Alahmad and Agarwal
Journal of Cloud Computing (2024) 13:46
https://doi.org/10.1186/s13677-024-00610-2

Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH

Open Access

Multiple objectives dynamic VM placement for application service availability in cloud networks

Yanal Alahmad[1*†] and Anjali Agarwal[2†]

Abstract

Ensuring application service availability is a critical aspect of delivering quality cloud computing services. However, placing virtual machines (VMs) on computing servers to provision these services can present significant challenges, particularly in terms of meeting the requirements of application service providers. In this paper, we present a framework that addresses the NP-hard dynamic VM placement problem in order to optimize application availability in cloud computing paradigm. The problem is modeled as an integer nonlinear programming (INLP) optimization with multiple objectives and constraints. The framework comprises three major modules that use optimization methods and algorithms to determine the most effective VM placement strategy in cases of application deployment, failure, and scaling. Our primary goals are to minimize power consumption, resource waste, and server failures while also ensuring that application availability requirements are met. We compare our proposed heuristic VM placement solution with three related algorithms from the literature and find that it outperforms them in several key areas. Our solution is able to admit more applications, reduce power consumption, and increase CPU and RAM utilization of the servers. Moreover, we use a deep learning method that has high accuracy and low error loss to predict application task failures, allowing for proactive protection actions to reduce service outage. Overall, our framework provides a comprehensive solution by optimizing dynamic VM placement. Therefore, the framework can improve the quality of cloud computing services and enhance the experience for users.

Keywords VM placement, Task scheduling, Application availability, Deep learning, Cloud computing, AntColony

Introduction

Cloud computing has emerged as a popular paradigm that offers Application Service Providers (ASPs) such as Netflix and Spotify the ability to leverage a pool of virtual infrastructure resources for hosting their applications. By accessing resources from Cloud Service Providers (CSPs) based on workload demands, ASPs are able to realize the pay-as-you-go business model where they only pay for resources they use. This cost-effectiveness has encouraged many ASPs to migrate their applications to the cloud. However, despite its many advantages, cloud computing presents quality of service (QoS) challenges that have become top priorities for ASPs. In particular, service availability has emerged as a key non-functional requirement, denoting the percentage of time a service is available to users [1]. Some users demand highly available (HA) services, with a ratio of 99.999% (aka five nines) or more of the time the service is available being the gold standard for service HA [1]. Other users require service continuity in order to resume the service from its last state before interruption. CSPs are responsible for

†Yanal Alahmad and Anjali Agarwal contributed equally to this work.

*Correspondence:
Yanal Alahmad
yanal.alahmad@concordia.ca
[1] Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada
[2] Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada

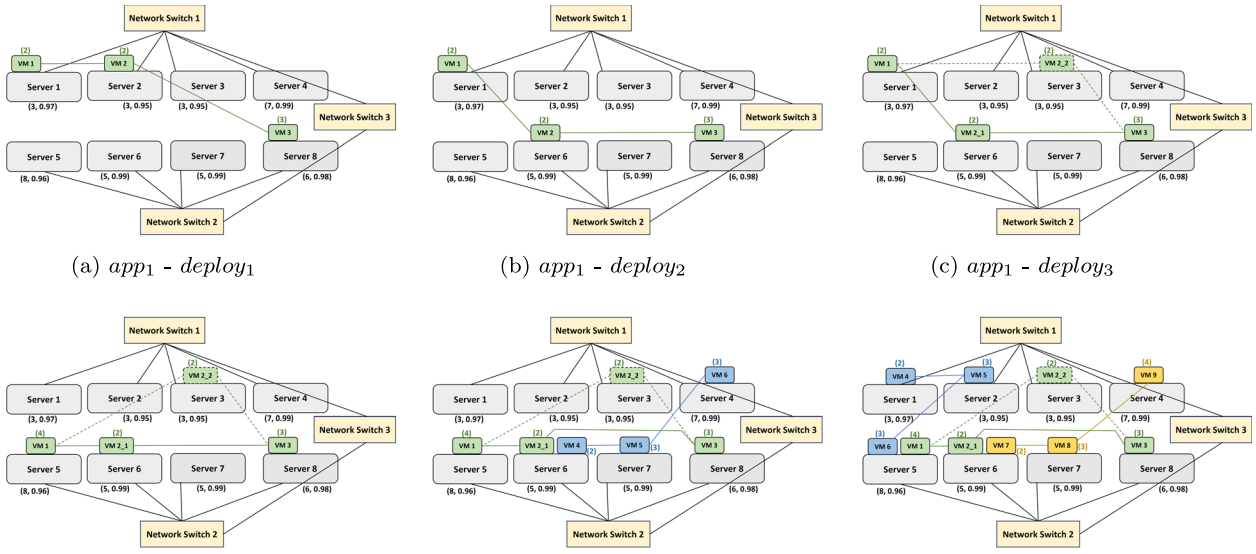
ensuring application service availability in accordance with the requirements laid out in Service Level Agreements (SLAs), which detail QoS expectations. Providing availability that is lower than the demand can have a significant negative impact on service performance and quality, resulting in significant losses, with downtime in web applications costing businesses up to $300,000 per hour [2]. Conversely, providing availability above the demand can raise costs and reduce the admissibility of new applications, lowering CSP profits. Therefore, effectively managing service availability is critical for both ASPs and CSPs in order to balance the service quality with profitability.

Managing service availability in the cloud is a complex task that comes with several challenges. Applications in the cloud are composed of software components hosted on virtual computing nodes such as Virtual Machines (VMs) or containers. These components depend on physical computing nodes, also known as servers, to host them. Due to this heterogeneous stack dependency, any failure in any layer can cause a service outage at any time. Detecting and quickly recovering from a resource failure requires an efficient monitoring and management mechanism. Moreover, cloud environments are highly dynamic, with resources being frequently added or removed on the fly. Failure to provide required resources at the right time can compromise service availability and quality. On the other hand, keeping extra resources for an extended period can increase operational and maintenance costs.

Ensuring high service availability in the cloud is not an easy task that requires a careful balance between cost and performance. Numerous approaches have been proposed to address this challenge. Reactive solutions focus on addressing outages as they occur by using redundancy models to failover to standby resources when an active resource fails. While effective, this approach can be costly due to the need for additional standby resources. Proactive solutions aim to predict and prevent service failure through prediction methods and protective actions. This approach can be cost-effective but relies heavily on the accuracy of the prediction method. Protection mechanisms can also increase the availability of application components, such as using a strong VM placement strategy. However, the proximity of VMs providing the same service can lower the overall availability level of the service, making VM placement a significant challenge. Clustering VMs together can help to reduce the total number of computing servers required to host the VMs, leading to reduced energy consumption and associated costs. However, placing active and standby VMs responsible for a specific service instance on the same server can result in a service outage in the event of server failure. Alternatively, distributing VMs across servers can improve workload balancing and server performance, but may increase the number of active computing servers and associated costs. Thus, choosing the optimal placement strategy for VMs is critical to maintaining high service availability in the cloud.

The management of virtual machine (VM) placement in cloud computing presents a challenging combinatorial problem that is NP-hard. Static VM placement is only suitable when a new VM is requested, while dynamic placement involves changes to the location of VMs, triggered at any time for any reason such as elasticity and migration. Achieving multiple objectives through VM placement can further complicate the problem, which grows exponentially with the number of VMs. In this paper, we introduce the "Multiple-Objectives Dynamic VM Placement for Application Availability in Cloud" (MoVPAAC) framework, which focuses on ensuring application service availability and optimizing resource usage. The framework comprises various modules that use a set of optimization solutions to handle dynamic VM placement during deployment, scaling, and application failure, with the aim of meeting availability requirements and achieving multiple objectives. The following are the main contributions of this research work:

- Introducing a formal definition for application service availability in cloud computing platforms, enabling better management and optimization of cloud resources.
- The proposed Multiple-Objectives Dynamic VM Placement for Application Availability in Cloud (MoVPAAC) framework is a novel approach to dynamic VM placement that integrates several optimization goals, including minimizing power consumption and resource waste, and maximizing service uptime, while ensuring high application availability.
- To tackle the complex multi-objective dynamic VM placement problem, this work formulates an integrated non-linear programming (INLP) model with a set of constraints, which can efficiently handle multiple optimization goals and complex requirements.
- The Ant Colony heuristic algorithm and VM protection method are employed in tandem to solve the INLP model, providing an effective approach to dynamic VM placement that is highly efficient and robust.
- To enhance the accuracy of failure prediction and protective actions, the deep learning Artificial Neural Network (ANN) method is utilized, delivering highly accurate results and enabling proactive protection of cloud applications and services.



(a) app_1 - $deploy_1$ (b) app_1 - $deploy_2$ (c) app_1 - $deploy_3$

(d) app_1 - $deploy_4$ (e) app_1, app_2 - $deploy$ (f) app_1, app_2, app_3 - $deploy$

Fig. 1 Applications Deployment Model in Cloud Data Center

- This work develops a prototype that showcases the proposed framework, algorithms, and methods, and provides a comparative analysis of the results against three existing VM placement solutions from the literature. The prototype provides an empirical validation of the effectiveness and efficiency of the proposed approach.

The remainder of this article is structured as follows: "Background" section provides a brief background of cloud application service availability and summarizes the problem statement. "Related work" section discusses the related work. "Multiple-objectives dynamic VM placement for application availability in cloud framework" section introduces the MoVPAAC proposed framework, which includes modules, problem formalization, and optimization solutions. "Experiments and results" section presents the results of the experiments. Finally, "Conclusion" section summarizes the conclusion.

Background

Cloud computing allows application service providers (ASPs) to request the deployment of end-to-end application services from cloud service providers (CSPs) based on specific requirements, such as availability. In response, the CSP gives the ASP online access to a set of virtual machines (VMs) where the application components can be deployed. Each application component is a software module that provides a specific type of functionality in a specific domain, such as web hosting (using an HTTP server) or networking (using network address translation or a firewall). In a data center (DC), each VM is hosted on a single physical server, and is associated with a specific application that requires a set of resources, such as CPU and RAM. Each server has its own set of properties, including availability and capacity for each resource type. The availability of a server s_j can be calculated using Eq. (1), where $MTTF_j$ is the mean time between two consecutive failures of server s_j, and $MTTR_j$ is the mean time to repair server s_j.

$$AV_j^s = \frac{MTTF_j}{MTTF_j + MTTR_j} \tag{1}$$

To illustrate how application availability is formulated, consider the following example. Figure 1a presents an abstract model of Application app_1, composed of three distinct functionalities provided by separate components located on different virtual machines (vm_1, vm_2, and vm_3), each hosted on a single server. The availability requirement for app_1 is set to 0.9, and specific resource demands are requested for each VM. To simplify the illustration, we only show the CPU demand next to each VM, along with the CPU capacity and availability next to each server. For application availability, we assume that VMs providing the same application functionality cannot be collocated on the same server, and that application availability depends on the availability of all its functionalities. We model an application as a set of functionalities provided by a set of VMs, with application availability depending on the availability of all the functionalities that together provide an end-to-end application service.

