A Framework for High Availability Management of Applications Services in Cloud

Yanal Alahmad

A thesis
In the Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Computer Science) at
Concordia University
Montréal, Québec, Canada

November, 2021

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Yanal Alahmad**

Entitled: **A Framework for High Availability Management of Applications Services in Cloud**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________ Chair
Dr. Alex De Visscher

______________________ External Examiner
Dr. Kim Khoa Nguyen

______________________ Examiner
Dr. Jamal Bentahar

______________________ Examiner
Dr. Thomas Fevens

______________________ Examiner
Dr. Emad Shihab

______________________ Supervisor
Dr. Anjali Agarwal

Approved ______________________
Dr. Leila Kosseim
Chair of Department or Graduate Program Director

November 30, 2021 ______________________
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

A Framework for High Availability Management of Applications Services in Cloud

Yanal Alahmad, Ph.D.
Concordia University, 2022

Cloud computing is a fast and growing paradigm for hosting applications services that belong to the Application Service Providers (ASPs). However, Quality of Service (QoS) remains an issue that opens different research areas in a distributed, elastic and dynamic cloud platform. One major issue raised for the ASP is service High Availability (HA). Service availability is a non-functional requirement that indicates the period of time the service is provided for the end customer. Managing the availability of different application services during the runtime in a cloud cluster is not an easy task to do due to several challenges. The key success to maintain service availability is to provide a mechanism to protect service against failure and recover the service once the failure happens as fast as possible.

This thesis proposes a general framework for availability management and enables continuity for applications services in the cloud computing environment. We address service availability and propose efficient solutions from different perspectives. First, the thesis proposes a reactive framework that can maintain HA of the application service in a virtualized computing cluster. Second, a proactive service availability framework is proposed. The framework uses deep learning methods to predict application task termination status (Success or Fail) in cloud cluster using three public available datasets. The results show the used methods can predict task termination status with high accuracy. Third, a failure-aware task scheduler approach is proposed. The scheduler uses a heuristic approach to solve task scheduling NP-hard problem with the objective to minimize failure probability and resources usage of tasks. The results show the ability of the scheduler to protect many tasks and save a large number of resources. Fourth, the thesis proposes an availability-aware Virtual Machine (VM) dynamic placement framework. The framework tackles VMs placement as a response to different request types that include deploying a new application, VM scaling, and migration. Moreover, an optimization approach that is based on the heuristic AntColony algorithm is proposed to solve the VM placement NP-hard problem. The approach targets multiple objectives to minimize power consumption, resources wastage, and failure of the active servers that are used to host the VMs. In addition, the placement approach tries to provide application service availability as close as possible to the requirements by ASP and avoids violation of the Service Level Agreement (SLA). The results show the ability of the framework to increase the admissibility of new applications that meet the availability requirements and enhance the resources utilization of servers, compared to the existing VM placement solutions in the literature.

Acknowledgments

All praises be to “ALLAH” Almighty who gave me opportunity, strength, and patient to complete this task successfully. His continuous grace and mercy were with me throughout my life and ever more during my study. I would like to express my gratitude to my supervisor, Prof. Anjali Agarwal whose expertise, understanding, support and patience made this thesis possible. I appreciate her vast knowledge and skills in many areas, and her assistance in research and writing papers. I doubt that I will be able to express my appreciation fully for supervising me during my PhD study, but I owe her my deepest gratitude and thanks. Deepest thanks also to the committee members for their advices and suggestions regarding the thesis and beyond. Moreover, I would like to thank NSERC, and MITACS Accelerate program in collaboration with Cistech for their fund and support.

No body has been more important to me in pursuing this long study journey than the members of my family. Therefore, I dedicate this thesis to every one of my family who waited me to finish my PhD. A very special thanks go out to my beloved and supporting wife Bayan, and my wonderful little son Tameem. Without their love, and encouragement, I would not have finished this thesis. Most importantly, I would like to thank my mother and father, whose love, support and guidance are with me during my entire life. Special thanks go to my two sisters and three bothers for standing beside me and continuous encouragement to finish my study.

I would never forget to thank my mother-in-law and father-in-law for praying for me to be successful. Finally, I would like to thank all my friends for any help or advice during my studies.

Contents

List of Figures

List of Tables

Acronyms

ANN	Artificial Neural Network
ASP	Application Service Provider
CC	Cloud Computing
CNN	Convolutional Neural Network
CSP	Cloud Service Provider
HA	High Availability
ILP	Integer Linear Programming
INLP	Integer Non-Linear Programming
MAE	Mean Absolute Error
MDP	Markov Decision Process
MSE	Mean Square Error
MTTF	Mean Time to Fail
MTTR	Mean Time to Repair
NFV	Network Function Virtualization
NFS	Network File Storage
NIST	National Institute of Standards and Technology
NS	Network Service
QoS	Quality of Service
SLA	Service Level Agreement
SFC	Service Function Chain
TSP	Telecom Service Provider
VM	Virtual Machine
VNF	Virtual Network Function
VNFFG	Virtual Network Function Forward Graph

Chapter 1

Introduction

Cloud computing (CC) is a popular paradigm that changes the traditional way of providing computing services to customers. National Institute of Standards and Technology (NIST) defines CC as a model to enable on-demand network access for a shared and large set of configurable resources such as networks, servers, applications, and services that can be quickly provisioned and de-provisioned with minimum management efforts [1]. Virtualization technology [2] is considered the magic key behind the prosperity of the cloud. Virtualization allows emulating the functionality of computing hardware device as a piece of software that is called a virtual node. In the cloud context, the virtual node can be either VM or container that can be executed on a physical computing node such as server (host). Servers are located at Data Centers (DCs) that belong to the Cloud Service Provider (CSP). CSP allows customers to get online access to virtual computing resources on-demand based on the workload fluctuations of the provisioned services. Customer is charged only for the consumed resources based on the business model 'pay-as-you-go'. This motivated many of the institutions and individuals to move their businesses to the cloud and become customers for CSPs. Although the cloud has many advantages, still QoS remains a main concern issue. With the fast growing number of application services in cloud, availability of the provisioned application service is raised as a main challenge to be satisfied according to the given requirements. Service availability is a non-functional requirement that refers to the percentage of time the service is available for the end customer [3]. Some customers expect their services to be highly available anytime from anywhere. High Availability (HA) refers to the ratio 99.999% (aka, five nines) of the time the service is available for the customer [3]. In other words, service outage should not exceed 5.6 minutes per year for

HA service [3]. Other customers demand to enable service continuity feature that allows resuming service from its last state before it is interrupted.

Managing service availability is very important to guarantee QoS, and enhance the overall performance of the resources that are used to provide the service. However, maintaining service availability during its entire lifetime in a distributed and dynamic cloud environment is not an easy task that faces several challenges. Our research works in this thesis focus on application service availability management, service continuity, service protection, recovery, and resource management from different perspectives. Therefore, we study and analyze the related research works and solutions from the literature. In addition, we propose a set of models, optimization methods, algorithms, data analysis, and machine learning methods, to provide efficient solutions for application service availability management in cloud. The rest of the chapter is organized as follows. Section 1.1 motivates our research work. Section 1.2 discusses the problem statement. Section 1.3 summarizes the main objectives of this research work. Section 1.4 highlights the main contributions of the research work, at the end Section 1.5 states the thesis organization.

1.1 Motivation

In the cloud environment, ASP can demand a set of resources from CSP to deploy application(s) to be provided for the end customers. Usually, ASP delegates the task of application service availability management to CSP which can offer what we call 'availability-as-service'. ASP and CSP can both agree on the type, number, requirements, and conditions of the offered resources and services through SLA. Satisfying the availability requirements of the application is necessary and beneficial for both ASP and CSP. Providing availability level below what ASP demands can lead to severe losses. According to Gartner [4], downtime of the web application can cost companies $5,600$$ per minute and up to $300,000$$ per hour. In addition, providing lower availability can lead to SLA violation that can cost CSP to pay penalties according to the agreement. Also, lower service availability can lead to performance degradation or outage (interruption) for the provisioned service that can impact QoS and user experience, which can reduce the overall profits of ASP and CSP. On the other hand, providing availability far above the demands can decrease the admissibility of new services, and hence reduce the profit of the CSP. Providing high availability levels to protect the service requires adding extra backup resources to protect the service. Extra resources add additional operational and expenditure costs for CSP. Therefore, it is necessary to provide service availability close to the demand by ASP. Through studying the related works, we found that most of the solutions do not target to satisfy the application availability requirements. There are few works targeted to maximize the availability of the application service which is not always a good solution because it can lead to decreasing the admissibility of new applications and waste resources. This motivated us to propose a solution that aims to provide service availability as close as possible to the required one as stated in SLA. The existing solutions focused on the availability of the infrastructure at virtual and physical layers and neglected the availability of components at the application layer. So we focus a part of our research work on monitoring, failure detection, recovery, and protection of the application component. In addition, the existing works lack the definition of end-to-end application service availability. A few works considered service continuity, most of the solutions try to restart the service from the beginning in case of failure. It is necessary to enable service continuity for stateful applications, that have a state, to resume service from its last state before it fails instead of from the beginning. Service continuity can be a requirement that needs to be applied to gain the satisfaction of the customers.

In the cloud context, ASP has the choice to select between VMs and containers to deploy application components that compose and provide the service. VM is based on a software image that is larger than the image of the container, so VM can host multiple components. Container provides a micro-service, so it hosts a maximum of one component. However, management of VM including creation, operation, migration, and deletion, takes a longer time than the management of container. Therefore, using VM or container can play a role in resource utilization, and speed up the process of service recovery in case of interruption. The existing availability solutions in the literature do not consider the impact of resources types at the service availability level. In this research work, we differentiate between using VMs and containers, as well as their roles from service recovery and outage perspectives. Cloud enables the elasticity mechanism that allows scaling resources in different directions based on the workload demands of the service. Adding or removing resources on the fly can decrease or increase the availability level of the provisioned service. Most of the existing elasticity solutions in the literature do not consider application service availability while taking resource scaling decisions. In this research work, we coordinate between the resource scaling actions and the service availability management. In addition, we propose optimization solutions to scale resources while considering the availability requirements of the service. Application task scheduling is a process to determine the sequence order to execute the task and to specify in which computing node to execute the task. Task scheduling is very important that can have an impact on the overall performance and throughput of the computing cluster, as well as the availability of the task. Application task in cloud cluster is prone to failure for several reasons such as lack of resource, hardware and software failures. Knowing the termination status of the task, whether success or failure, can help to protect the task in advance, and hence save consumed resources by the task. A set of existing research works in the literature proposed different solutions to predict task failure in cloud cluster. However, most of these solutions do not consider taking actions for tasks that are predicted as failed. In our work, we consider a set of proactive actions and rescheduling for tasks based on the prediction of the termination status. VM placement is a process that includes searching for a computing server to host VM. VM can host a set of components that belong to the same or different applications. Failure of the server will definitely lead to the failure of all the VMs that are hosted on the server, hence interrupting all the provisioned services by the VMs. Therefore, VM placement of VMs should be done carefully and consider the availability of the service. Many solutions in the literature tackled VM placement problem for different objectives. However, limited solutions consider meeting service availability requirements. In this research work, we tackle VM placement problem to achieve multiple objectives.

1.2 Research Problems

This thesis addresses the main challenges that threaten application service availability management in the cloud computing platform. We discuss the main problems considered in this work in the following subsections.

1.2.1 Resource Failure

Resource failure is considered as the main threat and a real challenge to service availability. At DC of cloud, application service is provided through a set of software components, where each component provides the functionality of a certain type towards providing the end-to-end service. Component can be hosted on virtual computing node such as VM or container that is hosted at the underlying physical computing server (host). Such tight dependency between resources at different layers makes the application service prone to outage due to resource failure at any time. Resource failure needs an efficient monitoring mechanism to detect the failure. In addition, it needs a management solution to resume the interrupted service as fast as possible. There are two main approaches to manage service availability, either reactive or proactive. The reactive approach usually uses a redundancy model that includes extra standby (backup) VMs or containers to protect the service. In case of failure of the active VM that provides the service, the management solution immediately failover the service to the standby VM that continues providing the service and becomes active. So the service outage depends mainly on the delay to failover the interrupted service from active to the standby VMs. To enable service continuity, active VM keeps updating the service state to the standby VM during the entire lifetime of the service. Although the reactive approach is efficient and fast for service recovery, it is considered costly because it requires adding extra standby resources to protect the service. In addition, the communication between the active and standby resources consumes large networking bandwidth. Extra costs for operational, maintenance, and power consumption will be added as a result of using extra resources. On the other hand, the proactive approach usually uses prediction methods such as machine learning, and probability models to predict service failure, and take the appropriate protection actions before failure actually happens. Although the proactive approach is not so costly since it does not require adding backup resources, its accuracy depends mainly on the accuracy of the method that is used to predict the failure. In addition, the prediction process depends on the data history that includes service failure patterns. Dataset is mainly used for training and testing the prediction method which normally takes a long time, as well as large computing resources. In DCs of the cloud, virtual and physical infrastructure have heterogeneous properties such as CPU, RAM, and storage capacities. Services demand and competing for the resources, and shortage of fulfilling the demanded resources by the service can jeopardize its availability. Therefore, both reactive and proactive availability management approaches should be aware of the demanded resources, as well as the capacities of the infrastructure in DCs to make the correct decision for service recovery and protection.

1.2.2 Resource Scaling

The elasticity feature in the cloud allows the resources to be added or removed automatically according to the workload demand of the provisioned service [5]. If the workload demands increase, new resources are added. While if the workload demands decrease, a set of the existing resources are removed. Scaling of the resources can be in any of the two directions, horizontal or vertical. For example, in VM-based computing cluster, horizontal scaling can include adding new VMs, or removing existing VMs. While vertical scaling can include adding virtual resources such as virtual CPU (vCPU) and RAM (vRAM) to an existing VM, or removing the resources from the VM. Scaling of resources is a real challenge that can lower service availability, hence violating SLA. For example, removing standby (backup) VM in case of horizontal scaling can leave the service without any protection plan in case of failure of the active VM. Vertical VM scaling can lead to a scenario where the server that hosts the VM may not have enough resources to continue hosting the VM after its upgrade. So the VM has to be migrated from its current hosted server to another server that can accommodate the upgraded VM. However, the VM migration has to be done carefully because it can impact the service availability. If the VM is migrated to a server with lower availability, this will lower the availability of the end-to-end application service for which the VM belongs. Migration of the active VM that is in charge of providing the service will lead to an interruption of the service until the migration process is done. Even if the interruption happens for a short period of time, it may not be tolerated by some customers, especially for HA services. Usually, at DC of the cloud, VM can have different candidate destination servers to be migrated to with different migration times. The problem gets more complex if a set of VMs that belong to different applications need to be migrated at the same time. So the VMs have to be migrated in such a way that can minimize the total migration time of all the VMs, without violating the availability requirement of any application. To summarize, VMs scaling and migration is a big challenge against service availability in cloud cluster and is considered as a combinatorial NP-hard problem that needs an efficient solution such as using a heuristic optimization approach.

1.2.3 Application Task Scheduling

Scheduling of application task at the underlying computing node, whether it is virtual or physical, is a challenge that can play a crucial role in the termination status of the task, whether successful or failure. Through our analysis of Google cloud cluster trace logs [6], we found 29% of all tasks that have been executed at the cluster were failed (terminated unsuccessfully). In addition, the data analysis shows that task can fail due to several reasons. The scheduling node to execute the task, the configuration of task, or shortage of resources are the main reasons behind the failure of the task. Moreover, the analysis shows that most of the failed tasks have been resubmitted for execution more than one time to give them a chance to finish successfully. Waiting for the task to fail, and resubmitting it several times for the execution can waste a lot of the resources that burden the performance and throughput of the cluster. So several existing research works tried to predict task failure using different methods and techniques. However, most of the works predict the task termination status before the task gets scheduled. These works neglect the runtime usage of the task which can be very helpful to increase the accuracy of the prediction process. Moreover, a limited number of works take actions to protect the task in case its termination status is predicted as failed. Through data analysis, we found a set of remedy actions that can be taken to protect the task before it actually fails. One effective action is to reschedule the task on a different computing node than where it is currently being executed. Usually, in a cloud cluster, there are many computing nodes that can execute the task with different failure probabilities. So the problem is to select which node to execute the task that can reduce its failure probability. The problem gets more complex if a set of tasks are required to be rescheduled at the same time. The next problem is therefore to find the sequence to schedule the tasks at the underlying nodes in such a way that reduces the resources usage by the tasks. Scheduling of tasks in cloud cluster from task failure, and resource utilization perspectives is a real challenge that is also considered as a combinatorial NP-hard problem that requires using an efficient optimization approach to solve it.

1.2.4 VM Placement

VM placement is a process to find a computing server to host the VM. The placement of VMs is also a big challenge that can have a direct impact on the quality and availability of the provisioned services by the VMs, as well as on the resource utilization and power consumption of the servers that host the VMs. Placement of VMs that provide the same service type close to each other can reduce the availability zone of the VMs which reduces the overall availability level of the service. For example, collocating active and standby (backup) VMs that are in charge of providing a specific service instance on the same server will definitely lead to a service outage in case of server failure. However, the placement of VMs close to each other helps to reduce the total number of active computing servers that are required to host the VMs. As a result, the total power consumption, operational and maintenance cost of the active servers will be reduced for CSP. Moreover, the placement of the VMs close to each other helps to reduce the communication delay between the VMs, as a result, the response time of the end-to-end application service will be reduced as well. On the other hand, the placement of the VMs far away from each other increases the availability zone of the VMs, hence increasing the overall availability level of the application service. In addition, distributing VMs at servers can result in load balancing among the servers that help to enhance their performance. However, distributing VMs far away from each other can end up with more active servers to host VMs. As a result, additional power consumption, operational, and maintenance costs for servers will be added for CSP. In addition, the network bandwidth and delay will increase between communicated VMs that also increasing the response time of the service. VM placement problem can be static or dynamic. Static placement is done only for the deployment of VMs that belong to a newly requested application service. While dynamic VM placement is triggered at any time for an already existing and hosted VM that provides a service. In some cases, current VM placement may need to be changed for several reasons such as VM scaling and migration requests. In this thesis, we tackle dynamic VMs placement NP-hard problem to achieve multiple objectives. The target is to minimize the power consumption, resources wastage, and failure ratio of the active servers that are used for placement of the VMs, without violating the availability requirements of the services that are provisioned through the VMs.

1.3 Research Objectives

In this thesis, the research works target the following objectives:

- Study, analyze, and classify research works in the area of service availability management and fault tolerance in cloud computing, for better understanding and build knowledge for the existing works that include proposed solutions, techniques, and approaches.
- Maintain availability of the application service during its lifetime.
- Enable service continuity to resume service from last updated state.
- Conduct experimental study to compare between VM and container to highlight their roles for service availability.
- Determine the main metrics to get service availability measurements.

- Formulate application service availability in the cloud computing platform.
- Predict termination status of applications tasks during the execution time.
- Minimize the failure ratio of tasks that are executed in cloud cluster compared to the existing solutions.
- Provide service availability above and near to the availability requirements of the ASP.
- Admit new requested applications, and increase the overall profit of the CSP.
- Enhance the power consumption of the active computing servers that are located at DCs, and used to host VMs that provision the services.
- Utilize resources and enhance the performance of the servers.
- Compare the results of the proposed algorithms, approaches, with existing and related solutions from the literature.
- Develop models, and prototypes to run experiments and get the results as a proof of concept, and to make comparisons.

1.4 Research Contributions

The main contribution of this thesis is to propose a general framework to enable, manage, and enhance application service availability in cloud computing. The contributions of this thesis can be generally divided into four main categories: reactive application service availability and continuity management in the cloud platform, proactive service availability management, failure-aware application task scheduling in cloud cluster, and availability-aware dynamic VM placement in DCs of cloud. In the following, the detailed contributions of the thesis are listed per each chapter. They are considered contributions because some of the proposed solutions include new methods and techniques that do not exist in the literature. In addition, some of the proposed solutions achieve better results and add new values compared to the existing solutions that are related to service availability.

1.4.1 Related Work (Chapter 2)

- Compare existing application service availability management and fault-tolerant solutions in the cloud.
- Compare existing task failure prediction methods.
- Compare existing failure-aware application task scheduling solutions.
- Compare existing Availability-Aware VM placement solutions.

1.4.2 Reactive Application Service Availability Management (Chapter 3)

- Propose a general framework to manage availability, and enable continuity of the application service during its entire lifetime. The framework has the ability to monitor the application component that provides the service, and detect its failure. In addition, it uses different redundancy models to failover and recover the service in case of failure.
- Integrate two distributed open-source availability middlewares named OpenSAF and Pacemaker to enable HA for application services.

1.4.3 Proactive Service Availability Framework (Chapter 4)

- Study and analyze three publicly available computing cluster datasets to determine the main characteristics and patterns of application task failure in the cloud platform.
- Propose a proactive failure-aware framework to predict the termination status of given tasks during the runtime.
- Identify the main features that are related to the task failure.
- Use deep learning methods named Artificial and Convolutional Neural Network, ANN, and CNN to predict task failure.

1.4.4 Failure-Aware Application Task Scheduling (Chapter 5)

- Formulate application task scheduling problem as Integer Linear Programming (ILP) optimization model with the objective to minimize failure ratio of tasks, and their resources usage at the same time.
- Propose a heuristic optimization algorithm to schedule tasks that are predicted as failed.
- Provide a proof of concept that includes statistical and empirical results to evaluate the proposed framework and methods.

1.4.5 Dynamic VM Placement for Application Service Availability (Chapter 6)

- Formulate the problem of computing application service availability in cloud platform.
- Propose a framework for dynamic VM placement for application service availability in the cloud.
- Formulate dynamic VMs placement problem as Integer Non-Linear Programming (INLP) optimization model with multiple objectives.
- Use AntColony heuristic optimization algorithm in conjunction with the VM standby protection approach to solve multiple objectives VMs placement problem efficiently.
- Develop a prototype to evaluate the proposed framework, algorithms, methods, and comparison with existing VM placement solutions from the literature.

1.5 Thesis Organization

The remainder of this thesis is organized as follows: In Chapter 2, we discuss the related work for application service availability and fault tolerance solutions in the cloud. In Chapter 3, we propose a reactive service availability management framework. In chapter 4, we propose a proactive service availability management framework. In Chapter 5, we propose optimization models and solutions for application task scheduling problem in cloud cluster. In chapter 6, we propose optimization models and solutions for dynamic VM placement problem from service availability perspectives in the cloud. The conclusion and future works are presented in chapter 7.

Chapter 2

Related Work

The industry and academic researchers proposed different solutions, algorithms, models, approaches that are related to availability, fault tolerance, reliability, protection, recovery, and continuity of application service in distributed systems in general and in cloud computing in particular. In this chapter, we study, analyze and categorize the related works for service availability management. We categorize the works into five main research areas that are closely related to service availability. In each area, we discuss the scope, contributions, metrics, and results of the works. The rest of this chapter is organized as follows. Section 2.1 discusses the existing solutions for reactive application service availability management. Section 2.2 discusses the main research works that address proactive service availability management. Section 2.3 discusses the existing failure-aware and fault tolerance solutions for task scheduling. Section 2.4 discusses the existing algorithms and techniques that address the availability-aware VM placement problem. Section 2.5 discusses the existing solutions that tackle the VNF placement problem for network service availability in Network Function Virtualization (NFV) platform, and Section 2.6 summarizes the major findings of studying the related work.

2.1 Reactive Application Service Availability Management

The huge and fast growing demands for deployment applications in the cloud motivated some IT businesses and researchers to propose solutions to maintain application service availability and continuity. VMware High Availability (VMwareHA) [7] is a cost-effective solution proposed by VMware [8] for managing HA of application in VM-based cluster. VMwareHA solution has the ability to protect the application service against both VM and physical computing host (server) failures. It can monitor the operation status of VM and host, and detect their failures. In case of host failure, VMwareHA evacuates all the VMs that are impacted by the failure to another active host where the VMs resume their execution. In case of VM failure, VMwareHA tries to restart the failed VM on the same or different host that the VM can resume its execution. VMwareHA does not necessarily need a redundant standby VM to protect the service in case of failure of the active VM that is in charge of provisioning the service. Therefore, it is considered a cost-affective solution for service HA. However, VMwareHA does not protect the application service against application component failure. The component represents the software process that provides the service and can be hosted at a computing node whether virtual or physical. VMwareHA does not support a monitoring mechanism to detect the application component failure, so it can not protect the service in such type of failure. Also, VMwareHA does not support service continuity because it does not depend on standby VMs to save and retrieve the state of the service. VMwareFault Tolerance (VMwareFT) [9] is another HA solution for application in VM-based cluster that is proposed by VMware [8]. VMwareFT depends on the existence of a standby VM that is continuously synchronizing its state with the current state of the active VM. The main role of the standby VM is to take over and continue providing the service in case of failure of the active VM. VMwareFT, therefore, supports service protection against the host, VM, and component failures. In addition, it enables the service to continue from the last updated state of the service before it gets interrupted due to any type of failure. However, VMwareFT is considered a costly solution because it requires standby VM(s) that consume more resources, and power that adds extra cost. In addition, VMwareFT consumes a lot of network bandwidth to synchronize the service state between active and standby VMs. HA-Lizard [10] is an open-source solution for HA management of application service that is provided through XenServer virtualization cluster [11]. HA-Lizard has the ability to detect VM and host failures, but not for the application component failure. Although HA-Lizard supports service continuity, its scalability is limited by a small cluster of a few physical servers. Table 2.1 summarizes the protection features of the existing HA solutions for application in VM-based cluster.

Table 2.1: Protection Features of Application HA Solutions

Service HA Solution	App Component Failure	VM Failure	Host Failure	Service Continuity
VMwareHA	No	Yes	Yes	No
VMwareFT	Yes	Yes	Yes	Yes
HA-Lizard	No	Yes	Yes	Yes

Cully et al. [12] proposed HA solution named Remus to manage the availability of application that is deployed in a virtual cluster. Remus uses the same fault tolerance synchronization technique of VMwareFT HA solution. Although Remus has the ability to detect application component failure, its synchronization process between redundant VMs consumes a lot of computing power, and network bandwidth that can affect the overall performance of the cluster and the quality of the running applications. Kanso and Lemieux [13] proposed a dynamic approach to enable availability feature for application in the cloud platform. Authors target to offer application availability management as a service in the cloud. The approach considers the dynamic nature of the cloud, so it can add the availability feature on the fly for any new requested application. They use open source HA middleware [14] to manage the life cycle of application component, and failover of the application service in case of component failure. However, authors do not provide any availability measurements for the proposed solution. Li and Kanso [15] presented a comparison between VM and container from service availability perspectives in cloud. They highlighted the main features and limitations of existing solutions. They concluded that HA feature in container-based cluster is not developed enough yet. So the same authors in [16] proposed HA solution that leverages the Linux containers to manage availability of application that is deployed in container-based cluster. The proposed solution restarts the failed container from its last updated image to recover the service, and enable its continuity. Authors do not provide any measurements to evaluate their proposed solution.

2.2 Proactive Service Availability Management

Using redundant resources approach to protect the service can add additional costs for operation, maintenance, and power consumption of the resources for the CSP. Therefore, some of the existing research works proposed proactive solutions to predict application failure before it happens, and take the appropriate protective actions. In general, a proactive approach is considered cost-effective because it does not necessarily require adding standby (backup) resources. There are existing research works in the literature that applied statistical, machine and deep learning methods using Google dataset [6] to predict application task failure. Chen et al. [17] studied the main features of application job and task failures in cloud computing. Authors analyzed events and resource usages of the jobs and tasks to determine features related to the failures. They applied deep learning Recurrent Neural Network (RNN) to predict job and task failures. RNN achieved an accuracy of around 84%. Soualhia et al. [18] explored the possibility of predicting application task failure in cloud platform to enhance the performance of resources that are used to execute tasks. The authors applied a set of statistical and machine learning models such as Decision Tree (DT), Boost, and Random Forest (RF) to predict task failure. The results show RF achieved the highest prediction accuracy of up to 95.8%, precision up to 97.4%, and recall up to 96.2%.

Jassas and Mahmoud [19] [20] proposed a framework to predict job failure in cloud. The framework compares between a set of machine learning models: DT, Logistic Regression (LR), K-Nearest Neighbors (K-NN), Naive Bayes (NB), RF, and Quadratic Discrimination Analysis (QDA) to select the method that achieves higher failure prediction accuracy. They used datasets from Google, Mustang [22], and Trinity [22] to evaluate the prediction models. Results show DT model achieved a higher prediction accuracy with precision up to 98% and recall up to 86%. Shetty et al. [23] discussed the statistical analysis of tasks resource usage in a distributed computing cluster. They argue that the patterns of the failed tasks are different than the successful tasks. The authors used XGboost classifier to predict task failure with 92% precision and 94.8% recall. Liu et al. [24] discussed the possibility of early task failure prediction in the cloud. They clustered jobs according to their similarity using Fuzzy C-Means (FCM). They consider each cluster as a separate dataset for training and testing the prediction models. They conducted a comparison between three prediction models named, extreme learning machine (ELM), support vector machine (SVM), and LR to predict task termination status at an early stage of its execution time. Results show both ELM and SVM achieved 89.96% accuracy, while LR achieved 80.93% accuracy. Gao et al. [25] used a deep learning method based on multi-layer Bidirectional Long Short Term Memory (Bi-LSTM) model to predict job and task failures. Authors extracted static features and derived dynamic ones for jobs and tasks to model training and testing data sets. Results show Bi-LSTM achieved a high accuracy of up to 93% to predict task failure, and up to 87% to predict job failure. Rosa et al. [26] analyzed patterns of failed jobs in the cloud. They inspected the workload of jobs and system attributes to identify root causes of job failure. Authors developed three online prediction models based on deep learning neural networks to classify job termination state upon its arrival time. The evaluation results show the prediction models can classify job with 94.4% accuracy, and classify task with 76.8% accuracy.

Islam and Manivannan [27] identified key features that are associated with the application failure in the cloud. They applied LSTM model to predict application termination status before it terminates. The results show LSTM achieved up to 87% accuracy. Liu et al. [28] claim that the offline working patterns that are adopted by many machine learning prediction models can not be used for online prediction in practical operations of the cloud platform. Therefore, the authors proposed a new method based on an online sequential extreme learning machine (OS-ELM) to predict job termination status. Results show OS-ELM achieved up to 93% accuracy. El-Sayed et al. [29] analyzed three log traces from large computing clusters of Google, Hadoop, and Trinity to know the root causes of jobs and tasks failures. The authors used RF to classify failure at job and task levels. Results show RF can predict failure with a recall of up to 94% and precision of up to 95%. Hongyan et al. [30] compared between Boosting Decision Tree (GBDT), KNN, RF, and LR machine learning classifiers to predict misconfiguration that can lead to job failure. The classifiers are evaluated using OpenCloud dataset. Results show GBDT achieved a high accuracy of up to 92%, 78% precision, and 52% recall. Padmakumari and Umamakeswari [31] applied different machine learning classifiers to predict task failure of scientific applications. The authors simulated dataset to train and test classifiers. Results show NB classifier achieved a high accuracy of up to 94.9%.

Although the above research works applied different methods using different datasets to predict failures of application jobs and tasks, they do not address the remedy actions to protect these jobs and tasks that are predicted as failed. Our work is distinct from the above discussed works, where we propose a set of remedy actions and immediate rescheduling for tasks that are predicted as failed. The works [32] [33] addressed mitigation actions for the failed jobs in Google dataset. Rosa et al. [32] applied the statistical methods named, Linear Discrimination Analysis (LDA), Quadratic Discrimination Analysis (QDA), and LR to predict job termination status online. The authors proposed to immediately terminate any job that is predicted as unsuccessful as a mitigation action. However, we believe only terminating the job is not a feasible solution which the end user may not accept. Islam and Manivannan [33] applied LSTM to early forecast the termination status of a task during its execution. The authors proposed to reschedule the task that is predicted as failed on a highly reliable computing node as a protection action. Results show LSTM achieved an accuracy of up to 91%. Our work is different than [32] [33], where we consider a variety set of remedy actions to increase the chance of tasks to terminate successfully. Soualhia et al. [34] proposed a fault tolerance task scheduling framework (ATLAS) for Hadoop platform. ATLAS uses different machine learning models named RF, NN, Conditional Tree (CT), Boost Tree (BT), and General Learning Model (GLM) to predict application task failure. The models are trained and tested using a real Hadoop cluster dataset. ATLAS can dynamically request to reschedule task that is predicted as unsuccessful. Results show ATLAS reduced the number of failed jobs up to 43% and failed tasks up to 59%. RF achieved the highest accuracy of up to 79.9% for map tasks, and up to 94.12% for reduced tasks. Marahatta et al. [35] applied deep learning neural network (DLNN) approach to predict task failure according to the requested resource of the task. In addition, the authors proposed a scheduling algorithm based on vector bin packing for the tasks that are predicted as failed. DLNN achieved 84% accuracy using Internet Dataset. Table 2.2 summarizes above discussed work since they are closely related to our work. Clearly, none of the works consider optimization of multiple remedy actions that can be taken for the tasks that are predicted as failed during the runtime.

There are few existing research works in the literature that proposed solutions for infrastructure failure prediction in the cloud computing platform. Actually, predicting physical infrastructure failure is very necessary for taking service protection actions, such as migrating VMs and rescheduling tasks on higher reliable infrastructure. Soualhia et al. [36] proposed a framework to predict different types of failures at the infrastructure level in the edge computing platform. The framework uses SVM and RF to detect CPU and hard drive (HD) overload faults. Samir and Pahl [37] presented a model to predict anomalies at the infrastructure level in the edge computing platform. The model uses Hidden Markov Model (HMM) to predict the anomalies. Mohammed et al. [38] applied SVM, RF, and K-NN on time series datasets from the computer failure data repository (CFDR) to predict infrastructure failure in virtualized HPC system. Li et al. [39] applied HMM to predict VM failure to enhance the reliability and performance of the cluster. Wang et al. [40] applied SVM to predict equipment failure in optical networks.

Table 2.2: Summary of Related Work for Task Failure Prediction

Reference	Method	Dataset	Accuracy	Precision	Recall	Apply Remedy Actions
Liu et al. [24]	ELM, SVM, LR to predict task failure	Google	ELM and SVM highest 89.96%	X	X	X
Chen et al. [17]	RNN to predict job and task failure	Google	81% for job, and 84% for task	X	X	X
Soualhia et al. [18]	DT, Boost, and RF to predict task failure	Google	RF highest 95.8%	97.4%	96.2%	X
Jassas and Mahmoud [19] [20]	DT, LR, K-NN, NB , RF and QDA to predict job failure	Google, Mustang, and Trinity	X	DT highest 98%	DT highest 86%	X
Shetty et al. [23]	XGboost to predict task failure	Google	X	92%	94.8%	X
Gao et al. [25]	Bi-LSTM to predict job and task failures	Google	Job 87% and task 93%	X	X	X
Rosa et al. [26]	NN To predict job and task failures	Google	Job 94.4% and task 76.8%	X	X	X
Islam and Manivannan [27]	LSTM to predict task failure	Google	87%	X	X	X
Liu et al. [28]	OS-ELM to predict job failure	Google	93%	X	X	X
El-Sayed et al. [29]	RF	Google, Hadoop, and Trinity	X	95%	94%	X
Hongyan et al. [30]	GBDT, KNN, RF and LR	OpenCloud	92%	78%	52%	X
Padmakumari and Umamakeswari [31]	NB	Simulated dataset	94.9%	X	X	X
Rosa et el. [32]	LDA, QDA and LR to predict job failure	Google	X	X	X	Termination of job
Islam and Manivannan [33]	LSTM to predict task failure	Google	91%	X	X	Reschedule task
Soualhia et al. [34]	RF, NN, CT, BT and GLM to predict task failure	Hadoop	RF for map task 79.9%, reduced task 94.12%	X	X	Reschedule task
Marahatta et al. [35]	DLNN	Internet Dataset	84%	X	X	Reschedule task

'X' denotes not considered

2.3 Failure-Aware Task Scheduling

Different existing research works in the literature addressed application task scheduling from different objectives such as fault tolerance, power consumption, and resource utilization. We are interested in the works and solutions that are related to failure-aware task scheduling. The works [33] [34] [35] proposed rescheduling solutions for the tasks that are predicted as failed. The works [41] [42] [43] proposed fault tolerance solutions for scheduling tasks in cloud to enhance availability of the running tasks. Marahatta et al. [44] developed energy-aware fault-tolerant dynamic application task scheduling (EFDTS) schema at the underlying VMs. The works [45] [46] proposed an online application task and job scheduling mechanism based on deep reinforcement learning (DRL) to assign submitted tasks at different VMs from different QoS perspectives. Frincu and Craciun [47] proposed multiple objective algorithm for scheduling application tasks at VMs. The objectives are to achieve HA for application, maximize resource utilization, and reduce the cost to provide the application.

