud-native web applications on cloud. It provided insight into the current state of the research topic. During this stage, the research gap and how the proposed approach will accomplish the overall aim were identified and discussed. The next section discusses the case study applications used in this research, the experimental evaluation method, and the evaluation environment for evaluating the proposed solutions.

3.3 Selection of Case Study Application

This section discusses the selected applications used in the different phases of the research.

3.3.1 Enhanced Cloud Design Pattern

A bespoke desktop business process modelling (BPM) application was evaluated to discover an effective cloud deployment approach. The BPM tool was later modified into a prototype using both cloud deployment patterns and SaaS principles to facilitate successful deployment to the cloud. Appropriate design principles such as maintainability, scalability, multi-tenancy, and interoperability (Bass et al., 2012) were utilised in building the prototype to allow for key features of a cloud-native application. In the prototype application, loose coupling, modularization, abstraction, and interface segregation were principles that were implemented via these patterns: Model-View-Controller (MVC), Service-Oriented Architecture (SOA), Representational State Transfer (REST), and multi-tenancy patterns (Adewojo, Bass, Allison, & Hui, 2015). MVC was used to decompose this system into three components: data storage, application server, and user interface component. SOA was used to present its data logic and application server component as a set of services. REST was employed to deliver these services to promote interoperability. The three degrees of multi-tenancy patterns were implemented in the data-tier of the web BPM prototype. The prototype used Amazon simpleDB NoSQL (Cloud, 2017) storage for its database system, and it was deployed on Amazon cloud.

To empirically evaluate the effect of multi-tenancy patterns, WordPress (WordPress, 2017) was reviewed and its data layer was modified to allow the implementation of the three degrees of multi-tenancy patterns. WordPress was chosen because it is an open-source distributed content management system that powers 27% of websites (WordPress, 2017). Furthermore, it is customisable and can be programmatically modified. The distributed form of WordPress was used to evaluate multi-tenancy patterns. Evaluating WordPress business logic revealed that the default data-tier pattern for a single instance of WordPress is the dedicated component pattern, while a hybrid of tenant-isolated and dedicated component patterns is the default data-tier pattern for the multi-site version of WordPress. In addition, to test the applicability of multi-tenancy patterns in different environments and to assess the possibility of migrating existing applications without modification to the cloud, Docker containers were utilised as deployment technologies for migrating to the

cloud. After WordPress was modified to include multi-tenancy patterns, three instances of the application, each representing an instance that implemented one of the multi-tenancy patterns, were bundled into three Docker containers and deployed on the cloud.

3.3.2 Novel Weight Assignment and Multi-Cloud Load Balancing Algorithm

An open-source multi-tenant E-Commerce application was used to test the proposed cloud load balancing algorithms. This application is a stateless E-commerce application deployed using the three-tier architecture style. The application is a mini version of EBay. The application was built on Orchard's core framework coupled with elastic search for its search functionalities (OrchardCore, 2022). Orchard's core framework is built upon the .NET core framework and C# language.

The E-commerce application consists of a data layer that runs a MySQL database loaded with similar products that can be found on eBay; a domain layer based on REST services and components that implement the buying and selling of products; and a web interface where users can search and buy products. The major tables contained in the database are UserTable, OrderTable, AddressTable, ProductTable, ProductCategoryTable, ProductOptionsTable, OptionsTable, OrderTable, and so on.

3.4 Experimental Research Approach

This section discusses the experimental environment preliminaries, environment setup and evaluations of the proposed solutions in this research.

3.4.1 Preliminaries

This section discusses common tools and evaluation metrics that will be used in this experiment.

Tools

1. **HAProxy** - High Availability Proxy is an open-source, reliable, superior performance, and most widely used software load balancer. It is fully extensible and provides advanced security (HAProxy, 2021). HAProxy was used in collaboration with the proposed load balancing solution to distribute the workload of the chosen case study application.

2. **Glances** - This is a cross-platform monitoring tool written in Python. It collects and exports system metrics to chosen storage (Hennion, 2021). Real-time monitoring can also be done via its restful API. Glances was installed via the Ubuntu command that pulls the installation package from the Linux distribution. Glances was used to collect statistics of CPU, Memory, Network Buffer, Network Bandwidth, and thread count from all the application server VMs.

3. **Apache Web Server** - This is an open-source and robust HTTP server for modern operating systems including UNIX and Windows (Wikipedia, 2017).

4. **Python** - This is a high-level general-purpose programming language that supports multiple programming paradigms such as structured, object-oriented and functional programming.

5. **MySQL** - This is an open-source relational database management system that is based on SQL (Siteground, 2004-2017).

6. **WordPress** - This is an open-source content management system that is commonly paired with MySQL or MariaDB database. It emphasises accessibility, performance, security, and ease of use (WordPress, 2017).

7. **Docker** - This is a platform-as-a-service product that use OS-level virtualization to deliver to deliver software in packages called containers (Docker, 2017)

8. **stress-ng** - This is a tool that can stress test a computer system in various selectable ways.

9. **InfluxDB** - InfluxDB is a cross-platform, open-source, and scalable time series database for metrics, events, and real-time analysis (Inc, 2021). InfluxDB was connected to Glances clients and the server to collect data exported from Glances clients from all application server VM.

10. **Grafana** - Grafana is a cross-platform data analysis tool (Labs, 2021). It uses a dashboard to display time-series analysis data. Grafana was used to visually examine the data that came from the application server VM during experiments.

11. **Apache Jmeter** - This is a cross-platform open-source software that is designed to load test functional behaviour and measure performance of web applications and other resources such as a database (A. S. Foundation, 1999-2017). This tool was used to simulate user requests to the deployed case study application on the experimental testbeds.

12. **PuTTY** - This is a software application that enables the connection to Linux servers. It supports different network connection protocols to enable remote connection to Linux servers from a windows machine (Techopedia, 2021). This tool was used to create SSH tunnels to VMs on the cloud.

13. **mRemoteNG** - This is an open-source multi-protocol remote connection manager for Windows operating system (mRemoteNG, 2021). It works in collaboration with PuTTY to enable a successful connection to remote servers. This tool was used to manage connections to the VM on the cloud infrastructure.

Evaluation metrics

1. **Response Time** - The time elapsed to resolve an HTTP request.
2. **Throughput** - This is the amount of HHTP requests that can be processed within a given time frame. This parameter imitates the capacity of a server (Alankar et al., 2020). It is usually measured per second.
3. **Number of Errors** - The number of errors encountered during the processing of HTTP requests.
4. **Number of Failed Requests** - The number of failed HTTP requests.
5. **Round Trip Time (RTT)** - This is the time it takes for a network request to go from a starting point to a destination and back again to the starting point. Measurements are taken in milliseconds (ms). This is also known as ***latency***.
6. **Fault Tolerance** - The capability of the load balancing algorithm to continue to function in the presence of faulty cloud components.
7. **Scalability** - The capability of the algorithm to adjust its capacity to function when the situation changes.

3.4.2 Enhanced Cloud Design Pattern

The goal of this experiment was to evaluate the performance of the three multi-tenancy patterns in a dockerised WordPress application. Three separate docker images of WordPress, each image representing a WordPress instance with one of the enhanced multi-tenancy patterns implemented in it, were created. Three VMs with the configurations of 500GB HDD, 3GB memory, and Ubuntu 16.04 LTS operating system were created in a private OpenStack cloud. The following tools were installed on each of the VMs: Apache Jmeter 3.2, docker 1.8.0, Apache web server, docker image of WordPress 4.8.2, and MySQL 5.7.21.

Jmeter scripts were used to load test WordPress. These scripts contained Jmeter samplers and parameters that translate to web pages, actions performed, number of users performing the actions, and how users perform various actions in WordPress. Three scripts representing a load test script for each multi-tenancy pattern were created. The script simulates tenants performing the business process of blog-posts creation in WordPress. Tenants were grouped into sets of twenty users and the scripts increased the number of users progressively till it reached 200. Each request from a user sends data of not less than 5kb at once. This data contained the content of a blog post such as texts, tables, pictures, and links.

The experimental setup is illustrated in Figure 3.2.

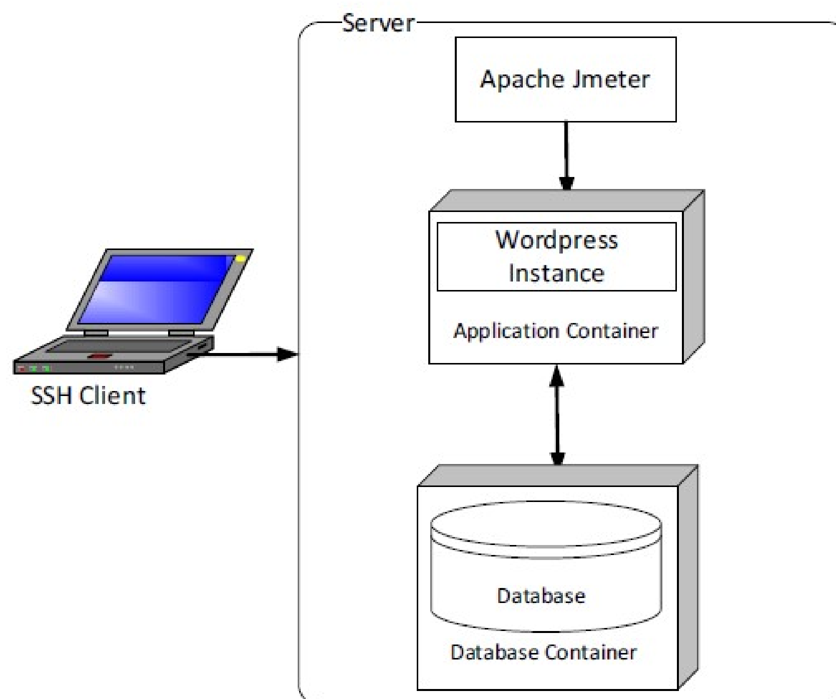


Figure 3.2: Experimental Architecture for Evaluation.

Evaluation Metrics

The evaluation metrics used in this experiment were average response times and number of errors.

3.4.3 Novel Weight Assignment Load Balancing Algorithm

In this experiment, the proposed novel weight assignment load balancing algorithm for three-tier web applications deployed on cloud will be evaluated. The proposed architectural brokering component will be designed for deploying three-tier web applications on cloud and benchmark algorithms will be compared to the proposed algorithm.

An OpenStack private cloud was used to evaluate the proposed novel weight assignment load balancing algorithm. There were seventeen heterogeneous VMs running Ubuntu 20.04 LTS in the experimental testbed, and their capacities are listed in Table 3.1.

Table 3.1: Capacity of VM used in Experiments.

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

Figure 3.3 illustrates the experimental setup on the private OpenStack. As a result of the OpenStack servers' network configuration, it is not possible to directly access the OpenStack dashboard from the Internet outwith the campus. Therefore, SSH tunnelling through PUTTY and mRemoteng was used for SSH requests. Port 8000 was the dedicated port used on the machine to access OpenStack.

The proposed novel load balancing algorithm was written in C# language. When deployed, the algorithm runs as a service. HAProxy server v2.4.2-1 and the novel load balancing algorithm solution were installed on two large-sized VMs after the algorithm was successfully developed. For the load balancing algorithm, the

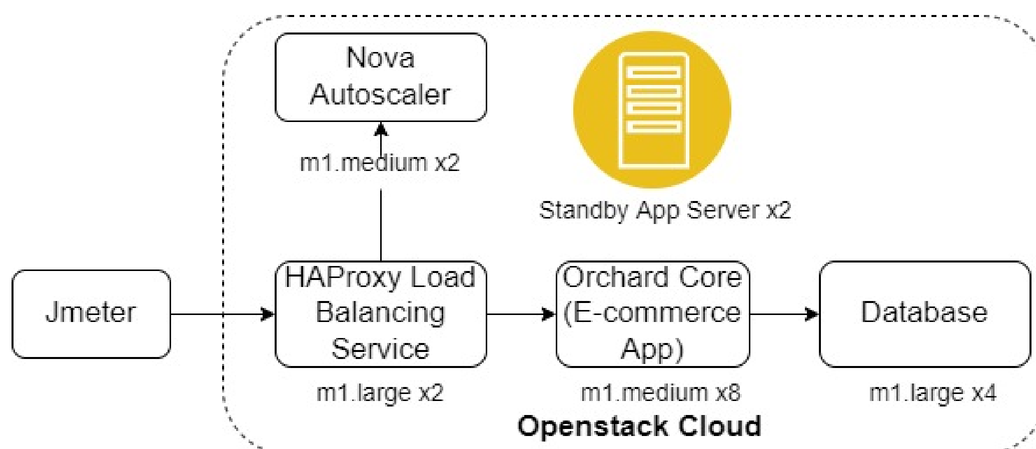


Figure 3.3: Single OpenStack Cloud Experimental Test-Bed.

HAProxy dataplane API v2.3.3 and statistics report for HAProxy provided load balancing statistics input.

A master application server VM was also configured with a medium size VM capacity. Apache web server v2.4.29, Glances v3.1.7 with PsUtil v5.8.0, InfluxDB v1.8.4 client, Python v3.6.9, Microsoft.NET Core 2.0/3.0 modules (runtimes and SDK), and stress-ng were all installed on the master application server VM. Apache web server and Glances were configured to start as services. Port 80 was configured as the default virtual host port for Apache web server. After the successful installation of the previous tools, the case study application was configured to run as a service on the application server. It can be accessed using the server's IP address and port 80. A snapshot instance of the master application server VM was created and replicated into ten medium-sized VMs. Two of the application server VMs served as standbys. MySQL v14.14 Distrib 5.7.39 database was configured on four large VMs. Each database server hosts a database that supports the deployed case study application. Two servers are a replication of each other, while the other two database servers were independent database servers for the purpose of testing for legislative rules and security.

A dedicated VM for collecting and analysing server data was also set up. InfluxDB v1.8.4 server, Grafana v7.0.4, and Glances v3.1.7 server were installed on the

statistics collection VM. To improve the performance statistics retrieved from Glances, a bespoke python script to retrieve network output such as bandwidth (Rx, and Tx), Network buffer, and thread count using netstat command was created. The bespoke Python script was configured as an add-in to work with Glances. Glances server co-ordinates all Glances clients running on each application server VM. Data retrieved by Glances server is stored in InfluxDB for future retrieval and manipulation. Grafana is used to monitor, visualize and analyse data retrieved from both InfluxDB and Glances server. Additionally, Microsoft Excel and Python scripts were used to analyse and visualise InfluxDB data.

Apache Jmeter v5.4.1 was installed on a standalone machine and configured to send HTTP requests to the deployed case study application. A backend listener that implements the InfluxDB datastore was included in the script setup. The backend listener is used to save throughput, sample size, deviation, and other necessary data into InfluxDB. This allows for a richer and comprehensive data collection of the same metrics from different sources. In addition, stress-ng was used to simulate CPU workload on the application server VM.

Workload

To simulate user requests for the case study application, Apache Jmeter was used. Apache Jmeter was hosted on a stand-alone machine in order to simulate realistic requests based on time and location. Apache Jmeter was used to generate the workload by simulating loads of various scenarios using a predetermined model by Chelbus *et al.* (Chlebus & Brazier, 2007) and a process described below. Specifically, the experiments aim to characterise the prominent performance metrics for load balancing identified by these researchers (S.-L. Chen et al., 2017; Z. Chen et al., 2018; Govind & González–Vélez, 2021; Grozev & Buyya, 2014b; Qu et al., 2017). Also, it compares the performance of the new load-balancing algorithm proposed with that of Grozev *et al.* (Grozev & Buyya, 2014b) and the round-robin algorithm.

The first step was to profile an application instance to determine how many requests can be processed in one second. The application server was then profiled again to determine how many requests can be handled with an SLA constraint that states that 90% of requests must be handled within one second, as recommended by (Qu et al., 2017). Based on a combination of the two profiles, the application server could process between 90 and 92 requests per second. Through this profiling, a workload emulating a nonstationary Poisson distribution was created. Poisson distributions are chosen because the nature of typical user requests follows a Poisson distribution. This means that each request is submitted independently and the occurrence of each request does not affect the probability that a second request will occur. Each of the experiments was repeated five times and an average of the results was used for evaluation purposes. The experiments were carried out over a period of 24 hours with an intensive workload during work hours and a lesser workload at other times. Therefore, the workload model starts with a minimum thread count of 300 users and increases to 2000 users with ramp-up times that correspond to the number of threads and a loop count of ten for each test execution.

Evaluation Metrics

Response time, throughput, scalability, latency, number of sessions, number of requests, and number of failed requests were the evaluation metrics used in this research.

3.4.4 Multi-Cloud Load Balancing Algorithm

In this experiment, the multi-cloud load balancing algorithm and decentralised architecture will be evaluated using the E-commerce case study application deployed across a heterogenous cloud data centre.

The experimental environment is composed of three heterogeneous data centres; a private cloud running OpenStack located in London, an Amazon Web Service

located in Tokyo: ap-northeast-1a and DigitalOcean cloud located in New York as depicted in Figure 3.4.

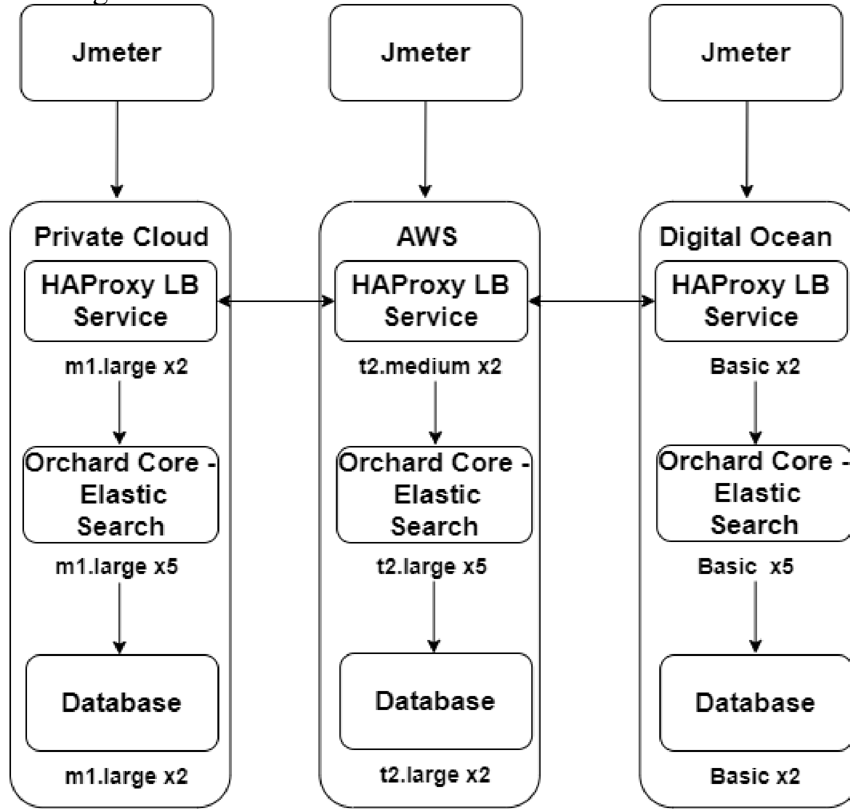


Figure 3.4: Multi-Cloud Experimental Test-bed.

There are nine heterogeneous VMs in each data centre. A VM is referred to as a droplet in DigitalOcean. The capacities of each VM in each data centre can be found in Figure 3.2. The Round-trip Time (RTT) latencies between the data centres were measured and recorded using the ping Linux command and are illustrated in Table 3.3.

Table 3.2: VM Capacity of Proposed Multi-Cloud Environment

	Private Cloud		**AWS**		**DigitalOcean**
	m1.medium	m1.large	t2.medium	t2.large	Basic
VCPUs	4	8	2	2	2
RAM	4GB	8GB	4GB	8GB	4GB
Disk Size	40GB	80GB	20GB	20GB	80GB

In each data centre, HAProxy server v2.4.2-1 was installed along with the proposed novel multi-cloud load balancing algorithm on two VMs. One VM acts as a

standby, depicting a high availability architecture. The second VM is in a standby state while the second virtual machine is used as the active server, dealing with all incoming requests in a normal state. On each data centre, a master application server is configured with the following applications: Apache web server v2.4.29, Glances v3.1.7 with PsUtil v5.8.0, Python v3.6.9, .NET Core 2.0/3.0 framework and SDK, stress-ng, and HAProxy Dataplane API v2.3.3 server. The master application server is replicated to create five application servers deployed in each data centre. Furthermore, a standard auto-scaler was deployed on a VM in each data centre. Two database server VMs that run MySQL v14.14 Distrib 5.7.39 were also configured in each data centre.

Table 3.3: Average Latencies between data centres in milliseconds

	Tokyo	New York
London	1.68	240.53

To collect and analyse data, the same configurations used in the single-cloud experiments were replicated for this experiment.

The new multi-cloud load balancing algorithm and architecture were tested by simulating flash crowds in each of the data centres. First, the novel algorithm was tested to ascertain that it distributes workload across the participating cloud data centres. Secondly, tests were carried out to verify that the algorithm respects the location of users by testing that it sends requests to the nearest data centre. Thirdly, experiments were carried out to check if the algorithm and HAProxy observe legislative compliance by testing that blacklists in HAProxy configurations were adhered to.

To test resource failures, some VMs were removed from the load balancer pool at 300ms time point over ten seconds and added back to the pool after five minutes to imitate recovery from failure. The length of 300ms over 10s was chosen because the launch time for an AWS server in an auto-scaling service is 300s (Amazon, 2000-2021; Qu et al., 2017). It allows the simulation of situations during which a commercial auto-scaler solely manages the application and demonstrates its results during overloads. The experiment was repeated for each of the data centres to simulate resource failure for each data centre. Furthermore, more experiments

were conducted to replicate where resource failures occurred in combinations of data centres.

Workload

To carry out a profiling test, E-commerce search requests were sent using Apache Jmeter to the cloud-deployed applications. Firstly, it was stipulated that an SLA requirement of 90% of requests should be replied to within one second. The SLA threshold of 90% is an experimentally fitted and approved threshold that guarantees low latency and consistent performance of E-commerce systems as shown in (Qu et al., 2017). Second, tests were performed to determine the average requests that each class of application server can handle without violating the SLA were performed. Workloads were created using the proposed workload model by (Bahga, Madisetti, et al., 2011). This workload model includes the following workload attributes: interval, think time, and session length because it forms a key part of determining the performance of an application when multiple users send requests. Research (Chlebus & Brazier, 2007; Qu et al., 2017) also shows that session arrivals and inter-arrival times constitute a Poisson process in which arrivals are independent and uniformly distributed.

Based on this workload model, three workloads were created for the three data centres using the parameters stated in Table 3.4. The experiments were carried out five times and an average number of requests that were handled by each data centre was recorded. On average, application servers handled 80 requests per second (private cloud), 65 requests per second (DigitalOcean), and 45 requests per second (AWS).

Table 3.4: Workload Parameter

	Mean	Min	Max	Deviation
ThinkTime (ms)	4000	100	20000	2
Intersession Interval	3000	100	15000	2
Session Length	10	5	50	2

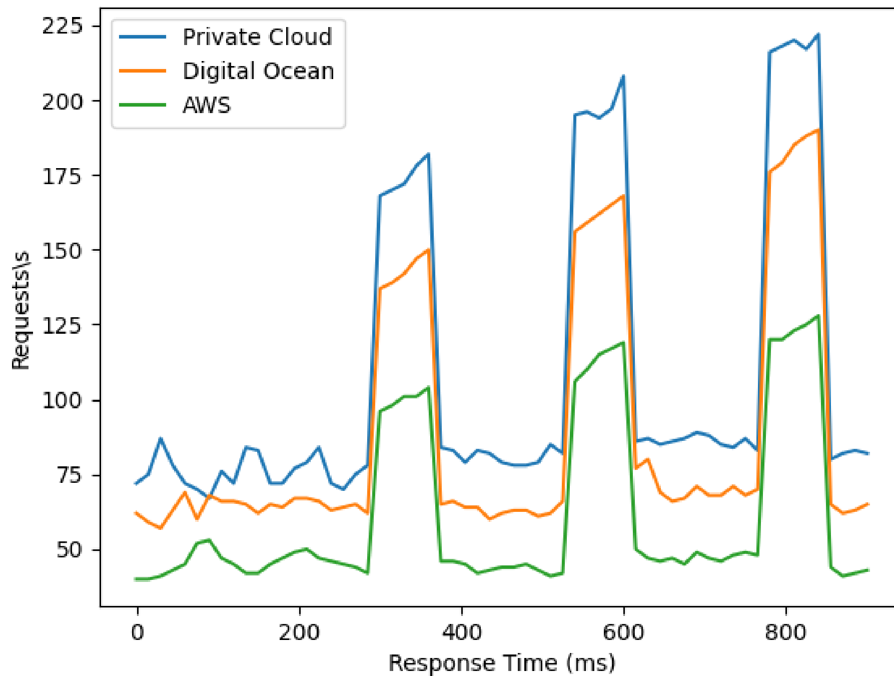


Figure 3.5: Workloads with Flash crowds range from 110% to 220%.

Evaluation Metrics

The evaluation metrics used in this experimental evaluation were response time, throughput, scalability, number of failed requests, and fault tolerance.

3.5 Summary

This chapter described the research approach, research process design, methodology, and strategy for collecting and evaluating data for this study. This research adopts the experimental research approach because the research encompasses different aspects of evaluating cloud deployment strategies. The research process design enabled the discovery and exploration of the underlying reasons behind

previous research done on deploying applications to the cloud. It also informed the use of mathematical techniques to test and evaluate the relationship among variables and how they affect the overall result of the research. Furthermore, the research process design informed the collection of data from software applications and helped to recommend suitable algorithms to improve the performance of cloud-deployed applications.

Using the research approach and design process, cloud migration methods were explored, case study applications were observed, and algorithms and models of cloud deployment were experimentally evaluated. This chapter explained how the proposed solution was achieved by describing the research activities for each phase of the research, tools employed, and metrics used for evaluating the proposed cloud algorithms and techniques. Therefore, the research approach contributed to achieving the research's aim and objectives.

Chapter 4

Enhanced Cloud Design Pattern Structure - Cloud Application Deployment and Architecture Pattern

4.1 Introduction

The preliminary stage of this research addressed design patterns in cloud-hosted applications. In this chapter, the structure of cloud design patterns by Fehling *et al.* (Fehling, Leymann, Retter, Schupeck, et al., 2014) was improved by mirroring a widely cited pattern catalogue by Gamma *et al.* (Gamma et al., 1994). The improved pattern structure was applied to multi-tenancy patterns. The enhanced multi-tenancy patterns showcased a formal description of the pattern, a UML description, and code snippets of its implementation in case study applications. A novel implementation of multi-tenancy patterns was done on two case study applications. These implementations were experimentally evaluated to test the applicability and performance of multi-tenancy patterns in the selected applications

and deployment environment.

