

# AutoMigrate: a framework for developing intelligent, self-managing cloud services with maximum availability

Mamadou H. Diallo<sup>1</sup> · Michael August<sup>1</sup> · Roger Hallman<sup>1</sup> · Megan Kline<sup>1</sup> · Scott M. Slayback<sup>1</sup> · Christopher Graves<sup>1</sup>

Received: 1 April 2017 / Accepted: 2 May 2017 / Published online: 29 May 2017  
© Springer Science+Business Media New York (outside the USA) 2017

**Abstract** Cloud services are on-demand services provided to end-users over the Internet and hosted by cloud service providers. A cloud service consists of a set of interacting applications/processes running on one or more interconnected VMs. Organizations are increasingly using cloud services as a cost-effective means for outsourcing their IT departments. However, cloud service availability is not guaranteed by cloud service providers, especially in the event of anomalous circumstances that spontaneously disrupt availability including natural disasters, power failure, and cybersecurity attacks. In this paper, we propose a framework for developing intelligent systems that can monitor and migrate cloud services to maximize their availability in case of cloud disruption. The framework connects an autonomic computing agent to the cloud to automatically migrate cloud services based on anticipated cloud disruption. The autonomic agent employs a modular design to facilitate the incorporation of different techniques for deciding when to migrate cloud services, what cloud services to migrate, and where to migrate the selected cloud services. We incorpo-

rated a virtual machine selection algorithm for deciding what cloud services to migrate that maximizes the availability of high priority services during migration under time and network bandwidth constraints. We implemented the framework and conducted experiments to evaluate the performance of the underlying techniques. Based on the experiments, the use of this framework results in less down-time due to migration, thereby leading to reduced cloud service disruption.

**Keywords** Cloud computing · Anomaly detection · Self-managing cloud services · Intelligent systems · Cloud service availability · Live migration of virtual machines

## 1 Introduction

Virtualization is at the core of cloud infrastructure, enabling portability of server operating systems. Services hosted in the cloud are typically deployed as sets of applications/processes running on one or more virtual machines (VMs). A cluster of VMs running multiple applications associated with a single service can then be moved all together as one unit from one subnet to another, thereby ensuring co-location of the VMs hosting a service and minimizing latency between VMs [1]. Virtual machine clusters also enable cloud service orchestration through high level cloud management tools, thereby simplifying deployment, provisioning, configuration, and scalability of cloud-based services.

Live migration of VMs has been used in various tasks, including IT maintenance [2–5] (e.g., by transparently migrating VMs off of a host which will be brought down for maintenance), load balancing [6–9] (e.g., by migrating VMs off of a congested host to a machine with a lower CPU or I/O load), power management [10–14] (e.g., by migrating VMs from multiple servers onto fewer servers

---

✉ Mamadou H. Diallo  
mamadou.h.diallo@navy.mil

Michael August  
michael.august@navy.mil

Roger Hallman  
roger.hallman@navy.mil

Megan Kline  
megan.kline@navy.mil

Scott M. Slayback  
scott.m.slayback@navy.mil

Christopher Graves  
christopher.t.grave@navy.mil

<sup>1</sup> US Department of Defense SPAWAR Systems Center Pacific (SSC Pacific), San Diego, CA, USA

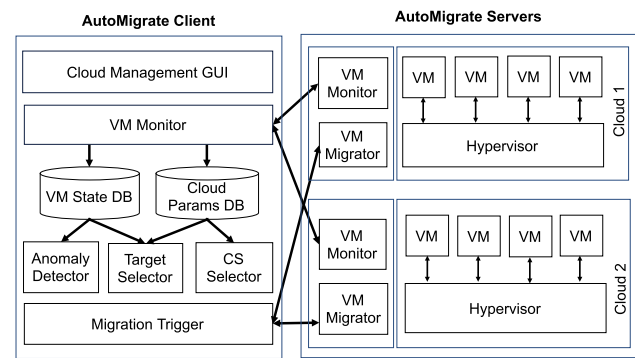
in order to reduce data center power consumption), and development-to-operations support (e.g., by migrating VMs residing in the development environment over to the test environment, and then to the operational environment) [15]. In this paper, we consider the use of live migration as a mechanism for improving resilience/availability of cloud services.

We assume a cloud infrastructure as a service (IaaS) model where cloud service providers manage virtual machine instances and offer them as a service to customers. When there is an anomaly in a cloud infrastructure that can result in disruption of the cloud (i.e., the cloud servers are no longer functional), then VMs will need to be migrated to preserve the availability of the services they are providing. However, migrating a large number of VMs can take a long time, which users may not have since the cloud is under disruption [16–18]. For instance, to migrate a 2 GB VM from source to destination host in the same subnet with reasonable bandwidth can take tens of seconds (Fig. 2). To address this problem, VMs need to be selected for migration based on how valuable they are to their owner. As a result, the priorities of cloud services should be used in determining which ones should be migrated so as to maximize the availability of the highest priority services. Therefore, in this work we focus on maximizing the availability of high priority cloud services when the cloud is under disruption.

In this paper, we propose a framework for developing intelligent systems that can monitor and migrate cloud services to maximize their availability under cloud disruption. The overall objective of this framework is to identify what cloud services should be migrated, when these cloud services need to be migrated, and where these services should be migrated to. This framework takes into account time, memory, and bandwidth constraints in its decision making process. At the heart of the framework is a VM Monitor, which keeps track of each VM's state and resource usage over time, and a Decision Agent, which uses the data collected from the VM Monitor to intelligently decide when to migrate cloud services, what cloud services to migrate, and where to migrate these cloud services.

In order to determine when to migrate, the framework facilitates the incorporation of algorithms for automatically detecting anomalies to trigger live migration of cloud services. As a proof of concept, we implemented a machine learning approach to detecting behavioral anomalies in VM resource usage (CPU, memory, disk, and network usage).

In order to determine what to migrate, the framework includes a virtual machine selection algorithm that maximizes the availability of high priority services during migration under time and network bandwidth constraints.



**Fig. 1** Architecture diagram

In order to determine where to migrate, the monitor provides information that can be used to determine the best location for migration.

To simplify the discussion, we will assume that each user's environment is monitored and controlled by only one instance of the *AutoMigrate* framework at a time, instead of multiple concurrent instances.

We implemented the framework and conducted experiments to evaluate the performance of the underlying techniques. The experiments compared the performance of three algorithmic approaches to selecting VMs for automated live migration: basic greedy selection, standard greedy selection, and improved greedy selection. We compared how well each of these three algorithms preserved the availability of high priority cloud services. The results of our comparison show that the improved greedy selection algorithm outperforms the standard greedy algorithm, which in turn performs better than the basic algorithm.

This paper is organized as follows: In Sect. 2, we discuss our proposed framework for automating live migration of VMs under time and bandwidth constraints. In Sect. 3, we give an overview of the decision agents used by the *AutoMigrate* framework when deciding when, what, and where to migrate. In Sect. 4, we describe the problem of VM selection to maximize the availability of cloud services. In Sect. 5, we present a solution to the selection problem based on the set-union knapsack problem (SUKP). In Sect. 6, we discuss the implementation and experimental results. In Sect. 7, we highlight related works. Finally, in Sect. 8 we present our conclusions and future work.

## 2 AutoMigrate framework overview

In this section, we describe the overall design of the *AutoMigrate* framework, focusing particularly on the different components of the framework, and how these components interact to automate the different tasks associated with the live migration process. Figure 1 shows the overall architecture of the framework, which is based on a modular design pattern.

## 2.1 Components of AutoMigrate

The *AutoMigrate* framework uses a client/server architecture. The client can be deployed in the cloud or on a local machine. The server is typically deployed in the cloud, co-located with the VMs under observation. The client-side components are: the *VM Monitor*, *Anomaly Detector*, *CS Selector*, *Target Selector*, and the *Migration Trigger*. Taken together, the *Anomaly Detector*, *CS Selector*, and *Target Selector* make up the *Decision Agent* within the system. Each of these components is dynamic, continually running in the background to determine when to trigger the migration process, to determine which VMs should be migrated at any given time to maximize the availability of the services they are providing, and to determine suitable destinations for the services being migrated. The combination of these components creates an autonomic system which is aware of the state of the VMs, the importance of services hosted on VMs, and the dependencies between services and VMs. Note that a cloud service can consist of multiple applications/processes running on one or more VMs. This autonomic system can then adapt to changes in the environment automatically with minimal user intervention.

The *VM Monitor* on the client-side is used to keep track of the behavior of each VM in the cloud. It communicates with the server-side *VM Monitor* to collect and store data about each VM running in the cloud. This includes checking the status of the VMs (on or off), and the status of cloud services (running or stopped). If a VM is on, it can also determine the CPU, memory, disk, and network utilization. The *VM Monitor* also facilitates the visualization of the collected data through the web-based *Cloud Management GUI*.

The *Anomaly Detector* is responsible for detecting anomalies, automatically or manually through user inputs, when a cloud is facing disruption. An anomaly can affect a single VM, a single hypervisor, or an entire data center. In our current approach, the *Anomaly Detector* can automatically detect anomalies based on the VM state data presented to it by the *VM Monitor*. It tries to predict what the future state of each VM will be. The *Anomaly Detector* can also incorporate other techniques for detecting anomalies. For instance, advanced intrusion detection and prevention systems (IDPS) can be used to detect attacks based on log analysis, attacks which can lead to anomalies. External anomalies such as natural disasters, which could lead to the disruption of cloud services, can also be manually provided to the framework by users through the *VM Monitor GUI*.