Rjoub et al. [48] used DRL and LSTM to predict for which VM the task should be scheduled with the goal to utilize virtual resources that helps to increase the termination success ratio of tasks. Pham et al. [50] introduced a two-stage machine learning approach to predict the execution time of task workflow that helps to make a decision for task scheduling in cloud cluster. Khan et al. [50] studied Google dataset to compare different task scheduling policies from the power and cost-saving perspectives in heterogeneous hybrid DCs. Sebastio et al [51] compared different configuration models for deploying software system in the container-based platform, while considering the availability of the system. They proposed stochastic model to analyze and estimate the system availability for each configuration deployment plan to make a decision. Zhang et al. [52] discussed the importance of application task scheduling for QoS in Cyber Physical Cloud Systems (CPCSs). The authors proposed a scheduling algorithm based on Priority Queuing Model with Negative Arrival for real-time CPCS tasks. The simulation results show the ability of the algorithm to reduce the average waiting time of task to be scheduled for execution. Hao et al. [53] formalized service scheduling and resource allocation in CPCS as a joint optimization model with the goal to minimize the response time of the service. They developed deep learning Q-network (DQN) algorithm to tackle service scheduling. Kuang and Zhang [54] proposed a task scheduling algorithm in CPCS based on task priority using value density in a real-time distributed task system. Zhou et al. [55] proposed an algorithm for scheduling workflow of the applications in CPCS. The objective is to maximize reliability and increase the security of the applications. Results show the ability of the scheduling algorithm to reduce task failure probability up to 52%. Yang et al. [56] proposed a multi-objective optimization task scheduling method for Cyber-Physical-Social services in the fog computing platform. The method aims to minimize the execution time and resource consumption of the scheduled tasks.

2.4 Availability-Aware VM Placement

There are several existing research works in the literature that tackled VM placement problem from different perspectives such as resources power consumption and operational cost, network bandwidth, and delay. However, a limited number of the works addressed end-to-end application service availability. Therefore, we highlight research works that tackle VM placement problem and consider availability, reliability, and fault tolerance solutions for application service in cloud computing. Jammal et al. [57] [58] proposed CHASE a component high availability-aware scheduler in the cloud environment. Authors map the scheduling problem as Integer Linear Programming (ILP) model with the objective to maximize the availability of the components. CHASE searches for the servers with maximum availability to schedule the components. The authors used IBM ILOG CPLEX optimization solver to find the best scheduling plan for the components. Zhu and Huang [59] addressed availability concerns of Mobile Edge Computing (MEC) application during the placement process of application's components. The authors proposed a stochastic model to track the cost and availability impact of changing the placement of the components. They used FirstFit and BestFit heuristic algorithms for the placement of MEC application. Lera et al. [60] addressed service placement for application fault tolerance in the fog computing platform. The authors proposed a two-phase placement strategy based on graph partitioning and traversal approach to optimize the placement process.

The works [61] [62] tackled reliability of VM placement (RVMP) problem. Yang et al. [61] proposed INLP model to find the minimum number of the computing nodes to host the VMs, and guarantee the availability of the VM placement plan to be above a requested requirement, and the communication delay between VMs peers is below a specific threshold. They used CPLEX to solve RVMP model. Liu et al. [62] mapped VM placement as ILP model with the objectives to minimize communication traffic and network bandwidth in DC, and increase reliability of the hosted VMs. The authors used the graph k-cut approach to solve ILP model. Yang et al. [63] developed a variance-based metric to measure the risk of violating application availability during VM placement process. The authors considered the risk of Top-of-Rack (ToR) switch and server failures in DC. They formalized VM placement as ILP model with the objective to minimize computing resources power consumption, and enhance the availability level for the requested applications. They proposed the concept of packing then distributing (PTD) the VMs during the placement.

The works [64] [65] [66] [67] [68] proposed cloud fault-tolerance solutions through VM placement, where they mapped the problem as ILP model. Li and Qian [64] addressed the placement problem for the multitenant cloud with the goal to reduce network traffic in DC. Jammal et al. [65] tackled the placement problem of VMs during the live migration process. The authors used VM live migration approach to mitigate service downtime in case of failure. They used CPLEX to find the destination servers for the migrated VMs. Zhou et al. [66] [67] addressed the optimal redundant VM placement (ORVMP) problem with the objective to minimize network resources consumption and increase the reliability of cloud service. The authors used genetic algorithms to solve ORVMP model. Gonzalez and Tang [68] used FirstFit algorithm for placement of VM replicas (backups) for service fault tolerance solution at DCs. Alameddine [69] proposed a protection plan that can determine the number and placement of the backup VMs, to guarantee bandwidth demands and meet availability requirements of the critical applications in the cloud. The authors mapped the problem as INLP model, and proposed a heuristic algorithm to solve the model. The works [70] [71] formalized a cost function to tackle VM placement problem. Chen and Jiang [70] proposed an adaptive selection method for application service fault-tolerance plan during VM placement process. The cost function targets multiple objectives that include minimizing application response time, resources usage consumption, and application failure rate. Zhang et al. [71] tackled VM placement in cloud DC of star topology to achieve service fault tolerance. The cost function targets multiple objectives that include minimizing SLA violation, resources power consumption, and failure rate.

2.5 VNF Placement for Network Service Availability in NFV

The works [72] [73] addressed VNF placement problem in NFV platform for network service availability, where they mapped the problem as INLP model. Ayoubi et al. [72] proposed a framework for embedding elastic and reliable virtual networks (VNs) in the cloud. The framework handles embedding of new VN request and only scaling up request of an existing VN in DC. Authors model VN as a composition set of connected VNFs, where each VNF is mapped as one VM. The goal of the model is to meet the availability requirement of VN during its entire lifetime and increase the admissibility chance of new VNs. The authors proposed to add backup VNFs, and use a tabu-search optimization approach for reliable VNF placement. Thiruvasagam et al. [73] addressed reliable virtual monitoring functions (vMFs) placement problem based on NS topological information. The main objective is to minimize the communication delay between Service Function Chains (SFCs) that compose NS, and the number of vMFs that are required to monitor NS. The authors used CPLEX to find the best placement plan of vMFs. The works [74] [75] [76] [77] [78] formulated VNF placement problem as ILP model. Yala et al. [74] used genetic algorithm to find VNF placement for virtual Content Delivery Network (vCDN). The objective is to find the trade-off between vCDN deployment cost and its availability level. Yang et al. [75] tackled stateful VNF placement for NS fault-tolerance. They mapped the problem as an optimization function with the goal to increase the user requests. Yansen et al. [76] proposed availability-aware SFC placement scheme at substrate network in NFV. The main objective is to minimize the end-to-end delay of SFC. The authors used a layered graph search approach to locate the placement of SFC. Sharma at al. [77] targeted the high availability of NS in NFV during VNF placement. The goal is to maximize the profit of the Telecom Service Provider (TSP), subject to the availability demand of NS by the service operator. The authors use redundant VNFs and geographic placement approach to solve the placement problem. Abdelaal et al. [78] tackled VNF Forwarding Graph (VNFFG) deployment problem, where they consider both VM and server failures in NFV platform. They target the deployment that achieves multiple objectives, minimize network bandwidth, convergence time to allocate/deallocate VNFs, and power consumption of the resources. They used redundant VNFG to protect VNF service against failures. Mao et al. [79] proposed online fault-tolerant SFC placement solution in NFV. The authors formulated the problem as Markov decision process (MDP) model with the objective to maximize the number of accepted user requests. They proposed a deep reinforcement learning (DRL) method for the placement of active and standby SFC instances. Table 2.3 summarizes the related work that are discussed in sections 2.4 and 2.5. Note that none of the existing research works compute application service availability in the cloud to meet the requirements during the component, VM, or VNF placement process.

Table 2.3: Summary of Related Work for VM placement

Research Topic	**Reference**	**Compute Application Availability**
Application Component Placement	[57], [58], [59], and [60]	X
Reliability-Aware VM Placement	[61], [62] and [63]	X
Fault Tolerance and Service Availability using VM Placement	[64], [65], [66], [67], [68], [69], [70], and [71]	X
VNF Placement for Network Service Availability in NFV	[72], [73], [74], [75], [76], [77], [78] and [79]	X

’X’ denotes not considered

2.6 Summary

This chapter summarizes the main works and solutions that exist in the literature and are related to application service availability management in cloud computing. The works are categorized into five main related research topics: reactive application availability management, proactive service availability management, failure-aware task scheduling, availability-aware VM placement, and VNF placement for network service availability in NFV. The main finding after exploring the related works can be summarized as follows. Most of the existing reactive availability solutions focus on virtual and physical infrastructure failure detection and do not consider failure at the application component level. Although there are multiple proposed solutions for failure-aware task scheduling in cloud cluster, a few works considered remedy actions to protect tasks. According to our best of knowledge, there is no work that formalizes application availability in the cloud platform. In addition, limited works considered application availability requirements during the VM placement process.

Chapter 3

Reactive Application Service Availability Management

This chapter introduces our proposed reactive framework to incorporate HA feature, and enable service continuity for application service in cloud virtual cluster. The framework includes four main modules that consider the availability requirements of tenant towards the deployment of HA application. The framework integrates two well-known open source HA middlewares with VMs and containers to manage service availability. The middlewares use different types of redundancy models to recover the service in case of VM/container failure. In addition, the chapter discusses the experiments that have been done in real virtual clusters, and availability measurements of the proposed framework. The rest of this chapter is organized as follows. Section 3.1 introduces service availability management. Section 3.2 presents the proposed framework including its modules. Section 3.3 describes the real experiments and discusses the measurements and results, and Section 3.4 gives a summary of the chapter.

3.1 Service Availability Management

Before we introduce our reactive framework for service availability management, we introduce the redundancy models and HA middlewares that are used by the framework.

3.1.1 Redundancy Models

The redundancy model defines number and the role type of components that are used to provide and protect the service. Usually, availability management service uses a specific configuration that defines the used redundancy model(s). Specifications of availability management framework (AMF) defines a standard set of redundancy models [80]:

- 2N: allows only one active component which provides all the services and only one standby component. The standby component takes over and continues providing the services in case of the failure of the active component.
- N+M: allows N number of active components which provide the services, and M standby components. The service can be failed over to any of the standby components that is capable of providing the same type of service.
- N-Way-Active: allows N active components to provide the same type of the service. In this redundancy model, no standby components are configured.
- N-Way: allows N components in which the component can be active for some services and standby for other services at the same time.

3.1.2 OpenSAF

OpenSAF is an open-source middleware that implements a set of standard service specifications by the Service Availability Forum (SAF) [14] to manage the availability of applications services in a distributed cluster system. AMF is the core service of OpenSAF is responsible for managing the availability of the service through controlling the life-cycle of the components that are in charge of providing the service. AMF uses a specific configuration (a.k.a AMF configuration) as XML file, where it defines one or more of the above redundancy models.

3.1.3 Pacemaker

Pacemaker is a well-known middleware for managing the availability of services in distributed cluster systems [17]. Many CSPs use the Pacemaker as an availability manager for the provisioned services to their tenants. Pacemaker is logically responsible for managing the life-cycle of the component(s) which provide the service within a cluster of computing nodes (virtual or physical). Pacemaker has the ability to detect the node failure using the Corosync communication service [82] and the Heartbeat monitoring service. Corosync service provides a closed process cluster communication model with synchronization capabilities between the processes of the same cluster. Heartbeat service is a daemon that provides inquiry service for the membership existence of the cluster resources. Pacemaker supports the configuration of the above redundancy models for the application components to protect the services against failures. Pacemaker describes the configuration through XML file.

3.2 Reactive Service Availability Management Framework

We propose a framework to incorporate availability, and enable continuity features for application service in virtual cluster. Figure 3.1 depicts the framework that includes four main modules: Tenant Requirements, Availability Configuration Manager, Deployment Manager, and the HA middleware.

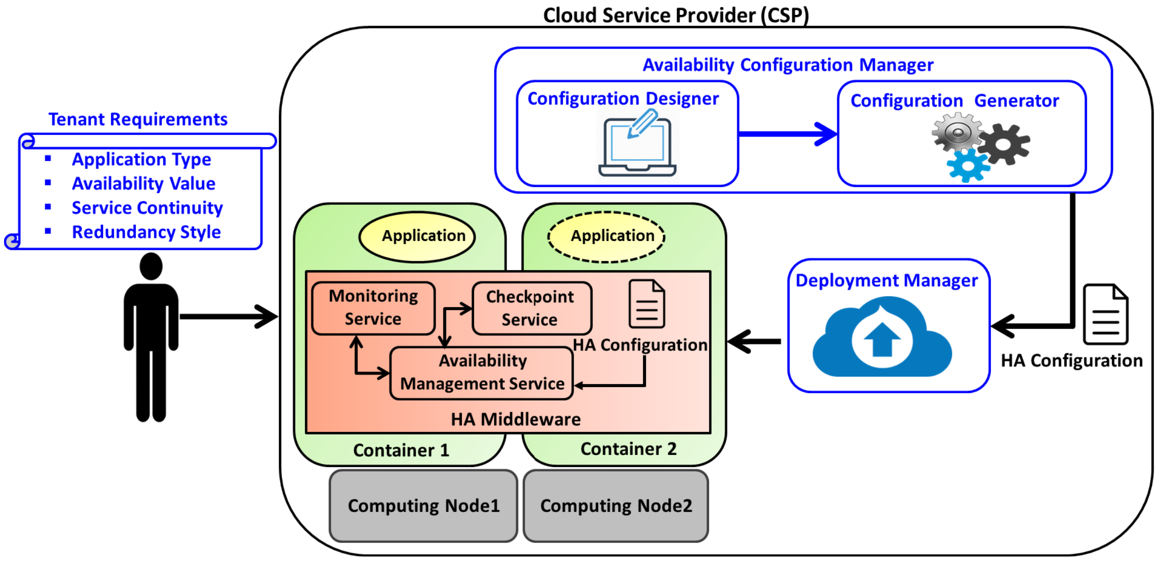


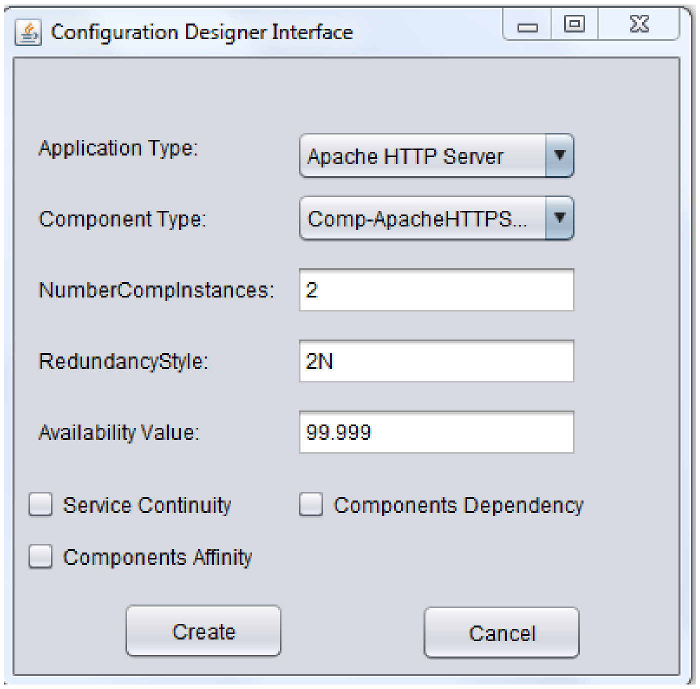
Figure 3.1: Reactive Application Service Availability Management Framework

3.2.1 Tenant Requirements

From the service availability perspective, the cloud tenant such as ASP can request a specific type of application service with availability requirements, for example, 5 nines (99.999%), during a specific period of time. In addition, the tenant can determine the redundancy model of the application components to protect the service against different types of failures. For example, tenant can determine the number of active and standby application components to provide and protect the service. Moreover, the tenant can request the service continuity feature in case of the stateful application which has a state that can be saved and retrieved at any point in time.

3.2.2 Availability Configuration Manager

The availability Configuration Manager module is responsible for the design and generation of the configuration for the HA middleware that is used by the CSP. The module includes two main functional units: Configuration Designer and Configuration Generator. The Configuration Designer is responsible for designing a high abstract level of HA configuration to provide and protect the services according to the tenant's requirements. Basically, the Configuration Designer is an expert person who has knowledge about the tenant requirements and the resources that are available at the CSP including the HA middleware. The designer can specify information about the type and number of the application components that are required to exist to provide and protect the requested services. In addition, the designer can determine information about the placement of the components. For example, design information can state whether two components from the same application type can have an affinity or anti-affinity relationship. Affinity relationship means the components should be located together on the same computing node, and the anti-affinity relationship means the resources should not be located together on the same computing node. Moreover, the designer can state the dependency relationships between the components, and hence this dependency should be respected during the deployment and the lifetime of the components. For example, when component A depends on component B, this indicates for the Deployment Manager module that component B should be deployed (existed) for component A to operate. We implemented an interface as shown in Figure 3.2, where the Configuration Designer can use it to design the intended configuration.


Configuration Designer Interface

Application Type: Apache HTTP Server

Component Type: Comp-ApacheHTTPS...

NumberCompInstances: 2

RedundancyStyle: 2N

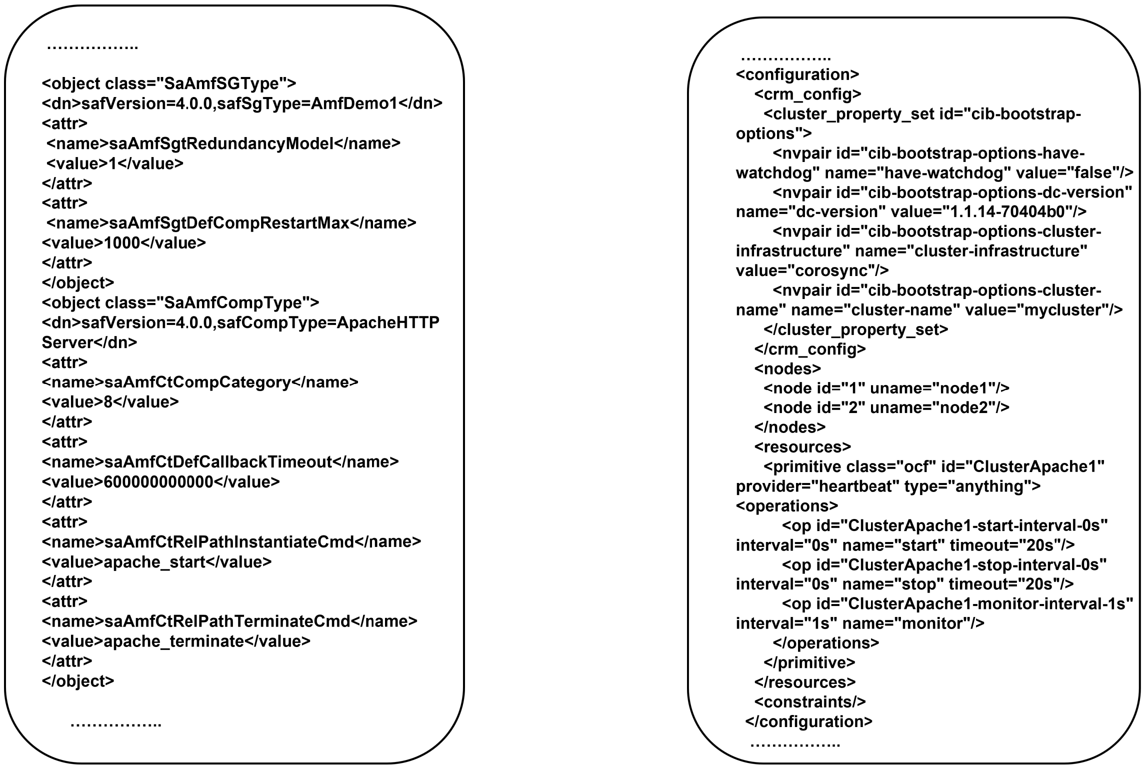
Availability Value: 99.999

Figure 3.2: HA Configuration Design and Generate Interface

The Configuration Generator unit is responsible for automatically generating the configuration of the corresponding HA middleware according to the configuration design by the Configuration Designer unit. We build a procedure to automatically transform the HA configuration design to a formatted HA configuration to be used by the corresponding HA middleware. Configuration is a structural description for the resources including the applications components, services, and the relationships between them. Configuration can include information about the identifications and names of the resources and the services. The configuration also can include availability actions that are used by the availability management service of the HA middleware in case of component failure such as restart/shutdown of the failed component. In addition, it can include the absolute paths for the software's scripts to start/stop the components. Figure 3.3a shows a part of OpenSAF configuration, and Figure 3.3b shows a part of Pacemaker configuration.

3.2.3 Deployment Manager

The Deployment Manager module takes the generated HA configuration by the Configuration Generator unit as input and deploys all the resources on their corresponding placement positions as specified in the configuration. In addition, the module considers all types of



```
................

<object class="SaAmfSGType">
<dn>safVersion=4.0.0,safSgType=AmfDemo1</dn>
<attr>
 <name>saAmfSgtRedundancyModel</name>
 <value>1</value>
</attr>
<attr>
 <name>saAmfSgtDefCompRestartMax</name>
<value>1000</value>
</attr>
</object>
<object class="SaAmfCompType">
<dn>safVersion=4.0.0,safCompType=ApacheHTTP
Server</dn>
<attr>
<name>saAmfCtCompCategory</name>
<value>8</value>
</attr>
<attr>
<name>saAmfCtDefCallbackTimeout</name>
<value>600000000000</value>
</attr>
<attr>
<name>saAmfCtRelPathInstantiateCmd</name>
<value>apache_start</value>
</attr>
<attr>
<name>saAmfCtRelPathTerminateCmd</name>
<value>apache_terminate</value>
</attr>
</object>

    ................
```

(a) Part of OpenSAF Configuration

```
 ................
<configuration>
   <crm_config>
     <cluster_property_set id="cib-bootstrap-
options">
        <nvpair id="cib-bootstrap-options-have-
watchdog" name="have-watchdog" value="false"/>
        <nvpair id="cib-bootstrap-options-dc-version"
name="dc-version" value="1.1.14-70404b0"/>
        <nvpair id="cib-bootstrap-options-cluster-
infrastructure" name="cluster-infrastructure"
value="corosync"/>
        <nvpair id="cib-bootstrap-options-cluster-
name" name="cluster-name" value="mycluster"/>
     </cluster_property_set>
   </crm_config>
   <nodes>
     <node id="1" uname="node1"/>
     <node id="2" uname="node2"/>
   </nodes>
   <resources>
     <primitive class="ocf" id="ClusterApache1"
provider="heartbeat" type="anything">
<operations>
          <op id="ClusterApache1-start-interval-0s"
interval="0s" name="start" timeout="20s"/>
          <op id="ClusterApache1-stop-interval-0s"
interval="0s" name="stop" timeout="20s"/>
          <op id="ClusterApache1-monitor-interval-1s"
interval="1s" name="monitor"/>
        </operations>
     </primitive>
   </resources>
   <constraints/>
  </configuration>
  ................
```

(b) Part of Pacemaker Configuration

Figure 3.3: Slices of HA Configuration

dependencies and constraints between the resources as stated in the configuration. After the deployment process of all resources is completed, the module feeds the configuration to HA middleware module. We implement the Deployment Manager unit as a procedure that takes the HA configuration as input and automatically deploys all the resources as stated in the configuration at the cloud platform of the CSP.

3.2.4 HA Middleware

We integrated two HA middlewares Pacemaker and OpenSAF with the images of VM and container that are used to host the components of the application to manage HA of the services provided by the components. Note that we integrated two HA middlewares for comparison purposes, where only one HA midleware is used at a time. Once the HA middleware takes the HA configuration as input, the middleware starts the application components (active ones) to provision the service to the tenant. During the lifetime of the active component, the checkpoint service of the HA middleware starts check-pointing the current state of the component and stores the state in shared storage. The middleware check-points the state of the active components for only the stateful applications, but not for the stateless applications which have no state. Note that the Pacemaker HA middleware does not have any mechanism to checkpoint the stateful applications. Therefore we develop a checkpoint service that periodically (every one second) reads the current state of the active application component and saves it at a shared Network File Storage (NFS). On the other hand, OpenSAF has already a checkpoint service for stateful applications. In order to be fair with the comparisons between the two HA midlewares, we integrated our checkpoint service with OpenSAF. In addition, the middleware frequently keeps monitoring the health status (liveness) of the active components through the monitoring service of the middleware. In case of failure detection of the active component, the availability management service of the middleware immediately starts the service recovery procedure. Therefore, the availability management service failover the service to the standby component which reads the last state of the active component before it fails from the shared storage and continues providing the service as a new active component. In case of failure of the active component of the stateless application, the standby component takes over and starts providing the service from the beginning of the application. Once the service is recovered again, the management service tries to repair the failed component through a repair action as defined in the configuration such as restarting the failed component. Figure 3.4 illustrates the sequence diagram of the HA middleware module managing the HA of stateful application service that is protected using 2N redundancy model.

3.3 Experiments and Results

3.3.1 Experimental Setup

As a proof of concept, we implemented our proposed HA framework using a private cloud platform at our computing lab. For the setup, we built two cloud platforms, one platform uses the virtual containers and the second platform uses VMs to host the application components. Both platforms have the same physical infrastructure. The infrastructure in our setup includes a physical cluster with two homogeneous nodes. Each node is a Dell workstation with 4 core Intel Xeon 3.6 GHz CPU, 32 GB RAM, and 1 TB ATA HD. The two nodes are connected through Linksys Ethernet switch with 100Mbps speed. CentOS 7.0 Operating System (OS) is hosted on each node for the Container-based platform. We

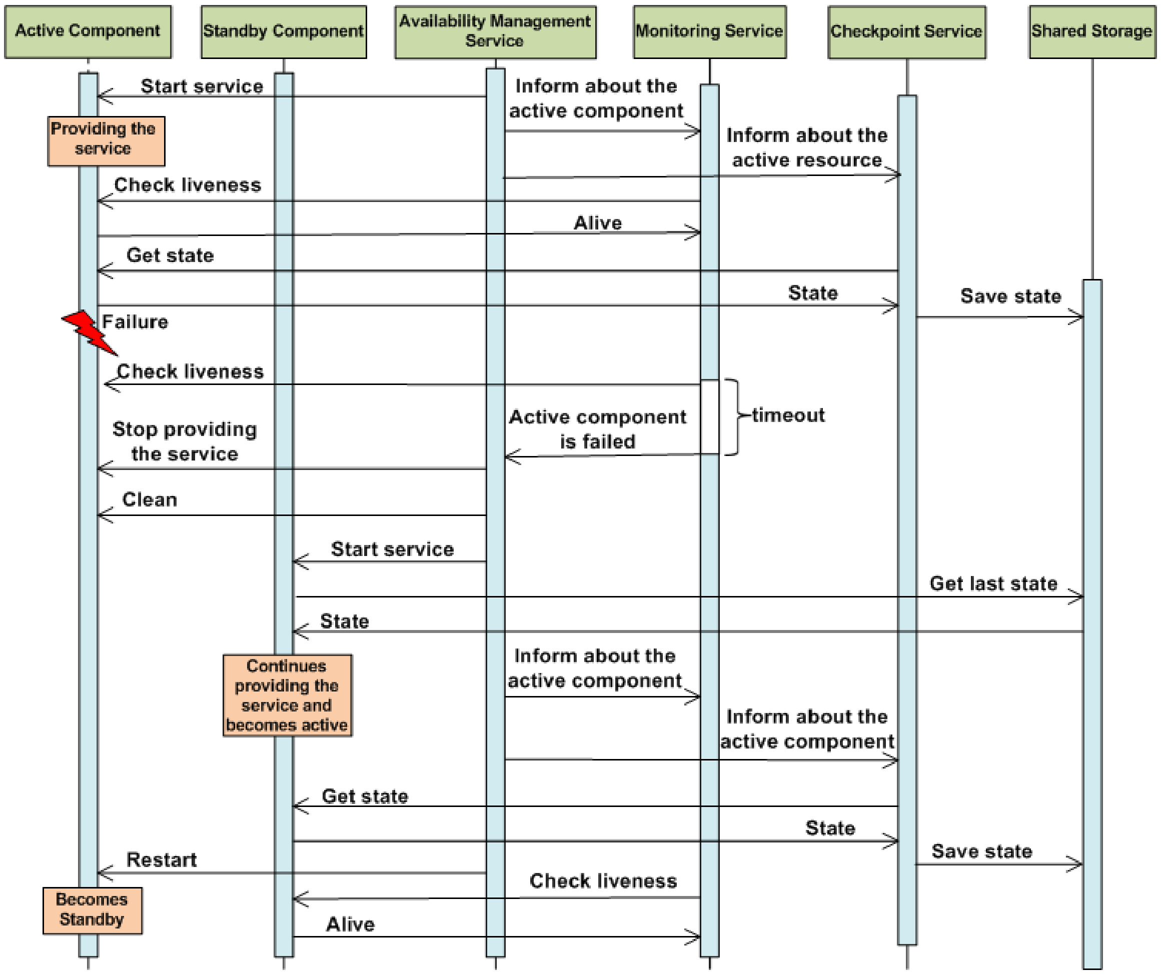


Figure 3.4: Sequence diagram of HA middleware to protect stateful application using 2N

replaced the CentOS by Xen Hypervisor for the VM-based platform because Xen is bare-metal hypervisor. The two nodes share a NFS that is configured on one of the physical hosts. We created LXC linux container image of Ubuntu base OS. We integrated Pacemaker and OpenSAF HA middlewares with the container image. In addition, we integrated VLC media player stateful application and Apache HTTP Server stateless application with the container image. Also, we created a VM image that includes Ubuntu 16.04 OS and the same middlewares and applications that are integrated with the container image.

3.3.2 Service Availability Metrics

For the implementation, we assume the tenant availability requirements are given as input. We design the HA configuration based on the tenant requirements and our knowledge with the OpenSAF and Pacemaker. After that, we pass the configuration design to the Configuration Generator unit. The Configuration Generator automatically generates a formatted HA configuration of the corresponding HA middleware, and then passes the generated configuration to the Deployment Manager module. The Deployment Manager starts automatically deploying all the resources, as described in the configuration, at the corresponding cloud platform on our private cloud setup. To evaluate our proposed framework, we run several types of experiments. The main goal is to prove the ability of the framework to manage HA of different types of applications services that are deployed at different cloud platforms and enable service continuity for the stateful applications. Therefore, we use the following service availability metrics that are defined by [80] and illustrated in Figure 3.5 to get the measurements:

- **Failure Time**: time when the active component failure happens.
- **Reaction Time**: time period for the first reaction by the HA middleware for the failure.
- **Service Recovery Time**: time period to recover the interrupted service since the reaction time.
- **Service Outage Time**: time period for the entire interruption (absence) of the service until its recovery. In other words, service outage time is the summation of the reaction time and the service recovery time.

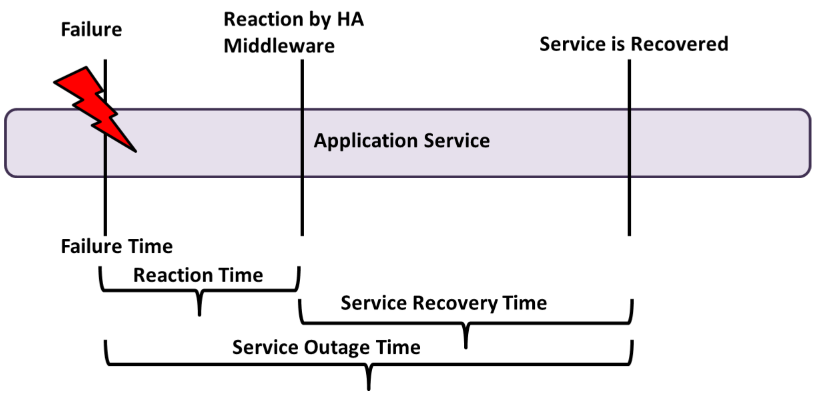


Figure 3.5: Service Availability Metrics

3.3.3 Experiments with Cloud Container-based Platform

For the experiments using the cloud Container-based platform, we designed two HA configurations for the Pacemaker, one configuration to manage HA of the stateful VLC application service and the second configuration to manage HA of the stateless Apache HTTP Server application service. In addition, we designed two HA configurations for the OpenSAF to manage HA of the same two applications. All the configurations include a cluster of two containers, each container to be hosted on a separate physical host. We designed 2N redundancy model for both Pacemaker, and OpenSAF. One container hosts the active component that provides the service, and the second container hosts the standby component to protect the service. Once the configurations are generated by the Configuration Generator unit, they are passed to the Deployment Manager module which automatically deploys the resources and then feeds the configuration to the corresponding HA middleware. The middleware starts managing HA of the service during its lifetime. Figure 3.6 depicts the private cloud Container-based platform after the deployment.

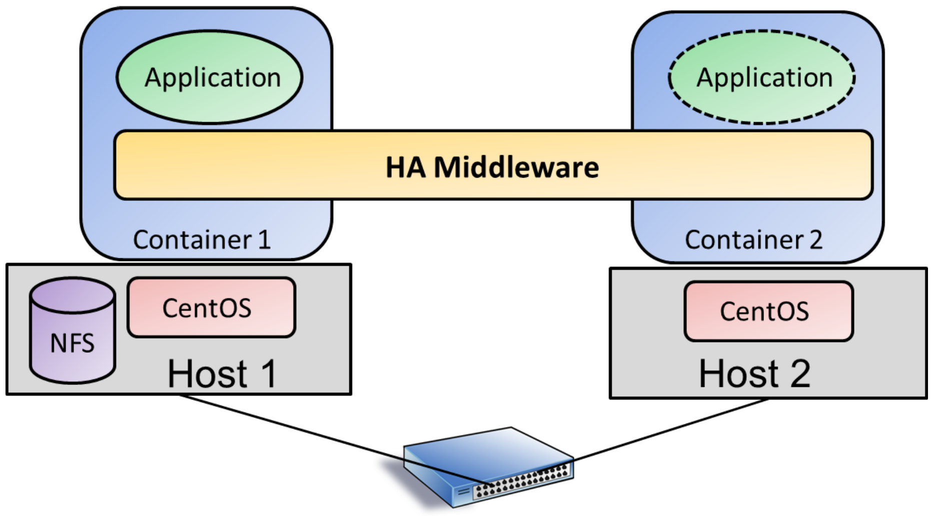


Figure 3.6: Private Cloud Container-based Platform

To get the availability measurements using Container-based platform, we run several experiments by injecting a failure for the active component and see the failover scenario of the service by the HA middleware, then collect the measurements. For the experiments with the Pacemaker, we simulate the component failure by stopping the Pacemaker itself on the container that hosts the active component. We register the time when the Pacemaker is completely stopped as the failure time. Note that the Pacemaker does not have a mechanism to detect the liveness (existence) of the process of the application component, therefore we simulate the application failure by stopping the Pacemaker. In addition, Pacemaker registers a notice with a timestamp at the system log indicating scheduling the Pacemaker for the shutdown. We consider this notice as the first reaction towards the recovery of the service. Therefore, the time period between the failure time and the notice time of scheduling the Pacemaker for the shutdown is considered as the reaction time. We also register the time when the Pacemaker failover the application service to the standby component. We consider the time between the reaction time and the service failover time as the service recovery time, and the entire time period since the failure to the time when the service is recovered as the service outage time.

For the experiments with the OpenSAF, we simulate the component failure by stopping (terminating) the process of the active component. Therefore, we consider the time when we stop the active process as the failure time. When the availability management service of OpenSAF receives a notification about the failure of the active component through the passive monitoring service of OpenSAF, the availability management service directly terminates the component in a graceful manner to guarantee the component is completely stopped and does not provide the service. We consider the termination action as the first reaction by OpenSAF towards the service recovery. Therefore, we consider the time period between the failure and terminating the component as the reaction time. As a result of the failure, the management service failover the application service to the standby component to continue providing the service. Service recovery time and service outage time are measured in the same way as they are measured for the Pacemaker experiments.

3.3.4 Experiments with Cloud VM-based Platform

For the experiments using the cloud VM-based platform, we designed new configurations. We replaced the containers with VMs, and the OS of the hosts by Xen hypervisor. Figure 3.7 depicts the private cloud VM-based platform after the deployment. We repeated the same type of experiments using the VM-based platform as we have done for the experiments using the Container-based platform, and collected the measurements.

