Received: 6 July 2016 | Revised: 12 March 2018 | Accepted: 7 June 2018
DOI: 10.1002/cpe.4834

WILEY

RESEARCH ARTICLE

RADAR: Self-configuring and self-healing in resource management for enhancing quality of cloud services

Sukhpal Singh Gill[1] | Inderveer Chana[2] | Maninder Singh[2] | Rajkumar Buyya[1]

[1]Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Melbourne, Australia
[2]Department of Computer Science and Engineering, Thapar Institute of Engineering and Technology, Patiala, India

Correspondence
Sukhpal Singh Gill, Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Melbourne, VIC 3000, Australia.
Email: sukhpal.gill@unimelb.edu.au

Funding information
Discovery Project of Australian Research Council (ARC), Grant/Award Number: DP160102414

Summary

Cloud computing utilizes heterogeneous resources that are located in various datacenters to provide an efficient performance on a pay-per-use basis. However, existing mechanisms, frameworks, and techniques for management of resources are inadequate to manage these applications, environments, and the behavior of resources. There is a requirement of a Quality of Service (QoS) based autonomic resource management technique to execute workloads and deliver cost-efficient and reliable cloud services automatically. In this paper, we present an intelligent and autonomic resource management technique named RADAR. RADAR focuses on two properties of self-management: firstly, self-healing that handles unexpected failures and, secondly, self-configuration of resources and applications. The performance of RADAR is evaluated in the cloud simulation environment and the experimental results show that RADAR delivers better outcomes in terms of execution cost, resource contention, execution time, and SLA violation while it delivers reliable services.

KEYWORDS

cloud computing, quality of service, resource provisioning, resource scheduling, self-configuring, self-healing, self-management, service level agreement

1 | INTRODUCTION

Cloud computing offers various services like Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). However, providing dedicated cloud services that ensure various Quality of Service (QoS) requirements of a cloud user and avoid Service Level Agreement (SLA) violations is a difficult task. Based on the availability of cloud resources, dynamic services are provided without ensuring the required QoS.[1] To fulfill the QoS requirements of user applications, the cloud provider should change its ecosystem.[2] Self-management of cloud services is needed to provide required services and fulfill the QoS requirements of the user automatically.

Autonomic management of resources manages the cloud service automatically as per the requirement of the environment, therefore maximizing resource utilization and cost-effectiveness while ensuring the maximum reliability and availability of the service.[3] Based on human guidance, a self-managed system keeps itself stable in uncertain situations and adapts rapidly to new environmental situations such as network, hardware, or software failures.[4] QoS based autonomic systems are inspired by biological systems, which can manage the challenges such as dynamism, uncertainty, and heterogeneity. IBM's autonomic model[3] based cloud computing system considers MAPE-k loop (Monitor, Analyze, Plan, and Execute) and its objective is to execute workloads within their budget and deadline by satisfying the QoS requirements of the cloud consumer. An autonomic system considers the following properties while managing cloud resources[1-3]:

- Self-healing recognizes, analyzes, and recovers from the unexpected failures automatically.
- Self-configuring adapts to the changes in the environment automatically.

In this paper, we have developed a technique for *self-configu**R**ing and self-he**A**ling of clou**D** b**A**sed **R**esources*, called ***RADAR***, with the focus of two properties of autonomic-management that provide self-healing by handling unexpected failures and self-configuration of resources and applications. The performance of RADAR is evaluated in the cloud environment and the experimental results show that RADAR delivers better outcomes in terms of QoS parameters and delivers cost-efficient and reliable cloud services. The **key contributions** of this research work are outlined as follows:

i) RADAR provides self-configuration of resources and applications by the re-installation of outdated or missing components and self-healing is offered by managing unexpected faults or errors automatically.
ii) RADAR schedules the provisioned cloud resources automatically and optimizes user's QoS requirements, which improves user satisfaction and reduces the human intervention. Therefore, the cloud providers provide effective cloud service delivery and avoid SLA violations.
iii) Based on self-managed properties of an autonomic system, RADAR offers algorithms for its four different phases (monitor, analyze, plan, and execute). RADAR monitors QoS value continuously during workload execution, analyzes the alert in case of degradation of performance, plans an appropriate action to manage that alert, and implements the plan to preserve the system's efficiency.
iv) RADAR reduces SLA violations, energy consumption, and resource contention and improves availability and reliability of cloud services when implemented in a cloud environment.

The rest of the paper is organized as follows. Section 2 presents the related work. The proposed technique is presented in Section 3. Section 4 presents the performance evaluation and experimental results. Section 5 presents conclusions and future work.

2 | RELATED WORK

Autonomic resource management (also known as self-management) is a big challenge due to the discovery and allocation of a best workload-resource pair for execution of cloud workloads. As the literature on this topic is vast, we focus on self-configuring and self-healing of resources for enhancing the quality of cloud services during workload execution. Interested readers can find a detailed survey on the QoS-aware autonomic management of cloud resources in the work of Singh and Chana.[1] This section briefly discusses the related work of self-configuring and self-healing in the cloud environment.

2.1 | Self-healing

An early attempt to incorporate self-healing into cloud resource management is done by Chen et al[4]; they proposed a self-healing framework (SHelp) for management of multiple application instances in a virtual cloud environment to reduce software failures. The authors applied error virtualization techniques and weighted rescue points to develop applications to avoid the faulty path. Further, SHelp uses a rescue point database, which stores the error handling information to decrease the forthcoming faults generated by similar bugs. SHelp improves the fault detection rate and recovers a system quickly from faults, but it executes only homogeneous cloud workloads. Mosallanejad et al[5] proposed an SLA based Self-Healing (SH-SLA) model to develop hierarchical SLA for the cloud environment, which effectively monitors SLA and detects SLA violation automatically. Further, related SLAs (with same QoS requirements) communicate with each other in a hierarchical manner. The SH-SLA model performs effectively in fault detection, but it is not able to prevent the fault occurrence, which reduces user satisfaction. Similar work has been done by Mosallanejad et al,[6] who applied a replication technique for fault management.

Alhosban et al[7] proposed a Self-Healing Framework (SHF), which uses the previous history to detect the occurrence of faults in cloud-based systems. Moreover, SHF develops a recovery plan to avoid future faults generated by similar bugs, but it needs an autonomic fault prevention mechanism to improve the performance of the system. Da Silva et al[8] proposed a Self-Healing Process (SHP) for effective management of operational workflow incidents on distributed computing infrastructures. Further, incident degrees of workflow activities (ie, task failure rate due to application errors) are measured using different metrics such as data transfer rate, application efficiency, and long-tail effect to detect faults occurring during the execution of workloads. Moreover, Virtual Imaging Platform[9] is used to evaluate the performance of SHP, which demonstrates the improvement in execution time of workloads.

Li et al[10] proposed a Self-Healing Monitoring and Recovery (SHMR) conceptual model, which composes cloud services into value-added services to fulfill the changing requirements of cloud users. SHMR works in three different steps: It (1) monitors the working of the system to identify the occurrence of faults, (2) finds out the properties of faults, and (3) recovers the fault using an undo strategy. Magalhaes and Silva[11] proposed a Self-healing Framework for Web-based Applications (SFWA) to fulfill the user SLA and improve resource utilization simultaneously through self-adaption of cloud infrastructure. Experimental results show that SFWA adjusts the infrastructure dynamically to detect anomalies, which reduces the delay during workload execution. Similar work has been done by Xin,[12] who suggested that the combination of data analytics and machine learning can be utilized to improve automatic failure prediction for cloud-based environments.

Rios et al[13] proposed an Application Modeling and Execution Language (AMEL) based conceptual model for self-healing, which models the multi-cloud applications in the distributed environment. Further, AMEL uses a security modeling language to design SLA in terms of security and privacy aspects. Azaiez and Chainbi[14] proposed a Multi-Agent System Architecture (MASA) for self-healing of cloud resources by analyzing the resource utilization continuously. Further, a checkpointing strategy is used in MASA to manage the occurrence of faults and it only considers static checkpoint intervals for a fault tolerance mechanism. The main difference between these works and ours is that none of them considers autonomic fault prevention mechanism for self-healing with dynamic checkpoint intervals, which is presented in this paper. The existing self-healing frameworks[4-12] focus on monitoring of faults to maximize fault detection rate, whereas our work focuses on fault detection and prevention mechanisms.

2.2 | Self-configuring

de Sa et al[15] examined the problem of fault tolerance for distributed systems, which deals with the accuracy and speed of detection of faults. The authors proposed a QoS-based Self-Configuring (QoS-SC) framework, which uses feedback control theory for detection of faults automatically. The MATLAB based performance evaluation testbed is used to validate the proposed framework. Maurer et al[16] investigated the impact of SLA violation on resource utilization in autonomic cloud computing systems. Further, an Adaptive Resource Configuration (ARC) framework is proposed for the effective management of cloud resources to execute the synthetically generated workloads. As a part of their work, a case-based reasoning approach is used to maintain the execution details of the workload in a centralized database. The ARC framework improves the utilization of cloud resources. Their framework considers multiple resources such as bandwidth, storage, memory, and CPU, whereas our model contains bundles of resources, ie, VM instances. Salaun et al[17] proposed a Self-Configuration Protocol (SCP) for management of distributed applications in a cloud environment without using centralized database,[16] and SCP measures the execution time of workloads with the different number of virtual machines. Similar work has been done by Etchevers et al,[18] who used reconfigurable component-based systems to design distributed applications to minimize execution time, but it increases their configuration interdependencies. Panica et al[19] and Wolinsky et al[20] examined the problem of execution of legacy distributed applications using self-configuration of cloud infrastructure to maximize the availability and reliability of cloud services.

Sabatucci et al[21] investigated the concept of composing mashups of applications in the autonomic cloud environment. Sabatucci et al[22] proposed a Goal-Oriented Approach for Self-Configuring (GOASC) for automatic composing mashups of applications distributed over the geographical cloud environment. In GOASC, available functionalities are defined in terms of capabilities (for example, maximum reliability and availability) at the cloud user side, whereas mashup logic (maximum resource utilization and energy efficiency) is defined in terms of goals at the cloud provider side. Cordeschi et al[20] proposed an Energy-Aware Self-Configuring (EASC) approach for virtualized networked data centers to execute workloads with the minimum value of energy consumption. As a part of their work, an energy-scheduler is also developed to enable a scalable and distributed cloud environment for scheduling of energy-efficient cloud resources. Their approach considers homogenous workloads, whereas our model considers both homogenous and heterogeneous workloads.

Lama and Zhou[23] proposed an Autonomic Resource allOcation MechAnism (AROMA) for effective management of cloud resources for workload execution, while satisfying their QoS requirements as described in SLA. Further, auto-configuration of Hadoop jobs and provisioning of cloud resources are performed using a support vector machine-based performance model. AROMA uses previous details of resource execution for allocation of resources to new workloads, which improves utilization of resources. Konstantinou et al[24] proposed a Cost-aware Self-Configured (COCCUS) framework for effective management of cloud query services, which execute queries for optimization of cloud services in terms of QoS parameters. Initially, a cloud user specifies his QoS requirements (budget and deadline constraint) and then COCCUS executes the user queries on the available set of resources within their deadline and budget. The details of every user and their query are maintained in a centralized component that is called CloudDBMS.

Bu et al[25] proposed a VM based Self-Configuration (CoTuner) framework to provide a coordination between virtual resources and user applications using a model-free hybrid reinforcement learning approach. CoTuner handles the changing requirements of user workloads using a knowledge guided exploration policy, which uses the information of memory and CPU utilization for self-management of cloud resources to improve the learning process. Their framework considers scheduling of single queued workload, whereas our model considers clustering of workloads based on their QoS requirements. There is a large body of research devoted to self-configuration of cloud resources for execution of homogenous cloud workloads only.[15-25] However, the limited investigation has been done on the execution of heterogeneous workloads in the context of autonomic cloud computing. Further, there is a need to execute workloads without the violation of SLA. The QoS-aware autonomic resource management approach (CHOPPER)[26] is an extended version of RADAR, which considers the self-optimization for improving QoS parameters and self-protection against cyber-attacks as well.

2.3 | Comparison of RADAR with existing techniques

The proposed technique (RADAR) has been compared with existing resource management techniques, as described in Table 1. The existing research works have considered either self-healing or self-configuring, but none of the existing works considers self-healing and self-configuring simultaneously in a single resource management technique to the best of the knowledge of authors. Moreover, most of the existing works consider homogeneous cloud workloads. None of the existing work considers provisioning-based resource scheduling. Existing techniques consider the execution of single queued workload instead of clustering of workloads. RADAR schedules the provisioned resources for the execution of clustered heterogeneous workloads with maximum optimization of QoS parameters.

3 | RADAR: PROPOSED TECHNIQUE FOR SELF-CONFIGURING AND SELF-HEALING OF CLOUD RESOURCES

Figure 1 shows the architecture of RADAR, which provides self-healing by handling unexpected failures and self-configuration of resources and applications for improving the utilization of resources and reducing human intervention. SLA is used to describe the QoS parameters for execution

TABLE 1 Comparison of RADAR with existing resource management techniques

Technique	Self-Healing: Fault Detection	Self-Healing: Fault Prediction	Self-Configuring: Auto Configuration	Self-Configuring: Reinstallation	Heterogeneous Workloads	Clustering of Workloads	Provisioning Based Scheduling	QoS Parameters
SHelp[4]	✓	x	x	x	x	x	x	Fault detection rate
SH-SLA[5]	✓	x	x	x	x	x	x	Fault detection rate
SHF[7]	✓	✓	x	x	x	x	x	Throughput
SHP[8]	✓	x	x	x	x	x	x	Execution time
SHMR[10]	✓	✓	x	x	x	x	x	Resource utilization
SFWA[11]	✓	x	x	x	x	x	x	SLA violation rate and waiting time
MASA[14]	✓	x	x	x	x	x	x	Fault detection rate
QoS-SC[15]	x	x	✓	x	x	x	x	SLA violation rate and resource utilization
ARC[16]	x	x	✓	x	x	x	x	SLA violation rate and resource utilization
SCP[17]	x	x	✓	x	x	x	x	Execution time
EASC[20]	x	x	✓	✓	x	x	x	Energy consumption
AROMA[23]	x	x	x	✓	x	x	x	Resource utilization
COCCUS[24]	x	x	x	✓	x	x	x	Execution time and cost
GOASC[22]	x	x	x	✓	x	x	x	Reliability and resource utilization
CoTuner[25]	x	x	✓	✓	x	x	x	Resource utilization
RADAR (Our proposed technique)	✓	✓	✓	✓	✓	✓	✓	Fault detection rate, waiting time, execution time, energy consumption, throughput, reliability, availability, resource contention, SLA violation rate, execution cost, turnaround time, and resource utilization