Section 4.2 describes the enhanced structure of cloud design patterns. Appendix A contains a detailed description of the enhanced multi-tenancy patterns. Section 4.3 describes the enhancement applied to multi-tenancy patterns. Section 4.4 describes the implementation of multi-tenancy patterns in case-studied applications. The results of the experimental evaluation of the novel implementation of multi-tenancy patterns are discussed in Section 4.5. A chapter summary is done in section 4.6. The improved cloud design pattern structure, implementation, and the evaluation of multi-tenancy patterns are published in (Adewojo, Bass, & Allison, 2015; Adewojo, Bass, Allison, & Hui, 2015; Adewojo & Bass, 2018). This chapter represents the content of these papers in full.

4.2 Enhanced Cloud Design Pattern Structure

The popular cloud pattern catalogue by Fehling *et al.* (Fehling, Leymann, Retter, Schupeck, et al., 2014) forms the foundation of the enhanced cloud patterns in this chapter. The structure of the cloud pattern catalogue was compared to the structure of a widely accepted pattern catalogue structure by Gamma *et al.* (Gamma et al., 1994). The format, structure, and graphical notation were the main basis for the comparison. A comparison of the structure of the cloud pattern catalogue and the structure of the OOP pattern revealed gaps in the structure of the cloud patterns. These gaps represent vital information that will make the description of cloud patterns robust, clear, consistent, formalised, and applicable to various users.

To close the gaps in the cloud pattern catalogue structure, this research reviewed the literature and best practise in computing cataloguing patterns. As a result, an enhanced cloud pattern structure was created. The new structure now includes logical headings that clearly explain each pattern. To compare, contrast, and show the journey from the initial to the enhanced cloud design pattern structure, tables 4.1 - 4.4 which contain the pattern structures were utilised. This enhanced cloud

OOP Pattern	**Cloud Design pattern**
Pattern Name	Pattern Name
Intent	Intent
Alias	Icon
Applicability	Driving Question
Structure	Context
Participants	Solution
Collaborations	Result
Consequences	Variations (this does not apply to all patterns)
Implementation	Related Patterns
Sample code	Known Uses
Known uses	
Related patterns	

Table 4.1: Initial Structure of OOP and Cloud Design Patterns.

design pattern structure clears the ambiguities in the implementation of cloud design patterns. Also, it enhances and improves software developers' and the architects' ability to implement these patterns.

Cloud Design pattern	**Enhanced Cloud Design pattern**
Pattern Name	Pattern Name
Intent	Intent
Icon	Icon
Driving Question	Motivation
Context	Applicability
Solution	Consequences
Result	Participants
Variations	Collaborations
Related Patterns	Related Patterns
Known uses	Known Uses
	Sample code
	Formal Description

Table 4.2: Initial and Enhanced Cloud Design Pattern Structure

Beginning of Table			
OOP Pattern	**Description**	**Cloud Design pattern**	**Description**
Pattern Name	The name used to identify the pattern	Pattern Name.	The name used to identify the pattern.
Intent	This section describes the purpose and content of the solution represented in the pattern.	Intent	The purpose and goal of the pattern, and the content of solution is described.
Alias	This is another name for the pattern.	Icon	A graphical representation of the pattern to be used in architectural diagrams for modelling cloud applications.
Applicability	The situations to which the pattern applies.	Context	This section describes the environment and forces leading to the problem solved by the pattern. It also may describe why naive solutions can be suboptimal. It showcases the context in which other patterns that form implemented by developers can be applied to

Continuation of Table 4.1			
OOP Pattern	**Description**	**Cloud Design pattern**	**Description**
Participants	This comprise of the classes and objects that make up the pattern.	Result	This section elaborates the solution in greater detail. It discusses new challenges that may arise after implementation, references to other patterns that may address the challenge, and a proposed architecture.
Collaborations	This describes how classes and objects in a pattern, relate to each other.	Variations	Because patterns can be applied in slightly different forms, the section showcases patterns that are not significant enough for a separate pattern description
Consequences	This refers to the culmination of the results, trade-offs, and side effects caused by the use of the pattern.	Related Patterns	The interrelation of patterns that are often applied together or that should be excluded from each other are discussed in this section.
Implementation	This describes the solution part of the pattern.	Known Uses	Existing applications that implement the pattern are covered here.

Continuation of Table 4.1			
OOP Pattern	**Description**	**Cloud Design pattern**	**Description**
Sample code	This illustrates how the pattern can be implemented in a programming language.	Driving Question	A question that captures the problem that is answered by the pattern.
Known uses	These are examples of real usages of the pattern.		-
Related patterns	This refers to the similarities (as well as differences) that can exist between the pattern of interest and another pattern.		-

Table 4.3: OOP and Cloud Design Pattern Structure

In Table 4.1 the OOP and cloud design pattern headings are compared and Table 4.2 compares the initial and enhanced cloud design pattern structure. Table 4.3 explains the meaning of each heading of both OOP and cloud design pattern structure.

Table 4.4 displays just the enhanced cloud design pattern structure and the meaning of each section. The enhanced cloud design pattern structure now includes new headings: Motivation, Applicability, Participants, Collaborations, Consequences, Sample Code, and Formal Description.

Motivation extracts the commonalities between intent, driving questions, and context in the cloud design pattern structure by Fehling *et al.* (Fehling, Leymann, Retter, Schupeck, et al., 2014). This description simplifies the reason for the pattern being described.

Applicability is an extract from context and driving question by Fehling *et al.*

Enhanced Design pattern	**Description**
Pattern Name	This is a unique identifier to aid identification and reference to a pattern.
Intent	This describes the aim and reason for using the pattern.
Icon	A graphical representation of the pattern to be used in architectural diagrams for modelling cloud applications.
Motivation	An idealised situation where the challenge and context is suitable for a design pattern.
Applicability	A circumstance in which the pattern is suitable for use.
Structure	A diagrammatic illustration of how to design the pattern.
Implementation	This describes the solution part of the pattern.
Related Patterns	This refers to the similarities (as well as differences) that can exist between the pattern of interest and another pattern.
Known Uses	These are examples of real usages of the pattern.
Consequences	This refers to the culmination of the results, trade-offs, and side effects caused by the use of the pattern.
Sample code	This illustrates how the pattern can be implemented in a programming language.
Formal Description	A mathematically based technique to describe a pattern using chosen formal language.

Table 4.4: Description of Enhanced Cloud Design Pattern Structure

(Fehling, Leymann, Retter, Schupeck, et al., 2014). It clearly defines the suitability of the pattern for different circumstances that a software architect might encounter.

Structure improves Icon by Fehling *et al.* (Fehling, Leymann, Retter, Schupeck, et al., 2014). It diagrammatically represents the pattern and how it can be designed by using a general purpose modelling language.

Implementation improves and simplifies Solution by Fehling *et al.* (Fehling, Leymann, Retter, Schupeck, et al., 2014). It highlights how the pattern can be

implemented in different scenarios of building a cloud-native web application.

Consequences improve and simplifies Result by Fehling *et al.* (Fehling, Leymann, Retter, Schupeck, et al., 2014). Consequences describe the trade-offs and side effects that can be caused by using the pattern being described.

Sample code improves solution by Fehling *et al.* (Fehling, Leymann, Retter, Schupeck, et al., 2014). It illustrates using real programming code to describe how to implement the solution the pattern offers.

Formal description improves Icon by Fehling *et al.* (Fehling, Leymann, Retter, Schupeck, et al., 2014). It describes precisely the pattern through the use of formal language.

4.3 Enhanced Multi-tenancy Patterns

As discussed in Chapter 2, multi-tenancy patterns are key architectural patterns and one of the essential patterns for cloud properties. This is because resource sharing is an essential cloud property and multi-tenancy is one of the architectural patterns that describe resource sharing.

Multi-tenancy is also classified as a type of cloud deployment pattern (Adewojo & Bass, 2018; Fehling et al., 2011; Fehling, Leymann, Retter, Schupeck, et al., 2014). Cloud deployment patterns play a major role in architectural restructuring and migration of on-premise software applications to the cloud. This is the result of its ability to describe the methodology needed to provision and manage cloud resources that will host the application. Also, as stated in Chapter 2, multi-tenancy occurs at different levels of cloud: infrastructure, application, and database layers. Therefore, applying it to the development and deployment of cloud-native application can be a challenge. But a comprehensive design pattern can help to reduce the implementation challenge.

In this section, the enhanced cloud design pattern was applied to multi-tenancy patterns. This stage of research focuses on implementing multi-tenancy in the database layer of a cloud-native application. The full description of the enhanced multi-tenancy patterns can be found in Appendix A.

4.3.1 Formal Description of Shared Component Pattern

This research further enhanced the existing multi-tenancy pattern of Fehling *et al.* (Fehling, Leymann, Retter, Schupeck, et al., 2014), by formally describing the patterns using the Z language. The formal description defines precisely what a multi-tenancy pattern is and how to implement it because it exposes extra decisions that will be in the final artifact. This formal description will help solution architects to better understand how to implement multi-tenancy patterns consequent to leading to applications with improved resource-sharing ability.

In this research, UML diagrams of the multi-tenancy patterns were first created and then further specified using the Z language. Z language was used because it is a state-based language. Its domain of use is not limited, and it is fairly easy to understand.

Figure 4.1 is the formal definition of the shared component pattern in Z language. It describes the state space of this pattern and is referred to as a schema. This schema deals with variables such as database schema, names, and tables. The first part of the schema declares variables that constitute the pattern, while the part below the line gives a relationship between the values of the variables.

- known – is the set of database names and their collections
- SharedDBItem - is a function which when applied to the database (SysDB-Name) will return shared tables associated with the database
- SharedTableItem - is another function that returns tenant IDs and tenants' data when applied to a shared table item (SharedTableItem)

The part of the schema below the line says that the set known is the same as the domain of the function SharedDBItem and SharedTableItem - the set of databases and tables which it can validly contain. The relationship shows that a shared pattern contains multiple shared tables with tenants' details in them.

$$
\begin{array}{l}
\textit{SharedDB} \\
\hline
known : \mathbb{P}\, SysDBName \\
SharedDBItem : SysDBName \nrightarrow SharedTableName \\
SharedTableItem : SharedTableName \nrightarrow TenantID \\
\hline
known = \mathrm{dom}\, SharedDBItem \cup SharedTableItem
\end{array}
$$

Figure 4.1: Z Representation of Shared Component Pattern (Adewojo, Bass, & Allison, 2015).

4.3.2 Formal Description of Tenant-Isolated Component Pattern

Figure 4.3.2 formally describes the tenant-isolated component pattern in Z language. This description also includes schema, names, and tables.

- known – is the set of database names and their collections
- DBSchemaName - represents a dedicated schema that contains tenant specific table items that are all contained in the main database (SysDBName)
- TenantIsolatedTableItem - represents a collection of tenant specific tables within a schema

The lower part of the schema emphasizes the fact that the database validly contains database schemas with the above variables.

$$
\begin{array}{|l}
\hline
\textit{TenantIsolatedDB} \\
known : \mathbb{P}\, SysDBName \\
DBSchemaName : \\
\mathbb{P}\, TenantIsolatedTableItem \cup SysDBName \\
TenantIsolatedTableItem : \mathbb{P}\, TenantIsolatedTableName \\
\hline
known = \operatorname{dom} DBSchemaName \cup SysDBName \\
\hline
\end{array}
$$

Figure 4.2: Z Representation of Tenant-Isolated Component Pattern (Adewojo, Bass, & Allison, 2015).

4.3.3 Dedicated Component Pattern

Figure 4.3.3 is the Z representation of the dedicated component pattern. This schema deals with variables such as database names and tables. The first part of the schema declares variables that constitute the pattern.

- known – is the set of dedicated database names and their collections
- DedicatedDBItem - is a function which when applied to the database (SysDBName) will return dedicated tables associated with the database; this implementation has all database component dedicated to a database instance and tenant

The lower part of the schema reflects the domain of the function. This means that each database is dedicated solely to the tenant.

$$
\begin{array}{|l}
\hline
\textit{DedicatedDB} \\
known : \mathbb{P}\, SysDBName \\
DedicatedDBItem : SysDBName \mathrel{\mapsto\!\!\!\!+} DedicatedTableItem \\
\hline
known = \operatorname{dom} SysDBName \\
\hline
\end{array}
$$

Figure 4.3: Z Representation of Dedicated Component Pattern (Adewojo, Bass, & Allison, 2015).

4.4 Implementation of Multi-tenancy Patterns in Case Study Applications

To evaluate the applicability of the enhanced multi-tenancy patterns, an implementation of the patterns were carried out on the data and domain layer of two case-studied application.

4.4.1 Implementation of Shared Component Pattern in Case Studied BPM

The shared component pattern was used to design the database table that accommodates the user details of tenants that used the BPM software application. It was also used to improve resource sharing of the underlying infrastructure of the software application.

4.4.2 Implementation of Tenant Isolated Pattern in Case Studied BPM

The tenant-isolated component pattern was implemented in the data storage component of the case study application. To implement the tenant-isolated component, the data storage component was designed to allow multiple customers to access a single instance of SimpleDB with domain-id as the customer's identity. A domain represents a company, and everything in a domain represents details of all defined process for that particular company. Items in the domain are Task, Role, RoleType, Department and Process. This means different companies share the same simpleDB instance but are in different domains. It offers a high degree of sharing without influencing other tenants. This method of implementation ensures isolation between companies by controlling their access, processing performance used, and separation of stored data. Also, each tenant's user is authenticated before

gaining access to the stored data, and each domain is uniquely identified. Hence customers can only access their data. The code fragment in Appendix A–section **sample code**–depicts how this is implemented.

4.4.3 Implementation of Dedicated Component Pattern in Case Studied BPM

Each domain of the WordPress instance represents different parts of the company's process. This offers the lowest degree of sharing but the highest degree of privacy. However, it is costly in terms of finance and technical management. The code snippet in Appendix A, section **sample code** shows a fragment of how this was implemented.

4.4.4 Implementation of Multi-tenancy Patterns in Containerized Cloud Hosted WordPress Application

To implement shared component pattern, tenants shared the same database instance, schema, and tables of WordPress Application. Each tenant is identified using a variable called the tenant ID. This allows the database to group users with same tenant ID as belonging to the same tenant. To implement tenant-isolated component pattern, tenants shared the same database instance, while the database schema and tables were dedicated to each tenant. To implement dedicated component pattern, a dedicated database instance, including schema and tables were allocated to each tenant so that it can support legislative requirements of the application.

4.5 Results

An average of twenty runs for each group of user requests is used in this setup. Equation (4.1) depicts how the average response time is calculated for each tenant, and its input parameters are represented below:

- N—Number of runs;
- T_n—Number of tenants;
- E—Elapsed time;

$$AvgResponseTimeforNRuns = \frac{\sum_{N}^{1}(\frac{\Sigma(E)}{T_n})}{N} \tag{4.1}$$

Figure 4.4 depicts the average response time of creating blog posts in the three experimental setups of WordPress that implement the multi-tenancy pattern. The average response time to create a blog-post using the dedicated pattern implementation varied between 1.64 seconds and 5.43 seconds. The average response time for the tenant-isolated pattern varied between 2.05 and 8.5 seconds. The average response time for the shared component pattern ranged between 2.99 and 6.85 seconds.

The result in Figure 4.4 also showed that the average response times of shared and tenant-isolated pattern reached a peak at 6.85 seconds and 8.5 seconds respectively before it dropped. The drop in the response was a result of errors in the number of requests being handled by WordPress at the time. Figure 4.5 shows the number of failed requests that occurred while evaluating the multi-tenancy patterns in each of the experimental setups of WordPress. There was an increase in the number of failed requests when the average response time peaked for both tenant-isolated and shared patterns.

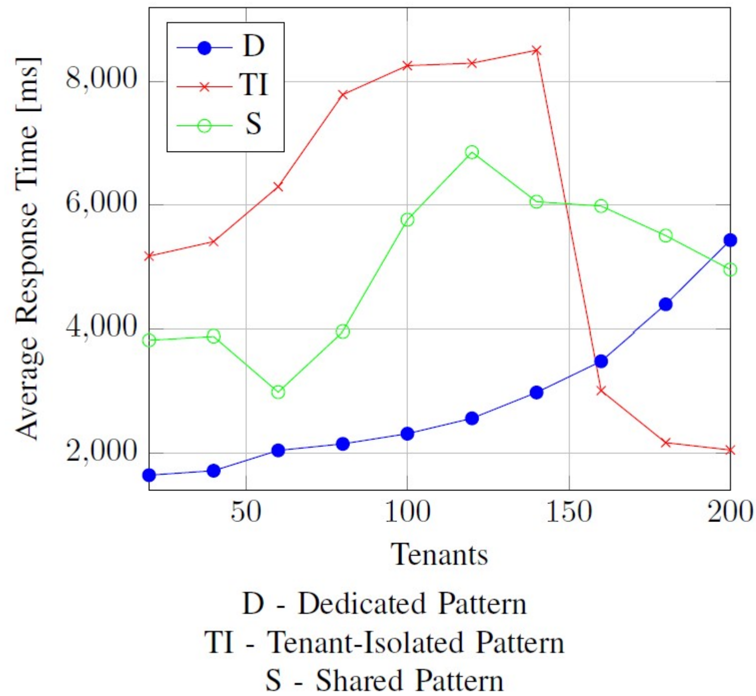


Figure 4.4: Average Response Time of the Multi-tenancy Patterns when creating Blog Posts.

Reflecting on the characteristic performance of multi-tenancy patterns in WordPress, it is concluded that dedicated component pattern suits WordPress applications used within an organisation where users are all grouped as a tenant. However, from another perspective, tenant-isolated pattern suits WordPress applications in two different instances: tenants are groups of users within an organisation or tenants are different groups of users that are not part of the same organisation. The tenant-isolated pattern approach of sharing resources will allow more groups of tenants to share the same WordPress instance with different customisation and database schema.

The shared pattern performed well in WordPress when tenants did not exceed 100 users. This number of users is relatively small and not realistic for most organisations. However, this pattern can be used to handle less critical data such as setting up user data of a SaaS application.

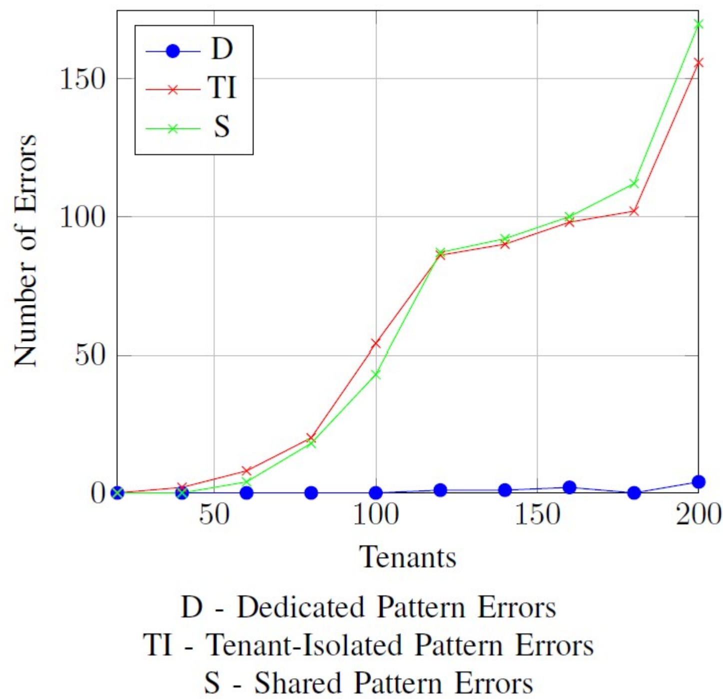


Figure 4.5: Number of Request Errors when creating Blog Posts.

4.6 Summary

Multi-tenancy is an important and frequently used architectural pattern in SaaS. It improves the utilisation of resources and reduces the effort in deploying and managing applications on the cloud. In addition, it reduces the total cost of ownership. Three degrees of multi-tenancy exist, and so three different types of its implementation exist, irrespective of the application or infrastructural layer that will implement it.

Choosing the right multi-tenancy pattern can be a challenge, so this chapter contributes to alleviating this challenge by enhancing the structure of cloud design pattern to include headings and descriptions that enhance the meaning of a cloud pattern. A formal description of the multi-tenancy pattern using the Z language was created. A novel data layer implementation of multi-tenancy patterns was developed in two case-studied applications. Furthermore, an empirical evaluation

of the three types of multi-tenancy patterns was performed. The experiments evaluated the performance of the three multi-tenancy patterns so that their suitability in a content management system can be recorded.

This chapter has contributed to research by systematically improving cloud design pattern structure and multi-tenancy patterns. The patterns were improved to include logical headings and elaborate definitions; a detailed description of how each pattern can be utilized was provided, and underlying theories of the pattern properties were explained. Furthermore, the enhanced pattern catalogue provides a common language that can help software developers to communicate using well-understood terminologies for software interactions. An enhanced pattern catalogue will help software developers to understand the difference between the cost and benefits of the different approaches of using multi-tenancy patterns.

The next chapter will focus on creating novel algorithm and techniques for mitigating the issues that cloud-hosted application encounter.

Chapter 5

A Novel Weight-Assignment Load Balancing Algorithm for Cloud Applications

5.1 Introduction

This chapter introduces and describes a novel weight assignment load balancing algorithm for three-tier web-based cloud applications. The content of this chapter has been previously published in (Adewojo & Bass, 2022b) and (Adewojo & Bass, 2023). The contributions of this chapter are as follows:

1. a design approach for deploying existing web applications to the cloud with minimal changes to the code base
2. a novel hybrid-dynamic load balancing algorithm and architecture that mitigates limitations of load balancing techniques such as single point of failure, excessive re-routing, and slow sensing of uncertainties
3. a load balancing algorithm that mitigates the negative effect of flash crowds

and resource failures

4. a redesigned architectural component for brokering activities within three-tier architecture for distributing workload and

5. an implementation and experimental evaluation of proposed and benchmark algorithms on a private cloud test-bed.

This chapter starts by describing the requirements and assumptions for the proposed novel load balancing algorithm. The proposed approach to the novel weight assignment and load balancing algorithm is discussed in section 5.3. The novel load balancing service and the weighting technique behind it is described in Section 5.4. An empirical experimental evaluation and results are presented in Section 5.5. Section 5.6 concludes and summarises the chapter.

5.2 Requirements and Assumptions of Proposed Load Balancing Algorithm

The proposed novel load balancing algorithm will establish a foundation for multi-cloud load distribution. Therefore, the solution will require requests that can be processed by application replicas deployed in more than one data centre. This involves some important factors, and session continuity is the most important factor for this research. Session continuity ensures that end-user sessions, established over any access network, will not lose connection or any internal state even when different servers process user requests. Stateless applications, such as search engines, do not save client data generated in one session for use in the next session with that client. Also, stateless applications can easily scale because they can be deployed across multiple servers without issues while ensuring session continuity. All these properties of stateless applications implicitly satisfy the requirement of session continuity. On the other hand, stateful applications require the persistence of end-user IP and session to a specific server. Consequently, stateful applications

and services cannot be managed with this approach. Hence, this study focuses on stateless applications.

It should be noted that a significant reason for deploying software layers across multiple tiers is to secure a balance among performance, scalability, fault tolerance, and security ("Rockford Lhotka - Should all apps be n-tier?", 2020). The presentation layer often executes at the users' end and is thus not the focus of this research work. The business and data layers of a three-tier web application executes in the backend server and can be scaled when they are deployed across different tiers. The data layer usually consists of one or more database servers. However, this layer typically becomes a performance choke point because of the requirements for transactional access and atomicity (Brewer, 2012; Grozev & Buyya, 2014b). Some techniques such as replication, caching, and sharding are recommended to ease the scaling of the data layer (Fowler, 2002; Grozev & Buyya, 2014b). These techniques are application specific, so will be impossible to include them in a generalised framework that targets any three-tier web applications. In sum, the proper balance of using the above technique is domain-specific. Therefore, this research does not cover application-specific data deployment. This study assumes that the data layer has been deployed appropriately and is focused on distributing workload effectively across the business tier or application server layer.

5.3 The Proposed Approach

The proposed approach to designing a hybrid-dynamic load-balancing algorithm using a novel weight-assignment policy is discussed in this section. The load balancer and the novel load balancing service are co-located in the same data centre to achieve fast detection of flash crowds and perform quick adaptations. The load balancing service comprises a monitoring module, the load balancer software, and the control module. To detect application or server overload, the monitoring module continuously monitors incoming requests and available resources. The control module contains the load balancing algorithm and other configuration files, which

are used to adjust the weight of each VM in response to requests. Furthermore, as discussed in section 5.2 about the focus of this research on stateless applications, a key principle behind the proposed approach is session continuity. Session continuity is an important factor in distributing requests across any application server. Session continuity is the principle behind the improved performance of stateless applications.

5.3.1 Load Balancing Architecture

Figure 5.1 is an illustration of the proposed high availability (HA) load balancing service. The load balancing algorithm (load balancer controller) is part of the load balancing service, along with a load balancer (HAProxy) and monitoring functionality. The load balancer controller returns the weight of all participating servers to the load balancer 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 design of the load balancing service is the message-based communication strategy. This strategy improves decoupling, flexibility, and maintainability (Archiveddocs, 2022a). 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 as the test web application. The co-location of these components helps to achieve fast detection of flash crowds and perform quick adaptations.

5.3.2 Overall Architecture

The overall architecture of the newly introduced load balancing system is depicted in Figure 5.2. The conventional three-tier web application was extended to include

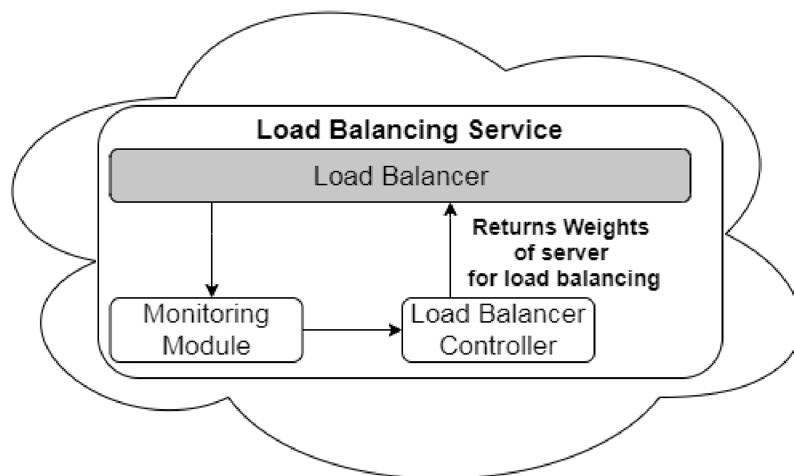


Figure 5.1: Proposed Load Balancing Architecture (Adewojo & Bass, 2022b)

an additional layer of components. This extra layer consists of the newly introduced load balancing service, HAProxy load balancer, and an auto-scaler. This architectural design approach facilitates a deployment approach for existing web applications with minimal changes to the code base.
In this architecture, users interact with the presentation layer, and requests are routed to an entry point consisting of VMs that host the novel load-balancing service. User requests are sent to the application server by the load-balancing service. These application servers host the domain layer on separate VMs. After interacting with the data layer, the domain layer sends back responses to the user based on the data it has manipulated. The data and domain layers consist of multiple servers installed across VMs. This process is repeated every time a request comes in, and the load-balancing service distributes the workload accordingly.