To compute application availability, denoted by AV_a^{app}, we multiply the availability of all the functionalities that comprise the application, as defined in Eq. (2), where AV_f^{func} is the availability of the functionality $func_f$ and F_a is the set of functionalities required to provide application app_a. The availability of $func_f$ is determined by the availability of the virtual machines that provide it, which can be computed as the complement of the failure probability of all the VMs that provide $func_f$, as defined in Eq. (3), where vm_v provides functionality f and V_f^{func} is the set of all VMs that provide functionality f. The failure of vm_v is equivalent to the failure of the server s_j that hosts it. The failure of a server s_j is defined as the complement of its availability, as in Eq. (4). According to Eq. (2), the availability of the deployed application (app_1) depicted in Fig. 1a can be calculated as $AV_1^{app} = AV_1^s * AV_2^s * AV_8^s = 0.97 * 0.95 * 0.98 = 0.9$, which meets the requested availability of app_1. By hosting vm_2 on server 6 instead of server 2, as shown in Fig. 1b, the availability of app_1 can be increased to $AV_1^{app} = AV_1^s * AV_6^s * AV_8^s = 0.97 * 0.99 * 0.98 = 0.94$. Moreover, adding a standby VM vm_{2_2} for vm_{2_1} and hosting it on server 3, as depicted in Fig. 1c, increases the availability of app_1 to $AV_1^{app} = AV_1^s * (1 - (Fail_6^s * Fail_3^s)) * AV_8^s = 0.97 * (1 - (0.010.05))0.98 = 0.95$. However, when a CPU scaling-up request is made for vm_1 with two additional units, server 1 cannot host vm_1 with the requested four CPU units because its CPU capacity is limited to three units. Therefore, vm_1 must be migrated to another server. As shown in Fig. 1d, vm_1 can be migrated to server 5, where $AV_1^{app} = 0.96 * (1 - (0.01 * 0.05)) * 0.98 = 0.94$.

$$AV_a^{app} = \prod_{f=1}^{|F_a|} AV_f^{func} \tag{2}$$

$$AV_f^{func} = 1 - \prod_{v=1}^{|V_f^{func}|} Fail_v^{vm} \tag{3}$$

$$Fail_j^s = 1 - AV_j^s \tag{4}$$

To illustrate the application admissibility issue, let's consider an additional scenario where a new application, app_2, is requested by another ASP. The application consists of three virtual machines, vm_4, vm_5, and vm_6, and requires an availability of 0.88. Suppose that the CSP has a placement policy that deploys VMs on servers with the highest availability. In this case, app_2 would be deployed as shown in Fig. 1e. The availability of app_2 can be calculated as $AV_2^{app} = AV_6^s * AV_7^s * AV_4^s = 0.99 * 0.99 * 0.99 = 0.97$ which meets the availability requirement. However, note that the CSP is providing a much higher availability than what the ASP requires for app_2. Now, let's assume that another ASP requests a new application, app_3, consisting of three virtual machines, vm_7, vm_8, and vm_9, with an availability requirement of 0.97. Based on the current state of the data center, as shown in Fig. 1e, the request for app_3 would be denied because the required availability cannot be met. This means that the CSP would lose the profit from hosting app_3. However, if the CSP adopts a policy of providing application availability that is close to the requested level, then app_3 could be admitted to the data center. Figure 1f shows the placement of all three applications in the data center, with app_1, app_2, and app_3 meeting their respective availability requirements of 0.94, 0.88, and 0.97. By adopting this policy, the CSP can satisfy the requirements of multiple ASPs and maximize its profits.

Related work

The literature on cloud computing has numerous studies that focus on different aspects of virtual machine (VM) placement and application task scheduling, such as resource utilization, network performance, and operational costs. However, only a limited number of studies have explored the problem of ensuring end-to-end application service availability. In light of this, we will examine previous research that deals with VM placement and task scheduling in cloud computing, as well as approaches that ensure availability, reliability, and fault tolerance.

Availability-aware VM placement

Jammal et al. [3, 4] proposed CHASE, a scheduler that takes into account high availability of application components in cloud-based systems. The authors formulated the scheduling problem as an Integer Linear Programming (ILP) model with the objective of maximizing component availability. To schedule components, CHASE selects servers with the highest availability. However, this work does not consider the problem of application admissibility. The authors used IBM ILOG CPLEX optimization solver to find the optimal scheduling plan for the components. In another work, Zhu and Huang [5] focused on the availability of Mobile Edge Computing (MEC) applications during the component placement process. The authors proposed a stochastic model to measure the cost of availability impact when changing the placement of components. The heuristic algorithms FirstFit and BestFit were used to place the MEC application. Lera et al. [6] proposed a two-phase placement strategy based on graph

partitioning and traversal approach to address service placement in the fog computing platform for application fault tolerance. The authors optimized the placement process to improve the fault tolerance of applications. Dehury et al. [7] addressed fault tolerance for application components in the cloud. They proposed a fault tolerance strategy based on the significance of each deployed component. The ranks of components were determined based on their communication, failure rate, failure impact, and historical performance. The proposed strategy used Markov Decision Process (MDP) to determine the number of replicas of each component.

The problem of reliability of VM placement (RVMP) has been addressed in works such as [8, 9]. Yang et al. [8] proposed an INLP model to determine the minimum number of computing nodes required to host VMs, ensuring that the VM placement plan's availability meets the requirement and the communication delay between VMs is less than a certain threshold. To solve the RVMP model, the authors used CPLEX. Similarly, Liu et al. [9] also mapped VM placement as an ILP model, but with the additional goals of reducing communication traffic and network bandwidth in DC while increasing the reliability of hosted VMs. To solve the ILP model, the authors employed a graph k-cut approach. Yang et al. [10] created a variance-based metric to assess the risk of application availability violations during the VM placement process. The authors examined the possibility of Top-of-Rack (ToR) switch and server failures in DC and formalized VM placement as an ILP model with the goal of reducing resource power consumption while increasing application availability.

The Virtual Network Function (VNF) placement problem in the Network Function Virtualization (NFV) platform has been the subject of several research works, including [11–19]. Ayoubi et al. [11] proposed a framework for elastic and dependable Virtual Networks (VNs) embedding in cloud environments, aiming to meet the availability requirement of VN throughout its lifetime and increase the admissibility of new VNs. The authors modeled VN as a collection of connected Virtual Network Functions (VNFs), each mapped to a single VM. The approach utilized backup VNFs and a tabu-search optimization method to achieve reliable VNF placement. Alahmad et al. [12] proposed a VNF placement model that prioritizes availability and minimizes Network Service (NS) failure probability in NFV, evaluated using CPLEX. Thiruvasagam et al. [13] tackled the placement of reliable virtual monitoring functions (vMFs) by minimizing communication delay between Service Function Chains (SFCs) in the NS while also reducing the number of vMFs. The authors used CPLEX to determine the best vMF placement strategy. Yala et al. [14] employed a genetic algorithm to determine the VNF placement in a virtual Content Delivery Network (vCDN) and to balance vCDN deployment cost and availability level. Yang et al. [15] addressed stateful VNF placement for NS fault-tolerance and modeled the problem as an optimization function, aiming to increase user request availability. In [16], the authors proposed an availability-aware SFC placement scheme for the NFV substrate network, aiming to reduce SFC's end-to-end delay. Sharma et al. [17] focused on maximizing the Telecom Service Provider's (TSP) profit by achieving high NS availability in NFV during VNF placement using redundant VNFs and a geographic placement approach. Abdelaal et al. [18] addressed the VNF Forwarding Graph (VNF-FG) deployment problem with the goals of minimizing network bandwidth, convergence time, and resource power consumption while protecting VNF service from failures using redundant VNFG. Mao et al. [19] proposed an online fault-tolerant SFC placement solution in NFV, modeled as a Markov decision process, using a deep reinforcement learning (DRL) method to maximize the number of accepted user requests.

Several works have proposed cloud fault-tolerance solutions using virtual machine (VM) placement. Li and Qian [20] focused on reducing network traffic in data centers by addressing multitenant cloud VM placement. Jammal et al. [21] addressed the issue of VM placement during live migration to reduce service downtime in the event of a failure. Zhou et al. [22, 23] aimed to minimize network resource consumption and increase cloud service reliability through optimal redundant VM placement (ORVMP) using genetic algorithms. Gonzalez and Tang [24] used the FirstFit algorithm to place VM replicas for service fault tolerance. Alameddine [25] proposed a protection plan to determine number of backup VMs and placement to meet critical cloud application's availability requirements. Cost functions were also used to address VM placement. Chen and Jiang [26] proposed an adaptive selection method for fault-tolerant application service during the VM placement process. Zhang et al. [27] investigated VM placement in cloud DCs using a star topology to minimize SLA violations, power consumption, and failure rate. Tran et al. [28] proposed a proactive fault-tolerant approach for Kubernetes containerized services using Bidirectional Long Short Term Memory (LSTM) node fault prediction and container-based service stateful migration mechanism. Finally, Saxena et al. [29] proposed the fault-tolerant elastic resource management (FTERM) framework to handle cloud outages based on online Multi-Input and Multi-Output Evolutionary Neural Network (MIMO-ENN) to predict resource failure and take action.

Fault-tolerance task scheduling

Previous research studies have explored the impact of task scheduling on application task failures in cloud computing clusters. However, many of these studies fail to account for recovery measures for failed tasks or preventative measures for predicted failures. Moreover, they do not assess the application task's availability in meeting specific requirements. Our research sets itself apart by considering the migration of virtual machines (VMs) that host predicted failed tasks and ensuring that the application meets its availability requirements throughout its operational lifetime.

Several studies have proposed fault tolerance solutions for cloud application task scheduling. Guo et al. [30] developed a fault-tolerant and energy-efficient primary-back scheduling architecture for real-time tasks in a cloud environment. Marahatta et al. [31] proposed an energy-aware and fault-tolerant dynamic task scheduling scheme that reduces rejection rates by replicating tasks in case of VM failure or delay. Sun et al. [32] introduced a QoS-aware task scheduling model with fault tolerance for an edge-cloud platform, using a primary-backup redundancy approach to improve task availability while adhering to time constraints. Yao et al. [33] analyzed fault-tolerant properties of task scheduling and migrating VMs based on the Primary-Backup model and proposed a fault-tolerant elastic algorithm for task scheduling that considers host and network device faults in a cloud data center. Additionally, Yao et al. [34] presented a hybrid fault-tolerant algorithm for scheduling tasks with deadlines in a cloud platform. The algorithm selects the most suitable fault-tolerant strategy, such as task resubmission or replication, based on the characteristics of the task and available resources. Weikert et al. [35] studied node failure in IoT networks and proposed a task allocation algorithm based on multiple objective optimization. The algorithm utilizes an archive-selection mechanism to identify the most reliable assignment for the backup task in case of node failure. Overall, while previous research has examined the effect of task scheduling on application task failures in cloud computing clusters, our research goes beyond existing works by incorporating migration measures and ensuring that the application meets its availability requirements. Additionally, a range of fault tolerance strategies have been proposed for cloud application task scheduling, including energy-efficient, QoS-aware, and hybrid fault-tolerant algorithms that consider host and network device faults, as well as multiple objective optimization techniques.