The *Cloud Service (CS) Selector* automatically selects candidate VMs for migration based on the priority of the services hosted across those VMs. Based on a comparison of the resource capacity of the available destination hosts, the *Target Selector* automatically determines an appropriate destination host for the migration of each selected VM.

The *Migration Trigger* takes the list of VMs to be migrated and the selected destination host and initiates the migration process for those VMs. It also provides a mechanism for users to manually initiate a migration in response to potential disruption of cloud services (e.g., a natural disaster impacting the cloud).

The server-side components are: the *VM Monitor* and the *VM Migrator*. The *VM Monitor* continuously monitors the state of each VM and the overall state of the cloud services. The client-side *VM Monitor* continuously pulls the data from the server-side *VM Monitor*, which is used by the *Anomaly Detector* and the *Target Selector* to analyze the data. The *VM Migrator* initiates migration of the specified VMs based on input from the client-side *Migration Trigger*. It also keeps track of the state of the migration process. If a VM fails to migrate, it can attempt to reinitiate the migration.

## 2.2 Operational view

In this section, we provide an end-to-end description of how the different components of *AutoMigrate* interact to autonomously perform live migration of VMs to preserve cloud service availability. This preservation may require live migration of multiple, interacting VMs. Migration operations are performed on individual VMs as supported by the underlying hypervisor. This means that a cloud service will not be available until after the last supporting VM has completed migration. If one of the VMs supporting a cloud service fails to migrate, then the service will not be fully available on the destination host. Applications are individual programs that run on VMs and each cloud service is composed of one or more applications. *AutoMigrate* uses applications to define the relationship between VMs and services. Note that any applications that are not part of any service still contribute to the memory footprint of a VM, but *AutoMigrate* does not need to address these applications individually.

We assume that the VMs are globally addressable and can be accessed via the Internet, which minimizes downtime. The client has an interface which a user can use to perform VM management tasks, including system initialization, configuration, and manual triggering of VM migration. From this user interface, the user can define the priority level of each cloud service as well as the mapping of cloud services across VMs. The user can also provide a list of available clouds and the preferred destination host for migration of VMs. For simplification, we assume that only one client is allowed to manage a set of VMs at a time.

Below is the operational workflow of *AutoMigrate*.

1. The user logs into the cloud management interface and starts up the *AutoMigrate* service, which consists of the server-side *VM Monitor* and *VM Migrator*.

2. The client-side *VM Monitor* connects to the server-side *VM Monitor* and initiates a request to pull VM state data.
3. The client-side *VM Monitor* stores the data in the *VM State DB* and makes it available to the user interface.
4. Periodically, the *Anomaly Detector* reads the *VM State DB* and performs behavioral analysis to detect the presence of anomalies. When an anomaly is detected, the *Anomaly Detector* sends a signal to the *Migration Trigger* to initiate the migration process.
5. The *CS Selector* takes as input the cloud parameters (set of VMs, set of cloud services, and the dependencies between them) from the *Cloud Params DB* and runs the VM selection algorithm to determine which VMs to migrate. The list of selected VMs is passed to the *Migration Trigger*.
6. The *Target Selector* takes as input the VM state and cloud parameters and determines the ideal destination host for the migrated VMs, which is passed to the *Migration Trigger*.
7. The *Migration Trigger* sends migration commands to the server-side *VM Migrator*. The list of VMs to be migrated and the destination host are sent along with the command.
8. During the migration process, the *Anomaly Detector* must disable itself in order to prevent any race conditions that could result from initiating redundant migrations of VMs. Once migration is complete, the *Anomaly Detector* resumes normal operations.

### 3 Decision agents overview

The *Decision Agent* consists of the *Anomaly Detector*, *CS Selector*, and *Target Selector*, which need to coordinate to achieve the *AutoMigrate*'s goal of migrating cloud services at the right time, with the right cloud services, and the right destination, to maximize availability. As mentioned previously, the *Anomaly Detector* is responsible for automatically determining when to initiate a migration of cloud services. The *CS Selector* is responsible for automatically selecting which cloud services to migrate once the decision to migrate has been made. The *Target Selector* is then responsible for automatically determining the destination hosts where the selected cloud services will be migrated to. These components rely on the metrics collected by the *VM Monitor* component to decide when, what, and where to migrate, respectively. These components need to be implemented using automated techniques in order to make these decisions. In this section, we describe the techniques that can be used by the *Anomaly Detector* and *Target Selector*. The remaining sections of the paper focus on the technique we propose for the *CS Selector*, which represents the main contribution of *AutoMigrate*. Before summarizing the techniques

the *Anomaly Detector* and *Target Selector*, we first describe the *VM Monitor* in more details.

#### 3.1 VM monitor

The state of a VM is a collection of metrics that characterize the utilization of the VM. The metrics include CPU, Memory, disk usage, and network usage. The OS running on the VM keeps track of all of these metrics. For instance, in Unix based OSes, the “top” Task Manager program monitors a number of metrics such as the CPU utilization, memory usage, and running state of each process running on the machine, as well as their aggregates for the whole machine. Our approach to collecting VM state metrics is through the use of a monitoring agent, called the *VM Monitor*, which continuously collects the metrics for each VM.

The *VM Monitor* supports two deployment models. The first deployment model makes use of an agent running on each VM in the cloud. The agent interacts directly with the OS to collect the metrics and reports them to the *VM Monitor*. This approach requires an agent to be deployed on each VM running in the cloud, which requires increased communication and coordination amongst the VMs.

The second model makes use of an agent running in the hypervisor. The hypervisor collects the metrics on the behalf of the *VM Monitor* and reports them to the *VM monitor*. Since a hypervisor can collect metrics on each of its underlying VMs, only one agent needs to be deployed on each hypervisor in the cloud, making this approach more lightweight than the first deployment model.

In our implementation, we make use of the second deployment model with a tool called “xentop”, which collects per VM metrics within the Xen hypervisor.

#### 3.2 Deciding when to migrate

Given the metrics of the VM, the anomaly detector needs to detect anomalous behavior. Machine Learning techniques can be used to detect anomalous behavior on the VMs running in the cloud based on the VM metrics. Below we describe three of such machine learning based techniques we researched with the objective of selecting the right solution for *AutoMigrate*.

The k-means clustering algorithm (KMeans) clusters data into a number of distinct groups using a distance measure. Using this algorithm, we first learn the normal behavior of each VM using the metrics collected from the VM. Then, using this historical data for each VM, data clusters are formed. As new data for the VM is collected, we can identify whether or not the VM is exhibiting anomalous behavior by comparing the current data to the historical data clusters. In the *AutoMigrate* anomaly detector, we use the k-means clustering to separate out the VM data into two groups—normal



behavior and anomalous behavior. There are many different libraries that implement this algorithm, making it easy to incorporate into AutoMigrate. However, since AutoMigrate can run many different types of services and different sizes of VMs, there may not be an easy way to get a good training set to train an algorithm to lead to accurate predictions.

Another anomaly detection library we looked at was the Twitter Anomaly Detection, developed by Twitter and written in R. It has an underlying algorithm called seasonal hybrid extreme studentized deviate (ESD). This library takes the underlying ESD algorithm and adapts it to detect anomalies on seasonal data. Twitter used this library to detect anomalies in number of tweets per day. The library was written such that it analyzes whole sets of data at once. To use this library, we would need to break up the data into discrete time chunks and run the algorithm over it. As a system that should be monitoring the VMs constantly this may not be ideal since we would need to wait for enough data to process it. However, since the library is open source we could modify the library to work using incremental data entry.

The last anomaly detection method we looked at is based on a Kalman Filter. This method was shown to take the data point we need (CPU, memory, etc.) and find anomalies in a time series of that data. The algorithm is iterative meaning that it processes the data points one at a time, not requiring entire sets of data. This algorithm converges to the correct answer so the algorithm learns from the VM and will provide accurate anomalies over time. There are a number of libraries that implement anomaly detection using the Kalman Filtering. In our initial attempt of implementing the anomaly detection in the anomaly detector, we extended the open source python module, `pykalman` [19]. The module implements two algorithms for tracking: the Kalman Filter and Kalman Smoother. From our preliminary findings, anomaly detection algorithms based on Kalman Filtering seem to be more promising in detecting accurately anomalies in VMs. Currently, we are performing more investigation on these algorithms to better understand their performance in terms of anomaly detection.

### 3.3 Deciding where to migrate

At the minimum, deciding where to migrate a set of selected cloud services requires taking into account the load of the cloud services in the source host as well as the availability of resources in the destination host. Ideally, the destination host needs to have enough memory to efficiently run the VMs hosting the cloud services, without compromising the availability of the cloud services. In addition, the availability of resources should be relatively stable. Otherwise, the probability of migrating the cloud services again in the near future may be high, which can lead to the degradation of availability due to frequent migration.

One basic technique for deciding where to migrate cloud services is to assess the resources of the available destination hosts after deciding what cloud services to migrate, and select the one that has the highest available resources. This is the technique we implemented with the current version of AutoMigrate. If all the destination hosts are less susceptible to anomalies and have constant resource usage, then this technique can result in a good performance. Migrating cloud services would be less frequent.

In order to improve this basic approach, techniques need to take into account the overall behavior of the destination hosts overtime. Automated techniques based on machine learning, which learn and predict the behavior of cloud hosts in the future will be necessary to guarantee maximum performance and availability of cloud services. Similar to the anomaly detection, we investigated a number of such techniques.