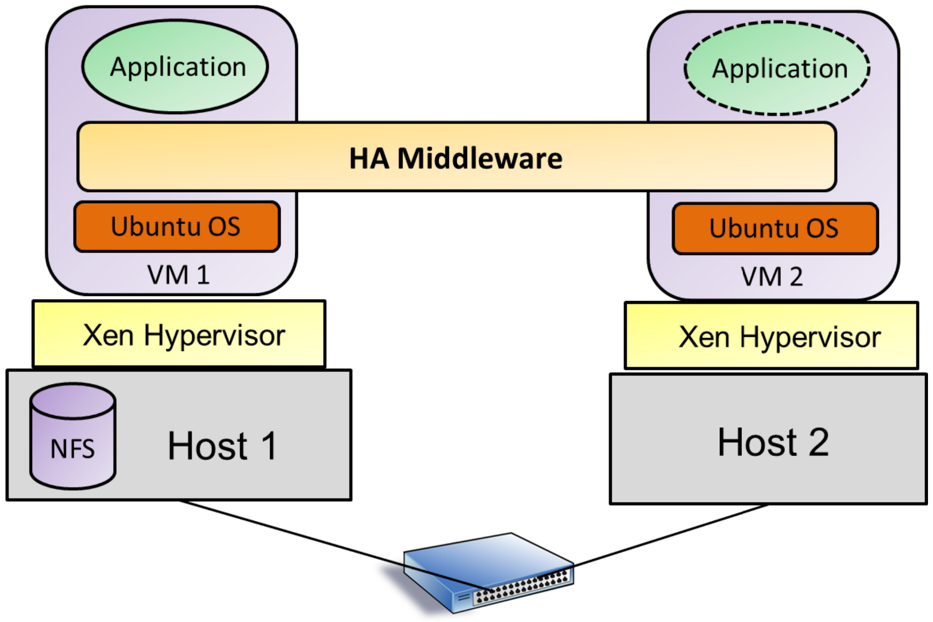


Figure 3.7: Private Cloud VM-based Platform

3.3.5 Results Analysis

We run several experiments using the Container-based platform (Figure 3.6), and VM-based platform (Figure 3.7) to get the availability measurements for the VLC and Apache HTTP services using Pacemaker and OpenSAF. Table 3.1 shows the availability measurements for the stateful VLC application service of the experiments using both platforms, while Table 3.2 shows the measurements for the stateless Apache HTTP Server application service of the experiments using the same platforms. Note that all the values in the tables represent the average values of 10 consecutive experiments that are measured in second time unit. As shown in the tables, the service recovery of all the applications using the containers is faster than using the VMs. Therefore, the service outage is shorter using the containers than using the VMs. For example, it takes an average of 0.181 second with the Pacemaker and 0.133 second with the OpenSAF to recover the stateful VLC service using the containers, while it takes 0.23 second with the Pacemaker and 0.2 second with the OpenSAF to recover the same VLC service using the VMs. Similar observation can be made for the stateless Apache HTTP Server application. The service recovers faster using the containers than using the VMs, 0.13 vs 0.18 second with the Pacemaker, and 0.11 vs 0.16 second for the OpenSAF. We believe the service recovery using the containers takes less time than using the VMs because the communication between containers does not go

Table 3.1: Availability Measurements for Stateful VLC Application

Unit:second	**Reaction Time**		**Service Recovery Time**		**Service Outage Time**	
	Container-based Platform	**VM-based Platform**	**Container-based Platform**	**VM-based Platform**	**Container-based Platform**	**VM-based Platform**
Pacemaker	0.128	0.25	0.181	0.23	0.309	0.48
OpenSAF	0.01	0.04	0.133	0.2	0.143	0.24

Table 3.2: Availability Measurements for Stateless Apache HTTP Server Application

Unit:second	**Reaction Time**		**Service Recovery Time**		**Service Outage Time**	
Pacemaker	**Container-based Platform**	**VM-based Platform**	**Container-based Platform**	**VM-based Platform**	**Container-based Platform**	**VM-based Platform**
	0.157	0.24	0.13	0.18	0.287	0.42
OpenSAF	0.01	0.03	0.11	0.16	0.12	0.19

through the extra hypervisor layer as the case for the communication between the VMs which impacts the recovery time. In addition, the container is considered more lightweight than the VM, and hence the number of CPU operations induced by the container is less than the number of CPU operations of the VM, which makes the container faster than the VM for detecting the component failure and recovering the service.

Moreover, the measurements show that the service recovery for the Apache HTTP Server application is faster than the service recovery for the VLC application in all the experiments using both platforms. In addition, the service outage is shorter for the Apache than for the VLC. For example, VLC service outage takes 0.309 second while it takes 0.287 second for the Apache service with the Pacemaker using Containers-based platform. With the OpenSAF using the same platform, the VLC service outage takes 0.143 second and the Apache service takes 0.12 second. We believe this is because the VLC is a stateful application and the HA middleware has to retrieve the last state of the active VLC component before its failure from the NFS storage to continue providing the service, which takes extra time. On the other hand, the middleware does not need to retrieve the state of the active Apache HTTP Server component because it is a stateless application. Also, the measurements show that the reaction of the OpenSAF for the component failure of both applications VLC and Apache using both platforms is faster than the reaction of the Pacemaker for the same types of failures. We believe this is because the Pacemaker does not have a mechanism to detect the application component failure, so we stop the Pacemaker to simulate the application failure. Therefore, the Pacemaker has to wait until its service completely stops and then reacts to the failure. While the OpenSAF has a mechanism to detect the application component failure at the process level, so there is no need to stop the OpenSAF to simulate the application failure, hence the OpenSAF can detect the component failure faster.

3.4 Summary

We proposed a reactive framework to manage availability, and enable service continuity for application service in a virtual cluster. The framework integrated two HA open-source middlewares OpenSAF and Pacemaker to maintain service availability. The framework is evaluated using our private cloud platform to manage the availability of VLC stateful application, and Apache HTTP Server stateless application. The results show the ability of the framework to recover Apache service faster than VLC service within less than a second time. In addition, the framework shows the ability to resume VLC streaming service from its last state before the VLC process fails.

Chapter 4

Proactive Service Availability Management

Usually, reactive approach uses redundant resources to recover the service and maintain its availability. Redundant resources add extra operational, maintenance, power consumption, and communication costs for the CSP. This motivated us to propose a proactive framework to predict the termination status of the application task (service) during the runtime, and take the appropriate protective action for any task that is predicted as failed. The framework includes four main modules that work together for a proactive approach to manage the availability of the application task that is executed in the cloud cluster. We analyze three publicly available large cluster datasets from Google, Alibaba, and Trinity, to characterize task failure in the cloud computing platform. The framework uses deep learning models named Artificial (ANN), and Convolutional Neural Network (CNN) of type LeNet-5 for different prediction purposes. The rest of this chapter is organized as follows. Section 4.1 describes the three datasets that are used in the prediction process. Section 4.2 discusses deep learning Neural Network. Section 4.3 introduces the proposed proactive framework including the modules. Section 4.4 discusses the experiments and results, and Section 4.5 gives a summary of the chapter.

4.1 Computing Cluster Datasets

4.1.1 Google Cluster Dataset

In 2011 Google released a huge dataset that includes valuable information about cloud computing cluster for a time period of 29 continuous days [6]. The dataset includes information of approximately 12,500 computing nodes. It also includes information about the states of jobs during their lifetime that have been requested by the users. Each job is a composition set of one or more tasks. Task represents a program unit of a special type that is executed on only one node. Task information includes the events and usages. Task events include event timestamp, event type, job id for which the task belongs, and demand resources such as CPU, RAM, and Disk. In addition, events include configuration information such as priority and scheduling class level of the task.

The task can go through several events from its submission until termination, as shown in Figure 4.1. When the task is initially submitted, it will be in the pending state until it gets scheduled on a computing node and is moved to a running state. If the task is terminated it would be in the dead state. However, a task is not always terminated successfully where it can fail for different reasons, such as software bug or exception. It also can be killed by the user or the system administrator or evicted by other tasks that belong to other jobs and have higher priority. The dataset shows a few tasks got lost for unknown reasons, so we do not consider lost tasks in this study. Tasks usage includes information about the resources usage that have been consumed by each task from the underlying computing node such as CPU, RAM, Disk I/O, and Disk space. Table 4.1 summarizes the statistics of application tasks based on the termination status (finished, failed, killed, and evicted) from the Google cluster. As shown in Table 4.1, a large number of tasks were terminated unsuccessfully (failed, killed, and evicted).

Table 4.1: Statistics of Tasks per Termination Status for Google dataset

Termination Status	**Tasks Count**	**Ratio (%)**
Finished	18217975	38
Failed	13829769	29
Evicted	5864353	12
Killed	10349680	21
Total	**48261777**	**100**

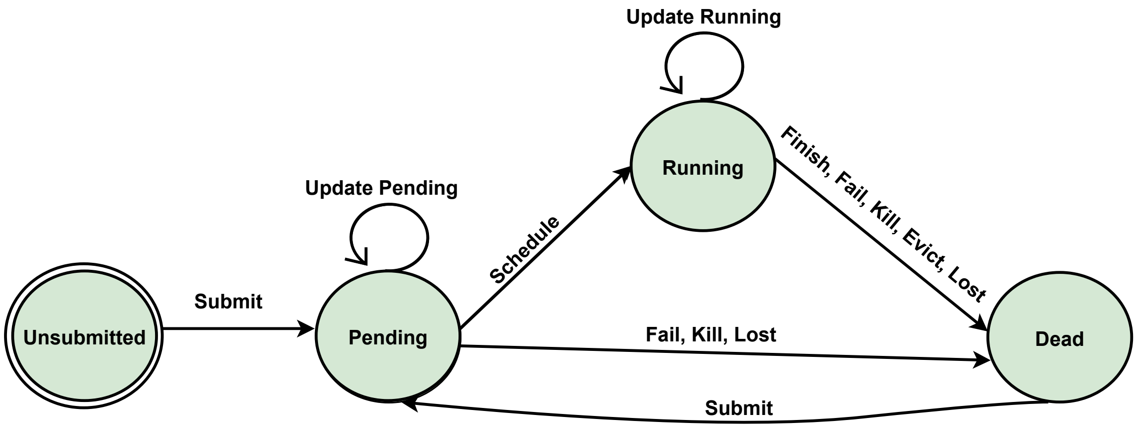


Figure 4.1: Task Events State Diagram

4.1.2 Alibaba Cluster Dataset

In 2017, Alibaba group released a dataset labeled 'cluster-trace-v2017' of a production computing cluster for a continuous 12 hours time period [21]. The dataset includes information about application jobs, tasks and their instances that have been executed on the cluster that contains 1,300 computing nodes. Job is composed of a set of tasks, and each task can have a set of instances of the same type that execute on the same or different nodes. Task information includes the demanded resources such as CPU and RAM, as well as the number of instances of each task. Task instance information includes the event types, scheduling node, and resources usage of the instance during its execution time. The instance goes through different events that change its state from its submission until termination. We focus on the termination status of the instance among the set of event types. Table 4.2 summarizes the statistics of tasks instances based on their termination status that can be Terminated, Failed, or Cancelled from Alibaba cluster. Terminated means instance was finished successfully. Failed means instance was failed and hence terminated unsuccessfully due to software or hardware error in the cluster. Cancelled means the instance was interrupted by administrative operation and did not start or finish its execution. We consider the cancelled instance as failed. As shown in Table 4.2, the ratio of the failed instances is 2.3%, which is very small compared to the high ratio of 97.6% for the terminated instances. However, the total number of the failed instances can have an impact on the overall performance and resource usage of the cluster, and are worth to be detected as early as possible to take actions.

Table 4.2: Statistics of Instances per Termination Status for Alibaba dataset

Termination Status	**Tasks Count**	**Ratio (%)**
Terminated	729982	97.6
Failed	17372	2.3
Cancelled	293	0.01
Total	**747647**	**100**

4.1.3 Trinity Cluster Dataset

Trinity is the largest supercomputer at the Los Alamos National Lab (LANL) and it is used for capability computing [22]. In 2018, LANL released a dataset for the jobs that have been executed on the computing nodes of the Trinity cluster for three months period from February to April 2016. At the time of the trace logs, Trinity included 9,408 identical nodes, a total of 301056 Intel Xeon E5-2698v3 2.3GHz cores and 1.2PB RAM, making it the largest cluster with a publicly available trace by the number of CPU cores [22]. The dataset includes information of 23,359 jobs of OpenScience workloads that were issued by 88 users. Job information includes the event types that the job goes through during its lifetime, and demanded resources such as the number of nodes and CPU cores to execute the job. Note that LANL does not reveal information about the tasks that belong to the jobs. Therefore, in this research work, we map the job as a single unit like a task. We focus on the job termination status that can be Ended, Failed, or Cancelled. Ended means the job was terminated successfully. Table 4.3 summarizes the statistics of jobs based on their termination status from the Trinity cluster. Note that the ratio of both Failed and Cancelled jobs is 27.6% of the total submitted jobs, which is considered high and can lead to performance degradation, and waste a large amount of resources of the Trinity cluster.

Table 4.3: Statistics of Jobs per Termination Status for Trinity dataset

Termination Status	**Jobs Count**	**Ratio (%)**
Ended	16923	72.4
Failed	5185	22.2
Cancelled	1251	5.4
Total	**23359**	**100**

4.2 Deep Learning - Neural Network

In the literature, some research works used statistical and machine learning methods such as Support Vector Machine (SVM) and Hidden Markov Model (HMM) on Google dataset to predict job/task termination status. Such methods assume data segments are independent of each other that makes them a poor choice for Google dense dataset where many of the data segments are dependent and correlated with each other. Therefore, we use deep learning (DL) models such as ANN and CNN (LeNet-5) to discover the hidden patterns that are related to the task termination status from the datasets, combine the patterns together, and build efficient decision rules. In addition, DL helps to identify the relationships between the extracted features of the task, as well as the relationships between tasks, and these help to increase the accuracy of the prediction process. In this thesis, we use DL approach to predict task termination status, whether a success or fail. Based on that a decision is taken if the task is to be protected which can decrease its failure probability.

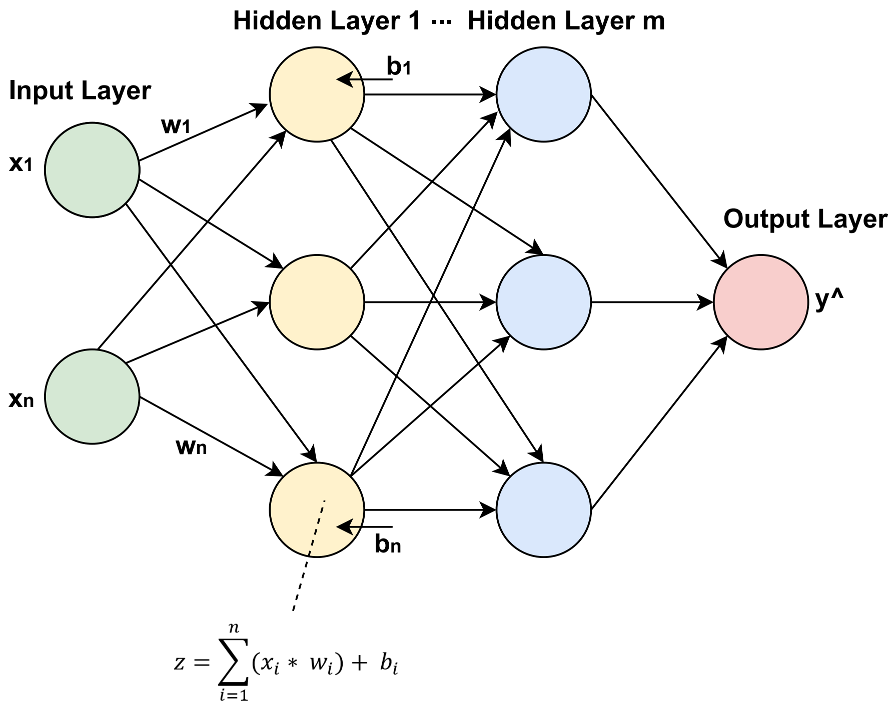

Hidden Layer 1 ··· Hidden Layer m
Input Layer
Output Layer
x1
xn
w1
wn
b1
bn
y^
$z = \sum_{i=1}^{n}(x_i * w_i) + b_i$

Figure 4.2: Structure of Deep Learning Neural Network

Figure 4.2 illustrates the general structure of deep learning neural network. It is a fully connected multilayer of neurons that includes one input layer, one or more hidden layers, and one output layer. The network gets deeper by the increasing number of intermediate hidden layers. Each layer is composed of a set of neurons (nodes), where each neuron is connected with all the other neurons of the next layer through synapses (connections). Each neuron takes a set of inputs, the corresponding weights of the input, and biases, then passes the result of $(\sum((inputs * weights) + biases))$ to activation function such as Sigmoid or Tanh, then passes the output of the function to all the neurons of the next layer until the final output layer is reached. Training of neural network means learning the weights that are associated with synapses for which the prediction error is minimum compared to the original (true) state of the input. Using feed-forward and backward iterations, a deep neural network updates the weights of synapses in such a way that minimizes the error ratio of predicting the output of a given input compared with its real value. It uses special optimizers to optimize the process of finding the optimal weights that minimize the error loss during the training process.

4.3 Proactive Service Availability Framework

We propose a proactive service availability framework to maintain the availability of application task during its execution until successful termination. The main goal of the framework is to predict the termination status of application tasks, either finished or failed, during the execution time as early as possible and to take the right remedy actions for these tasks that are predicted as failed. The framework also minimizes the failure ratio of application tasks and their resources usage. Figure 4.3 depicts the general proactive framework that includes four main modules: Data Cleaning and Preparation, Feature Extraction, Task Failure Prediction, and Failure-Aware Task Scheduler. The modules are discussed in detail in the subsequent sections. Note that the proposed framework is general and independent of the cloud platform. It can be used with any given dataset that includes related information to predict termination status and take remedy actions for both jobs and tasks. However, the framework requires configuration parameters for each distinct cluster platform.

4.3.1 Data Cleaning and Preparation

To know the main task failure patterns and characteristics in computing clusters, we studied and analyzed a large number of CSV trace logs that are collected from three different platforms named Google, Alibaba, and LANL Trinity. Logs contain information about task

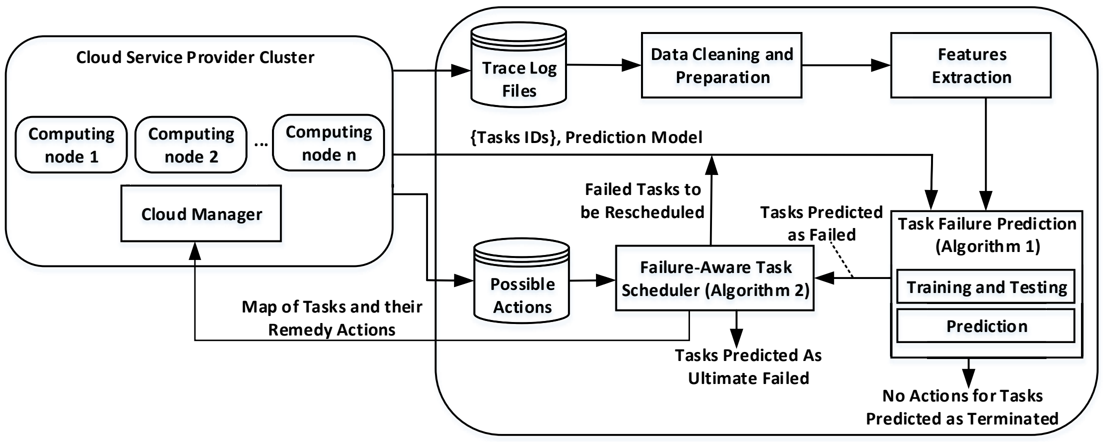

Cloud Service Provider Cluster
Computing node 1
Computing node 2
...
Computing node n
Cloud Manager
Trace Log Files
Data Cleaning and Preparation
Features Extraction
{Tasks IDs}, Prediction Model
Failed Tasks to be Rescheduled
Tasks Predicted as Failed
Task Failure Prediction (Algorithm 1)
Training and Testing
Prediction
Possible Actions
Failure-Aware Task Scheduler (Algorithm 2)
Map of Tasks and their Remedy Actions
Tasks Predicted As Ultimate Failed
No Actions for Tasks Predicted as Terminated

Figure 4.3: Proactive Service Availability Framework

events, usages, and scheduling nodes that are located in separate files. Each file contains a tabular set of records and their columns (attributes). Each record represents information for one task sample. In general, trace files contain many noisy and missing data (Null). Here we discuss some of the data cleaning and preparation that we have done for the input trace logs. For example, some numeric attributes contain characters instead of numbers. We neglect them as noisy data. Any missing data is filled with the corresponding default values. For example, we replace Null values of task usage with zero. Other modules of the framework require events and usage information together for tasks. Therefore, we wrote a script file to merge the files and get the required information for each task. Usages of tasks are logged periodically based on the platform. For example, in Google platform, task usage is logged every five minutes. Therefore, we wrote another script file to aggregate the total usage and compute the mean usage for each task. Tasks can have the same identification (ID) whether they belong to the same or different users. To uniquely identify tasks, we combined the ID of the task with the ID of the job for which the task belongs. The same task can run on multiple nodes at different times, so we combined together task ID, job ID, and ID of the node where the task was executed to distinguish the task usage per node.

4.3.2 Feature Extraction

Through analyzing large datasets, we find the correlations between the task's attributes (features) and its failure in each dataset separately. In Google trace logs, we found that the unsuccessful tasks take a longer execution time than the successfully terminated tasks.

We computed the cumulative distribution function (CDF) for the tasks extracted from the trace logs that were executed during the first day. Figure 4.4 shows the CDFs for the

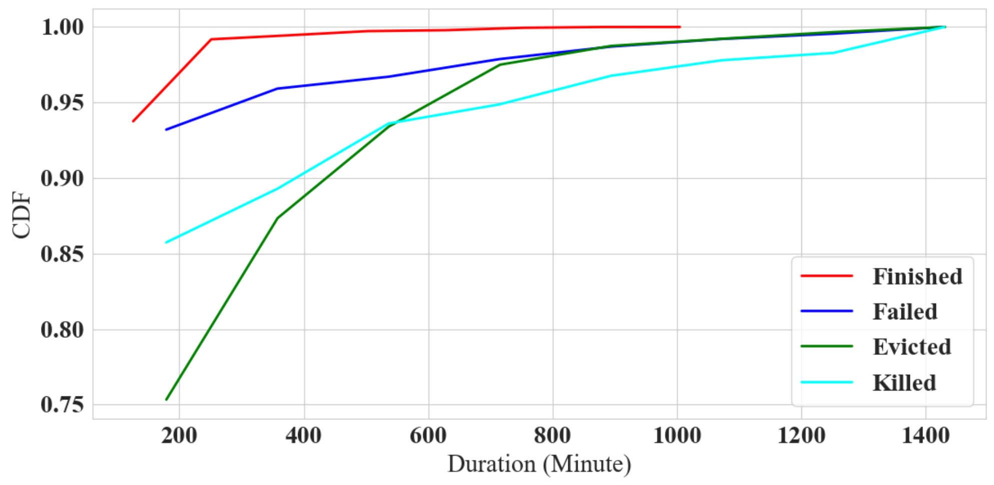


Figure 4.4: CDFs for Tasks Execution Time per Termination Status

execution time of tasks for each termination status. An unsuccessful task has a higher chance to be resubmitted more than one time until it is successful. Some tasks are of type debugging and testing, and take longer execution time before the user or the system administrator kills them. Usually, failure tasks get stuck or frozen for a while before they terminate unsuccessfully. In some situations, the task with a lower priority can run for a long time before getting evicted by another task that has a higher priority. Figure 4.5 shows the CDFs for tasks resubmission for each termination status during the first day. We can see the frequency of unsuccessful tasks resubmission is higher than the frequency of finished tasks resubmission. We found 60% of failed tasks, 20% of evicted tasks, and 11% of killed tasks are resubmitted, while only 5% of finished tasks are resubmitted during the entire trace period (29 days). Usually, the user keeps trying to resubmit unsuccessful task until it terminates successfully. In some cases, the user tries to resubmit the finished task for validation purposes. As we can see in Figure 4.5, some of the killed tasks have been resubmitted more than 250 times.

We studied the effect of both task priority and scheduling class level on task failure. Figure 4.6 shows ratios of tasks termination status per priority value for all the tasks executed during 29 days. We notice both tasks with lower and higher priority have a higher ratio of failure. We believe this is because tasks with lower priority are prone to get evicted

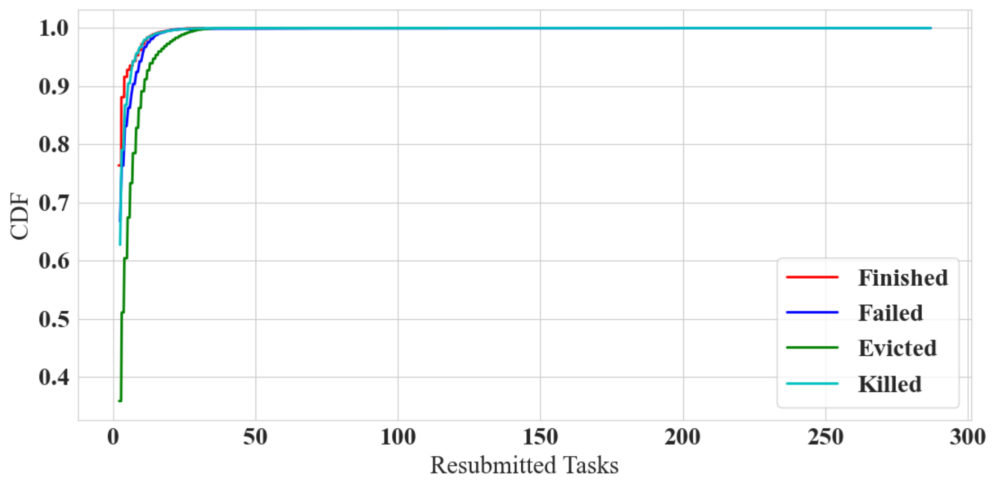


Figure 4.5: CDFs for Tasks Resubmission Count per Termination Status

by tasks with higher priority. On the other hand, usually tasks with higher priority demand more resources than the tasks with lower priority. Shortage of fulfilling the resource demands of any task can lead to its failure. Figure 4.7 shows ratios of tasks termination status per scheduling class level during the entire trace period. We notice tasks with lower class level have a lower chance to fail. We do not have a clear clue for this because the dataset does not reveal detailed information regarding the scheduling class level. All that we know, the class level is used during the task scheduling process to satisfy certain constraints. To investigate the relationship between resource usage and task termination status, we computed usage of CPU, RAM, and Disk I/O consumed by tasks for each termination status during the first day. Figures 4.8, 4.9, and 4.10 show the usage of CPU, RAM, and Disk IO respectively for tasks per termination status. We can see unsuccessful tasks consume a large amount of resources. Google dataset suffers from main limitations, that some of the features are with empty values for many of the tasks. In addition, the complete information of tasks is located in different schemas that need to be merged together. Through analyzing large Google traces, we found other features that can play a role in the termination status of the task. These features can be used in training and testing datasets for the prediction module to predict task status. We categorized features into static and dynamic. Static features can be extracted directly from the tasks events traces. They mainly represent resource demand and the configuration of tasks. We extracted the following static features for tasks from Google trace logs: *<Task ID, Job ID, Machine ID, Scheduling Class Level, Priority,*

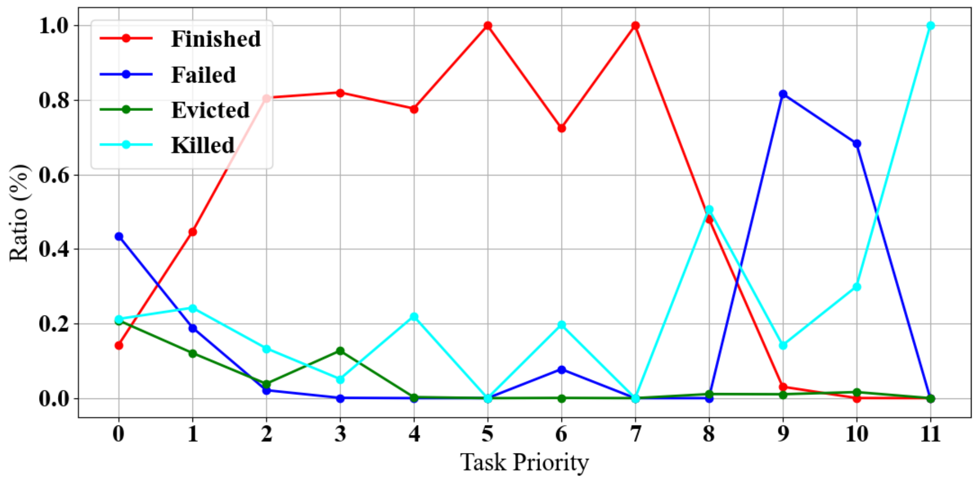


Figure 4.6: Ratios of Tasks Termination Status per Priority

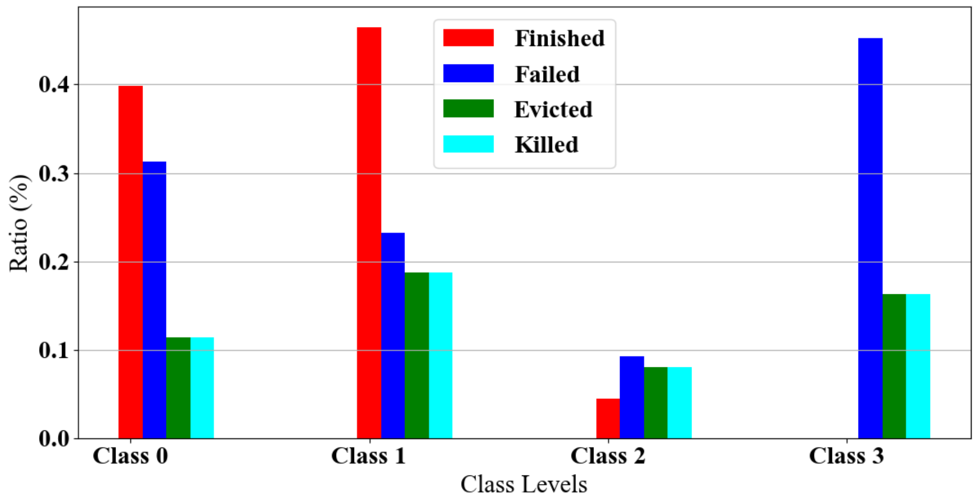


Figure 4.7: Ratios of Tasks Termination Status per Class Level

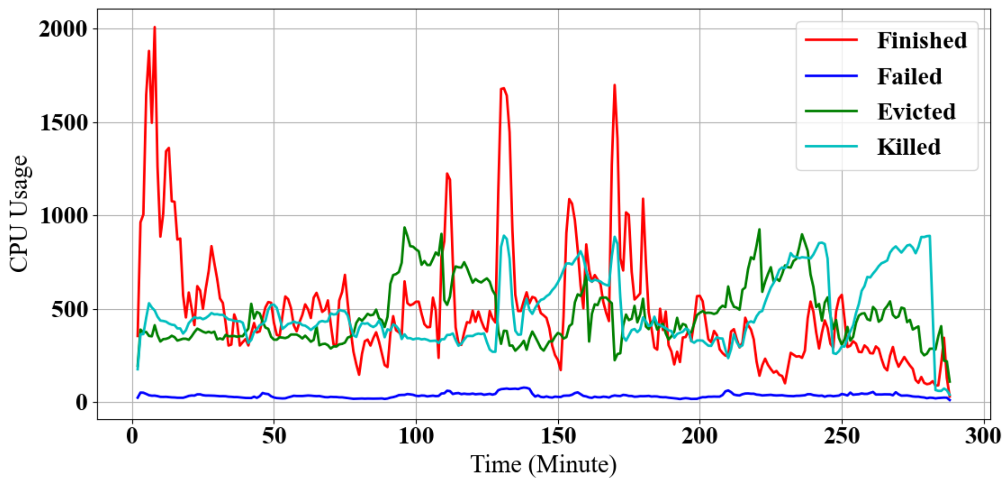


Figure 4.8: CPU Usage of Tasks per Termination Status

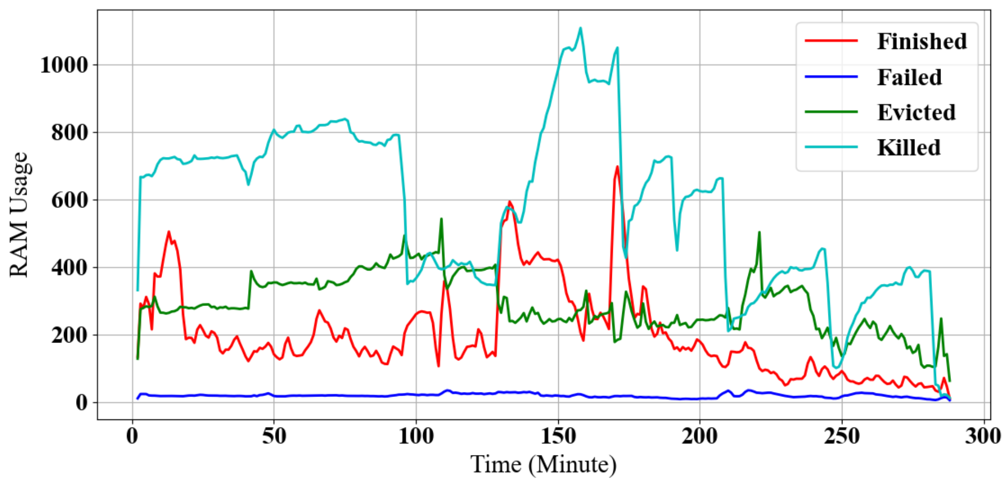


Figure 4.9: RAM Usage of Tasks per Termination Status

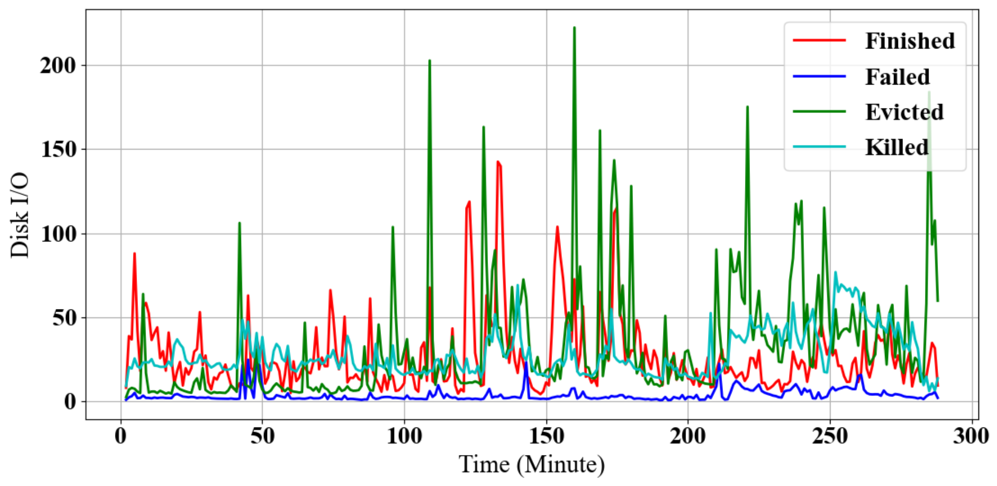


Figure 4.10: Disk Usage of Tasks per Termination Status

CPU Demand, RAM Demand, Disk Space Demand, Task Termination Status >. Dynamic features require special computations to be extracted because they change by changing the scheduling of the task, which can be extracted from tasks usage traces. We extracted the following dynamic features for tasks: <*Mean Execution Time, Mean CPU Usage, Mean RAM Usage, Mean Disk Space Usage, Mean I/O Usage* >. We merged both the static and dynamic features for each task to build the training and testing dataset for the prediction module.

In Alibaba trace logs, we noticed a correlation between task instance failure and the computing machine (node) that hosts the instance. For some machines, the count of the failed instances of the same or different tasks is high. On the other hand, some other machines register zero cases for instance failure. As shown in Figure 4.11, the count of the failed instances reaches up to 64 for some individual machines. We computed the failure occurrences of instances per each task ID. As shown in Figure 4.12, for some tasks, the number of the failed instances that belong to the same task type reaches up to 4,000. Some of the instances failed on the same or different computing machines. Although the ratio 2.3% of the failed instances is very small, it is worth mentioning that the failed instances consumed a total of 9,684 CPU units during 12 hours time period. The logs miss the RAM usage for many of the failed instances. We extracted the following static features for all task instances: <*Task ID, Job ID, Machine ID, Number of Instances, Requested CPU, Requested RAM, Instance Termination Status* >. We also extracted the following dynamic features: <*Mean CPU Usage, Total Sequence Number*>. Alibaba dataset includes a small number of tasks that have failed compared to the ones that have succeeded. We consider this as a limitation for us having small number of the failed tasks in the training dataset. In addition, the dataset misses the RAM and Disk usages information of tasks.

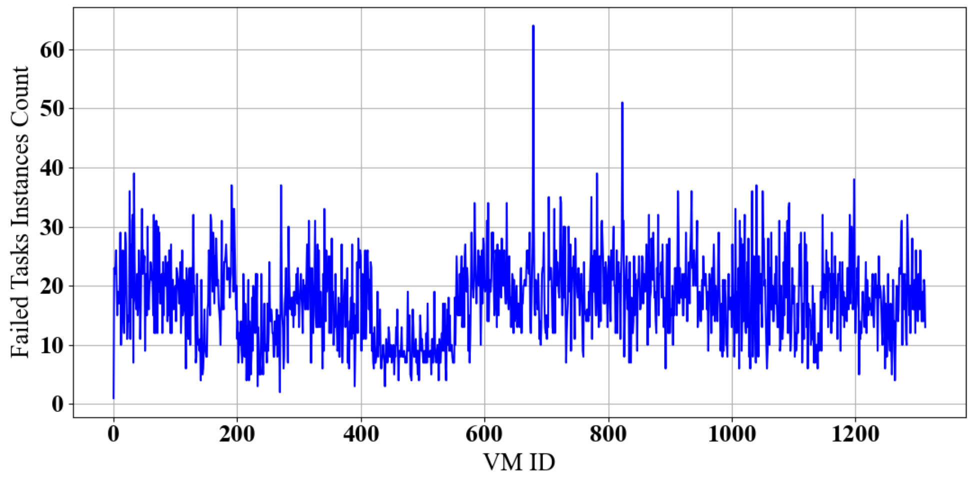


Figure 4.11: Count of Tasks Instances Failures per Virtual Machine ID

LANL Trinity trace logs do not include any information about tasks. However, it includes information about jobs that have been executed on the cluster. Therefore, we investigated the correlation between job features and their failure. We noticed that the failed jobs demand more computing nodes and resources to execute than the successfully ended jobs. As shown in Figure 4.13 the mean of the demanded nodes by the failed jobs is equal to 345, and 250 by the successfully terminated jobs. In addition, failed jobs have more tasks than successful jobs. As shown in Figure 4.14, the mean of the tasks counts of the failed and successfully terminated jobs is equal to 4,366 and 2,630 respectively. We believe this is because the requested resources that are not satisfied during the execution time will lead the job to its failure. We extracted the following static features from the Trinity trace logs: <*User ID, Group ID, Submit Time, Start Time, Queue Time, Requested Nodes, Tasks Number, Job Termination Status*>. Trinity dataset suffers from main limitations, that it only includes information about jobs and misses the information for their tasks. In addition, it does not include the information for the RAM and Disk usages of the jobs. Note that we defined the features for each dataset in the features extraction module. For any new dataset from a new platform, the features are required to be defined as well.