Several research studies have leveraged the Google cloud trace dataset [36] to predict application job and task failures in cloud cluster systems. Chen et al. [37] explored the critical characteristics of application job and task failures and used a deep learning Recurrent Neural Network (RNN) to predict such failures. To predict task failure, Soualhia et al. [38] combined machine learning methods, including Decision Tree (DT), Boost, and Random Forest (RF). Jassas and Mahmoud [39, 40] compared multiple prediction models, including DT, Logistic Regression (LR), K-Nearest Neighbors (K-NN), Naive Bayes (NB), RF, and Quadratic Discrimination Analysis (QDA), to select the most accurate method. Islam and Manivannan [41] employed a deep learning method called LSTM to predict task failure. While these works focused on predicting failures, other works proposed recovery actions for failing tasks or jobs. For instance, Rosa et al. [42] suggested terminating a job that is predicted to fail to save consumed resources, while Islam and Manivannan [43] proposed rescheduling tasks that are predicted to fail to a more reliable computing node. Soualhia et al. [44] proposed a fault-tolerant task scheduling framework (ATLAS) for Hadoop clusters, which can dynamically reschedule tasks that are predicted to fail. Our previous work [45] also utilized the Google dataset [36] to predict task failure during execution time, proposing three corrective actions to protect the task before it fails: changing the priority, scheduling class level, or task scheduling node. Chen et al. [46] proposed advance approach called IWC to improve the search method of Whale Optimization Algorithm (WOA) for Cloud task scheduling. Authors show IWC has better speed and accuracy to find the optimal task scheduling plan compared to existing meta-heuristic algorithms. Cheng et al. [47] proposed an enhanced deep reinforcement learning (DRL) to improve the existing studies that used DRL for job scheduling in Cloud platforms. They tried to optimize job execution time while meeting the expected response time of the users. Zhang et al. [48] proposed a new method called GA-DQN that combines DRL and Genetic Algorithms (GA) for scheduling jobs in cloud. The method benefits from the GA global search ability and awareness of decision-making of DRL to have optimized sub-task scheduling that can reduce the execution times of the jobs, and hence have better response time for the end users. Notably, none of these studies computed application service availability in the cloud to meet the requirements during VM placement or task scheduling procedures. Table 1 provides a summary of related work.

Multiple-objectives dynamic VM placement for application availability in cloud framework

We introduce a novel framework for dynamic VM placement in cloud platforms that prioritizes application service availability. Our framework generates and manages a comprehensive placement plan for VMs that provide services inside data centers, adhering to specific requirements to achieve multiple objectives and meet the availability needs of each application as requested by the ASP. Additionally, our framework has the ability to swiftly modify VM placement in response

Table 1 Summary of related work

Research Topic	Reference	App Availability
Application Component Placement	[3–7]	X
Reliability-Aware VM Placement	[8–10]	X
Network Service Fault Tolerance using VNF Placement	[11–19]	X
Fault Tolerance Solutions using VM/container Placement	[20–29]	X
Fault Tolerance Solutions using Task Scheduling	[30–35]	X
Application Job/Task Failure Prediction	[37–48]	X

'X' denotes not applicable

to application scaling or failure events. As shown in Fig. 2, the proposed MoVPAAC (Multi-Objective Virtual Machine Placement with Availability-Aware Computing) framework comprises three main modules: the Availability-Aware Application Deployment module, which optimizes VM placement to maximize availability; the Proactive Application Failure Detection module, which uses deep learning algorithms to detect potential application failures and take corrective actions before they occur; and the Dynamic Application Reconfiguration module, which allows for prompt reconfiguration of VM placement in response to application failures or changes in demand. We delve into the specific features of each module in detail in the following subsections.

Availability-aware application deployment

The Availability-Aware Application Deployment module is a critical component of our proposed framework, as it is responsible for generating the VM placement plan that will deploy the requested applications at the underlying servers located in the data center (DC). The module ensures that the objectives are achieved, while also taking into consideration the specific requirements of each application, particularly their availability as requested by ASPs. Given a set of applications with their respective requirements, each application is comprised of a set of VMs, and each VM provides a specific functionality towards providing end-to-end application services. The goal is to find the optimal placement plan for these VMs on the DC servers, such that power consumption, resource wastage, and server failure ratios are minimized, while ensuring that the availability requirements of the applications are maintained throughout their entire execution times. However, as we mentioned in the background section, VM placement is an NP-hard problem with contradictory objectives. To address this, we have formulated the problem as an INLP optimization model with multiple objectives and constraints. Moreover, we propose a heuristic approach based on the AntColony optimization method, in conjunction with the VM standby protection approach, to find a solution for the model and maximize the admissibility of the requested applications. Specifically, we define and formulate the problem statement we address in this manuscript as follows: assume there is a set

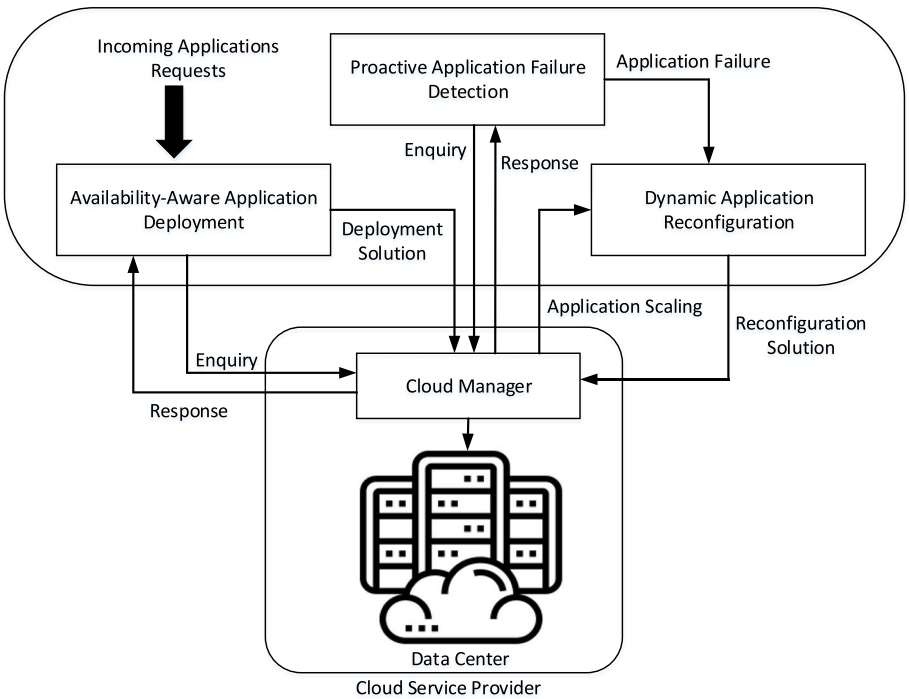


Fig. 2 Multiple-Objectives Dynamic VM Placement for Application Availability in Cloud (MoVPAAC) Framework

A of applications that are requested by ASPs. Each application $app_a \in A$ is requested to be deployed at Data Center (DC), and has availability requirement that is denoted by AV_a^{appReq}. Each application app_a is composed of a set of VMs V_a, each VM $vm_i \in V_a$ has a set of resources demands such as CPU, RAM and disk. The VMs of applications set A require to be placed (hosted) at the underlying set of servers S that are located in DC. Each server $s_j \in S$ has a resource capacity of different types such as CPU, RAM and disk. The main goal is to deploy (admit) applications set A at DC in such a way that can meet the availability requirement $AV_a^{app} >= AV_a^{appReq}$ for each $app_a \in A$, and achieve the following objectives. The first objective is to minimize the total power consumption of the active servers that are used to host VMs that compose applications in A. To compute the power consumption of server s_j in the DC, we adopt the linear relationship between server power consumption and its CPU utilization as described in [49]. We define the average power consumption of server s_j as P_j in Eq. (5), where P_j^{active} and P_j^{idle} are the average power consumption values when s_j is active and idle, respectively, and U_j^c is the CPU utilization of s_j, where $U_j^c \in [0, 1]$. The first objective is formulated in Eq. (6), where V is the set that includes all the VMs that compose all the requested applications in A, y_j is a binary decision variable where value 1 indicates that s_j is active and a value 0 indicates that s_j is idle, as defined in Eq. (10). R_i^c is the CPU resource demand by vm_i, and x_{ij} is a binary decision variable where value 1 indicates that vm_i is placed on s_j and value 0 otherwise, as defined in Eq. (11).

$$P_j = \left(P_j^{active} - P_j^{idle}\right) \times U_j^c + P_j^{idle} \tag{5}$$

$$Minimize \sum_{j=1}^{|S|} P_j = \sum_{j=1}^{|S|} \left(y_j \times \left(\left(P_j^{active} - P_j^{idle}\right) \times \sum_{i=1}^{|V|} \left(R_i^c \times x_{ij}\right) + P_j^{idle}\right)\right) \tag{6}$$

The second objective of the Availability-Aware Application Deployment module is to minimize the wastage of resources of active servers in the data center (DC). The cost of wasting resources for server s_j is denoted as W_j and is defined in Eq. (7). The remaining CPU, RAM, and Disk resources of server s_j are normalized and represented by L_j^c, L_j^r, and L_j^d respectively. U_j^c, U_j^r, and U_j^d represent the normalized resource usage of server s_j. To ensure a positive value, we set β as a very small value of 0.00001. The second objective is formulated in Eq. (8). T_j^c, T_j^r, and T_j^d represent the upper utilization thresholds of CPU, RAM, and Disk of server s_j respectively. These thresholds are set to the same value for all servers in the DC to prevent any server from reaching a full usage state that could negatively impact its performance. The RAM and Disk resource demand of vm_i are represented by R_i^r and R_i^d respectively. The third objective of the module is to minimize the overall failure ratio of servers in the DC. The module computes the failure of server s_j as the complement of its availability, as defined in Eq. (4), where AV_j^s is computed as defined in Eq. (1). The third objective is formalized in Eq. (9). By optimizing these objectives in a multi-objective optimization model, the module aims to find a placement plan for VMs on the DC servers that reduces power consumption, resource wastage, and failures ratio while meeting the availability requirements of the applications. To solve this problem, the module proposes a heuristic approach based on the Ant Colony Optimization method in conjunction with VM standby protection approach to maximize the admissibility of the requested applications.

$$W_j = \frac{||L_j^c - L_j^r| - L_j^d| + \beta}{U_j^c + U_j^r + U_j^d} \tag{7}$$

$$Minimize \sum_{j=1}^{|S|} W_j = \sum_{j=1}^{|S|} \left(y_j \times \left(||(T_j^c - \sum_{i=1}^{|V|} \left(R_i^c \times x_{i,j}\right)\right) - \left(T_j^r - \sum_{i=1}^{|V|} \left(R_i^r \times x_{i,j}\right)\right)| - \left(T_j^d - \sum_{i=1}^{|V|} \left(R_i^d \times x_{i,j}\right)\right)| + \beta\right) / \left(\sum_{i=1}^{|V|} \left(R_i^c \times x_{i,j}\right) + \sum_{i=1}^{|V|} \left(R_i^r \times x_{i,j}\right) + \sum_{i=1}^{|V|} \left(R_i^d \times x_{i,j}\right)\right) \tag{8}$$

$$Minimize \sum_{j=1}^{|S|} Fail_j^s = \sum_{j=1}^{|S|} \left(y_j \times Fail_j^s\right) \tag{9}$$

Our VM placement model is governed by a set of carefully defined constraints. Firstly, each server s_j can be either active or idle at any given time, as specified in Eq. (10). To indicate whether a VM vm_i is placed on a particular server s_j, we use a binary decision variable x_{ij}, as outlined in Eq. (11). Additionally, each VM can be placed on at most one server, as mandated by Eq. (12). To ensure that each server has adequate resources to host any VM, we impose constraints on the amount of CPU, RAM, and disk space available on each server. Specifically, Eqs. (13) through (15) outline the resource requirements that must be met for each server. We also enforce an "anti-affinity" constraint to ensure that VMs belonging to the same application app_a are not co-located on the same server. This helps to increase the availability of the application, as specified in Eq. (16). Our work considers the dependency between the

components of the same application. For example, peer, active-standby, proxy and proxied components of the same application should be hosted on different servers. Finally, to ensure that the requested applications are available to the application service provider (ASP) as required, we require that the availability of each application be greater than or equal to the level requested by the ASP. This requirement is captured in Eq. (17). By carefully balancing these constraints, we can optimize the placement of VMs to meet the needs of both users and service providers.