5.3.3 Design and Deployment Architecture

A load balancing algorithm that is efficient and tailored to the application it supports should incorporate different metrics relevant to the application (Kumar & Kumar, 2019). Therefore, the design of the proposed load balancing algorithm will address

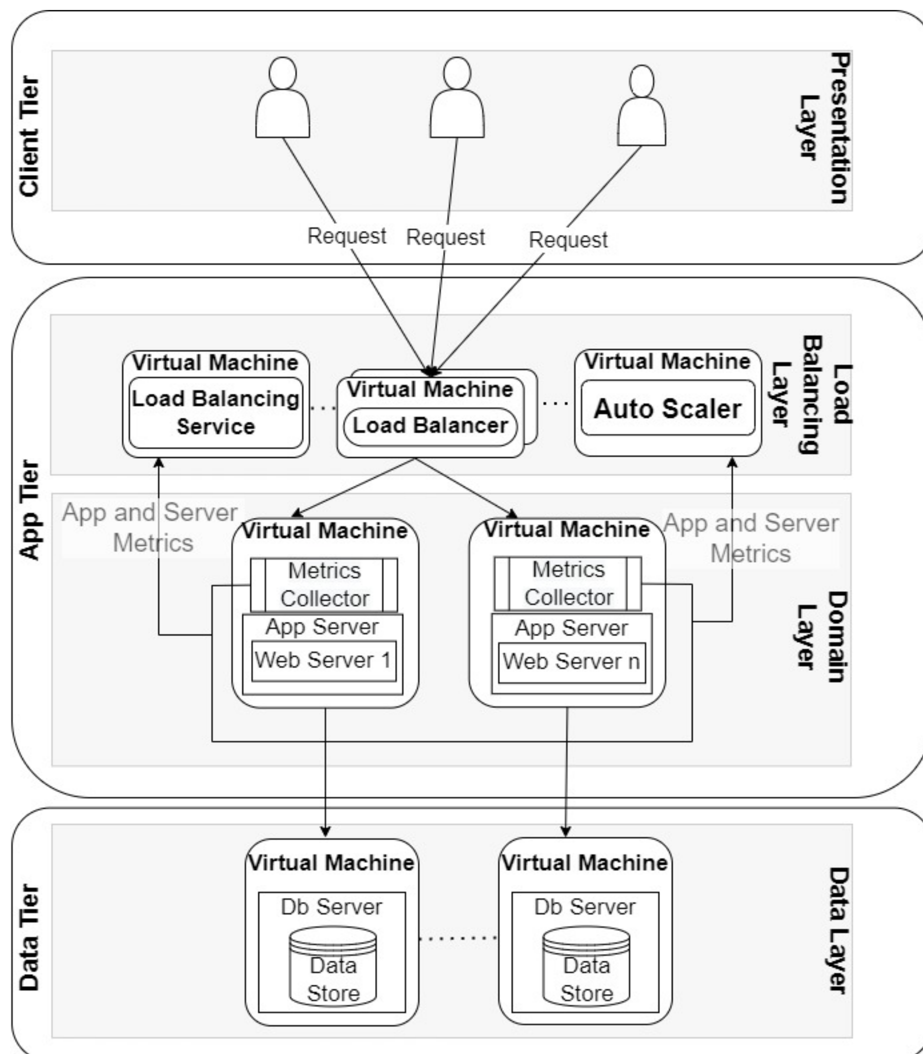


Figure 5.2: Proposed Overall Layered Architecture in a Single Data Center

issues and limitations of load balancing that were discussed in Chapter 2 by incorporating techniques and carefully selected metrics that are essential for load balancing an interactive three-tier web application. A number of these issues highlight areas where load balancing algorithms and architecture could be improved. The following sections discuss these issues and ways to mitigate them as reflected 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 (HAProxy, 2021). Furthermore, the use of HAProxy with the proposed novel algorithm mitigates the issues of slow sensing of uncertainties and excessive re-routing because HAproxy is a fast, proven and reliable load balancer software. HAproxy is also an open source and reliable reverse-proxy high-performance TCP/HTTP load balancer software (HAProxy, 2021). It is recommended for most websites and 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 (HAProxy, 2021). In addition, this research's experiments featured the removal and addition of a varied number of heterogeneous VMs to test scalability. The results proved the solution's ability to distribute load across a finite number of servers.

- Fault Tolerance: Cloud fault tolerance is the capability of the infrastructure to support uninterrupted functionality of deployed applications despite failures of components (Bala & Chana, 2012; "What is Fault Tolerance?", 2022). The ability of a load balancer to continue to deliver services despite failure (Shah et al., 2017) of a cloud component is highly desirable in guaranteeing the availability and reliability of cloud-deployed applications. Consequently, the cloud uses various fault tolerance approaches, including proactive approaches such as fault detection systems, reactive approaches such as VM placement models, and other approaches such as machine learning and meta-heuristics in algorithms (Kumari & Kaur, 2021). In this research, a proactive approach to fault tolerance is utilised. To mitigate the issue of single point of failure, the proposed architecture created multiple load balancing front-ends and standby application servers as fail-over systems. The proposed architecture also includes self-healing, job migration, and HAProxy replication policies as an additional fault tolerance technique. The newly developed approach utilised HAProxy's high availability keep alive technology to regularly monitor servers and services for a fast job migration. In addition, the approach used floating IP addresses that can be moved between load balancers to ensure

availability of service at optimum performance.

- 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 (Xu et al., 2013) in running applications. In a load balancer, performance overhead is influenced by several factors, including the load balancing algorithm (Shah et al., 2017; Zomaya & Teh, 2001). The proposed load balancing algorithm runs efficiently with negligible performance overhead of less than 2% when compared to other load balancing algorithms. Additionally, the new load balancing architecture exemplifies a decentralised approach to collecting and updating server metrics. This decentralised approach is a recommendation by authors in (Zomaya & Teh, 2001) to reduce performance overhead and latency. Also, in the proposed architecture, each participating VM is preinstalled with glances agent (Hennion, 2021), 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 instrumental role in determining the efficiency of a load balancing algorithm. Chen *et al.* (S.-L. Chen et al., 2017) encourage the use of varied but application-specific server metrics in load balancing algorithms. In its formulation, the proposed algorithm combines carefully selected key server metrics. These server metrics are specific to three-tier web applications and some have been recommended by these authors (Grozev & Buyya, 2014b; Sahu et al., 2013; Tychalas & Karatza, 2020; Wang & Casale, 2014). 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.

5.4 Proposed Load Balancing Service

This section describes the novel weight assignment technique and load balancing algorithm for three-tier web applications deployed on cloud.

5.4.1 Proposed Weighting Technique

The proposed VM weighting technique for the novel load balancing algorithm is an improvement on the research work by (S.-L. Chen et al., 2017) and (Z. Chen et al., 2018). The weighting technique combined four server metrics to calculate the weight of a VM. It also included a new 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 buffers/queues.

The weight of a VM represented as $W(X_k)$ requires first calculating the real-time load, $Lr(X_k)$, as shown in equation (5.1). The real-time load utilises the following server metrics: M_k, B_k, C_k, T_k. These metrics are retrieved in percentages from the monitoring tool and then converted to integer values by dividing each of them by 100.

$$\begin{aligned} Lr(X_k) = (e_1 * (C_k/100)) + (e_2 * (M_k/100)) + \\ (e_3 * (B_k/100)) + (e_4 * (T_k/100)) \end{aligned} \tag{5.1}$$

To compensate for the presence of bias and to reflect the influence each metric has on a VM, weight factors were introduced to all the metrics, as shown in equation (5.1). e_1, e_2, e_3, e_4 represents the weight of CPU, RAM, bandwidth, and thread count respectively. The sum of these weights is 1. The weight factor values were carefully chosen and proven experimental values. The 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 thread count. CPU utilisation has the most significant weight because the selected class of application becomes processor intensive when a significant amount of data is being passed. Memory utilisation has the next most significant weight because its influence on a VM can quickly lead to a nonresponsive server when the utilisation is high.

Network buffer was not included in equation (5.1), so no weight was assigned to it. The network buffer was omitted because it stores packets temporarily. Network buffers determine the ratio of network utilisation to network availability. Therefore, their 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.

The next step in the weighting technique is the definition of a threshold for load comparison. The threshold is defined as the average value of all participating application server VM load as shown in equation (5.2). n represents the number of all participating application server VMs.

$$Lr_{th} = \frac{\sum Lr(X_k)}{n} \tag{5.2}$$

To further improve and analyse the weight calculator, some modalities are set out as follows: if $Lr(X_k) \leq Lr_{th}$, this means that the application server's load is relatively small so that the weight assigned to the server can be increased. The assigned

weight should be decreased if it is the opposite, for the server's load is high. To define and quantify these changes, a modification parameter δ is defined as shown in equation (5.3).

$$\delta = \frac{Lr(X_k)}{\sum Lr(X_k)} \tag{5.3}$$

After the real-time load is calculated and a load comparison of servers is done using the aforementioned equations, the weight of a server is then calculated. The weight calculator will return the lowest real-time load value for the least utilised VM, but HAProxy, the chosen load balancer, functions in a reverse manner. This means that the load balancer expects or appropriates the largest weight value for the least utilised VM. To make the weight calculator function in line with HAProxy's weight policy, the inverse of the real-time load as shown in equation (5.4) is computed. This makes the lowest real-time load value the largest value.

Lastly, HAProxy's weight policy is bound to 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 equation (5.4).

$$W(X_k) = \begin{cases} (\frac{1}{Lr(X_k)} + \delta), Lr(X_k) \leq Lr_{th} \\ (\frac{1}{Lr(X_k)} - \delta), Lr(X_k) \geq Lr_{th} \end{cases} \tag{5.4}$$

5.4.2 Proposed Load Balancing Algorithm

The proposed novel load balancing algorithm is presented 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, the algorithm assigns a weight to each server during runtime using the newly introduced weighting technique. This process also changes the load balancer's configuration. In other words, the load balancer is immediately 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
- $Th_t r$—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. Again, to corroborate the threshold values, experimental profiling tests were performed on a medium-sized application server VM using workload generated from real web application requests. A predefined SLA that states that 90% of requests should be handled within one second, as recommended by (Qu et al., 2017) was used to determine the average utilisation percentage.

In line 7 of the algorithm, the function **TCPBufferOverloaded()** represents an extracted logic to check the TCP network buffer NB_k. This function uses the netstat command to check for the presence of a TCP socket whose buffer size ratio, Recv-Q and Send-Q values, is greater than 0.9. The value of 0.9 is a research-verified value by (Grozev & Buyya, 2014b). A ratio value greater than 0.9 means the network buffer is overloaded and requests will not arrive promptly at the VM.

The algorithm automatically retrieves the utilisation values of the chosen VM metrics. After retrieval in line 2, the algorithm loops through available server VMs and compares each utilisation metric of the current server with the set threshold. The algorithm then computes the weight using the novel weighting technique in equation (5.4), and performs the TCP network buffer check. Then, a weight is assigned to each server VM whose network buffer is not overloaded, as depicted on line 9. If the server VM is fully utilised - 100% or has passed the threshold utilisation value, a weight of 0 is assigned to the VM. This is shown in line 11. A fully utilised VM with a weight of zero means that the VM will not accept any more incoming requests until the utilisation rate is lower or equal to the set threshold.