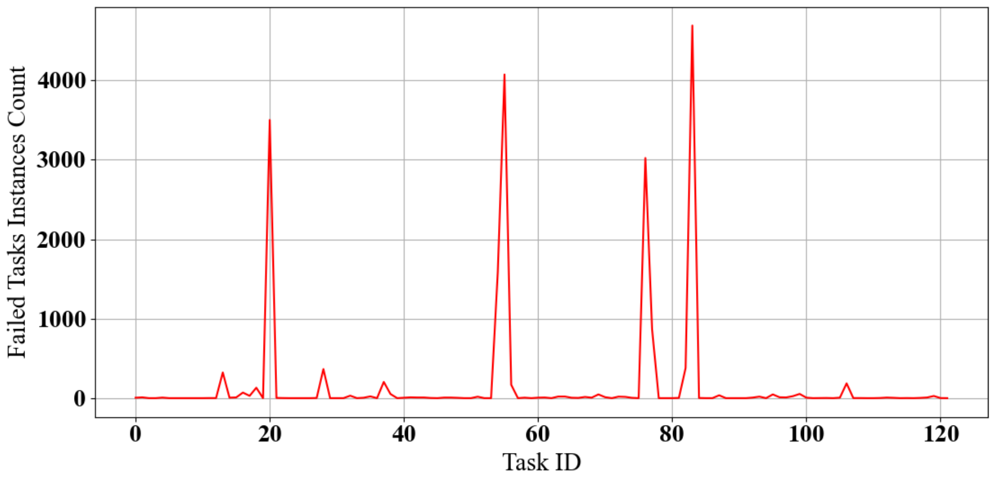


Figure 4.12: Count of Instances Failures per Task ID

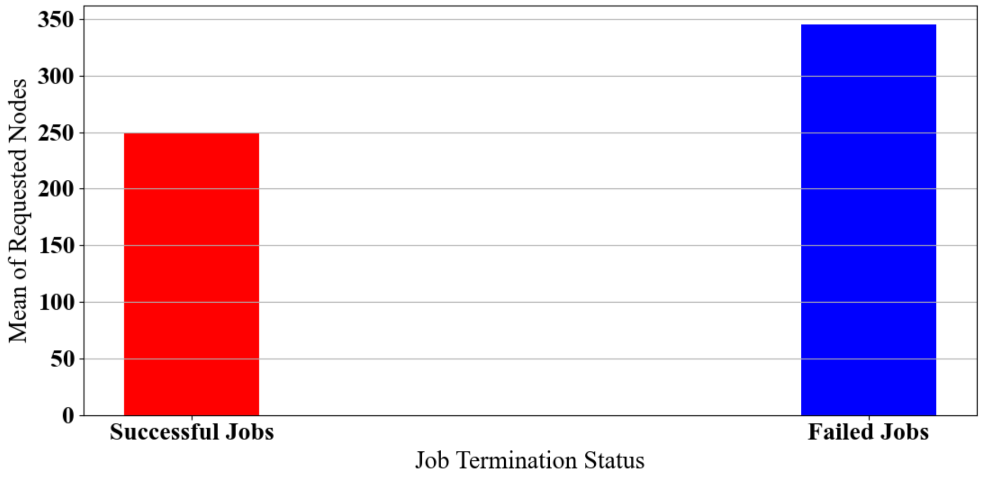


Figure 4.13: Mean of Requested Nodes per Job Termination Status

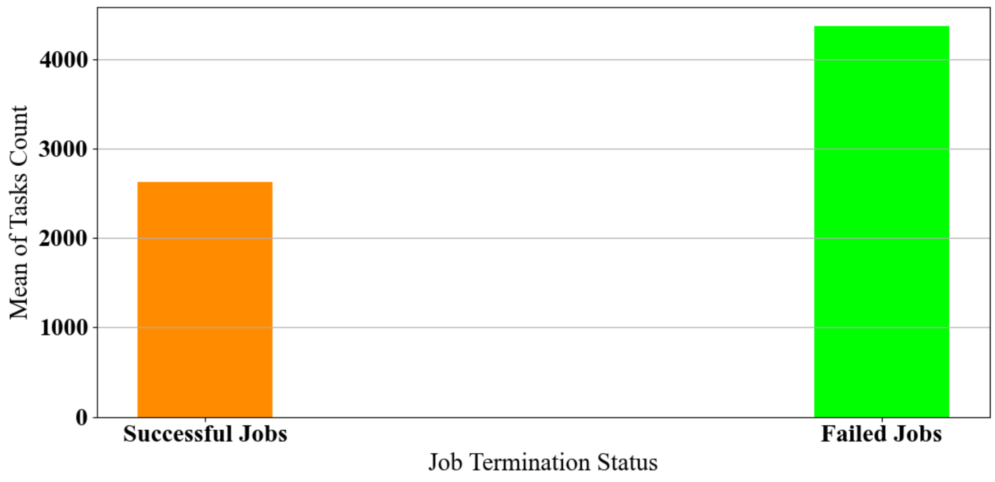


Figure 4.14: Mean of Tasks Count per Job Termination Status

4.3.3 Task Failure Prediction

Task failure prediction is an important module of the framework that uses one of the deep machine learning models, ANN, or CNN, to predict termination status for a given set of tasks during their execution time. Algorithm 1 describes the functionality of the task failure prediction module. The algorithm takes a set of tasks T, dataset D, prediction model M, and parameter O as input. It returns the map *predFailedTasksMap* as an output that includes the tasks for which the prediction status is predicted as failed and the failure probability of each task. Set T represents the tasks for which the termination status is required to be predicted. Set D represents the dataset that is extracted from input trace logs, cleaned by the cleaning and preparation module, filtered, and compliant with the feature extraction module. D is used to train and test the selected prediction model, and to extract the tasks information. The parameter M determines the required prediction model to be used during the prediction process. The parameter O is an administrative operation to determine the flow of the module, either for training and testing the model or to use the model for prediction purposes. The algorithm begins by initializing the map *predFailedTasksMap* with a *Null* value (line 1). It then selects the prediction model as either ANN or CNN based on the parameter M (line 2). If the parameter O is equal to ‘Training’ the selected model requires training and testing with list T as empty, otherwise, the model is used for the prediction. This is to avoid training and testing the model every time the prediction process is called to reduce the computing cost. Note that it is required to train and test the model at least one time before starting the first prediction request. The parameter O can be used any time, based on the request, to train and test the model using a new input dataset D. If the selected prediction model requires training (line 3), the algorithm assigns 80% of D as training dataset (line 4) and 20% as a testing dataset (line 5). The training and testing processes run in lines 6 and 7 respectively. In this case, the algorithm will return an empty $predFailedTasksMap$ and terminate (line 23) because the prediction process is not called. If O equals to ‘Prediction’, the prediction process for a given set of tasks T is required (line 9). For each task $t \in T$ (line 10), the algorithm extracts the complete information of t (line 11). The algorithm predicts the termination status (line 12) and the failure probability (line 13) of t using the selected prediction model. In case the termination status of t is predicted as ‘Failed’ (line 14), the algorithm adds a tuple that includes t and its $failureProbability$ to the map $predFailedTasksMap$ (line 15). Otherwise, if the termination status of t is predicted as ‘Terminated’ (line 17), it adds t to the list of tasks that are predicted as terminated $predTerminatedTasksList$ (line 18). Once the termination status is predicted for all tasks $\in T$, the algorithm returns the map $predFailedTasksMap$ and terminates (line 23).

For computational complexity of Algorithm 1, the prediction module uses the approach described in Algorithm 1 to predict the termination status for a given set T of tasks. In some cases, the algorithm requires to train and test the selected model before the prediction process starts. In the case of training and testing, the algorithm takes an input dataset of size n tasks, and processing costs a linear execution time $\mathscr{O}(n)$. In case the algorithm is used for prediction, it takes the set T of size n tasks. For each task $t \in T$, the prediction model predicts the termination status and failure probability of t, which costs $\mathscr{O}(2)$ execution time. If the termination status is predicted as ‘Failed’, t and its failure probability are added to the map of tasks that are predicted as failed which costs $\mathscr{O}(2)$ execution time. If the termination status is predicted as ‘Terminated’, t is added to the list of terminated tasks which costs $\mathscr{O}(1)$ execution time. In the worst-case scenario, each t costs $\mathscr{O}(4)$ execution time. So the entire cost to process all the tasks $\in T$ is $\mathscr{O}(4n)$ that can be simplified as a linear cost $\mathscr{O}(n)$ execution time.

Algorithm 1 Task Failure Prediction

```
Input:  Tasks Set $T$, Dataset $D$, Model $M$, Operation $O$
Output:  Map $predFailedTasksMap$
 1: $predFailedTasksMap = Null$
 2: $predModel \leftarrow (ANN, CNN)$ based on $M$
 3: if ($O ==$ '$Training$') then
 4:     $trainSet \leftarrow$ 80% of $D$
 5:     $testSet \leftarrow$ 20% of $D$
 6:     $predModel.startTraning(trainSet)$
 7:     $predModel.startTesting(testSet)$
 8: end if
 9: if ($O ==$ '$Prediction$') then
10:     for each $t \in T$ do
11:         $t \leftarrow getTaskInformation(D, t.taskID)$
12:         $terminationStatus \leftarrow predModel.predictTerminationStatus(t)$
13:         $failureProbability \leftarrow predModel.getFailureProbability(t)$
14:         if ($terminationStatus ==$ '$Failed$') then
15:             Add $< t, failureProbability >$ to $predFailedTasksMap$
16:         else
17:             if ($terminationStatus ==$ '$Terminated$') then
18:                 Add $t$ to $predTerminatedTasksList$
19:             end if
20:         end if
21:     end for
22: end if
23: return $predFailedTasksMap$
```

4.3.4 Application Task Scheduling

The failure-aware scheduler is considered the core module of the proposed framework. The main purpose of the module is to take the appropriate remedy actions for tasks that are predicted as failed by the prediction module and reschedule them immediately on computing nodes of the cluster to increase their chances to terminate successfully. We discuss the failure-aware scheduler module in detail in the next chapter 5.

4.4 Experiments and Results

4.4.1 Experimental Setup

We conducted several types of experiments using three different datasets collected from the computing clusters of Google [6], Alibaba [21], and LANL Trinity [22]. The purpose of the experiments is to evaluate our proposed framework including the modules and used methods in this research work. All the experiments are carried out using Python version 3 on 64-bit Windows 10 machine equipped with an Intel Core i7-8665U 2.11 GHz processor and 16 GB of RAM.

4.4.2 Evaluation of Prediction Models

The prediction module uses one of the deep learning models either ANN or CNN to predict the termination status of given tasks. Both ANN and CNN are implemented using sequential API from TensorFlow 2 platform. Each includes a neural network that is composed of one input, two hidden, and one output layer. For CNN model, all the layers are convolutional. The number of neurons for the input layer for both models varies based on the input dataset. For the Google dataset, the number of neurons in the input layer equals to the number of the input features of the task that is equal to 13. For Alibaba and Trinity datasets, the number of neurons in the input layer equals 8 and 7 respectively. The number of neurons in each hidden layer is equal to 100, and the number of neurons in the output layer is equal to 1 for all the input datasets and models. The models use Adam optimizer, Sigmoid activation function for classification, and Relu activation function for regression. In all the three datasets, we divided 80% of the input dataset for training and 20% for testing the prediction models (ANN, CNN). The models were trained for 100 epochs (iterations). The models were trained and then tested using different number of tasks based on the input dataset. Note that the used datasets for training and testing are not of time-series form. The total number of input tasks from Google equals 1,000,000. Using the Alibaba dataset, the total number of input tasks equals 35,330. Using the Trinity dataset, the total number of input jobs equals to 12,872. For the three input datasets, 50% of the total input tasks and jobs were terminated successfully and the other 50% were failed. Note that the total number of input tasks from Alibaba, and jobs from Trinity datasets are less as compared to Google because they include less number of the failed tasks and jobs. However, the models used all the failed tasks in Alibaba and Trinity.

To evaluate the prediction models, we measured the Accuracy and Error Loss of both models ANN and CNN using three input datasets. Accuracy is a commonly used measurement in machine learning that describes the percentage of the test samples that are predicted correctly. Error Loss is another commonly used measurement in machine learning that describes the distance (difference) between the true value of the input sample and its predicted value. Figure 4.15 shows the accuracy of the ANN model using the three datasets. ANN achieves a high accuracy of around 94% using the dataset of Google, 89% using Trinity, and 84% using Alibaba. Figure 4.16 shows the error loss of ANN. It achieves a low error loss of around 14% using Google dataset, 24% using Trinity, and 23% using Alibaba. We believe using the Google dataset, ANN achieves the highest accuracy and lowest error loss because the dataset includes a large input number of both terminated and failed tasks, where the model gets longer and deeper training to predict the task termination status. In addition, the Google input dataset includes more features (attributes) that help to increase the density of the data, hence increasing the similarity between tasks which leads to an increase in the accuracy.

The same thing is applied to CNN model. As shown in Figure 4.17, CNN achieves a high accuracy of around 92% using the dataset of Google, 88% using Trinity, and 83% using Alibaba. Also, CNN achieves a low error loss of around 18% using the dataset of Google, 28% using Trinity, and 25% using Alibaba as shown in Figure 4.18. We believe ANN achieves higher accuracy than CNN because ANN considers all the input features together during the training process, while CNN may not consider all the features together due to dimensional input transformation during the training process.

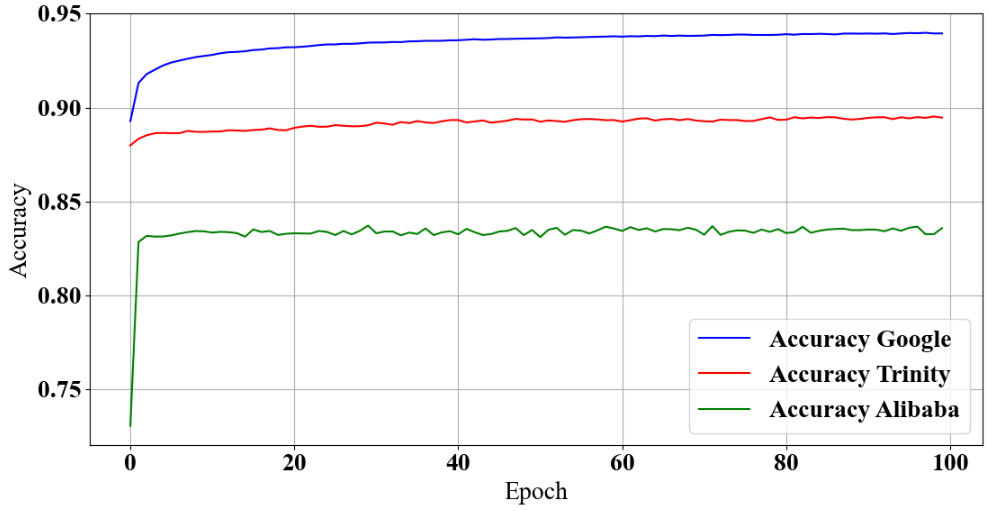


Figure 4.15: Accuracy of ANN using three datasets

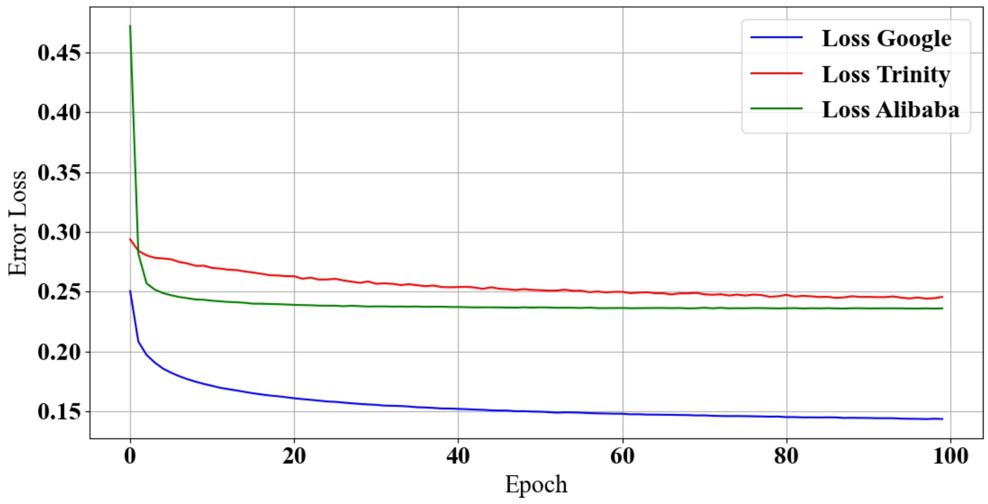


Figure 4.16: Error Loss of ANN using three datasets

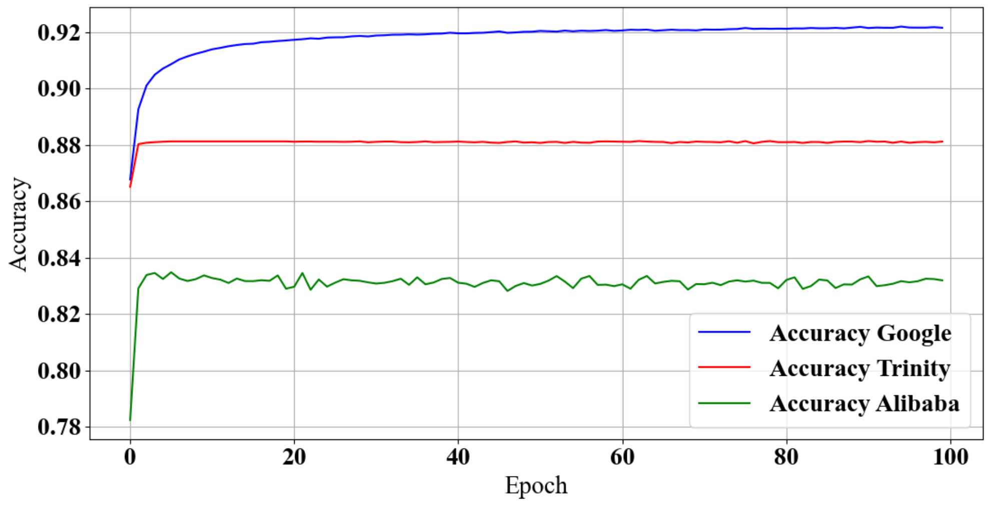


Figure 4.17: Accuracy of CNN using three datasets

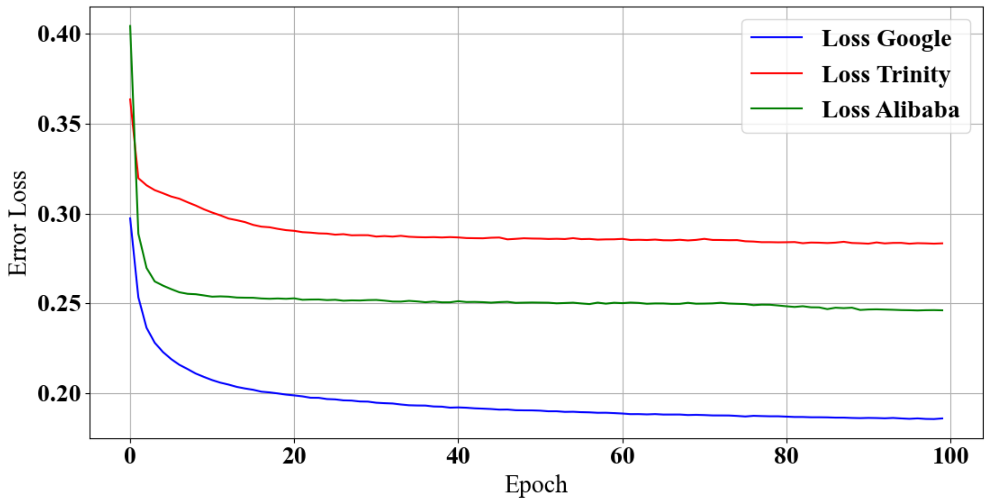


Figure 4.18: Error Loss of CNN using three datasets

4.5 Summary

We proposed a proactive service availability framework that can predict the termination status for a given set of applications tasks during the runtime. The framework uses deep learning ANN and CNN models for prediction. Both ANN and CNN were trained and tested using three datasets from Google, Alibaba, and Trinity clusters. The results show that ANN can predict task failure with high accuracy of around 94%, and a low error loss of around 14% using the Google dataset. CNN can predict task failure with an accuracy of around 92%, and error loss of around 18% using the same dataset.

Chapter 5

Failure-Aware Application Task Scheduling

Scheduling application tasks plays an important role in the availability of the task and the overall application. For example, scheduling a task at a computing node with a high available value will increase the availability of the task, while scheduling it at a node with lower availability will decrease the availability of the task. Successful termination of task helps to save and utilize resources of the cluster where the task is executed. In case of task failure, the task may need to be rescheduled several times before it terminates successfully that leads to waste more resources compared to what the task demands. Scheduling a set of tasks at the underlying computing nodes that are located in cloud DCs is considered NP-hard problem. This motivated us to propose a local heuristic solution for scheduling tasks that are predicted as failed to boost their chance to terminate successfully. In addition, we propose local heuristic solution to schedule containers to utilize resources of computing nodes. The rest of this chapter is organized as follows. Section 5.1 discusses thoroughly the failure-aware scheduler module of the proposed framework that is discussed in Section 4.3 of the previous chapter. Section 5.2 introduces heuristic solution for container scheduling for resource utilization at DC, and Section 5.3 provides a summary of the chapter.

5.1 Failure-Aware Task Scheduling

Failure-aware task scheduler is a core module of the proactive framework of Section 4.3. The main purpose of the module is to take a set of remedy actions including scheduling at computing nodes for the tasks that are predicted as failed by the prediction module of the same framework. We propose a set of remedy actions that can be taken by the scheduler module based on the analysis of the trace log files of each platform. From Google trace logs, we consider the following set of remedy actions: {*change task scheduling node, change task priority, change task scheduling level*}. Although there are other actions such as resource demands that can play a role in the termination status of the task, we believe such actions should not be changed because they are considered as requirements. From Alibaba trace logs, we consider only the remedy action {*change task scheduling node*} because the traces do not include other scheduling information that can be changed for the tasks, such as priority and scheduling level. From Trinity trace logs, we do not consider any action because the trace logs miss the configuration information and scheduling nodes for jobs. Note that the set of remedy actions is an input for the failure-aware scheduler module. We define the actions for Google and Alibaba datasets in the Possible Actions repository of the framework. For any new platform, the actions have to be defined as well. Although the actions are limited, the scheduler module faces two main challenges. The first challenge is to determine the order of failed tasks to apply the remedy actions. The second challenge is to select the actions. To illustrate the actions selection challenge, for example in the Google dataset, there are 12,500 available nodes to schedule tasks, where task priority values range between 0-11, and task class level values range between 0-3. This leaves the scheduler module with a very large combination of actions equal to 12,500*12*4= 600,000 to take for only one task. To handle all the tasks, this number is multiplied by the total number of tasks.

Actually, knowing the sequence of tasks and actions plays a key role in the performance of the cluster in general, and in the termination status of tasks in particular. We believe using the brute force approach to know the best sequence of tasks and actions will take a longer time and require higher computations. Therefore, the brute force approach is not a feasible solution, especially where the actions have to be selected and taken during the runtime, and where some tasks can not tolerate the delay due to their requirements. Therefore, we map tasks sequence and actions selection as an optimization problem with multiple constraints. We formulate the problem as an integer linear programming (ILP) model with an objective function and a set of constraints. In addition, we propose a local heuristic solution for the model. The objective function is to find a set of actions to apply for a set of tasks that are predicted as failed, in such a way that minimizes the failure probability of tasks as well as their resources (CPU and RAM) usage. The objective function is formalized in Equation (5.1), where T is the set of tasks that are predicted as failed by the prediction module, A is the set of actions that are available in the possible actions repository and can be taken for T, $failure_{ta}$ is the failure probability, $CpuUsage_{ta}$ is the CPU usage, and $RamUsage_{ta}$ is the RAM usage of task t if the action a is taken. The binary decision variable X_{ta} with value 1 indicates that action a is taken for task t, and 0 value otherwise, as defined in Equation (5.2).

$$\text{minimize} \sum_{t=1}^{|T|} \sum_{a=1}^{|A|} (failure_{ta} + CpuUsage_{ta} + RamUsage_{ta}) \times X_{ta} \tag{5.1}$$

$$X_{ta} = \begin{cases} 1, \text{ if action } a \text{ is taken for task } t \\ 0, \text{ otherwise} \end{cases} \tag{5.2}$$

The set of constraints for the model, as in Equations (5.3 - 5.9), are defined to control the actions selection process and respect the task scheduling requirements. The constraints force the following: at least one action a should be taken for any task t as formalized in Equation (5.3). Task t should be scheduled on a maximum of one node, where N is a set of nodes that are available for scheduling tasks as in Equation (5.4). Note that $N \in A$, where the scheduling nodes are considered as actions that can be taken for tasks $\in T$. In other words scheduling task t at node $n \in N$ is considered as action $a \in A$. The selected node $n \in N$ should have enough resources, CPU, RAM, and Disk, to host task t as in Equations (5.5 - 5.7). Note that we assume scheduling task t at node $n \in N$ does not have any impact on the other tasks that are already scheduled on n. Task t should not be collocated with any other dependent task k on the same selected node $n \in N$ as in Equation (5.8), where the set D_i includes the dependent tasks. Task y should not be collocated with any other redundant (backup) task z on the same selected node as in Equation (5.9), where the set R_i includes the dependent tasks.

$$\sum_{a=1}^{|A|} X_{ta} \geq 1 \qquad \forall t \in T \tag{5.3}$$

$$\sum_{a=1|a \in N}^{|A|} X_{ta} \leq 1 \qquad \forall t \in T \tag{5.4}$$

$$\sum_{t=1}^{|T|} CPU_t \times X_{ta} \leq CPU_a \quad \forall a \in N \tag{5.5}$$

$$\sum_{t=1}^{|T|} RAM_t \times X_{ta} \leq RAM_a \quad \forall a \in N \tag{5.6}$$

$$\sum_{t=1}^{|T|} Disk_t \times X_{ta} \leq Disk_a \quad \forall a \in N \tag{5.7}$$

$$X_{ta} + X_{ka} = 1 \quad \forall t \textbf{ and } k \in T, \forall t \textbf{ and } k \in D_i, \forall a \in N \tag{5.8}$$

$$X_{ya} + X_{za} = 1 \quad \forall y \textbf{ and } z \in T, \forall y \textbf{ and } z \in R_i, \forall a \in N \tag{5.9}$$

5.1.1 Heuristic Solution

We propose a local heuristic approach to solve the proposed ILP model and find the sequence of tasks and remedy actions that achieve the objectives. To avoid the burden of the brute force approach, the heuristic approach aims to reduce the search space to find a solution for ILP model in a short time. Therefore, we divide the scheduling nodes that are defined in the Possible Actions repository into three groups according to the availability of the nodes regardless of the input dataset. This categorization is based on our assumption that can be used in ILP model. The high availability group includes nodes with availability values greater than or equal to 0.9, medium availability group with values greater than or equal to 0.75 and less than 0.9, low availability group with values lower than 0.75. We compute node availability as the ratio of the node failure to its recovery as defined in Equation (5.10), where MTTF refers to the mean time between two consecutive failures, and MTTR refers to the mean time to repair the node after the failure. Both Google and Alibaba datasets include information about events of nodes such as adding, updating, and removing the nodes including the timestamp of each event. We consider events updating and removing as node failure, and adding events as node repair. The heuristic approach tries to schedule tasks with high failure probability at the nodes with high availability values to boost the chances of tasks terminating successfully.

$$Availability(node) = \frac{MTTF_{node}}{MTTF_{node} + MTTR_{node}} \tag{5.10}$$

Algorithm 2 describes the proposed local heuristic approach to find the sequence of tasks and their remedy actions. The algorithm takes the tasks map T that is sent by the prediction module, and the reference R for the Possible Actions repository as input. It returns the map *TasksRemedyActionsMap* that includes the sequence of tasks and their remedy actions to be taken. The cloud manager at the platform applies the remedy actions. The algorithm begins by initializing the map *TasksRemedyActionsMap* (line 1), and the list of ultimately failed tasks $UltimateFailedTasksList$ (line 2) with a Null value. Then it sorts tasks $\in T$ in descending order according to the failure probability (line 3). This is to give priority to the tasks that have high failure probability values. It retrieves the set N of scheduling nodes (line 4), and the set A of configuration (line 5) possible actions from R. The algorithm groups and sorts the nodes N into the set G in descending order according to their availability (line 6). For each task $t \in T$, the algorithm considers initially that no solution exists for t (line 8), and the list of remedy actions *RemedyActionsList* is empty (line 9). The minimum cost *minCost* of the selected remedy for t is initialized with a high value (line 10). The remedy actions selection process starts with actions of type scheduling node. It searches for scheduling node n per group $g \in G$ (line 11) instead of all the nodes $\in N$. This will help to reduce the search space to find the scheduling node, and hence reduce the execution time to find the solution. For each node n, the algorithm checks if n has enough resources such as CPU and RAM to host task t, and that task t has no conflict, such as redundancy, with any other tasks that are hosted on n (line 13). If n passes the constraints check to host t, the algorithm sets n as a candidate scheduling node for t (line 14). With each selected n, the algorithm tries all other possible configuration actions $\in A$ (line 15). For each configuration $a \in A$, it sets a for t (line 16). With the new actions, the algorithm uses the selected prediction model (ANN, CNN) by the prediction module to predict the termination status of t (line 17). If the predicted termination status of t is changed to 'Terminated' (line 17), the algorithm marks a solution is found for t (line 18). Then the *cost* for the selected actions is computed as the summation of the failure probability, predicted CPU, and RAM usage of t (line 19). If the *cost* is less than the current minimum cost *minCost* (line 20), the algorithm sets *cost* as minimum cost *minCost* (line 21), n as a candidate scheduling node remedy action (line 22), and a as a candidate configuration action (line 23) that achieve the minimum cost so far. Once the approach tries all the scheduling nodes $\in$ group g and the actions $\in A$, it checks if a solution is found so far for t (line 29). If so, the approach adds n (line 30) and a (line 31) that achieve the minimum cost to the list of remedy actions *RemedyActionsList*. Then, a tuple includes t and its corresponding remedy actions that are stored in *RemedyActionsList* is added to the map *TasksRemedyActionsMap* (line 32). If no solution is found for t yet, in other words, the termination status of t is still 'Failed', the algorithm continues exploring the solution using the next $g \in G$ (line 11). The algorithm tries all the actions and if the predicted termination status stays as 'Failed', the algorithm considers these tasks to fail ultimately. For each task t that is not located in the map *TasksRemedyActionsMap*, the algorithm adds t to the list of ultimate failed tasks *UltimateFailedTasksList* (lines 37-41). At the end, the algorithm returns the map *TasksRemedyActionsMap* and terminates (line 42). Note that the cloud manager can use *TasksRemedyActionsMap* to execute the remedy actions for the corresponding tasks at the cloud platform.

5.1.2 Computational Complexity Analysis

The failure-aware scheduler module uses the approach described in Algorithm 2 to handle tasks that are predicted as failed. In the beginning, the algorithm sorts the input tasks map T and the scheduling nodes set N, and groups them based on the availability into the set G. Here any sorting algorithm can be applied. Sorting T costs $\mathcal{O}(nlogn)$ execution time, and grouping N costs $\mathcal{O}(n)$ execution time. After sorting and grouping, the algorithm handles each task $t \in T$ individually. So for each t, it tries to find the candidate scheduling node n and the configuration a as remedy actions for t that achieves the minimum cost. We optimize the actions selection process to reduce the number of selected actions. So the algorithm sorts and divides the nodes set N into groups set G based on the availability. This is to reduce the search space to find remedy actions for t including the scheduling node n. So for each $t \in T$, the algorithm searches for n in each group $g \in G$ starting from g with the highest availability. With each $n \in g$, the algorithm tries each configuration action $a \in A$ for t. At the end of each g and A, the algorithm checks if a solution is found for t that includes n and a for which the termination status of t is changed to 'Terminated' and achieve the minimum cost. If so, the algorithm stops exploring other actions for t. If not, it continues exploring other actions starting from the next g. In the average case, our experiments show, the prediction process takes $\mathcal{O}(n)$ for each t. The actions selection process takes logarithmic time $\mathcal{O}(logn)$ for each t. So for all the n number of tasks in T, the prediction and actions selection processes cost an average $\mathcal{O}(n^2 + nlogn)$ execution time. In the worst-case scenario which has a low chance to happen, the scheduler may

Algorithm 2 Failure-Aware Task Scheduler

```
Input: Tasks Map $T$, Possible Actions Repository $R$
Output: Tasks Actions Map $TasksRemedyActionsMap < Task, Actions >$
 1: $TasksRemedyActionsMap = Null$
 2: $UltimateFailedTasksList = Null$
 3: Sort $T$ descending based on failure probability
 4: $N \leftarrow R.getScheduleNodesActions()$
 5: $A \leftarrow R.getConfigurationActions()$
 6: $G \leftarrow$ group and sort $N$ descending based on availability
 7: for each $t \in T$ do
 8:     $findSolution = False$
 9:     $RemedyActionsList = Null$
10:     $minCost = \infty$
11:     for each $g \in G$ do
12:         for each $n \in g$ do
13:             if $(n.getResources() \geq t.getDemandResources())$ and
    ($t$ has no dependency or redundancy with any other task hosted on $n$) then
14:                 $t.setScheduleNodeAction() \leftarrow n$
15:                 for each $a \in A$ do
16:                     $t.setConfigurationAction() \leftarrow a$
17:                     if ($predModel.predictTerminationStatus(t)$ ==‘Terminated’)
    then
18:                         $findSolution =$ True
19:                         $cost = predModel.predictFailProbability(t)+$
    $predModel.predictCpuUsage(t)+predModel.predictRamUsage(t)$
20:                         if ($cost < minCost$) then
21:                             $minCost \leftarrow cost$
22:                             $minSchedNodeAction \leftarrow n$
23:                             $minConfigAction \leftarrow a$
24:                         end if
25:                     end if
26:                 end for
27:             end if
28:         end for
29:         if $findSolution$ then
30:             $RemedyActionsList.add(minSchedNodeAction)$
31:             $RemedyActionsList.add(minConfigAction)$
32:             $TasksRemedyActionsMap.add\ (t, RemedyActionsList)$
33:             break
34:         end if
35:     end for
36: end for
37: for each $t \in T$ do
38:     if $t \notin TasksRemedyActionsMap$ then
39:         $UltimateFailedTasksList.add(t)$
40:     end if
41: end for
42: return $TasksRemedyActionsMap$
```

explore all the actions in the repository R including all the nodes in the groups, and the prediction termination status of each $t \in T$ stays 'Failed' (not changed). In such a case, for T the prediction and actions selection processes cost $\mathscr{O}(2n^2)$, which is greater than the average-case scenario.

5.1.3 Experiments and Results

For the experiments to evaluate the failure-aware task scheduler, we used the same setup to evaluate the modules of the proactive framework as described in Section 4.4 of the previous chapter. We evaluated the ability of the failure-aware task scheduler module for taking the right remedy actions and protecting tasks that were predicted as failed. As shown in Figure 5.1, using Google dataset the scheduler could protect around 185,000 (37%) out of 500,000 total input failed tasks by changing the scheduling node among 12,500 input available nodes, 159,000 (31.8%) by changing task priority, and 29,000 (5.8%) by changing the scheduling class level. We notice the scheduling node as a remedy action has more effect on the task termination status than the priority and class level because the failure of the node will lead to an inevitable failure of all tasks that are executed on the node. Task priority has more effect than the scheduling class level. We believe this is because priority plays a key role in the task eviction. In addition, the range of the priority values is larger than the range of the class level values. As a result, the scheduler has more options to change the priority of the task and finish successfully. To check the impact of the available nodes in the cluster, we found the ratio of the protected tasks to the scheduled tasks increases by increasing the number of the nodes. As shown in Figure 5.1, the number of the protected tasks by changing the scheduling node using 12,500 nodes is 185,000 (37%), which is higher than 170,000 (34%) protected tasks using 1,300 nodes. This is because the scheduler will have more chances to reschedule the failed task, hence the task will have a higher chance to terminate successfully. Using the Alibaba dataset, the scheduler could protect around $7,052$ (40%) of all $17,665$ input failed tasks by changing the scheduling nodes of tasks as the only available remedy action by the dataset. The scheduler could save more tasks using the Alibaba dataset than using the Google dataset. We believe this is because the scheduling node in Alibaba dataset plays a crucial role in the success or failure of the tasks that are hosted on the node. As shown in Figure 4.11, some nodes in the Alibaba cluster failed more frequently. So rescheduling the task on a stable and highly available node can boost the chance of the task to terminate successfully. A similar figure that can show the number of the protected tasks using the Alibaba dataset is not possible, since it includes only one remedy action.