Subject to:

$$y_j = \begin{cases} 1, \text{if } s_j \text{ is active} \\ 0, \text{if } s_j \text{ is idle} \end{cases} \tag{10}$$

$$x_{ij} = \begin{cases} 1, \text{if } vm_i \text{ is placed on } s_j \\ 0, \text{otherwise} \end{cases} \tag{11}$$

$$\sum_{j=1}^{|S|} x_{ij} \leq 1 \quad \forall i \in |V| \tag{12}$$

$$\sum_{i=1}^{|V|} \left(R_i^c \times x_{ij}\right) \leq T_j^c \quad \forall j \in |S| \tag{13}$$

$$\sum_{i=1}^{|V|} (R_i^r \times x_{ij}) \leq T_j^r \quad \forall j \in |S| \tag{14}$$

$$\sum_{i=1}^{|V|} (R_i^d \times x_{ij}) \leq T_j^d \quad \forall j \in |S| \tag{15}$$

$$x_{ij} + x_{zj} <= 1 \quad \forall vm_i, vm_z \in V_a^{app}, \forall j \in S \tag{16}$$

$$AV_a^{app} \geq AV_a^{appReq} \quad \forall app_a \in A \tag{17}$$

To address the INLP model and determine the optimal placement of VMs for requested applications, we introduce a heuristic algorithm called Availability-Aware Applications Deployment (AvAAD) (Algorithm 1). The AvAAD algorithm employs an AntColony optimization approach to achieve its objectives of VM placement, while utilizing a standby protection technique to ensure the availability requirements of the applications are met. The AvAAD takes a list of requested applications, their requirements, available servers at the data center, and VMs as input. It returns a list of non-admitted applications as output. Initially, the algorithm initializes three empty variables: *paretoSet*, *violatedAvApps*, and *nonAdmittedApps*. It then calls the MOAntColony algorithm with VMs and servers as arguments. MOAntColony returns a *paretoSet* that includes the placement of VMs at the available servers. Using the *paretoSet*, AvAAD computes the availability AV_a^{app} of each requested application app_a. It adds each application that violates its availability requirement ($AV_a^{app} < AV_a^{appReq}$) to *violatedAvApps*. For each application in *violatedAvApps*, the algorithm tries to enhance its availability to meet the requirement. Specifically, it attempts to add a new standby VM for the functionality with the minimum availability AV_f^{func} among all the functionalities in the application. The algorithm adds one standby VM at a time until it meets the availability requirement of the application or the number of added standby VMs reaches the threshold of app_a. The newly added standby VM is placed on the server with the maximum value of $\frac{1}{P_j+W_j+Fail_j^s}$ among all servers, without violating any of the constraints defined in Eqs. (10 - 17). This maintains consistency with the objectives of the MoVPAAC framework. After AvAAD handles all violated applications, it checks again for any applications that still violate their availability requirements. If an application still violates its requirement, AvAAD considers it rejected and adds it to the list of non-admitted applications (*nonAdmittedApps*) that is returned at the end of the algorithm execution. AvAAD optimizes VM placement while ensuring application availability, making it a robust and effective solution for the INLP model.

Algorithm 1 Availability-Aware Application Deployment (AvAAD)

Input: A, S, V
Output: $nonAdmittedApps$
1: *initialize* $paretoSet$, $violatedAvApps$, $nonAdmittedApps$ as *empty*
2: $paretoSet = MOAntColony(V, S)$
3: **for** each $app_a \in A$ **do**
4: $A_a^{app} \leftarrow computeAvailability(app_a)$ // using (Eq.2)
5: **if** $A_a^{app} < A_{req_a}^{app}$ **then**
6: *Add* app_a *to* $violatedAvApps$
7: **end if**
8: **end for**
9: **for** each $app_a \in violatedAvApps$ **do**
10: $vmsAddedStandby = 0$
11: **while** $((A_a^{app} < A_{req_a}^{app})$ and $(vmsStandbyAdded < app_a.maxAddVmsStandby))$ **do**
12: *Add standby* vm *to functionality* $f \in app_a$ *which has minimum* A_f^{func}
13: place standby vm at s_j which has maximum $\frac{1}{P_j+W_j+Fail_j^s}$ *among* $s_j \in S$ *without violating constraints (Eq.10 - Eq.17)*
14: $vmsStandbyAdded+ = 1$
15: $A_a^{app} \leftarrow computeAvailability(app_a)$
16: **end while**
17: **end for**
18: **for** each $app_a \in violatedAvApps$ **do**
19: **if** $A_a^{app} < A_{req_a}^{app}$ **then**
20: *Add* app_a *to* $nonAdmittedApps$
21: **end if**
22: **end for**
23:
24: **return** $nonAdmittedApps$

The time complexity of AvAAD (Algorithm 1), can be analyzed as follows. At line (2), the algorithm calls

MOAntColony (Algorithm 2) to find the placement plan of the virtual machines (VMs) in V at the servers in S. The performance of AvAAD mainly depends on the performance of MOAntColony. AntColony is a meta-heuristic algorithm that takes a polynomial execution time of $\mathcal{O}(n^k)$ to find the optimal solution [50]. In the context of the VM placement problem, the value of k mainly depends on the number of iterations, ants, VMs, and servers that AntColony uses to find the placement solution. At lines (3-8), the algorithm takes $\mathcal{O}(n)$ to determine the list of applications in *violatedAvApps* that violate their availability requirements. At lines (9-17), it takes $\mathcal{O}(n^2)$ to satisfy the availability for each application that violates its required availability. At lines (18-22), the algorithm takes $\mathcal{O}(n)$ to determine the list of rejected applications in *nonAdmittedApps* that cannot be admitted at the data center (DC) since they violate their availability requirements. Therefore, the total time complexity of Algorithm 1 can be expressed as $\mathcal{O}(n^k) + \mathcal{O}(n) + \mathcal{O}(n^2) + \mathcal{O}(n)$, which can be simplified to $\mathcal{O}(n^k)$. It is worth noting that the performance of the algorithm may vary depending on the input parameters, such as the number of VMs, servers, and applications.

To achieve the objectives of application deployment, we propose a heuristic algorithm called Multiple Objectives AntColony (MOAntColony) that utilizes the Ant Colony Optimization (ACO) algorithm to find the placement of VMs for requested applications. Algorithm 2 outlines the steps of MOAntColony. The algorithm begins by initializing the parameters and pheromone trials. In each iterative step, an ant z receives a set of VMs V that need to be placed in a set of servers S located at the data center (DC). The ant z then selects a server s_j and starts placing the VMs in V at s_j using the pseudo-random-proportional rule [37]. The desirability of selecting the next vm_i to place at s_j depends on the pheromone concentration level and the heuristic information that guides ant z. After each movement (placement) step, the local pheromone concentration level is updated. Ant z continues moving until it completes the placement of V and builds its solution. Once all ants complete and build their solutions, a global pheromone is updated based on the pareto set PS that includes the best-located solutions. The algorithm initializes the pheromone level τ_0 using Eq. (18). Here, n is the total number of VMs that require placement, $P^{'}(sol_0)$ is the normalized power consumption of the servers listed in the initial placement solution sol_0 generated by the FirstFit VM placement algorithm, $W^{'}(sol_0)$ and $Fail'(sol_0)$ are the resource wastage and server failures of sol_0, respectively. Equation (19) defines $P^{'}(sol_0)$, where P_j^{max} is the maximum power consumption of server j, and M is the total number of servers used in solution sol_0. $W^{'}(sol_0)$ and $Fail'(sol_0)$ are defined in Eqs. (22) and (23), respectively. The heuristic information $\eta_{i,j}$ indicates the desirability of an ant z to place vm_i at server s_j. The desirability $\eta_{i,j}$ considers the partial contribution for each objective. Every ant z begins with V and starts placing them sequentially on the available servers in S, which are arranged randomly. The sequence of servers from 1 to j is known during the placement of vm_i at s_j. The partial contributions of the first, second, and third objectives are defined in Eqs. (24), (25), and (26), respectively. These contributions are combined for the heuristic placement decision, as defined in Eq. (27).

$$\tau_0 = \frac{1}{n \times (P^{'}(sol_0) + W^{'}(sol_0) + Fail^{'}(sol_0)} \tag{18}$$

$$P^{'}(sol_0) = \sum_{j=1}^{M} \left(P_j / P_j^{max}\right) \tag{19}$$

$$W^{'}(sol_0) = \sum_{j=1}^{M} (W_j) \tag{20}$$

$$Fail^{'}(sol_0) = \sum_{j=1}^{M} (Fail_j) \tag{21}$$

$$W^{'}(sol_0) = \sum_{j=1}^{M} (W_j) \tag{22}$$

$$Fail^{'}(sol_0) = \sum_{j=1}^{M} (Fail_j) \tag{23}$$

$$\eta_{i,j,1} = \frac{1}{\beta + \sum_{k=1}^{j} \left(P_k / P_k^{max}\right)} \tag{24}$$

$$\eta_{i,j,2} = \frac{1}{\beta + \sum_{k=1}^{j} W_k} \tag{25}$$

$$\eta_{i,j,3} = \frac{1}{\beta + \sum_{k=1}^{j} Fail_k} \tag{26}$$

$$\eta_{i,j} = \eta_{i,j,1} + \eta_{i,j,2} + \eta_{i,j,3} \tag{27}$$

Ant z uses the pseudo-random-proportional rule, as defined in Eqs. (28) [37], to select the next VM, vm_i, to be placed on server s_j. The rule employs the parameter α to control the importance of pheromone trails, and q is a random number between 0 and 1. If q is less than or equal to the fixed value of q_0 (where $0 < q_0 < 1$), it falls under exploitation, otherwise it falls under exploration, as specified in Eq. (28). U denotes the set of VMs that can be hosted on s_j. $\eta_{u,j}$ represents the pheromone value, as defined in Eq. (27), while $\tau_{u,j}$ is the local pheromone update, as defined in Eq. (30). Furthermore, Pr denotes the probability distribution of the random-proportional rule, as described in Eq. (29) [37]. The pheromone is updated locally and globally. During the local update, ant z assigns vm_i to s_j and updates the pheromone, as described in Eq. (30). Here, τ_0 represents the initial pheromone level, and $0 < \rho_l < 1$ denotes the local pheromone evaporation parameter. The current iteration is denoted as t. The global pheromone update is performed based on the rule stated in Eq. (31), where $0 < \rho_g < 1$ is the global pheromone evaporation parameter. The coefficient λ, as defined in Eq. (32), incorporates the number of ants Z and iterations T^g needed to locate the global solution sol_g in the pareto set PS. Furthermore, $P'(sol_g)$, $W'(sol_g)$, and $Fail'(sol_g)$ represent the normalized power consumption, resource wastage, and failures, respectively, of the servers listed in the solution sol_g. It is important to note that algorithm 2 primarily utilizes the Ant Colony metaheuristic optimization algorithm, which requires an execution time of $\mathcal{O}(n^k)$ [37]. The value of k depends on the number of iterations T, ants Z, VMs in V, and servers in S used by the Ant Colony algorithm to determine the placement plan for V.