The first technique we considered is the autonomic management system (AMS). The AMS makes use of multiple managers to dynamically evaluate each system under management by collecting metrics such as workload, and memory usage. The AMS technique can be extended so that, in addition to getting global view of all the VMs in a given host, it can keep track of the historical view of the states of the VMs. In this case, a technique based on machine learning can be used to accurately predict the resource availability of the hosts for an extended period of time in the future. The prediction will be significant in making a better decision when choosing a destination host for migration. For instance, if snapshots of the states of all hosts are taken, one host may appear to have the highest resource available. But, when considering a period of time, the host might have the lowest available resource on average.

The second technique we investigated is the VM placement in a cloud infrastructure [20]. The VM placement techniques seeks the best physical machines that can host VMs to achieve desired factors affecting the data centers such as performance, resource utilization, and power consumption. In AutoMigrate, the goal of the Target Selector component is to maximize the cloud services availability after placing the VMs in the destination host. The proposed VM placement techniques are mainly based on the concepts of “First Fit”, “Next Fit”, “Random Fit”, “Least full first”, and “Most Full First”. Different VM placement schemes based on these concepts such as integer programming-based, constraint programming-based, and graph theory-based VM placement, have been proposed. These schemes can be classified as traffic-aware, energy-aware, cost-aware, resource-aware, and cost-aware [20]. The resource-aware schemes are more tailored to achieve the goal of the Target Selector in AutoMigrate. As a case study, we looked at the applicability of the backward speculative placement (BSP) VM placement algorithm proposed in [15]. In this schema, a monitor is used to collect historical demand traces of the deployed

VMs on a source host, and then the algorithm predicts the future behavior of a VM on a target host. This algorithm can potentially be used by the Target Selector of AutoMigrate to make the right decision in choosing where to migrate cloud services.

#### 4 VM selection as a knapsack problem

Managing the live migration of cloud services involves identifying which set of VMs to migrate, when a migration is needed, and where to migrate VMs. The *AutoMigrate* framework addresses these concerns through a modular design as described in Sect. 2. The focus of the remainder of this paper will be the underlying functionality of *CS Selector*.

The *CS Selector* component within *AutoMigrate* is responsible for selecting a set of cloud services for migration according to the memory budget. A live migration is triggered when the *Anomaly Detector* detects anomalies on the host machines. The type of the anomalies detected will place a limit on the time available for the migration process. The exact time-to-failure is unknown, but failure is possible that it will occur before all of the machines can be migrated. In fact, the migration of a single VM may take tens to hundreds of seconds on a network with reasonable bandwidth [17,21]. With a set of cloud services that uses a large number of VMs, the migration process may take a long time, which might not be feasible under time and bandwidth constraints. For instance, systems that depend on satellite communications have very low bandwidth compared to systems using land-based networks. Therefore, to address this concern, we propose a pre-defined time bound that a user will provide for the migration process. This time limit can be translated into a threshold on migratable memory, or memory budget, which is the total amount of memory that can be migrated within the time limit, using the known bandwidth of the environment.

##### 4.1 Terminology and assumptions

We consider a cloud infrastructure as a service (IaaS) model where cloud service providers manage VM instances and offer them to customers, who can then use the VMs to deploy cloud services. We assume that each service provider owns one or more hypervisors for generating and managing virtual machines on demand. We consider a set of cloud service providers, where each service provider uses hypervisors that are compatible in terms of migration in such a way that the VMs can be migrated between the different hypervisors. We assume that customers can acquire VMs from multiple service providers to deploy their services. The terminology used throughout the paper is summarized in Table 1.

**Table 1** Terminology

Variable	Description
$S$	Set of all cloud services
$S_i$	A particular cloud service
$S^*$	Set of selected cloud services
$S^+$	Set of cloud services ordered by BCR
$V$	Set of all VMs
$V_j$	A particular VM
$V^*$	Set of selected VMs
$V'$	Set of candidate VMs
$C(V_j)$	Cost of individual VM, defined by memory footprint
$P(S_i)$	Priority level of a single cloud service (1–10)
$P(S^*)$	Priority value of selected cloud services
$C(S_i)$	Cost of individual cloud service
$C(S^*)$	Cost of selected cloud services
$M$	Total memory budget
$BCR(S_i)$	Benefit cost ratio of a given cloud service
$F(V_j)$	Frequency of a particular VM
$C_F(S_i)$	Frequency adjusted cost of a cloud service

**Cloud services** We define a cloud service as a set of applications that make use of one or more VMs that are hosted in the cloud with a given cloud service provider. For instance, an accounting application may have the payroll managed in a database in one virtual machine, and the web server in another virtual machine. A Hadoop-based cloud service can use multiple virtual machines to distribute the Hadoop nodes. On the other hand, a VM may have multiple services running on it, which leads to a many-to-many relationship between cloud services and VMs.

**Definition 1** Dependent VMs: Let us denote  $S$ , a set of  $n$  cloud services,  $S = \{S_1, S_2, \dots, S_n\}$ , and  $V$  a set of  $m$  virtual machines,  $V = \{V_1, V_2, \dots, V_m\}$ . Let us denote  $V'$  a subset of  $V$ ,  $V' \subseteq V$ . Then, we say that  $S_i$  depends on  $V'$  if and only if all the VMs in  $V'$  are used to deploy  $S_i$ .

**Live migration of cloud services to preserve their availability** Live migration is the process of moving a virtual machine from one physical host to another without interrupting the processes running in memory on the virtual machine. Cloud service migration involves moving a cluster of virtual machines. A cloud service is fully migrated if and only if all of its dependent VMs are successfully migrated. A cloud service is only partially migrated if some, but not all, of its dependent VMs are successfully migrated, thereby leading to the cloud service not being fully available on the destination host. If a host in the source cloud fails, any services provided by VMs resident on that host that have not already been migrated will not be available.

### 4.2 VM selection as a set-union knapsack problem

The VM selection problem can be directly mapped to the set-union knapsack problem (SUKP) [22]. For a given cloud service provider, let  $S$  denote a set of cloud services  $S = \{S_1, S_2, \dots, S_n\}$  and  $V$  a set of virtual machines  $V = \{V_1, V_2, \dots, V_m\}$ . Each  $S_i$  is a subset of elements of  $V$ . Each cloud service  $S_i$  has a nonnegative priority given by  $P(S_i) : S \rightarrow \mathbb{Z}_+$  and each virtual machine  $V_j$  has a nonnegative cost given by  $C(V_j) : V \rightarrow \mathbb{Z}_+$ . The cost of each  $V_j$  is the total memory used by the virtual machine. The goal is to find a subset of the services  $S^* \subseteq S$  such that the priority of  $S^*$ ,  $P(S^*)$ , is maximized and the cost of  $S^*$ ,  $C(S^*)$ , is bounded by  $M$ . Achieving this goal leads to prioritized cloud service migration.

### 4.3 Maximizing cloud service availability

In order to maximize cloud service availability using the *Set-Union Knapsack* problem, we need to define a number of concepts that will be needed by the algorithm to solve this problem. These concepts are the building blocks used in constructing the algorithm.

#### 4.3.1 Benefit

The benefit is defined in terms of priorities of the different cloud services. A *Priority Level* is an integer value associated with each cloud service, which determines the importance of the service as defined by the user. The *Priority Level* is a user defined value and corresponds to the value added by the service in the overall user environment. The *Priority Value* is an integer value associated with a set of cloud services. It is calculated by summing the *Priority Level* of the individual services in the set. The *Priority Value* is formally defined as follows:

**Definition 2** Priority value: The priority value,  $P(S'')$ , of a set  $S'' \subseteq S$ , of cloud services is the total of all service priorities within the set, given by:

$$P(S'') = \sum_{S_i \in S''} P(S_i)$$

#### 4.3.2 Cost

We define the cost of migrating a single cloud service as the memory used by its dependent VMs, which is proportional to the migration time of the VMs. Other metrics can be used to determine the migration costs of VMs such as VM downtime, response time, and power consumption [17].

**Definition 3** Cloud service cost: The cost of a particular cloud service,  $S_i$ , is obtained by taking the sum of the costs for all dependent VMs. Formally:

$$C(S_i) = \sum_{V_j \in S_i} C(V_j)$$

**Definition 4** Cloud service set cost: The cost of a set of cloud services is defined by taking the sum of all the costs of all services in the set. Formally:

$$C(S'') = \sum_{S_i \in S''} C(S_i)$$

The frequency adjusted cost takes into account the cost of a VM that is shared by multiple cloud services.

**Definition 5** Frequency adjusted cost (FAC): The frequency adjusted cost of a cloud service is calculated as:

$$C_F(S_i) = \sum_{V_j \in S_i} \frac{C(V_j)}{F(V_j)}$$

The frequency,  $F(V_j)$ , of a VM is the number of cloud services making use of that VM.

#### 4.3.3 Benefit-cost ratio

The benefit-cost ratio of a service is used to decide which set of high priority cloud services to migrate.

**Definition 6** Benefit-cost ratio: The benefit-cost ratio,  $BCR(S_i)$ , is calculated as the ratio of the priority value of the cloud service to its cost. Formally:

$$BCR(S_i) = \frac{P(S_i)}{C(S_i)}$$

#### 4.3.4 Preserving service availability

The overall objective is to preserve the availability of cloud services with high priority through the use of live migration. This objective can be reduced to the problem of maximizing the priority of migrated cloud services under the constraint of a memory budget. The set of cloud services covers the set of VMs.