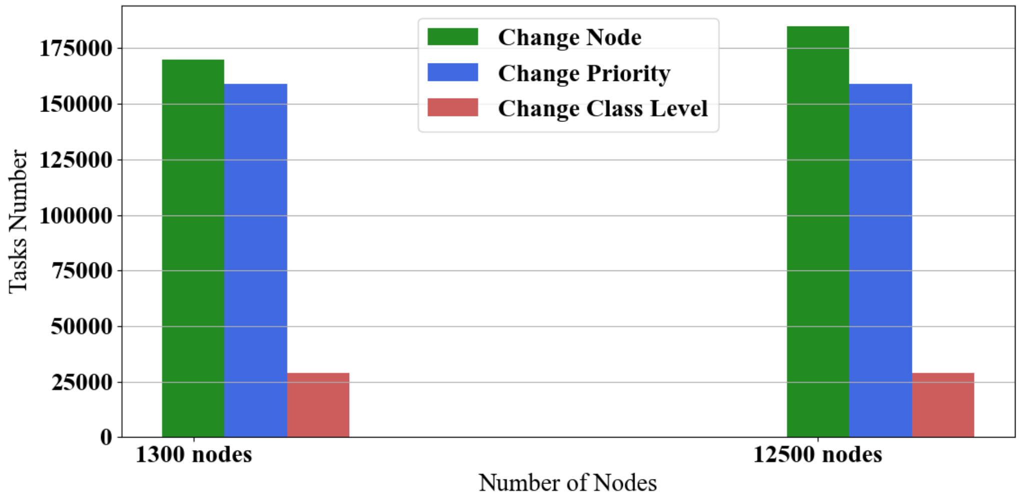


Figure 5.1: Protected Tasks - Google Dataset

To recognize the significance of early task failure prediction and taking the remedy actions, we compare the resource usage with and without using our proposed proactive failure-aware task scheduling framework. We assume the prediction process is done after 10% of elapsed execution time for each task. Figures 5.2, 5.3, 5.4, and 5.5 depict the comparison for the usages of CPU, RAM, Disk space, and Disk I/O respectively with and without using the framework for tasks that have been executed during the first day at Google cluster. As we can see, using the framework will reduce the resource usage of different types remarkably. As shown in the Figures 5.2, and 5.3, at some points in time the framework saves up to 2,000 CPU and 1,500 RAM units. This is because the framework will help to detect tasks failures at an early stage of their execution and take the appropriate remedy actions immediately to protect the tasks from subsequent failures. In other words, the framework will not wait until the tasks actually fail and resubmit them, which can save the wastefulness of the resources. Using the Alibaba dataset, it is difficult to show the significance of the framework in saving the resources, and this is because the ratio of the failed tasks is very small compared to the terminated task. In addition, the failed tasks are spread during the 12 hours trace period. However, it is worth mentioning here, for the Alibaba dataset our framework could save a total of 9,684 CPU units during the 12 hours. We could not compute the saved RAM units because many of tasks in the input trace logs miss this information. Using the Trinity dataset, the prediction module could predict the termination status for a given set of jobs, however, the scheduler module could not take remedy actions for the failed jobs and this is because the trace logs miss the scheduling nodes and configuration parameters for the jobs. Therefore, we assumed the jobs that are predicted as failed will be terminated immediately without taking any further action. In other words, the failed job will not be resubmitted. Figure 5.6 shows the amount of CPU units that our framework could save during the entire period of Trinity trace logs, as a result of terminating the jobs that are predicted as failed. Other figures that show the amount of saved resources, such as RAM and Disk, are not possible to generate for the Trinity dataset, since the dataset includes only the CPU information of the jobs.

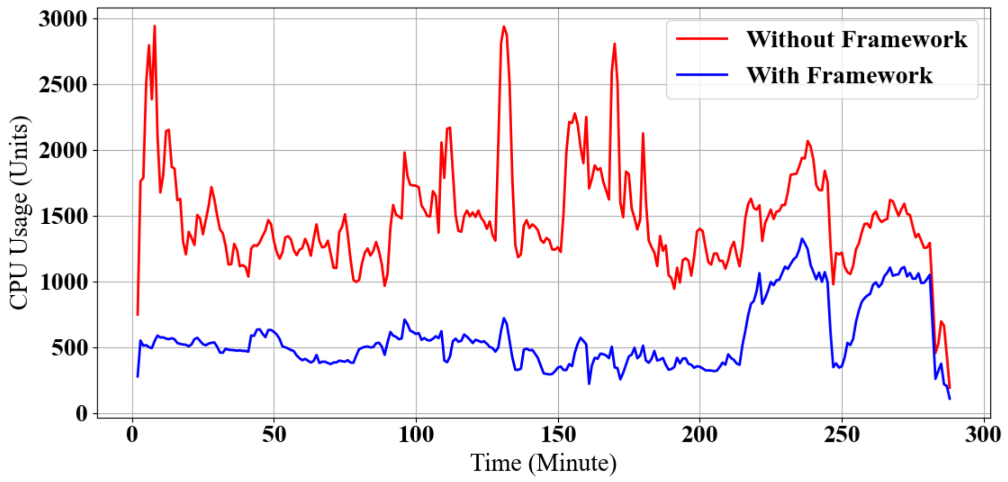


Figure 5.2: CPU Usage Comparison - Google Dataset

To measure the scalability and performance of the framework, we measure the average execution time that the framework takes to predict the termination status for different sets of tasks and to take the right remedy actions for the tasks that are predicted as failed. Using both Google and Alibaba datasets, we measure the average execution time for 7 sets of tasks with different sizes that range between 2,000 and 128,000 tasks. Using the Google dataset, we measure the average execution time for the same sets of tasks using 3 different sets of computing nodes with different sizes that are equal to 1,300, 5,000, and 12,500 (maximum number) nodes. Using the Alibaba dataset, we measure the average execution time for the same input sets of tasks using only one set of computing nodes that is equal to 1,300 nodes which is the maximum number of nodes that are available in the

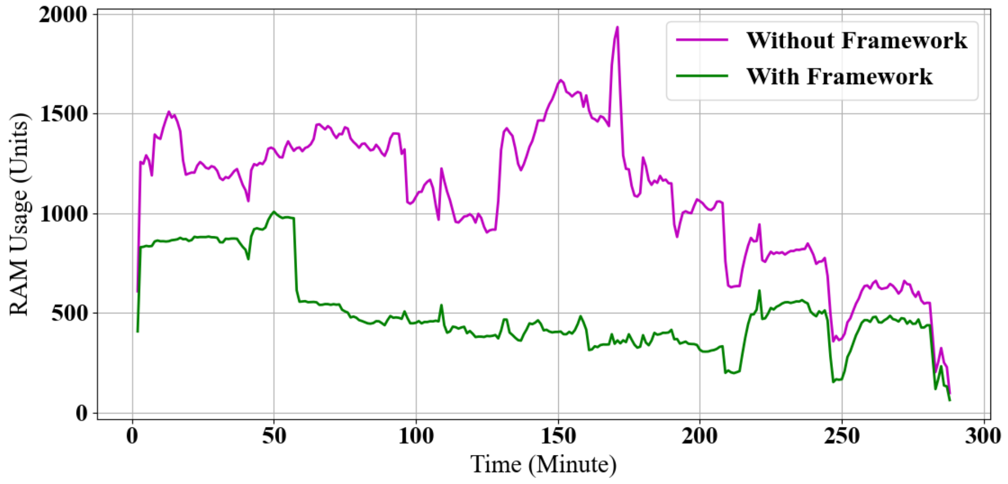


Figure 5.3: RAM Usage Comparison - Google Dataset

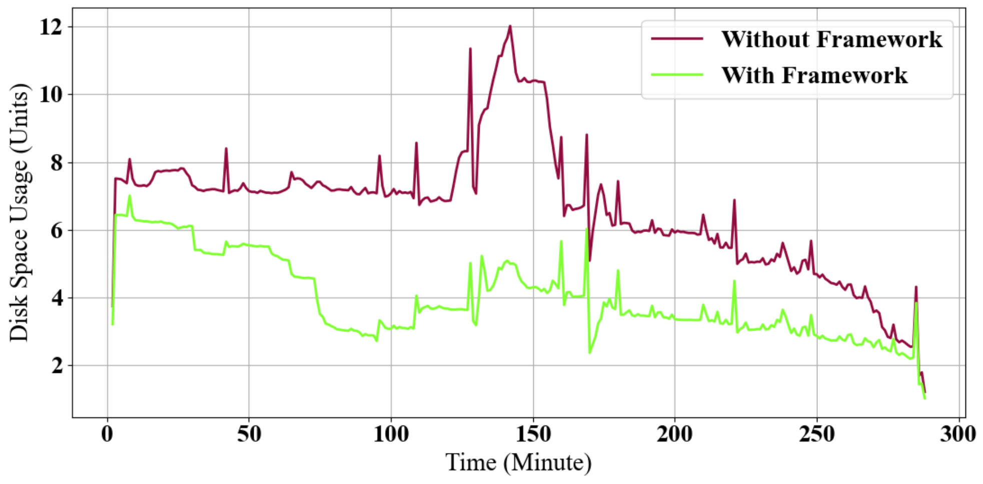


Figure 5.4: Disk Usage Comparison - Google Dataset

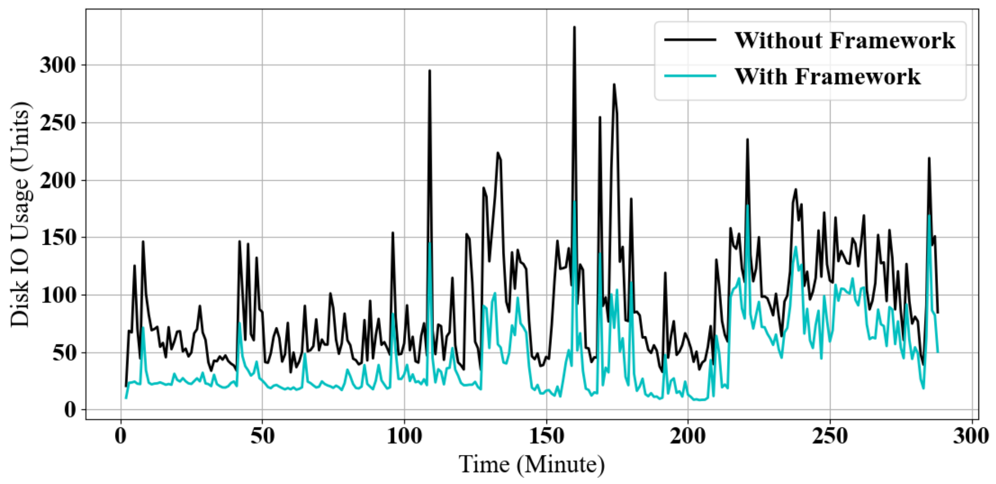


Figure 5.5: Disk IO Usage Comparison - Google Dataset

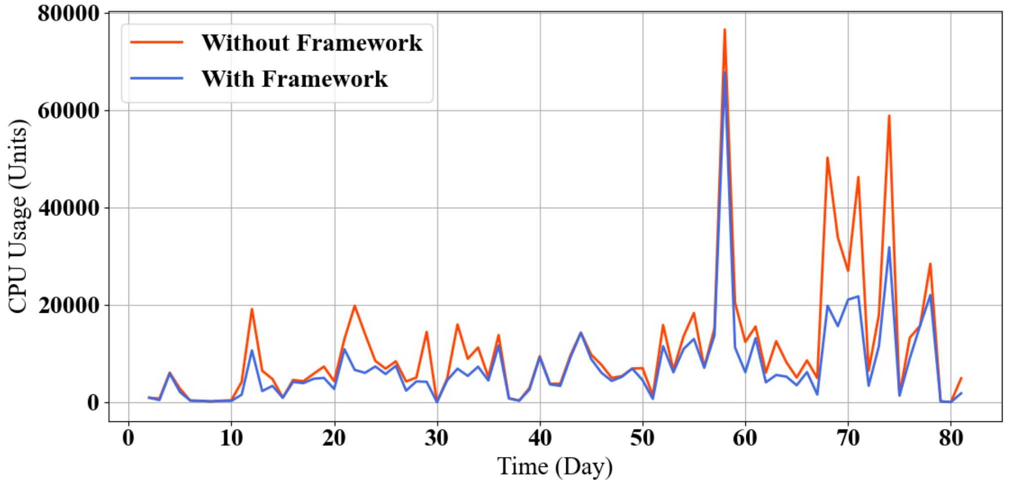


Figure 5.6: CPU Usage Comparison - Trinity Dataset

Alibaba computing cluster. Note that all the tasks and nodes sets from both Google and Alibaba datasets are selected randomly. As shown in Figure 5.7, using Alibaba dataset the framework took less execution time than using Google dataset for the same size of input tasks sets and using 1,300 available computing nodes. For example, the framework took around 8 minutes to predict and take remedy actions for tasks set of size 128,000 using 1,300 nodes that are taken from Alibaba dataset, while the scheduler took around 9.5 minutes to predict and take actions for the same size of tasks set and nodes set that are taken from Google dataset. We believe this is because the number of the remedy actions that are available from Google dataset for the scheduler is higher than the number of actions that are available from the Alibaba dataset. The scheduler, therefore, tries more actions to protect tasks that are predicted as failed when using the Google dataset than using the Alibaba dataset. In addition, using the Google dataset the scheduler has a high chance to deal and handle a higher number of failed tasks than using the Alibaba dataset even for the same input tasks set. This is because the Alibaba dataset has a smaller number of failed tasks as compared to the Google dataset. As a result, the scheduler takes more time to handle tasks that are taken from the Google dataset. To check the impact of the available nodes in the cluster on the performance of the framework, we found the more nodes are available the longer execution time the framework takes. As shown in Figure 5.7, the framework took around 7.7 minutes to handle 32,000 tasks using 5,000 nodes, while it took a longer execution time of around 9 minutes to handle the same set of tasks using 12,500 nodes. This is because, with a high number of computing nodes, the scheduler will have more nodes options to check and compute the cost function to select the scheduling node for each task. Hence, the scheduler will take a longer execution time to process all the given tasks.

5.2 Container Scheduling for Resource Utilization

5.2.1 Container Scheduling Model

Scheduling of containers at the underlying computing nodes that are located at the DC plays an important role to utilize the resources of the nodes. Resource utilization of nodes helps to enhance their performance and availability which can be reflected on the containers that provision the application services. In other words, resource utilization at container-based cluster helps to enhance the availability and quality of the provided services. Analysis of

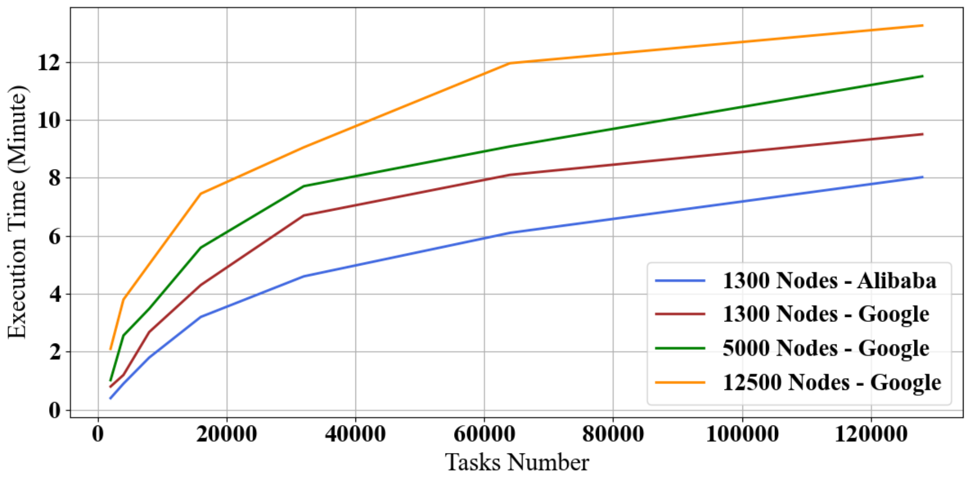


Figure 5.7: Average Execution Time of the Failure-Aware Scheduler

Alibaba dataset for container-based cluster [21] show that the average resources usage of the containers is very low compared to the average resource demand of the same containers. As shown in Figure 5.8, the maximum average CPU usage is 13.7% out of the total 6 CPU units demanded. Figure 5.9 shows that the maximum average RAM usage is 60% out of the total 5 RAM units requested.

As we notice there is a big difference between the demand and usage of the resources by the containers that were executed in the Alibaba computing cluster. We believe this is due to the container scheduling issue, where the used scheduler does not map the demand resources with the current usage of the computing nodes (servers in the Alibaba cluster). This motivated us to propose a scheduling solution for containers at computing servers in DCs. We formalized container scheduling as ILP model with an objective and set of constraints. The main objective of the scheduling process is to schedule containers to be hosted at the computing servers based on their resources demands, in such a way to minimize the total resources usage of the containers as defined in Equation (5.11). Minimizing the resources usage by the containers helps to reduce the workload at the servers that help to increase their availability, hence this is reflected on the availability of the service that is related to the container that is hosted at the server. The $resourcesUsage(container_i)$ is the resource usage of the $container_i$ that is predicted by ANN method. The binary decision variable X_{ij} with value equals to 1 indicates that $container_i$ is hosted on the $server_j$, and with value 0 otherwise as in Equation (5.12). The selected $server_j$ should have enough resources (CPU,

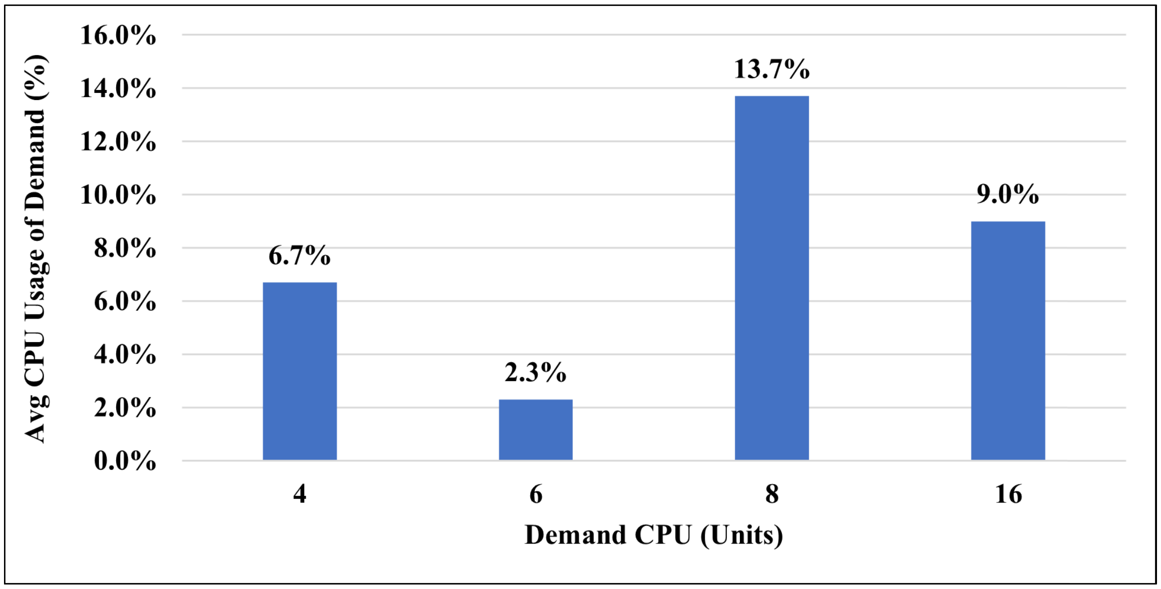


Figure 5.8: Containers Average CPU Usage out of the Demand

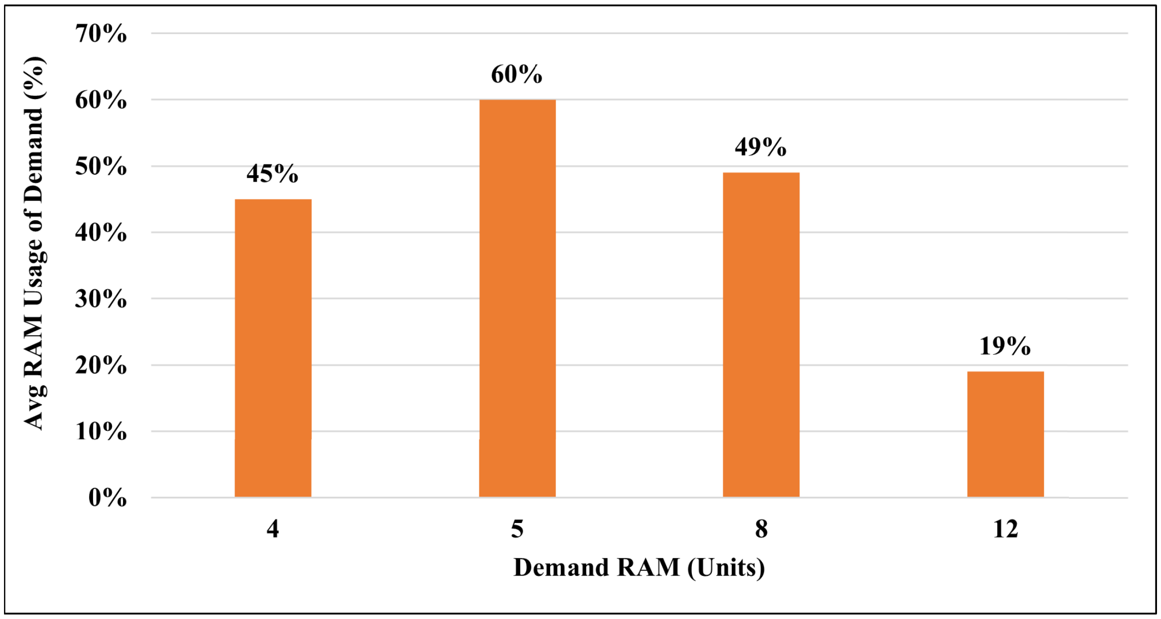


Figure 5.9: Containers Average RAM Usage out of the Demand

Ram, and Disk) to host $container_i$. So we define a set of constraints as defined in Equations (5.13-5.15) to control the scheduling process.

$$\text{minimize} \sum_{i=1}^{|C|} \sum_{j=1}^{|S|} resourcesUsage(container_i) \times X_{ij} \tag{5.11}$$

$$X_{ij} = \begin{cases} 1, & \textit{if } server_j \textit{ hosts } container_i \\ 0, & otherwise \end{cases} \tag{5.12}$$

Subject to:

$$\sum_{i=1}^{|C|} requestedCpu(container_i) \times X_{ij} <= availableCpu(server_j) \forall server_j \in S \tag{5.13}$$

$$\sum_{i=1}^{|C|} requestedRam(container_i) \times X_{ij} <= availableRam(server_j) \forall server_j \in S \tag{5.14}$$

$$\sum_{i=1}^{|C|} requestedDisk(container_i) \times X_{ij} <= availableDisk(server_j) \forall server_j \in S \tag{5.15}$$

5.2.2 Heuristic Solution

We propose a local heuristic approach that uses deep learning ANN to solve ILP model and schedule containers to be executed at servers available in the cluster. ANN is trained and tested using a clean dataset that was extracted from Alibaba dataset according to specified features. After we analyzed Allibaba dataset, we found the following features that can have a correlation with the container resources usage and demand: *<containerID, serverID, cpuDemand, ramDemand, diskDemand, containerCpuUsage, containerRamUsage, containerDiskUsage, serverCpuCapacity, serverRamCapacity, serverDiskCapacity, serverCpuUsage, serverRamUsage, serverDiskUsage >.*

Algorithm 3 describes the local heuristic approach. It takes the set of Containers C that are waiting to be scheduled and a set of available Servers S as input. It returns the map $schedulingMap$ that includes scheduling and placement of Containers at Servers as output. In the beginning, the algorithm sorts the Containers set C in descending order according to their demand of resources, and store them in the list $sortedContainersList$ as per line 1. For each container c located in sorted list $sortedContainersList$, the algorithm finds the server s that is capable to host c, and c consumes the minimum number of the resources. This is

Algorithm 3 Heuristic Containers Scheduling

```
Input: set of Containers $C$, set of Servers $S$
Output: $schedulingMap < container, server >$
 1: Sort $C$ based on demand resources in decreasing order, and store them in $sortedContainersList$
 2: for each $c \in sortedContainersList$ do
 3:     $minResUsage = \infty$
 4:     $minServer = Null$
 5:     for each $s \in S$ do
 6:         if $s$ can host $c$ then
 7:             set c.setServerID()=s.getServerID()
 8:             $totContResUsage = ANN.predCpuUsage(c) + ANN.predRamUsage(c) + ANN.predDiskUsage(c)$
 9:             if $totContResUsage < minResUsage$ then
10:                 $minResUsage = totContResUsage$
11:                 $minServer = s$
12:             end if
13:         end if
14:     end for
15:     add $< c, minServer >$ to $schedulingMap$
16: end for
17: return $schedulingMap$
```

because the allocated resources may be different than the demanded resources. So for each $s \in S$, it checks if server s has enough resources to host c as per line 5. If so, the algorithm schedules c to be executed at s (line 7). After that, the total resources usage as a result of placing c at s is computed. The total usage equals the summation of predicted CPU usage, Ram usage, and Disk usage by c as per line 8. Each of these resource usage is predicted using the deep learning ANN as a regression prediction value. The algorithm checks if the total usage is less than the current minimum usage (line 9), if yes it replaces the current minimum usage by total usage and marks s as minimum server as per lines 10-11. Once the algorithm checks all servers $\in S$ for container c, it finds the server s where c consumes minimum resources. As a result, it adds the pair $< c, minServer >$ to the scheduling map $schedulingMap$ as per line 15. At the end, the algorithm returns the map $schedulingMap$ as per line 16.

5.2.3 Experiments and Results

For the experiments to evaluate the proposed container scheduling heuristic approach, we used the same setup for the experiments that were conducted to evaluate the prediction module as in section 4.4. Before we train ANN with the Alibaba dataset, we cleaned the dataset and then extracted the corresponding features that are listed above. For example, some attributes such as Disk usage are missing (null values). For these, we replace the null with zero values. In addition, some containers have a failure termination event type. We do not consider failed containers. We only consider the containers that have terminated successfully as indicated in the trace log. We split 90% of the input cleaned dataset into a training set and 10% for a testing set. Both sets are given to deep learning ANN for training and testing. ANN is trained for 10 epochs and then tested. The structure of the used ANN includes one input layer with 14 neurons, two hidden layers with 100 neurons each, and one output layer with 3 neurons.

We computed Mean Square Error (MSE) and Mean Absolute Error (MAE) to evaluate ANN. MSE measures the prediction error difference between output and input values. MAE measures the prediction absolute error difference between output and input values. Figure 5.10 depicts the MSE for ANN, as shown ANN has the ability to predict resources CPU, Ram, and Disk, with a small prediction error that gives a good indication for the accuracy of the method. Figure 5.11 depicts the MAE of ANN, as shown ANN also achieves a small absolute prediction error. We believe this is because ANN is trained with large data

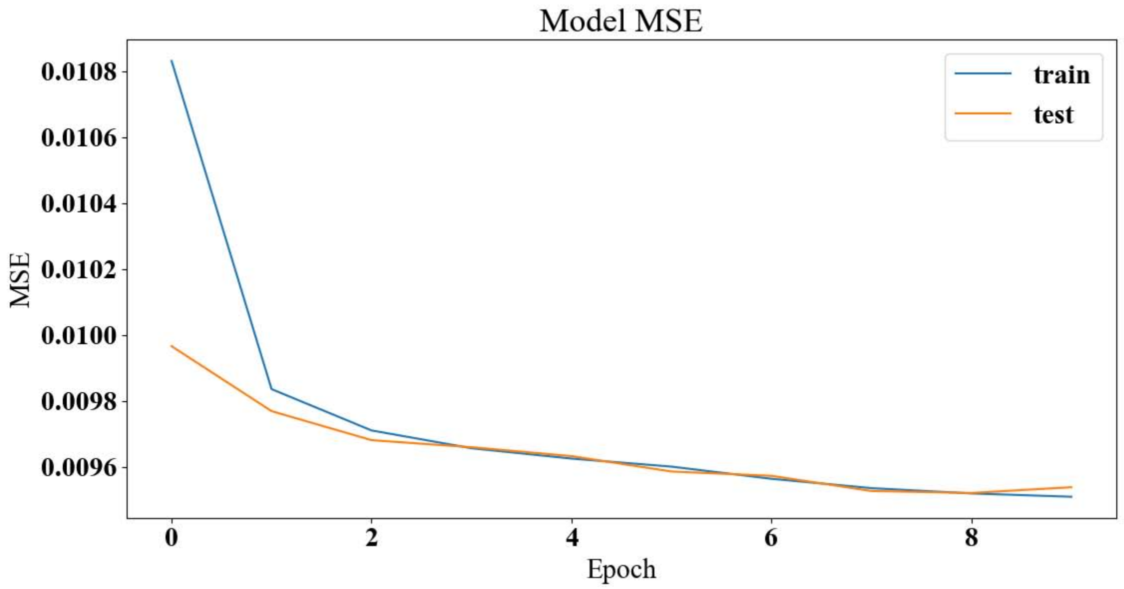


Figure 5.10: Mean Square Error of ANN

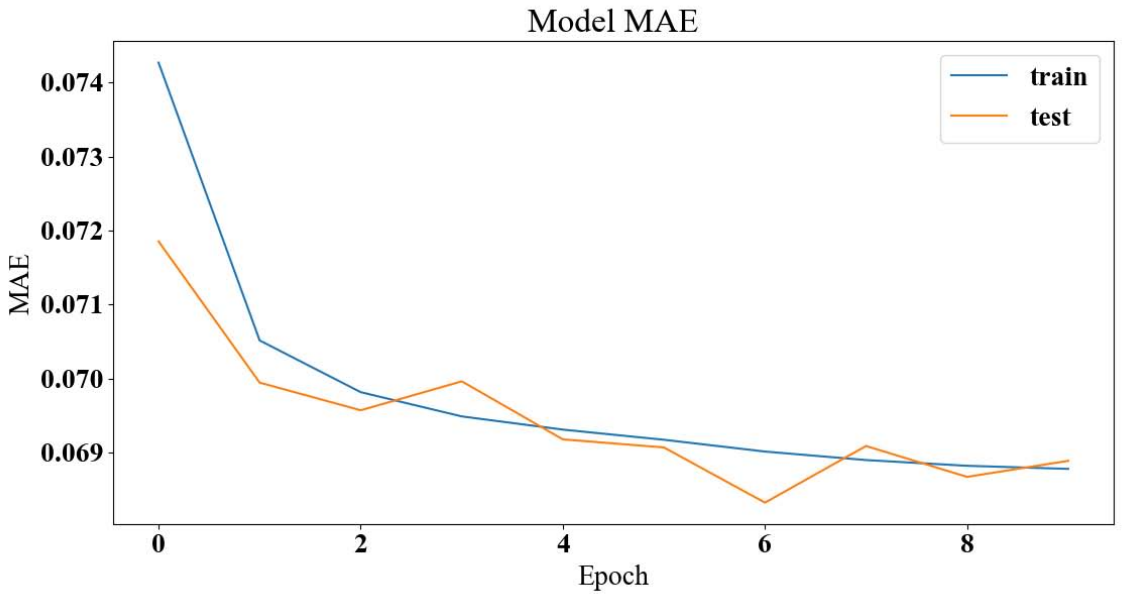


Figure 5.11: Mean Absolute Error of ANN

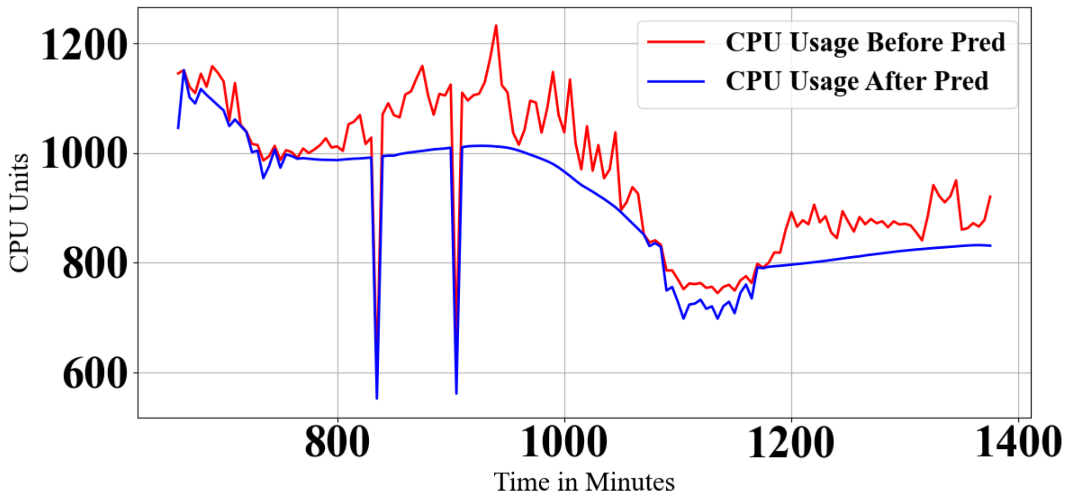


Figure 5.12: CPU Usage with/without using scheduling model

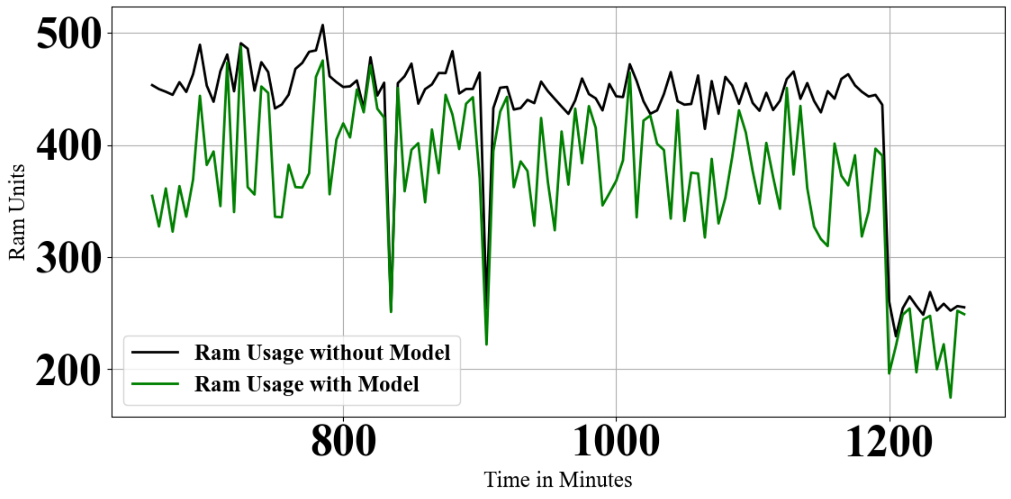


Figure 5.13: Ram Usage with/without using scheduling model

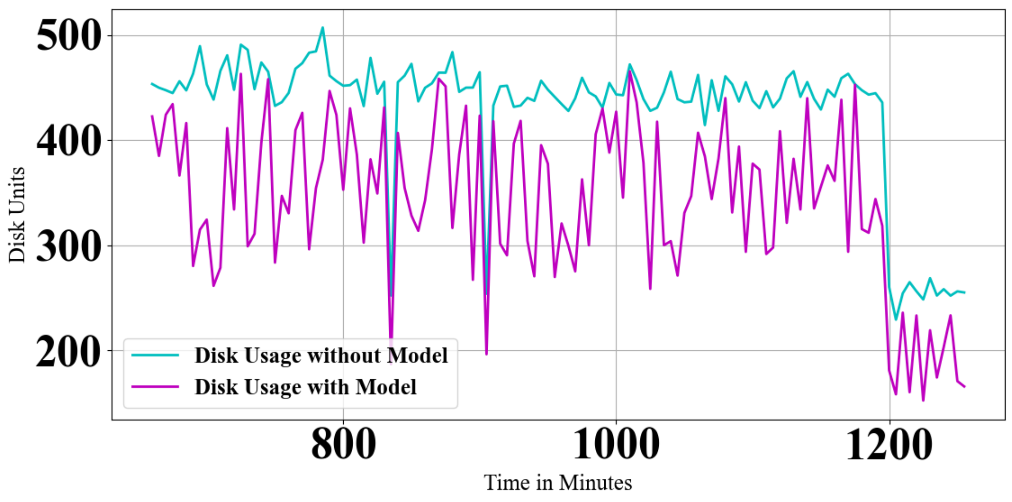


Figure 5.14: Disk Usage with/without using scheduling model

samples, and it passes through different layers and many iterations to reduce the prediction error. To evaluate the significance of using the proposed containers scheduling model, we compare the resources CPU, Ram, and Disk usages with and without using the model. Figures 5.12, 5.13, and 5.14 show the comparison for the CPU, Ram, and Disk usages respectively. Note that using the proposed model can reduce the resources usage during the execution time of the system. We believe this is because ANN is aware of the pattern usage of the containers and servers from the training dataset, so ANN can predict the resources usage of the container once it is scheduled to be executed on a specific server. The proposed scheduling approach searches for the server that has the capability to host the container and induces minimum resources usage. We could not plot the resources usage in the time period before 600 minutes in the figures 5.12, 5.13, and 5.14 because the information is missing in Alibaba dataset.

5.3 Summary

Failure-aware application task scheduler module is proposed in this Chapter. The module takes the remedy actions, including the rescheduling, for tasks that are predicted as failed. Task scheduling is formulated as ILP model with the objective to minimize the failure ratio, and resources usage of tasks. A local heuristic approach is proposed to find the scheduling plan for the tasks. The results show the ability of the scheduler module to protect up to 37%

and 40% of tasks that were extracted from Google, and Alibaba datasets respectively. In addition, the results show the scalability of the scheduler module to handle a large number of tasks at a cluster of a large number of computing nodes. Moreover, local heuristic optimization solution is proposed for containers scheduling at computing servers. The solution uses deep learning ANN to predict the resource usage of scheduling container at a specific server. The results show the ability of ANN to predict the CPU usage with lower MSE, and MAE, and could save a large amount of CPU units.