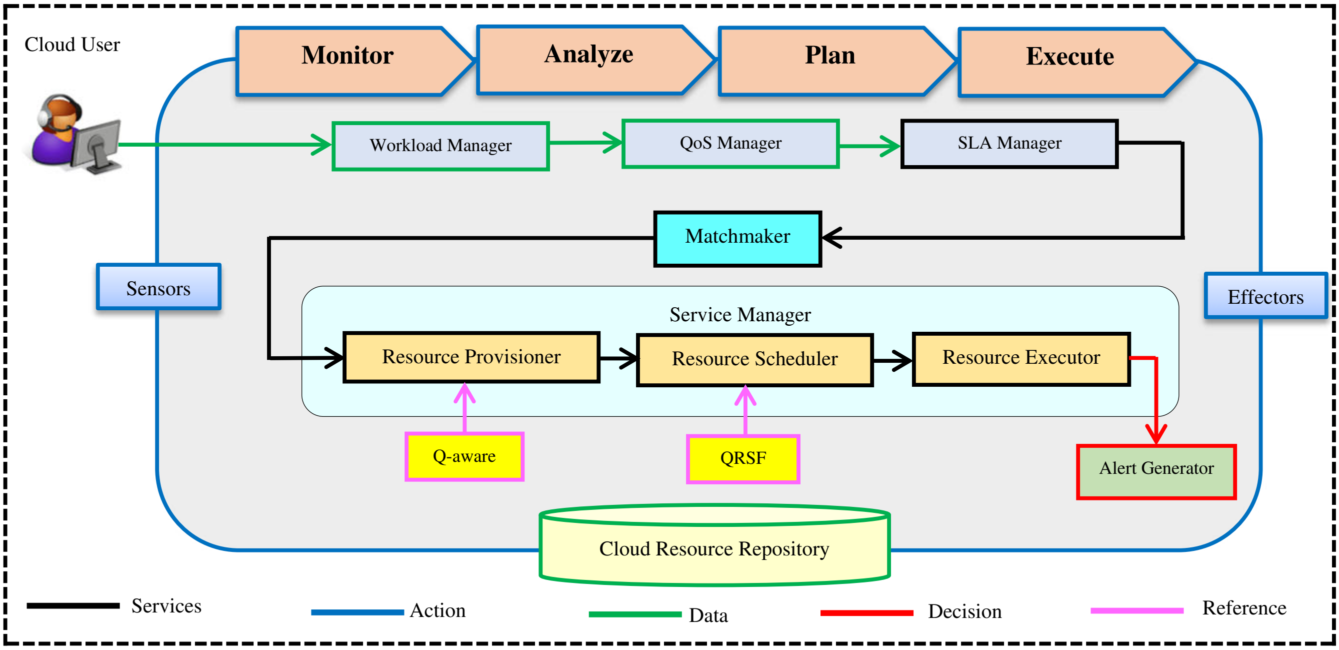

FIGURE 1 RADAR architecture

of workloads using RADAR. RADAR is an autonomic resource management technique, which ensures to serve a huge number of requests without SLA violation and manages the cloud resources dynamically.

For an execution of cloud-based applications, the mapping of cloud workloads to appropriate resources is found to be an optimization problem. Mathematically, the problem of resource scheduling for workload execution can be expressed as a set of independent workloads $\{w_1, w_2, w_3, \ldots, w_m\}$ mapped on a set of cloud resources $\{r_1, r_2, r_3, \ldots, r_n\}$. For the continuous problem, $R = \{r_k \mid 1 \le k \le n\}$ is a resource set where n is the total number of resources, whereas $W = \{w_i \mid 1 \le i \le m\}$ is a workload set where m is the total number of cloud workloads. RADAR comprises the following units.

3.1 | Workload manager

To identify its QoS requirements of a workload, *workload manager* looks at different characteristics of that workload. *Workload manager* consists of three subunits: workload queue, workload description, and bulk of workloads, as shown in Figure 2. The *Bulk of workloads* are those which are submitted by users for execution. The QoS requirements and user constraints such as deadline and budget are described as *workload description*. Table 2 lists the various types of workloads and their QoS requirements,[3,26,27] which are considered for evaluation.

All the feasible user workloads are stored in a *workload queue* for provisioning of resources before actual resource scheduling.[26] Further, *a K-means* based clustering algorithm is used for clustering the workloads for execution on the different set of resources.[28] The final set of workloads that we have chosen for evaluation is shown in Table 3.

Workload manager decides the type of workload after submission based on their priority: non-QoS (non-critical workloads) or QoS-oriented workloads (critical workloads). The priority of execution of workload is calculated based on their deadline.

3.2 | QoS manager

Based on their QoS requirements, *QoS Manager* puts the workloads into non-critical (non-urgent workloads) and critical queues (urgent workloads),[2,27] as shown in Figure 2. For *critical workloads*, it calculates the expected execution time of a workload and identifies the completion time of a workload, which is an addition of execution time and waiting time.[3] The expected execution time of the workloads can be derived from workload task length or historical trace data.[27] All the QoS oriented workloads are put into the critical queue and sorted based on their priority decided by QoS Manager. The resources are provisioned immediately with the available resources if completion time is lesser than desired deadline.[26] Otherwise, SLA is negotiated again for the extra requirement of resources and gets required resources from the reserved pool for workload execution, as shown in Figure 2. A penalty will apply in case of not fulfilling the deadline of critical workloads.

For *non-critical workloads*, the *QoS manager* checks whether the resources are free for execution. If required resources are available, then the workload will be executed directly; otherwise, put the workload into a *waiting queue* of non-critical workloads. If there is no condition (more requirement of resources and urgency), then use available resources to execute the workload immediately; otherwise, (if the required resources are lower than the provided resources) put that workload into an *under-scheduling state* (the workload in a waiting state due to unavailability of resources) till the availability of required resources.

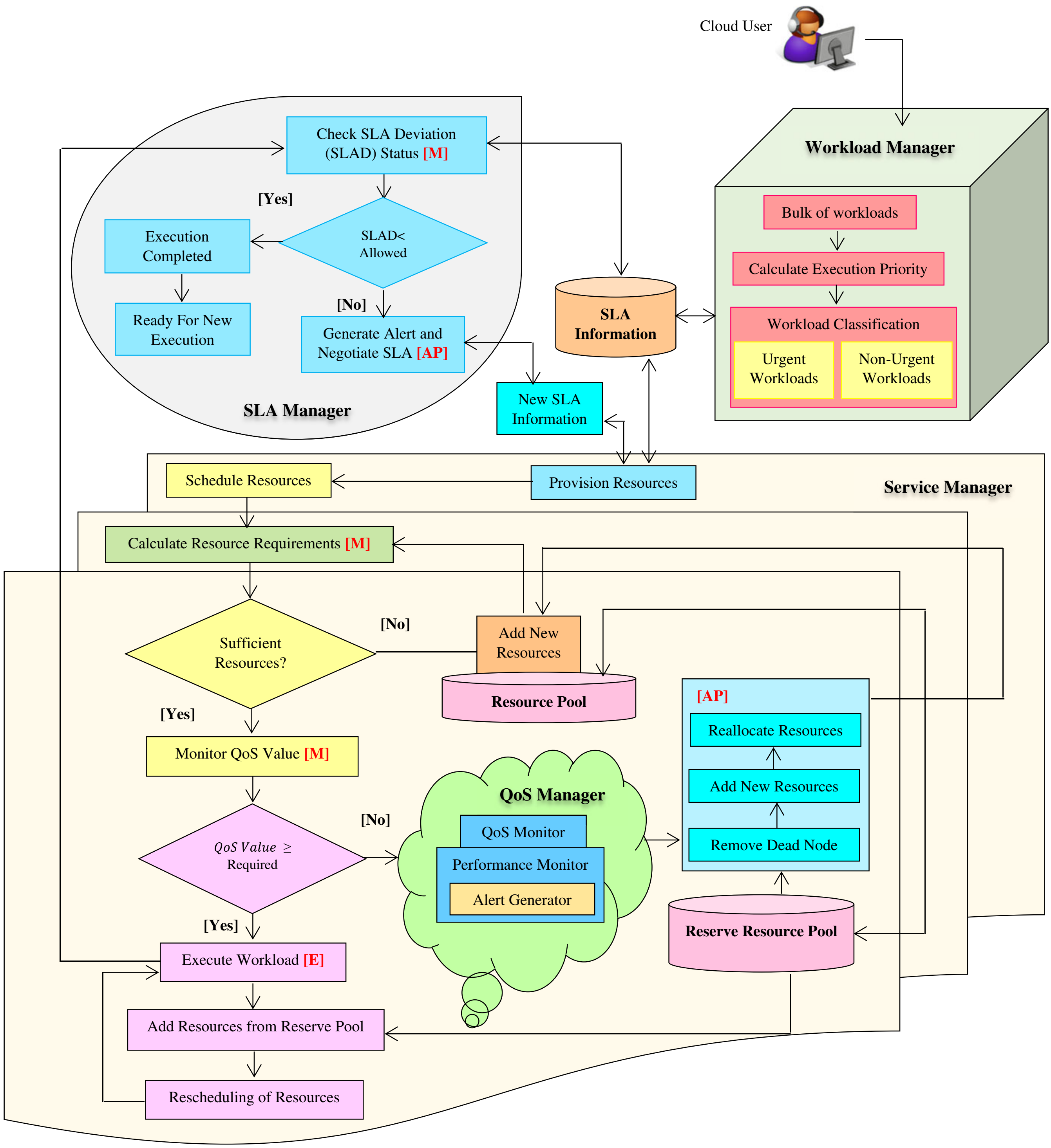

FIGURE 2 Working of RADAR

3.2.1 | QoS based metrics

The following metrics[2,3,26-29] are used to calculate the value of QoS parameters such as reliability, availability, execution time, energy consumption, throughput, waiting time, resource contention, fault detection rate, resource utilization, turnaround time, execution cost, and SLA violation rate.

Resource Utilization is a ratio of an execution time of a workload executed by a particular resource to the total uptime of that resource. The total uptime of resource is the amount of time available with a cloud resource set for execution of workloads. We have designed the following formula to calculate resource utilization (*Ru*) [Equation 1]:

$$Ru = \sum_{i=1}^{n} \left(\frac{\textit{execution time of a workload executed on the } i^{th} \textit{ resource}}{\textit{total uptime of the } i^{th} \textit{ resource}} \right), \tag{1}$$

where *n* is the number of resources.

TABLE 2 Cloud workloads and their key QoS requirements

Workload Name	QoS Requirements
Technological computing[1]	Computing capacity
Websites	High availability, high network bandwidth, and reliable storage
E-commerce	Customizability and variable computing load
Graphics oriented	Visibility, data backup, latency, and network bandwidth
Backup and storage services	Persistence and reliability
Endeavor software[2]	Correctness, customer confidence level, high availability, and security
Productivity applications[3]	Security, data backup, latency, and network bandwidth
Critical Internet applications[4]	Usability, serviceability, and high availability
Mobile computing services	Portability, reliability, and high availability
Software/project development and testing	Testing time, flexibility, and user self-service rate
Central financial services	Integrity, changeability, high availability, and security
Online transaction processing	Usability, Internet accessibility, high availability, and security
Performance testing	SLA violation rate, resource utilization, energy, cost, and time

[1]Technological computing: It contains numerical computations, atmospheric modeling, and bioinformatics. [2]Endeavor software: It contains enterprize content management, SAP (System Application and Product), and email servers. [3]Productivity applications: It contains word editors and users signing up for emails. [4]Critical Internet applications: It contains web applications including the huge amount of scripting languages.

Energy Consumption: The energy model is developed on the basis that resource utilization has a linear relationship with energy consumption.[29] Energy Consumption (*ENCN*) of resources can be expressed as [Equation 2]

$$ENCN = ENCN_{Processor} + ENCN_{Transceivers} + ENCN_{Memory} + ENCN_{Extra}. \tag{2}$$

$ENCN_{Processor}$ represents the processor's energy consumption, $ENCN_{Transceivers}$ represents the energy consumption of all the switching equipment. $ENCN_{Memory}$ represents the energy consumption of the storage device. $ENCN_{Extra}$ represents the energy consumption of other parts, including fans, the current conversion loss, and others. For a resource r_k at given time t, the resource utilization $RESU_{t,k}$ is defined as [Equation 3]

$$RESU_{t,k} = \sum_{i=1}^{m} ru_{t,k,i}, \tag{3}$$

where m is the number of cloud workloads running at time t and $ru_{t,k,i}$ is the resource usage of workload w_i on resource r_k at given time t. The actual energy consumption (E_{con}) is $ECON_{t,k}$ of a resource r_k at given time t, which is defined as [Equation 4]

$$E_{con} = ECON_{t,k} = (ENCN_{\max} - ENCN_{\min}) \times RESU_{t,k} + ENCN_{\min}, \tag{4}$$

where $ENCN_{max}$ is the energy consumption at the peak load (or 100% utilization) and $ENCN_{min}$ is the minimum energy consumption in the active/idle mode (or as low as 1% utilization), which can be calculated using [Equation 2].

Execution Cost (E_{cost}): It is the minimum cost spent to execute workload and measured in terms of Cloud Dollars (C$) and is defined as [Equation 5]

$$E_{cost} = min\,(c\,(r_k w_i))\ \ for\ 1 \le k \le n\ and\ 1 \le i \le m\ \ , \tag{5}$$

where $c(r_k, w_i)$ is the cost of workload w_i, which executes on resource r_k, as defined below:

$$\mathrm{c}(r_k, w_i) = \frac{completion\ (w_{i,} r_k)}{completion_m\ (w_i)} + Penalty\ Cost. \tag{6}$$

$completion_m(w_i)$ denotes the maximal completion time of the cloud workload. Before the estimation of the execution time, the completion time of a resource should be defined as

$$completion_m\ (w_i) = max_{r_k \in R}\ completion\ (w_{i,} r_k). \tag{7}$$

Completion time or $completion\ (w_{i,} r_k)$ is the time in which a resource can finish the execution of all the previous workloads in addition to the execution of workload w_i on resource r_k, which is defined as

$$completion\ (w_{i,} r_k) = available\_time_{r_k} \pm PTC_m\ \ (w_i)\,, \tag{8}$$

where

$$PTC_m\ \ (w_i) = max_{r_k \in R}\ PTC\ (w_{i,} r_k), \tag{9}$$

where m is the number of workloads. $available\_time_{r_k}$ is the switching time to transfer workload from a waiting queue to ready queue for execution on resource r_k. *Penalty Cost* is defined as an addition to penalty cost for different workloads (if applicable).

$$Penalty\ Cost = \sum_{i=1}^{C} (\ PC_i) \tag{10}$$

TABLE 3 K-means based clustering of workloads

Cluster	Cluster Name	Workloads
C1	Compute	Performance testing and technical computing
C2	Storage	Backup and storage services and E-commerce
C3	Communication	Mobile computing services, critical Internet applications, and websites
C4	Administration	Graphics oriented, software/project development and testing, productivity applications, central financial services, online transaction processing, and endeavor software

TABLE 4 A 10 × 6 subset of the PTC matrix

Workloads	r_1	r_2	r_3	r_4	r_5	r_6
w_1	112.14	141.44	136.65	109.66	170.46	137.58
w_2	152.61	178.26	149.78	114.26	198.92	148.69
w_3	147.23	190.23	180.26	121.65	141.65	152.69
w_4	103.62	159.63	192.85	107.69	139.89	139.36
w_5	178.65	171.35	201.05	127.65	169.36	201.66
w_6	193.62	142.65	205.36	132.26	188.33	207.72
w_7	187.24	138.23	217.58	147.69	112.39	210.98
w_8	124.13	110.65	212.39	141.26	135.88	169.35
w_9	138.56	123.65	170.26	181.65	116.61	142.87
w_{10}	131.29	129.65	142.69	199.34	125.36	147.69

Delay time is defined as the time difference between expected completion time and actual completion time. It is expressed as

$$Delay\ Time = Expected\ Completion\ Time - Actual\ Completion\ Time \tag{11}$$

$$PC = \begin{cases} Penalty_{minimum}, & if\ Expected\ Completion\ Time \geq Actual\ Completion\ Time \\ Penalty_{minimum} + \left[Penalty\ Rate \times |Delay\ Time|\right], & if\ Expected\ Completion\ Time < Actual\ Completion\ Time, \end{cases} \tag{12}$$

where $c \in C$, C is the set of penalty costs with different levels specified in RADAR.