**Definition 7** Maximizing priority of migrated cloud services:

$$\begin{aligned} & \text{maximize } \sum_{S_i \in S} P(S_i) \cdot y_i, \\ & \text{subject to } \sum C(V_j) \cdot x_j \leq M \end{aligned}$$

$y_i \in \{0, 1\}$ , where  $y_i = 1$  indicates an available service.  
 $x_j \in \{0, 1\}$ , where  $x_j = 1$  indicates a migrated VM.

## 5 Service availability through optimization of cloud service selection

Our problem of selecting cloud services to migrate is directly mapped to the set-union knapsack problem. However, the set-union knapsack problem is an NP-hard problem. Due to the computational complexity associated with NP-hard problems, solving it efficiently requires a heuristic based approach. There are a number of approximation approaches which attempt to reduce the complexity of solving the problem [22,23], including approaches using a greedy algorithm. In this section, we describe our solution based on the greedy algorithm proposed in [22]. The greedy algorithm uses the benefit-to-cost ratio of all cloud services as a heuristic to minimize the amount of computation. We have implemented two versions of the greedy algorithm, a standard greedy algorithm (*StandardGreedy*), and an improved greedy algorithm (*ImprovedGreedy*). We also implemented a basic greedy algorithm (*BasicGreedy*) as a baseline for assessing the performance of the proposed greedy algorithms.

### 5.1 Greedy algorithms

Our implementation of the three greedy algorithms takes as input a set of services mapped onto a set of VMs and the memory footprint of each VM. *BasicGreedy* simply selects the services with the highest *priority* that fit within the memory budget. *StandardGreedy* sorts the set of services in descending order of their *BCR*, calculated as  $BCR = \frac{priority}{Cost}$ , and iterates over this set, selecting services (and their corresponding VMs) which fit into the memory budget. Once calculated, the *BCR* values are fixed throughout the iterations of the algorithm. *ImprovedGreedy* follows the *StandardGreedy* approach, except it computes the *BCR* using the *FAC* instead of the *Cost* ( $BCR = \frac{priority}{FAC}$ ). In addition, the *BCR* values are re-calculated in each iteration after removing the selected service for migration. In the following section, we describe the details of *ImprovedGreedy*.

**ImprovedGreedy algorithm:** This algorithm sorts the cloud services in descending order of their *BCR*, and iterates over this set, selecting services (and their corresponding VMs) which fit into the memory budget. After a service is selected in each iteration, the *BCR* is re-computed to account for shared VMs between services to avoid the duplication of cost of such VMs.

An outline of *ImprovedGreedy* is as follows:

- Copy the set of services into the set of remaining services
- While the set of remaining services is not empty and there is still a service that can be selected within the budget
  - For each remaining service determine the benefit cost ratio (*BCR*)
  - Sort the set of services in descending order of *BCR*

- Iterate through the sorted set
  - Compute the cost of the current service
  - If the running cost of the selected services plus the cost of the current service is less than or equal to the budget, then:
    - Add the cost of the current service to the running cost of the selected services
    - Add the current service to the set of selected services
    - Remove the current service from the remaining set of services
    - Add the VMs of the current service to the list of selected VMs
    - Set the cost of the VMs of the selected service to zero in the set of remaining services
    - Re-compute the *BCR* on the remaining services
    - Sort the set of remaining services in descending order of *BCR*
- Compare the total priority of the set of selected services against that of the individual service with the largest priority
- Return whichever of these has the greatest priority.

The *ImprovedGreedy* algorithm keeps track of the selected cloud services ( $S^*$ ) as well as the list of selected VMs ( $V^*$ ). The pseudocode for the algorithm appears below:

```

ImprovedGreedy ( $S, V, M$ ){
   $S_{remain} = S$  /*Set of remaining services*/
   $S^* = \emptyset$  /*Selected services*/
   $V^* = \emptyset$  /*Selected VMs*/
   $C_{previous} = -1$  /*Previous costs*/
   $C_{current} = 0$  /*Current costs*/
  while ( $S_{remain} \neq \emptyset$  &  $C_{current} \neq C_{previous}$ )
     $C_{previous} = C_{current}$ 
    calculateBCR( $S_{remain}$ )
     $S^+ = \text{sortServicesByBCR}(S_{remain})$ 
    size = getNumberOfServices( $S^+$ )
    for ( $i=1$  to size) ( $S_i^+ \in S^+$ )
       $C(S_i^+) = \text{CalculateCost}(S_i^+)$ 
      if ( $C_{current} + C(S_i^+) \leq M$ )
         $C_{current} = C_{current} + C(S_i^+)$ 
         $S^* = S^* \cup \{S_i^+\}$ 
         $V' = \text{GetListVMs}(S_i^+)$ 
         $S_{remain} = S^+ \setminus \{S_i^+\}$ 
        insertSelectedVMs( $V^*, V', S_{remain}$ )
      BREAK
    end if
  end for
end while
  Return  $S^*$ 
}

```



The subroutine  $calculateBCR(S_{remain})$  computes the  $BCR$  for each service  $S_i$  in  $S_{remain}$ . The subroutine  $sortServicesByBCR(S_{remain})$  sorts the set of the remaining cloud services  $S_{remain}$  in descending order based on the  $BCR$  of each cloud service.

**Calculating Cost** The cost associated with each cloud service  $S_i$  is calculated as follows:

```

CalculateCost ( $S_i$ ){
   $C_{total}(S_i) = 0$  /*Initializes the cost of  $S_i$ */
   $V' = getListVMs(S_i)$  /*Get the list of VMs in  $S_i$ */
  for each ( $V'_j \in V'$ )
     $C_{total}(S_i) = C_{total}(S_i) + C(V'_j)$ 
  end for each
  Return  $C_{total}(S_i)$ 
}
    
```

The subroutine  $getListVMs(S_i)$  gets the list of VMs associated with the cloud service  $S_i$ .

**Adding Candidate VMs to the List of Selected VMs** The following subroutine inserts the candidate VMs from the newly selected cloud service into the list of selected VMs. In addition, it sets the cost of each VM from the selected  $S_i$  to zero ( $C(V_j) = 0$ ) in all remaining services. This prevents duplication of the cost of a VM shared by multiple services.

```

InsertSelectedVMs ( $V^*, V', S_{remain}$ ){
  for each ( $V'_j \in V'$ )
     $V^* = V^* \cup V'_j$  /*Add VM to selected VMs*/
    for each ( $S_i \in S_{remain}$ )
       $V = getListVMs(S_i)$  /*Get list of VMs*/
      for each ( $V_j \in V$ )
        if ( $V_j = V'_j$ )
           $C(V_j) = 0$  /*Set cost VM to zero*/
        end if
      end for each
    end for each
  end for each
  Return  $V^*$ 
}
    
```

### 5.2 Illustrative example

In order to demonstrate the differences between the algorithms described above, we show a simple example. In this example, there are five services utilizing four virtual machines, constrained with a budget  $M = 7 GB$ . The parameters used can be seen in Tables 2 and 3. Table 3 shows the mapping of services to VMs, where each VM used by a service is indicated by a non-zero cost in the table. For example,

**Table 2** List of VMs

VM	Cost	Frequency	Frequency adjusted cost
$V_1$	2	2	1.00
$V_2$	5	1	5.00
$V_3$	7	1	7.00
$V_4$	4	2	2.00

$S_4$  uses  $V_1$  and  $V_4$ , which both have non-zero costs under the  $S_4$  column. The *BasicGreedy* algorithm uses only the priority when determining which VMs to migrate.  $S_1$  is the highest priority service, with priority 10, so it is selected first.  $S_1$  has a cost of 7, which equals the budget. Since no other service can be selected while still keeping the total cost within the budget,  $S_1$  is the only service selected for migration. The total priority preserved by this algorithm is 10. The *StandardGreedy* algorithm sorts the services in order of  $BCR$ , using the actual memory usages of the VMs comprising each service as the total cost for that service. The services with highest  $BCR$  are  $S_2$  and  $S_3$ , with a  $BCR$  of 4.00 and 1.20, respectively. The total cost of  $S_2$  and  $S_3$  is 7.0 GB. Since adding any additional services would lead to a total cost exceeding the budget, no other services can be selected for migration. The total priority preserved by this algorithm is 14. The *ImprovedGreedy* algorithm also sorts the services in order of  $BCR$ , but uses the  $FAC$  in place of the actual memory used when calculating the cost of each service. The services with highest  $BCR$  are  $S_2$ ,  $S_4$ , and  $S_5$ , with a  $BCR$  of 8.00, 2.33, and 2.00, respectively. The total cost of  $S_2$ ,  $S_4$ , and  $S_5$  is 7.0 GB. Since adding any additional services would lead to a total cost exceeding the budget, no other services can be selected for migration. The total priority preserved by this algorithm is 19, which is optimal. The  $FACs$  for each of the *StandardGreedy* and the *ImprovedGreedy* algorithms are calculated without and with the VM frequencies, respectively. The reason that the *ImprovedGreedy* algorithm has better performance than the *StandardGreedy* algorithm is that it incorporates frequencies which eliminate the duplicate costs of VMs shared by multiple services. Also, during each iteration of the *ImprovedGreedy* algorithm, the remaining services are re-ordered based on the services already selected for migration.

## 6 Implementation and experiments

In this section, we summarize the implementation of the *AutoMigrate* framework and describe the experiments we performed to analyze its performance.