$$i = \begin{cases} max_{u \in U}\{\alpha \times \tau_{u,j} + (1-\alpha) \times \eta_{u,j}\}, & q \leq q_0 \\ Pr, \text{otherwise} \end{cases} \tag{28}$$

$$Pr_{u,j} = \begin{cases} \frac{\alpha \times \tau_{u,j} + (1-\alpha) \times \eta_{u,j}}{\sum_{u=1}^{|U|} (\alpha \times \tau_{u,j} + (1-\alpha) \times \eta_{u,j})}, & u \in U \\ 0, otherwise \end{cases} \tag{29}$$

$$\tau_{i,j}(t) = (1-\rho_l) \times \tau_{i,j}(t-1) + \rho_l \times \tau_0 \tag{30}$$

$$\tau_{i,j}(t) = (1-\rho_g) \times \tau_{i,j}(t-1) + \frac{\rho_g \times \lambda}{P'(sol_g) + W'(sol_g) + Fail'(sol_g)} \tag{31}$$

$$\lambda = \frac{Z}{t - T^g + 1} \tag{32}$$

Algorithm 2 MOAntColony

Input: V, S
Output: PS
Initialize values of parameters τ_0, α, q_0,ρ_l, ρ_g, Z, and T
Initialize PS as empty
Initialize all pheromone values to τ_0
for t=1 to T **do**
  for z=1 to Z **do**
    sort S in random order
    while not all $vm_i \in V$ are placed **do**
      select s_j
      while there is a vm_i can be placed at s_j **do**
        for each remaining vm_i can be placed at s_j **do**
          Calculate desirability for vm_i as in (Eq.28)
          Calculate probability for vm_i as in (Eq.29)
        end for
        Select vm_i to assign at s_j
        Generate q randomly
        if $q \leq q_0$ **then**
          exploitation as in (Eq.28)
        else
          exploration as in (Eq.28)
          Update local pheromone as in (Eq.30)
        end if
      end while
    end while
  end for
  Calculate the objectives for each solution sol_k generated by each ant z as in (Eq.6, Eq.8, Eq.9)
  if S is non-dominated by any other solution **then**
    Add sol_k to PS
    Remove solutions $\in PS$ that are dominated by sol_k
  end if
  for each $sol_k \in PS$ **do**
    Update global pheromone as in (Eq.31)
  end for
end for

return PS

Proactive application failure detection

The proactive application failure detection module is crucial for detecting application failure at an early stage, before it actually occurs. Service outages caused by application failures can lead to significant negative impacts on QoS, SLA compliance as well as negative end user experience. The module uses proactive approach to detect task failure regardless of its type from historical dataset. The dataset includes historical information about failures of tasks and their types such as network, hardware, software failures. Note the module does not react to instant failures of any type. Detecting failures at an early stage allows for appropriate service recovery actions to be taken quickly. The module adopts polling communication approach to get information about the current status of the cluster and hosted applications from the Cloud Manager. The information is used as a historical data for training and testing the used prediction method Artificial Neural Network (ANN) to predict the application failure. To validate this module, we conducted an analysis of the Google dataset [36] in our previous work [45]. This dataset consists of logs of application jobs and their associated tasks executed on a cloud cluster for 29 consecutive

days in 2011. We extracted information about the resources required and used by each task, as well as the termination status (finished, failed, evicted, or killed) of the tasks. Out of 48,261,777 tasks, 38% were successfully terminated, while 29% failed. Through our analysis, we identified several features that were correlated with task termination status, including the task ID, job ID, machine ID, CPU and RAM demands, mean CPU and RAM usage, and termination status. We trained a deep learning ANN method on this data to predict task failure. To detect predicted failed tasks and initiate recovery actions, our proactive application failure module employs the approach outlined in Algorithm 3. The input for the algorithm is a list of tasks that need to have their termination status predicted, and it returns a list of predicted failed tasks. It is worth noting that the ANN is trained and tested on a cleaned and prepared dataset before it is used by Algorithm 3. In terms of time complexity, Algorithm 3 takes $\mathcal{O}(n)$ time to predict the termination status of each task in the input list. By proactively detecting and responding to application failure, we can minimize service outages and maintain high levels of QoS and SLA compliance.

Algorithm 3 Proactive Application Failure Detection

Input: $tasksList$
Output: $failPredTasksList$
for each $task \in tasksList$ **do**
  $terminationStatus = ANN.predictTerminationStatus(task)$
  if $terminationStatus ==$ 'Failed' **then**
    Add $task$ to $failPredTasksList$
  end if
end for

return $failPredTasksList$

Algorithm 4 VM Placement for Application Recovery

Input: $failPredTasksList$, S
Output: $vmsPlacementMap$
for each $task_t \in failPredTasksList$ **do**
  Add vm_i provides $task_t$
  $minCost = \infty$
  $minServer = empty$
  for each $s_j \in S$ **do**
    if s_j can host vm_i and satisfy constraints in (Eq.10 - Eq.17) **then**
      $serverCost = P_j + W_j + Fail^s_{s_j}$
      if $serverCost < minCost$ **then**
        $minCost = serverCost$
        $minServer = s_j$
      end if
    end if
  end for
  Add $< vm_i, minServer >$ to $vmsPlacementMap$
end for
return $vmsPlacementMap$

Algorithm 5 VM Placement for Application Scaling

Input: $vmsScaleList$, S, $scaleType$
Output: $vmsPlacementMap$
if $scaleType =$'out' **then**
  for each $vm_i \in vmsScaleList$ **do**
    $minCost = \infty$
    $minServer = empty$
    for each $s_j \in serversList$ **do**
      if s_j can host vm_i and satisfy constraints in (Eq.10 - Eq.17) **then**
        $serverCost = P_j + W_j + Fail^s_{s_j}$
        if $serverCost < minCost$ **then**
          $minCost = serverCost$
          $minServer = s_j$
        end if
      end if
    end for
    Add $< vm_i, minServer >$ to $vmsPlacementMap$
  end for
else if $scaleType ==$'up' **then**
  for each $vm_i \in vmsScaleList$ **do**
    $s_j = vm.hetHostedServer()$
    if s_j can not host vm **then**
      $minMigrationCost = \infty$
      $minDestServer = empty$
      for each $s_d \in serversList$ **do**
        if s_d can host vm and satisfy constraints in (Eq.10 - Eq.17) **then**
          $serverMigrationCost = getVmMigrTime(vm_i, s_j, s_d)$
          if $serverMigrationCost < minMigrationCost$ **then**
            $minMigrationCost$
            $minDestServer = s_d$
          end if
        end if
      end for
      Add $<vm, minDestServer>$ to $vmsPlacementMap$
    end if
  end for
else if $scaleType ==$'in' OR $scaleType ==$'down' **then**
  for each $vm_i \in vmsScaleList$ **do**
    if scale action for vm_i violates the availability constraint in (Eq.17) **then**
      reject scale action for vm_i
    end if
  end for
end if

return $vmsPlacementMap$

Dynamic application reconfiguration

The dynamic application reconfiguration module is responsible for handling reconfiguration requests that arise when the availability requirements of provisioned applications are threatened to be violated. These requests can originate from either the proactive application failure module, which notifies the module of predicted failed applications, or from the cloud manager, which sends scaling requests. In the case of a proactive notification, the module adds a new VM to replace the existing VM responsible for each predicted failed task. The placement of these new VMs is crucial to the successful recovery of the application services. The proposed placement process is designed to fulfill the objectives outlined in formulas Eqs. (6), (8), and (9) while respecting the constraints

defined in formulas Eqs. (10) through (17), which align with the objectives of the MoVPAAC framework. Algorithm 4 outlines the placement procedure for these new VMs to recover the application services. The algorithm takes in a list of application tasks predicted as failed, *failPredTasksList*, and a list of servers, *S*, as input. It returns a map that includes the placement of the new VMs required to provision the failed tasks as output. For each failed task in *failPredTasksList*, the algorithm adds a new VM to provide the task and searches for a server $s_j \in S$ that can host the VM and has the minimum summation value of power consumption, resource waste, and failure without violating any constraints defined in formulas Eqs. (10) through (17). The algorithm then adds the record $< vm_i, s_j >$ to the map *vmsPlacementMap*. Finally, the algorithm returns the map *vmsPlacementMap*. The time complexity of Algorithm 4 is $\mathcal{O}(n^2)$ because for each added VM, the algorithm searches for the best server s_j among *S* that can host the VM.

The cloud manager at CSP can request one of four scaling types: scaling out, scaling up, scaling in, or scaling down. Scaling out request involves adding a set of new virtual machines (VMs), while scaling up request involves adding virtual resources, such as virtual central processing units (vCPUs) and virtual random-access memory (vRAM), to an existing set of individual VMs. Scaling in request involves removing a set of existing individual VMs, and scaling down request involves removing virtual resources from an existing set of VMs. If the request is for application scaling out, the reconfiguration module handles the placement of the new VMs in the same way that it handles requests from the proactive application failure module. However, in some cases, scaling up may require migrating VMs to other servers that can accommodate the updated resources without violating any constraints. The migration process must be done carefully, as it can significantly affect the outage period of the application service. The problem can be summarized as finding the optimal way to migrate all the VMs with minimum migration time while obeying the constraints.

To solve the problem, we propose an integer nonlinear programming (INLP) model with the objective of minimizing the migration time of the VMs that need to be migrated while obeying the constraints. The model includes a set of VMs that need to be migrated (*G*), a set of available servers at the data center (*S*), and the time to migrate a VM from a source server to a destination server ($migrationTime_{i,j,d}$). Binary decision variables (x_{ij} and z_{id}) are defined to indicate the hosting server of each VM and whether a VM needs to be migrated to a specific server, respectively. We also propose a heuristic approach described in Algorithm 5 to solve the INLP model and find the placement servers of the VMs that require scaling. The algorithm takes as input the set of VMs that need to be scaled (*vmsScaleList*), available servers (*S*), and the scaling type (*scaleType*) and returns a map that includes the placement of the VMs on the servers in the data center. Note that Algorithm 5 is called for one corresponding application at a time where the scaling request is required to fulfill the needs of the application. If the scaling type is out, the algorithm searches for a server that can host each added VM with minimum summation value of power consumption, resource waste, and failure, while meeting all the constraints. For scaling up, the algorithm determines which VMs need to be migrated and finds a destination server that minimizes the total migration time. For scaling in and down, the algorithm rejects any scaling action that violates the application availability requirement constraint. The time complexity of Algorithm 5 can be analyzed as follows. For a scale out request at lines (1-15), the algorithm searches for the best server with minimum cost that can host each vm_i. Since this operation is performed for each vm_i, the time complexity of this operation is $\mathcal{O}(n^2)$, where *n* is the number of available servers. Similarly, for a scaling up request at lines (16-33), the algorithm searches for the server that can host each vm_i with minimum migration time. Again, this operation is performed for each vm_i, resulting in a time complexity of $\mathcal{O}(n^2)$. For scaling requests of type in or down, the algorithm takes $\mathcal{O}(n)$ to check whether the scaling action should be taken or rejected. Overall, the time complexity of the algorithm is the sum of the time complexities of each operation, which is $\mathcal{O}(n^2) + \mathcal{O}(n^2) + \mathcal{O}(n) + \mathcal{O}(n)$. This can be simplified to $\mathcal{O}(n^2)$. Therefore, the time complexity of the algorithm is quadratic in the number of available servers *n*.

Experiments and results

To evaluate the effectiveness of the MoVPAAC framework, we conducted a variety of experiments testing its modules and algorithms. As a proof of concept for our research, we developed a simulation that models the key elements of the framework, including data centers, servers, VMs, and applications, and implemented it using the C++ programming language. All experiments were conducted on a 64-bit Windows 10 machine equipped with an Intel Core i7-8665U 2.11GHz processor and 16 GB of RAM, ensuring reliable and consistent results.