Execution Time (E_{time}): It is the finishing time L_w of the latest workload and can also be represented as PTC workload w_i on resource r_k.

$$E_{time} = min(L_{w_i})\ w_i \in W. \tag{13}$$

The value of *[Number of workloads × number on resources]* for every workload on resources is calculated from the Predictable Time to Compute (PTC) matrix.[29] The columns of the PTC matrix demonstrate the estimated execution time for a specific resource, whereas the rows on the PTC matrix demonstrate the execution time of a workload on every resource. In this research work, the PTC benchmark simulation model is used, which was introduced in the work of Braun et al[30] to address the problem of resource scheduling. The expected execution time of the workloads can be derived from workload task length or historical trace data. A high variation in execution time of the same workload is generated using the gamma distribution method. In the gamma distribution method, a mean workload execution time and coefficient of variation are used to generate the PTC matrix.[31] Table 4 shows a 10 × 6 subset of the PTC matrix and the results provided in this research work used the matrix of size 90 × 36.

The columns of the PTC matrix demonstrate the estimated execution time for a specific resource, whereas rows on the PTC matrix demonstrate the execution time of a workload on every resource.

Resource Contention: It occurs when the same resource is shared by more than one workload.[26] The main reasons for resource contention are (i) when the same resource is used to execute more than one workload, (ii) unavailability of a required number of resources to execute the current set of workloads, and (iii) the number of workloads with an urgent deadline, which are trying to access the required same resources; more workloads with an urgent deadline creates more resource contention. Resource contention (*ResCon*) is defined during scheduling of resources at time ***t*** and is calculated as

$$ResCon\ (t) = \sum_{r \in ResourceList} ResCon\ (t, r) \tag{14}$$

$$ResCon\ (t, r) = \sum_{rt \in ResourceType} ResCon\ (t, r, rt), \tag{15}$$

where ***r*** is the list of resources and ***rt*** specifies the type of resource (overloaded or not). We have considered W_Q as a set of total workloads (Equation 16), which is executed by different resources.

$$W_Q = \{W_1, W_2, \ldots\ldots\ldots\ldots W_m\} \tag{16}$$

During the execution of workloads, some workloads overload the resources, which is denoted by a set called *OVERLOAD* (Equation 17).

$$OVERLOAD = \{W_1, W_2, \ldots\ldots\ldots\ldots W_o\} \tag{17}$$

RCStatus (t, r, rt) specify the current status of resource contention (Equation 18) in terms of Boolean statements, ie, True or False.[26]

$$RCStatus\ (t, r, rt) = \begin{cases} 1, & \sum_{w \in W_Q\ (tr)} \left(rt \in w.OVERLOAD == TRUE\ ?\ 1\ :\ 0\right) \geq 1 \\ 0, & otherwise, \end{cases} \tag{18}$$

where $w.\,OVERLOAD$ specifies the set of workloads, which overloads the resource at time $\boldsymbol{t}$. $(rt \in w.\,OVERLOAD = = TRUE\ ?\ 1 : 0)$ finds the status of a resource, which is overloaded or not. If its value is equal to or more than one, then the value of *RCStatus* (t, r, rt) is one; otherwise, its value is zero.

$$ResCon\ (t, r, rt) = \begin{cases} \sum_{w \in W_Q\ (tr) \wedge rt \in w.OVERLOAD} w.ResourceRequirment\ [rt], & RCStatus\ (t, r, rt) = 1 \\ 0, & otherwise, \end{cases} \tag{19}$$

$w.\,ResourceRequirment$ specifies the resource requirement of $\boldsymbol{w}$ in terms of capacity (storage, processor, or memory), and throughout all experiments, this value, measured in seconds, is as a value for comparison and not an exact time for resource contention.

SLA violation Rate is defined as the product of Failure rate and weight of SLA[3] and is calculated as

$$\text{List of SLA} = < m_1, m_2 \ldots\ldots\ldots\ldots\ldots\ldots m_n >, \text{where } n \text{ is the total number of SLAs.}$$

$$Failure\,(m) = \begin{cases} 1, & m \text{ is not violated} \\ 0, & m \text{ is violated} \end{cases} \tag{20}$$

Failure rate (Equation 21) is computed as a ratio of the summation of all the SLA violated to the total number of SLAs.

$$Failure\ Rate = \sum_{i=1}^{n} \left(\frac{\text{Failure}\ (m_i)}{n}\right) \tag{21}$$

SLA violation rate is a product of failure rate and summation of every weight for every SLA.

$$SLA\ Violation\ Rate = Failure\ Rate \times \sum_{i=1}^{n} (w_i), \tag{22}$$

where w_i is the weight for every SLA, which is calculated using (Equation 23). The consequence of collected data is used by the following formula[27] to calculate the weight of quality attributes (Equation 23):

$$w_i = \frac{1}{N_f \times (M_v + q)} \times \sum_{a=1}^{N_f} R_a \times 100, \tag{23}$$

where i is cloud workload, q is level of measurement of quality attribute (q-value), N_f is number of research papers used to collect data, M_v is maximum value for a quality attribute, and R_a is the sum of responses for an attribute; the value of w_i will be in the range 0% - 100%. An analysis has been conducted to acquire the data from different research papers of cloud computing from reputed journals about cloud workloads with the objective to know that how to assign the weights to the quality attributes according to significance.[27] Subsequently receiving the responses,

TABLE 5 Conversion metric

Approximate Weight (%)	Weight
0-20	1
20-40	2
40-60	3
60-80	4
80-100	5

TABLE 6 Level of measurement of quality attribute

Level of Measurement of Quality Attribute	q-Value
High	3
Medium	2
Low	1

an industry standard baseline and adequate weights to the quality attributes have been defined. The conversion metric is used to assign the values (minimum = 1 and maximum = 5) corresponding to the percentage,[28] as shown in Table 5.

The level of measurement of a quality attribute will be of three types: High, Medium, and Low, as described in Table 6. The result of the data analysis is as follows; the number of research papers of different contexts have been studied and the maximum probable value for a quality attribute is 5. For example, computing the average of the "Availability" quality attribute under workload "*Websites*" is as follows.

$N_f = 11$, Workload = Website and Quality Attribute = Availability, $M_v = 5, q = 3$ (because availability should be high, described in QoS requirements of SLA and the q-value has been calculated with the help of QoS metrics) and the sum of the responses $\sum_{k=1}^{11} R_a = 29$.

$$w_i = \frac{1}{11 \times (5+3)} \times 29 \times 100 = 32.95.$$

For $w_i = 32.95$, the average weight assigned for availability is 2 by using Table 5. Through this technique,[27,28] the average weights for every quality attribute have been calculated.

Fault Detection Rate is the ratio of the number of faults detected to the total number of faults existing. Fault Detection Rate (FDR) is calculated as

$$FDR = \frac{Number\ of\ Faults\ Detected}{Total\ number\ of\ Faults}. \tag{24}$$

Faults can be a network, software, or hardware, which is detected based on the violation of SLA.

Throughput is the ratio of the total number of workloads to the total amount of time required to execute the workloads, and it is calculated as

$$Throughput = \frac{\text{Total Number of Workloads } (W_n)}{Total\ amount\ of\ time\ required\ to\ execute\ the\ workloads\ (W_n)}. \tag{25}$$

Reliability of the resource has to be checked for scheduling of the resources.[29] With the help of the reliability parameter, we can check the fault tolerance of the resource. Reliability of the resource is calculated as

$$re = e^{-\lambda t}. \tag{26}$$

re = reliability of resource, *t* = time for the resource to deal with its request for any workload's execution, and λ = the failure rate of the resource at the given time, which is calculated using [Equation 21].

Availability is defined as an interval of the real line. Availability is represented as

$$A = \sum_{t=0}^{c} A\ (t), \tag{27}$$

where c is an arbitrary constant ($c > 0$) and its value is chosen to select the time interval, for which the availability of system can be tested. We have used $c = 850$ seconds for *Test Case 4* to find the experiment statistics (Section 4). Therefore, the availability $A\ (t)$ at time $t > 0$ is represented by

$$A\ (t) = \Pr\left[X(t) = 1\right] = E\ \left[X(t) = 1\right]. \tag{28}$$

Further, the status function is defined as

$$X\ (t) = \begin{cases} 1, & RADAR\ functions\ at\ time\ t \\ 0, & Otherwise. \end{cases} \tag{29}$$

Average Waiting Time or Waiting Time is a ratio of the interval computed between workload execution start time (WE_i) and workload submission time (WS_i) to the number of workloads. It is calculated as

$$W\ (n) = \sum_{i=1}^{n} \left(\frac{WE_i - WS_i}{n}\right), \tag{30}$$

where *n* is the number of workloads.

Turnaround Time is a ratio of the interval computed between workload completion time (WC_i) and workload submission time (WS_i) to the total number of workloads. It is calculated as

$$T\ (n) = \sum_{i=1}^{n} \left(\frac{WC_i - WS_i}{\mathrm{n}}\right), \tag{31}$$

where *n* is the number of workloads.

3.3 | SLA manager

An SLA document is prepared based on SLA information finalized between user and provider. The SLA document contains information about SLA Violation (the value of minimum and maximum deviation and compensation or penalty rate in case of violation of SLA). The deviation status estimates the deviation of QoS from predictable values.[3] A penalty will be imposed if the deviation is more than the allowed for urgent workloads, and penalty

TABLE 7 Types of workloads and their urgency details

Workload Type	Slack Time (Seconds)	Delay Time (Seconds)	Deviation Status	Minimum Penalty	Penalty Rate
Non-Critical (Relaxed	60	0-50	5%	50 Seconds	2%
Deadline)		51-100	10%	100 Seconds	3%
		101-150	15%	150 Seconds	4%
Critical (Urgent	10	0-50	5%	200 Seconds	5%
Deadline)		51-100	10%	400 Seconds	6%
		101-150	15%	600 Seconds	7%

can be an allocation of reserve resources to a specific workload for compensation. RADAR minimizes the effect of inaccuracy by (1) adding slack time during scheduling and (2) considering the penalty-compensation clause in SLAs in case of its violation. The SLA manager of RADAR tries to execute the workload within user-specified deadline and budget with maximum resource utilization and without violation of SLA.

Based on the workload deadline, workloads are classified into two categories: (i) critical and (ii) non-critical. Table 7 describes the details about Non-Critical (Relaxed Deadline) and Critical (Urgent Deadline) workloads and the calculation of compensation and penalty.[26]

The following example shows the estimation of penalty or compensation for "CRITICAL" workload with Delay Time = 50 (Deviation Status = 5%) seconds:

$$Compensation = Penalty_{minimum} + \left[Penalty\ Rate \times Delay\ Time\right]$$

$$Compensation = 200\ Seconds + \left[5 \times 50\ Seconds\right] = 450\ Seconds.$$

It will provide 450 seconds free cloud service as a compensation or penalty.

3.4 | Service manager

Initially, submitted workloads are moved to workload queue (W_Q= {W_1, W_2, ... W_m}) as an input. The *Matchmaker* maps the workloads to the suitable resources using Q-aware,[27] which uses SLA information for resource provisioning. Q-aware[27] is a resource provisioning technique, in which resource provisioner provisions resources to different cloud workloads based on their QoS requirements, as described in SLA. Further, the resource scheduler of QoS based Resource Scheduling Framework (QRSF) schedules the provisioned resources[28] for workload execution. Figure 3 shows the interaction of the cloud user, the SLA manager, and the service manager for execution of workloads, in which SLA is signed initially. Based on SLA information, RADAR generates the workload schedule. A workload will be executed within its specified budget and deadline with maximum resource utilization and without violation of SLA. In case of the SLA violation, RADAR uses STAR[3] to renegotiate SLA again with a new deadline and budget. Workload will be dispatched for execution after verification of every critical parameter, and the resource executor executes the workload after payment.

RADAR returns the resources to resource pool after successful execution of workloads. Finally, updated workload's execution information returns to the corresponding cloud user. Figure 2 shows the execution of workloads using subunits of an autonomic model[3] such Monitor [M], Analyze and Plan [AP], and Executor [E]. The performance monitor monitors the performance of workload execution continuously and generates alerts in case of degradation of performance. RADAR generates following alerts in two different cases, as shown in Figure 2.

- Alert 1: If there is an unavailability of required resources for workload execution, then perform an *Action 1* for Reallocation of resources.
- Alert 2: If the value of SLA deviation is more than allowed, then perform an *Action 2* to Renegotiate SLA.

RADAR performs the same action two times. The system is treated as down if RADAR fails to correct it. To interact with the outside environment, RADAR uses two interfaces of an autonomic model[3]: Sensors and Effectors. *Sensors* read the QoS value to check the performance of the system. Initially, the coordinator node gets the updated information (QoS value, faults or new updates) from processing nodes and transfers this information to Monitors. For example, the energy consumption of workload execution can be one reason for performance degradation. *Sensors* read the value of consumption of energy and compare with its threshold value to test the performance of the system. RADAR continues the workload execution if the value of consumption of energy is less than its threshold value. Otherwise, add new resources to improve energy efficiency using these steps: (i) declare the current node as dead, (ii) eliminate the node, (iii) add new resources, and (iv) start the execution after reallocation of resources and send the updated information to the coordinator node.[29,32] *Effector* exchanges the updated information about new alerts, rules, and policies to the other nodes. Service Manager works in two phases: self-configuring and self-healing, as shown in Figure 3.

3.4.1 | Self-configuring

Some components of the system need updates, and other components need reinstallation due to changing conditions of a computing environment. Based on the generated alert, RADAR offers self-configuring of installation of outdated or missed components automatically. We propose an algorithm for self-configuring, which offers an autonomic installation of new components or reinstallation of outdated or missed components.