### 6.1 Implementation

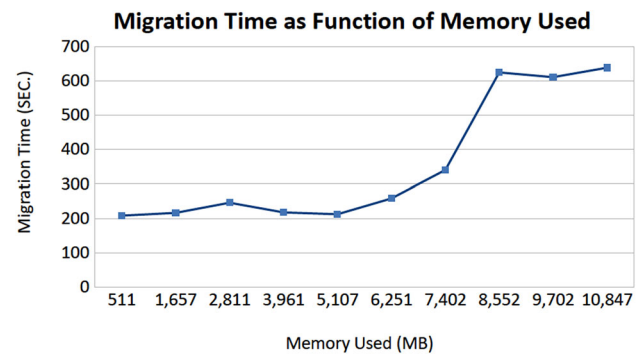
We implemented the *AutoMigrate* framework, depicted in the architecture diagram in Fig. 1, using a modular design pat-

**Table 3** Comparison of the three algorithms

Cloud 1	Services	$S_1$	$S_2$	$S_3$	$S_4$	$S_5$
VMs	Priority	10	8	6	7	4
$V_1$	Cost	0.00	2.00	0.00	2.00	0.00
	FAC	0.00	1.00	0.00	1.00	0.00
$V_2$	Cost	0.00	0.00	5.00	0.00	0.00
	FAC	0.00	0.00	5.00	0.00	0.00
$V_3$	Cost	7.00	0.00	0.00	0.00	0.00
	FAC	7.00	0.00	0.00	0.00	0.00
$V_4$	Cost	0.00	0.00	0.00	4.00	4.00
	FAC	0.00	0.00	0.00	2.00	2.00
Basic	Cost	7.00	2.00	5.00	6.00	4.00
	Order	10	8	6	7	4
	Selection	✓				
Standard	Cost	7.00	2.00	5.00	4.00	4.00
	BCR	1.43	4.00	1.20	1.17	1.00
	Selection		✓	✓		
Improved	FAC	7.00	1.00	5.00	3.00	2.00
	BCR	1.43	8.00	1.20	2.33	2.00
	Selection		✓		✓	✓

Costs and FACs are measured in GB of memory

tern. With this design pattern, each of the components can be replaced with a different implementation without changing the overall framework. We use Xen [24] as the underlying hypervisor for the VM management in the cloud. When migrating a VM, Xen uses a pre-copy live migration algorithm [2], which copies all memory pages from the source host to the destination host. Then, the VM is suspended on the source host and resumed on the destination host. With reasonable bandwidth, this algorithm achieves live migration with minimal downtime. We use the Thrift [25] framework to abstract away the client/server communication between the *VM Monitor*, the *Migration Trigger*, and the *VM Migrator* components. For ease of administration, the web-based dashboard provides a GUI through which the end user can configure and control the system. The dashboard is connected to the client-side *VM Monitor*. We used the Thrift framework to connect the *VM Monitor* and the *VM Migrator* to the Xen hypervisor, which enables the creation/instantiation/deletion of VMs, the collection of metrics from VMs, and the initiation of the live migration process. For our initial approach to anomaly detection, we implemented a simplistic Naive Bayes machine learning algorithm using the Python Scikit-learn library [26]. The algorithm takes as input VM historical state data (CPU, memory, disk, and network utilisation), and tries to detect anomalies with the VMs. Note that this algorithm is just for building a proof of concept for the framework. Other advanced techniques for detecting anomalies in systems and networks can be used, including intrusion detection and pre-

**Fig. 2** Live migration performance

vention systems (IDPS), to enhance the anomaly detection process.

## 6.2 Experiments

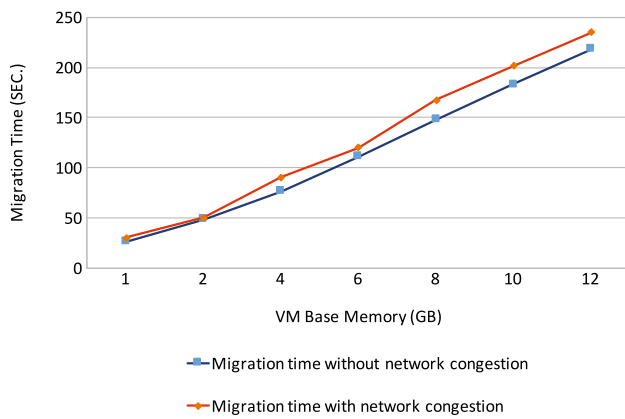
We performed a number of experiments to analyze the performance of our proposed VM selection algorithms. In each experiment, we compared our proposed algorithms, *StandardGreedy* and *ImprovedGreedy*, to the priority-only based VM selection algorithm, *BasicGreedy*. Note that we have not found any other approaches, which attempt to solve this problem of VM selection under time and bandwidth constraints, to compare against.

Before going into the details of our experiments for the algorithms, let us first look at the complexity of migrating virtual machines between hosts.

### 6.2.1 Memory size and network bandwidth impact on migration

We experimentally analyzed how the network bandwidth and memory used by virtual machines can impact the performance of live migration. In the first set of experiments, we analyzed the live migration performance of the Xen hypervisor using two laptops with 16 GB total memory running a Ubuntu 12.04 OS. Figure 2 shows the time it takes to migrate a VM as a function of the memory used by the VM. The memory used ranges from 2 GB to 12 GB. The diagram illustrates that the migration time increases exponentially as the memory used increases. Migration of a 2 GB VM takes more than 200 s on average over a connection with 1 Gbps bandwidth.

In the second set of experiments, we analyzed how the performance of live migration is impacted by network congestion, which reduces the bandwidth. Figure 3 shows the results of the experiment, which compared the time it takes to migrate a VM with and without network congestion. As can be observed, the network congestion increases the migration time. This result is disappointing, as network congestion can



**Fig. 3** Live migration performance as a function of VM base memory with respect to network congestion.

be difficult to predict, especially if many users are sharing the same cloud service resources as typically happens in a multi-tenant cloud environment. These two sets of experiments highlight the need for intelligent live migration management systems, systems which take into account memory used and network bandwidth available when deciding what, when, and where to migrate VMs.

### 6.2.2 Experimental setup for the algorithms

We deployed a Xen based cloud infrastructure and integrated our *AutoMigrate* framework into it. We performed simple experiments to analyze the feasibility of the framework and the proposed algorithms. Additionally, in order to scale up the size of the experiments, we auto-generated the input data, which enables us to perform a better analysis of our proposed approach. This data set includes service priority, VM memory footprint, and the relationships between VMs and services.

The *priority level* of a cloud service is represented as an integer from 1 to 10, where 1 is the lowest priority and 10 is the highest priority. Each VM is assigned a random *memory footprint* between 2 and 12 GB. In different runs, we vary the total number of VMs and total number of cloud services on each source host from 100 to 500 to 1000. All of the experimental results are an average over 100 iterations.

The metrics used to determine the performance of the algorithms are the utility ( $\mathcal{U}$ ) and availability ( $\mathcal{A}$ ) of the cloud services. Utility is defined as  $\mathcal{U} = \frac{P(S')}{P(S)}$ , where  $P(S')$  is the sum of all the priorities of the completely migrated services and  $P(S)$  is the sum of the priorities of all of the services in the experiment. Availability is defined as  $\mathcal{A} = \frac{|S'|}{|S|}$ , where  $S'$  is the number of completely migrated cloud services and  $S$  is the total number of cloud services in the experiment.

**Table 4** Experiment 1—utility ( $\mathcal{U}$ ) and availability ( $\mathcal{A}$ ) percentages when varying number of VMs and memory budget

VMs	M	Basic		Standard		Improved	
		$\mathcal{U}(\%)$	$\mathcal{A}(\%)$	$\mathcal{U}(\%)$	$\mathcal{A}(\%)$	$\mathcal{U}(\%)$	$\mathcal{A}(\%)$
100	200	2.97	1.64	8.88	6.25	10.65	7.33
	400	5.43	3.00	14.03	10.25	16.51	12.11
	600	7.63	4.21	17.81	12.91	20.00	16.16
	800	10.07	5.56	21.10	15.35	21.76	19.93
	1000	12.98	7.14	25.47	18.79	21.57	20.20
500	200	2.38	1.31	5.46	3.51	14.24	9.82
	400	4.51	2.50	9.12	5.92	26.10	18.48
	600	6.56	3.60	12.57	8.22	35.42	25.57
	800	8.67	4.79	15.60	10.35	43.36	31.45
	1000	10.58	5.82	18.20	12.17	50.00	36.57
1000	200	2.00	1.12	3.69	2.29	6.06	3.91
	400	3.82	2.11	6.69	4.15	12.32	7.93
	600	5.66	3.12	9.30	5.83	17.73	11.54
	800	7.57	4.18	11.95	7.53	22.57	14.77
	1000	9.19	5.06	14.05	8.92	26.95	17.73

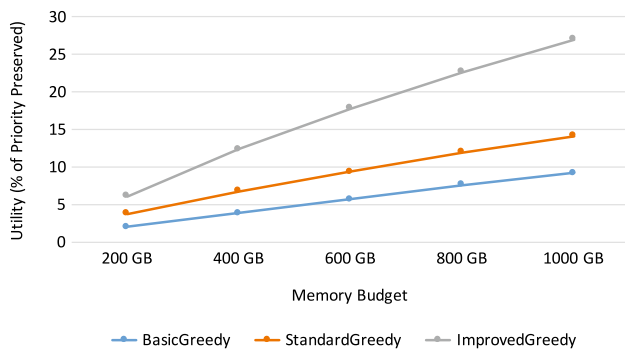
### 6.2.3 Experiment 1: algorithm comparison under varying memory budget and varying number of VMs

In this experiment, we compare the three algorithms, *BasicGreedy*, *StandardGreedy*, and *ImprovedGreedy*, by varying the memory budget over {200, 400, 600, 800, 1000} GB and varying the number of VMs over {100, 500, 1000}. We fixed the number of cloud services to 500 and the maximum number of VMs per service to 10. For this experiment, we expected to see improvements in the utility as a result of our proposed two algorithms.