On 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_t r, VM_as
1  RetrieveAllocateToInputThresholdValues ();
2  for each VM, vm_i ∈ VM_as do
3  |  Utl_cpu ← CPU utilisation of vm_i;
4  |  Utl_ram ← RAM utilisation of vm_i;
5  |  Utl_bw ← Bandwidth utilisation of vm_i;
6  |  UtlThreadCount ← Threadcount of vm_i;
7  |  if (Utl_cpu ¡ Thc & Utl_ram ¡ Thr & Utl_bw ¡ Thbw & Utl_ThreadCount ¡
   |    Th_t r & !TCPBufferOverloaded ()) then
8  |  |  W(X_k) ← CalculateWeightbyutilisation (Utl_cpu,
   |  |    Utl_ram, Utl_bw, UtlThreadCount, VM_as, vm_i);
9  |  |  assignweighttoVM (vm_i, W(X_k));
10 |  else
11 |  |  assignweighttoVM (vm_i, 0);
12 |  end
13 end
14 HAProxyAssignRequest (s_i, VM)
```

A visual representation of the novel load balancing algorithm and the weighting technique is depicted in Figure 5.3.

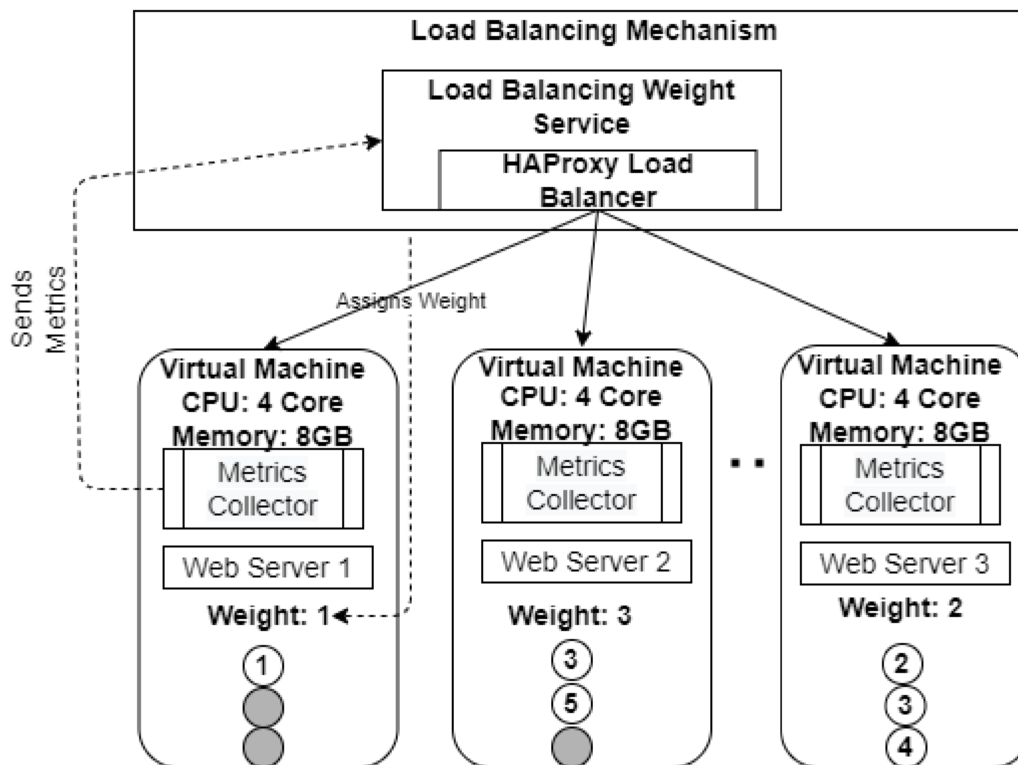


Figure 5.3: Visual Representation of Novel Load Balancing Algorithm.

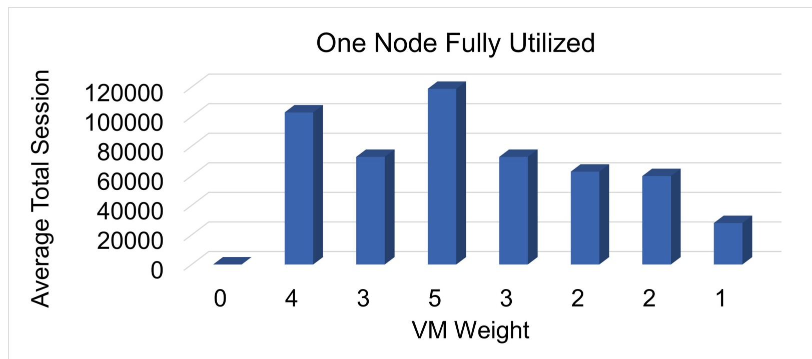
5.5 Results and Performance Analysis

This section discusses the analysis and performance comparison of the novel load balancing algorithm and the benchmarks.

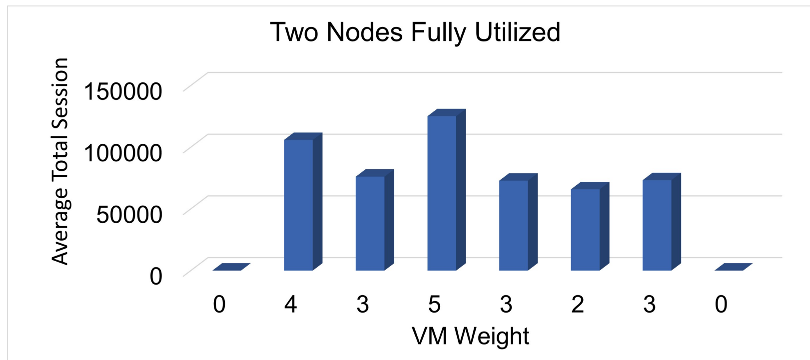
5.5.1 Performance under Resource Failure

In order to test the novel algorithm's applicability, a basic workload distribution test was first done. The novel load balancing algorithm's ability to distribute workload across VM was first validated. The baseline algorithm by Grozev *et al.* (Grozev & Buyya, 2014b) was also validated to determine whether the bespoke replication of the baseline algorithm produces similar results recorded by the researchers. Varying numbers (1 – 5) of over-utilised VM were mimicked as depicted in Figures

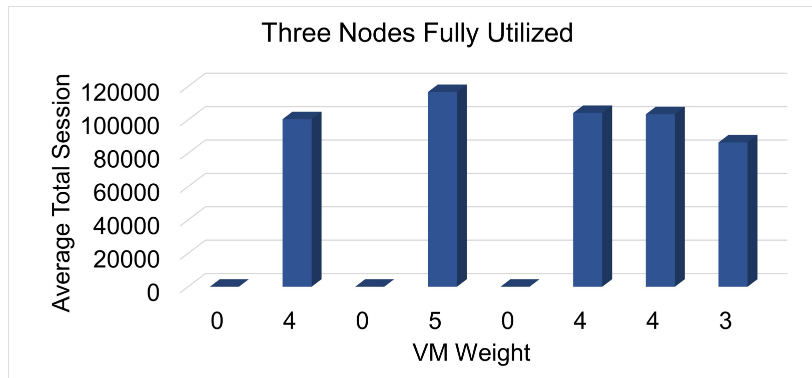
5.5. The aim of this test is to check if the novel load balancing algorithm will assign the correct weight to the VMs while it redirects workload to other VMs. The result is that the newly developed load balancing algorithm distributed workload among available VMs based on their current capacity, such that the stipulated SLA was not violated.



(a) One Fully Utilised Node.

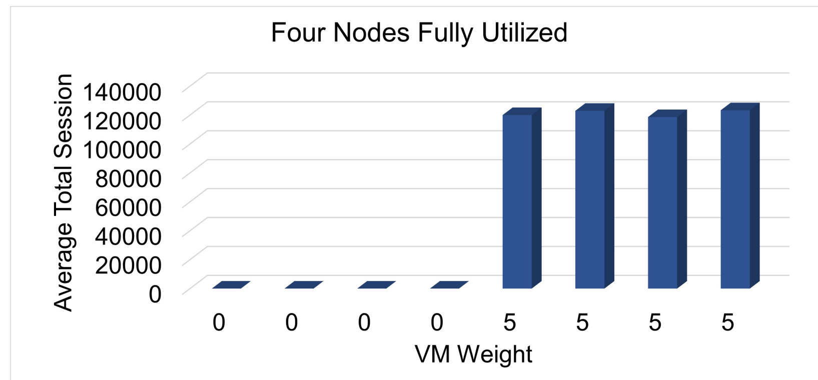


(b) Two Fully Utilised Nodes.

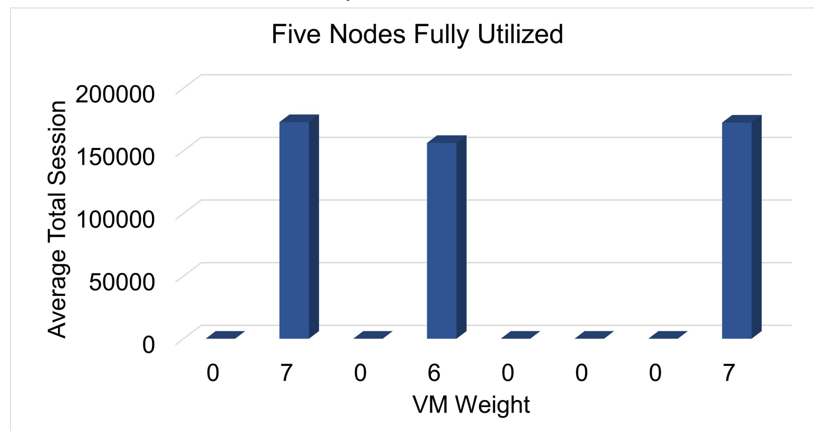


(c) Three Fully Utilised Nodes.

To test fault tolerance and scalability of the load balancing system, tests in resource



(a) Four Fully Utilised Nodes.



(b) Five Fully Utilised Nodes.

Figure 5.5: Fully Utilised VM Nodes.

failure situations were performed. 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 in every experiment for over ten seconds.

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 algorithms (novel load balancing algorithm, baseline algorithm, and round-robin algorithm) were compared in all the experiments. The round-robin algorithm performed worst, as depicted in Figure 5.7. The number of requests the round-robin algorithm handled when it encountered between one

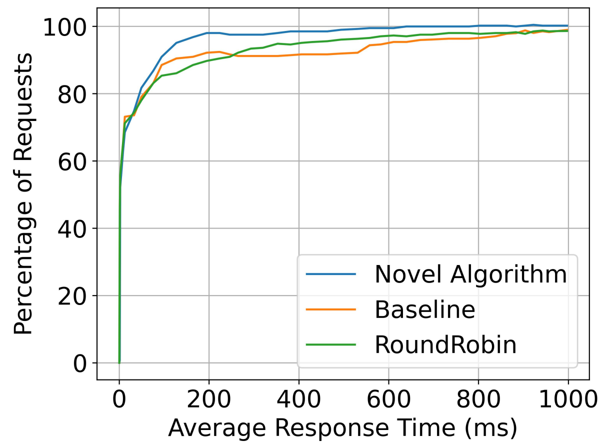
and two server failures was 10% less than the stipulated SLA requirement. The performance of the round-robin algorithm declined when server failure was more than four. The round-robin algorithm handled less than 80% of requests within one second. This indicates that distributing workload randomly, even uniformly, is not enough for this class of application. This type of load balancing might be sufficient for applications that are not critical or resource intensive.

The baseline algorithm by (Grozev & Buyya, 2014b) performed better than the round-robin algorithm, as shown in Figure 5.7 below. 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 performance degradation. The response times were high, with values between 1.2 and 1.6 seconds. The percentage of requests handled declined to less than 80% of requests. Furthermore, it was observed that at the peak of server failures, the algorithm became unresponsive, resulting in errors as shown in Figure 5.8 below.

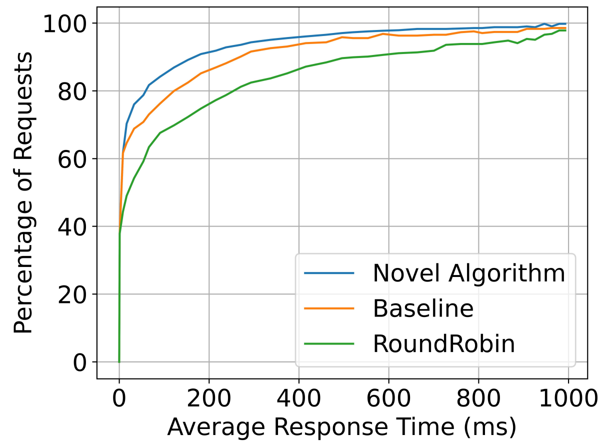
The novel load balancing algorithm distributed requests within the stipulated SLA as shown in Figure 5.7. It did not violate the SLA at any point of the server failure. Response times were constant and improved compared to the two other algorithms. During all server failures, the novel algorithm attended to over 90% of requests between 0 ms and 998ms. In the study, it was found that response times of the novel algorithm fell within an acceptable range. In addition, they were less than the response times of the two benchmarked algorithms. This proves that the proposed novel algorithm functions best for the chosen class of applications.

To explore boundaries where the new load balancing approach starts to break down, the ratio of the number of server failures to available servers was experimented with. The results showed that to achieve the SLA recommendation of 90% of requests to be handled in one second, using the proposed workload model with number of requests between 1000 and 50,000 requests, a minimum of three VMs should be available and running. Figure 5.8 showed that the novel algorithm had less than 1500 failed requests when there were five server failures, unlike round-robin and

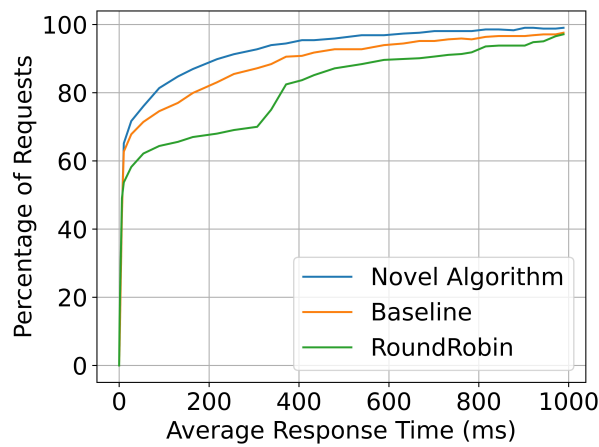
the baseline algorithm that had over 3000 failed requests with response times of more than one second.



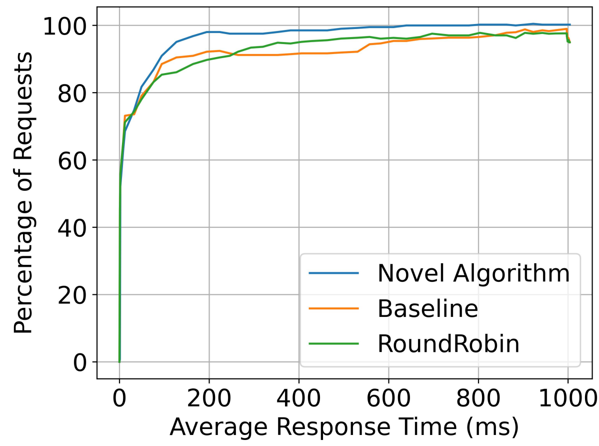
(a) 1 Server Failure.



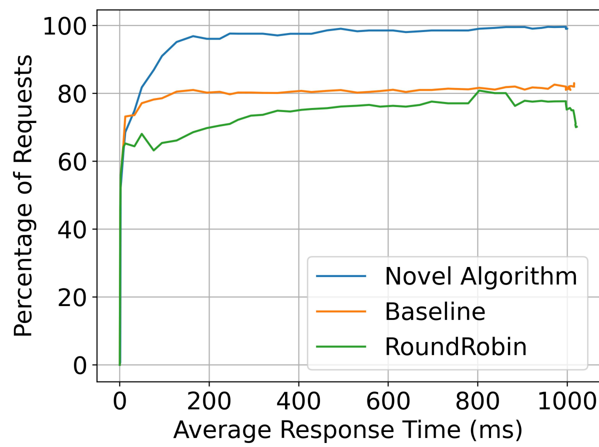
(b) 2 Server Failures.



(c) 3 Server Failures.



(a) 4 Server Failures.



(b) 5 Server Failures.

Figure 5.7: Distribution Values of Server Failures within Private Cloud.

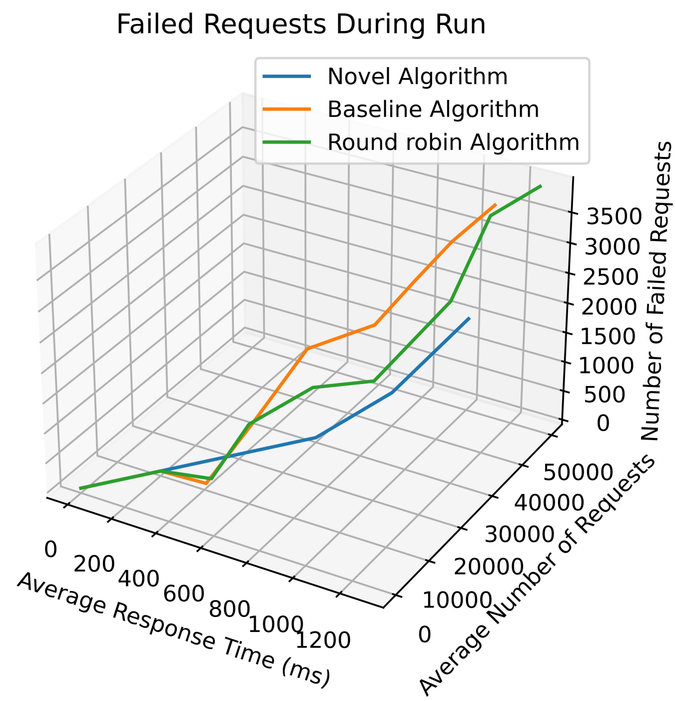


Figure 5.8: Failed Request Chart — Error Rates

5.5.2 Performance under Flash crowd

The performance of the novel algorithm was tested under flash crowd scenarios. Firstly, three VMs out of eight were saturated to 100% utilisation over a period of five minutes. The novel algorithm consistently distributed the requests, maintained SLA and the average response time was low and became lower when the auto-scaler provisioned a new VM. The baseline algorithm by Grozev *et al.* (Grozev & Buyya, 2014b) continued to distribute requests but had more failed requests within four seconds of the flash crowds. The round-robin algorithm became unresponsive at the five-second mark because it had a queue of requests trying to access the VM.

Secondly, the request workload was updated 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 until the peak of the flash crowd exceeded 150%. The three algorithms handled 90% or more requests within one second when the peak of flash crowds was not more than 125%, as depicted in Figure 5.9 below. When the flash crowd reached 150%, the algorithm's response times varied. The novel algorithm still maintained the SLA constraint of attending to 90% of requests within one second, when, however, the benchmarked algorithms started to violate the SLA.

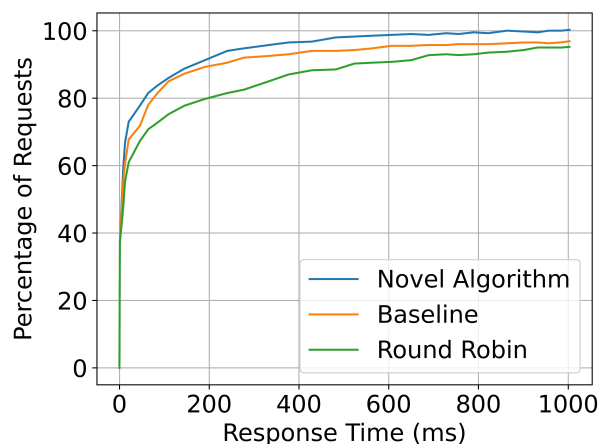


Figure 5.9: 125% Increased Request Flash Crowds.

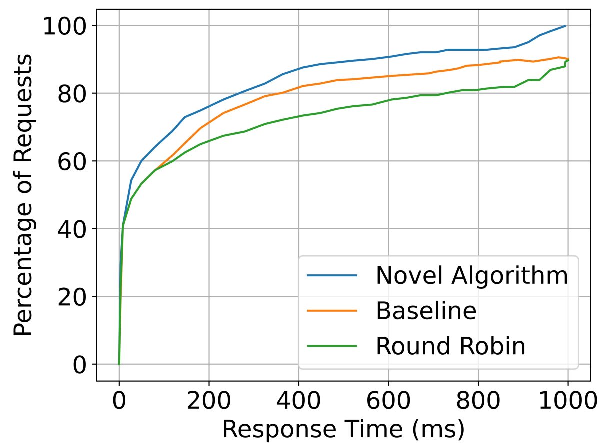


Figure 5.10: 150% Increased Request Flash Crowds.

Figure 5.11 revealed that flash crowds of 188% have begun to deplete the baseline and round-robin algorithms' performance. The baseline algorithm by Grozev *et al.* (Grozev & Buyya, 2014b) had failed requests of 8% of the overall requests.

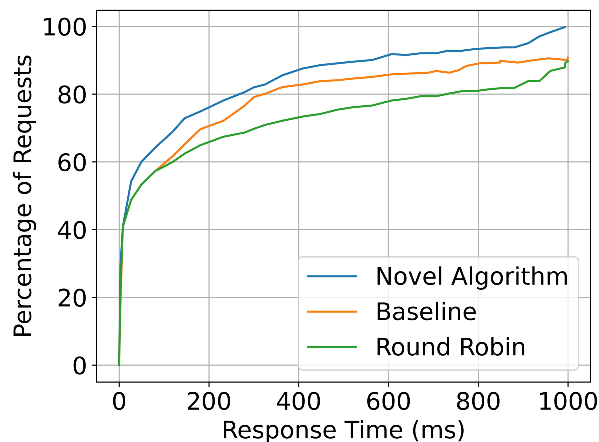


Figure 5.11: 188% Increased Request Flash Crowds.

Flash crowds of 240% revealed that the round-robin algorithm could only handle 50% of requests at approximately 300ms when the first flash crowd happened, and 56% of requests when flash crowds occurred a second time. Furthermore, Figure 5.12 revealed that the round-robin algorithm took 1190 ms before it could attend to 67% of requests. This violates the SLA constraint. The baseline algorithm as

shown in Figure 5.12 performed better than the round-robin algorithm when it experienced a flash crowd of 240% above normal workload. The baseline algorithm handled 70% of requests at around 300ms when the first flash crowds hit, and then served 81% of requests when the second flash crowd happened. The baseline algorithm handled a total of 89% of requests at 999.987ms, a response time very close to the SLA. The baseline algorithm could not complete 90% of requests within one second.

The novel algorithm as shown in Figure 5.12 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 maintained the handling of a minimum of 90% of requests within one second, even though the response time of 999.999ms was higher than the response times of previous occurrences of flash crowds with response times between 960.905ms and 992.798ms.

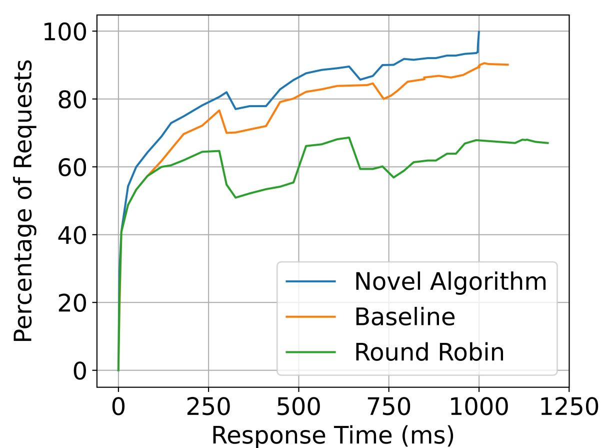


Figure 5.12: 240% Increased Request Flash Crowds.

A flash crowd of 320% above the normal is shown in Figure 5.13. Round-robin algorithm processed 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 respond to 70% of requests within one second before the algorithm started to time out. In contrast, the novel algorithm handled 70% and 80% of requests when flash crowds of 320% occurred. The novel algorithm still completed 90% of the workload

but not within one second. The ability of the algorithm to consistently distribute workload during varying scenarios confirms and validates our choice of carefully selected metrics.

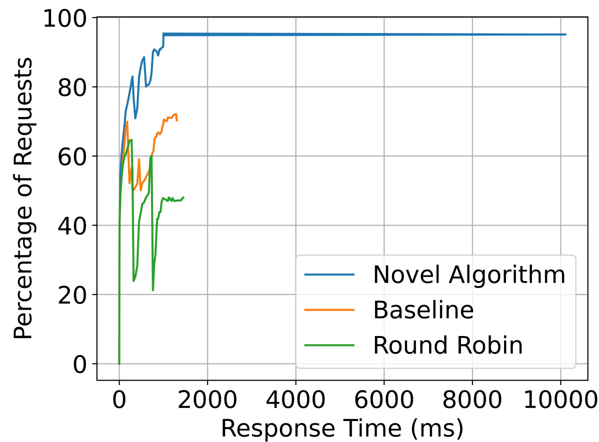


Figure 5.13: 320% Increased Request Flash Crowds.

To test the effect of the autoscaler when there are flash crowds and how the novel 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. They were used by the autoscaler to launch extra VMs when required. Figure 5.13 showed the effect of auto-scaling after experiencing a flash crowd of 320% above normal workload.

Figure 5.14 showed the amount of failed requests 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 round-robin which had almost 19000 failed requests and the baseline algorithm, which had 14560 failed requests.

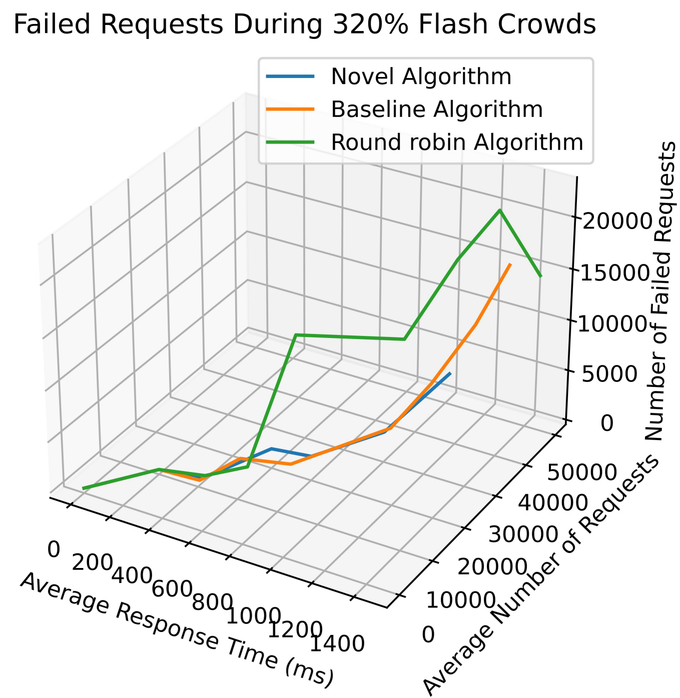


Figure 5.14: Failed Request Chart — Error Rates During Peak Flash Crowds

5.6 Summary

In this chapter, the research introduced a novel weight assignment load balancing algorithm and a load balancing service architecture for three-tier web applications that are deployed on cloud. Previous research confirms that web applications commonly suffer from flash crowds and resource failures, which leads to performance degradation experienced by users of these applications. To combat the challenges faced by web applications, most especially three-tier web applications deployed on the cloud, this research proposed a load balancing algorithm, service and architecture that utilised five carefully selected server metrics in determining and assigning weight to application server VMs. These server metrics are as follows: thread count, CPU, RAM, network buffer, and network bandwidth utilisation.

The proposed approach to load balancing the workload of the chosen class of application included using server metrics to determine the weight of each VM by analysing their utilisation values to determine a VM's real-time load. The overall load balancing architecture featured the ability to migrate and load balance the workload of existing web applications to the cloud with less modification to code base. The approach also included the use of a highly available deployment architecture and standardised load balancer software. This helped overcome common challenges of load balancing solutions such as single point of failure, reliability, and scalability. Furthermore, the architecture featured the deployment of the newly developed load balancing service in the same data centre. This allowed for quick adaptation to changes in the environment, reduced communication overhead, and faster network capabilities.

The load balancing solution and two benchmark algorithms were applied on a three-tier web application and evaluated in a private cloud data centre running OpenStack. The novel algorithm was first validated on the testbed by testing its efficiency. Second, the performance of the novel algorithm was compared with a baseline load balancing algorithm by Grozev *et al.* (Grozev & Buyya, 2014b) and round-robin algorithm. The experiments involved sending varied user request workloads through Apache Jmeter to the deployed application servers.

The workload was modelled using a Poisson distribution to emulate real user behaviours. The experiment measured and compared response times of the case study application during the simulation of the workload.

The proposed novel algorithm improved overall average response times in resource failure situations by 20.7% and 21.4% when compared to the baseline and round-robin algorithms, respectively. During flash crowd situations, the novel algorithm improved the average response time by 12.5% and 22.3% when compared to the baseline and round-robin algorithms, respectively. The proposed approach consistently distributed the workload of the chosen class of application without violating the stipulated SLA constraints of attending to 90% of requests within one second and within reasonable response times. These results prove that the novel algorithm can adapt to flash crowds and resource failures without performance degradation.

Chapter 6

Multi-Cloud Load Distribution Algorithm and Architecture

6.1 Introduction

This chapter presents a novel decentralised multi-cloud load distribution algorithm and architecture that can alleviate the challenges of vendor lock-in issues, legislative compliance issues, low network latency issues, and the negative effects of flash crowds and resource failures. It also describes the multi-cloud deployment pattern and how it can be leveraged to support the deployment of three-tier web applications on the cloud. The content of this chapter has been previously published in (Adewojo & Bass, 2022a). The contributions of this chapter are as follow:

1. a decentralised multi-cloud load balancing architecture that includes a multi-cloud load balancing algorithm to distribute the workload of web applications across multiple clouds
2. an improved communication protocol of multi-cloud load balancing system;
3. a flexible and general application brokering architecture that can facilitate

resource management and efficient workload distribution, especially for existing web applications that will be migrated to the cloud

4. implementation and an experimental evaluation of the proposed multi-cloud solution using a heterogeneous experimental environment - a combination of one private and two public clouds

This chapter starts by describing the requirements for the proposed multi-cloud algorithm in section 6.2. The deployment and overall architecture of the algorithm and deployment strategy are discussed in section 6.3. The novel multi-cloud load distribution service including the proposed weighting technique and algorithm are discussed in section 6.4. An empirical experimental evaluation and results are presented in section 6.5. Section 6.6 concludes and summarises the chapter.

6.2 Application Requirements

The target applications for this study are three-tier web-based business applications deployed across multi-cloud. In addition, to support request forwarding, the application instance in each data centre should be able to communicate with instances deployed in other data centres. This approach requires session continuity and data locality to support the processing of requests by application replicas deployed across multiple clouds.

Session continuity ensures uninterrupted service experience to the user, regardless of changes to the server or equipment's IP address. Stateless applications, such as search engines and applications that utilise web services to achieve statelessness, do not save client data generated in one session for use in the next session with that client. Also, stateless applications can easily scale because they can be deployed across multiple servers and/or cloud data centres without issue while ensuring session continuity. These properties of stateless applications implicitly satisfy the requirement of session continuity.

Data locality ensures that data resides close to the system it supports. In the context of this research, data locality means that data should be replicated across multi-cloud, since requests can only be forwarded to data centres with available data. To corroborate this concept for the proposed system, Grozev *et al.* (Grozev & Buyya, 2014b) supports data replication for multi-cloud applications because it is a good performance indicator (Henderson et al., 2015; Jacob et al., 2008), and thus, improves applicability of this approach.

6.2.1 Communication Protocol

A unique communication protocol was employed in this research to support a streamlined communication approach and to reduce the communication overhead incurred. The load balancing solution was deployed on each participating data centre. Each load-balancing solution in a data centre communicates with each other using a peer-to-peer client-server communication protocol as depicted in Figures 6.2 and 6.1. Each solution relays its system state to another solution in a different data centre at a regular predefined time interval of two seconds and every time the load balancer distributes workload. Each communicated system's state always comprises the originated state and the states of the peered system. This chosen mode of communication protocol helps to reduce network overhead associated with node communication by only broadcasting to the peered node. It provides significantly better spatial reuse characteristics, irrespective of the number of nodes. As the number of nodes increases significantly, there might be a slight degradation in performance, but the advantages definitely outweigh this drawback.

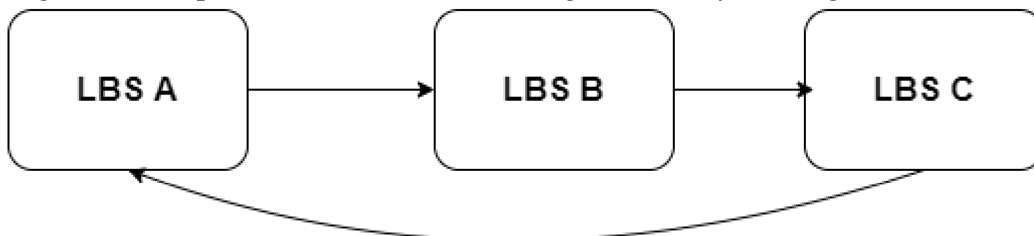


Figure 6.1: Communication Protocol

6.3 Deployment and Overall Architecture

The proposed decentralised architectural design is presented in Figure 6.2. This architectural design features a dynamic load balancing algorithm and technique proposed by (Adewojo & Bass, 2022b) and forms part of the multi-cloud load balancing service. The proposed load balancing service (LBS) is deployed as an extra layer of component that augments the three-tier architecture. This extra layer of component is an architectural approach to improve the migration of existing web applications to the cloud with less modification to the code base of the application. Each LBS is deployed alongside the proposed solution application in the same data centre; this deployment design helps to reduce latency in detecting workload requests. The services are connected to each other through a virtual private network to ensure communication.

Each LBS consist of monitoring, controller, and communication modules. The monitoring module constantly monitors the incoming requests and the status of available resources to detect resource failures, increased workload, application, or server workload. A periodic health check to detect all types of failures is also carried out by the monitoring module. The load balancing service recalculates the weights of VM and checks the available capacity of each VM on a regular basis. If a failure happens before the check, a recalculation is done immediately to properly distribute requests both within the data centre and across all data centres to avoid performance degradation. The controller module is used to modify the weight of each VM to accommodate request workload. The communication module communicates the capacity and status of each data centre on a regular basis.

In this multi-cloud architecture, users are redirected to the nearest data centre to be served. This is determined using their IP address location and latency calculation implemented in the algorithm. The load balancing service's entry point coordinates this. User requests are, then, sent to the application server based on the load balancing algorithm's weight calculation. After a successful interaction of the requests, responses are sent back to users and the process is repeated for every user request sent.

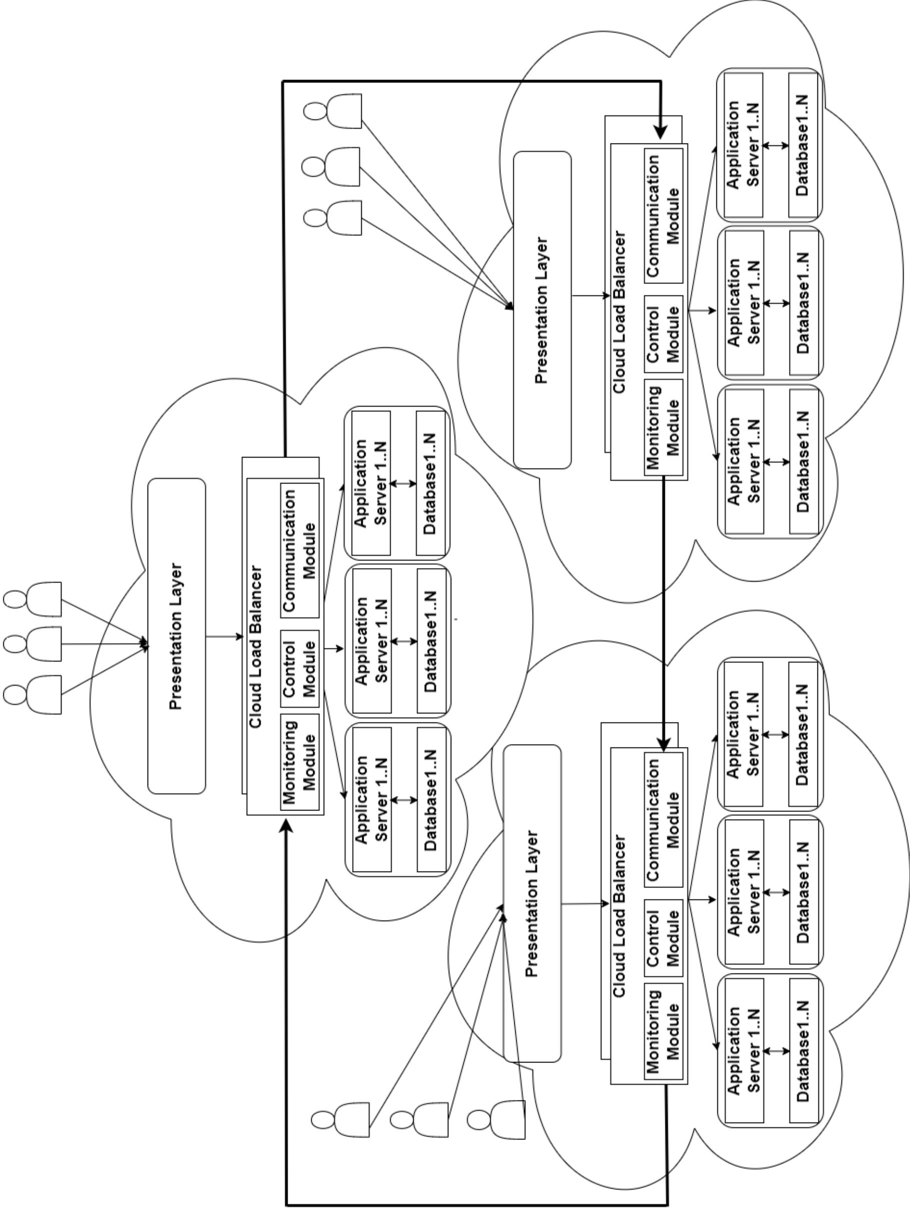


Figure 6.2: Load Balancing Overall Architecture.

6.4 Proposed Multi-Cloud Load Distribution Service

This section presents the approach to calculating the weights of each data centre and the multi-cloud load balancing algorithm.

6.4.1 Proposed Multi-Cloud Weighting Technique

To detect and overcome flash crowds and resource failures, this research used key server metrics of an application server to determine the state of the application servers. The original algorithm of Adewojo *et al.* (Adewojo & Bass, 2022b) implements a unique weighting technique that combines five carefully selected server metrics utilisation (CPU, Memory, Bandwidth, Network Buffer, and thread count) to compute the weight of a VM. This multi-cloud load balancing solution improves the algorithm by including the calculated weight of each data centre that will be used in load distribution and the network latencies between data centres.

To calculate the weight of each data centre, the definition of a real-time load $Lr(X_k)$ as described by (Adewojo & Bass, 2022b) in chapter 5 is used to calculate the weight of each data centre, as shown in equation (6.1). Therefore, aside from each application server's weight, a data centre will also be assigned a weight to determine its capacity to handle user requests.

$$W(DC_i) = \frac{\sum \frac{1}{Lr(X_k)}}{n} \tag{6.1}$$

6.4.2 Multi-Cloud Load Balancing Algorithm

The novel multi-cloud load balancing algorithm is abstracted in Algorithm 2. The first step in the algorithm is to receive and set an overall threshold for the input parameters. The values for these thresholds and how they were calculated can be

found in (Adewojo & Bass, 2022b). The algorithm loops through a list of VMs and compares each utilisation values against the set threshold. The weight of each VM is then computed and assigned to VMs as described in (Adewojo & Bass, 2022b). The algorithm in line 5 further loops through all remote data centres and calculates the weight of each data centre using equation (6.1). Line 7 assigns the weight of each data centre. If a VM or data centre cannot accommodate any more requests, it sets the weight to zero. The requests are then assigned to servers and data centres based on the assigned weights, as shown in Line 9.

The network latency between data centres is used to determine the nearest data centre to route requests, as shown in line 9 in the algorithm. This means that before sending a request to the data centre, the latency between users and data centres is checked and data centres are arranged based on capacity and latency. Then, requests are sent to the applicable data centre based on the calculations that determine the best-fit data centre.

There are two types of input parameters to the algorithm: application-dependent and infrastructure-dependent. The input parameters are as follows:

Application Dependant Parameters

- Th_c—CPU threshold
- Th_r—RAM threshold
- Th_{bw}—Bandwidth threshold
- Th_{tc}—Thread count threshold

Infrastructure Dependant Parameters

- VM_{as}—list of currently deployed application server VMs

- VM_{dc}—list of currently deployed application server VMs per remote data centre
- $clouds$—list of participating remote data centres
- L_i—Latency to the ith data centre from the forwarding data centre

6.4.3 Algorithm Implementation

The proposed algorithm is implemented as a separate software program that connects to a state-of-the-art load balancer (HAProxy 2.4.2-1). The reason for creating a separate software program is because HAProxy does not support complex configurations featured in this algorithm. The load balancing program was colocated with the HAProxy load balancer to reduce network latency. HAProxy's health monitor is configured to monitor the performance indicators and VM's health every 2s.

The program's monitoring module periodically fetches required monitored information using HAProxy's stats application programming interface (API). Then, it extracts and manipulates performance values using the mathematical method proposed in Chapter 5 and the health statuses of the attached VM and passes them to the control module. The control module activates the algorithm to determine the weight of each VM and data centre. The control module passes the weights to the load balancer and, also, updates the communication module.

Request distribution and admission control were implemented by dynamically changing HAProxy's configuration. When required, the control module dynamically creates a new configuration file for HAProxy during runtime. This process automatically reloads the new configuration to the running HAProxy load balancer before the load balancer distributes requests among the data centre.

To activate request forwarding, each new configuration file contains the IP addresses of the load balancers located in other participating data centres and represents them as normal servers with individual weights. The multi-cloud load balancing service

assigns weight to each server. The assigned weight will determine the amount of requests that can be distributed across data centres and VMs in each data centre. HAProxy then uses the weighted round-robin algorithm to distribute requests among application server VMs.

Admission control was implemented through use of Access Control List (ACL) mechanism of HAProxy. It was used to define the black and white lists of IP locations. HAProxy's customised default page was used to create a customised error page that informs users of delay when there is a surge in user requests that consequently affect response times.

Algorithm 2: Multi-Cloud Request Handling Algorithm

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

1 `RetrieveAllocateToInputAllThresholdValues` ();
2 **for** *each VM,* $vm_i \in VM_{as}$ **do**
3   `assignweighttoVM` $(vm_i, W(X_k))$ according to (Adewojo & Bass, 2022b);
4 **end**
5 **for** *each cloud,* $vm_j \in VM_{dc}$ **do**
6     $W(DC_k) \leftarrow$ `CalculateWeightofDataCentre` (Lr_k, VM_{dc}, vm_j);
7   `assignweighttoDC` $(vm_{dc}, W(DC_k))$;
8 **end**
9 `HAProxyAssignRequest` $(s_i, VM \in clouds, L_i)$

6.5 Performance Analysis and Results

This section discusses the performance analysis of the novel multi-cloud algorithm with respect to improving performance of the chosen web application when flash crowds and resource failure situations occur. As discussed in Chapter 3, the novel algorithm was evaluated using a hetergenous cloud infrastructure. A combination of a private cloud and two public clouds were used for the experiments.

6.5.1 Benchmarks

To validate and compare the performance of the novel multi-cloud load balancing algorithm, the results were benchmarked with the following:

- Request Queuing: This benchmarking process queues up all requests in the local servers, imposes no admission control, does no geographical balancing, and uses just the round-robin algorithm. This imitates the situation that an auto-scaler is booting a new VM within a data centre.
- Admission control: This benchmarking process directly imposes admission control when distributing requests. It lets the load balancer redirect requests at first and if there is no capacity to accept the redirected requests, it sends a message to users to tell them they are in a queue.

6.5.2 Performance under Flash crowds

The next set of experiments tested the system under flash crowds using the workloads described in Chapter 3. Flash crowd was simulated across data centres at each experiment.

Figure 6.3 shows the performance of the novel algorithm and benchmarks. This represents a flash crowd of 140% above the normal workload across the data centres every five minutes for a period of fifteen minutes. The percentage of requests handled by the novel mutli-cloud algorithm was consistent across the time and it was able to keep to the SLA of attending to 90% of requests within one second. The performance of admission control was very similar and kept to the SLA. The performance of Request queuing was consistent but could not keep to the SLA because the percentage of requests handled was 12% less than the stipulated SLA.

Using the same approach of testing, Figure 6.4 shows the algorithm's performance and benchmarks when experiencing a flash crowd of 190% above the normal work-

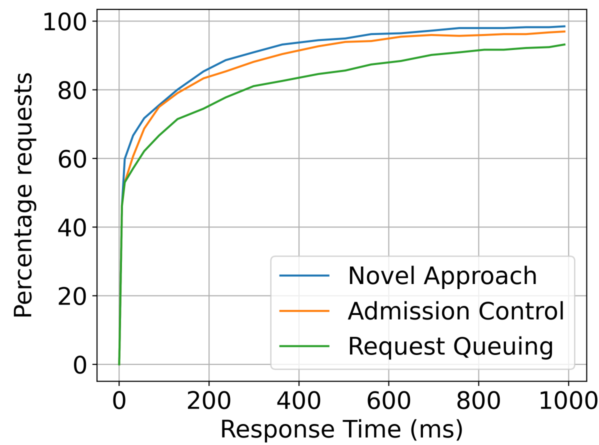


Figure 6.3: Cumulative Distribution Values of Flash Crowds – 140% increased flash crowd.

load. Request queuing had started to encounter performance degradation because it was handling an average of 65% of requests. Admission control benchmark still maintained a consistent performance but could not keep to the SLA requirement. The Novel approach was able to keep to the SLA. However, the rate at which it started attending to 80% and above requests was not as fast as the previous experiment.

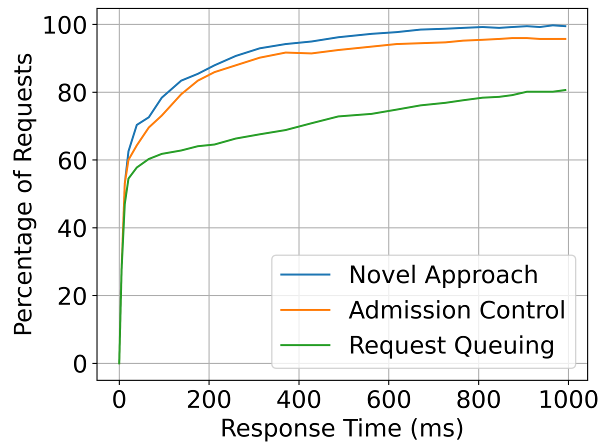


Figure 6.4: Cumulative Distribution Values of Flash Crowds – 190% increased flash crowd.

Again, using the same approach of testing, Figure 6.5 shows the algorithm's performance and benchmarks when experiencing a flash crowd of 240% above the

normal workload. The novel approach was able to keep to the SLA requirement for request handling. Admission control and Request queuing could not handle up to 90% of requests within one second.

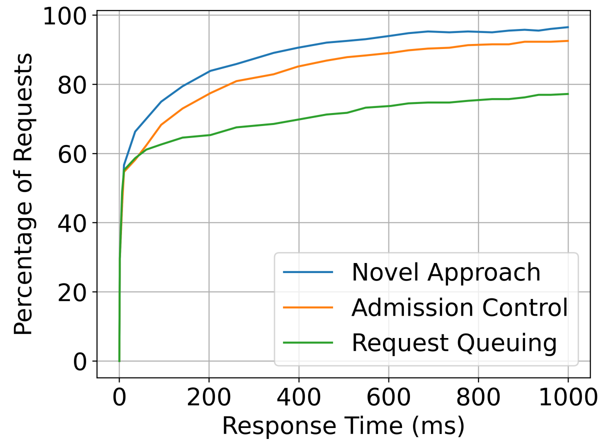


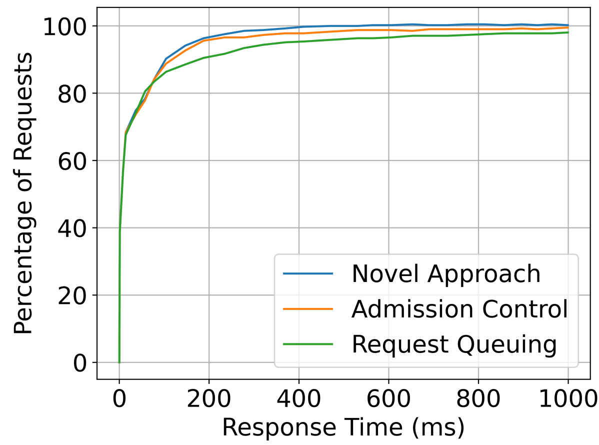
Figure 6.5: Cumulative Distribution Values of Flash Crowds – 240% increased flash crowd.

Overall, experimental results showed that the novel multi-cloud load balancing algorithm and approach outperformed the benchmarks at every instance of flash crowds. An improvement in the percentage of requests handled was recorded. The novel approach improved response times by 4.08% and 20.05% relative to admission control and request queuing benchmarks, respectively. This confirms that the solution can consistently distribute the request of web-based three-tier business applications even during flash crowds. It is noted that the size of the VM also determines the performance, therefore a better-optimised VM for web applications will offer a lesser response time if it is coupled with this new solution.

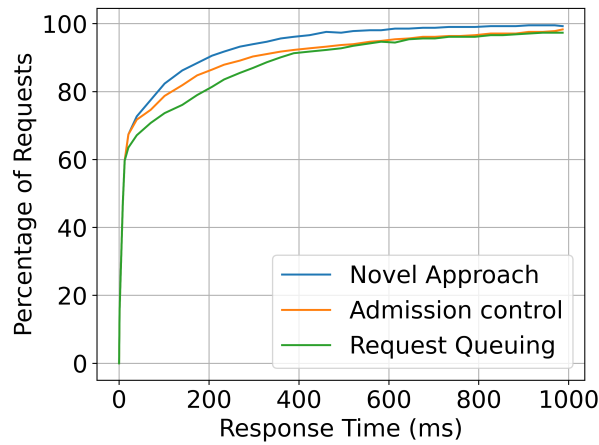
6.5.3 Performance under Resource Failure

Figures 6.6a, 6.6b, 6.6c show the performance of the novel and benchmark algorithms during one server failure in each of the cloud data centres. The performance of the algorithms were similar during one server failure in the private cloud and DigitalOcean cloud. However, in AWS, the performance differed. Though the novel algorithm and Admission control kept to the SLA, request queuing could not keep to the SLA. This is partly the consequence that the VMs in AWS were smaller.

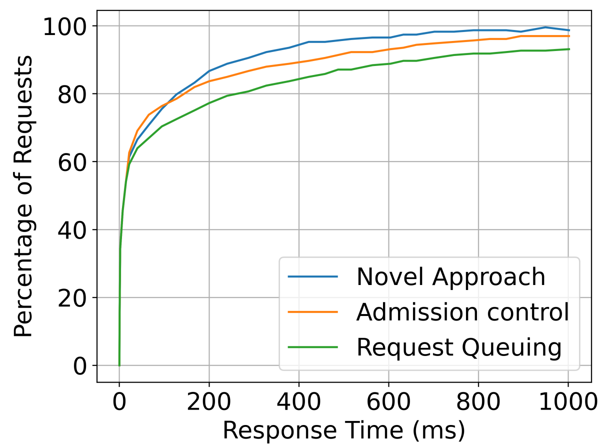
Figures 6.7a, 6.7b, 6.7c show the performance of the novel and benchmark algorithms when two server failures occurred in each data centre. In all the experiments across the three data centres, the queuing benchmark performed far below the SLA. It experienced performance degradation in all the data centre. The novel algorithm handled up to 90% of requests in all the data centre. Admission control kept to SLA only in the private cloud. Also, Admission control exhibited higher response times in all the data centre and this indicates performance degradation.



(a) 1 Server Failure in Private Cloud

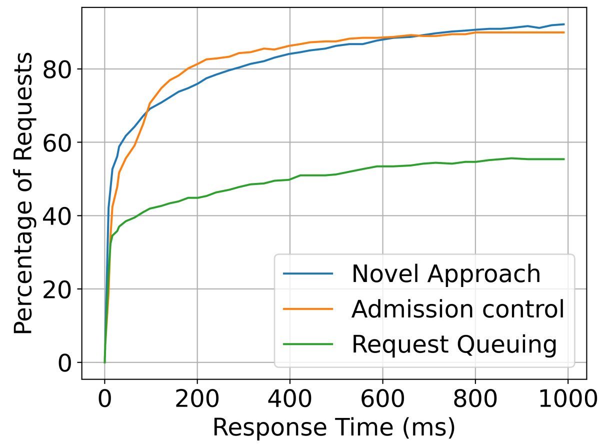


(b) 1 Server Failure in DigitalOcean

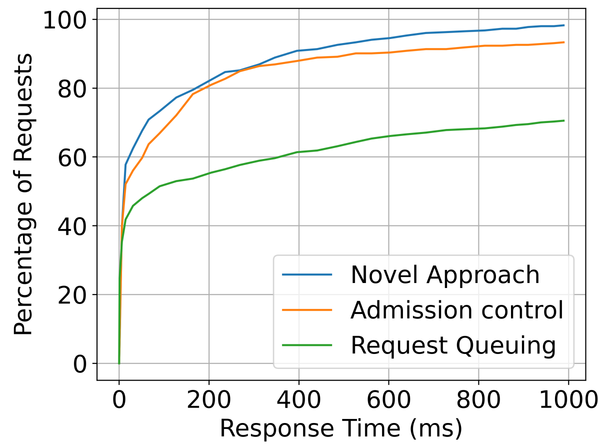


(c) 1 Server Failure in AWS

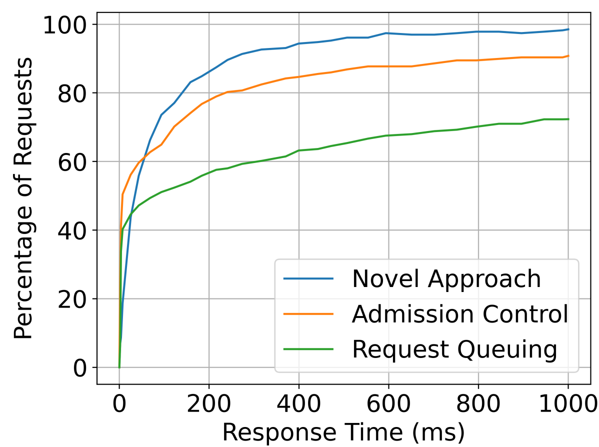
Figure 6.6: Cumulative Distribution Values of One Server Failure across Cloud Data Centres.



(a) 2 Server Failures in Private Cloud



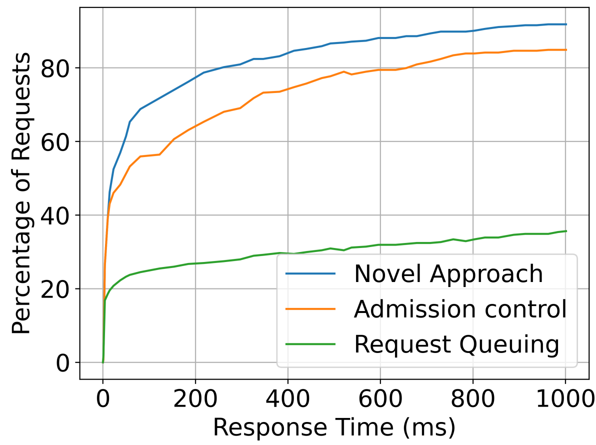
(b) 2 Server Failures in DigitalOcean Data Centre



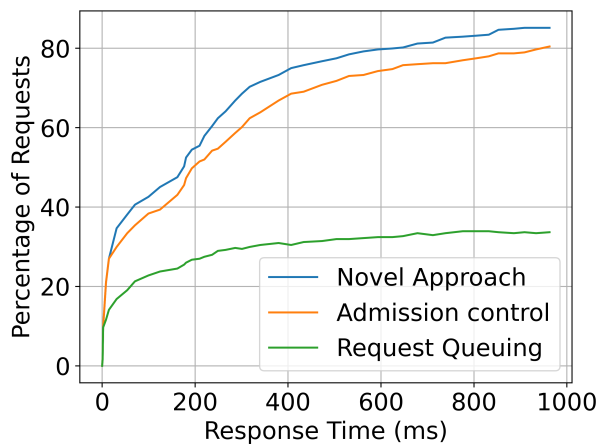
(c) 2 Server Failures in AWS Data Centre

Figure 6.7: Cumulative Distribution Values of Two Server Failures in One Data Centre Each.