We conducted experiments to evaluate the performance of the availability-aware application deployment module in the proposed MoVPAAC framework. The experiments were divided into two groups. The first group consisted of a set of application deployment requests with no standby VMs. The first group includes four requests for applications deployment by different ASPs. Each request includes deployment of different

number of applications. Each application has availability requirement (Req Availability), number of functionalities that compose the application (Funct No) which we add to simulate real-world applications and emphasize the concept of redundancy, (VMs No) which indicates number of VMs that host the components that provide the functionalities of the application. We simulated one DC with 85 heterogeneous servers. Server properties, such as CPU and RAM capacities and availability levels, were randomly generated using a uniform distribution with values ranging from 8-15 units and 0.7-0.99, respectively. For all servers, P^{active} and P^{idle} were set to 215 and 162, respectively. Table 2 describes the structure, number of VMs, and availability requirements of the applications. VM CPU and RAM demands were randomly generated using a uniform distribution with values ranging from 2-5 units. We submitted each request in Table 2 separately to the availability-aware application deployment module to deploy the applications and return the VM placement plan. We used the MOAntColony algorithm with a set number of iterations and ants for VM placement. We compared the placement results generated by AvAAD algorithm with three other baseline algorithms from the literature, CHASE [4], Convolutional Neural Network (CNN) [51], and FirstFit. We selected these algorithms based on their awareness of application availability during VM placement. CHASE is aware of application availability and is very close to our work, CNN is not aware but we incorporate the application availability into it, and FirstFit is unaware. Based on our best knowledge, existing VMs placement algorithms do not consider application availability as an objective. Table 3 summarizes the parameters used in the experiments. We simulated VMs placement for 1000 applications with their availability requirements and achieved ones after their placement at DC to train and test CNN.

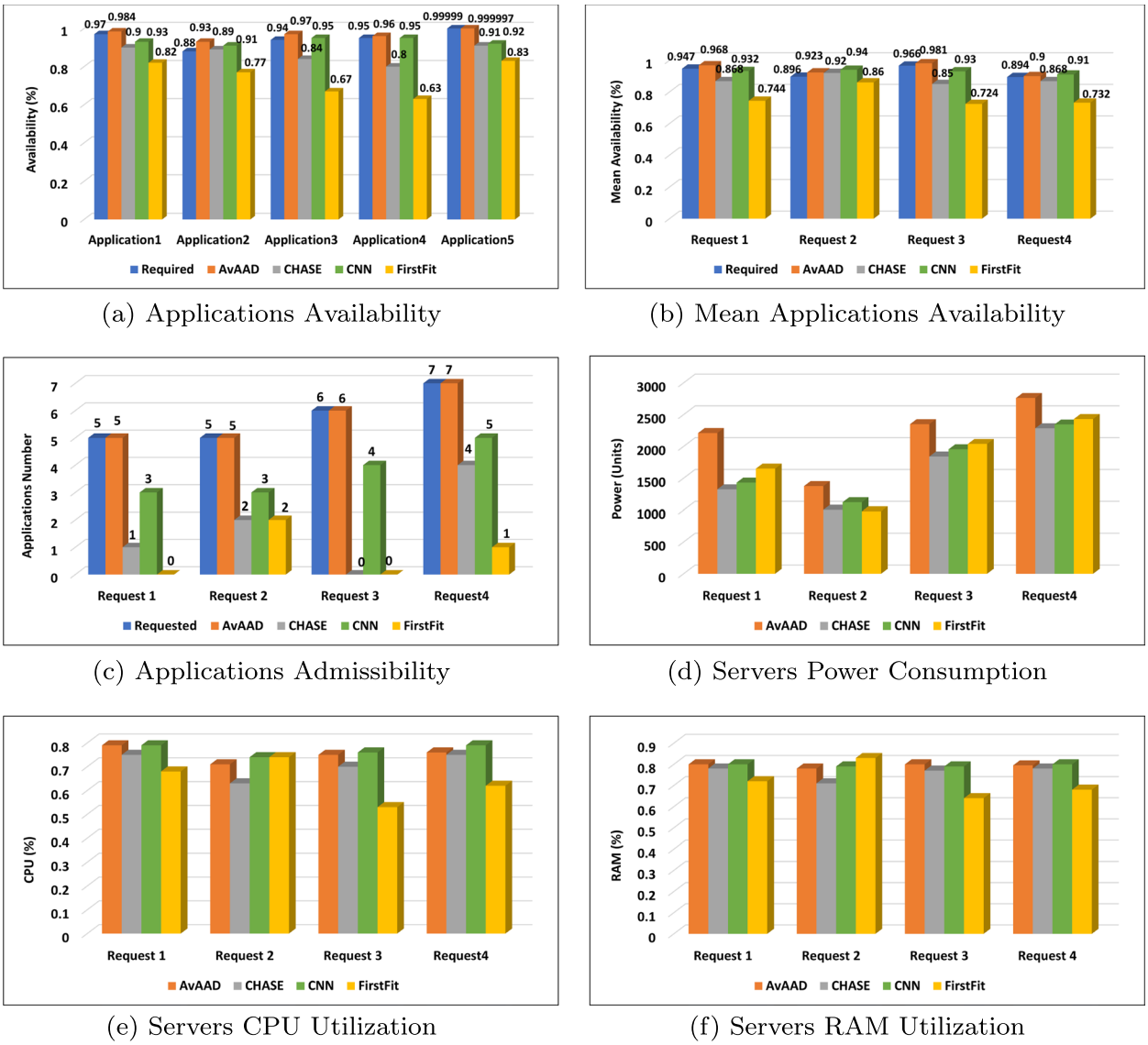
We conducted an availability comparison of the applications deployed by the proposed AvAAD algorithm and three other VM placement baseline algorithms, CHASE, CNN and FirstFit, to evaluate the ability of the deployment module to deploy applications while satisfying their availability requirements. We computed the availability of all applications after deployment and compared it with the requested availability by ASPs. For request 1, Fig. 3a shows the achieved availability by the suggested placement for 5 applications of request 1 in Table 2 by each algorithm. As Fig. 3a shows, AvAAD algorithm met the availability requirements because it is greater than or equal to the requested availability for all of the requested applications, CHASE algorithm violated the availability requirements of 4 applications out of 5 requested ones, CNN algorithm met the availability of 3 applications out of requested applications, and FirstFit did not meet any availability requirement for any requested applications. Application admissibility refers to the ability of hosting (placement) the application and meet its requirements including the requested availability at the data center. If the application meets its requirements we count it as admitted based on its suggested placement by each algorithm. In other words, the availability plays a major decision to admit or reject the application. As Fig. 3c shows, for request 1, AvAAD admitted all the

Table 2 Description of applications requests - group 1

	Req Availability	Funct No	VMs No
Request 1			
Application 1	0.97	3	3
Application 2	0.88	4	4
Application 3	0.94	5	5
Application 4	0.95	6	6
Application 5	0.99999	3	3
Request 2			
Application 1	0.95	3	3
Application 2	0.93	3	3
Application 3	0.98	2	2
Application 4	0.8	2	2
Application 5	0.82	2	2
Request 3			
Application 1	0.97	4	4
Application 2	0.98	4	4
Application 3	0.96	4	4
Application 4	0.95	5	5
Application 5	0.98	4	5
Application 6	0.96	6	6
Request 4			
Application 1	0.85	3	3
Application 2	0.87	4	4
Application 3	0.83	3	3
Application 4	0.8	4	4
Application 5	0.99	5	5
Application 6	0.96	5	5
Application 7	0.98	6	6

Table 3 Parameters used in the experiments

Parameter	Value
A_j^s	0.6-0.99
pactive	215
pidle	162
T	10 and 15
Z	12, 16, 20, 24, and 28
taks	1000000
epochs	100



(a) Applications Availability

(b) Mean Applications Availability

(c) Applications Admissibility

(d) Servers Power Consumption

(e) Servers CPU Utilization

(f) Servers RAM Utilization

Fig. 3 Evaluation of Availability-Aware Application Deployment Module - Group 1

5 requested applications since it met their availability, CHASE admitted 1 application and violated 4 applications out of 5 requested, CNN admitted 3 and violated 2 out of 5 requested, and FirstFit did not admit any application since it violated their requested availability. AvAAD algorithm is completely aware of the requested application availability, so it searches for any possible VMs placement for application to meet its availability requirement. CHASE tries to select servers that have maximum availability to host the VMs, but it does consider the entire application availability. So CHASE can assign VMs of application with low availability requirement at high available servers, and assign VMs of application with high requested availability at low available servers. CNN learns from the previous and historical applications that are hosted on the same DC, so it is trained and hence can predict the requested availability and place the VMs of the application accordingly. Therefore CNN achieved good results compare to AvAAD. FirstFit is not aware at all of the application availability, it places the VM on the first available server. Therefore, FirstFit achieved worst results in terms of availability. We selected FirstFit algorithm to emphasize the point that the existing VM placement algorithms do not consider application availability, which can have impact on quality of the application service and experience of the end users. For all requests in the first group, Fig. 3b shows that both AvAAD and CNN achieved mean availability close to the mean of the

required availability of the applications, while CHASE and FirstFit achieved mean availability far from the required ones.

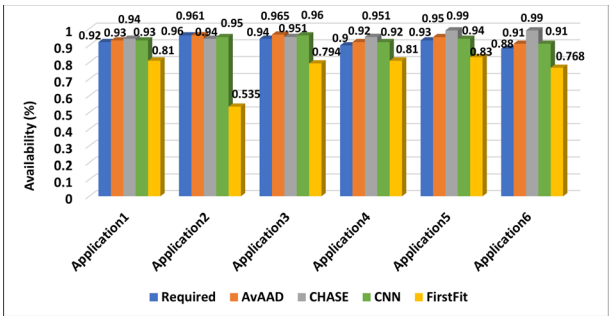
In order to evaluate the performance of the servers in the data center with various placement algorithms, we conducted an analysis of the mean power consumption of the servers that host VMs of the requested applications for first and second groups. As seen in Fig. 3d, AvAAD has a higher power consumption compared to CHASE, CNN and FirstFit algorithms. This can be attributed to approach of AvAAD by adding extra standby VMs to meet the availability requirements of only those applications that violate their availability. Consequently, the additional standby VMs consume more power, contributing to a higher overall power consumption. We also computed the mean CPU and RAM utilization of the servers after the deployment of the applications for each request. Figure 3e and f show the CPU and RAM utilization of the servers, respectively. We consider utilization of the resources as indicator for usage of the resources. The more resources utilization the better usage and lower wastage. AvAAD achieved stable and high CPU and RAM utilization, as one of its primary objectives is to minimize wastage of the resources. It is worth noting that the CPU and RAM utilization of AvAAD does not exceed the 80% utilization ratio, unlike the other algorithms, which sometimes exceed this ratio for certain requests. This is because we have set an upper threshold of 80% for both CPU and RAM utilization to prevent any server from reaching a full state of VMs, which could have a negative impact on the availability of the server as well as its performance.

In the second group of experiments, we included standby virtual machines (VMs) in the applications to recover the application service in case of active VM failure. The structure of the applications in the second group is described in Table 4, and we maintained the same VM and server properties as in the first group of experiments, except that we randomly generated availability values for servers using a uniform random distribution with a new range of 0.6-0.9, for illustrative purposes. Figure 4a displays the availability achieved by each placement algorithm for the six applications that belong to request number 6 of Group 2 in 4. The AvAAD algorithm can satisfy availability requirements for applications without adding standby VMs, which helps reducing the overall power consumption in the data center, as shown in Fig. 4d. The CNN and CHASE algorithm could satisfy availability requirements for most requested applications because standby VMs are present and they target availability during the VM placement process. Still FirstFit algorithm violates availability requirements for most of the requested applications because its approach that is not aware of the availability concept. Therefore, applications admissibility is high for the algorithms except for FirstFit as shown in Fig. 4c. As shown in Fig. 4d, using AvAAD and CNN result in servers consuming less power compared to using CHASE and FirstFit, as AvAAD does not require the addition of extra standby VMs for the protection approach. CNN searches for similar applications that have lower power consumption and can meet the requested availability.