Chapter 6

Availability-Aware Dynamic VM Placement

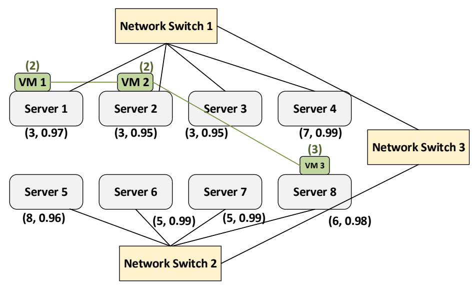
VM placement is a well-known combinatorial NP-hard problem to assign a set of VMs at a set of servers located at DC(s) and to target specific objective(s). VMs placement can be static that is triggered only for new VMs request. It also can be dynamic, where the placement process can be triggered at any time for new VMs request, or for already hosted VMs. Placement of the VM may require to be changed for several reasons such as service scaling, workload demand changes, and VM migration. Achieving multiple objectives through VMs placement can make the problem more complex that grows exponentially with growing number of the VMs. In this thesis, we tackle the dynamic VM placement problem for maintaining the availability of the provisioned applications in the cloud. We propose a framework that includes three modules to handle VMs placement during deployment, scaling, and failure of applications to achieve multiple objectives and meet the availability requirements of ASPs. We formulate application service availability in a cloud computing platform. In addition, we formulate the dynamic VMs placement problem as an INLP model with multiple objectives and a set of constraints. We propose to use AntColony heuristic optimization algorithm to find the placement plan of VMs at the underlying servers. The rest of this chapter is organized as follows. Section 6.1 discusses the formulation of application service availability in the cloud and the impact of dynamic VM placement at the availability level of the service. Section 6.2 introduces the dynamic VMs placement framework including the modules. Section 6.3 discusses the experiments and presents the results, and Section 6.4 provides a summary for the chapter.

6.1 Formulation of Application Service Availability in Cloud

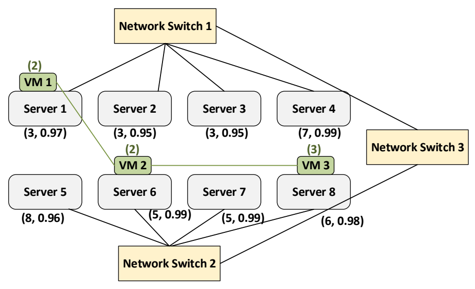
In cloud computing, ASP can request from CSP to deploy an end-to-end application service according to specific requirements including availability. As a result, CSP provides ASP with online access for a set of VMs to deploy the application components. An application component is simply software that provides a functionality type in a specific domain such as web (HTTP server), and networking (NAT, firewall). Each VM is deployed on only one physical computing node (server) at DC. Each VM belongs to a specific application and demands a set of resources of different types such as CPU and RAM. Each server has a specific capacity of each resource type and a set of properties such as availability to operate and host VMs. Availability of server j, A_j^s, can be computed as in Equation (6.1), where $MTTF_j$ refers to the mean time between two consecutive failures of server s_j, and $MTTR_j$ is the mean time to repair server s_j. Both MTTF and MTTR can be known for CSP via the operational patterns history of the server.

$$A_j^s = \frac{MTTF_j}{MTTF_j + MTTR_j} \tag{6.1}$$

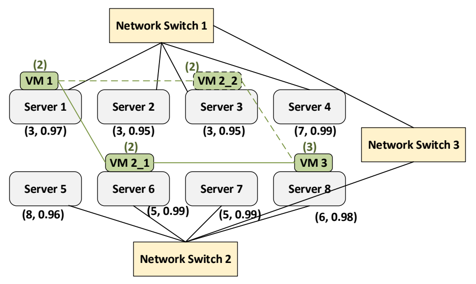
To illustrate how we formalize application service availability in cloud platform, we discuss the following example. Figure 6.1a shows an abstract model of application service app_1 that is composed of three different functionalities, where each functionality is provided through a separate VM, which are vm_1, vm_2, and vm_3. The service app_1 is requested by ASP with availability requirement of 0.9, and demand of resources for each VM. For simplicity, we show only the CPU demand beside each VM. Each VM is hosted on one server, we show the CPU capacity and availability beside each server. For availability purposes, we assume VMs that belong to the same application can not be collocated on the same server. Availability of VM depends on the availability of the server where the VM is hosted. If server fails, all the VMs that are hosted on the server will fail as well. So we consider availability of VM same as availability of the server that hosts the VM. We model application as a chain of functionalities that are provided through a set of VMs. Therefore, availability of application depends on the availability of all the functionalities that together provide end-to-end application service. Availability of application app_a that is denoted as A_a^{app} can be computed as the multiplication of the availability of all the functionalities that compose application app_a as defined in Equation (6.2), where



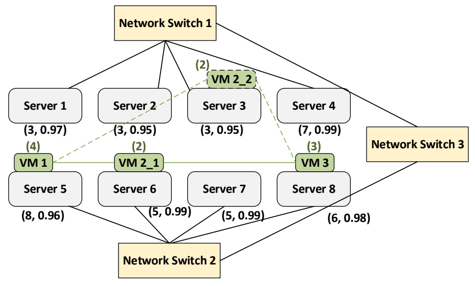
(a) app_1 Deployment 1



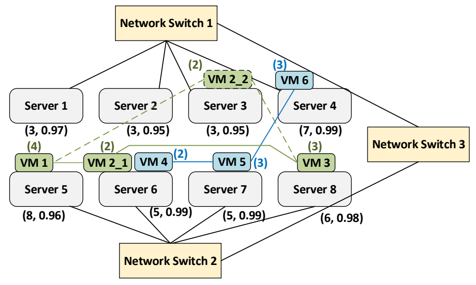
(b) app_1 Deployment 2



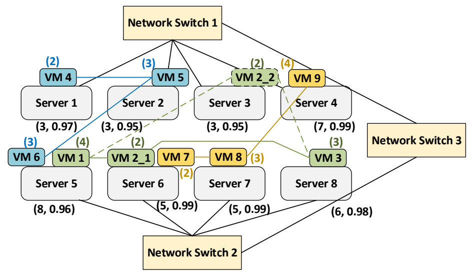
(c) app_1 Deployment 3



(d) app_1 Deployment 4



(e) app_1 and app_2 Deployment



(f) app_1, app_2 and app_3 Deployment

Figure 6.1: Applications Deployment Model in Cloud Data Center

A_f^{func} is the availability of the functionality $f \in F_a$, and F_a is the set of functionalities that are required to provide application app_a. Availability of functionality f depends on the availability of the VMs that provide f. In other words, functionality f is available as long as there is at least one VM that provides f. So A_f^{func} can be computed as the complement for the failure probability of all VMs that provide f as defined in Equation (6.3), where vm_v provides functionality f, and V_f^{func} is the set of all VMs that provide functionality f. Failure of vm_v is equal to failure of the server j that hosts vm_v. Failure of server s_j is equal to the complement of its availability as defined in Equation (6.4). According to Equation (6.2), availability of the deployed application app_1 that is shown in Figure 6.1a can be computed as $A^{app}_{app_1} = A^s_{server_1} * A^s_{server_2} * A^s_{server_8} = 0.97 * 0.95 * 0.98 = 0.9$ that is greater than or equal to the required availability of app_1 by ASP. Availability of app_1 can be improved if vm_2 is hosted on $server_6$ instead of $server_2$ as depicted in Figure 6.1b, where $A^{app}_{app_1} = A^s_{server_1} * A^s_{server_6} * A^s_{server_8} = 0.97 * 0.99 * 0.98 = 0.94$. The availability can be further improved if a standby VM vm_{2_2} is added for vm_{2_1}, and hosted on $server_3$ as shown in Figure 6.1c. The availability is computed as $A^{app}_{app_1} = A^s_{server_1} * (1 - (Fail^s_{server_6} * Fail^s_{server_3})) * A^s_{server_8} = 0.97 * (1 - (0.01 * 0.05)) * 0.98 = 0.95$. Now if a CPU scaling up request for vm_1 with 2 additional units is demanded, in this case $server_1$ will not be able to host vm_1 with the new 4 CPU units demand because $server_1$ has only 3 CPU units capacity. So vm_1 has to be migrated to other server with at least 4 CPU units capacity, and without violating availability requirement of app_1. vm_1 can be migrated to $server_5$ as shown in Figure 6.1d, where $A^{app}_{app_1} = 0.96 * (1 - (0.01 * 0.05)) * 0.98 = 0.94$.

To illustrate the application admissibility problem, let us assume another ASP requesting a new application app_2 that includes three VMs vm_4, vm_5, and vm_6, with availability requirement of 0.88. If CSP adopts a placement policy that hosts VMs at the servers that has maximum availability to increase the application availability, app_2 will be deployed in DC as depicted in Figure 6.1e. So, $A^{app}_{app_2} = A^s_{server_6} * A^s_{server_7} * A^s_{server_4} = 0.99 * 0.99 * 0.99 = 0.97$ that satisfies the requirement. Note that for app_2, CSP provides availability greater than and far from the required availability by ASP. Let assume the same or different ASP requests a new application app_3 that includes three VMs vm_7, vm_8, and vm_9 with availability requirement equals to 0.97. According to the current situation of DC that is shown in Figure 6.1e, the request for app_3 is rejected, because app_3 can not be admitted and deployed at DC since it violates the availability requirement. As a result, CSP will lose the profit of hosting app_3. However, CSP could admit app_3 and all other applications app_1 and app_2 at DC, if CSP adopts the policy to provide application availability that is close to the requested one. As shown in Figure 6.1f, the three applications are admitted and deployed in the DC while they satisfy their availability requirements, where availability of app_1, app_2, and app_3 are equal to 0.94, 0.88, and 0.97 respectively.

$$A^{app}_a = \prod_{f=1}^{|F_a|} A^{func}_f \tag{6.2}$$

$$A^{func}_f = 1 - \prod_{v=1}^{|V^{func}_f|} Fail^{vm}_v \tag{6.3}$$

$$Fail^s_j = 1 - A^s_j \tag{6.4}$$

6.2 Dynamic VM Placement for Service Availability in Cloud Framework

We propose a framework named 'Multiple-Objectives Dynamic VM Placement for Application Availability in Cloud' (MoVPAAC). The framework generates a placement plan for a set of VMs that belong to a set of requested applications at the servers in the DC of the cloud. The goal of the placement is to achieve multiple objectives and satisfy the availability requirement of each application as requested by ASP. In addition, the framework has the ability to dynamically change the placement of VMs in cases of application scaling or failure. Figure 6.2 depicts the proposed MoVPAAC framework. It includes three main modules: Availability-Aware Application Deployment, Proactive Application Failure Detection, and Dynamic Application Reconfiguration. We discuss the modules in detail separately in the next subsections.

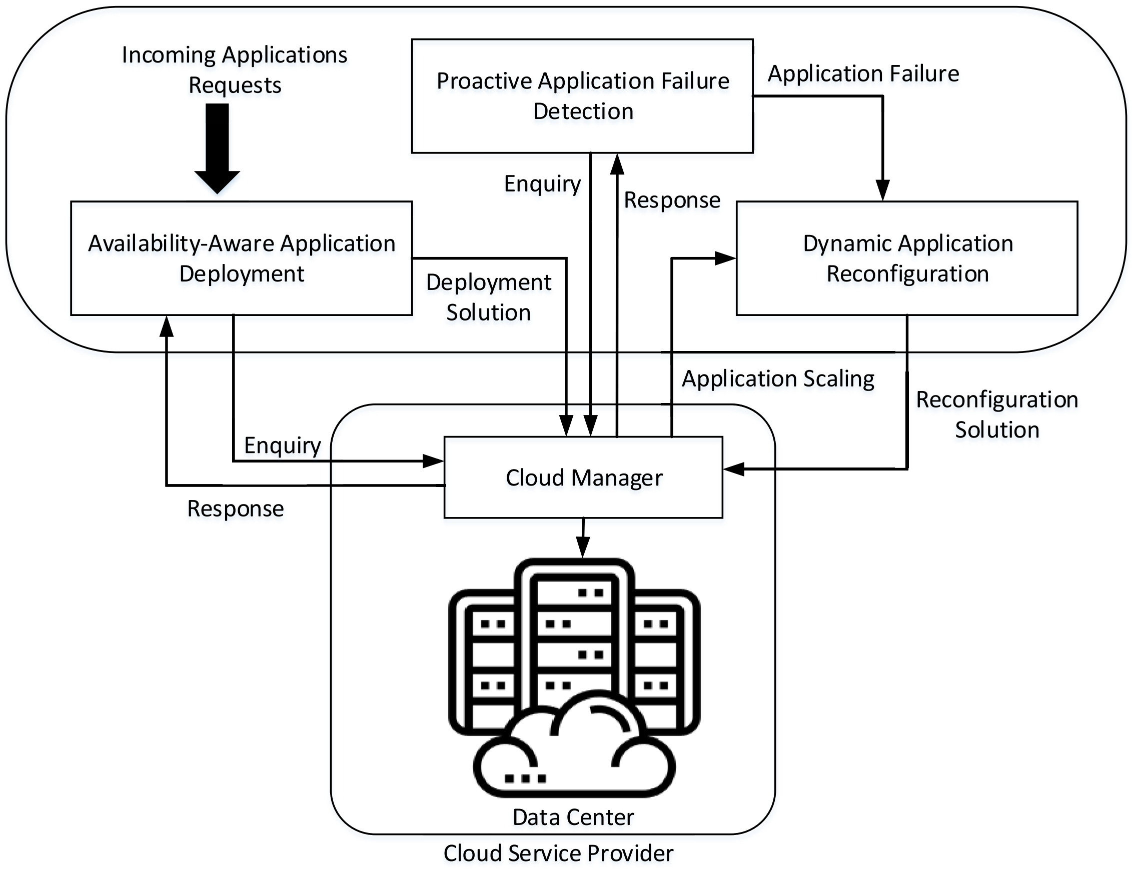


Figure 6.2: Multiple-Objectives Dynamic VM Placement for Application Availability in Cloud (MoVPAAC) Framework.

6.2.1 Availability-Aware Application Deployment

Availability-Aware Application Deployment module is mainly responsible for generating VMs placement plan for deployment of applications at the underlying servers located at DC according to the objectives and requirements. The applications include a set of VMs. The module returns the VMs placement plan for the cloud manager to deploy the requested applications. VM placement includes contradictory objectives and is considered NP-hard problem. The module handles VM dynamic problem that is described as follows. Given a set of applications with their requirements including availability as they are requested by ASPs, where each application includes a set of VMs, and each VM provides the functionality of a specific type towards providing the end-to-end application service. The goal is to find the placement plan for VMs at the servers that are located at DC targeting the three objectives, minimize power consumption, resources wastage, and failures ratio of servers, and seek to maintain availability requirements of the applications during their entire execution times. We formalize the problem as INLP optimization model with multiple objectives and a set of constraints. In addition, we propose a local heuristic approach based on Ant-Colony optimization method in conjunction with the VM standby protection approach to find a solution for the model and deploy all the requested applications.

The problem statement can be formalized as follows. Given a set of applications A that are requested by a set of ASPs, where each application $app_a \in A$ has availability requirement $A^{app}_{req_a}$ as requested by the corresponding ASP. Each app_a includes a set of VMs V^{app}_a. Each VM $vm_i \in V^{app}_a$ has a resource demand of different types such as CPU, RAM, and Disk. On the other side, DC includes a set of servers S, where each server $s_j \in S$ has a resource capacity of different types to host VMs. The goal is to place each vm_i at one s_j in such a way that achieves the three objectives, and $A^{app}_a \geq A^{app}_{req_a}, \forall app_a \in A$.

The first objective is to minimize the total power consumption of the active servers that are used to host VMs that belong to A. To compute the power consumption of server s_j in DC, we adopt the linear relationship between server power consumption and its CPU utilization as described in [83]. We define the average power consumption of s_j that is denoted as P_j in Equation (6.5), where P^{active}_j and P^{idle}_j are the average power consumption values when s_j is active and idle respectively, and U^c_j is the CPU utilization of s_j where $U^c_j \in [0,1]$. The first objective is formalized in Equation (6.6), where V^A is the set that includes all the VMs that compose all the requested applications located in A, x_{ij} is binary decision variable where value 1 indicates that vm_i is placed on server s_j and a value 0 indicates otherwise as defined in Equation (6.10), $R^{vm}_{i,c}$ is the CPU resource demand by vm_i.

$$P_j = (P^{active}_j - P^{idle}_j) \times U^c_j + P^{idle}_j \tag{6.5}$$

$$Minimize \sum_{j=1}^{|S|} P_j = \sum_{j=1}^{|S|} (((P^{active}_j - P^{idle}_j) \times \sum_{i=1}^{|V^A|} (R^{vm}_{i,c} \times x_{i,j}) + P^{idle}_j)) \tag{6.6}$$

The second objective is to minimize the total resources wastage of active servers. We define the cost of wasting resources of server s_j that is denoted as W_j in Equation (6.7), where L^c_j, L^r_j, and L^d_j represent the normalized remaining CPU, RAM, and Disk resources of server s_j respectively. β is a very small value that we set as 0.00001. U^c, U^r, and U^d represent the normalized CPU, RAM, and Disk resource usage respectively of server s_j. The second objective is formalized in Equation (6.8), where $T^s_{j,c}$, $T^s_{j,r}$, and $T^s_{j,d}$ represent the upper thresholds of CPU, RAM, and Disk utilization of server s_j respectively. We define an upper threshold for each resource type which is the same for all the servers in DC to avoid any server from reaching to a full usage state that may impact the performance of the server. $R^{vm}_{i,r}$ and $R^{vm}_{i,d}$ are the RAM and Disk resource demand by vm_i respectively.

The third objective is to minimize the total failures ratio of the servers at DC. We compute failure of s_j that is $Fail^s_j$ as the complement for availability of s_j as in Equation (6.4), where A^s_j is computed as in Equation (6.1). The third objective is formalized in Equation (6.9).

$$W_j = \frac{||L^c_j - L^r_j| - L^d_j| + \beta}{U^c + U^r + U^d} \tag{6.7}$$

$$\begin{aligned} Minimize \sum_{j=1}^{|S|} W_j = \sum_{j=1}^{|S|} (((||(T^s_{j,c} - \sum_{i=1}^{|V^A|} (R^{vm}_{i,c} \times x_{i,j})) - (T^s_{j,r} - \sum_{i=1}^{|V^A|} (R^{vm}_{i,r} \times x_{i,j}))| \\ - (T^s_{j,d} - \sum_{i=1}^{|V^A|} (R^{vm}_{i,d} \times x_{i,j}))| + \beta)/(\sum_{i=1}^{|V^A|} (R^{vm}_{i,c} \times x_{i,j}) + \\ \sum_{i=1}^{|V^A|} (R^{vm}_{i,r} \times x_{i,j}) + \sum_{i=1}^{|V^A|} (R^{vm}_{i,d} \times x_{i,j})) \end{aligned} \tag{6.8}$$

$$Minimize \sum_{j=1}^{|S|} Fail^s_j = \sum_{j=1}^{|S|} \sum_{i=1}^{|V^A|} Fail^s_j \times x_{ij} \tag{6.9}$$

We define a set of constraints to control the placement of VMs. Server s_j can be in only one state either active or idle. Binary decision variable x_{ij} where value 1 means vm_i is placed at server s_j, and a value 0 otherwise as defined in Equation (6.10). Any vm_i is hosted on maximum one server as defined in Equation (6.11). Any server s_j should have enough resources, CPU, RAM, and Disk to be able to host any vm_i as defined in Equations (6.12 - 6.14). VMs that belong to the same application app_a should not be collocated on the same server. We define such anti-affinity constraint in Equation (6.15), to force placement of VMs that provide and protect end-to-end application service on different servers to increase the availability of the service [72]. Availability of any requested application should be greater than or equal to the required availability that is requested by ASP as defined in Equation (6.16). *Subject to*:

$$x_{ij} = \begin{cases} 1, & \textit{if } vm_i \textit{ is placed on } s_j \\ 0, & \textit{otherwise} \end{cases} \tag{6.10}$$

$$\sum_{j=1}^{|S|} x_{ij} \leq 1 \quad \forall i \in I \tag{6.11}$$

$$\sum_{i=1}^{|V^A|} (R^{vm}_{i,c} \times x_{ij}) \leq T^{s}_{j,c} \quad \forall j \in J \tag{6.12}$$

$$\sum_{i=1}^{|V^A|} (R^{vm}_{i,r} \times x_{ij}) \leq T^{s}_{j,r} \quad \forall j \in J \tag{6.13}$$

$$\sum_{i=1}^{|V^A|} (R^{vm}_{i,d} \times x_{ij}) \leq T^{s}_{j,d} \quad \forall j \in J \tag{6.14}$$

$$x_{ij} + x_{zj} < 1 \quad \forall vm_i, vm_z \in V^{app}_a, \forall s_j \in S \tag{6.15}$$

$$A^A_{app_a} \geq A^A_{req_{app_a}} \quad \forall app_a \in A \tag{6.16}$$

We propose a local heuristic algorithm named Availability-Aware Applications Deployment (AvAAD) to solve INLP model and find the placement of VMs that compose the requested applications. The AvAAD algorithm uses AntColony optimization algorithm to solve multiple objectives VMs placement, in conjunction with the proposed standby protection approach to satisfy the availability requirements of the applications. Algorithm 4 describes AvAAD, it takes the list of the requested applications *appsList* including their requirements, available servers list *serversList* at DC as input. The algorithm returns the list of non admitted (rejected) applications *nonAdmittedAppsList* as output. In the beginning the algorithm initializes variables $vmsList$, $paretoSet$, $violatedAvAppsList$, and $nonAdmittedAppsList$ as empty. For each $vm \in V_a^{app}$, the algorithm adds the vm to $vmsList$. Then it calls MOAntColony (Algorithm 5) and passes the arguments $vmsList$ and $serversList$ to it. MOAntColony returns back $paretoSet$ that includes placement of $vmsList$ at $serversList$. Using $paretoSet$, the algorithm computes availability A_a^{app} of each $app_a \in appsList$. It adds each app_a that violates its availability requirement ($A_a^{app} < A_{req_a}^{A}$) to $violatedAvAppsList$. For each $app_a \in violatedAvAppsList$, the algorithm tries to enhance its availability by adding new VM standby to protect the functionality that has minimum availability among functionalities $\in app_a$. It keeps adding standby VMs one at a time until either ($A_a^{app} \geq A_{req_a}^{app}$), or count of added standby VMs reaches the maximum number of standby VMs that can be added for app_a. Any new added standby VM is placed on the server $s_j \in serversList$ that has maximum value of $\frac{1}{P_j+W_j+Fail_j^s}$. This is to keep the consistency with the three objectives of MOAntColony algorithm. Once AvAAD handles all violated applications, it checks again if there are applications that still violate their availability requirements. If so, it considers applications as rejected that can not be admitted in DC and adds violated app_a to $nonAdmittedAppsList$ that returns at the end of the algorithm.

For the time complexity analysis of Algorithm 4, in lines (2-5) it takes $\mathcal{O}(n)$ execution time to extract VMs that belong to requested applications. At line (7), it calls MOAntColony Algorithm 5 to find placement plan of $vmsList$ at servers located in $serversList$. Note that the performance of Algorithm 4 depends mainly on the performance of MOAntColony. AntColony is a meta heuristic algorithm that takes a polynomial execution time $\mathcal{O}(n^k)$ to find the optimization solution [37]. In the context of VM placement problem, k value depends mainly on number of iterations, ants, VMs and servers that AntColony uses to find the placement solution. In lines (8-13), the algorithm takes $\mathcal{O}(n)$ execution time to determine the applications that violate their availability requirements. At lines (14-21), it takes $\mathcal{O}(n^2)$ execution time to satisfy the availability for each application that violates its required availability. At lines (22-26), the algorithm takes $\mathcal{O}(n)$ execution time to determine the rejected applications that can not be admitted at DC since they violate their availability requirements. So in total, Algorithm 4 takes $\mathcal{O}(n) + \mathcal{O}(n^k) + \mathcal{O}(n) + \mathcal{O}(n^2) + \mathcal{O}(n)$ which can be simplified to $\mathcal{O}(n^k)$ execution time.

Algorithm 4 Availability-Aware Application Deployment

Input: $appsList$, $serversList$

Output: $nonAdmittedAppsList$

```
1: initialize $vmsList$, $paretoSet$, $violatedAvAppsList$, $nonAdmittedAppsList$ as $empty$
2: for each $app_a \in appsList$ do
3:     for each $vm \in V_a^{app}$ do
4:         Add $vm$ to $vmsList$
5:     end for
6: end for
7: $paretoSet = MOAntColony(vmsList, serversList)$
8: for each $app_a \in appsList$ do
9:     $A_a^{app} \leftarrow computeAvailability(app_a)$ // using Equation (6.2)
10:    if $A_a^{app} < A_{req_a}^{app}$ then
11:        Add $app_a$ to $violatedAvAppsList$
12:    end if
13: end for
14: for each $app_a \in violatedAvAppsList$ do
15:    $vmsAddedStandby = 0$
16:    while (($A_a^{app} < A_{req_a}^{app}$) and ($vmsAddedStandby <$ app.getMaxAddVmStandby())) do
17:        Add standby vm to the first functionality of min availability $\in app_a$
18:        place standby $vm$ at $s_j$ with max $\frac{1}{P_j + W_j + Fail_j^s}$
19:        $vmsAddedStandby += 1$
20:    end while
21: end for
22: for each $app_a \in violatedAvAppsList$ do
23:    if $A_a^{app} < A_{req_a}^{app}$ then
24:        Add $app_a$ to $nonAdmittedAppsList$
25:    end if
26: end for
27: return $nonAdmittedAppsList$
```

In order to achieve the objectives of applications deployment, we propose a local heuristic algorithm named Multiple Objectives Ant Colony (MOAntColony) that uses Ant Colony Optimization (ACO) algorithm to find the placement of VMs that belong to the requested applications. MOAntColony algorithm is described in Algorithm 5. In the beginning, the algorithm initializes the parameters and all the pheromones trials. In each iterative step, each ant z receives *vmsList* that compose *appsList* that are requested to be deployed. Ant z picks a server *currServer* and starts the placement process of *vmsList* at *currServer* using pseudo-random-proportional rule [84]. The rule defines the desirability of an ant to select a specific *vm* as the next one to place at *currServer*. It depends on the pheromone concentration level and the heuristic information that can guide ant z to select *vm*. The local pheromone concentration level is updated once ant z makes a movement (placement) step. Ant z keeps moving until it finishes placement of *vmsList* and builds its solution. Once all ants finish and build their own solutions, a global pheromone is updated based on the pareto set *PS* that includes best-located solutions. The algorithm initializes the pheromone level τ_0 as defined in Equation (6.17), where n is the total number of VMs that require placement, $P^{'}(S_0)$ is the normalized power consumption of the servers listed in the initial placement solution S_0 that is generated by FirstFit VM placement algorithm, $W^{'}(S_0)$ and $Fail'(S_0)$ are the resource wastage and failures of servers listed in S_0 respectively. $P^{'}(S_0)$ is defined in Equation (6.18), where P_j^{max} is the maximum power consumption of the server j. $W^{'}(S_0)$ and $Fail'(S_0)$ are defined in Equations (6.19, 6.20) respectively. The heuristic information $\eta_{i,j}$ indicates the desirability of an ant z for placement of vm_i at server s_j. The desirability $\eta_{i,j}$ considers the partial contribution for each objective. Every ant z starts with *vmsList* to place them at a set of available servers that are arranged randomly in the list L. Ant z starts placement of *vmsList* on servers $\in L$ that are selected sequentially one by one. As a result, the sequence of servers from 1 to j is known at the placement step of vm_i at s_j. So the partial contribution of the current placement step for the first, second and third objectives are defined in Equations (6.21, 6.22, 6.23) respectively. We combine the three partial contributions together for the heuristic placement decision as defined in Equation (6.24).

$$\tau_0 = \frac{1}{n \times (P'(S_0) + W'(S_0) + Fail'(S_0))} \tag{6.17}$$

$$P^{'}(S_0) = \sum_{j=1}^{M} (P_j / P_j^{max}) \tag{6.18}$$

$$W^{'}(S_0) = \sum_{j=1}^{M} (W_j) \tag{6.19}$$

$$Fail^{'}(S_0) = \sum_{j=1}^{M} (Fail_j) \tag{6.20}$$

$$\eta_{i,j,1} = \frac{1}{\beta + \sum_{k=1}^{j} (P_k / P_k^{max})} \tag{6.21}$$

$$\eta_{i,j,2} = \frac{1}{\beta + \sum_{k=1}^{j} W_k} \tag{6.22}$$

$$\eta_{i,j,3} = \frac{1}{\beta + \sum_{k=1}^{j} Fail_k} \tag{6.23}$$

$$\eta_{i,j} = \eta_{i,j,1} + \eta_{i,j,2} + \eta_{i,j,3} \tag{6.24}$$

Ant z selects vm_i as the next VM to be placed at the current server s_j based on pseudo-random-proportional rule as defined in Equation (6.25) [84], where α is a parameter to control pheromone trail importance, q is a random number $\in [0,1]$, q_0 is a fixed value $0 < q_0 < 1$. If q is less than or equal to q_0, this is a case for exploitation, otherwise it is for exploration as defined in Equation (6.25). U is the set of VMs that can be hosted on s_j. $\eta_{u,j}$ is the pheromone value as defined in Equation (6.24), $\tau_{u,j}$ is the local pheromone update as defined in (6.27). Pr is random-proportional rule probability distribution as defined in Equation (6.26) [84]. There are two steps to update the pheromone, local and global. When ant z assigns vm_i at s_j, it does local pheromone update as defined in Equation (6.27), where τ_0 is the initial pheromone level, and $0 < \rho_l < 1$ is the local pheromone evaporating parameter, and t is the current iteration. The global pheromone update is done according to the rule as defined in Equation (6.28), where $0 < \rho_g < 1$ is the global pheromone evaporation parameter. λ is a coefficient that is defined in Equation (6.29), where Z is the number of the ants, T^S is the number of the iterations where the global solution Sol is located in pareto set PS. $P^{'}(Sol)$, $W^{'}(Sol)$, and $Fail^{'}(Sol)$ are the normalized power consumption, resource wastage, and failures respectively of servers that are listed in the solution Sol. As we discussed previously, Algorithm 5 uses mainly AntColony meta heuristic optimization algorithm that takes $\mathcal{O}(n^k)$ execution time [84]. The value of k depends on number of iterations T, ants Z, VMs in *vmsList*, and servers in *serversList* that are used by AntColony algorithm to find the placement plan for the VMs.

$$i = \begin{cases} max_{u \in U}\{\alpha \times \tau_{u,j} + (1-\alpha) \times \eta_{u,j}\}, & q \leq q_0 \\ Pr, & otherwise \end{cases} \tag{6.25}$$

$$Pr_{u,j} = \begin{cases} \frac{\alpha \times \tau_{u,j} + (1-\alpha) \times \eta_{u,j}}{\sum_{u=1}^{|U|} (\alpha \times \tau_{u,j} + (1-\alpha) \times \eta_{u,j})}, & u \in U \\ 0, & otherwise \end{cases} \tag{6.26}$$

$$\tau_{i,j}(t) = (1-\rho_l) \times \tau_{i,j}(t-1) + \rho_l \times \tau_0 \tag{6.27}$$

$$\tau_{i,j}(t) = (1-\rho_g) \times \tau_{i,j}(t-1) + \frac{\rho_g \times \lambda}{P'(Sol) + W'(Sol) + Fail'(Sol)} \tag{6.28}$$

$$\lambda = \frac{Z}{t - T^{Sol} + 1} \tag{6.29}$$

Algorithm 5 MOAntColony

```
Input: vmsList, serversList
Output: Pareto Set PS
1: Initialize values of parameters $\tau_0$, $\alpha$, $q_0$,$\rho_l$, $\rho_g$, $Z$, and $T$
2: Initialize PS as empty
3: Initialize all pheromone values to $\tau_0$
4: for t=1 to T do
5:     for z=1 to Z do
6:         sort serversList in random order
7:         while not all $vms \in vmsList$ are placed do
8:             currServer=select a new server from serversList
9:             while there is a vm can be placed at currServer do
10:                for each remaining vm can be placed at currServer do
11:                    Calculate desirability For vm as in Equation (6.25)
12:                    Calculate probability For vm as in Equation (6.26)
13:                end for
14:                Select vm to assign
15:                Generate q randomly
16:                if $q \leq q_0$ then
17:                    exploitation as in Equation (6.25)
18:                else
19:                    exploration as in Equation (6.25)
20:                    Update local pheromone as in Equation (6.27)
21:                end if
22:            end while
23:        end while
24:    end for
25:    Calculate the three objectives For each solution Sol generated by each ant z
26:    if S is non-dominated by any other solution then
27:        Add Sol to PS
28:        Remove solutions $\in PS$ that are dominated by Sol
29:    end if
30:    for each $Sol \in PS$ do
31:        Update global pheromone as in Equation (6.28)
32:    end for
33: end for
34: return PS
```

6.2.2 Proactive Application Failure Detection

The main goal of the proactive application failure detection module is to detect application failure at an early stage of its provisioning before the failure actually happens. Failure of application leads to a service outage that can impact QoS, and SLA violation. Detection of application failure at an early stage helps to take the appropriate service recovery action as fast as possible. The module uses the same prediction approach that is used by the task failure prediction module of the framework that is discussed in Section 4.3 of Chapter 4 to predict failure of a set of given tasks during the runtime.

6.2.3 Dynamic Application Reconfiguration

This module receives a reconfiguration request for a set of provisioned applications for which the availability requirements are threatened to be violated. The request can come either from the proactive application failure module as a notification for applications that are predicted as failed, or from the cloud manager as a notification for scaling requests. For the request from the proactive module, the dynamic application reconfiguration module adds a new VM that provides the same task that is predicted as failed to replace the existing VM. This is applied for all the tasks that are predicted as failed. Since the newly added VMs will be hosted on the servers at DC, the reconfiguration module takes care of the placement process. We propose a placement process for the newly added VMs to recover the application services that are predicted as failed. The process targets the three objectives as defined in Equations (6.6, 6.8, 6.9), and respect the constraints that are defined in Equations (6.10-6.16), to be consistent with the objectives of the proposed framework MoVPAAC. The placement procedure for the added VMs to recover applications services is described in Algorithm 6. The algorithm takes the list of application tasks that are predicted as failed $failPredTasksList$, and the list of servers at DC *serversList* as input. It returns the map that includes the placement of newly added VMs to provision tasks as output. For each $task \in failPredTasksList$, the algorithm adds a new *vm* to provide $task$. For each added *vm*, the algorithm searches for one *server* $\in$ *serversList* that can host *vm*, has minimum summation value of power consumption, resources waste, and failure, and without violating any of the constraints that are defined in Equations (6.10-6.16). Once the algorithm finds the *server*, it adds the record $< vm, server >$ to the map $vmsPlacementMap$. In the end, the algorithm returns the map *vmsPlacementMap*. For the time complexity of Algorithm 6, it takes $\mathcal{O}(n^2)$ because for each added *vm*, the algorithm searches for the best server among *serversList* that can host *vm* with minimum cost.

Algorithm 6 VM Placement For Application Recovery

```
Input: $failPredTasksList$, $serversList$
Output: $vmsPlacementMap$
1:  for each $task \in failPredTasksList$ do
2:      Add $vm$ provides $task$
3:      $minCost = \infty$
4:      $minServer = empty$
5:      for each $server \in serversList$ do
6:          if $server$ can host $vm$ and satisfy constraints in Equations (6.10 - 6.16) then
7:              $serverCost = P_{server} + W_{server} + Fail^{s}_{server}$
8:              if $serverCost < minCost$ then
9:                  $minCost = serverCost$
10:                 $minServer = server$
11:             end if
12:         end if
13:     end for
14:     Add $< vm, minSerer >$ to $vmsPlacementMap$
15: end for
16: return $vmsPlacementMap$
```

For the scaling request from the cloud manager at CSP, the request can be one of the four scaling types (directions): scaling out, scaling up, scaling in, or scaling down. Scale out includes a request to add a set of new VMs, while the scale up is to add resources such as virtual CPU (vCPU), and RAM (vRAM) to an existing set of VMs. Scale in includes a request to remove a set of existing VMs, while the scale down is to remove virtual resources from an existing set of VMs. For application scaling out request, the reconfiguration module handles placement of the newly added VMs the same way it handles the request from the proactive application failure module. For the scaling up, in some cases, some servers may not have enough resources to continue hosting the VMs after they are updated with the new resources. In such a case, the VMs have to be migrated to other servers that can accommodate them without violating any of the constraints that are defined in Equations (6.10-6.16). The migration process has to be done carefully because it can have a big influence on the outage period of the application service. The problem can be summarized as follows, one VM can be migrated to different servers, without violating constraints, with different migration times. There are different VMs that belong to different applications and need to be migrated. The goal is to migrate all the VMs with minimum migration time, hence reducing the outage time of the applications. We formalize the problem as INLP model with the objective to minimize the migration time of the VMs that need to be migrated as defined in Equation (6.30), that obey the constraints as defined in Equations (6.10-6.16), where G is the set of VMs that need to be migrated, S is the set of the available servers at DC, $migrationTime_{n,j,d}$ is the time to migrate vm_n from source server s_j to the destination server s_d. The binary decision variable x_{nj} with a value 1 means vm_n is hosted on server s_j and 0 otherwise, z_{nd} is another binary decision variable that is defined in Equation (6.31), with value 1 means vm_n need to be migrated to the server d, and 0 otherwise. We propose a local heuristic approach, that is described in Algorithm 7, to solve the INLP model and find the placement servers of the VMs that demand scaling. The algorithm takes the set of VMs that require scaling $vmsScaleList$, available servers at DC $serversList$, and the scaling type $scaleType$ as input. It returns a map that includes the placement of the VMs that demand scaling at the servers in DC. The algorithm checks type of the scaling request. If the scaling type is out, that means a new set of VMs needs to be added. For each added *vm*, the algorithm searches for a server that can host *vm*, has minimum summation value of power consumption, resource waste, and failure, and meet all the constraints that are defined in Equations (6.10-6.16). Once the algorithm finds the placement server *minServer* to host *vm*, it adds the record $< vm, minServer >$ to the map $vmsPlacementMap$. If the scaling type is up, the algorithm determines the VMs that need to be migrated from their hosted servers. For each *vm* that needs to be migrated, the algorithm finds a destination server that contributes to minimize the total migration time as defined in Equation (6.30). If the algorithm finds the destination server $minDestServer$ to host *vm*, it adds the record $< vm, minDestServer >$ to the map $vmsPlacementMap$. For both scaling types in and down, for any $vm \in vmsScaleList$, the algorithm rejects any scaling action for *vm* that violates the application availability requirement constraint that is defined in Equation (6.16), otherwise, it allows the action to be applied by the cloud manager.