Table 4 shows the results of all three algorithms when varying the memory budget from 200 to 1000 GB and varying the number of VMs from 100 to 500 to 1000. We show only the graph for 1000 VMs, which can be seen in Fig. 4. Note that all of the experiments exhibit similar performance characteristics. As can be seen, the two proposed algorithms perform significantly better than the basic greedy selection algorithm. From Table 4, one can see that increasing the number of VMs decreases the utility of the migrated services for *BasicGreedy* and *StandardGreedy*. However, *ImprovedGreedy* performs best when the number of VMs is equal to the number of cloud services.

### 6.2.4 Experiment 2: algorithm comparison under varying memory budget and varying number of cloud services

In this experiment, we compare the three algorithms, *BasicGreedy*, *StandardGreedy*, *ImprovedGreedy*, by varying the budget over {200, 400, 600, 800, 1000} GB and varying the



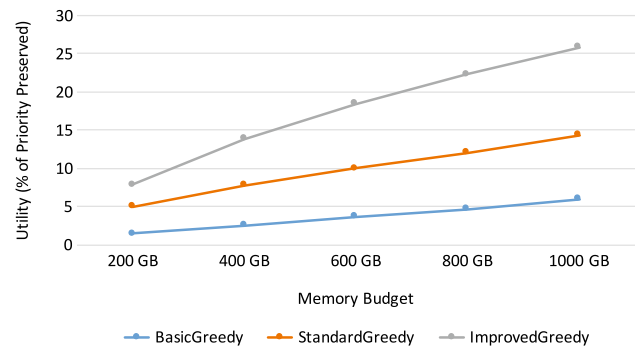
**Fig. 4** Experiment 1 Results: utility as a function of memory budget when the number of VMs is fixed to 1000 and the number of CSs is fixed to 500.

**Table 5** Experiment 2—utility ( $U$ ) and availability ( $A$ ) percentages when varying number of cloud services (CSs) and memory budget

CSs	M	Basic		Standard		Improved	
		$U(\%)$	$A(\%)$	$U(\%)$	$A(\%)$	$U(\%)$	$A(\%)$
100	200	6.15	3.40	7.82	4.66	9.33	5.68
	400	12.54	6.99	14.64	8.86	18.47	11.36
	600	18.95	10.61	21.25	12.97	27.33	16.90
	800	24.58	13.99	27.01	16.73	35.19	22.16
	1000	30.72	17.76	33.10	20.67	42.85	27.25
500	200	2.27	1.26	5.23	3.37	14.07	9.70
	400	4.48	2.48	9.21	5.98	26.39	18.67
	600	6.47	3.56	12.53	8.27	35.29	25.45
	800	8.78	4.82	15.70	10.41	43.06	31.35
	1000	10.63	5.87	18.50	12.50	49.76	36.60
1000	200	1.45	0.80	4.98	3.41	7.87	5.10
	400	2.55	1.41	7.79	5.46	13.88	9.54
	600	3.63	2.01	10.00	7.03	18.46	12.96
	800	4.67	2.58	12.02	8.44	22.36	16.03
	1000	6.00	3.31	14.35	10.12	25.82	18.59

number of cloud services over  $\{100, 500, 1000\}$ . We fixed the number of VMs to 500 and the maximum number of VMs per service to 10. As the number of services increases, the expectation is that the utility will improve.

Table 5 shows the results of all three algorithms when varying the memory budget from 200 to 1000 GB and varying the number of services from 100 to 500 to 1000. The results of this experiment show similar performance characteristics to the results of the previous experiment. Table 5 shows that as the number of services is increased from 100 to 1000, the total utility decreases for the *BasicGreedy* and *StandardGreedy* algorithms. As expected, the utility of the algorithms decreases as the number of services is increased. However, *ImprovedGreedy* performs best when the number of VMs is equal to the number of cloud services. The results of this experiment can be seen in Fig. 5 when fixing the num-



**Fig. 5** Experiment 2 Results: utility as a function of memory budget when the number of VMs is fixed to 500 and the number of CSs is fixed to 1000.

**Table 6** Experiment 3—utility ( $U$ ) and availability ( $A$ ) percentages when varying the maximum number of VMs assigned to a cloud service

Max VMs	Basic		Standard		Improved	
	$U(\%)$	$A(\%)$	$U(\%)$	$A(\%)$	$U(\%)$	$A(\%)$
1	23.16	13.15	32.79	23.67	32.79	23.67
2	11.43	6.37	20.11	8.74	31.71	23.00
3	10.79	5.99	14.71	9.41	31.43	22.92
4	9.70	5.44	13.62	8.84	29.30	22.00
5	9.36	5.29	13.43	8.82	28.28	21.26
6	8.68	4.87	13.07	8.63	27.69	20.63
7	7.48	4.23	11.79	7.77	26.21	19.66
8	7.08	4.03	11.11	7.38	24.49	18.71
9	6.72	3.81	11.18	7.56	22.71	17.37
10	6.46	3.70	10.97	7.54	21.53	16.39

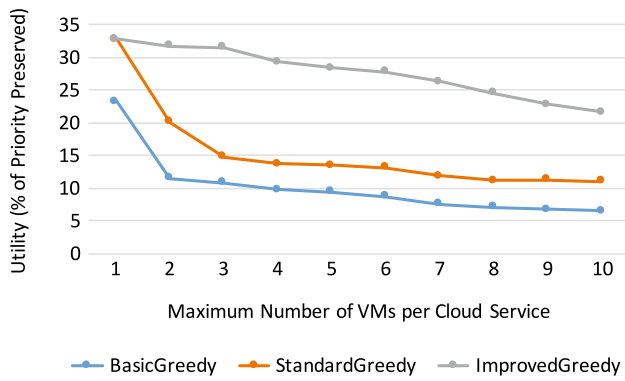
ber of services to 1000. Notice also that as the memory budget is increased, the utility of the services increases.

### 6.2.5 Experiment 3: analyzing the impact of VM distribution across services

In this experiment, we vary the maximum number of VMs used to deploy a service, and we compare the utility of the three algorithms. The number of VMs assigned to a service is randomly selected from a range of 1 to the assigned maximum. This maximum value is varied from 1 to 10 in this experiment. Both the number of VMs and the number of services are fixed to 100 and the memory budget is fixed to 1000 GB. We note that each service is assigned to at least one VM, and each VM supports at least one cloud service. The expectation is that the algorithms will improve over the basic greedy selection as services are distributed across an increasing number of VMs.

Table 6 highlights the results of all three algorithms when varying the maximum number of VMs per service from 1 to 10. These results can also be seen in Fig. 6. As the





**Fig. 6** Experiment 3 Results: utility as a function of the maximum number of VMs per CS when fixing both the number of VMs and the number of CSs to 100.

number of VMs assigned to a service is increased, the performance of the two proposed algorithms is better than the basic greedy selection. Note that the utility of each of the proposed algorithms is significantly larger than that of the basic greedy selection. As the the number of VMs assigned to a service is increased, the utility decreases for all three algorithms when the number of VMs and cloud services are fixed. The rationale for this behavior is that as services are more widely distributed across VMs, the larger the potential cost for migrating a complete service. This will result in a decreased ability to migrate complete services, thereby driving down the total priority value of selected services.

#### 6.2.6 Experiment 4: comparison of algorithm running time

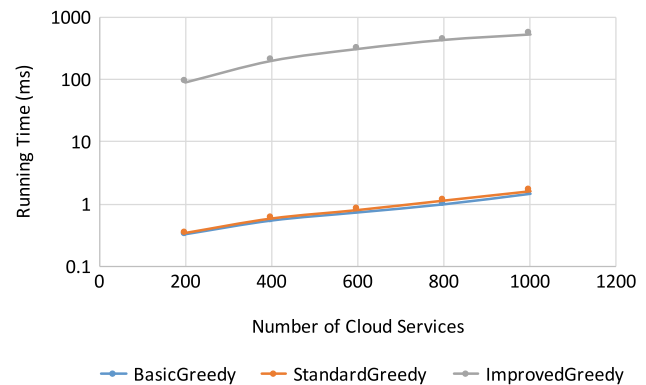
In this experiment, we analyze the performance of the three algorithms in terms of their running times. We calculate the average running time of each algorithm over 100 runs.