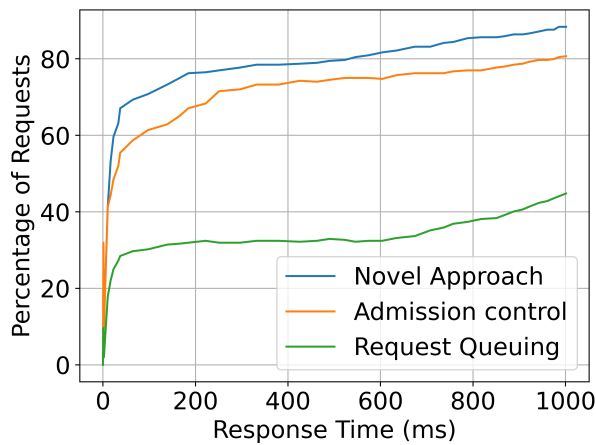
Figures 6.8a, 6.8b, 6.8c show the performance of the novel and benchmark algorithms when two server failures occurred in combinations of two data centres. The novel algorithm outperformed the admission and request queuing benchmarks in the combination of the private cloud and DigitalOcean cloud. Request queuing performed worst and could only attend to 38% of requests in one second. Similarly in the combination of Private cloud and AWS, the novel algorithm outperformed the two benchmarks. Admission control could not handle up to 90% of requests in this combination. Request queuing consistently encounters performance degradation with high response time. The combination of DigitalOcean and AWS showed an improvement in Request queuing, but this benchmark still performed the worst. The benchmarks could not all respond to 90% of requests at a lesser response time compared to the new load balancing approach.



(a) 2 Server Failures across OpenStack and DigitalOcean Data Centres



(b) 2 Server Failures across OpenStack and AWS Cloud Data Centres



(c) 2 Server Failures across DigitalOcean and AWS Cloud Data Centres

Figure 6.8: Cumulative Distribution Values of Two Server Failures in Two Data Centres Each.

Figure 6.9 shows a cumulative performance of the algorithm and the benchmarks when there were three VM failures in each of the cloud data centres. This approach made the data centres become unresponsive. The novel approach also encountered performance degradation and could not maintain the defined SLA. The response times were high considering the number of requests handled in all the scenarios. This showed that there is a minimum requirement for the number of VMs that can run in the data centre to assure consistent performance of applications deployed on cloud.

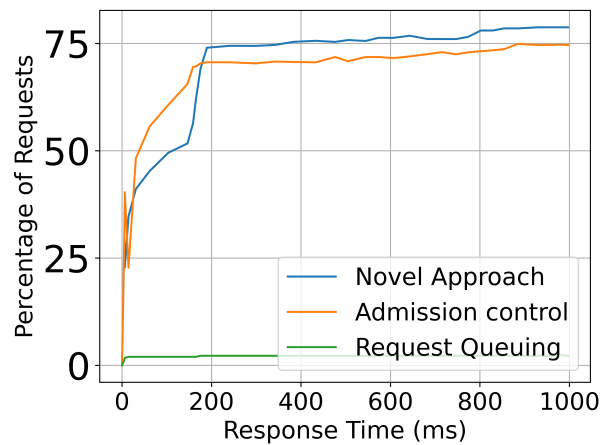


Figure 6.9: Three Server Failures across Cloud Data Centres.

In summation, the performance of the novel approach outperformed the performance of both admission control and request queuing benchmarks. The response time of the novel approach was 6.7% better than the response time of the admission control benchmark. This means the novel approach can handle more workload with an acceptable response time during server failure scenarios.

6.6 Summary

Web-based business applications commonly experience spikes in user requests called flash crowds. Flash crowds in web applications might result in resource failure and/or performance degradation. Furthermore, the traditional approach of hosting web applications using a single cloud leads to challenges such as service unavailability, vendor lock-in, regulatory compliance, network latency between end users and cloud sites, exorbitant billing, lack of varied cloud resources, and so on. This class of application would benefit from using a targeted load balancer and a multi-cloud deployment architecture to alleviate these challenges.

This chapter presents a decentralised multi-cloud load balancing architecture that includes a novel multi-cloud load balancing algorithm. The algorithm distributes web-based business application workloads using geographical dynamic load balancing to minimise performance degradation resulting from flash crowds and resource failures while improving response times. By improving the dynamic load distribution algorithm introduced by Adewojo *et al.* (Adewojo & Bass, 2022b), which uses five carefully selected server metrics to determine a server's capacity before distributing requests, the proposed approach creates a multi-cloud load balancing algorithm.

In the first experiment, the novel multi-cloud algorithm was compared to load distribution benchmarks. Additionally, an experimental evaluation of the proposed solution was conducted on a multi-cloud test bed that consists of one private cloud and two public clouds. A standard exponential benchmark and workload model were used to simulate flash crowds. It simulated resource failure by shutting down virtual machines in some chosen data centres. Next, careful measurements of response times in these various scenarios were made. The experimental results showed that the proposed solution improved application performance by 6.7% during resource failure periods, 4.08% and 20.05% during flash crowd situations when compared to Admission Control and Request Queuing benchmarks respectively. This validates that the proposed approach improves the performance of multi-cloud deployed web-based three-tier applications and effectively distributes the workload

of these applications.

Chapter 7

Discussion

7.1 Introduction

This chapter discusses the research contributions and answers the research question posited in Chapter 1. Specifically, it explains the research contributions in relation to the research question and objectives. The entire chapter is structured in terms of the major areas of contribution within the research: patterns, cloud infrastructure, novel weighting algorithm, and multi-cloud deployment. Also, it explains the limitations of this research.

The first objective of this research, which is to investigate, analyse and experiment with different deployment, hosting and brokering approaches has been fulfilled by conducting a comprehensive literature review in Chapter 2

7.2 Patterns

One of the contributions of this research is the creation of an improved cloud design pattern structure for cloud properties. In addition, by incorporating the enhanced

cloud design pattern structure, the multi-tenancy patterns catalogue was improved. Also, this research has resulted in a formal description of the multi-tenancy patterns and a novel implementation of the multi-tenancy patterns in different software applications that were deployed on a heterogeneous cloud environment while using various deployment standards. Multi-tenancy patterns are the subject of this research contribution because they are an essential cloud property that determines the success of deploying applications to the cloud (Fehling, Leymann, Retter, Schupeck, et al., 2014) (Fowler, 2002).

7.2.1 Enhanced Cloud Design Pattern Structure

In this research, an enhanced cloud design pattern structure for cloud properties was created because of the gap found in the catalogue description. The structure of the pattern catalogue by Fehling *et al.* (Fehling, Leymann, Retter, Schupeck, et al., 2014) did not describe in detail cloud properties and their challenges. The pattern structure also lacked programming template views, lacked abstraction of patterns, and could not target a wide range of cloud users. Similarly to the work of Fehling *et al.* (Fehling, Leymann, Retter, Schupeck, et al., 2014), Fowler *et al.*(Fowler, 2002) created a cloud pattern catalogue, but this catalogue was a simplified view of cloud properties and the structure was not consistent for each cloud properties. Another similar work to the cloud pattern by Fehling and Fowler, is the research by Wilder *et al.* (Wilder, 2012). Wilder *et al.* (Wilder, 2012) discussed in their book cloud patterns that are useful for architecting cloud-native applications. However, these patterns were not catalogued because the authors argued that each pattern is applicable to multiple architectural concerns, hence these patterns cannot be catalogued.

In some other research works, such as those found in (Walraven et al., 2012; Wang & Casale, 2014), cloud properties are briefly discussed as cloud patterns. However, these works were inadequate in describing cloud properties and their consistency. This challenge of inconsistency in pattern structure, no full description and inability to target varied cloud users, led to the creation of an enhanced cloud

pattern structure in this research. The enhanced cloud pattern structure adopts a widely accepted and highly cited pattern structure by Gamma *et al.* (Gamma et al., 1994). The details of this enhanced cloud pattern structure are fully described in Chapter 4 and Appendix A and it addresses the challenge of inconsistency found in previous cloud pattern structures.

The enhanced cloud pattern structure now benefits from a systematic description of each pattern. Therefore, cloud properties that can be described as patterns can be described using the improved pattern structure because it offers consistent and comprehensive headings to describe cloud properties. This improvement benefits different cloud users so that they can utilise and implement these patterns with ease. Furthermore, the enhanced cloud pattern structure provides a programming template in Java, one of the most popular and widely used programming languages (Ezenwoye, 2018; Gavrilović et al., 2018; Jiang, 2020). It also provides a formal description section that aims to improve clarity for software engineers.

7.2.2 Formal Description of Multi-tenancy Patterns

This research has resulted in a formal description of the multi-tenancy patterns, which in consequence improved the accuracy and efficiency of implementing this pattern in the data layer of data-driven web applications. The data layer is the area of focus because it is the layer that experiences major bottlenecks among the various layers of a software application (Brewer, 2012; Grozev & Buyya, 2014b).

Formal language abstracts the general characteristics of programming language (Linz & Rodger, 2022). Since this research already implemented a programming template of the multi-tenancy pattern, the next intuitive thing to do was to formally describe the pattern. This research used the Z language to create a formal model for multi-tenancy patterns. The choice of language is because at the time the research was conducted, Z language dominated the space (Lana et al., 2019; Saratha et al., 2017). In contrast, other pattern catalogues and cloud property descriptions (Fehling, Leymann, Retter, Schupeck, et al., 2014; Fowler, 2002; Wang & Casale,

2014; Wilder, 2012) generalised the description of the multi-tenancy patterns.

Furthermore, because multi-tenancy patterns can be implemented in various layers of a data-driven web application, there is a need to reduce ambiguity in specifying the use of this pattern. Therefore, this research enhances the capacity of software engineers and architects in understanding and implementing multi-tenancy patterns in the data layer of a web application through formal methods. In addition, the formal description of multi-tenancy patterns enhances the capacity to test software applications whose components make use of multi-tenancy patterns.

7.2.3 Novel Implementation and Validation of Multi-tenancy Pattern

A novel implementation of multi-tenancy patterns in the data layer of case-studied software applications was presented in this study. It further empirically characterised the effect of multi-tenancy patterns in the case-studied applications.

Following the enhancement of the structure of multi-tenancy patterns, the three degrees of multi-tenancy patterns - shared component, tenant-isolated component, and dedicated component patterns - were implemented in the data layer of two case-studied applications. The first case study application is a bespoke BPM system that was transformed into a web application. The application is supported by SimpleDB, a NoSQL database, and was deployed to the AWS cloud. The second case study application is a containerised WordPress application, supported by a MySQL database and hosted on an OpenStack cloud. The three degrees of multi-tenancy patterns were successfully implemented in these case study applications.

The results of the implementation revealed that the dedicated component pattern guaranteed the highest isolation in both case-studied applications and performed best in terms of response time, throughput, and number of executed workloads. However, this is at the cost of resources being used. Furthermore, even though the volume of data and requests had an impact on the degree of isolation required, the

dedicated component still maintained the highest level of isolation.

The tenant-isolated component pattern optimised available resources with minimal impact on performance and isolation. Also, the tenant-isolated pattern performed better than the shared pattern in both case-studied applications. The results recorded from the use of the tenant-isolated pattern showed that this pattern was able to handle most request scenarios with much better response time and minimal errors. The number of error scenarios was minimal and there were no transactional locks at the database layer. However, for maximum security and privacy, it is recommended to pair this pattern with the dedicated component pattern in the data layer of systems that require security.

The shared component pattern maximises resources. However, this is at the cost of performance. In comparison with dedicated and tenant-isolated component patterns, this pattern did not perform as well as the other multi-tenancy patterns. The shared component pattern could not handle as many requests as the other multi-tenancy patterns. This is because it recorded the highest request and database errors with very high response times. Though, in the case study applications, the shared component pattern did better when used in a NoSQL database than when used in a MySQL database. This is partly the result of the fact that the application utilising the NoSQL database is a bespoke application that follows design principles which closely fit the multi-tenancy patterns. Also, the data transfer and storage mechanism in NoSQL provides for faster implementation. Consequently, it is recommended to use shared component patterns in the data layers of applications that does not require high performance but cost-effective in implementation.

In contrast to the existing pattern catalogue (Fehling, Leymann, Retter, Schupeck, et al., 2014; Fowler, 2002; Wang & Casale, 2014; Wilder, 2012), multi-tenancy patterns were empirically evaluated and implemented to characterise their effect on the different components of the web application as described in section 4.5 of Chapter 4. In addition, in contrast to the research work of Wang *et al.* (Wang & Casale, 2014) that conducted an evaluation of multi-tenancy patterns, this research conducted a novel implementation of multi-tenancy patterns in both SQL

and NoSQL scenarios. Also, in contrast to the research work by Vanhove *et al.* (Vanhove et al., 2014), multi-tenancy patterns were compared within the data layer of software applications in this research.

Analysis of the experimental evaluation revealed that the number of tenants, tenants' activities, and size of cloud resources affect the performance of the three degrees of Multi-tenancy patterns. The enhanced pattern structure provides insight into the successful implementation of the pattern. It also reduces the time required to implement the pattern.

In summary, this contribution lays the foundation for the third and fourth objectives of this research. The enhanced cloud design pattern structure and definition of multi-tenancy patterns guided the process of the successful novel implementation of the pattern in the two case-studied applications. Also, the clarity in the description of the pattern revealed that a combination of multi-tenancy and cloud patterns can be used to achieve success in the design and deployment of data-driven web applications on cloud. The multi-tenancy patterns were successfully combined with an architectural pattern (the use of docker) to deploy applications on cloud. Additionally, the dedicated component, shared component, and tenant-isolated component patterns were combined differently to achieve better resource sharing, increased performance, and lower cost of resources for running WordPress applications. This indicates that hybrid patterns can be implemented and contribute to better performance of cloud-deployed web applications.

7.3 Cloud Infrastructure

This research also contributes to the development of bespoke cloud experimental testbeds based on real and heterogeneous cloud infrastructures. To test and evaluate the proposed and existing algorithms, a bespoke private cloud that runs OpenStack was first created. To demonstrate heterogeneous multi-cloud resources, the private cloud was combined with two public clouds.

Located in one data centre, the private cloud consists of six Dell PowerEdge rack servers; four serve as OS servers and two serve as management servers. There are 120 virtual CPUs, 755GB RAM, and just under 3TB of storage available. A node includes one or more Intel Xeon E5-26XX Processor(s) with a speed range of 2.3GHz to 3GHz and a memory range of 64GB to 256GB. In order to distribute and connect Dell blades, a Cisco Nexus 9372 switch provides 20GB access links and 20GB port-channel uplinks to a Nexus 9K switch.

7.3.1 Experimental Environment for Novel Weighting Algorithm

This contribution fulfills the second objective of this research. As explained in Chapter 3, an OpenStack experimental environment that comprised Nova compute service, CINDER storage, Neutron networking, Keystone identity service, glance image service, Senlin clustering service, EC2 API, and Horizon dashboard was configured in this research. The nova-compute service enables the creation of virtual machines, and the experimental environment created eighteen virtual machines. Ten of the virtual machines were configured as application servers which ran Ubuntu 20.1 operating systems, with Apache installed as the web server. Two Virtual machines were used to install the load balancer, and four virtual machines were configured as database servers running MySQL as the database system.

To assess the performance of the novel and benchmark algorithms, the environment was set up with an open-source web application that was supported by databases. User requests were modelled and sent through the use of Apache Jmeter. The proposed algorithm and benchmark algorithm were implemented and run on separate machines as part of the load balancing solution. The data collector which is part of the load balancing solution includes local collector services deployed on each application server to monitor and collect required utilisation data. The experimental environment successfully validated benchmark algorithms depicting their relevance. It was also used to assess the performance of the novel and benchmark algorithms successfully.

The experimental setup in this research is an advancement in evaluating cloud models because it used real cloud infrastructure and novel configurations. This is in contrast to the experimental setup of Sahu *et al.* (Sahu et al., 2013), Devi *et al.* (Devi & Uthariaraj, 2016), and Grozev *et al.* (Grozev & Buyya, 2014b), where cloud properties were modelled using CloudSim simulator. The ability to test cloud algorithms on real infrastructure provides a degree of validity for the algorithm. Therefore, this research's experimental testbed is far more proven in terms of results and real-life expectations.

7.3.2 Experimental Environment for Multi-Cloud Deployment

This research also resulted in the creation of a bespoke multi-cloud experimental testbed. The multi-cloud consists of three heterogeneous cloud environments. An AWS cloud located in Tokyo, DigitalOcean located in New York, and a private OpenStack cloud located in London. The newly engineered multi-cloud experimental testbed was used to evaluate and compare the performance of the novel multi-cloud load balancing algorithm with benchmark algorithms. Each data centre consisted of nine heterogeneous VMs. Each data centre had HAProxy server and the novel load balancing solution deployed in two VMs: one of the VMs acts as a standby, displaying the failover architecture. Apache Web server, which acts as an application server, was installed on five VMs. MySQL database server was deployed on two VMs. In addition, a standard auto-scaler was installed in each data centre. To mimic real user requests and locations, Apache Jmeter–the workload simulator, was set up on an external standalone machine. The standalone machine's configuration is as follows: 4-Core, Intel Core i7, 2.8GHz CPU, and Gigabit Ethernet NIC. An open source stateless web application that was built using the Orchard core framework was deployed on the application servers. This case study application was chosen because it is stateless and meets the prerequisites for three-tier web applications.

The bespoke experimental environment depicts a highly available topology that ensures high performance, reliability, and fault tolerance. Also, the experimental

environment depicts a reliable cloud architecture because of the use of a variety of cloud providers. In this research, a multi-cloud deployment strategy similar to the multi-cloud testbed by Qu *et al.* (Qu et al., 2017) was used. However, in contrast to (Qu et al., 2017), where all data centres were owned by the same cloud provider, this research's experimental environment incorporated data centres from different cloud providers.

Also in contrast to the experimental environment of Grozev *et al.* (Grozev & Buyya, 2014b), where the cloud data centres were models of the same cloud provider created within a simulator, this research's experimental setup used a combination of real infrastructures and different cloud providers.

This experimental environment displays an advanced and improved experimental setup. The results from this testbed and from experiments suggest seamless scalability. Furthermore, the use of private open source cloud infrastructure allows organisations to meet regulatory and compliance requirements and therefore enhance security.

7.4 Novel Weighting Algorithm

An important contribution of this research is the creation of a novel weight assignment algorithm to improve resource provisioning for three-tier applications deployed on the cloud. This contribution consolidates and improves earlier work done on cloud architectural and deployment patterns. To create the algorithm, a technique to estimate the weight of a VM for better workload distribution was introduced in this research. The VM's weight calculation was implemented in a state-of-the-art load balancer software called HAProxy (HAProxy, 2021). In addition, a new architectural component to complement the three-tier architecture was introduced to improve limitations commonly found in load balancing techniques such as single point of failure and quick sensing of uncertainties. The novel architecture also facilitates the migration of existing web applications with

less modification in the codebase of the application. This architectural component ensures that QoS does not diminish when web applications suffer from common challenges such as flash crowds and resource failures. The novel algorithm was evaluated using real cloud infrastructure, and the results were compared and validated with results from standard industry practices and high-cited research outputs. In part, these contributions fulfills the last four objectives of this study.

7.4.1 Key Server Metrics for Load Balancing Algorithm

In this research, a key server metric, called thread count was identified. This server metric noticeably impacts the performance of three-tier web applications. As well as the newly identified server metric, existing key server metrics that directly impact three-tier cloud-hosted applications were also combined in the novel weight assignment algorithm. In contrast to similar research found in (S.-L. Chen et al., 2017; Devi & Uthariaraj, 2016; Grozev & Buyya, 2014b; Sahu et al., 2013) where fewer and inadequate server metrics were used, the novel load balancing algorithm benefits from carefully selected and robust server metrics.

Chen *et al.* (S.-L. Chen et al., 2017) utilised CPU speed, CPU idle rate, and RAM idle size in their load balancing algorithm. Devi *et al.* (Devi & Uthariaraj, 2016) used CPU utilisation and the number of instructions per second to calculate the capacity of a VM. Sahu *et al.* (Sahu et al., 2013) considered CPU, RAM, and Bandwidth utilisation in determining the current load of a host machine. Grozev *et al.* (Grozev & Buyya, 2014b) used CPU, RAM, network utilisation and buffer size to evaluate the capacity of a VM. The novel algorithm shares some similarities with previous work on utilising server metrics. Nevertheless, in contrast, the novel algorithm combined CPU utilisation, RAM utilisation, network bandwidth utilisation, network buffer, and thread count in determining the capacity of a VM/server.

CPU, RAM, and network bandwidth utilisation are popular metrics that indicate a server's efficiency. However, based on experimental monitoring of these metrics,

these metrics do not suffice in determining the capacity of a server when an interactive web application is involved. Therefore, Network Buffer was introduced in this research as recommended by Grozev *et al.* (Grozev & Buyya, 2014b), I also identified and included thread count in the list of metrics to calculate the capacity of a server. By using a network buffer, a load balancing solution can better determine CPU utilisation since a larger buffer size improves CPU utilisation. The introduction of thread count is also vital because this indicates the number of threads and processes running even when utilisation values of CPU, RAM, and Bandwidth are low. Thread count is a reliable indicator of processes that could affect overall performance. By combining these metrics, the novel algorithm has the advantages of being adaptive and reactive at the same time. This enables the algorithm to improve its responsiveness to the distribution of load of the chosen class of applications. The benefits of using the proposed weighting technique to determine the application server that handles user requests include reduced response times, improved throughput, and improved utilisation of resources as shown in Chapter 5.

7.4.2 Novel Weighting Algorithm for Load Balancing in Cloud

This research resulted in the development of a dynamic weighting technique and an adaptive load balancing algorithm that combines carefully selected key server metrics specific to cloud-hosted three-tier web applications. The novel algorithm calculates the weight of a server using the selected server metrics in a proposed mathematical formula described in section 5.4 of Chapter 5. The novel algorithm incorporates the selected metrics to create an adaptive resource provisioning and workload distribution approach that overcomes the limitations of load balancing techniques and the negative effects of flash crowd and resource failures.

On one hand, the newly introduced algorithm combined widely used server metrics like those found in the research work by Grozev *et al.* (Grozev & Buyya, 2014b). On the other hand, contrary to Grozev *et al.* (Grozev & Buyya, 2014b), this algorithm combines popular and newly identified key server metrics through the

use of a novel weighting mathematical method that calculates a server's real-time load. The server's real time load and other threshold parameters are then utilised to assign weights to servers and also in the overall load distribution algorithm.

In contrast to the research work by Chen *et al.* (S.-L. Chen et al., 2017) where the use of priority service value and polling methods determines where requests will be sent, the novel weighting algorithm used the concept of weighted average and bounded numerical values to calculate the weight of servers before requests are sent to servers. However, similar to the research work by Chen *et al.* (S.-L. Chen et al., 2017), the novel algorithm uses a standard weighted round-robin mechanism to distribute requests among servers.

In addition, the novel weighting algorithm considers the real-time capacity of each server at a regular interval and is specifically for three-tier web applications on cloud, unlike the algorithm proposed by Devi *et al.* (Devi & Uthariaraj, 2016) which targets Non-preemptive dependent tasks.

7.4.3 Overall Design Architecture of Proposed Load Balancing Solution

This contribution fulfils the third objective of this research. In this research, a redesigned architecture that addresses the issues and limitations of common load-balancing techniques, as discussed in the research output (Adewojo & Bass, 2022b) was created. This adds a layer, an architectural component framework, to the three-tier architecture.

This approach does not modify the three-tier pattern but introduces additional components that manage resource provisioning within the cloud environment. As a result of this new architectural component, existing applications can be migrated to the cloud with little modification to the codebase, while new three-tier applications can be developed leveraging existing architectural frameworks and existing technologies. This architecture also improves

1. Scalability through the use of a very fast and reliable reverse-proxy load balancer
2. Fault tolerance through the use of multiple load-balancing front ends and standby application servers as fail-over systems, this, combined with the use of keep alive technology of HAProxy
3. Reduced overhead and latency through the use of an efficient algorithm that runs with negligible performance overhead with measurement parameters as recommended by (Shah et al., 2017; Zomaya & Teh, 2001) and
4. Performance through the use of carefully selected server metrics that determines the efficiency of a load balancing algorithm.

7.4.4 Empirical Evaluation of Algorithms

This contribution partly fulfils the fifth objective of this research. An empirical evaluation of the novel algorithm using the proposed private OpenStack environment was done. The evaluation compared, validated, and contrasted the novel algorithm with a standard load balancing algorithm and a baseline algorithm by Grozev *et al.* (Grozev & Buyya, 2014b). The experimental results were the average of five repeated experiments over a 24-hour period. This is in contrast with a lot of research work on cloud algorithms and models (Bambrik, 2020; Byrne et al., 2017; Fakhfakh et al., 2017; Grozev & Buyya, 2014b; Makaratzis et al., 2018; Sahu et al., 2013; H. Zhang et al., 2017) that were evaluated in a simulation environment. Simulation experiments rely heavily on parameters to be accurate, so there is a challenge in choosing accurate parameters and parameter values. As a result, if the parameters are not right an incorrect simulation result is inevitable.

Furthermore, the simulator tools are not exhaustive because each is designed to address a specific cloud process (Bambrik, 2020). Hence evaluating a complex cloud algorithm such as the one found in this research in a simulation environment will either require a modification to the simulator or a combination of different

cloud simulators. There is also a danger of losing time because the use of any cloud simulator tool requires an understanding of the programming language employed. Therefore, time can be lost in the process of getting up to speed in learning a new programming language, when the researcher could have selected a known and agnostic language to build their algorithm which will run on most cloud platforms. Furthermore, not all cloud simulation tools are open source, commonly advanced simulators are expensive, and this defeats the claim that using simulators are financially cheaper when compared to real cloud infrastructure (Bambrik, 2020; Fakhfakh et al., 2017). Therefore, a real cloud infrastructure environment is more suitable for evaluating cloud algorithms. This is because it provides the flexibility of using real cloud resources and produces results that can be relied upon.

Benchmark algorithms were first validated to ascertain the efficiency of the environment and similar parameters and metrics were evaluated for the overall experiments. In contrast to research work by Qu *et al.* (Qu et al., 2017) and Sahu *et al.* (Sahu et al., 2013), there were no validation of benchmark algorithms. This validation further proves the results of the newly developed weighting technique and load balancing algorithm as presented in Chapter 5. In addition, the OpenStack environment enabled extensive experimental characterisation of the metrics used in the proposed weighting technique. This was because there was no restriction on the number of resources that could be used. Furthermore, the behaviour of the metrics was effectively characterised because the environment is a replica of what will be found in a public cloud.