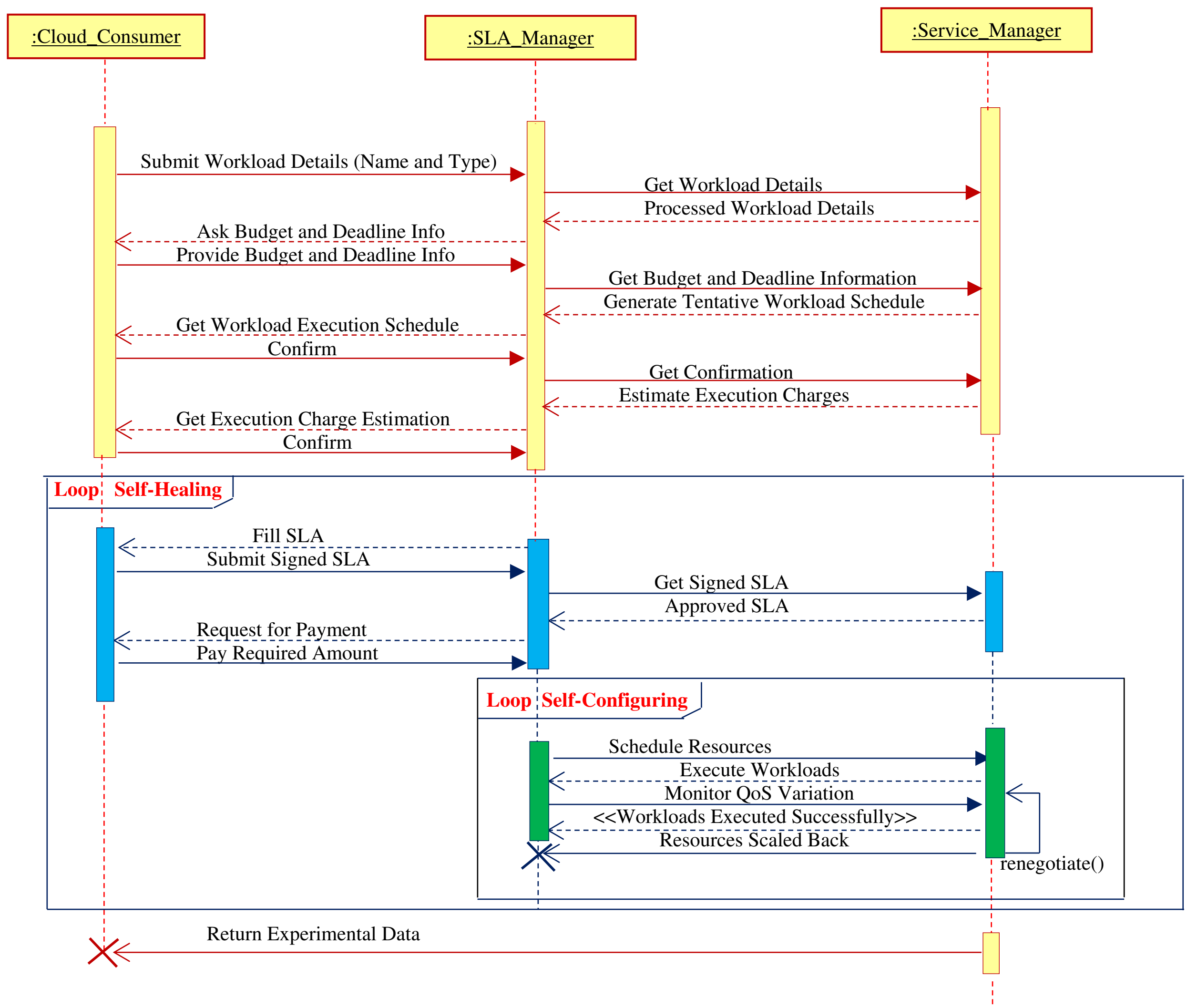

FIGURE 3 Design description

[Algorithm 1: Self-Configuring] works in three different sub-modules: (i) monitoring, (ii) analyzing and planning, and (iii) executing. Figure 4 shows the graphical representation of the pseudocode of *Algorithm* 1.

Initially, *Monitor* gathers the updated information (QoS value) from different sensors and monitors the performance variations continually by doing the comparison between the actual and expected value of QoS. Basically, an expected value of QoS is considered as its threshold value, which also contains the maximum value of SLA deviation. We have observed the actual value of QoS based on SLA violation, new updates (missing or outdated components), and faults (hardware, software, or network).

For *Monitoring Module*, Figure 4 shows that RADAR checks the status of active components in the monitoring unit by using a software and hardware component agent. The software component agent monitors the status of active software components, which can be 'MISSING' or 'OUTDATED.' If the status is 'MISSING,' then the software component agent reinstalls the component after uninstalling the existing component. If the status is 'OUTDATED,' then the software component agent installs the new version of that component. The hardware component agent monitors the status of active hardware components, and it uses the log information to track the status of different hardware components and generate an alert in case of error. Log information[26] has fields such as (1) *Time Stamp* (the time of occurrence of error in that event, (2) *Event Type* (the type of event occurred, ie, 'CRITICAL' OR 'ERROR'), (3) *Source* (source is the software that logged the event, which can be either a program name, such as "SQL Server," or a component of the system or of a large program, such as a driver name), and (4) *Event Id* (Event has a unique identity number). An alert will be generated if any of the events ('CRITICAL' OR 'ERROR') occurs and uses log information [Component_Name and Component_Id] to update the database.

A "machine checks log" is used in a RADAR to manage the failures of hardware components and it can generate an alert for internal errors quickly. RADAR uses a centralized database to maintain the machine check logs related to hardware. Further, lexical analyzer-based freeware tools such as MCat (windows) and MCELogs (Linux)[33] are used to refine the log information (*Event Id*, *Event Type* ($Event_{Type}$), *Time Stamp and Source*) before storing it into the database. RADAR captures logs with the event type: "CRITICAL" or "ERROR".

Analyzing and Planning Module analyzes the generated alert and behavior of a software and hardware component. For *hardware component*, it declares the component as 'DOWN' if the status of a component is either 'CRITICAL' OR 'ERROR' and restarts it to its status again. If its status changes to 'ACTIVE,' then continue execution; the otherwise 'INACTIVE' component will be replaced with a new component and start execution.[26] For *software component*, if its status is either 'MISSING' or 'OUTDATED' (Event Type), then the software agent performs the following steps:

Algorithm 1 : Self-configuring

 # MONITORING
BEGIN
Set of Components: $Set_{components} = \{COM_1, COM_2 \ldots\ldots\ldots\ldots\ldots\ldots COM_y\}$
Set of Active Components: $Set_{Active\ components} = \{COM_1, COM_2 \ldots\ldots\ldots\ldots\ldots\ldots COM_z\}$, where $z \le y$
while true **do**
For all software components
for all [$Set_{Active\ components}$] Check $\text{Component}_{\text{Status}}$
if ($\text{Component}_{\text{Status}}$ == '*MISSING*') **then**
Uninstall and Reinstall the component for *RECONFIGURATION*
end if
if ($\text{Component}_{\text{Status}}$ == '*OUTDATED*') **then**
Create Alert [For new version of component]
end if
end for
For all hardware components
Trace Log
for all [$Set_{Active\ components}$] Get detail of status [$Event_{Type}$, Time_Stamp, Event_Id]
if ($Event_{Type}$ == '*CRITICAL*' OR '*ERROR*') **then**
Use log information [Component_Name and Compoenent_Id] to Update Database
Create Alert
else
'*TAKE NO ACTION*'
end if
end for
end while
 # ANALYZING and PLANNING
 # Process logs
 # Assess $\text{Component}_{\text{Status}}$ [Hardware Component]
for all [$Set_{Active\ components}$]
if ($Event_{Type}$ == '*CRITICAL*' OR '*ERROR*') **then**
Set status [$Set_{Active\ components}$] = 'DOWN'
Assess $\text{Component}_{\text{Status}}$ [$Set_{Active\ components}$]
if $\text{Component}_{\text{Status}}$ [$Set_{Active\ components}$] = ! '*ACTIVE*'
Create Alert
end if
end for
 # Assess $\text{Component}_{\text{Status}}$ [Software Component]
for all [$Set_{Active\ components}$] if (Event_Type == '*OUTDATED*' OR '*MISSING*')
if ($\text{Component}_{\text{Status}}$ [$Set_{Active\ components}$] = '*OUTDATED*') **then**
Perform Component Replacement with updated version of component
else if ($\text{Component}_{\text{Status}}$ [$Set_{Active\ components}$] = '*MISSING*') **then**
Reinstall the component for Reconfiguration
Assess $\text{Component}_{\text{Status}}$ [$Set_{Active\ components}$]
if $\text{Component}_{\text{Status}}$ [$Set_{Active\ components}$] = ! '*ACTIVE*' **then**
Create Alert
end if
end if
end for
 # EXECUTION
if (Component = '*New*') **then**
Add component [bind component by exchange messages with other existing components]
Begin Execution of Newly Added Component
Assess Performance Status
if ($E_{time} \le D_t$ && $E_{cost} \le B_E$] or [$E_{Con} \le E_{Threshold}$) == '*TRUE*' **then**
Continue Execution of Component
else
Replace current component with new component
end if
end if
if (Component = 'EXISTING') **then**
if Existing Component == '*ERROR*' **then**
Create Backup of Data
Based on type of failure, Send Message to Restart Agent to Restart
end if
end if

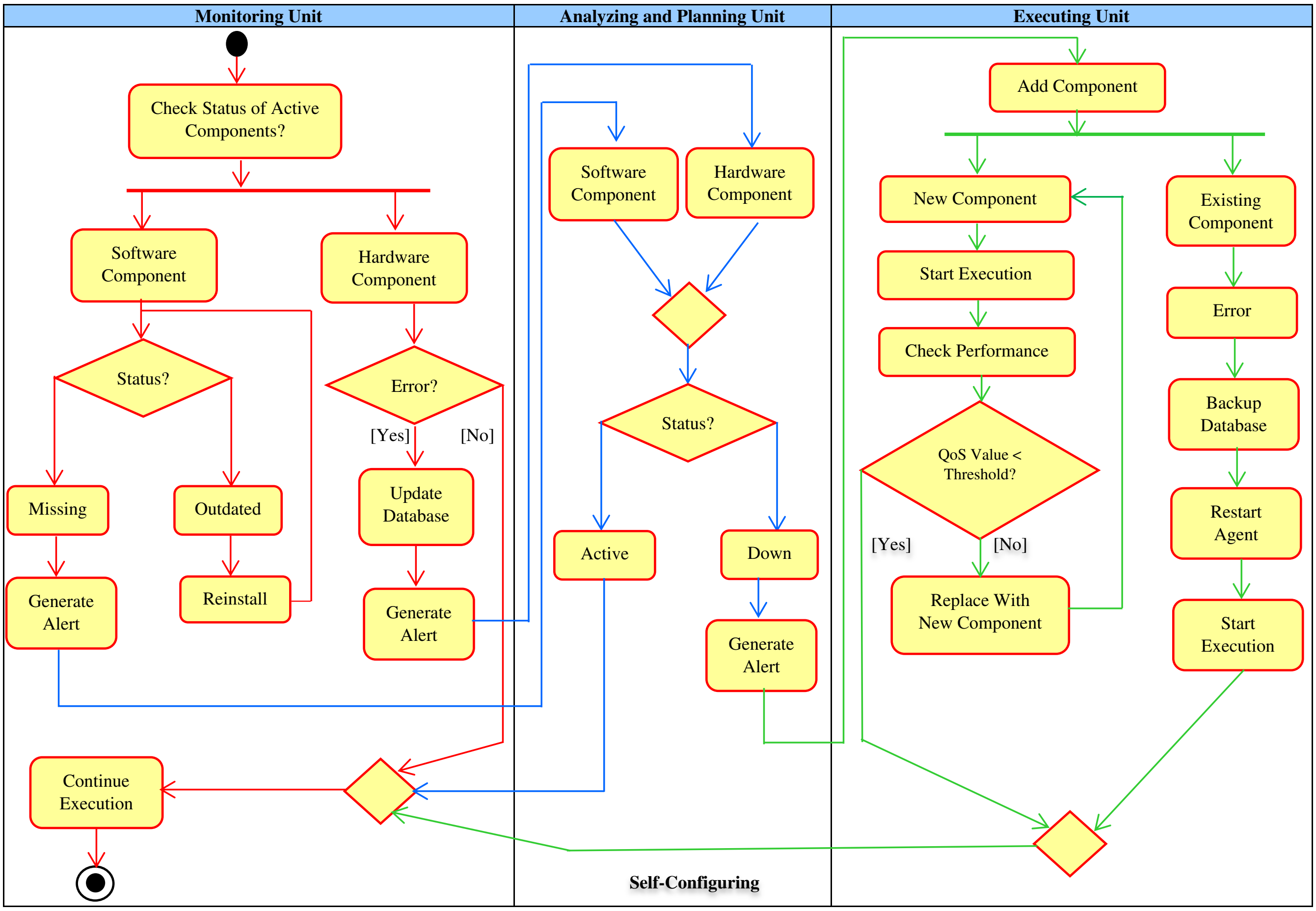

FIGURE 4 Process of self-configuring

(1) if Event Type is 'MISSING,' then reinstall the component and, (2) if Event Type is 'OUTDATED,' then that component is replaced with an updated version. After analysis of the status of hardware and software component, RADAR makes a plan to correct these errors automatically. Further, RADAR checks the status of active components [$Set_{Active\ components}$] continually for future performance analysis.

For new component, the *Execution* module implements the plan and binds the new component with existing components by exchanging messages between them. Further, it starts the execution of a new component, as shown in Figure 4. During the execution of workload, the value of Execution Time (E_{time}), Average Cost (E_{cost}), and Energy Consumption (E_{con}) is calculated for every workload. If this condition ([$E_{time} \leq D_t$ && $E_{cost} \leq B_E$] or [$E_{Con} \leq E_{Threshold}$]) is false, then generate an alert and replace this component with another qualified component.

$E_{Threshold}$ is the maximum allowed value for energy consumption.[26,28] The estimated budget (B_E) is the maximum value of cost that the user wants to spend and measured in Cloud Dollars (C$). Deadline Time ($Dt$) is defined as the time duration between the current time (Ct_i) and workload deadline (Wd_i) and Equation (32) is used to calculate Deadline Time.

$$Deadline\ Time\ (Dt_i) = \sum_{i=1}^{n} (Wd_i - Ct_i), \tag{32}$$

where Wd_i is the workload deadline and Ct_i is the current time. For the existing component, restart the component after saving its state if an error occurs. If the component is still not performing effectively, then the issue can be resolved in two ways: (a) install an updated version of a component or (b) reinstall the component. Further, RADAR manages all the updates and stores in a centralized database and maintains its backup as a database replica. Backup Database (BD) can be used in the future if master Data Base (DB) goes down[33] and BD acts as the master until the master database is up again.

3.4.2 | Self-healing

RADAR offers self-healing to ensure the proper functioning of a system by making mandatory changes to recover from the different types of faults, which can be a network, software, or hardware fault. Network fault can occur due to network breakage, physical damage, packet loss, and lack of scalability in distributed networks. Hardware fault can occur due to non-functioning of components such as hard disk, RAM, or processor. Software fault can occur due to unavailability of a required number of resources, storage space, resource contention (deadlock), and unhandled exception in high resource intensive workloads. [Algorithm 2: Self-Healing] is working in three different sub-modules: (i) monitoring,)ii) analyzing and planning, and (iii) executing. Figure 5 shows the graphical representation of the pseudocode of *Algorithm* 2. *Monitoring Module* comprises of hardware, network, and a software agent to manage the different types of faults. The hardware agent continually monitors the performance of hardware components such as CPU and MEMORY.