Table 7 shows the results of this experiment when varying the number of cloud services from 200 to 1000 and varying the number of VMs from 100 to 1000 while keeping the memory budget fixed at 1000 GB. In this table, all running times are expressed in milliseconds. From the table, one can see that the running time of the *ImprovedGreedy* algorithm is higher than that of the other two algorithms. This is the performance tradeoff for the higher utility and availability generated by the *ImprovedGreedy* algorithm. Nonetheless, the *ImprovedGreedy* algorithm takes only about 1 s to run in the worst case when using the largest input parameters in the experiment. The *StandardGreedy* algorithm has significantly better utility than the *BasicGreedy* algorithm, yet its running time is comparable to that of *BasicGreedy*. Figure 7 shows the running times of the three algorithms on a log-scale graph when varying the number of cloud services from 200 to 1000, and fixing the number of VMs to 1000 and the memory budget to 1000 GB. From this figure, it is clear that the running time of the *StandardGreedy* algorithm is almost

**Table 7** Experiment 4—running time of the algorithms when varying the number of cloud services (CSs) and VMs, and fixing the memory budget to 1000 GB

VMs	CSs	Basic	Standard	Improved
100	200	0.326	0.343	89.772
	400	0.545	0.587	200.237
	600	0.738	0.804	311.410
	800	1.001	1.142	436.117
	1000	1.483	1.615	533.685
500	200	0.389	0.402	101.464
	400	0.776	0.800	334.696
	600	0.922	0.979	593.332
	800	1.127	1.188	844.289
	1000	1.335	1.430	1063.041
1000	200	0.538	0.617	94.468
	400	0.799	0.826	271.558
	600	1.022	1.050	590.802
	800	1.234	1.264	987.971
	1000	1.448	1.552	1408.885

The running time is expressed in milliseconds



**Fig. 7** Experiment 4 Results: running time of the algorithms as a function of the number of cloud services when fixing the number of VMs to 500, and the memory budget to 1000 GB. Note that the vertical axis is log-scale.

identical to the running time of the *BasicGreedy* algorithm while the utility of *StandardGreedy* is higher than that of *BasicGreedy*.

## 7 Related work

Ahmad et al. [21] conducted a survey on VM migration and open research questions. While the majority of their survey covers migration of single VMs, the live migration of multiple VMs is only briefly considered. The bulk of the migration schemes surveyed do not assign dedicated resources for migration because of the increased computa-

tional cost to the cloud data center. VM migration granularity determines whether single or multiple VMs are migrated. This affects degradation duration, service downtime, and co-hosted applications quality of service. Moreover, VMs are migrated in a predefined order and the total migration time for each VM is increased due to queuing delay at the sender side. In this solution, only some information is required for each managed entity with respect to the desired system management goals. This approach differs from ours in that we only consider one instance of AutoMigrate controlling and managing a user cloud environment. Therefore, conflicts cannot occur.

Vedhanayagam et al. [27], developed metrics to study queuing systems and handle VM service disruptions. This model recommends standing up a second VM to redeem the original VM when service interruption is likely. However, this model accounts for single VM resilience while cloud services often require the support of multiple correlated VMs. Moreover, the replacement VMs are on the same data center servers as the original VM, so this model cannot be applied to situations where cloud services must be maintained while migrating between physically distant data centers.

## 7.1 When to migrate

Cloud data centers host increasingly large and diverse workloads, which poses extraordinary challenges to their hardware infrastructure and software stacks. Consequently, disruptions to, and degradation of service are frequent occurrences. Efforts to combat the degradation and disruption of cloud-based services fall under anomaly detection methods, which trigger VM migrations. Anomaly detection is either log-based or system metrics-based [28].

VM migration may be triggered for reasons other than anomalies. Mhedheb and Streit [29] describe VM scheduling for energy efficiency in data centers. They developed a thermal aware scheduler (ThaS), which inspects the change of load and temperature on host machines, triggering VM migration to avoid critical overheating or overloading scenarios.

### 7.1.1 Log-based anomaly detection

Tools for log-based anomaly detection extract information from logs and use machine learning techniques to build models that automatically detect system anomalies [30,31]. Lou et al. [30] developed machine learning-based log analysis tools which processes log message parameters to discover invariants. This detection process matches the invariants from new input logs with already learned invariants. Mismatches among invariants constitute anomalies. Xu et al. [31], designed and implemented a method of mining console logs to automatically detect system errors and anomalies.

Feature vectors were created from the logs and a principal component analysis (PCA) algorithm was applied, detecting anomalies. These techniques are characterized by the use of log parsers which mine console logs, creating models for detecting anomalies. These parsers require application source code, which may not always be available, for recovering log syntax. Moreover, log-based anomaly detectors typically do not give meaningful explanations of detected problems [28].

### 7.1.2 System metrics-based anomaly detection

System metrics-based anomaly detection methods account for the elasticity of cloud environments [28]. While implementing them in a larger cloud environment is extremely complex, the tools must be configured for multiple layers of data monitoring and analysis, system metrics-based anomaly detection tools do provide more effective and accurate detection. Because they use application-level in anomaly detection, system metrics-based detection tools generate immense volumes of data in large-scale cloud environments, which poses difficulties for effective processing.

Wang [32] created an online tool for detecting anomalies that utilized a distribution of metrics, entropy time series construction and processing across multiple layers of monitoring. Specifically, Wang uses an entropy-based anomaly detection scheme measuring and analyzing the concentration and dispersal of distributions across the cloud stack to form entropy time-series. Then a collection of tools, including spike detection, signal processing, and subspace methods to detect anomalies in the entropy time-series. Kang et al. [33], use a canonical correlation analysis (CCA) technique to extract the correlations between multiple application instances, where attributes of the instances are system resource metrics—e.g., CPU utilization, memory utilization, network traffic—and raises an alarm to announce anomalies when some correlations drop significantly. This method is capable of detecting application-level or VM-level anomalies because, but it requires large-scale statistical analysis and knowledge of hosted applications. Barbhuiya, et al. [28], develop a lightweight anomaly detection tool (LADT) which monitors system-level and virtual machine (VM)-level metrics in Cloud data centers to detect node-level anomalies using simple metrics and correlation analysis. LADT works on the hypothesis that, in an anomaly-free Cloud data center, there is a strong correlation between the node level and VM-level performance metrics and that this correlation diminishes significantly in the case of abnormal behaviour at the node-level. The LADT algorithm raises an anomaly alarm when the correlation coefficient value between the node-level and VM-level metrics drops below a threshold level.

## 7.2 What to migrate

Ye et al. [34], conducted early work on multiple machine live migration. They considered the impact of resource reservation on migration efficiency as well as analyzing parallel and workload-aware migration strategies. They determined that several optimization techniques could improve migration efficiency. Adjusting memory and CPU resources in the VM, or migrating VMs with smaller memory first, were suggested optimization methods. It was also determined that VM migration decisions should be made accounting for workload characteristics of the target machine. However, this work does not consider the case where migration is constrained by time and bandwidth considerations. Deshpande et al. [35] propose a live gang migration scheme in LAN environments. Their scheme de-duplicates page- and sub-page-level among co-located VMs and provides differential compression of nearly identical content. They refer to the simultaneous migration of multiple active VMs from one physical host to another as gang migration. Improvements in migration efficiency were made by offline and sub-page deduplication, re-hashing, and transferring only the differences between nearly identical pages. Their work did not consider WAN environments, nor did they prioritize applications and services.

Sun et al. [36], propose an improved serial migration strategy with a post-copy scheme integrated. They then propose a  $m$  mixed migration strategy based on the improved serial migration strategy and complemented by a parallel migration strategy with the goal of satisfying the maximum downtime constraint and minimizing the total migration time. First,  $m$  VMs are parallel migrated by a pre-copy migration strategy. When the  $m$  VMs stop running, the rest of the VMs are stopped and serially migrated by a post-copy strategy. Song et al. [37], also proposed PMigrate, which leverages data parallelism and pipeline parallelism to parallelize the live migration operation. These approaches focus only on accelerating the live migration process itself, not on filtering VMs for migration based on priorities.

Liu and He [38] detail VMBuddies, a system that attempts to optimize the live migration of correlated VMs that make up multi-tiered web applications. Multi-tiered web architecture is typical of internet applications, with each tier providing a specific functionality. Most multi-tier applications consist of three layers: a presentation layer (web tier), business logic layer (app tier), and a data access layer (DB tier). Each layer will be run on different VMs, each having different memory access patterns. The VMs are called correlated because they must all be migrated to another server in order to completely and efficiently serve requests in that data center. Correlated VM migrations can cause considerable performance degradation for multi-tiered applications because, if the VMs are not migrated together, communication and data access traf-

fic must be routed to the new servers. VMBuddies addresses the challenges of correlated VM migration with a synchronization protocol to assure that correlated VMs complete their migration simultaneously, thus avoiding data exchange across data centers. The scenarios where multiple VMs have the same function as well as where applications have difficult topologies and multiple VMs have different functions. While this approach does show significant improvement in response times, the authors do not prioritize multi-tiered applications for migration management.

## 7.3 Where to migrate

Berthier et al. [39] give an in-depth discussion of AMS coordination problems and provide a new design methodology that addresses them. Autonomic managers react to one or more aspects of the management task, and multiple managers make up an AMS. Different managers within the AMS may issue contradictory instructions. Coordination of managers to avoid contradiction is an example of an entity-level coordination problem. Managers must be coordinated to avoid contradictory instructions. Managers within a system may require “glue code” or special rules to operate, both of which are slow and error prone. Global consistency issues may arise, where managers issuing instructions for one set of events cause ramifications for other managers (for instance, due to workload dependencies). A solution is presented which allows the AMS to implement, maintain, and interpret a registry of “global knowledge” about the managed system.