For the time complexity analysis of Algorithm 7, at lines (1-15) it takes $\mathscr{O}(n^2)$ execution time because for each scale out *vm*, the algorithm searches for the best server with minimum cost that can host *vm*. For the scaling up request at lines (16-33), the algorithm takes $\mathscr{O}(n^2)$ execution time because for each *vm* that requires to be migrated, the algorithm searches for the server that can host the *vm* with minimum migration time. For the scaling request of type in or down, the algorithm takes $\mathcal{O}(n)$ execution time to allow or reject the scaling action. In total the algorithm takes $\mathcal{O}(n^2)+\mathcal{O}(n^2)+\mathcal{O}(n)+\mathcal{O}(n)$ that can be simplified to $\mathcal{O}(n^2)$ execution time.

$$Minimize \sum_{j=1}^{|S|} \sum_{d=1|d\neq j}^{|S|} \sum_{n=1}^{|G|} (migrationTime_{n,j,d} \times x_{nj} \times z_{nd}) \tag{6.30}$$

$$z_{nd} = \begin{cases} 1, & \textit{if } vm_n \textit{ requires to be migrated to } s_j \\ 0, & \textit{otherwise} \end{cases} \tag{6.31}$$

Subject to constraints that are defined in Equations (6.10-6.16)

6.3 Experiments and Results

6.3.1 Experimental Setup

We conducted several types of experiments to evaluate the proposed MoVPAAC framework including its modules and used algorithms in this research work. We built a simulation to simulate the elements including their properties and requirements, that we need as a proof of concept for this research work, such as DC, servers, VMs, and applications. The simulation is implemented using C++ programming language. All the experiments are carried out on 64-bit Windows 10 machine equipped with an Intel Core i7-8665U 2.11GHz processor and 16 GB of RAM.

To evaluate the availability-aware application deployment module, we divided the experiments into two groups. The first group includes a set of application deployment requests including their requirements as requested by a set of ASPs, where the applications do not include any standby (backup) VMs. Note that the first request (number 1) includes five applications with different structures and availability requirements. We set the availability requirement of application number 5 with a value equal to 0.99999 (aka five nines) to show the ability of our approach to meet such availability value demands by ASPs. We select this structure type for the applications to evaluate the ability of the deployment module to add standby protection plan to satisfy the availability requirements of the applications. In addition, to check the impact of adding extra standby resources on the performance of the servers at DC. We used four requests, where each request includes a different number of applications. For the experiments of the first group, we simulated one DC that includes 85

Algorithm 7 VM Placement For Application Scaling

```
Input: $vmsScaleList$, $serversList$, $scaleType$
Output: $vmsPlacementMap$
 1: if $scaleType = \text{'}out\text{'}$ then
 2:     for each $vm \in vmsScaleList$ do
 3:         $minCost = \infty$
 4:         $minServer = empty$
 5:         for each $server \in serversList$ do
 6:             if $server$ can host $vm$ and satisfy constraints in Equations (6.10 - 6.16) then
 7:                 $serverCost = P_{server} + W_{server} + Fail_{server}$
 8:                 if $serverCost < minCost$ then
 9:                     $minCost = serverCost$
10:                     $minServer = server$
11:                 end if
12:             end if
13:         end for
14:         Add $< vm, minSerer >$ to $vmsPlacementMap$
15:     end for
16: else if $scaleType = \text{'}up\text{'}$ then
17:     for each $vm \in vmsScaleList$ do
18:         $currServer = vm.hetHostedServer()$
19:         if $currServer$ can not host $vm$ then
20:             $minMigrationCost = \infty$
21:             $minDestServer = empty$
22:             for each $server \in serversList$ do
23:                 if $server$ can host $vm$ and satisfy constraints in Equations (6.10 - 6.16)
    then
24:                     $serverMigrationCost = getVmMigrTime(vm, currServer, server)$
25:                     if $serverMigrationCost < minMigrationCost$ then
26:                         $minMigrationCost$
27:                         $minDestServer = server$
28:                     end if
29:                 end if
30:             end for
31:             Add $<vm, minDestServer>$ to $vmsPlacementMap$
32:         end if
33:     end for
34: else if $scaleType = \text{'}in\text{'}$ OR $scaleType = \text{'}down\text{'}$ then
35:     for each $vm \in vmsScaleList$ do
36:         if scale action For $vm$ violates the availability constraint in Equation (6.16) then
37:             reject scale action For $vm$
38:         end if
39:     end for
40: end if
41: return $vmsPlacementMap$
```

servers. We assign heterogeneous resource properties to the servers. The CPU and RAM capacities of the servers are generated randomly based on a uniform distribution with values ranging between 8-15 units. The availability levels of the servers are also generated randomly based on a uniform distribution with values ranging between 0.7-0.97. We set P^{active} and P^{idle} with the values 215, and 162 respectively which are the same for all the servers. Table 6.1 describes the structure, number of VMs, availability requirement for all the applications of the four requests in the first group of the experiments. The CPU and RAM demands of the VMs are also generated randomly based on a uniform distribution with values ranging between 2-5 units. We submitted each request in Table 6.1 separately to the availability-aware application deployment module to deploy the applications and return the placement plan of the VMs that compose the applications. For MOAntColony algorithm, we set the number of iterations $T = 10$ and the number of ants $Z = 12$ for placement of VMs. We compare the placement results that are generated by AvAAD Algorithm 6 with the results of two other VM placement algorithms from the literature that are CHASE [58] and FirstFit.

6.3.2 Results

To evaluate the ability of the deployment module to deploy the requested applications while satisfying their availability requirements, we computed the availability of all the applications after their deployments are done by the three placement algorithms, AvAAD, CHASE, and FirstFit, and compared them with the requested availability of the applications by ASPs. Figure 6.3 shows the availability comparison of the five applications of request 1 after they are deployed in DC by each of the three placement algorithms. As we can see AvAAD algorithm could deploy all the applications and meet their availability requirements. CHASE algorithm violated the availability requirements for 4 applications out of 5, and FirstFit algorithm violated availability for all the applications. This is because AvAAD algorithm uses VM standby protection approach to satisfy the availability requirements of the applications. Even though CHASE tries to maximize the availability of the application by selecting the servers with maximum availability values, still it can violate the availability because it does not target to meet the application availability requirement that is requested by ASP. FirstFit violates availability requirements because it depends on the selection of the first server that can host VM that requires a placement. So if the availability of the first selected server is low, the end-to-end application availability will be low as well. Figure 6.4

Table 6.1: Description of Applications Requests - Group 1

Request 1			
	Req Availability	**Funct Number**	**VMs Number**
Application 1	0.97	3	3
Application 2	0.88	4	4
Application 3	0.94	5	5
Application 4	0.95	6	6
Application 5	0.99999	3	3
Request 2			
	Req Availability	**Funct Number**	**VMs Number**
Application 1	0.95	3	3
Application 2	0.93	3	3
Application 3	0.98	2	2
Application 4	0.8	2	2
Application 5	0.82	2	2
Request 3			
	Req Availability	**Funct Number**	**VMs Number**
Application 1	0.97	4	4
Application 2	0.98	4	4
Application 3	0.96	4	4
Application 4	0.95	5	5
Application 5	0.98	4	5
Application 6	0.96	6	6
Request 4			
	Req Availability	**Funct Number**	**VMs Number**
Application 1	0.85	3	3
Application 2	0.87	4	4
Application 3	0.83	3	3
Application 4	0.8	4	4
Application 5	0.99	5	5
Application 6	0.96	5	5
Application 7	0.98	6	6

shows the mean availability of the applications per each request separately. AvAAD algorithm achieved mean availability close to the required mean availability of the applications, while CHASE and FirstFit achieved mean availability far from the required one. Figure 6.5 shows the admissibility of the applications per each request. AvAAD contributed to admit all the applications of all the requests at DC because it meets their requirements including the availability. Both CHASE and FirstFit contributed to reject many of the applications at DC because they violate mainly their availability requirements.

To evaluate the placement algorithms on the performance of the servers at DC, we computed the mean power consumption of the servers that host VMs that compose the applications per each request. As shown in Figure 6.6, AvAAD consumes power consumption higher than both CHASE and FirstFit. This is because AvAAD adds extra standby VMs to

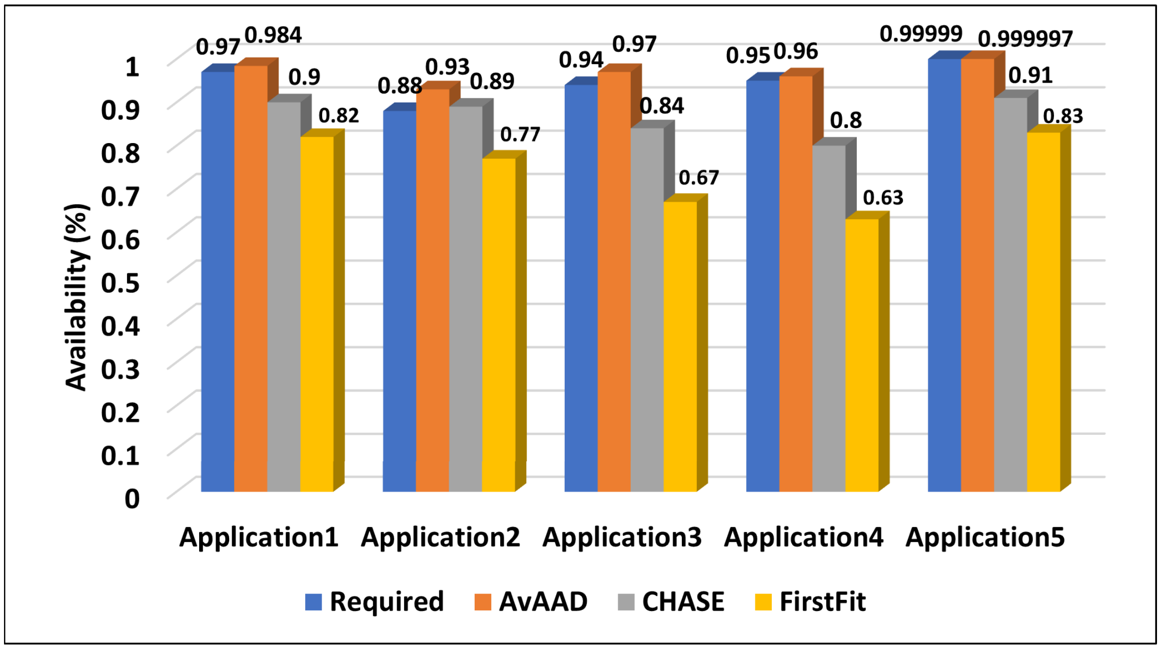


Figure 6.3: Applications Availability - Request 1

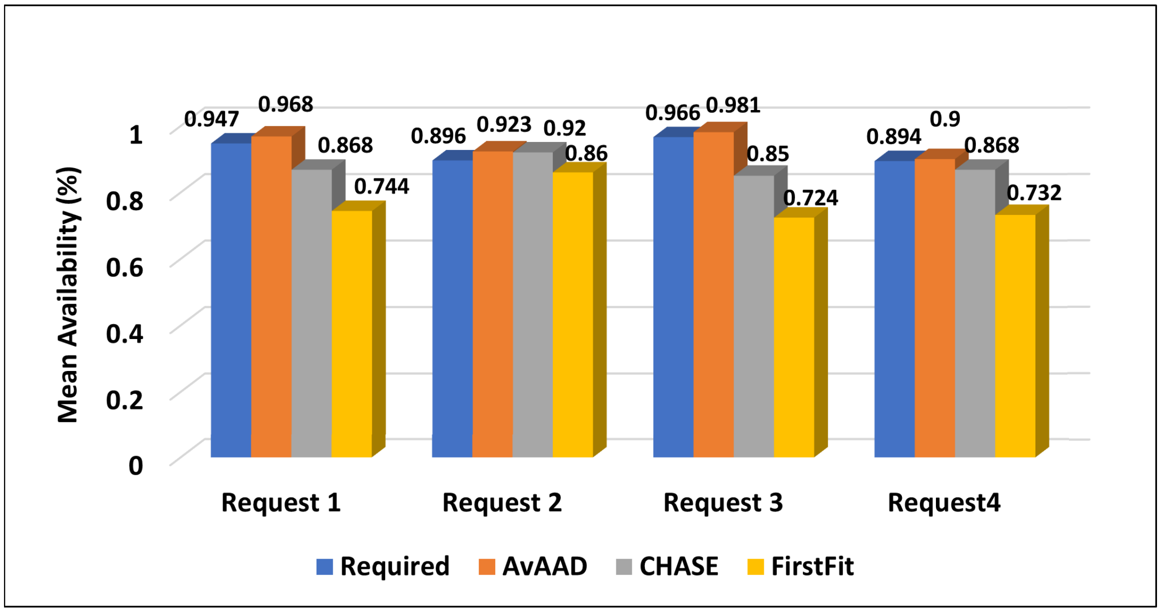


Figure 6.4: Mean Applications Availability - Group 1

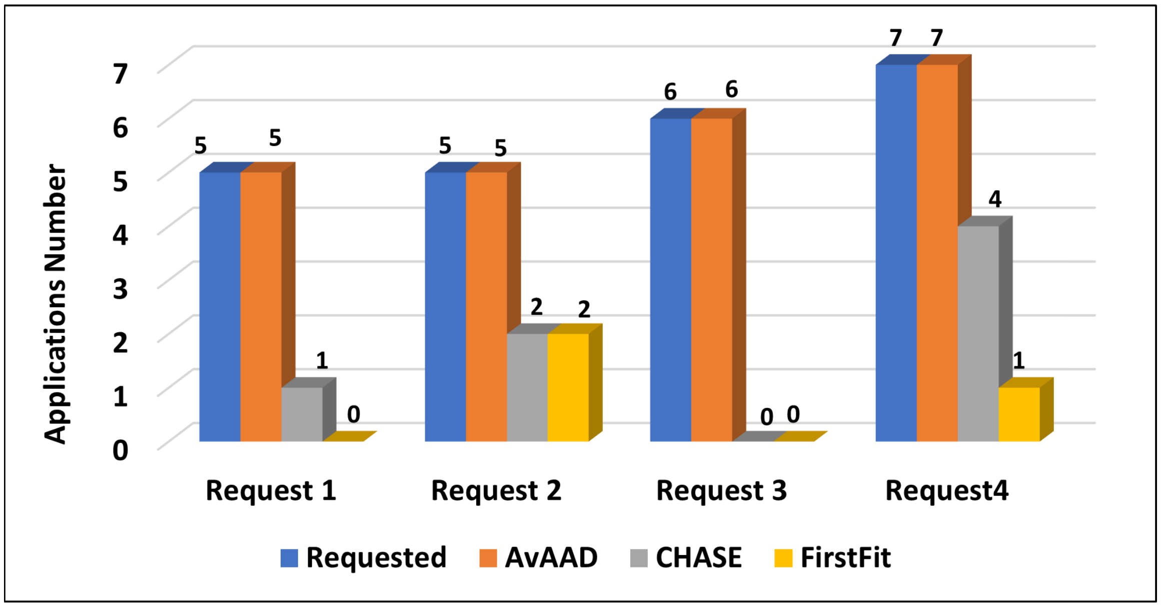


Figure 6.5: Applications Admissibility- Group 1

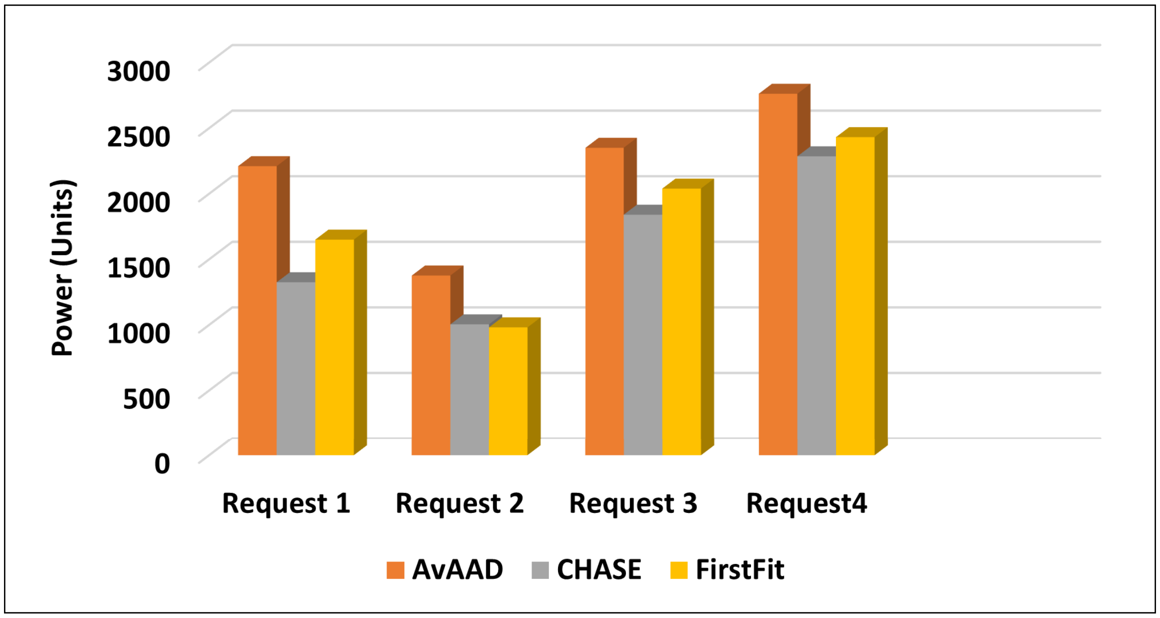


Figure 6.6: Servers Power Consumption - Group 1

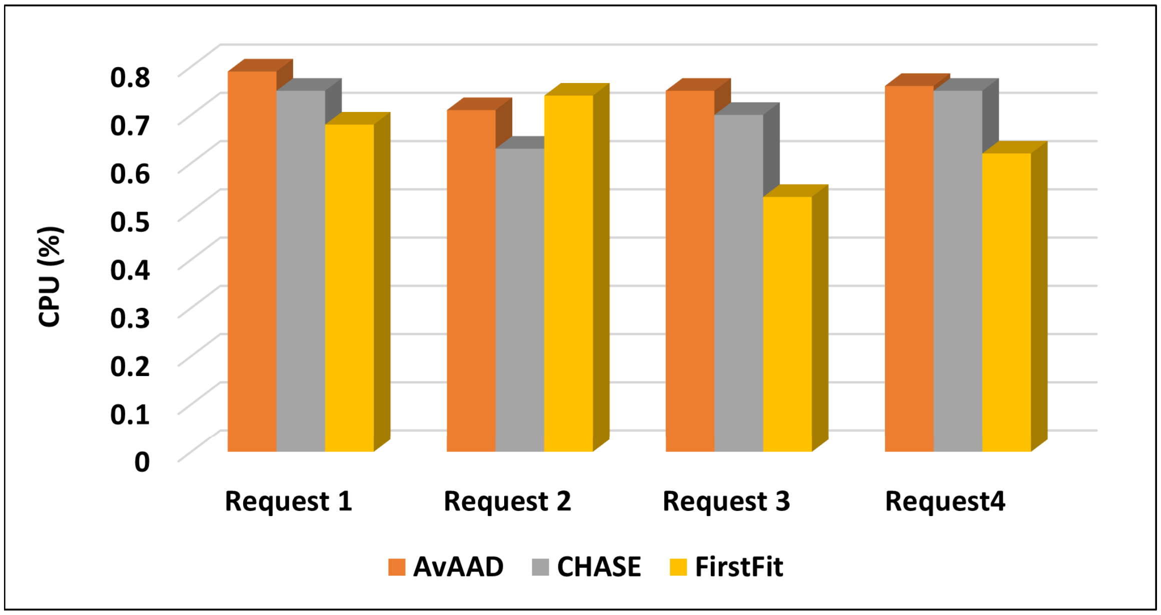


Figure 6.7: Servers CPU Utilization

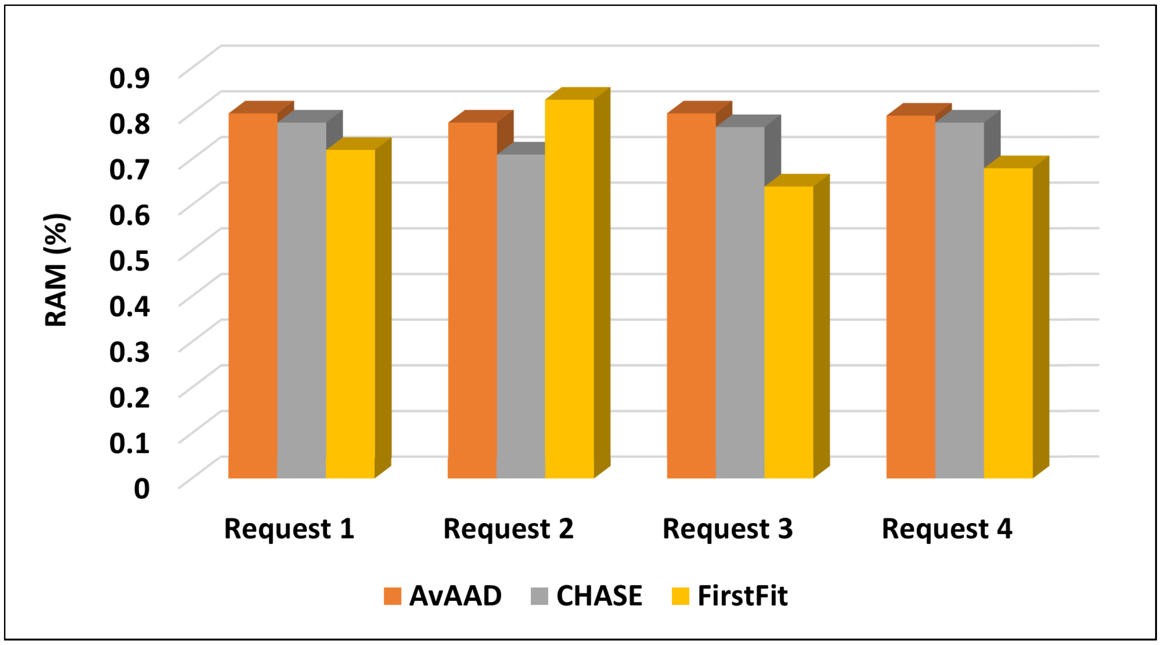


Figure 6.8: Servers RAM Utilization

satisfy the availability requirements for only the applications that violate their availability. As a result, extra standby VMs consume additional power consumption. We computed the mean of CPU and RAM utilization of the servers after deployment of the applications per each request. Figures 6.7 and 6.8 show CPU and RAM utilization of the servers respectively. AvAAD achieves stable and high CPU and RAM utilization because one of its objectives is to minimize wastefulness of the resources. Note that the CPU and RAM utilization by AvAAD does not cross the ratio 80% as the case for CHASE and FirstFit algorithms for some requests. This is because we set an upper threshold that is equal to 80% for both CPU and RAM utilization, to avoid any server from reaching a full state of VMs which can impact the performance of the server.

Table 6.2: Description of Applications Requests - Group 2

Request 5			
	Req Availability	**Funct Number**	**VMs Number**
Application 1	0.9	2	4
Application 2	0.88	2	4
Application 3	0.93	2	4
Application 4	0.87	2	4
Application 5	0.85	2	4
Request 6			
	Req Availability	**Funct Number**	**VMs Number**
Application 1	0.92	3	6
Application 2	0.96	3	6
Application 3	0.94	3	6
Application 4	0.9	3	6
Application 5	0.93	3	6
Application 6	0.88	3	6
Request 7			
	Req Availability	**Funct Number**	**VMs Number**
Application 1	0.95	3	9
Application 2	0.93	3	9
Application 3	0.94	3	9
Application 4	0.89	3	9
Application 5	0.93	3	9
Application 6	0.94	3	9
Request 8			
	Req Availability	**Funct Number**	**VMs Number**
Application 1	0.9	4	8
Application 2	0.94	4	8
Application 3	0.91	4	8
Application 4	0.95	4	8
Application 5	0.85	4	8
Application 6	0.9	4	8
Application 7	0.94	4	8

In the second group of the experiments, the applications include standby VMs to recover the application service in case of failure of the active VM(s). We do this type of the experiments to evaluate the performance of AvAAD algorithm without using the standby protection plan to meet the availability requirements. Table 6.2 describes the structure of the applications that belong to the second group of the experiments. Note that we kept the same properties of VMs and servers that are used in the first group of the experiments, except we change the availability values of the servers that are generated randomly based on a uniform random distribution with the new range between 0.6-0.9, for illustrative purposes. Figure 6.9 shows the availability that is achieved by each placement algorithm for the six applications that belong to request number 6. AvAAD algorithm can satisfy availability requirements for applications without adding standby VMs, that helps to reduce the overall power consumption of the servers at DC as shown in Figure 6.12. CHASE algorithm satisfied availability for most of the applications since there are standby VMs. However, still CHASE can violate availability for the applications because it does not prioritize applications based on their availability requirements. In other words, using CHASE, application with low availability requirement can be placed on servers with high availability values and these servers get full. This can lead to deploy application with high availability requirement on servers with low availability values, and hence reduces the overall availability of the application and violate its requirement. FirstFit violates availability requirements for most of the requested applications because it does not target availability requirement during the server selection process to host VMs. Figure 6.10 shows the mean availability of the applications per each request. AvAAD algorithm achieved mean availability greater than and closer to the required one, while CHASE achieved mean availability greater than and far from the required one. FirstFit achieved mean availability less than the required one. Although the applications of all the requests in the second group of the experiments include redundant VMs as shown in Table 6.2, it is not necessarily meeting the HA (aka five nines) requirements of the requested application. In such case, our AvAAD algorithm still needs to add extra standby VM(s) to meet the HA, while CHASE and FirstFit do not do that, hence they fail to meet HA. Note that, in case of existing redundant VMs, AvAAD may not need to add large number of the standby VMs to achieve HA. Figure 6.11 shows the admissibility of the applications per each request, where AvAAD admitted all the requested applications without adding extra standby VMs. Using standby VMs, CHASE could admit most of the applications, FirstFit rejected many of the applications due to its servers selection strategy. As shown in Figure 6.12, using AvAAD the servers consumed less power than using CHASE and FirstFit. This is because AvAAD did not need to add extra standby VMs for the protection approach. In addition, one of AvAAD's objectives is to minimize the power consumption of the active servers.

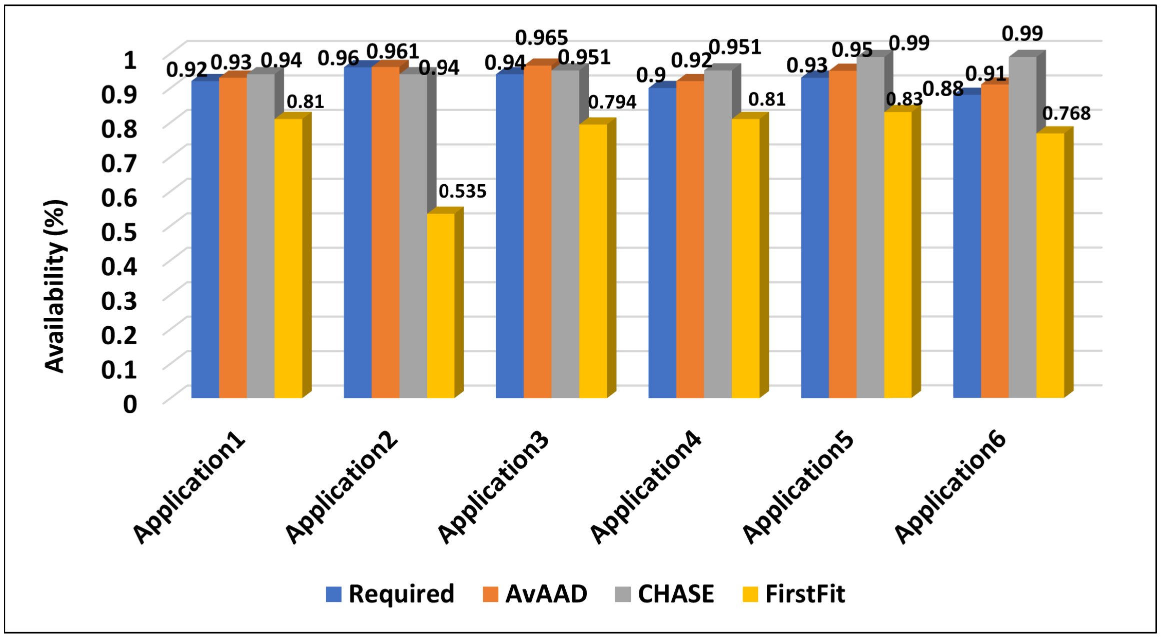


Figure 6.9: Applications Availability - Request 6

To compare the performance of the three VM placement algorithms, we computed the average execution time that each algorithm takes to place different sets of VMs with numbers ranging between 30 and 54 VMs. Figures 6.13, 6.14, and 6.15 show the average execution time of AvAAD, CHASE and FirstFit algorithms respectively. AvAAD takes less and reasonable time compared to CHASE, for example, AvADD took around 4.2 seconds to find servers for placement of 54 VMs, while CHASE took around 380 seconds for placement of the same VMs. FirstFit is the fastest algorithm that took less than a second for placement of the same 54 VMs. We believe this is because AvAAD uses an optimization MOAntColony algorithm that optimizes the servers search process for placement of the VMs. CHASE uses CPLEX to solve the placement problem, whereas CPLEX uses some pruning techniques such as branch-and-bound algorithm to solve VM placement combinatorial problem, still it takes a long execution time that grows with a large number of VMs. FirstFit just searches for the first server that can host the current VM that reducing the search time dramatically. Note that the execution of AvAAD depends on the number

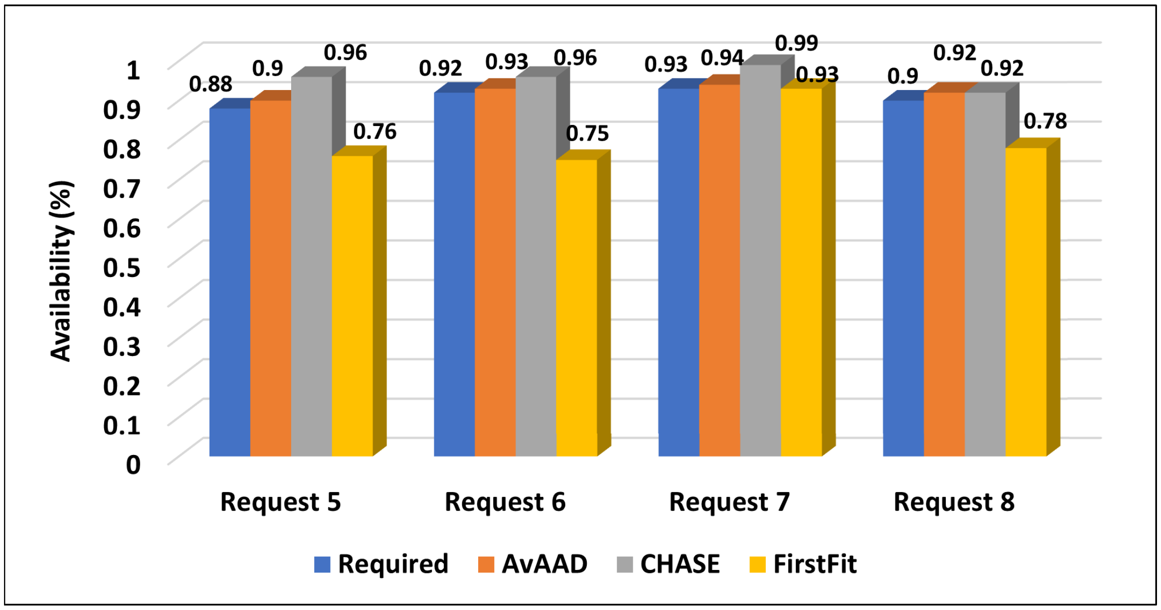


Figure 6.10: Mean Applications Availability - Group 2

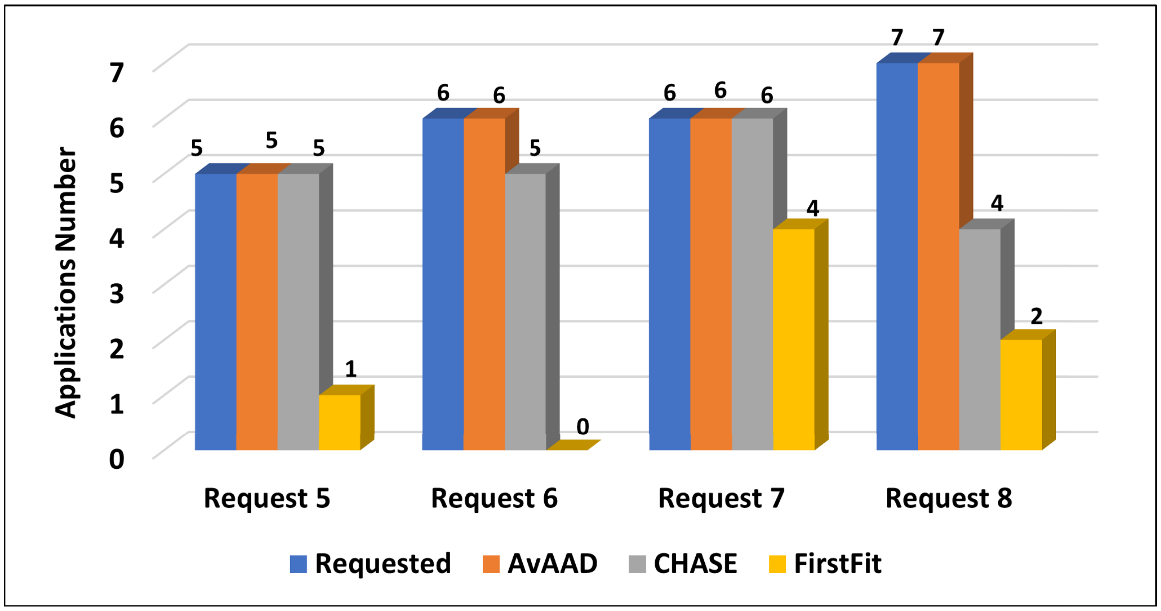


Figure 6.11: Applications Admissibility - Group 2

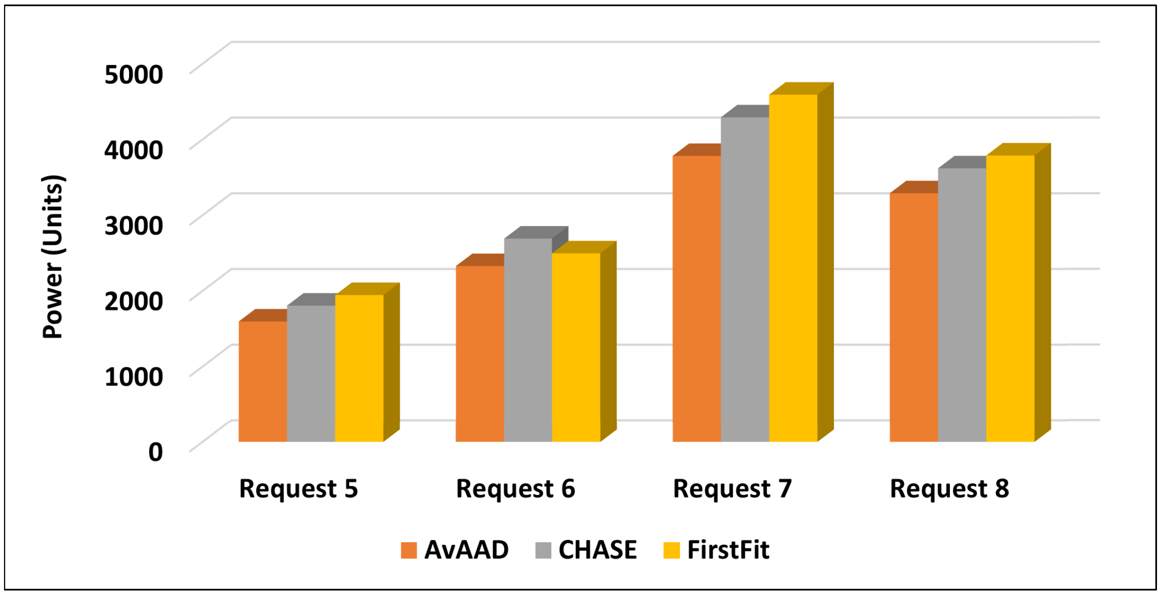


Figure 6.12: Servers Power Consumption - Group 2

of iterations T as well as the number of ants Z that are used by MOAntColony algorithm. So we measured the execution time of AvAAD for placement of 42 VMs using different number of ants and iterations. As shown in Figure 6.16, the execution time of AvAAD algorithm increases by increasing the number of ant or number of iterations. To test the execution time of AvAAD for placement of large number of VMs, we computed its average execution time for placement of different sets of VMs ranging between 60 and 300 VMs. As shown in Figure 6.17, AvAAD takes reasonable execution time to host a large number of VMs, where it increases by increasing the number of the VMs.