Algorithm 2 : Self-healing

 # MONITORING
Begin
List of Nodes: $Node_{List} = \{Node_1, Node_2 \ldots\ldots\ldots\ldots\ldots\ldots .Node_n\}$, where $Node_{current}$ *is current node*
if ($Node_{current} \cap Node_{List} ==$ NULL) **then**
  Scan drivers and assess replica of original drivers
  Add node [$Node_{List} \leftarrow Node_{current}$]
else
  Node is already existed [Create Alert]
end if
for all Hardware Node ($Node_{Status}$)
  Get detail of status [$Event_{Type}$, Time_Stamp, Event_Id]
  if ($Event_{Type} ==$ '*CRITICAL*' OR '*ERROR*') **then**
    Use log information [Node_Name and MAC_Address] to Update Database
    Create Alert
  end if
end for
for Software Monitoring [CPU and MEMORY]
  if (Status ['*CPU*' || '*MEMORY*'] > THRESHOLD VALUE) **then**
    Create Alert
    Update Memory and CPU information
  end if
end for
 # ANALYZING and PLANNING
 # Process logs
#Assess Hardware Errors
for all $Node_{current}$ Where [$Event_{Type} ==$ '*CRITICAL*' OR '*ERROR*']
  Set status $Node_{current} =$ '*DOWN*'
  Restart the Node [$Node_{current}$]
  if $Node_{current} ==$ '*RESTARTED*' **then**
    Assess $Node_{Status}$
      if $Node_{Status}$ [$Node_{current}$] = ! '*ACTIVE*'
        Create Alert
      end if
  end if
end for
 # Assess for Software Errors
for all $Node_{current}$([CPU || MEMORY] > THRESHOLD VALUE)
  do
  Set status $Node_{current} =$ '*DOWN*'
  Restart the Node [$Node_{current}$]
    if $Node_{current} ==$ '*RESTARTED*' **then**
      Assess $Node_{Status}$
      if $Node_{Status}$ [$Node_{current}$] = ! '*ACTIVE*' **then**
      Create Alert
      end if
    end if
end for
 # EXECUTION
if New_Workload_Submission **then**
  if (Nominated_Node [$Node_{current}$] $\subset$ **FAULT_NODE_LIST**) then
  Choose Another Node
  end if
end if
if (New_Workload_Submission == '*ERROR*') **then**
  Create Backup of Data
  Based on type of failure, Send Message to Restart Agent to Restart
end if

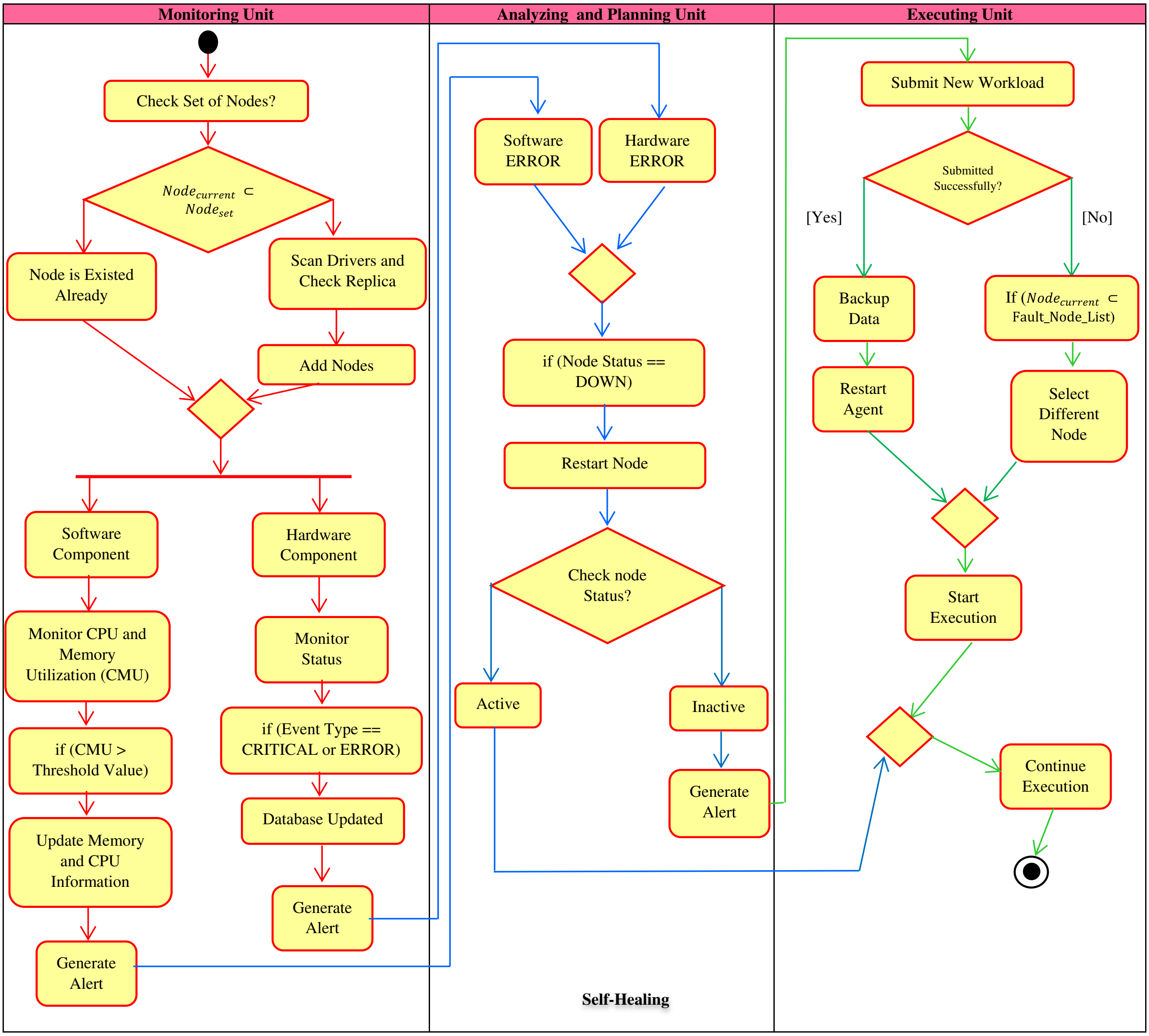

FIGURE 5 Process of self-healing

To maintain the performance of the system, RADAR performs hardening[34] to ensure that the driver is working properly. Hardware Hardening is the process in which the driver works correctly even though faults occur in the device that it controls or other faults originating from outside the device.[34] A hardened driver should not hang the system or allow the uncontrolled spread of corrupted data as a result of any such faults.[33] Whenever the node is added to the network, Hardware Hardening Agent (HHA) checks for its device drivers and performs the process of hardening. [ALGORITHM 3: Hardware Hardening Process (HHP)] shows the process of hardware hardening in RADAR.

Initially, HHA starts the process of hardening by scanning the drivers to be hardened.[33] Further, when the new node is added, a replica of the original drivers is created. RADAR uses the concept of Carburizer,[34] which pushes the code on that node when a new node is added. The process of hardware hardening consists of these consecutive steps: (1) scan the source code for all drivers, (2) find the chance of failure in the code, (3) replace the code after identification of the code, (4) replace the original drivers with the hardened drivers after hardening, (5) the hardware agent monitors the performance of the hardened driver continuously, and (6) replaces the hardened driver with original driver in the case of performance degradation. RADAR uses machine check log to maintain the database of hardware failures. RADAR continually monitors the status of the system, which contains the information about the event [Event Type, Event Id, Timestamp] and event type, which can be either 'CRITICAL' OR 'ERROR'. Further, the hardware agent generates the alert and maintains the log information [Node_Name and MAC_Address], as shown in Figure 5. The software component agent checks the usage of CPU and MEMORY continually. To test the performance of MEMORY and CPU, RADAR fixes the threshold value of their usage.

Algorithm 3 : Hardware hardening process

1. Start
2. Set of Hardened Nodes: $Node_{Hardened} = \{Node_1, Node_2 \ldots\ldots\ldots\ldots\ldots\ldots .Node_n\}$, where $Node_{current}$ *is current node*
3. **if** ($Node_{current} \cap Node_{Hardened} ==$ NULL) **then**
4. Begin Process of Hardening
5. Scan drivers and create list of drivers to be hardened, $List_{Hardened}$
6. Generate replica of original drivers of $List_{Hardened}$
7. **for all** $List_{Hardened}$ **do**
8. Using carburizer engine to harden driver
9. Add node $[Node_{Hardened} \leftarrow Node_{current}]$
10. **else**
11. Node is already Hardened
12. **end if**

Figure 5 shows that the system generates an alert if the value CPU and MEMORY usage is greater than its threshold value. RADAR uses the network agent to monitor the data transfer rate from one node to another. Manager nodes receive the status from all the processing nodes in a specific network. There can be network failure if processing nodes do not respond periodically.

The *Analyzing and Planning* unit analyzes the alert for software or hardware, which is generated by the software or hardware agent, as shown in Figure 5. The hardware agent changes status of node *N* as 'DOWN' if the event type is either 'ERROR' OR 'CRITICAL' and restarts the failed node using the restart agent. Further, it measures the status of restarted node again and continues execution if the status of node changes to 'ACTIVE'; otherwise, it generates an alert and replaces the current node (down) with another stable node. The stability of a node is identified from their log information during their past performance. The software agent analyzes an alert if the value of CPU and MEMORY usage is greater than its threshold value and restarts the node using a restart agent, as shown in Figure 5. Further, it measures the status of the restarted node again and continues execution if the status of the node changes to 'ACTIVE'; otherwise, it generates an alert for replacement of the current node with a more stable node to continue execution. The network agent analyzes the behavior of the network and finds out the reason of failure, which can be network breakage, physical damage, packet loss, and lack of scalability in distributed networks. Then, the network agent selects a plan from the past network log to correct it. The *Restart agent* reboots the system automatically to recover the system from hardware failures.

The *Execution* module implements the plan by replacing the current node (down) with a different node, which is more stable among available nodes. This module also saves the state of the node and performs restarting if an error occurs during workload execution. Further, a faulty component will be replaced by a new component if restart fails to recover from failure.

4 | PERFORMANCE EVALUATION

Figure 6 shows the cloud testbed, which is used to evaluate the performance of RADAR. We have modeled and simulated a cloud environment using the CloudSim toolkit.[35] For the experimental setup, the computing nodes are simulated that resembles resource configuration, as shown in Table 8. Three servers with different configurations have been used to create virtual nodes in this experimental work. Each virtual node consists of Execution Components (ECs) for workload execution, and the cost for every EC is defined in C$/EC time unit (Sec). The configuration details of the cloud testbed are shown in Table 8, and we have assigned manually an access cost in Cloud Dollars (C$). When ECs have different capabilities, then this cost does not essentially reflect the execution cost.[26,29,32] In this experimental work, access cost is translated into C$ for each resource to find out the relative execution cost for workload execution.

4.1 | Fault management

We have used Fault Injection Module (FIM-SIM)[36] to inject faults automatically to test the reliability of RADAR. FIM-SIM is working based on event-driven models and injects faults into the CloudSim[35] using different statistical distributions at runtime. We selected the Weibull Distribution[37] to inject faults in this research work. We have injected three types of faults: VM creation failures, host failures (Processing Elements failure and memory failure), and high-level failures like cloudlets failures (which are caused by any networking problem that CloudSim[35] cannot handle). The injection of faults affects cloud resources during the simulation period. Interested readers can find detailed information about fault injection in the work of Nita et al.[36] Carburizer[34] is used in RADAR to harden the device drivers in the Fault Manager to detect the faults and prevent the system from faults efficiently.

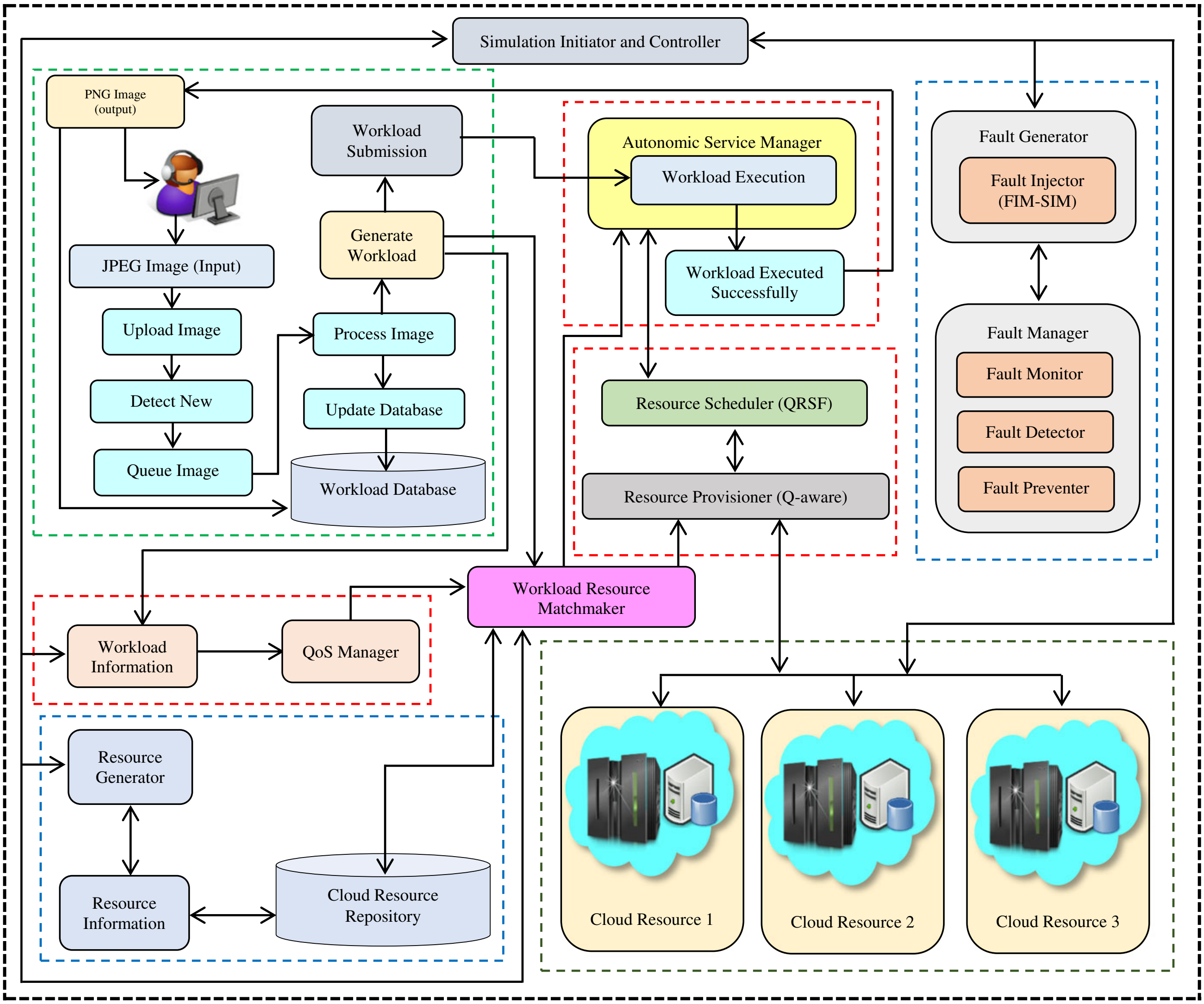

FIGURE 6 Cloud testbed

TABLE 8 Configuration details

Resource Configuration	Specifications	Operating System	Number of Virtual Node	Number of ECs	Price (C$/EC time unit)
Intel XEON E 52407-2.2 GHz	2 GB RAM and 160 GB HDD	Linux	2 (1 GB and 60GB)	6	4
Intel Core i5-2310-2.9GHz	4 GB RAM and 160 GB HDD	Linux	4 (1 GB and 40 GB)	12	3
Intel Core 2 Duo - 2.4 GHz	6 GB RAM and 320 GB HDD	Windows	6 (1 GB and 50 GB)	18	2

4.2 | Workloads

For performance evaluation, we have selected four different types of cloud workload from every cluster of workloads, as given in Table 3. Table 9 shows the different cloud workloads, which are considered to test the performance of RADAR. To find the experiment statistics, 3000 different workloads are executed. RADAR processes different workloads using the different number of resources to test its performance with different resource configuration. RADAR also maintains the details of every executed workload and stores into the workload database, which can be used to test the efficiency of RADAR in the future. Figure 6 shows the execution of the *"Performance Testing"* workload; similarly, we executed other workloads [(i) Storage and Backup Data, (ii) Websites, and (iii) Software Development and Testing] with the same experimental setup. *Note:* The detailed description of heterogeneous workloads is described in our previous research work.[3]

TABLE 9 Details of cloud workloads

Workload	Cluster	Description
Performance Testing	Compute (C1)	RADAR processes and converts an image file (713 MB) to PNG format from JPG format.[38,29] The conversion of a single JPG file into PNG is considered as a workload (in the form of Cloudlet).
Storage and Backup Data	Storage (C2)	Storing a larger amount of data (5 TB) and creating a backup of data[3] are considered as workload
Websites	Communication (C3)	A large number of users accessing a website of Thapar Institute of Engineering and Technology [http://www.thapar.edu] during admission period of the year 2016 are considered as workload.[3]
Software Development and Testing	Administration (C4)	Developed and tested Agri-Info Software to find out the productivity of a crop[39] is considered as a workload.