7.4.5 Architectural Component Collaboration and Performance

The novel algorithm and redesigned architecture complement and work cooperatively with existing and state-of-the-art auto-scaling mechanisms in cloud data centres so improving the performance of the application that relies on it. This is in contrast to research work found in (S.-L. Chen et al., 2017; Devi & Uthariaraj, 2016; Grozev & Buyya, 2014b; Sahu et al., 2013; Wang & Casale, 2014) where autoscaling mechanisms were specifically designed for the research. Addition-

ally, it works cooperatively with the existing layers of the three-tier infrastructure with negligible overhead as described in Chapter 5. Moreover, the architecture of the component influences the load balancing process, ensuring that applications' performance is optimised and reliable regardless of challenges they may face.

Experimental results show that our approach improves response time, session delay, and throughput over the baseline. This was achieved because the proposed novel algorithm computes the weight of an application server based on its current utilisation and capacity. Using this approach, the architecture can detect changes quickly and adapt to improve the performance of deployed applications, especially when they are affected by flash crowds and resource failures. This approach combines these capabilities to create a multi-faceted defence against performance problems in cloud-hosted web applications.

7.4.6 Extensible Feature of Weighting Algorithm

The novel load distribution algorithm is easily extensible to work for multi-cloud configurations. This was achieved because the proposed approach to creating the novel algorithm incorporated the SOLID principles of programming and various aspects of load balancing, which is agnostic of the number of clouds it is being used on. This is an adaptive concept that was built into the algorithm so that it can evolve and be relevant to a new concept. The algorithm incorporated server metrics that can be found in all application servers. The mathematics behind the weight calculation can be implemented, deployed, and run on any web server. This is because the language used in developing the algorithm is platform-independent. Furthermore, the ability to include new functionality and extend existing functions was built into the algorithm design and implementation. These built-in features are unlike algorithms designed specifically for either simple cloud services such as found in (Devi & Uthariaraj, 2016; Wang & Casale, 2014) and cloud systems such as the research work of Chen *et al.* (S.-L. Chen et al., 2017).

Overall, this research resulted in the creation of a novel weighting and load balanc-

ing algorithm to improve resource provisioning, workload distribution, and cloud deployment across a single cloud. Redesigned architectural components were introduced to distribute workloads of a cloud-deployed three-tier web application. Additionally, the architecture and algorithm can be extended to accommodate improved deployment methodologies.

It is pertinent to note that the majority of organisations have legislative requirements and also the need to leverage the different services offered by different cloud providers to achieve economies of scale. As a result, multiple clouds can enhance the quality of services offered by the organisation. These organisations can overcome challenges such as vendor lock-in, unavailability, and so on. Therefore, the last overall contribution of this research proposed a multi-cloud deployment pattern and a multi-cloud workload distribution algorithm to address resource provisioning and workload distribution problems, especially when applications encounter flash crowds and resource failures.

7.5 Multi-Cloud Deployments

Lastly, this research makes a contribution to multiple clouds. A novel approach to generalising the deployment of three-tier web applications across multiple clouds is presented in this study. This contribution improves previous work on resource provisioning within a single cloud. It also improves the work done in combating flash crowds and resource failures in cloud-based web applications. In this research, a multi-cloud load distribution algorithm was developed to optimise application performance across multiple clouds using a dynamic and reactive approach. Using the proposed approach, features common to IaaS providers are implemented, which do not rely on interoperability capabilities between participating providers. The novel multi-cloud algorithm was evaluated on two public clouds and one private cloud.

The experimental evaluation used realistic workloads to imitate flash crowds and

resource failures. The results of the experiments show that the proposed approach improves application performance in both flash crowd and resource failure scenarios. The proposed solution demonstrates that multi-cloud not only addresses flash crowds and resource failure issues but also lets organisations exploit multiple cloud resources even at competitive rates. It also improves reliability, helps companies stay within regulatory rules, and helps them improve their digital footprint in terms of the number and variety of clients who can access and use the company's software applications.

These contributions combined with the novel weighting algorithm's contributions completely fulfill the objectives of this research through the contributions below.

7.5.1 Multi-Cloud Load Distribution Algorithm and Deployment Architecture

In this study, a novel adaptive and reactive algorithm was developed to distribute workload across multi-clouds, an extension of the novel weighting algorithm in the previous study of this research. The newly developed multi-cloud load distribution algorithm includes the calculation of network latency between data centres, the weight of application servers in each data centre, and the weight of each participating data centre. First, the algorithm determines which data centre will serve users and, secondly, which application server will handle their workload. The algorithm combines the use of carefully selected server metrics (CPU, RAM, Bandwidth utilisation, Network buffer, and thread count) to determine the weight of application servers and carefully selected data centre metrics including latency, geographical location, Round Trip Time, and capacity to determine the weight of a data centre.

Unlike popular load distribution strategies such as AWS Elastic Load Balancer (ELB) (Amazon, 2021c) and Azure load balancer (Azure, 2021b) where requests are distributed within a data centre or multiple data centres of the same provider, the novel multi-cloud algorithm distributes requests to multiple non-related inde-

pendent data centres using an adaptive approach.

Also, unlike the research conducted by Gandhi *et al.* (Gandhi et al., 2014) and Junior *et al.* (de Paula Junior et al., 2015) where their algorithm reactively provisions resources after it detects increased incoming requests, this algorithm proactively and reactively distributes requests to avoid the challenges of flash crowds and resource failures.

Further, this algorithm utilises not only common server metrics such as those found in (Grozev & Buyya, 2014b) and (Qu et al., 2017), but a combination of proven server and data centre metrics that affect interactive web applications. Moreover, the algorithm ensures that imperative constraints are met without sacrificing quality of service.

In addition, this algorithm and deployment architecture ensures that the negative effects of flash crowd and resource failures are mitigated. The algorithm utilises the combination of HAProxy's ability and the novel algorithm to effectively distribute workload across multiple clouds while ensuring latency requirements, and legislative requirements are considered. The deployment architecture also displays a decentralised cloud deployment pattern for hosting web applications such that vendor lock-in and other single cloud hosting issues are mitigated.

7.5.2 Design Approach for Three-Tier Multi-Cloud Web Application

A novel design approach that does not modify the three-tier architectural pattern is implemented in this research: instead, it augments the existing architectural pattern. As a result of this approach, existing and new web applications can be deployed on the cloud, benefiting from the improved design.

In similarity to the work done by Qu *et al.* (Qu et al., 2016; Qu et al., 2017) where focus was on mitigating challenges common to web applications, the novel

design approach addressed challenges common to web applications. However, in contrast to their research work where they focused on web applications generally, the focus of this design approach is on distributing the workload of three-tier web applications such that performance is enhanced irrespective of the common challenges that may arise. Furthermore, this novel design approach requires session continuity and data locality in order to provide high availability and reliability since multiple data centres can handle the same workload request.

In contrast to the research work by Grozev *et al.* (Grozev & Buyya, 2014b) where the focus was on load balancing and autoscaling using a similar design approach to this research's approach, this research focused primarily on load distribution within multi-cloud while the autoscaling feature was left to be handled by matured and widely used technologies; this makes this approach robust and able to complement state-of-the-art autoscaling tools and techniques.

7.5.3 Enhanced Multi-Cloud Architecture and Communication Protocol

Another result of this research is a novel decentralised multi-cloud architecture composed of the proposed enhanced three-tier architectural pattern. Unlike Qu *et al.* (Qu et al., 2017) where a decentralised approach with load balancing agents was used, load balancing agents were not used in this research to communicate among the decentralised centres. This is because one of the aims of this research's contribution was to reduce the number of network broadcasts. Furthermore, this proposed architecture and load balancing solution do not need to wait for an overload to occur before reacting, unlike the research work by Qu *et al.* (Qu et al., 2017), where the aim is to react when there are flash crowds, overload, or resource failure.

The novel architecture is a further step toward combating flash crowds. Flash crowds refer to legitimate, rapid, and increased user requests that can cause performance degradation in cloud applications. The multi-cloud deployment pattern and

a unique adaptive reactive multi-cloud load distribution algorithm were combined to mitigate this challenge in this research. This solution uses the novel multi-cloud algorithm to determine the weight of data centres and application servers. The load balancing algorithm then builds on the weighting system to redirect user requests to suitable data centres when overload as a result of flash crowds occurs. The distribution algorithm dynamically changes to reflect the current situation of the system. The sequence of redistributing requests to other data centres also complements autoscalers because autoscalers can efficiently provision or decommission resources without causing performance degradation. This is different from common and popular methodologies such as those found in (Amazon, 2021c; Azure, 2021a; Qu et al., 2016) where the autoscaler is heavily involved in creating new resources when issues with the application or underlying resources occur.

Moreover, the enhanced architecture and load balancing solution prevent resource failures. Resource failures occur without notice, which can cause chaos in applications dependent on the resource. Furthermore, performance degradation in web applications using the resource is glaring when the loss of resources is beyond the capacity of the locally unused resource. In order to address this issue, an enhanced architecture and multi-cloud deployment pattern to increase the availability and accessibility of unused resources were used in this research. Also, the solution implements periodic health checks to detect all types of failures. This contrasts with the research of Qu *et al.* (Qu et al., 2017), where hardware failures were the area of focus. In addition, the load balancing service dynamically recalculates the weights of VMs and data centres leading to an improved multi-cloud load distribution strategy.

7.6 Limitations of Research

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 (Cash et al., 2016; Yin, 2014).

Firstly, in this research, potential variables and their measures were identified through the review of similar literature on research work. This was followed by a careful selection of widely cited variables and their units of measurement from research work such as (S.-L. Chen et al., 2017; Fehling, Leymann, Retter, Schupeck, et al., 2014; Grozev & Buyya, 2014b, 2015; Qu et al., 2017). 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 (Bhandari, 2020; Cash et al., 2016; Yin, 2014). To create strong internal validity for this research study, the appropriateness of the variables were considered before deciding to use them. Furthermore, following Yin's (Yin, 2014) guidelines, convergence among multiple and different sources of information on variables used were looked for.

As part of this research, design patterns were implemented in the data layer of software applications. To characterise these patterns, this research study examined factors that affected the speed at which data was written and read from databases. It also examined the security of data and the response time for users. Additionally, the research on resource provisioning and load distribution within single and multiple clouds utilised and assessed variables that were specific to the number of requests attended to, time required to respond to requests, load on servers, and so on. The use of standard tools and data measuring software applications guarantees a consistent procedure for carrying out experiments. These tools were also automated to reduce the risk of inconsistency in implementation. Therefore, the same set of data was used to characterise the effect of the algorithms tested. Because of this, there is assurance that the effects of tuning the parameters are not influenced by external factors. The use of standard and widely used software

applications to collect numerical data also increases confidence in the accuracy of the measurements.

To minimise the effect of maturation threat (Bhandari, 2020), experiments were carried out multiple times over a minimum of twenty-four hours in each case. In this study, the average of each experimental result was primarily used. As a result, the experimental design used in this study was standardised; a powerful method for controlling threats to internal validity.

External validity refers to how much the experiment affects the measured variable (Cash et al., 2016). It is the extent to which the findings of a study can be generalised to other situations or measures (Baldwin, 2018; Cash et al., 2016). This research ensured external validity through theoretical and experimental replications of findings using different deployment platforms.

Furthermore, this study focused on the use of cloud architectural patterns to alleviate the challenges of brokering, deploying, and hosting data-driven web applications to single and multiple clouds. Therefore, the situation effect poses a threat to the external validity of the research's experimental design, inputs, and outputs. The situation effect includes factors such as setting, time of day, location, and so on that limit the generalisability of findings.

Finally, different stages and the results of this research contribute to different scenarios. The factors and generalisability of the research output are discussed below.

1. This study restructured the cloud design pattern catalogue by Fehling *et al.* (Fehling, Leymann, Retter, Schupeck, et al., 2014) to create a systematic and improved pattern structure. The enhanced pattern structure is applicable to any cloud solution that can be described as a pattern. In addition, the implementation and the associated algorithms presented in this research primarily apply to multi-tenancy patterns.
2. This study focused on existing bespoke business process modelling software

applications and web applications that implement the three-tier deployment architecture. Therefore, other types of business applications and cloud-hosted services are not within the scope of this study. So, the findings of this study do not apply to all applications and cloud-hosted services.

3. The resource provisioning and load distribution algorithm apply to three-tier web applications: therefore, the behaviour of the algorithm is uncertain when used in other cloud services.

4. The number and size of user requests used in the experimental evaluations were within the limits of a private cloud, the free tier of a public cloud and the resources available to use. Therefore, the results of this study should not be generalised to extremely large public clouds.

5. The use of a real and diverse 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, while cloud varieties are representative of location neutrality.

7.7 Summary

This chapter discusses the objectives and contributions of this research. There are four main contributions. Each subsection of the contributions addresses the research objectives as stated in Chapter 1.

Firstly, an improved cloud pattern structure and catalogue was developed in this research and applied to multi-tenancy patterns. A formal and UML description of the multi-tenancy pattern was created. A novel implementation of the multi-tenancy patterns was created in the data layer of three case study applications. Each of the multi-tenancy patterns was experimentally characterised to validate their behaviour in web applications deployed on the cloud.

Secondly, novel experimental testbeds were configured using a private and public

cloud setup.

Thirdly, a novel weight assignment load-balancing algorithm for three-tier web applications deployed on the cloud was created. As part of this research, a key server metric that contributes to the load distribution of three-tier applications on the cloud was also identified and used. It also introduced a unique architectural component for brokering activities for the newly developed load-balancing solution.

Finally, a multi-cloud load distribution algorithm to combat flash crowds and resource failures in three-tier web applications on the cloud was created in this research. In addition, a novel approach to generalise the deployment of three-tier web applications across multiple clouds was introduced in this research. This approach complements previous work on load distribution in a single cloud and improves the interoperability and reliability of applications using multiple clouds.

The chapter concludes with a discussion of the scope and limitations of the study. The limitations of the research were discussed in light of internal and external validity. The research was conducted using an experimental research approach. An in-depth analysis of literature on similar works, an analysis of case studies and the creation of a prototype of the same application were all part of the research design. Experiments were also conducted to characterise and validate cloud patterns and novel algorithms developed during the research.

Chapter 8

Conclusions and Future Direction

8.1 Thesis Summary

Cloud computing has emerged as a paradigm shift in information technology infrastructure. It offers scalable and flexible computing resources to individuals and organisations on demand through the Internet. Cloud implements the reality of computing as a utility, and many organisations are taking advantage of it. This makes the service of renting a dynamic pool of computing resources a better alternative to procuring and maintaining in-house infrastructures for many organisations. Renting IT resources as a service frees organisations from heavy administrative duties. It allows them to focus on their core businesses, their application deployment options, to reach more customers, and to respond promptly to dynamic and unpredictable workloads.

Despite the benefits of using the cloud, deploying existing and newly created web applications on the cloud is not straightforward. There are several methodologies that have been proposed to deploy these applications on cloud. There are also challenges such as the inability to use and share cloud resources effectively, service unavailability due to flash crowds and resource failures, regulatory compliance is-

sues, vendor lock-in, network latency between end users and cloud sites, exorbitant billing, and lack of varied cloud resources.

These challenges are of special importance for business applications, especially web-based applications serving global users. In order to alleviate the problems mentioned above, design patterns play a crucial role in directing software architects on how to create reliable and efficient software applications. It removes ambiguity and complexity in the interpretation of the solution proposed. Aside from the role that design patterns play, the three-tier architectural pattern is the predominant design approach for building most of these business applications. To combat the aforementioned challenges, the use of appropriate deployment patterns, approaches, and architecture setups that are tailored to the architecture behind these applications is of the utmost importance. As a result, the use of single and multi-cloud deployment patterns in conjunction with dynamic load distribution mechanisms is deemed a viable approach to solving the aforementioned challenges.

Some gaps in the overall and relevant literature were filled in this research. The design process of this research combined an exploratory study, case study application selection, and experimental research approach to solve the challenges associated with designing, deploying and hosting web applications on cloud. An enhanced cloud design pattern structure that reduces the ambiguity in the description of cloud patterns has been created. This cloud design pattern structure was adapted from a widely accepted design pattern catalogue. The enhanced cloud pattern structure was applied to multi-tenancy patterns to create an improved description of multi-tenancy patterns. A key cloud deployment pattern, multi-tenancy patterns, was examined and empirically evaluated in this study. Multi-tenancy pattern was implemented in a business process modelling application and a WordPress application.

The main contribution of this research is the creation of a novel weight assignment load distribution architecture and algorithm that combines five carefully selected server metrics to efficiently distribute the workload of three-tier web applications among virtual machines. This newly introduced load distribution approach allevi-

ates common challenges of cloud-hosted web applications such as flash crowds and resource failures. To verify the applicability of this novel approach, the load distribution technique was experimentally evaluated on a real cloud infrastructure, and the performance was benchmarked with two other load distribution techniques. A three-tier open-source e-commerce application was used to test the algorithms. A private cloud that runs OpenStack was used in this research to configure an experimental platform to facilitate experimental evaluation.

Finally, in this research, a novel decentralised multi-cloud architecture and algorithm to combat flash crowds and resource failures were created. This algorithm distributes the workload of three-tier web applications using geographical load distribution techniques and an improved load distribution algorithm. This architecture alleviates common load balancing and web application challenges such as compliance issues, vendor lock-in, network latency between end users and cloud sites, exorbitant billing, and lack of varied cloud resources. This newly introduced multi-cloud model was evaluated on a multi-cloud infrastructure that comprised three heterogeneous cloud data centres. The multi-cloud algorithm was applied to a three-tier open-source e-commerce application.

The performance of the novel cloud algorithms was compared by simulating resource failures and flash crowds, while carefully measuring response times, throughput, and number of errors. Experimental evaluation of these cloud algorithms showed a significant improvement in response times, number of errors, and throughput. These experimental results show that the novel algorithms and architecture can ensure efficient hosting and deployment of three-tier web applications on cloud.

8.2 Key Research Contributions

As earlier stated in Chapter 1, the aim of this research is to create novel cloud load distribution management and deployment techniques, which can alleviate the

challenges of hosting web applications on cloud.

Chapter 7 described how the objectives of this research were met which has resulted in the following key contributions.

1. **An enhanced cloud pattern structure, enhanced multi-tenancy patterns structure and a novel implementation of multi-tenancy patterns (Chapter 4).** These contributions are linked to these research outputs (Adewojo, Bass, & Allison, 2015; Adewojo, Bass, Allison, & Hui, 2015; Adewojo & Bass, 2018). This output contributes to fulfilling the constraint **stringent reliability of cloud-native web applications** because design pattern is the foundation of a stable and robust application. If software architects and developers utilise design patterns and standard procedures in building software applications, they can build cloud-native applications to better utilise cloud properties.

2. **A novel weight assignment load balancing algorithm to mitigate the negative effects of flash crowd and resource failures and limitations of load balancing techniques (Chapter 5).** This also includes a novel redesigned architectural brokering component for three-tier web applications on cloud. These contributions are linked to these research outputs (Adewojo & Bass, 2022b, 2023). This output contributes to **minimised changes to existing applications that will be migrated to the cloud**. The redesigned architecture does not modify the three-tier architecture; rather, it is an additional architectural component. This additional architectural component acts as a broker among existing architectural layers and so existing applications can be migrated with little to no modification to the codebase. The novel load-balancing algorithm introduces a new server metric and a unique method for selecting application servers for load distribution. This novel approach is a robust approach that takes into consideration the common symptoms of a challenged cloud-hosted web application. This contribution fulfils the constraint **maintain quality of service in flash crowds and resource failures.**

3. **A novel experimental testbed using real cloud infrastructures.** Cloud experimental test beds were created using both private and public clouds to characterize and evaluate the proposed and existing algorithms. By using these experimental testbeds, algorithms could be tested using real-world settings.

4. **Multi-cloud load distribution algorithm for three-tier applications deployed across multiple cloud. This also includes a highly available design approach for this class of applications and enhanced multi-cloud architecture that combats flash crowds and resource failures (Chapter 6).** These contributions are linked to this research output (Adewojo & Bass, 2022a). This novel multi-cloud algorithm also fulfils the constraint **maintain quality of service in flash crowds and resource failures**. This is achieved by the ability to reroute workload to another data centre when a data centre is negatively affected. Also, the algorithm makes the most efficient use of the available server VM of each data centre, thus ensuring effective distribution of workload. The novel architecture and algorithm fulfills the constraint **end users are served at the nearest geographical location**. This is done by checking the location of the user's IP and the latency between cloud data centre and the user. This further helps to experience quick response time and to adhere to legislative rules where one exists with respect to how data or user details are transferred across data centres.

The conclusion of this research is that load balancing algorithms should be designed dynamically because user requests change dynamically. To improve the performance of cloud-native applications, maximise resource utilisation, and address the negative effects of flash crowds and resource failures on cloud-native applications, load balancing algorithms should incorporate comprehensive application-specific server metrics. Additionally, this research reinforces the importance of the use of design patterns and a clearly structured design pattern catalogue in building a robust cloud-native application. When developers understand how design patterns should be used, software applications will be built to standard, and cloud-native applications will feature cloud properties that can make them stable in light of

workload and resource changes. The correct use of cloud design patterns will result in cloud-native applications that are built to standard and that feature cloud properties that are necessary for a robust cloud-native application.

8.3 Future Work

In this research, deployment and hosting of web applications in the cloud were examined and novel deployment and load distribution techniques within single and multiple clouds were proposed. Nevertheless, there is scope for future work, as detailed below.

8.3.1 Generalisation to Multi-tier Applications and Serverless Deployments

While the three-tier architecture is a common and well-established architecture to build and deliver business and interactive systems, technology has improved. Microservice architecture is now becoming the de-facto standard for large-scale and cloud-native commercial applications. Therefore, it is highly recommended that resource management approaches for multi-tier architectures, microservice architectures, as well as the use of serverless computing for deployment approaches be investigated.

8.3.2 Creation of Realistic Workload Model or Dataset for Web Applications

It is difficult to get a ready-made realistic workload model. Furthermore, it is even more difficult to get a sample web request dataset. The majority of datasets and their workload models are outdated and not fit for purpose. It is recommended

that researchers focus on tools that can create realistic datasets and provide varied workload models that fit typical interactive web applications. Furthermore, it is recommended to have an open-source project for this idea because the dataset and workload models need to be regularly updated to reflect changes in technology.

8.3.3 Metrics Measurement

The use of a variety of variables for measuring the performance of cloud algorithms is recommended. Variables such as those mentioned in (Buyya et al., 2018; Govind & González–Vélez, 2021) are highly recommended.

References

Abdullah, M., & Othman, M. (2013). Cost-based multi-qos job scheduling using divisible load theory in cloud computing [2013 International Conference on Computational Science]. *Procedia Computer Science*, *18*, 928–935.

Adewojo, A. A., Bass, J. M., & Allison, I. K. (2015). Enhanced cloud patterns: A case study of multi-tenancy patterns. *2015 International Conference on Information Society (i-Society)*, 53–58.