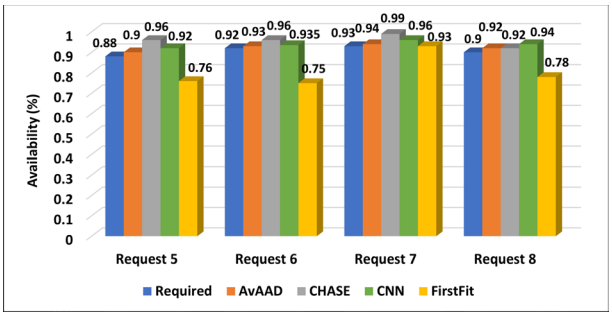
Table 4 Description of applications requests - group 2

	Req Availability	Funct No	VMs No
Request 5			
Application 1	0.9	2	4
Application 2	0.88	2	4
Application 3	0.93	2	4
Application 4	0.87	2	4
Application 5	0.85	2	4
Request 6			
Application 1	0.92	3	6
Application 2	0.96	3	6
Application 3	0.94	3	6
Application 4	0.9	3	6
Application 5	0.93	3	6
Application 6	0.88	3	6
Request 7			
Application 1	0.95	3	9
Application 2	0.93	3	9
Application 3	0.94	3	9
Application 4	0.89	3	9
Application 5	0.93	3	9
Application 6	0.94	3	9
Request 8			
Application 1	0.9	4	8
Application 2	0.94	4	8
Application 3	0.91	4	8
Application 4	0.95	4	8
Application 5	0.85	4	8
Application 6	0.9	4	8
Application 7	0.94	4	8

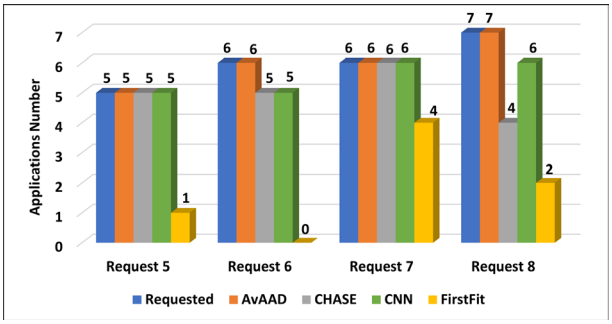
We conducted a performance comparison of the four VM placement algorithms by measuring the execution time required to place different large sets of VMs, ranging from 500 to 3000. The results are presented in Fig. 5a. AvAAD algorithm outperformed CHASE in terms of execution time, took around 1700 seconds to place 500 VMs while CHASE required around 2215 seconds to do the same. This is because CHASE requires optimization solver to find solution that maximizes the availability of the hosted VMs which usually consumes extra time to find final solution. Still AvAAD takes a long time to



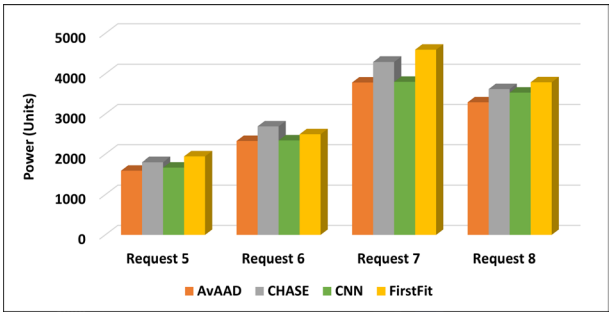
(a) Applications Availability



(b) Mean Applications Availability



(c) Applications Admissibility



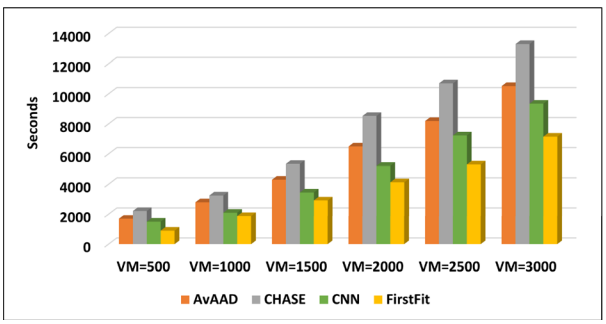
(d) Servers Power Consumption

Fig. 4 Evaluation of Availability-Aware Application Deployment Module - Group 2

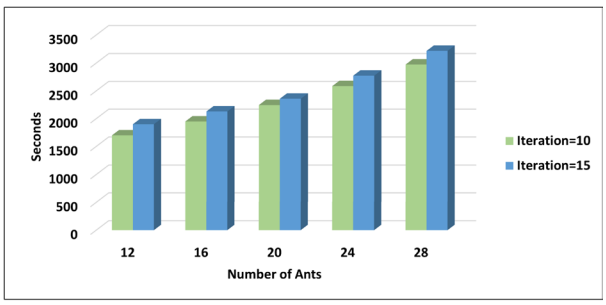
find placement solution for large set of VMs and this is because it uses meta-heuristic AntColony to find initial placement of VMs for all the requested application that consumes more time. On the other hand, CNN took less time than both AvAAD and CHASE because it only requires to predict the placement of the VMs based on historical dataset. However, FirstFit algorithm was the fastest taking less than a second to place the same number of VMs. This is because FirstFit only looks for the first available server that can host the current VM.

To see the impact of AntColony on the performance of AvAAD algorithms, we measure the execution time of AvAAD using different number of ants of two different number of iterations 10 and 15 for placement 500 VMs. As Fig. 5b shows, the execution time increases by increasing number of ants and iterations. For example, AvAAD took around 1700 seconds to place 500 VMs for 10 iterations using 12 ants, while took around 1900 seconds with the 15 iterations to place the same number of VMs using the same number of ants.

To evaluate the effectiveness of the Artificial Neural Network (ANN) prediction method for application task failure, we utilized the same ANN structure as [45]. Our training and testing process employed a cleaned dataset

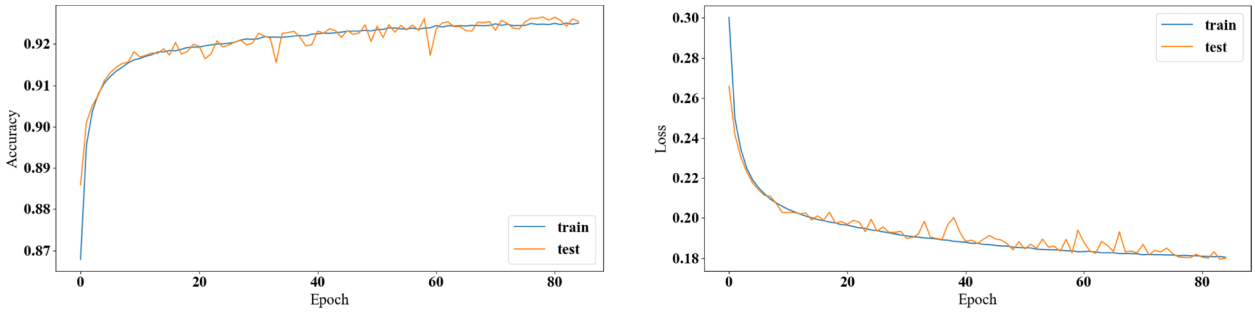


(a) AvAAD Execution Time



(b) AvAAD Execution Time-Variable Ants and Iterations

Fig. 5 Execution Time of VM Placement Algorithms



(a) Accuracy of ANN (b) Error Loss ANN

Fig. 6 Evaluation of Proactive Application Failure Detection Module

containing 1 million tasks over 100 epochs, with 500,000 tasks marked as "finished" and the other half marked as "failed". The accuracy and error loss of the ANN method were computed as illustrated in Fig. 6a and b, respectively. Accuracy denotes the percentage of correct predictions for task termination status. The ANN achieved a high accuracy of up to 93%, whereas the error loss was low, up to 14%.

Conclusion

In this research, we address the challenge of the dynamic placement of virtual machines (VMs) in the cloud, with a focus on ensuring application availability. To achieve this, we formalize the concept of application availability and model the dynamic VM placement problem as an INLP model with multiple objectives and a set of constraints. We propose a comprehensive framework that includes three modules to handle VM placement during deployment, failure, and scaling requests. The deployment module uses an AntColony optimization algorithm and a VM standby protection approach to achieve multiple objectives and satisfy the availability requirements of the applications. The results demonstrate that our proposed VM placement algorithm outperforms CHASE, CNN and FirstFit algorithms in terms of application service availability, accommodating higher number of applications, and CPU and RAM utilization. The prediction module of our framework employs deep learning ANN to predict application task failure, with an accuracy of up to 93% and a low error loss of up to 14%. Finally, the dynamic application reconfiguration module of the framework uses a heuristic approach to migrate VMs during scaling up requests. The migration solution is capable of migrating VMs with a lower migration time without compromising the availability requirements of the applications.

As future work, we plan to validate the overall performance of our proposed framework MoVPAAC including a large dataset and check for the possible comparisons with more existing methods from the literature. For example, the communication cost between the modules of the framework has a room for validation. In addition, we plan to incorporate the concept of application availability into existing cloud simulators such as CloudSim and validate our work using it.

Abbreviations

A	Set of requested applications for deployment
AV_j^s	Availability of server of index j
AV_f^{func}	Availability of functionality of index f
app_a	Application of index a
α	Parameter to control pheromone trail importance
β	Factor with value = 0.00001
$Fail_v^{vm}$	Failure of vm of index v
$Fail_j^s$	Failure of server of index j
$Fail^j(sol_0)$	Normalized failures ratio of serves located in solution sol_0
L_j^c	Normalized remaining CPU utilization of server of index j
L_j^r	Normalized remaining RAM utilization of server of index j
M	Number of servers that are used in solution sol_0
$MTTF_j$	Mean time to fail server of index j
$MTTR_j$	Mean time to repair server of index j
$\eta_{i,j}$	Desirability of placement vm of index i at server of index j
$\eta_{i,j,1}$	Contribution to the first objective of placement vm_i at s_j
$\eta_{i,j,2}$	Contribution to the second objective of placement vm_i at s_j
$\eta_{i,j,3}$	Contribution to the third objective of placement vm_i at s_j
ρ_l	Local pheromone evaporating parameter
ρ_g	Global pheromone evaporating parameter
λ	Coefficient
P_j	Average power consumption of server j
P_j^{active}	Average power consumption of server j when it is active
P_j^{idle}	Average power consumption of server j when it is idle
P_j^{max}	Maximum power consumption of server of index j
$Pr_{u,j}$	Probability placement vm of index u at server of index j
PS	Pareto set
q	Random number with value between 0 and 1
q_0	Constant number with value between 0 and 1
R_i^c	CPU demand of vm of index i
R_i^r	RAM demand of vm of index i
S	Set of servers located at data center
s_j	Server of index j
sol_0	Initial placement solution that is used by MOAntColony
sol_g	Global solution in PS
T	Number of iterations that are used by MOAntColony
T_j^c	Upper threshold of CPU utilization of server of index j
T_j^d	Upper threshold of Disk utilization of server of index j
T_j^r	Upper threshold of RAM utilization of server of index j
T^g	Number of the iterations to find the global solution
τ_0	Initial local pheromone level

$\tau_{i,j}$	Local pheromone of placement vm of index *i* at server *j*
U	Set of VMs that can be hosted at server of index *j*
U_j^c	Normalized CPU utilization of server of index *j*
U_{jd}	Normalized Disk utilization of server of index *j*
U_j^r	Normalized RAM utilization of server of index *j*
V	Set of VMs that compose all the requested applications
V_a^{app}	Set of VMs that provide application of index *a*
V_f^{func}	Set of VMs that provide functionality of index *f*
vm_i	Virtual machine of index *i*
W_j	Average resources wastage of server of index *j*
$W'(sol_0)$	Normalized resources wastage of servers in solution sol_0
x_{ij}	Binary decision variable if vm_i is hosted at s_j or not
y_j	Binary decision variable if server s_j is active or not
Z	Number of ants that are used by MOAntColony
z_{id}	Binary variable if vm_i is migrated to server *d* or not

Authors' contributions
All authors have participated in conception and design, drafting the article and revising it critically for important intellectual content. All authors read and approved the final manuscript.