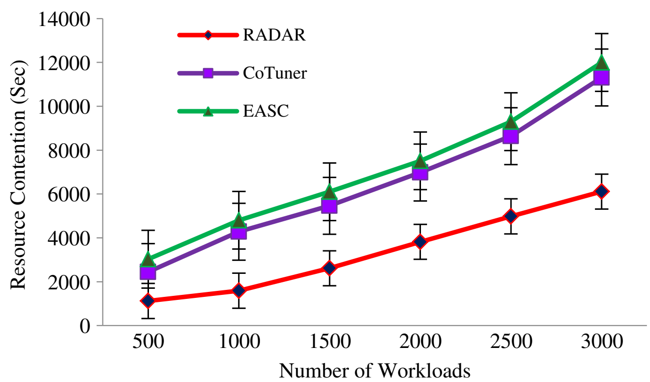

FIGURE 7 Resource contention vs the number of workloads

4.3 | Experimental results

RADAR has been verified for two aspects: (1) self-configuring and (2) self-healing. We selected existing resource management techniques from literature to test the performance of RADAR in the cloud environment. All the experiments have been conducted with 500 to 3000 workloads to validate RADAR. We have considered 1500 workloads to measure the performance of QoS parameters with fault percentage for both self-configuring and self-healing. *Fault percentage* is defined as the percentage of faults existing in the system during execution of workloads, and it is determined based on the SLA violation rate. To test the capability of RADAR for detection of the failures, FIM-SIM[36] is used to inject different percentages of faults (0% to 5%). We provided standard error bars for every graph to show the variation in the experimental results.

4.3.1 | Self-configuring verification

We have selected two existing autonomic resource management techniques, ie, EASC[20] and CoTuner,[25] to test the performance of RADAR in terms of resource contention, resource utilization, execution cost, and SLA violation rate. Both EASC[20] and CoTuner[25] have been discussed and compared with RADAR in Section 2.

Test Case 1: Resource Contention – Figure 7 shows the variation of resource contention for RADAR, EASC, and CoTuner with the increase in the number of workloads. As the number of workloads increases from 500 to 3000 workloads, the value of resource contention increases. The average value of resource contention in RADAR is 4.3% and 6.64% less than CoTuner and EASC, respectively. We have considered the six different values of fault percentage to test the performance of RADAR. Figure 8 shows the variation of resource contention for RADAR, EASC, and CoTuner with the different value of fault percentage (0% to 5%). The average value of resource contention in RADAR is 7.79% and 9.11% less than EASC and CoTuner, respectively. This is expected as the workload execution is done using RADAR, which is based on Q-aware.[27] Based on the deadline and priority of workload, clustering of workloads is performed, and resources are provisioned for effective scheduling. This is also because of the low variation in execution time across various resources as the resource list that is obtained from the resource provisioning unit is already filtered using Q-aware.[27]

Test Case 2: Resource Utilization – The value of Resource Utilization (RU) increases with the increase in the number of cloud workloads, as shown in Figure 9. The average value of RU in RADAR is 5.25% and 8.79% more than CoTuner and EASC, respectively. Figure 10 shows the variation of RU with respect the fault percentage. For fault percentage, the average value of RU in RADAR is 4.16% and 6.93% more than CoTuner and EASC, respectively. Based on the QoS requirements of a specific workload, resource provisioning consumes little more time to find out the best resources,[29] but later, it increases the overall performance of RADAR. Therefore, underutilization and overutilization of resources will be assuaged or avoided, which reduces the further queuing time.

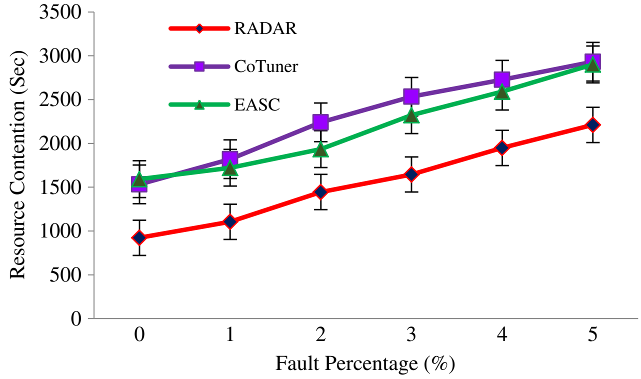

FIGURE 8 Resource contention vs fault percentage

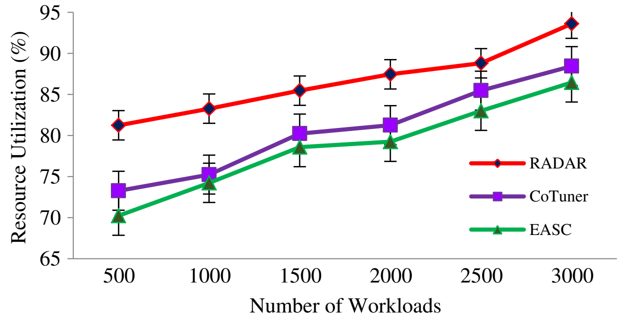

FIGURE 9 Resource utilization vs the number of workloads

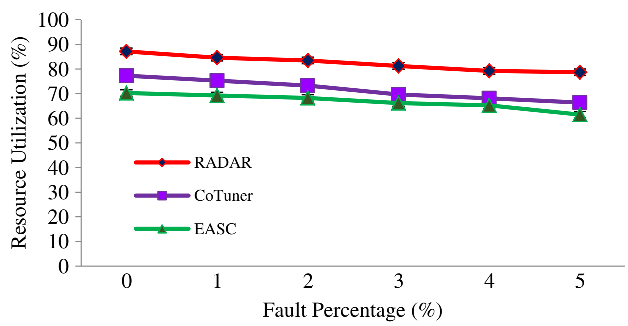

FIGURE 10 Resource utilization vs fault percentage

Test Case 3: Execution Cost – The execution cost rises with the increase in the number of workloads, as shown in Figure 11. The average value of execution cost in RADAR is 3.33% and 5.83% less than CoTuner and EASC, respectively. Figure 12 shows the variation of execution cost with the different values of fault percentage and the average value of execution cost in RADAR is 4.98% and 6.80% less than CoTuner and EASC, respectively. The reason is that CoTuner and EASC do not consider the effect of other workloads in the resource scheduler at the time of workload submission, but in RADAR, the resource manager considers the effect of workloads in the resource scheduler before execution of workload according to both user and resource provider's perspectives.[27] The other reason is that, with the provisioned approach (Q-aware), due to a large number of workloads, these and later workloads had to be executed on left out resources, which may not be very cost effective.

Test Case 4: SLA Violation Rate – The impact of variation in the number of workloads on SLA Violation Rate (SVR) is analyzed. As shown in Figure 13, SVR increases with the increase in the number of workloads and it shows that the average value of SVR in RADAR is 8.16% and 14.98% less than CoTuner and EASC, respectively. Figure 14 shows the variation of SVR with different values of fault percentage, and RADAR has 5.56% and 6.16% less SVR than CoTuner and EASC, respectively. This is because, RADAR uses admission control and reserve resources for execution of workloads in advance based on their QoS requirements specified in the SLA document. Further, RADAR outperforms as it regulates the resources at runtime based on the user's new QoS requirements during its execution to avoid SLA violation.

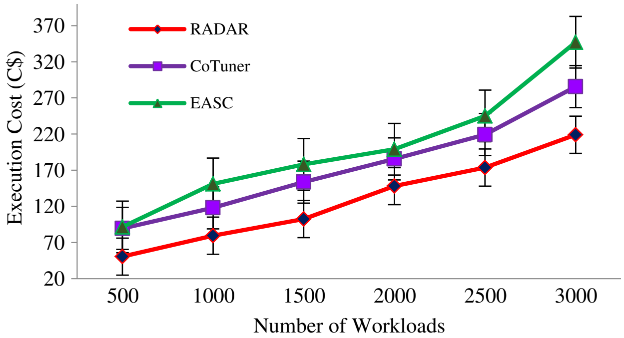

FIGURE 11 Execution cost vs the number of workloads

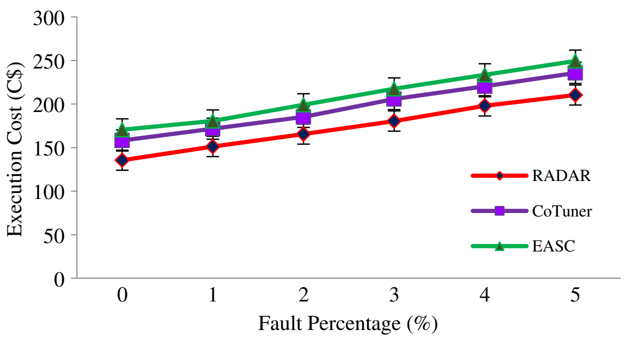

FIGURE 12 Execution cost vs fault percentage

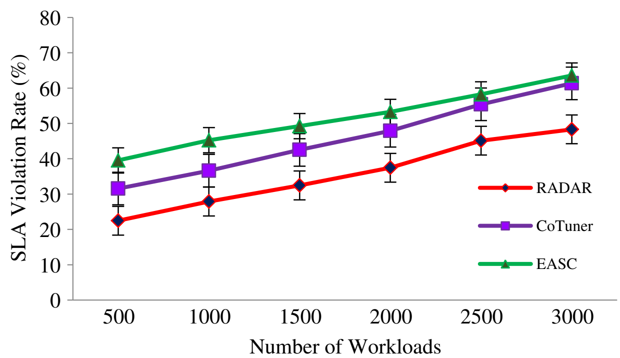

FIGURE 13 SLA violation rate vs the number of workloads

4.3.2 | Self-healing verification

We have selected two existing autonomic resource management techniques, ie, SH-SLA[5] and MASA,[14] to test RADAR's performance in terms of reliability, fault detection rate, throughput, waiting time, execution time, energy consumption, availability, and turnaround time. Both SH-SLA[5] and MASA[14] have been discussed and compared with RADAR in Section 2.

Test Case 1: Fault Detection Rate – Figure 15 shows the variation of Fault Detection Rate (FDR) with the different number of workloads. With an increase in the number of workloads, the value of FDR decreases. From 500 to 1500 cloud workloads, the value of FDR reduces, but RADAR performs better than SH-SLA and MASA. The average value of FDR in RADAR is 13.72% and 16.88% more than SH-SLA and MASA, respectively. RADAR uses the hardware hardening process to decrease the frequency of fault occurrence and it hardens the device drivers by using the concept of Carburizer,[34] which keeps system's working without degradation of performance even though faults occur.[33] Once the hardening process is over, the status of nodes is forwarded to the monitor component (autonomic service manager) to prevent future faults. To avoid the same kind of future faults, RADAR replaces the hardened driver with an original driver if an alert is generated because of the driver's misbehavior.

Test Case 2: Throughput – Figure 16 shows the variation of throughput with the different number of workloads. The average value of throughput in RADAR is 11.1% and 14.50% more than SH-SLA and MASA, respectively. We have injected a number of faults (fault percentage) to verify the throughput of RADAR with 1500 workloads. Figure 17 shows the comparison of throughput of RADAR with SH-SLA and MASA. From the experimental result, it has been found that the maximum throughput is at 0% fault percentage and the minimum is at 5%. The average value of throughput in

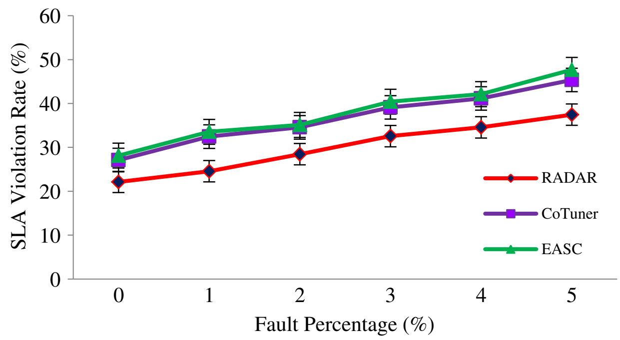

FIGURE 14 SLA violation rate vs fault percentage

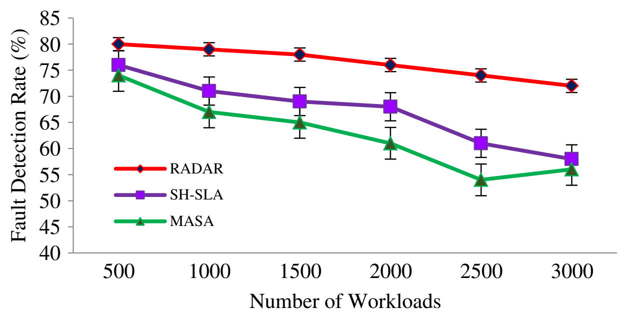

FIGURE 15 Fault detection rate vs the number of workloads

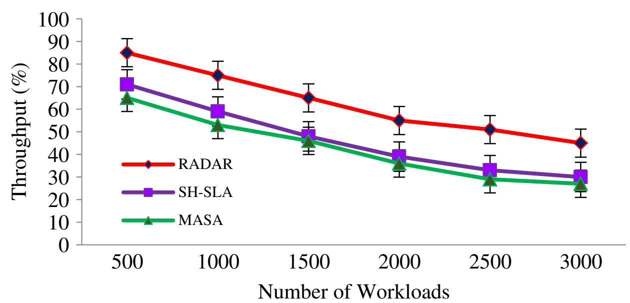

FIGURE 16 Throughput vs the number of workloads

RADAR is 8.33% and 10.51% more throughput than SH-SLA and MASA, respectively. RADAR identifies the software, hardware, and network faults automatically and it also prevents the system from future faults, which improves the throughput of RADAR, as compared with SH-SLA and MASA.