Adewojo, A. A., Bass, J. M., Allison, I. K., & Hui, K. (2015). Cloud deployment patterns: Migrating a database driven application to the cloud using design patterns. *Proceedings of the World Congress on Engineering and Computer Science*, *1*, 198–203.

Adewojo, A. A., & Bass, J. M. (2018). Evaluating the effect of multi-tenancy patterns in containerized cloud-hosted content management system. *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 278–282.

Adewojo, A. A., & Bass, J. M. (2022a). Multi-cloud load distribution for three-tier applications. *CLOSER 2022: 12th International Conference on Cloud Computing and Services Science*, 296–304.

Adewojo, A. A., & Bass, J. M. (2022b). A novel weight-assignment load balancing algorithm for cloud applications. *CLOSER 2022: 12th International Conference on Cloud Computing and Services Science*, 86–96.

Adewojo, A. A., & Bass, J. M. (2023). A novel weight-assignment load balancing algorithm for cloud applications. *SN Computer Science*, *4*(3), 270.

Akintoye, S. B., & Bagula, A. (2019). Improving quality-of-service in cloud/fog computing through efficient resource allocation. *Sensors (Basel)*, *19*(6), 1267.

Alankar, B., Sharma, G., Kaur, H., Valverde, R., & Chang, V. (2020). Experimental setup for investigating the efficient load balancing algorithms on virtual cloud. *Sensors*, *20*(24).

Alexander, C. (1977). *A pattern language: Towns, buildings, construction*. Oxford university press.

Alexander, C. et al. (1979). *The timeless way of building* (Vol. 1). New york: Oxford university press.

Ali, S., Hafeez, Y., Jhanjhi, N. Z., Humayun, M., Imran, M., Nayyar, A., Singh, S., & Ra, I.-H. (2020). Towards pattern-based change verification framework for cloud-enabled healthcare component-based. *IEEE Access*, *8*, 148007–148020.

Ali-Eldin, A., Seleznjev, O., Sjöstedt-de Luna, S., Tordsson, J., & Elmroth, E. (2014). Measuring cloud workload burstiness. *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, 566–572.

Amazon. (2021a). *Amazon route 53*. Retrieved December 6, 2021, from https://aws.amazon.com/route53/

Amazon. (2021b). *Aws serverless multi-tier architectures with amazon api gateway and aws lambda aws whitepaper*. Retrieved January 1, 2021, from https://docs.aws.amazon.com/whitepapers/latest/serverless-multi-tier-architectures-api-gateway-lambda/three-tier-architecture-overview.html

Amazon. (2021c). *Elastic load balancing*. Retrieved January 1, 2021, from https://aws.amazon.com/elasticloadbalancing/

Amazon. (2000-2021). *Elastic load balancing documentation*. Retrieved January 2, 2021, from https://docs.aws.amazon.com/elasticloadbalancing/index.html

Andrikopoulos, V., Song, Z., & Leymann, F. (2013). Supporting the migration of applications to the cloud through a decision support system. *2013 IEEE Sixth International Conference on Cloud Computing*, 565–572.

Archiveddocs. (2022a). *Chapter 18: Communication and Messaging*. Retrieved August 16, 2022, from https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658118(v=pandp.10)

Archiveddocs. (2022b). *Chapter 19: Physical Tiers and Deployment*. Retrieved August 15, 2022, from https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658120(v=pandp.10)

Ardagna, D., Di Nitto, E., Casale, G., Petcu, D., Mohagheghi, P., Mosser, S., Matthews, P., Gericke, A., Ballagny, C., D'Andria, F., Nechifor, C.-S., & Sheridan, C. (2012). Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds. *Proceedings of the 4th International Workshop on Modeling in Software Engineering*, 50–56.

Ari, I., Hong, B., Miller, E. L., Brandt, S. A., & Long, D. D. (2003). Managing flash crowds on the internet. *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003.*, 246–249.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A., Stoica, I., & Zaharia, M. (2010). *Above the clouds: A berkeley view of cloud computing* (tech. rep.). EECS Department, University of California, Berkeley.

Azure, M. (2021a). *Azure autoscale — microsoft azure*. Retrieved January 10, 2021, from https://azure.microsoft.com/en-us/features/autoscale/

Azure, M. (2021b). *Load balancer documentation*. Retrieved September 10, 2021, from https://docs.microsoft.com/en-gb/azure/load-balancer/

Bahga, A., Madisetti, V. K. et al. (2011). Synthetic workload generation for cloud computing applications. *Journal of Software Engineering and Applications*, *4*(07), 396.

Bala, A., & Chana, I. (2012). Fault tolerance-challenges, techniques and implementation in cloud computing. *International Journal of Computer Science Issues (IJCSI)*, *9*(1), 288.

Baldwin, L. (2018). *Research concepts for the practitioner of educational leadership*. Brill.

Baltes, S., & Ralph, P. (2022). Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering*, *27*(4), 1–31.

Bambrik, I. (2020). A survey on cloud computing simulation and modeling. *SN Computer Science*, *1*(5), 1–34.

Bass, L., Clements, P., & Kazman, R. (2012). *Software architecture in practice*. Pearson Education.

Beaulah Soundarabai, P., Thriveni, J., Venugopal, K., & Patnaik, L. M. (2012). Comparative study on load balancing techniques in distributed systems. *nternational Journal of Information Technology and Knowledge Management*, *6*(1), 53–60.

Beck, K. (1997). *Smalltalk best practice patterns. volume 1: Coding*. Prentice Hall, Englewood Cliffs, NJ.

Bernstein, P. A., & Newcomer, E. (2009). Chapter 3 - transaction processing application architecture. In P. A. Bernstein & E. Newcomer (Eds.), *Principles of transaction processing (second edition)* (Second Edition, pp. 73–97). Morgan Kaufmann.

Bhandari, P. (2020). *Internal Validity in Research — Definition, Threats & Examples*. Retrieved October 24, 2022, from https://www.scribbr.com/methodology/internal-validity/

Bharadwaj, V., Ghose, D., & Robertazzi, T. G. (2003). Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, *6*, 7–17.

Brewer, E. (2012). Cap twelve years later: How the" rules" have changed. *Computer*, *45*(2), 23–29.

Brookshear, J. G. (1989). *Theory of computation: Formal languages, automata, and complexity*. Benjamin-Cummings Publishing Co., Inc.

Buyya, R., Broberg, J., & Goscinski, A. M. (2010). *Cloud computing: Principles and paradigms* (Vol. 87). John Wiley & Sons.

Buyya, R., Srirama, S. N., Casale, G., Calheiros, R., Simmhan, Y., Varghese, B., Gelenbe, E., Javadi, B., Vaquero, L. M., Netto, M. A., et al. (2018). A manifesto for future generation cloud computing: Research directions for the next decade. *ACM computing surveys (CSUR)*, *51*(5), 1–38.

Byrne, J., Svorobej, S., Giannoutakis, K. M., Tzovaras, D., Byrne, P. J., Östberg, P.-O., Gourinovitch, A., & Lynn, T. (2017). A review of cloud computing simulation platforms and related environments. *International Conference on Cloud Computing and Services Science*, *2*, 679–691.

Cai, Z., Zhao, L., Wang, X., Yang, X., Qin, J., & Yin, K. (2015). A pattern-based code transformation approach for cloud application migration. *2015 IEEE 8th International Conference on Cloud Computing*, 33–40.

Cardoso, A., Moreira, F., & Simões, P. (2014). A survey of cloud computing migration issues and frameworks. In Á. Rocha, A. M. Correia, F. . B. Tan, & K. . A. Stroetmann (Eds.), *New perspectives in information systems and technologies, volume 1* (pp. 161–170). Springer International Publishing.

Cash, P., Stanković, T., & Štorga, M. (2016). *Experimental design research: Approaches, perspectives, applications*. Springer International Publishing.

Chen, S.-L., Chen, Y.-Y., & Kuo, S.-H. (2017). Clb: A novel load balancing architecture and algorithm for cloud services. *Computers & Electrical Engineering*, *58*, 154–160.

Chen, Z., Zhang, H., Yan, J., & Zhang, Y. (2018). Implementation and research of load balancing service on cloud computing platform in ipv6 network environment. *Proceedings of the 2nd International Conference on Telecommunications and Communication Engineering*, 220–224.

Chlebus, E., & Brazier, J. (2007). Nonstationary poisson modeling of web browsing session arrivals. *Information Processing Letters*, *102*(5), 187–190.

Cloud, S. (2017). *Simple cloud api*. Retrieved October 24, 2017, from http://simplecloud.org/

Cloud adoption to accelerate IT modernization — McKinsey. (2022). Retrieved July 20, 2022, from https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/cloud-adoption-to-accelerate-it-modernization

Coad, P. (1992). Object-oriented patterns. *Commun. ACM*, *35*(9), 152–159.

Cretella, G., & Di Martino, B. (2014). An overview of approaches for the migration of applications to the cloud. In L. Caporarello, B. Di Martino, & M. Martinez (Eds.), *Smart organizations and smart artifacts* (pp. 67–75). Springer International Publishing.

Cruz, E. H., Diener, M., Pilla, L. L., & Navaux, P. O. (2019). Eagermap: A task mapping algorithm to improve communication and load balancing in clusters of multicore systems. *ACM Transactions on Parallel Computing (TOPC)*, *5*(4), 1–24.

Cunningham, W., & Beck, K. (1989). Constructing abstractions for object-oriented applications. *Journal of Object-Oriented Programming*, *2*(2), 17–19.

De Lauretis, L. (2019). From monolithic architecture to microservices architecture. *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 93–96.

de Paula Junior, U., Drummond, L. M., de Oliveira, D., Frota, Y., & Barbosa, V. C. (2015). Handling flash-crowd events to improve the performance of web applications. *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 769–774.

Devi, D. C., & Uthariaraj, V. R. (2016). Load balancing in cloud computing environment using improved weighted round robin algorithm for nonpreemptive dependent tasks. *The scientific world journal*, *2016*.

Docker. (2017). *What is docker*. Retrieved October 24, 2017, from https://www.docker.com/what-docker

Elgedawy, I. (2015). Sultan: A composite data consistency approach for saas multi-cloud deployment. *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, 122–131.

Elmroth, E. (2022). *"15 years of cloud control"*. Retrieved April 27, 2022, from https://closer.scitevents.org/PreviousInvitedSpeakers.aspx

Erl, T., Cope, R., & Naserpour, A. (2015). *Cloud computing design patterns*. Prentice Hall Press.

Ezenwoye, O. (2018). What language?-the choice of an introductory programming language. *2018 IEEE Frontiers in Education Conference (FIE)*, 1–8.

Fakhfakh, F., Kacem, H. H., & Kacem, A. H. (2017). Simulation tools for cloud computing: A survey and comparative study. *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*, 221–226.

Fehling, C., Leymann, F., Rette, R., Schupeck, W., & Arbitter, P. (2014). Cloud computing patterns. *Springer, Wien. DOI*, *10*, 978–3.

Fehling, C., Leymann, F., Retter, R., Schumm, D., & Schupeck, W. (2011). An architectural pattern language of cloud-based applications. *Proceedings of the 18th Conference on Pattern Languages of Programs*, 2:1–2:11.

Fehling, C., Leymann, F., Retter, R., Schupeck, W., & Arbitter, P. (2014). *Cloud computing patterns. fundamentals to design, build, and manage cloud applications*. Springer.

Foster, I., Zhao, Y., Raicu, I., & Lu, S. (2008). Cloud computing and grid computing 360-degree compared. *2008 Grid Computing Environments Workshop*, 1–10.

Foundation, A. S. (n.d.). *Apache delta cloud*. Retrieved September 14, 2014, from https://deltacloud.apache.org/

Foundation, A. S. (1999-2017). *Apache jmeter*. Retrieved October 24, 2017, from http://jmeter.apache.org/

Foundation, T. A. S. (2019). *Nuvem - incubator*. Retrieved April 10, 2019, from https://cwiki.apache.org/confluence/display/incubator/Nuvem

Foundation, T. A. S. (2021). *One interface to rule them all*. Retrieved January 14, 2021, from https://libcloud.apache.org/

Four architecture choices for application development in the digital age: Which application architecture model is best for you in the cloud era? (2020). Retrieved January 10, 2023, from https://www.ibm.com/cloud/blog/four-architecture-choices-for-application-development

Fowler, M. (2002). *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc.

Freeman, E., Robson, E., Bates, B., & Sierra, K. (2008). *Head first design patterns*. " O'Reilly Media, Inc."

Gahlyan, P., & Narayan Singh, S. (2018). Analysis of catalogue of gof software design patterns. *2018 8th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, 814–818.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1994). *Design patterns: Elements of reusable object-oriented software* (1st ed.). Addison-Wesley Professional.

Gandhi, A., Dube, P., Karve, A., Kochut, A., & Zhang, L. (2014). Adaptive, model-driven autoscaling for cloud applications. *11th International Conference on Autonomic Computing ({ICAC} 14)*, 57–64.

Gavrilović, N., Arsić, A., Domazet, D., & Mishra, A. (2018). Algorithm for adaptive learning process and improving learners' skills in java programming

language. *Computer Applications in Engineering Education*, *26*(5), 1362–1382.

Govind, H., & González–Vélez, H. (2021). Benchmarking serverless workloads on kubernetes. *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 704–712.

Grozev, N., & Buyya, R. (2014a). Inter-cloud architectures and application brokering: Taxonomy and survey. *Software: Practice and Experience*, *44*(3), 369–390.

Grozev, N., & Buyya, R. (2014b). Multi-cloud provisioning and load distribution for three-tier applications. *ACM Trans. Auton. Adapt. Syst.*, *9*(3), 13:1–13:21.

Grozev, N., & Buyya, R. (2015). Performance modelling and simulation of three-tier applications in cloud and multi-cloud environments. *The Computer Journal*, *58*(1), 1–22.

Gunka, A., Seycek, S., & Kühn, H. (2013). Moving an application to the cloud: An evolutionary approach. *Proceedings of the 2013 International Workshop on Multi-cloud Applications and Federated Clouds*, 35–42.

Ha, J., Kim, J., Park, H., Lee, J., Jo, H., Kim, H., & Jang, J. (2017). A web-based service deployment method to edge devices in smart factory exploiting docker. *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, 708–710.

Hanafy, W. A., Mohamed, A. E., & Salem, S. A. (2017). Novel selection policies for container-based cloud deployment models. *2017 13th International Computer Engineering Conference (ICENCO)*, 237–242.

Hansen, P. B. (1995). *Studies in computational science: Parallel programming paradigms*.

HAProxy. (2021). *Haproxy technologies - the world's fastest and most widely use load balancing solution*. Retrieved January 1, 2021, from https://haproxy.com/

Haq, M. S., Anwar, Z., Ahsan, A., & Afzal, H. (2017). Design pattern for secure object oriented information systems development. *2017 14th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, 456–460.

Hellemans, T., Bodas, T., & Van Houdt, B. (2019). Performance analysis of workload dependent load balancing policies. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, *3*(2), 1–35.

Henderson, T., Michalakes, J., Gokhale, I., & Jha, A. (2015). Chapter 2 - numerical weather prediction optimization. In J. Reinders & J. Jeffers (Eds.), *High performance parallelism pearls* (pp. 7–23). Morgan Kaufmann.

Hennion, N. (2021). *Glances an eye on your system. a top/htop alternative for gnu/linux, bsd, mac os and windows operating systems.* Retrieved January 1, 2021, from https://glances.readthedocs.io/en/latest/

Hohpe, G., & Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional.

Homer, A., Sharp, J., Brader, L., Narumoto, M., & Swanson, T. (2014). Cloud design patterns. *Redmon, Washington, United States: Microsoft*.

IBM Cloud Education, I. C. E., IBM Cloud Education. (2020). *Three-tier architecture*. Retrieved October 28, 2020, from https://www.ibm.com/cloud/learn/three-tier-architecture

Inc, I. (2021). *Influxdb contains everything you need in a time series data platform in a single binary.* Retrieved January 1, 2021, from https://www.influxdata.com/products/influxdb/

Jacob, B., Ng, S. W., & Wang, D. T. (2008). Chapter 3 - management of cache contents. In B. Jacob, S. W. Ng, & D. T. Wang (Eds.), *Memory systems* (pp. 117–216). Morgan Kaufmann.

Jamshidi, P., Pahl, C., Chinenyeze, S., & Liu, X. (2014). Cloud migration patterns: A multi-cloud service architecture perspective. *ICSOC Workshops*.

Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., & Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, *35*(3), 24–35.

Javadi, B., Abawajy, J., & Buyya, R. (2012). Failure-aware resource provisioning for hybrid cloud infrastructure. *Journal of parallel and distributed computing*, *72*(10), 1318–1331.

JClouds. (2017). *Jclouds*. Retrieved October 24, 2017, from http://www.jclouds.org/

Jiang, Y. (2020). Research on application value of computer software development in java programming language. *Journal of Physics: Conference Series*, *1648*(3), 032152.

Kazemi, S. M., Ghanbari, S., Kazemi, M., & Othman, M. (2023). Optimum scheduling in fog computing using the divisible load theory (dlt) with linear and nonlinear loads. *Computer Networks*, *220*, 109483.

Khajeh-Hosseini, A., Sommerville, I., Bogaerts, J., & Teregowda, P. (2011). Decision support tools for cloud migration in the enterprise. *2011 IEEE 4th International Conference on Cloud Computing*, 541–548.

Krebs, R., Wert, A., & Kounev, S. (2013). Multi-tenancy performance benchmark for web application platforms. *Proceedings of the 13th International Conference on Web Engineering*, 424–438.

Kumar, P., & Kumar, R. (2019). Issues and challenges of load balancing techniques in cloud computing: A survey. *ACM Computing Surveys (CSUR)*, *51*(6), 1–35.

Kumari, P., & Kaur, P. (2021). A survey of fault tolerance in cloud computing. *Journal of King Saud University - Computer and Information Sciences*, *33*(10), 1159–1176.

Kurmangali, A., Rana, M. E., & Ab Rahman, W. N. W. (2022). Impact of abstract factory and decorator design patterns on software maintainability: Empirical evaluation using ck metrics. *2022 International Conference on Decision Aid Sciences and Applications (DASA)*, 517–522.

Labs, G. (2021). *Dashboard overview*. Retrieved January 1, 2021, from https://grafana.com/docs/grafana/latest/dashboards/

Lahmann, G., McCann, T., & Lloyd, W. (2018). Container memory allocation discrepancies: An investigation on memory utilization gaps for container-based application deployments. *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 404–405.

Lana, C. A., Guessi, M., Antonino, P. O., Rombach, D., & Nakagawa, E. Y. (2019). A systematic identification of formal and semi-formal languages and techniques for software-intensive systems-of-systems requirements modeling. *IEEE Systems Journal*, *13*(3), 2201–2212.

Lasso, A., & Kazanzides, P. (2020). Chapter 35 - system integration. In S. K. Zhou, D. Rueckert, & G. Fichtinger (Eds.), *Handbook of medical image computing and computer assisted intervention* (pp. 861–891). Academic Press.

Le, Q., Zhanikeev, M., & Tanaka, Y. (2007). Methods of distinguishing flash crowds from spoofed dos attacks. *2007 Next Generation Internet Networks*, 167–173.

Linz, P., & Rodger, S. H. (2022). *An introduction to formal languages and automata*. Jones & Bartlett Learning.

Makaratzis, A. T., Giannoutakis, K. M., & Tzovaras, D. (2018). Energy modeling in cloud simulation frameworks. *Future Generation Computer Systems*, *79*, 715–725.

Martin, R. C. (n.d.). *Design Principles and Design Patterns* (tech. rep.).

Mell, P., Grance, T. et al. (2011). The nist definition of cloud computing.

Mendonca, N. C. (2014). Architectural options for cloud migration. *Computer*, *47*(08), 62–66.

Meng, X., Shi, J., Liu, X., Liu, H., & Wang, L. (2011). Legacy application migration to cloud. *2011 IEEE 4th International Conference on Cloud Computing*, 750–751.

mRemoteNG. (2021). *Multi-remote next generation*. Retrieved January 1, 2021, from https://mremoteng.org/

Muddinagiri, R., Ambavane, S., & Bayas, S. (2019). Self-hosted kubernetes: Deploying docker containers locally with minikube. *2019 International Conference on Innovative Trends and Advances in Engineering and Technology (ICITAET)*, 239–243.

Musser, J. (2012). Enterprise-class api patterns for cloud and mobile. *CITO Research*.

Niu, Y., Luo, B., Liu, F., Liu, J., & Li, B. (2015). When hybrid cloud meets flash crowd: Towards cost-effective service provisioning. *2015 IEEE Conference on Computer Communications (INFOCOM)*, 1044–1052.

Ochei, L. C., Petrovski, A., & Bass, J. M. (2015). Evaluating degrees of tenant isolation in multitenancy patterns: A case study of cloud-hosted version

control system (vcs). *2015 International Conference on Information Society (i-Society)*, 59–66.

Odun-Ayo, I., Misra, S., Abayomi-Alli, O., & Ajayi, O. (2017). Cloud multi-tenancy: Issues and developments. *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*, 209–214.

OrchardCore. (2022). *Getting started - orchard core*. Retrieved December 19, 2022, from https://orchardcore.net/

Pawluk, P., Simmons, B., Smit, M., Litoiu, M., & Mankovski, S. (2012). Introducing stratos: A cloud broker service. *2012 IEEE Fifth International Conference on Cloud Computing*, 891–898.

Peddigari, B. P. (2011). Unified cloud migration framework using factory based approach. *2011 Annual IEEE India Conference*, 1–5.

Petcu, D., Crăciun, C., Neagul, M., Panica, S., Di Martino, B., Venticinque, S., Rak, M., & Aversa, R. (2011). Architecturing a sky computing platform. *Proceedings of the 2010 International Conference on Towards a Service-based Internet*, 1–13.

Ponce, F., Márquez, G., & Astudillo, H. (2019). Migrating from monolithic architecture to microservices: A rapid review. *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, 1–7.

Prathiba, S., & Sowvarnica, S. (2017). Survey of failures and fault tolerance in cloud. *2017 2nd International Conference on Computing and Communications Technologies (ICCCT)*, 169–172.

Priyadarsini, R. J., & Arockiam, L. (2013). Failure management in cloud: An overview. *International Journal of Advanced Research in Computer and Communication Engineering*, *2*(10), 2278–1021.

ProgrammableWeb. (2011-2019). *Kaavo webservices api*. Retrieved April 1, 2011, from http://jmeter.apache.org/

Qu, C., Calheiros, R. N., & Buyya, R. (2016). A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances. *Journal of Network and Computer Applications*, *65*, 167–180.

Qu, C., Calheiros, R. N., & Buyya, R. (2017). Mitigating impact of short-term overload on multi-cloud web applications through geographical load balancing. *concurrency and computation: practice and experience*, *29*(12), e4126.

Rajavaram, H., Rajula, V., & Thangaraju, B. (2019). Automation of microservices application deployment made easy by rundeck and kubernetes. *2019 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, 1–3.

Ramirez, A. O. (2000). Three-tier architecture. *Linux J.*, *2000*(75es).

Révész, G. E. (1991). *Introduction to formal languages*. Courier Corporation.

Riehle, D., Harutyunyan, N., & Barcomb, A. (2021). Pattern discovery and validation using scientific research methods. *arXiv preprint arXiv:2107.06065*.

Rimal, B. P., & Maier, M. (2017). Workflow scheduling in multi-tenant cloud computing environments. *IEEE Transactions on Parallel and Distributed Systems*, *28*(1), 290–304.

Rockford Lhotka - Should all apps be n-tier? (2020). Retrieved August 15, 2022, from https://web.archive.org/web/20200802111420/http://www.lhotka.net:80/weblog/ShouldAllAppsBeNtier.aspx

Saboor, A., Mahmood, A. K., Hassan, M. F., Shah, S. N. M., Hassan, F., & Siddiqui, M. A. (2021). Design pattern based distribution of microservices in cloud computing environment. *2021 International Conference on Computer & Information Sciences (ICCOINS)*, 396–400.

Sahu, Y., Pateriya, R., & Gupta, R. K. (2013). Cloud server optimization with load balancing and green computing techniques using dynamic compare and balance algorithm. *2013 5th International Conference and Computational Intelligence and Communication Networks*, 527–531.

Saratha, P., Uma, G. V., & Santhosh, B. (2017). Formal specification for online food ordering system using z language. *2017 Second International Conference on Recent Trends and Challenges in Computational Models (ICRTCCM)*, 343–348.

Scalr. (2011-2021). *Collaboration and automation for terraform — scalr*. Retrieved January 2, 2021, from http://jmeter.apache.org/

Shafiq, D. A., Jhanjhi, N. Z., Abdullah, A., & Alzain, M. A. (2021). A load balancing algorithm for the data centres to optimize cloud computing applications. *IEEE Access*, *9*, 41731–41744.

Shah, J. M., Kotecha, K., Pandya, S., Choksi, D., & Joshi, N. (2017). Load balancing in cloud computing: Methodological survey on different types of

algorithm. *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, 100–107.

Siteground. (2004-2017). *What is mysql*. Retrieved October 24, 2017, from https://www.siteground.co.uk/tutorials/php-mysql/mysql.htm

Strauch, S., Breitenbuecher, U., Kopp, O., Leymann, F., & Unger, T. (2012). Cloud data patterns for confidentiality. *CLOSER*, *12*, 387–394.

Techopedia. (2021). *Putty*. Retrieved January 1, 2021, from https://www.techopedia.com/definition/4335/putty

Tenant isolation using TopLink. (2022). Retrieved July 16, 2022, from https://docs.oracle.com/middleware/12212/toplink/solutions/multitenancy.htm#TLADG615

Tupper, C. D. (2011). 21 - object and object/relational databases. In C. D. Tupper (Ed.), *Data architecture* (pp. 369–383). Morgan Kaufmann.

Tychalas, D., & Karatza, H. (2020). An advanced weighted round robin scheduling algorithm. *24th Pan-Hellenic Conference on Informatics*, 188–191.

Vale, G., Correia, F. F., Guerra, E. M., de Oliveira Rosa, T., Fritzsch, J., & Bogner, J. (2022). Designing microservice systems using patterns: An empirical study on quality trade-offs. *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, 69–79.

Vanhove, T., Vandensteen, J., Van Seghbroeck, G., Wauters, T., & De Turck, F. (2014). Kameleo: Design of a new platform-as-a-service for flexible data management. *2014 IEEE Network Operations and Management Symposium (NOMS)*, 1–4.

Varghese, B., & Buyya, R. (2018). Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems*, *79*, 849–861.

Varia, J. (2010). Migrating your existing applications to the aws cloud. *A Phase-driven Approach to Cloud Migration*, 1–23.

Varia, J. (2011). Best practices in architecting cloud applications in the aws cloud. *Cloud Computing: Principles and Paradigms*, *18*, 459–490.

Venugopal, S., Desikan, S., & Ganesan, K. (2011). Effective migration of enterprise applications in multicore cloud. *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, 463–468.

Walraven, S., Monheim, T., Truyen, E., & Joosen, W. (2012). Towards performance isolation in multi-tenant saas applications. *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, 6:1–6:6.

Wang, W., & Casale, G. (2014). Evaluating weighted round robin load balancing for cloud web services. *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 393–400.

Ward, C., Aravamudan, N., Bhattacharya, K., Cheng, K., Filepp, R., Kearney, R., Peterson, B., Shwartz, L., & Young, C. C. (2010). Workload migration into clouds challenges, experiences, opportunities. *2010 IEEE 3rd International Conference on Cloud Computing*, 164–171.

Wettinger, J., Andrikopoulos, V., Leymann, F., & Strauch, S. (2017). Middleware-oriented deployment automation for cloud applications. *IEEE Transactions on Cloud Computing*, *PP*(99), 1–1.

What is Fault Tolerance? Definition & FAQs. (2022). Retrieved August 18, 2022, from https://www-stage.avinetworks.com/glossary/fault-tolerance/

Wikipedia. (2017). *Apache http server*. Retrieved October 24, 2017, from https://en.wikipedia.org/wiki/Apache_HTTP_Server

Wilder, B. (2012). *Cloud architecture patterns*. " O'Reilly Media, Inc."

WordPress. (2017). *Wordpress features*. Retrieved October 24, 2017, from https://codex.wordpress.org/WordPress

Xu, F., Liu, F., Jin, H., & Vasilakos, A. V. (2013). Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions. *Proceedings of the IEEE*, *102*(1), 11–31.

Yin, R. K. (2014). *Case study research: Design and methods*. USA:sage publications.

Zhang, H., Zhang, J., Bai, W., Chen, K., & Chowdhury, M. (2017). Resilient datacenter load balancing in the wild. *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 253–266.

Zhang, M., & Wang, K. (2000). Implementing undo/redo in pdf studio using object-oriented design pattern. *Proceedings 36th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS-Asia 2000*, 58–64.

Zhang, W. B., Roman, A. J., & Huru, H. A. (2009). Migrating legacy applications to the service cloud. *Proceedings of the 14th Conference companion on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 59–68.

Zhao, J., Rodriguez, M. A., & Buyya, R. (2020). High-performance mining of covid-19 open research datasets for text classification and insights in cloud computing environments. *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, 302–309.

Zhao, J.-F., & Zhou, J.-T. (2014). Strategies and methods for cloud migration. *International Journal of Automation and Computing*, *11*(2), 143–152.

Zomaya, A. Y., & Teh, Y.-H. (2001). Observations on using genetic algorithms for dynamic load-balancing. *IEEE transactions on parallel and distributed systems*, *12*(9), 899–911.

Appendix A

Enhanced Multi-tenancy Patterns

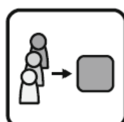
A.1 Shared Component Pattern

This section implements the new cloud pattern catalogue structure to describe the shared component pattern.

Pattern Name

Shared Component

Icon



Intent

Allow multiple tenants to access a component of an application to leverage economies of scale (Fehling, Leymann, Retter, Schupeck, et al., 2014).

Motivation

To address a large number of customers and in turn leverage economies of scale. When tenants are different departments of an organisation using an application, they commonly share IT resources that power the application. Therefore, the shared component pattern can be used to share the application component instance and not just the IT resources that power the application. (Adewojo, Bass, Allison, & Hui, 2015).

Applicability

Use shared components when: Web services are used for authentication and user rights management within the scope of one company.

Structure

A database structure is used to exemplify the shared component pattern in this section. One database is created and the tenant shares the database. Tables in the database are also shared among tenants, the tenant and row ID identify each row of a table. The process of combining and identifying tenants' data does not completely guarantee the security and privacy of data, but it utilises resources efficiently and reduces the cost of database connections. The structure is represented in Figure A.1.

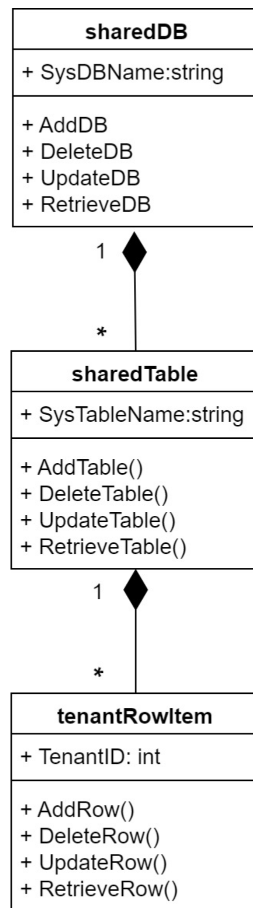


Figure A.1: UML Diagram for Shared Component Pattern

Consequences

The functionality provided by a shared component is unaware of the actual tenant for which request is being executed. Hence, the behaviour of one tenant may affect other tenants. This is commonly seen as a negative consequence. A positive consequence is an ability to uniformly scale instances of a shared component depending on the overall workload and the set limit. This process reduces the commissioning and decommissioning effort of administrating tenants' applications.

Implementation

The shared component pattern can be implemented in various forms. These forms include implementing a shared database table that contains the user details data of different departments in an organisation; implementing a web service that will be used by different organisations; and implementing a customised but shared user interface of a system. This pattern allows components to be shared by multiple tenants without much restriction on identifying the tenant (Adewojo, Bass, Allison, & Hui, 2015). To successfully implement this pattern, the following conditions should be considered before implementing it.

1. Tenant's Influence: Influences among tenants that may occur when sharing components should be avoided.
2. Tenant's Requirements: Other requirements of tenants that disallow sharing of resources.

Sample Code