Funding
This declaration is "not applicable".

Availability of data and materials
Any required data or material will be available upon request via email yanal.alahmad@concordia.ca.

Declarations

Ethics approval and consent to participate
This declaration is "not applicable".

Competing interests
The authors declare no competing interests.

Received: 23 July 2023 Accepted: 6 February 2024
Published online: 17 February 2024

References

1. Siewiorek D, Gray J (1991) High-availability computer systems. Computer 24(09):39–48
2. The Cost of Service Downtime. https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime. Accessed 8 Sep 2021
3. Jammal M, Kanso A, Shami A (2015) High availability-aware optimization digest for applications deployment in cloud. In: 2015 IEEE International Conference on Communications (ICC), IEEE, pp 6822–6828
4. Jammal M, Kanso A, Shami A (2015) CHASE: Component High Availability-Aware Scheduler in Cloud Computing Environment. In: 2015 IEEE 8th International Conference on Cloud Computing, IEEE, pp 477–484
5. Zhu H, Huang C (2017) Availability-Aware Mobile Edge Application Placement in 5G Networks. In: GLOBECOM 2017 - 2017 IEEE Global Communications Conference, IEEE, pp 1–6
6. Lera I, Guerrero C, Juiz C (2019) Availability-Aware Service Placement Policy in Fog Computing Based on Graph Partitions. IEEE Internet Things J 6(2):3641–3651
7. Dehury CK, Sahoo PK, Veeravalli B (2021) RRFT: A Rank-Based Resource Aware Fault Tolerant Strategy for Cloud Platforms. IEEE Trans Cloud Comput 11(2) (2003)
8. Yang S, Wieder P, Yahyapour R (2016) Reliable Virtual Machine placement in distributed clouds. In: 2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM), IEEE, pp 267–273
9. Liu X, Cheng B, Yue Y, Wang M, Li B, Chen J (2019) Traffic-Aware and Reliability-Guaranteed Virtual Machine Placement Optimization in Cloud Datacenters. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), IEEE, pp 91–98
10. Yang Z, Liu L, Qiao C, Das S, Ramesh R, Du AY (2015) Availability-aware energy-efficient virtual machine placement. In: 2015 IEEE International Conference on Communications (ICC), IEEE, pp 5853–5858
11. Ayoubi S, Zhang Y, Assi C (2016) A Reliable Embedding Framework for Elastic Virtualized Services in the Cloud. IEEE Trans Netw Serv Manag 13(3):489–503
12. Alahmad Y, Agarwal A, Daradkeh T (2020) Cost and Availability-Aware VNF Selection and Placement for Network Services in NFV. In: 2020 International Symposium on Networks, Computers and Communications (ISNCC), IEEE, pp 1–6
13. Thiruvasagam PK, Chakraborty A, Mathew A, Murthy CSR (2021) Reliable Placement of Service Function Chains and Virtual Monitoring Functions With Minimal Cost in Softwarized 5G Networks. IEEE Trans Netw Serv Manag 18(2):1491–1507
14. Yala L, Frangoudis PA, Lucarelli G, Ksentini A (2018) Cost and Availability Aware Resource Allocation and Virtual Function Placement for CDNaaS Provision. IEEE Trans Netw Serv Manag 15(4):1334–1348
15. Yang B, Xu Z, Chai W, Liang W, Tuncer D, Galis A, Pavlou G (2018) Algorithms for Fault-Tolerant Placement of Stateful Virtualized Network Functions. In: 2018 IEEE International Conference on Communications (ICC), IEEE, pp 1–7
16. Xu Y, Kafle VP (2019) An Availability-Enhanced Service Function Chain Placement Scheme in Network Function Virtualization. J Sensor Actuator Networks 8(2):34
17. Sharma S, Kushwaha A, Somani A, Gumaste A (2019) Designing Highly-Available Service Provider Networks with NFV Components. In: 2019 28th International Conference on Computer Communication and Networks (ICCCN), IEEE, pp 1–9
18. Abdelaal MA, Ebrahim GA, Anis WR (2021) High Availability Deployment of Virtual Network Function Forwarding Graph in Cloud Computing Environments. IEEE Access 9:53861–53884
19. Mao W, Wang L, Zhao J, Xu Y (2020) Online Fault-tolerant VNF Chain Placement: A Deep Reinforcement Learning Approach. In: 2020 IFIP Networking Conference (Networking), IEEE, pp 163–171
20. Li X, Qian C (2015) Traffic and failure aware VM placement for multi-tenant cloud computing. In: 2015 IEEE 23rd International Symposium on Quality of Service (IWQoS), IEEE, pp 41–50
21. Jammal M, Hawilo H, Kanso A, Shami A (2016) Mitigating the Risk of Cloud Services Downtime Using Live Migration and High Availability-Aware Placement. In: 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, pp 578–583
22. Zhou A, Wang S, Cheng B, Zheng Z, Yang F, Chang RN, Lyu MR, Buyya R (2017) Cloud Service Reliability Enhancement via Virtual Machine Placement Optimization. IEEE Trans Serv Comput 10(6):902–913
23. Zhou A, Wang S, Hsu CH et al (2019) Virtual machine placement with (m, n)-fault tolerance in cloud data center. Cluster Comput 22:11619–11631
24. Gonzalez C, Tang B (2020) FT-VMP: Fault-Tolerant Virtual Machine Placement in Cloud Data Centers. In: 2020 29th International Conference on Computer Communications and Networks (ICCCN), IEEE
25. Alameddine HA, Ayoubi S, Assi C (2017) An Efficient Survivable Design With Bandwidth Guarantees for Multi-Tenant Cloud Networks. IEEE Trans Netw Serv Manag 14(2):357–372
26. Chen X, Jiang J (2016) A method of virtual machine placement for fault-tolerant cloud applications. Intell Autom Soft Comput 22:587–597
27. Zhang W, Chen X, Jiang J (2021) A multi-objective optimization method of initial virtual machine fault-tolerant placement for star topological data centers of cloud systems. Tsinghua Sci Technol 26(1):95–111
28. Tran M-N, Vu XT, Kim Y (2022) Proactive Stateful Fault-Tolerant System for Kubernetes Containerized Services. IEEE Access. 10:102181–102194
29. Saxena D, Gupta I, Singh AK, Lee C-N (2022) A Fault Tolerant Elastic Resource Management Framework Toward High Availability of Cloud Services. IEEE Trans Netw Serv Manag 19(3):3048–3061
30. Guo P, Liu M, Wu J, Xue Z, He X (2018) Energy-Efficient Fault-Tolerant Scheduling Algorithm for Real-Time Tasks in Cloud-Based 5G Networks. IEEE Access 6:53671–53683
31. Marahatta A, Wang Y, Zhang F, Kumar A, Tyagi SS, Liu Z (2018) Energy-Aware Fault-Tolerant Dynamic Task Scheduling Scheme for Virtualized Cloud Data Centers. Mob Netw Appl 24:1–15
32. Sun H, Yu H, Fan G, Chen L (2020) QoS-Aware Task Placement With Fault-Tolerance in the Edge-Cloud. IEEE Access 8:77987–78003
33. Yao G, Li X, Ren Q, Ruiz R (2022) Failure-aware Elastic Cloud Workflow Scheduling. IEEE Transactions on Services Computing, pp. 1–14

34. Yao G, Ren Q, Li X, Zhao S, Ruiz R (2022) A Hybrid Fault-Tolerant Scheduling for Deadline-Constrained Tasks in Cloud Systems. IEEE Trans Serv Comput 15(3):1371–1384
35. Weikert D, Steup C, Mostaghim S (2022) Availability-Aware Multiobjective Task Allocation Algorithm for Internet of Things Networks. IEEE Internet Things J 9(15):12945–12953
36. Reiss C, Wilkes J, Hellerstein JL (2011) Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View
37. Chen X, Lu C, Pattabiraman K (2014) Failure Prediction of Jobs in Compute Clouds: A Google Cluster Case Study. In: 2014 IEEE International Symposium on Software Reliability Engineering Workshops, IEEE, pp 341–346
38. Soualhia M, Khomh F, Tahar S (2015) Predicting Scheduling Failures in the Cloud: A Case Study with Google Clusters and Hadoop on Amazon EMR. In: 2015 IEEE 17th International Conference on High Performance Computing and Communications, IEEE, pp 58–65
39. Jassas MS, Mahmoud QH (2019) Failure Characterization and Prediction of Scheduling Jobs in Google Cluster Traces. In: 2019 IEEE 10th GCC Conference & Exhibition (GCC), IEEE, pp 1–7
40. Jassas MS, Mahmoud QH (2020) Evaluation of a failure prediction model for large scale cloud applications. In: Canadian Conference on Artificial Intelligence. Springer, pp 321–327
41. Islam T, Manivannan D (2017) Predicting Application Failure in Cloud: A Machine Learning Approach. In: 2017 IEEE International Conference on Cognitive Computing (ICCC), IEEE, pp 24–31
42. Rosá A, Chen LY, Binder W (2015) Predicting and Mitigating Jobs Failures in Big Data Clusters. In: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, IEEE, pp 221–230
43. Islam T, Manivannan D (2019) FaCS: Toward a Fault-Tolerant Cloud Scheduler Leveraging Long Short-Term Memory Network. In: 2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/ 2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom), IEEE, pp 1–6
44. Soualhia M, Khomh F, Tahar S (2020) A Dynamic and Failure-Aware Task Scheduling Framework for Hadoop. IEEE Trans Cloud Comput 8(2):553–569
45. Alahmad Y, Daradkeh T, Agarwal A (2021) Proactive Failure-Aware Task Scheduling Framework for Cloud Computing. IEEE Access 9:106152–106168
46. Chen X et al (2020) A WOA-Based Optimization Approach for Task Scheduling in Cloud Computing Systems. IEEE Syst J 14(3):3117–3128
47. Cheng L, Wang Y, Cheng F, Liu C, Zhao Z, Wang Y (2023) A Deep Reinforcement Learning-Based Preemptive Approach for Cost-Aware Cloud Job Scheduling. IEEE Trans Sustain Comput (2003):1–12
48. Zhang J, Cheng L, Liu C, Zhao Z, Mao Y (2023) Cost-aware scheduling systems for real-time workflows in cloud: An approach based on Genetic Algorithm and Deep Reinforcement Learning. Expert Syst Appl 234(2023):120972
49. Fan X, Weber W, Barroso L (2007) Power provisioning for a warehouse-sized computer. In: the 34th Annual International Symposium on Computer Architecture, ACM SIGARCH computer architecture news, pp 13–23
50. Ashraf A, Porres I (2018) Multi-objective dynamic virtual machine consolidation in the cloud using ant colony system. Int J Parallel Emergent Distrib Syst 33(1):103–120
51. Abdelsalam M, Krishnan R, Sandhu R (2019) Online malware detection in cloud auto-scaling systems using shallow convolutional neural networks. In: IFIP Annual Conference on Data and Applications Security and Privacy. Springer, pp 381–397

Publisher's Note