Test Case 3: Reliability – The value of reliability decreases with variation in the number of workloads, but RADAR performs better than SH-SLA and MASA, as shown in Figure 18. The maximum reliability at 500 workloads is 91.45%. The value of reliability in RADAR is 8.32% and 11.23% more than SH-SLA and MASA, respectively. Figure 19 shows the variation of reliability with a different value of fault percentage, and the value of reliability in RADAR is 13.26% and 14.31% more than SH-SLA and MASA, respectively. The main difference between these works (SH-SLA and MASA) and ours (RADAR) is that none of them considers an autonomic fault prevention mechanism for self-healing with dynamic checkpoint intervals, which uses RADAR. The SH-SLA and MASA techniques focus on monitoring faults to maximize fault detection rate, whereas RADAR focuses on fault detection, as well as prevention mechanisms. The efficient management of faults in RADAR improves the reliability of cloud services.

Test Case 4: Availability – The value of availability for RADAR, SH-SLA and MASA is calculated and the value of availability decreases with increase in the number of workloads, as shown in Figure 20. The average value of availability in RADAR is 5.23% and 5.96% more than SH-SLA and MASA respectively. Figure 21 shows the variation of availability with the different value of fault percentage and the value of availability in RADAR is 3.45% and 4.46% more than SH-SLA and MASA respectively. This is expected as the recovering faulty task manages the faults efficiently in RADAR, which further improves the availability of cloud services.

Test Case 5: Execution Time – As shown in Figure 22, the execution time increases with the increase in the number of workloads. The value of execution time in RADAR is 6.15% and 6.95% less than SH-SLA and MASA, respectively. After 1500 workloads, execution time increases suddenly but RADAR produces better outcomes than SH-SLA and MASA. At 3000 workloads, the execution time in RADAR is 11.11% and 21.23% less than SH-SLA and MASA, respectively. Figure 23 shows the variation of execution time with the different value of fault percentage, and the value of execution time in RADAR is 10.23% and 19.98% less than SH-SLA and MASA, respectively. This is expected as RADAR is keeping track of the

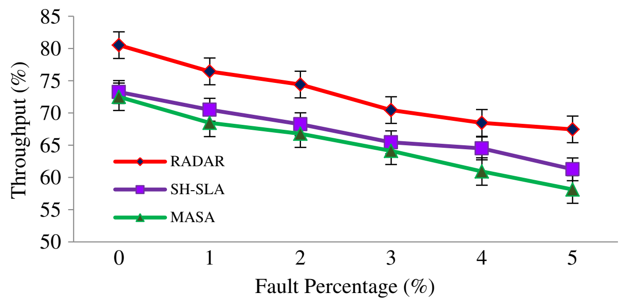

FIGURE 17 Throughput vs fault percentage

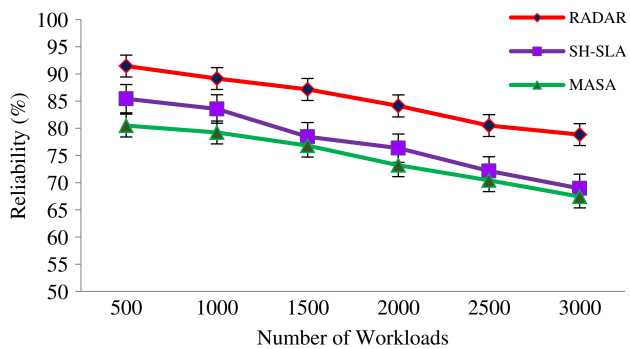

FIGURE 18 Reliability vs the number of workloads

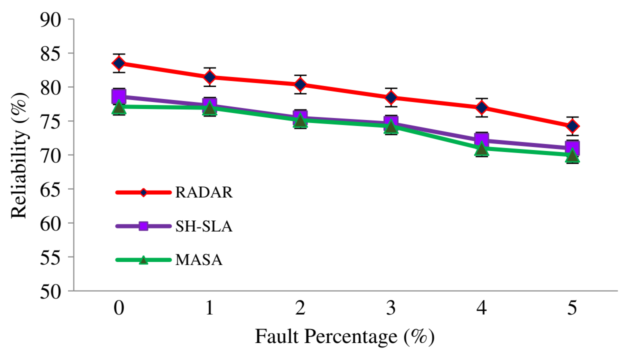

FIGURE 19 Reliability vs fault percentage

state of all resources at each point of the time automatically, which enables it to take an optimal decision (minimum execution time) than SH-SLA and MASA.

Test Case 6: Energy Consumption – With the different number of cloud workloads, energy consumption is calculated in kilo Watt hour (kWh) for RADAR, SH-SLA, and MASA, as shown in Figure 24. Energy consumption also increases, with the increasing number of cloud workloads. The minimum value of energy consumption is 46.12 kWh at 500 workloads. The average value of energy consumption in RADAR is 8.11% and 9.73% less than SH-SLA and MASA, respectively. Figure 25 shows the variation of energy consumption with the different value of fault percentage. Energy consumption in RADAR is 5.89% and 11.25% less than SH-SLA and MASA, respectively. With the capability of automatically turning on and off nodes according to demands, RADAR provisions and schedules resources efficiently and intelligently for execution of clustered workloads instead of individual workloads. Further, workload clustering reduces the significant amount of network traffic, which leads to reducing the number of active switches that also reduces the wastage of energy.

Test Case 7: Waiting Time – The performance of RADAR in terms of waiting time with the different number of workloads is verified. Figure 26 shows the variation of waiting time with the different number of workloads and the value of waiting time in RADAR is 15.22% and 19.75% less than SH-SLA and MASA, respectively. The variation of waiting time with the different value of fault percentage is shown in Figure 27 and the value of waiting time in RADAR is 8.33% and 9.96% less than SH-SLA and MASA, respectively. The cause is that RADAR adjusts the provisioned resources dynamically according to the QoS requirements of workload to fulfill their required deadline, which also reduces the waiting time of workload in a queue.

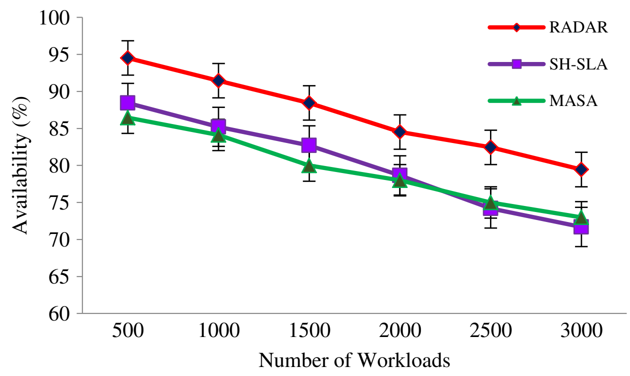

FIGURE 20 Availability vs the number of workloads

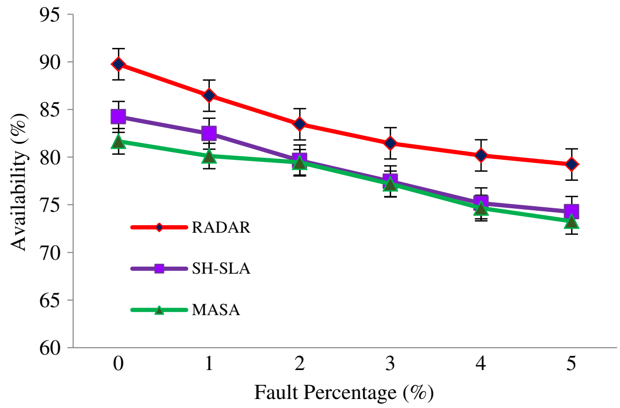

FIGURE 21 Availability vs fault percentage

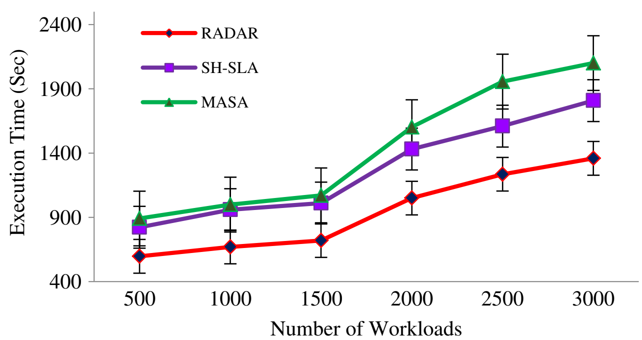

FIGURE 22 Execution time vs the number of workloads

Test Case 8: Turnaround Time – The value of turnaround time in RADAR with the different number of workloads is shown in Figure 28. The average value of turnaround time in RADAR is 13.33% and 17.45% less than SH-SLA and MASA, respectively. Figure 29 shows the variation of turnaround time with the different value of fault percentage and the average value of turnaround time in RADAR is 7.14% and 12.75% less than SH-SLA and MASA, respectively. SH-SLA and MASA consider scheduling of single queued workload, whereas RADAR considers clustering of workloads based on their QoS requirements. This is also because RADAR is keeping track of the state of all resources at each point of the time automatically, which enables it to take an optimal decision (minimum execution time) than SH-SLA and MASA.

4.4 | Statistical analysis

Statistical significance of the results has been analyzed by Coefficient of Variation (CoV), which compares the different means and furthermore offers an overall analysis of the performance of RADAR used for creating the statistics. It states the deviation of the data as a proportion of its average value[26] and is calculated as follows [Equation (33)]:

$$\mathrm{CoV} = \frac{SD}{M} \times 100, \tag{33}$$

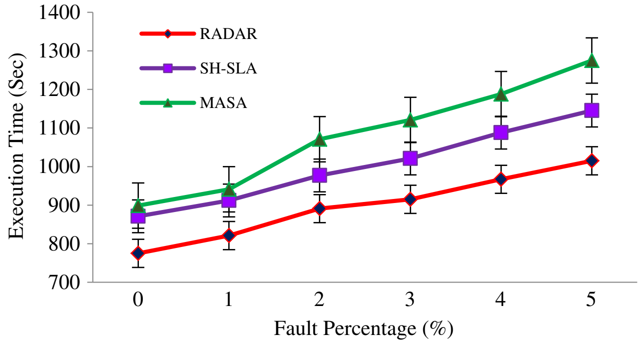

FIGURE 23 Execution time vs fault percentage

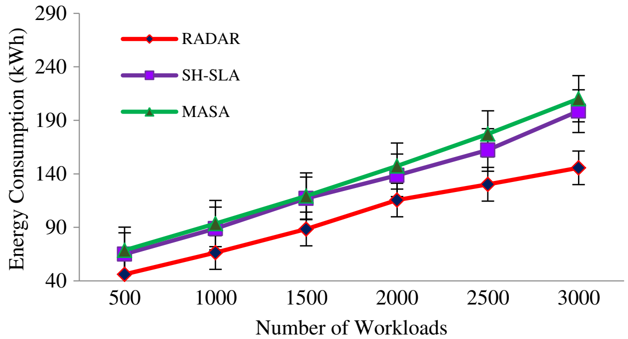

FIGURE 24 Energy consumption vs the number of workloads

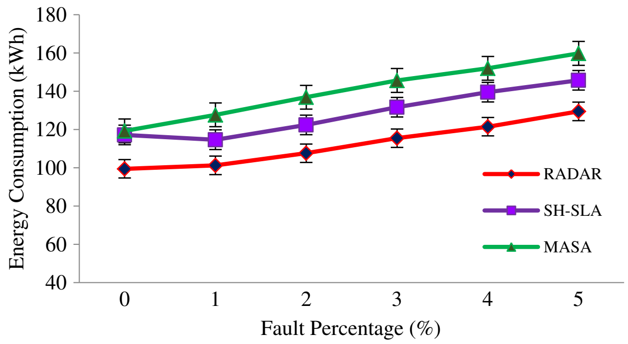

FIGURE 25 Energy consumption vs fault percentage

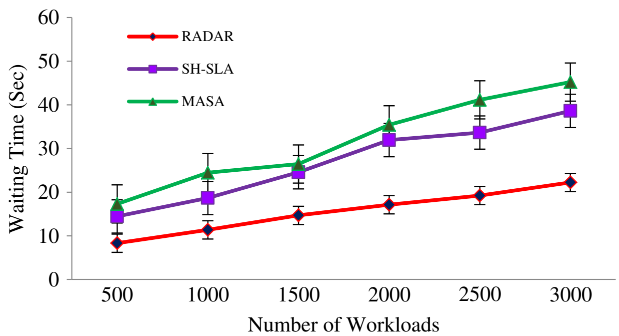

FIGURE 26 Waiting time vs the number of workloads

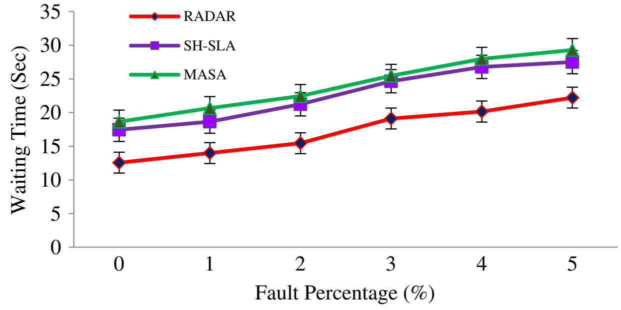

FIGURE 27 Waiting time vs fault percentage

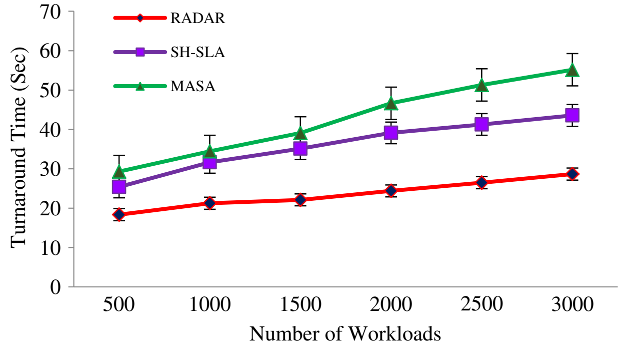

FIGURE 28 Turnaround time vs the number of workloads

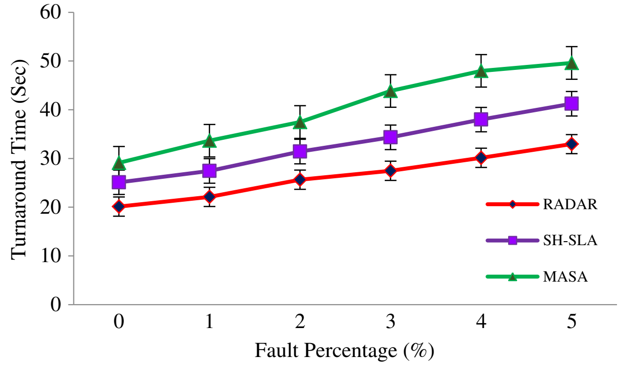

FIGURE 29 Turnaround time vs fault percentage

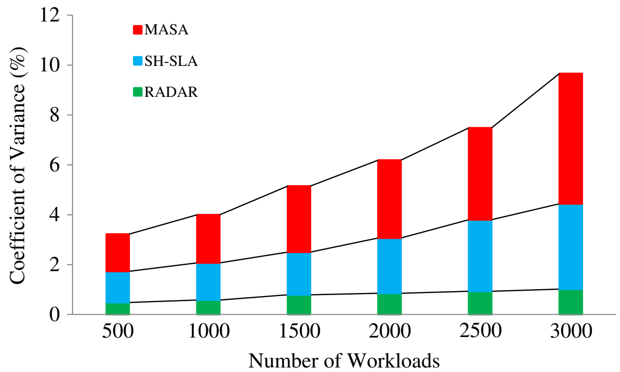

FIGURE 30 Coefficient of variation for fault detection rate

where *SD* is the Standard Deviation and *M* is the Mean. CoV of fault detection rate of RADAR, SH-SLA, and MASA is shown in Figure 30. The range of CoV (0.48% - 1.02%) for fault detection rate approves the stability of RADAR.