```
//creates database (DB) for all tenants
CREATE DATABASE [SysDBName];

// creates table for all tenants
CREATE TABLE [SysDBName].[SysTableName] (
TenantID datatype,
TenantColData1 datatype,
TenantColData2 datatype,
......);

// Add data based on tenant Id
INSERT INTO [SysDBName].[SysTableName]
```

```
(TenantID, TenantColData1, TenantColData2, ...)
VALUES (TenantID1, Tenant1Data1, Tenant1Data2,.),
(TenantID1, Tenant1Data3, Tenant1Data4, ...), ...
WHERE TenantID = tenantId;
```

Known Uses

The shared component pattern is widely used in situations where sharing of component resources will less likely affect the performance, security, and privacy of an organisation's data or workload. Examples of usage are: National Weather Service, provided by the National Oceanic and Atmospheric Administration (Fehling, Leymann, Retter, Schupeck, et al., 2014), and Salesforce SaaS.

Related Patterns

Tenant-Isolated and dedicated patterns are suitable patterns that can replace the shared component pattern. These patterns are useful if sharing of application components is unsuitable for tenants. Management configuration, periodic workload patterns (Fehling, Leymann, Retter, Schupeck, et al., 2014), private clouds and hypervisors are combinations of alternative patterns and cloud deployment models that can be used to complement the shared component pattern; especially when different parts of the same organisation will share application components.

A.2 Tenant Isolated Component Pattern

This section implements the new cloud pattern catalogue structure to describe the tenant isolated component pattern.

Pattern Name

Tenant Isolated Component

Icon

Intent

Components will be shared by tenants and influences of tenants that affect assured performance, security, available storage capacity, and accessibility are avoided (Fehling, Leymann, Retter, Schupeck, et al., 2014) (Adewojo, Bass, & Allison, 2015).

Motivation

To maximise the use of available resources without jeopardising assured performance, available storage, security, privacy, and accessibility. This will address many customers, reduce tenants' management efforts, and in turn leverage economies of scale (Fehling, Leymann, Retter, Schupeck, et al., 2014). Applications can be configured to suit the tenant's requirement, and the workload of the tenant will not affect other tenants (Adewojo, Bass, Allison, & Hui, 2015).

Applicability

Use tenant-isolated component pattern for:

- Authentication Isolation: this aids the customer in incorporating the pattern with an on-premise application to verify users and isolate user activities (Adewojo, Bass, & Allison, 2015)
- Access Control Isolation: this enforces access right within tenant isolated component.
- Information Protection Isolation: this handles access control to tenant data handled by tenant isolated component.
- Performance Isolation: this ensures that the workload of a tenant does not affect another tenant.
- Fault Isolation: this ensures that failures in a tenant-isolated component do not affect all tenants, but are contained within the affected component
- Administration Isolation: this ensures that only a tenant's administrator can make access rights and administrative changes to its tenant's application.

Structure

This section uses a database structure to describe the tenant-isolated component pattern. A database is created, and a tenant with a unique tenant name owns each table. Database tables contain many rows with unique row IDs. This structure allows for the privacy and security of data because tenants will not have their data mixed with other tenants' data. This will however increase the cost of database connections and minimise resource utilisation. The structure is represented in Figure A.2.

Consequences

Tenant isolated component pattern ensures that a tenant is authenticated before the tenant gains access to its application. This pattern provides the highest degree of resource sharing without adversely affecting other tenants. The tenant isolated

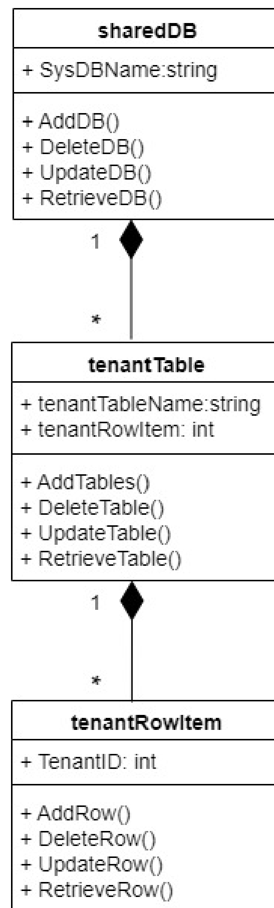


Figure A.2: UML Diagram for Tenant Isolated Component Pattern

pattern's sharing model still reduces the runtime cost per tenant because resources are still being shared, albeit not at a great level as compared to the shared component pattern. This affords a cloud application provider to target a larger market with little to no compromise of delivered services (Adewojo, Bass, & Allison, 2015).

Implementation

The tenant-isolated component pattern allows components to be shared by multiple tenants based on their identifier. This pattern is instrumental in the development and sharing of application stacks that will be used by different tenants. Tenant isolated

pattern ensures isolation between tenants by controlling tenant's access, processing performance usage, and separation of stored data. A sample implementation of tenant isolated patterns is by creating a database that will be used by multiple tenants to store various details of their organisations.

Tenant's configuration requirements, especially those that disallow sharing of resources, should be considered when using tenant isolated pattern. The requirements can still be honoured by implementing some other cloud patterns in collaboration with tenant-isolated pattern (Adewojo, Bass, Allison, & Hui, 2015).

Sample Code

```
//creates database (DB) for all tenants
CREATE DATABASE [SysDBName];

 // create table for tenant A
CREATE TABLE [SysDBName].[TenantA.TenantTableName] (
TenantId datatype,
TenantColData1 datatype,
TenantColData2 datatype,
......);

// create table for tenant B
CREATE TABLE [SysDBName].[TenantB.TenantTableName] (
TenantId datatype,
TenantColData1 datatype,
TenantColData2 datatype,
......);

// Add data based on tenant table name
INSERT INTO [SysDBName].[TenantB.TenantTableName]
(TenantId, TenantColData1, TenantColData2, ...)
```

```
VALUES (TenantID1, Tenant1Data1, Tenant1Data2, ...),
(TenantID1, Tenant1Data3, Tenant1Data4, ...), ...
WHERE TenantName = tenantName or schemaName;
```

Known Uses

Tenant isolated pattern is commonly used in applications that are shared by divisions of a large organisation. Examples are a single payroll application that is used by multiple divisions of a company. Each division's data source can be implemented as a tenant isolated component using Oracles Toplink middleware with a feature called table-per-tenant multi-tenancy ("Tenant Isolation Using TopLink", 2022).

Related Patterns

The shared component pattern is an alternative pattern that can be used when tenants do not require assured tenant isolation. This pattern can be configured with other patterns such as management configuration and periodic workload pattern (Fehling, Leymann, Retter, Schupeck, et al., 2014) to improve its applicability.

The dedicated component pattern is the second alternative pattern to the tenant isolated pattern. This is a great option for tenants who desire the highest degree of isolation. It guarantees tenant isolation in regard to performance, security, and data privacy (Adewojo, Bass, & Allison, 2015).

A.3 Dedicated Component Pattern

This section implements the new cloud pattern catalogue structure to describe the dedicated component pattern.

Pattern Name

Dedicated Component

Icon

Intent

To provide exclusive access to components that provides critical functionality, while other components can still be shared among tenants.

Motivation

Requirements to offer dedicated access to components that offer critical functionality, component functionality that have security and privacy rules and regulations they must abide by, and components that require specific configurations that cannot be shared by tenants.

Applicability

This pattern is applicable to applications with component parts that cannot be shared because of rules and regulations, legacy applications whose components are not designed to be shared by tenants but need to integrate into distributed environments, and applications that offer functionality that is too critical or that must be configured specifically for individual tenants.

Structure

In the context of a database application, the database is created as many times as there is a need to support a critical component of an application. Each database is dedicated to a tenant, therefore it has dedicated tables and rows. This structure affords the highest form of privacy and security of data, albeit the storage space used is increased. The structure is represented in Figure A.3.

Consequences

Components are exclusive to tenants using the application, so components cannot be shared with tenants. Furthermore, component resource usage will not be maximised.

Implementation

Dedicated components allow tenants to adjust components easily to suit their requirements. A dedicated user interface component, processing component, and data access component are available to each tenant. However, this implementation reduces the degree of sharing and the ability to benefit from economies of scale.

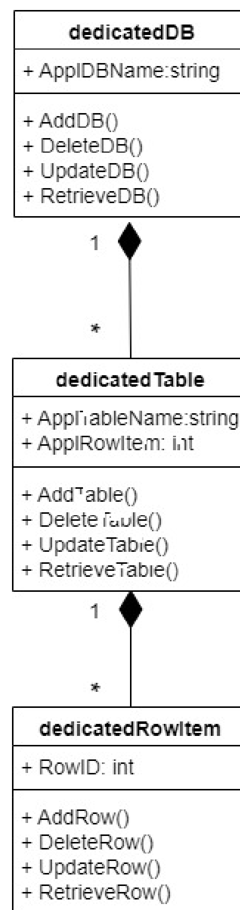


Figure A.3: UML Diagram for Dedicated Component Pattern

Sample Code

```
//creates database (DB) for one tenant
CREATE DATABASE [ApplDBName];
// create tables for one tenant
CREATE TABLE [ApplDBName].[ApplTableName] (
RowId datatype,
TableColData1 datatype,
TableColData2 datatype,
......);
```

```
CREATE TABLE [ApplDBName].[ApplTableName] (
RowId datatype,
TableColData1 datatype,
TableColData2 datatype,
......);
// Add data based on table name
INSERT INTO [ApplDBName].[ApplTableName]
(RowId, TableColData1, TableColData2, ...)
VALUES (RowId, TableData1, TableData2, ...),
(RowId, TableData3, TableData4, ...), ...;
```

Known Uses

Salesforce CRM system offered as a PaaS (Adewojo, Bass, & Allison, 2015)

Related Patterns

The shared component pattern is an alternative pattern that can be used when tenants do not require assured tenant isolation. This pattern can be configured with other patterns such as management configuration and periodic workload pattern (Fehling, Leymann, Retter, Schupeck, et al., 2014) to improve its applicability.

Tenant Isolated component pattern is another alternative pattern that can be used when there is a need to maximise resource usage while guaranteeing some degree of tenant isolation.

Appendix B

Approved Ethical Application Document

B.1 Ethical Clearance