To evaluate the availability-aware VMs migration (AvAVMmigration) procedure that is used by the dynamic reconfiguration application module, we computed the migration time that AvAVMmigration takes to migrate different sets of VMs with different numbers that range between 2 and 10 VMs. We compare the migration time by AvAVMmigration with the time that is taken by the procedure, named VMmigrationMaxAvServer, that migrates VMs to the servers with the maximum availability and can host the migrated VMs. We generated a random migration time based on a uniform distribution between servers in the range between 60 and 200 seconds. Figure 6.18 shows the comparison of the VMs migration time by the two migration procedures. The proposed AvAVMmigration procedure takes less time to migrate VMs between servers than the procedure VMmigrationMax-AvServer. This is because AvAVMmigration searches for the destination servers for which

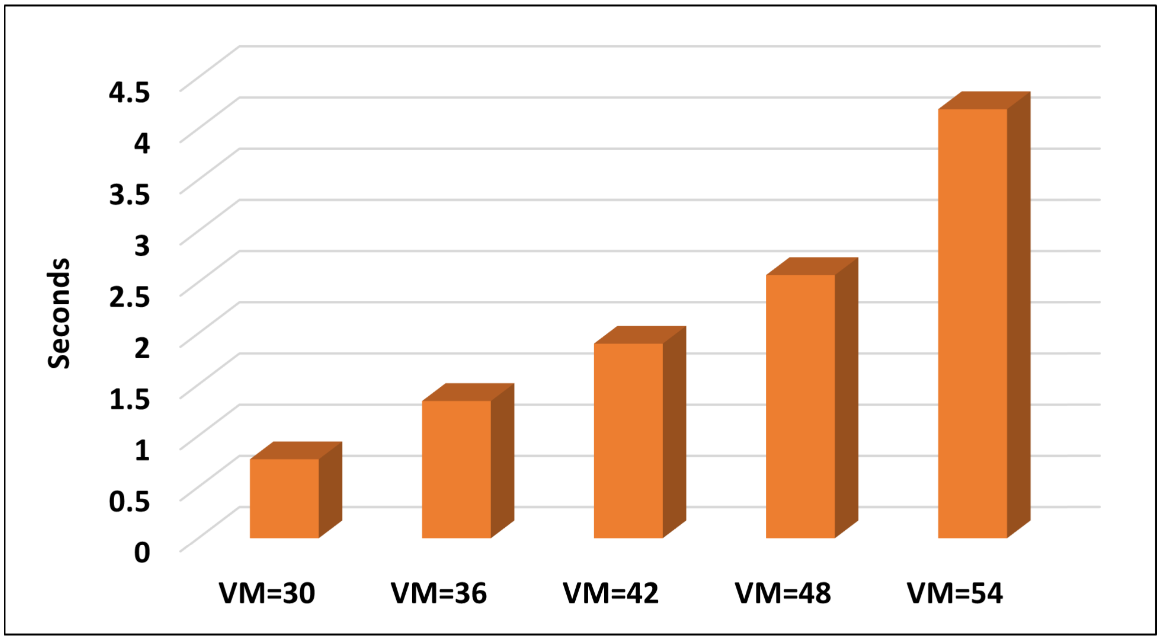


Figure 6.13: AvAAD Average Execution Time

the migration time is shorter to migrate the VMs without violating constraints, instead of migrating the VMs to the servers with high availability that may take longer time as shown in Figure 6.18.

6.4 Summary

In this chapter, we tackled the dynamic VM placement NP-hard problem from application availability perspectives. We formalized application availability in cloud and dynamic VM placement problem as INLP model with multiple objectives and set of constraints. In addition, a framework is proposed to handle VMs placement to deploy applications and maintain their availability according to the requirements of ASPs. The framework includes three main modules to handle VMs placement during deployment, failure, and scaling requests of the applications. The deployment module uses AntColony optimization algorithm, and VM standby protection approach to achieve multiple objectives and satisfy the availability requirements of the requested applications. The results show the ability of the proposed VM placement algorithm to admit a higher number of applications compared to CHASE and FirstFit VM placement algorithms from the literature. In addition, the proposed placement solution achieved less power consumption, and high CPU and RAM utilization of the

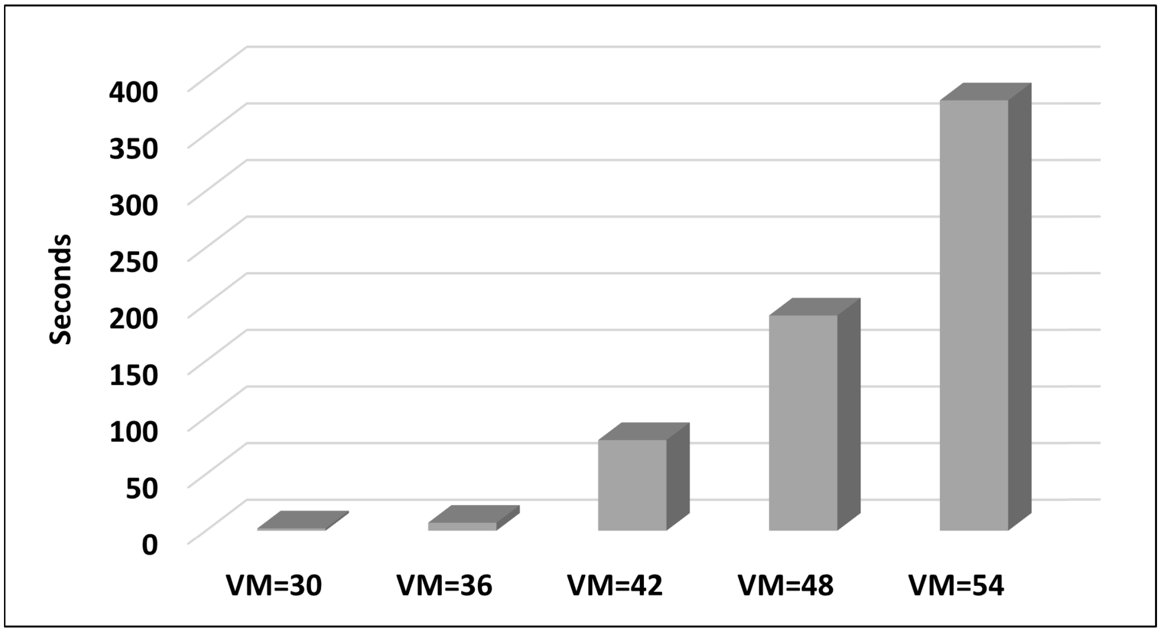


Figure 6.14: CHASE Average Execution Time

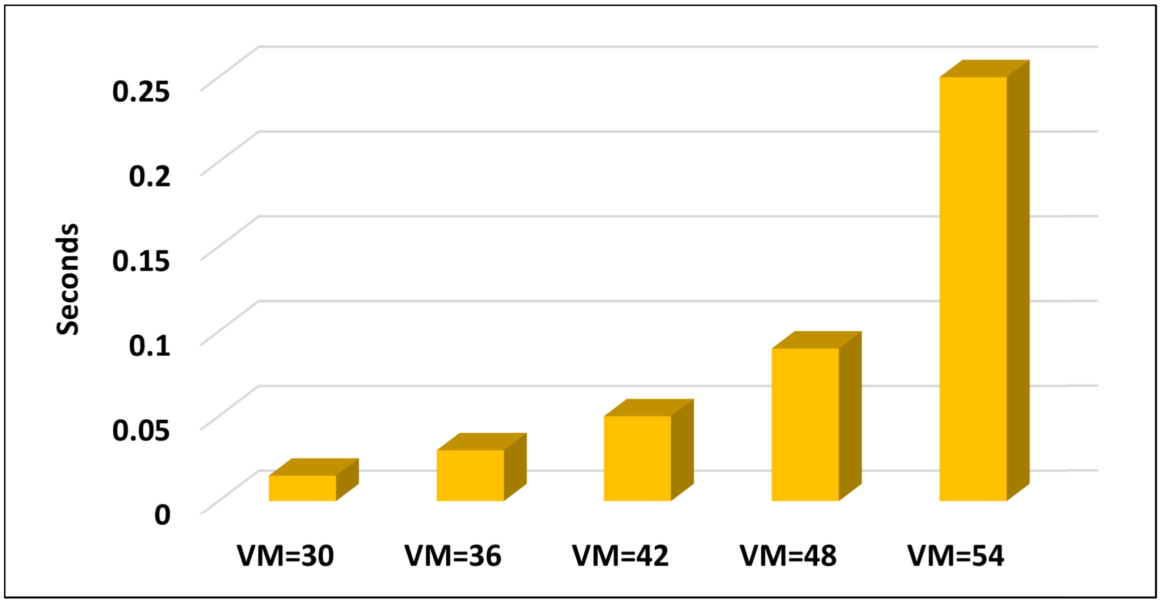


Figure 6.15: FirstFit Average Execution Time

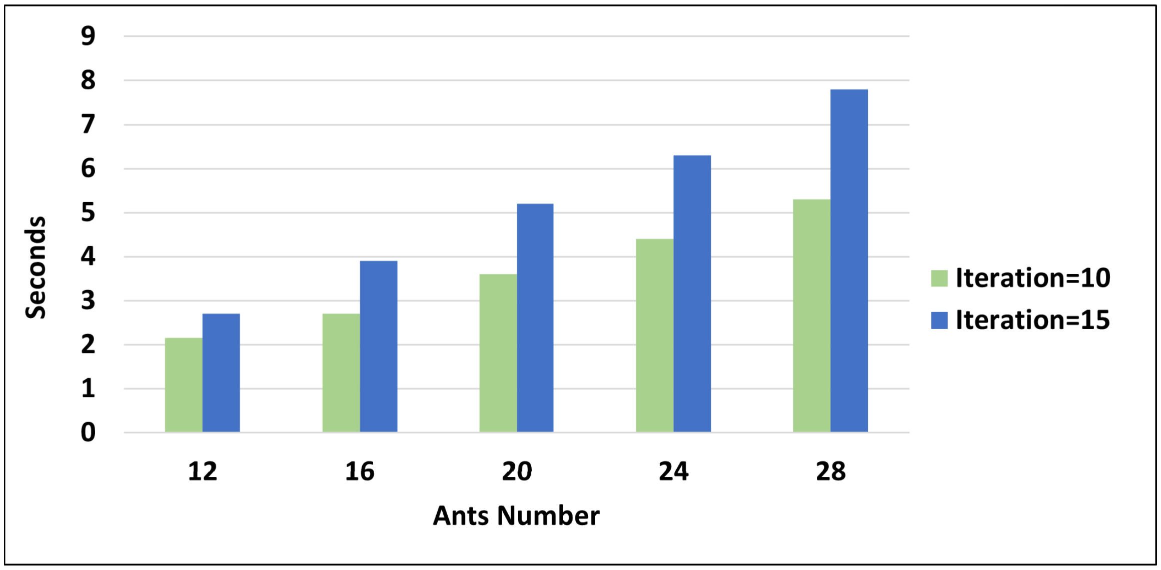


Figure 6.16: AvAAD Average Execution Time-Variable Ants and Iterations

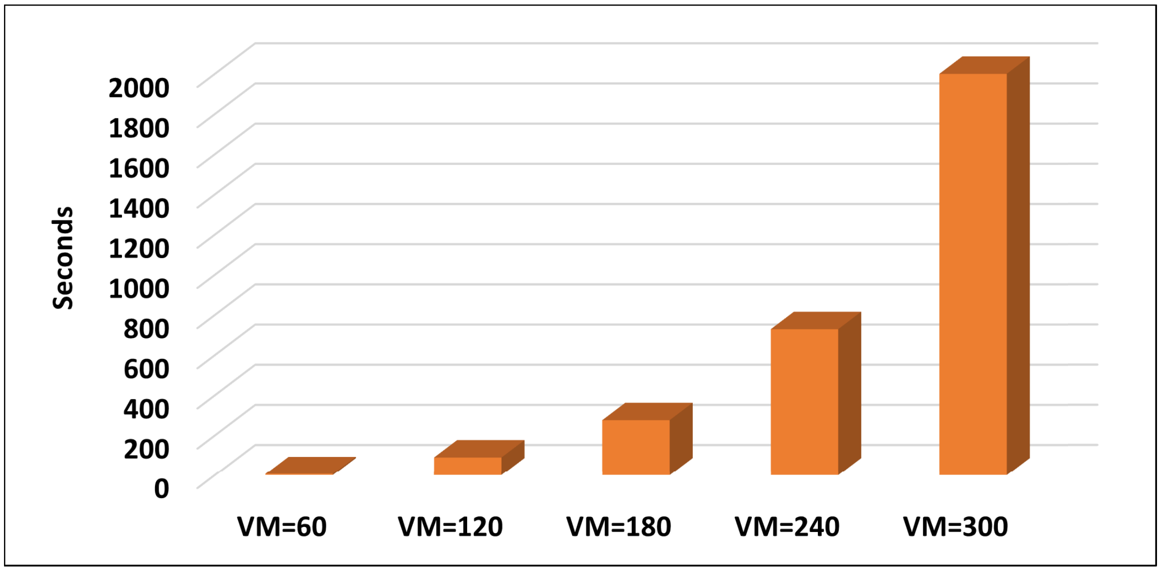


Figure 6.17: AvAAD Average Execution Time-Variable VMs

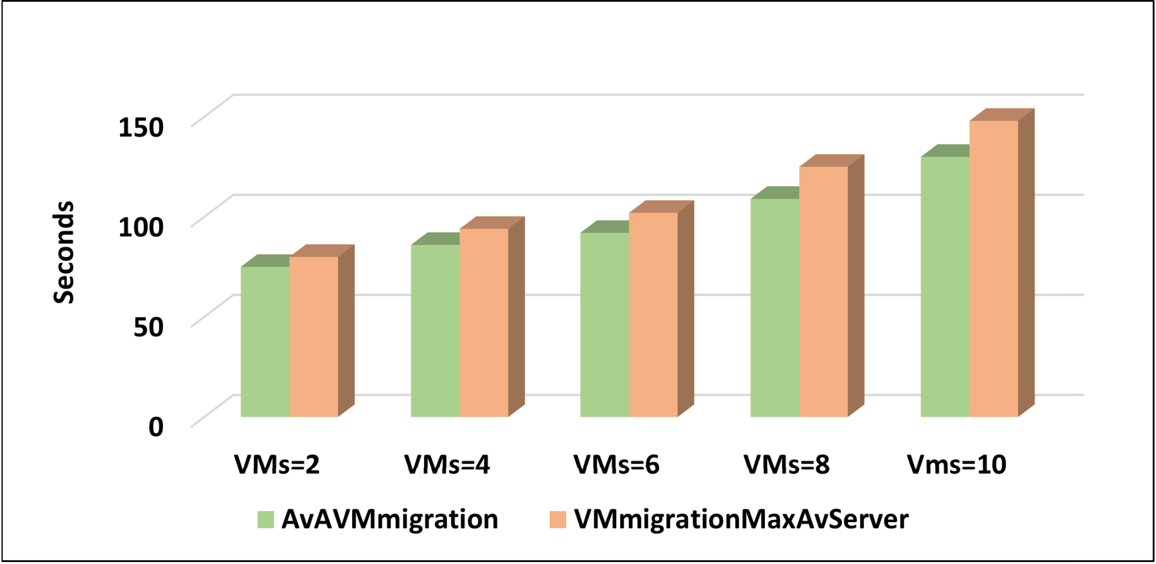


Figure 6.18: Evaluation of Availability-Aware VMs Migration Procedure

servers.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis tackled the main challenges that face application service availability in cloud computing. It addressed different issues and factors that are related to service availability management from different perspectives and actors such as ASP and CSP. Therefore, the thesis proposed different models, solutions, algorithms, and approaches that can help to improve service availability, fault tolerance, resources utilization, and cluster performance in the cloud environment.

The main contributions of this thesis can be summarized as follows. First, the thesis proposed a comprehensive study, analysis, and classification of the research works and solutions that exist in the literature. Second, a reactive framework is proposed to manage application service HA and enable its continuity in the virtual computing cluster. The framework can protect the service against failures of application component, VM, and physical computing server using redundancy models. Third, proactive service availability management is also proposed. The framework analyzed three public available cluster datasets from Google, Alibaba, and Trinity to identify the main features that are related to the failure of the application task (service). In addition, the framework uses ANN and CNN deep learning models to predict task failure during its execution time. Both ANN and CNN could predict task failure with high accuracy and low error. Fourth, the thesis mapped the task scheduling NP-hard problem as ILP model, to minimize the number of the failed tasks and their resources usage. In addition, a heuristic optimization solution is proposed to find the scheduling of tasks that are predicted as failed. The results show the ability of the scheduler to protect a large number of tasks and save their wasted resources. Finally, the thesis tackled the dynamic VM placement problem for service availability. It formalized the application availability in the cloud and the VM placement problem as INLP model. The model targets three main objectives to minimize power consumption, resources wastage, and failure of the active servers that are used to host the VMs. In addition, the model tries to provide application availability above and near the requirements by ASP. AntColony heuristic optimization algorithm in conjunction with VM standby protection approach is used to find VMs placement plan that achieves multiple objectives. Dynamic VM placement is resolved for different request types, deployment of a new application, VM scaling, and migration. The results show the ability of the proposed placement solution to meet the availability requirements and increase the admissibility of the requested applications.

7.2 Future Work

We believe there are still open research issues that are worth to be investigated to improve application service availability, and performance of the cluster in cloud computing. The proposed reactive service availability management framework depends on the system administrator to specify the role of the application components and their deployment positions at DCs of CSP. It would be a good idea to develop the framework in such a way that can automatically determine the role of the components based on the requirements of the tenant. In addition, the framework can automatically deploy the components based on their roles at DCs, and respect the constraints. The HA module of the framework depends mainly on the monitoring service of the used HA middleware to monitor the status of the component. This can force some limitations, where the period of component status checking depends on the ability of the middleware. In addition, the existing monitoring services monitor only the active component. However, for some applications monitoring the standby component status can help to reduce the service recovery time. It would be very beneficial to build a separate dynamic monitoring engine that can be integrated with the HA middleware. The monitoring engine can allow configuring the period of component liveness checking. In addition, the monitoring engine can be used to monitor the status of both active and standby components, and report that to the availability management service of the HA middleware to take recovery action. For the stateful application, the framework stores the state of the active component at a shared storage to be retrieved in case of failover. The frequent updates and retrievals of the states of the active components can impact the overall performance of the systems and QoS. Therefore, it can be helpful to investigate other mechanisms to find an efficient procedure to update and retrieve the status of the active component.

The task failure prediction module of the proactive framework assumes that the prediction process is triggered based on a request from the cluster manager. Actually, it would be more effective to have an automatic service that can trigger the prediction process for a set of tasks. The service can specify which tasks to predict their termination status, as well as the time to trigger the prediction process. In addition, the service can adapt the time of the prediction based on the performance of the computing cluster, and the status of the running tasks. It would be very helpful to develop a smart elasticity engine that can take the scaling actions based on the workload of the service, and coordinate with the service availability manager to avoid SLA violation. In addition, developing a resource manager that can consolidate the physical computing servers with reactive/proactive availability management solutions can help to reduce the overall cost of provisioning the service, as well as the power consumption of the servers at DCs. Moreover, other objectives such as minimizing the network bandwidth and delay of the provisioned application service can be targeted by the proposed availability-aware dynamic VM placement solution.

Bibliography

[1] NIST Definition of Cloud Computing. Accessed: September. 6, 2021. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf

[2] Virtualization Overview. Accessed: September. 6, 2021. [Online]. Available: https://www.vmware.com/pdf/virtualization.pdf.

[3] D. Siewiorek and J. Gray, "High-Availability Computer Systems" in Computer, vol. 24, no. 09, pp. 39-48, 1991.

[4] The Cost of Service Downtime. Accessed: September. 1, 2021. [Online]. Available: https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/.

[5] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in cloud computing: What it is, and what it is not," in Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13), 2013, pp. 23-27.

[6] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format + schema," *Technical report, Google Inc.* Mountain View, CA, USA, November 2011.

[7] VMware High Availability. Accessed: 15 September, 2021. [Online]. Available: https://www.vmware.com/pdf/ha_datasheet.pdf.

[8] VMware. Accessed: 15 September, 2021. [Online]. Available: https://www.vmware.com/.

[9] VMware vSphere 6 Fault Tolerance. 15 September, 2021. [Online]. https://learnvmware.online/wp-content/uploads/2018/02/vmware-vsphere6-ft-arch-perf.pdf.

[10] Ha-lizard - High Availability Solution for XenServer. Accessed: 15 September, 2021. [Online]. Available: https://www.halizard.com/.

[11] Citrix Hypervisor. Accessed: 15 September, 2021. [Online]. Available: https://www.citrix.com/products/citrix-hypervisor/.

[12] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: high availability via asynchronous virtual machine replication," In Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08). USENIX Association, USA, 2008, pp. 161–174.

[13] A. Kanso and Y. Lemieux, "Achieving High Availability at the Application Level in the Cloud," 2013 IEEE Sixth International Conference on Cloud Computing, 2013, pp. 778-785.

[14] Open Service Availability Middleware. Accessed: 15 September, 2021. Available: [Online]. https://opensaf.sourceforge.io/index.html.

[15] W. Li and A. Kanso, "Comparing Containers versus Virtual Machines for Achieving High Availability," 2015 IEEE International Conference on Cloud Engineering, 2015, pp. 353-358.

[16] W. Li, A. Kanso and A. Gherbi, "Leveraging Linux Containers to Achieve High Availability for Cloud Services," 2015 IEEE International Conference on Cloud Engineering, 2015, pp. 76-83.

[17] X. Chen, C. Lu and K. Pattabiraman, "Failure Prediction of Jobs in Compute Clouds: A Google Cluster Case Study," 2014 IEEE International Symposium on Software Reliability Engineering Workshops, 2014, pp. 341-346.

[18] M. Soualhia, F. Khomh, and S. Tahar,"Predicting Scheduling Failures in the Cloud: A Case Study with Google Clusters and Hadoop on Amazon EMR," 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015, pp. 58-65.

[19] M. S. Jassas and Q. H. Mahmoud, "Failure Characterization and Prediction of Scheduling Jobs in Google Cluster Traces," 2019 IEEE 10th GCC Conference & Exhibition (GCC), 2019, pp. 1-7.

[20] M. S. Jassas and Q. H. Mahmoud, "Evaluation of a failure prediction model for large scale cloud applications," Canadian Conference on Artificial Intelligence. Springer, 2020, pp. 321–327.

[21] Alibaba Cluster Trace. Accessed: Mar. 10, 2021. [Online]. Available: https://github.com/alibaba/clusterdata/tree/master/cluster-trace-v2017.

[22] Atlas Cluster Trace. Accessed: Mar. 15, 2021. [Online]. Available: https://ftp.pdl.cmu.edu/pub/datasets/ATLAS/.

[23] J. Shetty, R. Sajjan, and S. G, "Task Resource Usage Analysis and Failure Prediction in Cloud," 9th International Conference on Cloud Computing, Data Science Engineering (Confluence), 2019, pp. 342-348.

[24] C. Liu, L. Dai, Y. Lai, G. Lai, and W. Mao, "Failure prediction of tasks in the cloud at an earlier stage: A solution based on domain information mining," Computing, vol. 102, no. 9, pp. 2001-2023, Sep. 2020.

[25] J. Gao, H. Wang, and H. Shen, "Task Failure Prediction in Cloud Data Centers Using Deep Learning," IEEE Transactions on Services Computing, pp. 1–1, 2020, doi: 10.1109/TSC.2020.2993728.

[26] A. Rosà, L. Y. Chen and W. Binder, "Failure Analysis and Prediction for Big-Data Systems," IEEE Transactions on Services Computing, vol. 10, no. 6, pp. 984-998, 2017.

[27] T. Islam and D. Manivannan, "Predicting Application Failure in Cloud: A Machine Learning Approach," 2017 IEEE International Conference on Cognitive Computing (ICCC), 2017, pp. 24-31.

[28] C. Liu, J. Han, Y. Shang, C. Liu, B. Cheng and J. Chen, "Predicting of Job Failure in Compute Cloud Based on Online Extreme Learning Machine: A Comparative Study," IEEE Access, vol. 5, pp. 9359-9368, 2017.

[29] N. El-Sayed, H. Zhu and B. Schroeder, "Learning from Failure Across Multiple Clusters: A Trace-Driven Approach to Understanding, Predicting, and Mitigating Job Terminations," 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), 2017, pp. 1333-1344.

[30] T. Hongyan, L. Ying, W. Long, G. Jing and W. Zhonghai, "Predicting Misconfiguration-Induced Unsuccessful Executions of Jobs in Big Data System," 2017

IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), 2017, pp. 772-777.

[31] P. Padmakumari and A. Umamakeswari, “Task Failure Prediction using Combine Bagging Ensemble (CBE) Classification in Cloud Workflow,” Wirel. Pers. Commun. vol. 107, no. 1, pp. 23–40, 2019.

[32] A. Rosà, L. Y. Chen and W. Binder, “Predicting and Mitigating Jobs Failures in Big Data Clusters,” 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2015, pp. 221-230.

[33] T. Islam and D. Manivannan, “FaCS: Toward a Fault-Tolerant Cloud Scheduler Leveraging Long Short-Term Memory Network,” 2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/ 2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom), 2019, pp. 1-6.

[34] M. Soualhia, F. Khomh and S. Tahar, “A Dynamic and Failure-Aware Task Scheduling Framework for Hadoop,” IEEE Transactions on Cloud Computing, vol. 8, no. 2, pp. 553-569, 2020.

[35] A. Marahatta, Q. Xin, C. Chi, F. Zhang and Z. Liu, “PEFS: AI-driven Prediction based Energy-aware Fault-tolerant Scheduling Scheme for Cloud Data Center,” IEEE Transactions on Sustainable Computing, 2020.

[36] M. Soualhia, C. Fu, and F. Khomh, “Infrastructure Fault Detection and Prediction in Edge Cloud Environments,” 2019 4th ACM/IEEE Symposium on Edge Computing, 2019, pp. 222-235.

[37] A. Samir and C. Pahl, “Detecting and Predicting Anomalies for Edge Cluster Environments using Hidden Markov Models,” 2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC), 2019, pp. 21-28.

[38] Mohammed, B., Awan, I., Ugail, H. et al, “Failure prediction using machine learning in a virtualised HPC system and application,” Cluster Comput, vol. 22, pp. 471–485, 2019.

[39] Z. Li, L. Liu and D. Kong, “Virtual Machine Failure Prediction Method Based on AdaBoost-Hidden Markov Model,” 2019 International Conference on Intelligent Transportation, Big Data & Smart City (ICITBS), 2019, pp. 700-703.

[40] Z. Wang, M. Zhang, D. Wang, C. Song, M. Liu, J. Li, L. Lou, and Z. Liu, “Failure prediction using machine learning and time series in optical network,” Opt. Express, vol. 25, pp. 18553-18565, 2017.

[41] P. Guo, M. Liu, J. Wu, Z. Xue and X. He, “Energy-Efficient Fault-Tolerant Scheduling Algorithm for Real-Time Tasks in Cloud-Based 5G Networks,” IEEE Access, vol. 6, pp. 53671-53683, 2018.

[42] H. Sun, H. Yu, G. Fan and L. Chen, “QoS-Aware Task Placement With Fault-Tolerance in the Edge-Cloud,” IEEE Access, vol. 8, pp. 77987-78003, 2020.

[43] P. Guo, M. Liu and Z. Xue, “Fault-Tolerant Scheduling Algorithm for Periodic Real-Time Tasks in Clouds,” 2018 IEEE 4th Information Technology and Mechatronics Engineering Conference (ITOEC), 2018, pp. 467-470.

[44] A. Marahatta, Y. Wang, F. Zhang, A. Kumar, S. S. Tyagi, and Z. Liu, “Energy-Aware Fault-Tolerant Dynamic Task Scheduling Scheme for Virtualized Cloud Data Centers,” Mobile Networks and Applications, vol. 24, pp. 1-15, 2018.

[45] L. Ran, X. Shi and M. Shang, “SLAs-Aware Online Task Scheduling Based on Deep Reinforcement Learning Method in Cloud Environment,” 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2019, pp. 1518-1525.

[46] Y. Wei, L. Pan, S. Liu, L. Wu and X. Meng, “DRL-Scheduling: An Intelligent QoS-Aware Job Scheduling Framework for Applications in Clouds,” IEEE Access, vol. 6, pp. 55112-55125, 2018.

[47] M. E. Frincu and C. Craciun, “Multi-objective Meta-heuristics for Scheduling Applications with High Availability Requirements and Cost Constraints in Multi-Cloud Environments,” 2011 Fourth IEEE International Conference on Utility and Cloud Computing, 2011, pp. 267-274.

[48] G. Rjoub, J. Bentahar, O. Abdel Wahab and A. Bataineh, “Deep Smart Scheduling: A Deep Learning Approach for Automated Big Data Scheduling Over the Cloud,” 2019 7th International Conference on Future Internet of Things and Cloud (FiCloud), 2019, pp. 189-196.

[49] T. Pham, J. J. Durillo and T. Fahringer, “Predicting Workflow Task Execution Time in the Cloud Using A Two-Stage Machine Learning Approach,” IEEE Transactions on Cloud Computing, vol. 8, no. 1, pp. 256-268, 2020.

[50] Khan, A., M. Zakarya, I. Rahman, Rahim Khan and R. Buyya, “HeporCloud: An energy and performance efficient resource orchestrator for hybrid heterogeneous cloud computing environments,” Journal of Network and Computer Applications, vol. 173, pp. 102869, 2021.

[51] S. Sebastio, R. Ghosh and T. Mukherjee, “An Availability Analysis Approach for Deployment Configurations of Containers,” in IEEE Transactions on Services Computing, vol. 14, no. 1, pp. 16-29, 2021.

[52] L. Zhang, D. Lai, B. Xu and C. Liu, ”Scheduling Algorithms for Cloud Based Cyber-Physical Systems Specification,” 2018 24th International Conference on Automation and Computing (ICAC), 2018, pp. 1-6.

[53] Y. Hao, M. Chen, H. Gharavi, Y. Zhang and K. Hwang, “Deep Reinforcement Learning for Edge Service Placement in Softwarized Industrial Cyber-Physical System,” in IEEE Transactions on Industrial Informatics, vol. 17, no. 8, pp. 5552-5561, 2021.

[54] L. Kuang and L. Zhang, “Level value density task scheduling algorithm for cyber physical systems on cloud,” 2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), 2017, pp. 1-8.

[55] J. Zhou, J. Sun, M. Zhang and Y. Ma, “Dependable Scheduling for Real-Time Workflows on Cyber–Physical Cloud Systems,” IEEE Transactions on Industrial Informatics, vol. 17, no. 11, pp. 7820-7829, 2021.

[56] M. Yang, H. Ma, S. Wei, Y. Zeng, Y. Chen and Y. Hu, “A Multi-Objective Task Scheduling Method for Fog Computing in Cyber-Physical-Social Services,” IEEE Access, vol. 8, pp. 65085-65095, 2020.

[57] M. Jammal, A. Kanso and A. Shami, “High availability-aware optimization digest for applications deployment in cloud,” 2015 IEEE International Conference on Communications (ICC), 2015, pp. 6822-6828.

[58] M. Jammal, A. Kanso and A. Shami, “CHASE: Component High Availability-Aware Scheduler in Cloud Computing Environment,” 2015 IEEE 8th International Conference on Cloud Computing, 2015, pp. 477-484.

[59] H. Zhu and C. Huang, “Availability-Aware Mobile Edge Application Placement in 5G Networks,” GLOBECOM 2017 - 2017 IEEE Global Communications Conference, 2017, pp. 1-6.

[60] I. Lera, C. Guerrero and C. Juiz, “Availability-Aware Service Placement Policy in Fog Computing Based on Graph Partitions,” IEEE Internet of Things Journal, vol. 6, no. 2, pp. 3641-3651, April 2019.

[61] S. Yang, P. Wieder and R. Yahyapour, “Reliable Virtual Machine placement in distributed clouds,” 2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM), 2016, pp. 267-273.

[62] X. Liu, B. Cheng, Y. Yue, M. Wang, B. Li and J. Chen, “Traffic-Aware and Reliability-Guaranteed Virtual Machine Placement Optimization in Cloud Datacenters,” 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), 2019, pp. 91-98.

[63] Z. Yang, L. Liu, C. Qiao, S. Das, R. Ramesh and A. Y. Du, “Availability-aware energy-efficient virtual machine placement,” 2015 IEEE International Conference on Communications (ICC), 2015, pp. 5853-5858.

[64] X. Li and C. Qian, “Traffic and failure aware VM placement for multi-tenant cloud computing,” 2015 IEEE 23rd International Symposium on Quality of Service (IWQoS), 2015, pp. 41-50.

[65] M. Jammal, H. Hawilo, A. Kanso and A. Shami, “Mitigating the Risk of Cloud Services Downtime Using Live Migration and High Availability-Aware Placement,” 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2016, pp. 578-583.

[66] A. Zhou, S. Wang, B. Cheng, Z. Zheng, F. Yang, R. N.Chang, M. R. Lyu, and R. Buyya, “Cloud Service Reliability Enhancement via Virtual Machine Placement Optimization,” in IEEE Transactions on Services Computing, vol. 10, no. 6, pp. 902-913, 1 Nov.-Dec. 2017.

[67] Zhou, A., Wang, S., Hsu, CH. et al. "Virtual machine placement with (m, n)-fault tolerance in cloud data center," Cluster Comput 22, 11619–11631 (2019).

[68] C. Gonzalez and B. Tang, "FT-VMP: Fault-Tolerant Virtual Machine Placement in Cloud Data Centers," 2020 29th International Conference on Computer Communications and Networks (ICCCN), 2020.

[69] H. A. Alameddine, S. Ayoubi and C. Assi, "An Efficient Survivable Design With Bandwidth Guarantees for Multi-Tenant Cloud Networks," in IEEE Transactions on Network and Service Management, vol. 14, no. 2, pp. 357-372, June 2017.

[70] Chen, X. and J. Jiang. "A method of virtual machine placement for fault-tolerant cloud applications," Intell. Autom. Soft Comput. 22 (2016): 587-597.

[71] W. Zhang, X. Chen and J. Jiang, "A multi-objective optimization method of initial virtual machine fault-tolerant placement for star topological data centers of cloud systems," in Tsinghua Science and Technology, vol. 26, no. 1, pp. 95-111, Feb. 2021.

[72] S. Ayoubi, Y. Zhang and C. Assi, "A Reliable Embedding Framework for Elastic Virtualized Services in the Cloud," in IEEE Transactions on Network and Service Management, vol. 13, no. 3, pp. 489-503, Sept. 2016.

[73] P. K. Thiruvasagam, A. Chakraborty, A. Mathew and C. S. R. Murthy, "Reliable Placement of Service Function Chains and Virtual Monitoring Functions With Minimal Cost in Softwarized 5G Networks," in IEEE Transactions on Network and Service Management, vol. 18, no. 2, pp. 1491-1507, June 2021.

[74] L. Yala, P. A. Frangoudis, G. Lucarelli and A. Ksentini, "Cost and Availability Aware Resource Allocation and Virtual Function Placement for CDNaaS Provision," in IEEE Transactions on Network and Service Management, vol. 15, no. 4, pp. 1334-1348, Dec. 2018.

[75] B. Yang, Z. Xu, W. Chai, W. Liang, D. Tuncer, A. Galis, and G. Pavlou, "Algorithms for Fault-Tolerant Placement of Stateful Virtualized Network Functions," 2018 IEEE International Conference on Communications (ICC), 2018, pp. 1-7.

[76] Xu, Yansen, and Ved P. Kafle. "An Availability-Enhanced Service Function Chain Placement Scheme in Network Function Virtualization," Journal of Sensor and Actuator Networks 8, no. 2: 34, 2019.

[77] S. Sharma, A. Kushwaha, A. Somani and A. Gumaste, "Designing Highly-Available Service Provider Networks with NFV Components," 2019 28th International Conference on Computer Communication and Networks (ICCCN), 2019, pp. 1-9.

[78] M. A. Abdelaal, G. A. Ebrahim and W. R. Anis, "High Availability Deployment of Virtual Network Function Forwarding Graph in Cloud Computing Environments," in IEEE Access, vol. 9, pp. 53861-53884, 2021.

[79] W. Mao, L. Wang, J. Zhao and Y. Xu, "Online Fault-tolerant VNF Chain Placement: A Deep Reinforcement Learning Approach," 2020 IFIP Networking Conference (Networking), 2020, pp. 163-171.

[80] Availability Management Framework Specification. Accessed: 18 September, 2021. [Online]. Available: https://opensaf.sourceforge.io/SAI-AIS-AMF-B.04.01.AL.pdf.

[81] Pacemaker 1.1 clusters from scratch. Accessed: 18 September, 2021. [Online]. Available: http://clusterlabs.org/doc/Cluster/_from/_Scratch.pdf.

[82] The Corosync cluster engine. Accessed: 18 September, 2021. [Online]. Available: http://corosync.github.io/corosync/.

[83] X. Fan, W. Weber, and L. Barroso, "Power provisioning for a warehouse-sized computer, " the 34th Annual International Symposium on Computer Architecture, 2007, pp. 13–23.

[84] A. Ashraf and I. Porres, "Multi-objective dynamic virtual machine consolidation in the cloud using ant colony system," International Journal of Parallel, Emergent and Distributed Systems, vol. 33, no. 1, pp. 103-120, 2018.