TABLE 10 Variation of QoS parameters for workload distribution on different numbers of resources

Number of Workloads	Number of Resources R1	R2	R3	Total Virtual Nodes	Execution Time (Sec)	Execution Cost (C$)	Waiting Time (Sec)	Fault Detection Rate (%)	Throughput (%)	Resource Utilization (%)	Energy Consumption (kWh)	Resource Contention (Sec)	SLA Violation Rate (%)	Turnaround Time (Sec)	Reliability (%)	Availability (%)
1500	1	0	0	1	422.2	65.71	8.72	85.23	82.22	78.26	18.56	2197	10.33	60.25	68.78	72.56
1500	1	1	0	2	418.36	68.12	8.42	86.12	83.15	79.5	19.5	2143	10.03	61.65	70.05	74.45
1500	2	1	0	3	412.65	70.12	7.91	86.88	84.66	81.62	22.61	2064	9.35	63.35	73.15	77.12
1500	2	2	0	4	407.98	71.65	7.23	87.14	85.17	81.99	25.15	1927	9.02	63.93	77.14	81.62
1500	3	2	0	5	401.15	72.99	6.69	87.59	86.22	82.45	26.99	1847	8.55	65.25	79.98	83.45
1500	4	2	0	6	393.93	73.45	6.01	88.01	87.59	83.26	28.45	1768	8.13	67.18	80.05	84.62
1500	4	2	1	7	381.65	75.64	5.70	89.5	88.23	84.76	29.55	1601	7.94	69.22	81.25	85.96
1500	4	3	1	8	375.15	76.25	4.91	89.99	88.92	85.65	30.15	1498	7.71	70.12	82.14	86.12
1500	5	3	1	9	367.81	77.18	4.20	90.83	89.12	86.11	31.28	1365	7.41	71.56	84.56	86.85
1500	5	3	2	10	359.74	79.98	3.61	92.22	89.95	86.89	32.75	1241	7.23	71.99	86.91	87.77
1500	5	4	2	11	354.23	80.93	2.99	93.49	90.56	87.87	34.15	1205	7.01	72.05	87.25	88.25
1500	6	4	2	12	342.2	81.66	2.46	94.45	91.62	88.65	35.97	1120	6.91	72.17	88.71	90.12
3000	1	0	0	1	811.23	92.66	18.11	83.25	81.31	81.12	45.61	4236	16.17	76.26	65.71	52.65
3000	1	1	0	2	845.26	98.11	16.91	84.25	81.85	81.95	47.26	4012	15.25	76.59	67.18	56.45
3000	2	1	0	3	899.75	103.98	16.01	85.41	82.16	82.45	53.26	3883	14.93	77.18	69.37	58.72
3000	2	2	0	4	955.65	107.45	15.26	85.98	82.98	82.88	57.45	3571	14.21	78.01	70.85	63.35
3000	3	2	0	5	988.45	116.78	14.93	86.56	83.56	83.61	58.61	3243	13.55	79.50	71.12	66.78
3000	4	2	0	6	1020.20	222.12	14.20	87.45	84.45	84.77	62.45	2961	13.12	80.12	72.56	69.92
3000	4	2	1	7	1060.38	227.93	13.59	88.78	85.69	85.12	66.98	2719	12.89	81.66	75.45	73.26
3000	4	3	1	8	1110.23	232.54	12.22	89.12	86.33	85.91	70.44	2465	11.90	82.82	78.06	76.66
3000	5	3	1	9	1170.65	239.16	10.95	90.25	86.91	86.03	73.36	2211	11.11	83.19	80.15	80.12
3000	5	3	2	10	1225.65	242.79	9.11	91.06	87.21	86.75	76.95	2105	10.26	83.97	81.39	83.45
3000	5	4	2	11	1301.85	245.45	7.52	92.45	88.01	87.22	78.78	2022	10.05	84.15	83.06	85.50
3000	6	4	2	12	1445.65	147.65	6.78	93.56	89.78	87.98	80.42	1968	9.94	84.68	84.69	87.45

The value of CoV increases as the number of workloads increases. The small value of CoV signifies that RADAR is more efficient and stable in the management of cloud resources in those circumstances, where the quantity of workloads is varying. RADAR attained better results in the cloud for fault detection rate and has been exhibited with respect to the number of workloads.

Table 10 describes the comparison of different QoS parameters, which are measured while the different number of resources/VMs are used to process various heterogeneous workloads (1500 and 3000). It shows that RADAR optimizes the QoS parameters with increase in the number of VMs gradually. RADAR executes 1500 workloads with one virtual node running on Server R1, and its execution completed in 422.2 seconds whereas the same number of workloads finished in 342.6 seconds using 12 virtual nodes (2 virtual nodes running on R3, 4 virtual nodes running on R2, and 6 virtual nodes running on R1). Table 10 clearly shows that execution time decreases by adding additional virtual nodes. Similarly, by increasing the number of virtual nodes, other QoS parameters are also performing better.

5 | CONCLUSIONS AND FUTURE WORK

We have proposed an autonomic technique for cloud-based resources called RADAR to efficiently manage the provisioned cloud resources. RADAR improves user satisfaction by maintaining SLA based on QoS requirements of a cloud user and has an ability to manage resources automatically through properties of self-management, which are self-healing (find and react to sudden faults) and self-configuring (capability to readjust resources) with minimum human intervention. The performance of RADAR has been evaluated using the CloudSim toolkit and the experimental results show that RADAR performs better in terms of different QoS parameters and deals with software, network, and hardware faults, as compared with existing resource management techniques, as shown in test cases. Experimental results demonstrate that RADAR improves the fault detection rate by 16.88%, resource utilization by 8.79%, throughput by 14.50%, availability by 5.96%, and reliability by 11.23% and it reduces the resource contention by 6.64%, SLA violation rate by 14.98%, execution time by 6.95%, energy consumption by 9.73%, waiting time by 19.75%, turnaround time by 17.45%, and execution cost by 5.83%, as compared with resource management techniques. The small value of Coefficient of Variation (CoV) signifies that RADAR is more efficient and stable in the management of cloud resources in those circumstances, where the quantity of workloads is varying. RADAR improves user satisfaction by fulfilling their QoS requirements and increases reliability and availability of cloud-based services. Further, RADAR can be extended to exhibit the other property of self-management such as self-protecting (detection and protection of cyber-attacks).

ACKNOWLEDGMENTS

One of the authors, Dr. Sukhpal Singh Gill [Postdoctoral Research Fellow], gratefully acknowledges Discovery Project of Australian Research Council (ARC) (grant no. DP160102414), The University of Melbourne, Australia, for awarding him the Fellowship to carry out this research work. We would like to thank all the anonymous reviewers for their valuable suggestions. We thank Professor Rao Kotagiri, Adel N. Toosi, Bowen Zhou, Manmeet Singh, and Amanpreet Singh for their comments on improving the paper.

ORCID

Sukhpal Singh Gill http://orcid.org/0000-0002-3913-0369

REFERENCES

1. Singh S, Chana I. QoS-aware autonomic resource management in cloud computing: a systematic review. *ACM Comput Surv*. 2015;48(3):1-46.
2. Singh S, Chana I, Buyya R. STAR: SLA-aware autonomic management of cloud resources. *IEEE Trans Cloud Comput*. 2017;4:1-14. https://doi.org/10.1109/TCC.2017.2648788
3. Singh S, Chana I, Singh M, Buyya R. SOCCER: self-optimization of energy-efficient cloud resources. *Clust Comput*. 2016;19(4):1787-1800.
4. Chen G, Jin H, Zou D, Zhou BB, Qiang W, Hu G. SHelp: automatic self-healing for multiple application instances in a virtual machine environment. Paper presented at: 2010 IEEE International Conference on Cluster Computing; 2010; Heraklion, Greece.
5. Mosallanejad A, Atan R, Murad MA, Abdullah R. A hierarchical self-healing SLA for cloud computing. *Int J Digit Inf Wirel Commun*. 2014;4(1):43-52.
6. Mosallanejad A, Atan R, Abdullah R, Murad MA, Javdani T. HS-SLA: a hierarchical self-healing SLA model for cloud computing. Paper presented at: The Second International Conference on Informatics Engineering & Information Science; 2013; Kuala Lumpur, Malaysia.
7. Alhosban A, Hashmi K, Malik Z, Medjahed B. Self-healing framework for cloud-based services. Paper presented at: 2013 ACS International Conference on Computer Systems and Applications (AICCSA); 2013; Ifrane, Morocco.
8. da Silva RF, Glatard T, Desprez F. Self-healing of operational workflow incidents on distributed computing infrastructures. Paper presented at: 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing; 2012; Ottawa, Canada.
9. Glatard T, Lartizien C, Gibaud B, et al. A virtual imaging platform for multi-modality medical image simulation. *IEEE Trans Med Imaging*. 2013;32(1):110-118.
10. Li W, Zhang P, Yang Z. A framework for self-healing service compositions in cloud computing environments. Paper presented at: 2012 IEEE 19th International Conference on Web Services; 2012; Honolulu, HI.
11. Magalhaes JP, Silva LM. A framework for self-healing and self-adaptation of cloud-hosted web-based applications. Paper presented at: 2013 IEEE 5th International Conference on Cloud Computing Technology and Science; 2013; Bristol, UK.

12. Xin R. Self-healing cloud applications. Paper presented at: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST); 2016; Chicago, IL.
13. Rios E, Iturbe E, Palacios MC. Self-healing multi-cloud application modelling. In: Proceedings of the 12th International Conference on Availability, Reliability and Security; 2017; Reggio Calabria, Italy.
14. Azaiez M, Chainbi W. A multi-agent system architecture for self-healing cloud infrastructure. In: Proceedings of the International Conference on Internet of Things and Cloud Computing; 2016; Cambridge, UK.
15. de Sá AS, Macêdo RJdA. QoS self-configuring failure detectors for distributed systems. In: *Distributed Applications and Interoperable Systems: 10th IFIP WG 6.1 International Conference, DAIS 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings*. Vienna, Laxenburg: International Federation for Information Processing; 2010:126-140.
16. Maurer M, Brandic I, Sakellariou R. Adaptive resource configuration for cloud infrastructure management. *Futur Gener Comput Syst*. 2013;29(2):472-487.
17. Salaün G, Etchevers X, De Palma N, Boyer F, Coupaye T. Verification of a self-configuration protocol for distributed applications in the cloud. In: *Assurances for Self-Adaptive Systems: Principles, Models, and Techniques*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2013:60-79.
18. Etchevers X, Coupaye T, Boyer F, de Palma N. Self-configuration of distributed applications in the cloud. Paper presented at: 2011 IEEE 4th International Conference on Cloud Computing; 2011; Washington, DC.
19. Panica S, Neagul M, Craciun C, Petcu D. Serving legacy distributed applications by a self-configuring cloud processing platform. In: Proceedings of the 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems; 2011; Prague, Czech Republic.
20. Wolinsky DI, Liu Y, St Juste P, Venkatasubramanian G, Figueiredo R. On the design of scalable, self-configuring virtual networks. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis; 2009; Portland, OR.
21. Sabatucci L, Lopes S, Cossentino M. A goal-oriented approach for self-configuring mashup of cloud applications. Paper presented at: 2016 International Conference on Cloud and Autonomic Computing (ICCAC); 2016; Augsburg, Germany.
22. Sabatucci L, Lopes S, Cossentino M. Self-configuring cloud application mashup with goals and capabilities. *Clust Comput*. 2017;20(3):2047-2063.
23. Lama P, Zhou X. AROMA: automated resource allocation and configuration of MapReduce environment in the cloud. In: Proceedings of the 9th International Conference on Autonomic Computing; 2012; San Jose, CA.
24. Konstantinou I, Kantere V, Tsoumakos D, Koziris N. COCCUS: self-configured cost-based query services in the cloud. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data; 2013; New York, NY.
25. Bu X, Rao J, Xu C-Z. Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. *IEEE Trans Parallel Distrib Syst*. 2013;24(4):681-690.
26. Gill SS, Chana I, Singh M, Buyya R. CHOPPER: an intelligent QoS-aware autonomic resource management approach for cloud computing. *Clust Comput*. 2017:1-39. https://doi.org/10.1007/s10586-017-1040-z
27. Singh S, Chana I. Q-aware: quality of service based cloud resource provisioning. *Comput Electr Eng*. 2015;47:138-160. https://doi.org/10.1016/j.compeleceng.2015.02.003
28. Singh S, Chana I. QRSF: QoS-aware resource scheduling framework in cloud computing. *J Supercomput*. 2015;71(1):241-292.
29. Gill SS, Buyya R, Chana I, Singh M, Abraham A. BULLET: particle swarm optimization based scheduling technique for provisioned cloud resources. *J Netw Syst Manag*. 2018;26(2):361-400.
30. Braun TD, Siegel HJ, Beck N, et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *J Parallel Distrib Comput*. 2001;61(6):810-837.
31. Ali S, Siegel HJ, Maheswaran M, Hensgen D, Ali S. Representing task and machine heterogeneities for heterogeneous computing systems. *Tamkang J Sci Eng*. 2000;3(3):195-207.
32. Gill SS, Buyya R. Resource provisioning based scheduling framework for execution of heterogeneous and clustered workloads in clouds: from fundamental to autonomic offering. *J Grid Comput*. 2018:1-33. https://doi.org/10.1007/s10723-017-9424-0
33. Chopra I, Singh M. SHAPE–an approach for self-healing and self-protection in complex distributed networks. *J Supercomput*. 2014;67(2):585-613
34. Kadav A, Renzelmann MJ, Swift MM. Tolerating hardware device failures in software. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles; 2009; Big Sky, MT.
35. Calheiros RN, Ranjan R, Beloglazov A, De Rose CAF, Buyya R. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw Pract Exp*. 2011;41(1):23-50.
36. Nita M-C, Pop F, Mocanu M, Cristea V. FIM-SIM: fault injection module for CloudSim based on statistical distributions. *J Telecommun Inf Technol*. 2014;4(4):14-23.
37. Liu J, Wu Z, Wu J, Dong J, Zhao Y, Wen D. A Weibull distribution accrual failure detector for cloud computing. *PLOS One*. 2017;12(3):e0173666.
38. Thieling L, Schüer A, Hartung G, Büchel G. Embedded image processing system for cloud-based applications. Paper presented at: 2014 International Conference on Systems, Signals and Image Processing (IWSSIP); 2014; Dubrovnik, Croatia.
39. Gill SS, Chana I, Buyya R. IoT based agriculture as a cloud and big data service: the beginning of digital India. *J Organ End User Comput*. 2017;29(4):1-23.

How to cite this article: Gill SS, Chana I, Singh M, Buyya R. RADAR: Self-configuring and self-healing in resource management for enhancing quality of cloud services. *Concurrency Computat Pract Exper*. 2019;31:e4834. https://doi.org/10.1002/cpe.4834