Deshpande and Keahey [40] recently proposed a traffic-sensitive approach to VM migration that mitigates network contention between VM migration and VM application traffic. When host and destination servers are equipped with 1 gigabit Network Interface Cards, 1Gbps per second bandwidth is available in each direction. There is no conflict when traffic flows in opposite directions, but when migration and application traffic flow in the same direction both must contend for bandwidth. In our work, we analyzed the impact of network congestion on the time taken to perform live migration. Rybina et al. [41], have investigated the influence of workload as well as interference effects on multiple VMs, showing that migration time is proportional to the volume of memory copied between the source and the destination machines. Experimentation was conducted on underutilized servers as well as heterogeneous servers, and interference effects were caused by co-locating VMs on the same server. Experimental results suggested that top candidates for live migration are VMs running CPU intensive tasks, rather than VMs running memory intensive tasks. Using our approach, a high priority level can be assigned to services running intensive CPU tasks, which increases their likelihood of being selected for migration.

Mhedheb and Steit's ThaS [29] decides during runtime which VMs will be allocated to which hosts, targeting VM migration towards hosts with more favorable load and temperature statistics. Dong and Herbert [42] likewise propose a forecast-based VM placement algorithm which reduces power consumption and prevents Service Level Agreement violation. This model utilized cloud-based R servers which built forecast models for each VM and made predictions about their future CPU resource requirements, which then informed VM placement decisions. Traffic-Aware VM grouping partitions VM into sets of VM groups in order to minimize the overall inter-group traffic volume while the overall intra-group traffic is maximized. Fang et al. [43], implemented a cloud migration strategy that placed VMs in traffic aware groups, optimally assigned VM groups to server racks to the total inter-traffic loads in the network, as well as move and aggregate network traffic onto fewer paths for data center energy efficiency.

## 8 Conclusion

In this paper, we presented a novel framework for maximizing cloud service availability in response to the threat of cloud service disruption. The framework preserves cloud service availability through live migration of virtual machines, thereby leading to enhanced cloud resiliency. This framework identifies what cloud services should be migrated, when these cloud services need to be migrated, and where these services should be migrated to. Systems developed using this framework are able to autonomously manage the migration of VMs so as to minimize cloud disruption due to anomalous events. As part of the framework, we developed two greedy algorithms for selecting cloud services to migrate in order to maximize the availability of high priority cloud services under time and bandwidth constraints. We deployed a cloud and integrated the framework with the two algorithms. We performed experiments to analyze the performance of the two algorithms. The experimental results show that the two algorithms significantly outperform a basic greedy selection algorithm. These algorithms could be used to decide how to distribute a cloud service workload across the VMs in a data center.

One avenue for future work will be investigating alternative cyber threat detection technologies which can be used to strengthen the anomaly detector component within the framework. Such cyber threat detection technologies include intrusion detection systems, intrusion prevention systems, and cyber indicators of compromise. Automatic schemes for assigning service priorities and choosing budgets, which can be learned from the system instead of being chosen manually as is done in this AutoMigrate framework, is a natural extension of this work. Another direction of future work will

be integrating the AutoMigrate framework with commercial cloud service providers. In our previous work on data security and privacy in the cloud, we proposed the Nomad framework [44,45]. An interesting research avenue will be to integrate the Nomad and AutoMigrate frameworks, which will enable both data privacy controls and self-managing cloud services, to ensure privacy and availability in the cloud. Docker based migration of cloud services has the potential for further improving the availability of cloud services. Reducing the migration workload by shutting down low priority applications running on VMs, can be added to the AutoMigrate framework, to further improve the availability of services with higher priority.

**Acknowledgements** We would like to thank Luis Angel Bathen for his contributions in starting this project. We also would like to thank Vic Beach, Tonya R. Nishio, and Ronald A. Wolfe for their managerial support.

## References

1. Shrivastava, V., Zerfos, P., Lee, K., Jamjoom, H., Liu, Y., Banerjee, S.: Application-aware virtual machine migration in data centers. In: INFOCOM 2011. 30th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 10–15 April 2011, Shanghai, China, pp. 66–70 (2011)
2. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines. In: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation (NSDI'05), Vol. 2, pp. 273–286. USENIX Association, Berkeley, CA, USA (2005)
3. Huang, T., Zhu, Y., Wu, Y., Bressan, S., Dobbie, G.: Anomaly detection and identification scheme for VM live migration in cloud infrastructure. *Futur. Gener. Comput. Syst.* **56**, 736–745 (2016)
4. Nagafuchi, Y., Teramoto, Y., Hu, B., Kishi, T., Koyama, T., Kitazume, H.: Routing optimization for live VM migration between datacenters. In: 2015 10th Asia-Pacific Symposium on Information and Telecommunication Technologies (APSITT), pp. 1–3. IEEE (2015)
5. Shetty, S., Yuchi, X., Song, M.: Towards a network-aware VM migration: Evaluating the cost of vm migration in cloud data centers. In: Moving Target Defense for Distributed Systems. Springer, Berlin (2016)
6. Li, X., He, Q., Chen, J., Ye, K., Yin, T.: Informed live migration strategies of virtual machines for cluster load balancing. *IEEE Trans. Softw. Eng.* **PP**(99), 111–122 (2011)
7. Lu, P., Barbalace, A., Palmieri, R., Ravindran, B.: Adaptive live migration to improve load balancing in virtual machine environment. In: Euro-Par 2013: Parallel Processing Workshops 2013, Aachen, Germany, August 26–27, 2013. Revised Selected Papers, pp. 116–125 (2013)
8. Wood, T., Shenoy, P.J., Venkataramani, A., Yousif, M.S.: Black-box and gray-box strategies for virtual machine migration. *NSDI* **7**, 17–17 (2007)
9. Zhao, Y., Huang, W.: Adaptive distributed load balancing algorithm based on live migration of virtual machines in cloud. In: International Conference on Networked Computing and Advanced Information Management, NCM 2009, pp. 170–175 (2009)



10. Beloglazov, A., Abawajy, J.H., Buyya, R.: Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Futur. Gener. Comp. Syst.* **28**(5), 755–768 (2012)
11. Beloglazov, A., Buyya, R.: Energy efficient resource management in virtualized cloud data centers. In: *Proceedings of the 2010 10th IEEE/ACM international conference on cluster, cloud and grid computing*, pp. 826–831. IEEE Computer Society (2010)
12. Lin, M., Wierman, A., Andrew, L.L.H., Thereska, E.: Dynamic right-sizing for power-proportional data centers. *IEEE/ACM Trans. Netw.* **21**(5), 1378–1391 (2013)
13. Liu, H., Jin, H., Xu, C., Liao, X.: Performance and energy modeling for live migration of virtual machines. *Clust. Comput.* **16**(2), 249–264 (2013)
14. Strunk, A., Dargie, W.: Does live migration of virtual machines cost energy? In: *2013 IEEE 27th International Conference on advanced Information Networking and Applications (AINA)*, pp. 514–521 (2013)
15. Calcavecchia, N.M., Biran, O., Hadad, E., Moatti, Y.: Vm placement strategies for cloud scenarios. In: *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pp. 852–859. IEEE (2012)
16. Hines, M.R., Deshpande, U., Gopalan, K.: Post-copy live migration of virtual machines. *Oper. Syst. Rev.* **43**(3), 14–26 (2009)
17. Strunk, A.: Costs of virtual machine live migration: a survey. In: *Eighth IEEE World Congress on Services, SERVICES 2012, Honolulu, June 24–29, 2012*, pp. 323–329 (2012)
18. Voorsluys, W., Broberg, J., Venugopal, S., Buyya, R.: Cost of virtual machine live migration in clouds: a performance evaluation. *CoRR* (2011). [arXiv:1109.4974](https://arxiv.org/abs/1109.4974)
19. pykalman: the kalman filter, kalman smoother, and em library for python (2017). <http://pykalman.github.io>
20. Masdari, M., Nabavi, S.S., Ahmadi, V.: An overview of virtual machine placement schemes in cloud computing. *J. Netw. Comput. Appl.* **66**, 106–127 (2016). doi:[10.1016/j.jnca.2016.01.011](https://doi.org/10.1016/j.jnca.2016.01.011)
21. Ahmad, R.W., Gani, A., Hamid, S.H.A., Shiraz, M., Xia, F., Madani, S.A.: Virtual machine migration in cloud data centers: a review, taxonomy, and open research issues. *J. Supercomput.* **71**(7), 2473–2515 (2015)
22. Arulselvan, A.: A note on the set union knapsack problem. *Discret. Appl. Math.* **169**, 214–218 (2014)
23. Goldschmidt, O., Nehme, D., Yu, G.: Note: on the set-union knapsack problem. *Nav. Res. Logist. (NRL)* **41**(6), 833 (1994)
24. Project, X.: Xen project: a linux foundation collaborative project. (2016). <http://www.xenproject.org/>
25. The apache thrift software framework (2016). <https://thrift.apache.org/>
26. scikit-learn: Machine learning in python (2016). <http://scikit-learn.org/stable/>
27. Vedhanayagam, P., S., S., Balusamy, B., Vijayakumar, P., Chang, V.: Analysis of measures to achieve resilience during virtual machine interruptions in iaas cloud service. In: *Proceedings of the International Conference on Internet of Things, Big Data and Security, Vol. 1, IoTBDS* (2017)
28. Barbhuiya, S., Papazachos, Z., Kilpatrick, P., Nikolopoulos, D.S.: A lightweight tool for anomaly detection in cloud data centres. In: *Proceedings of the 5th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER*, pp. 343–351 (2015). doi:[10.5220/0005453403430351](https://doi.org/10.5220/0005453403430351)
29. Mhedheb, Y., Streit, A.: Energy-efficient task scheduling in data centers. In: *CLOSER 2016—Proceedings of the 6th International Conference on Cloud Computing and Services Science, Vol. 1, April 23–25, pp. 273–282, Rome* (2016). doi:[10.5220/0005880802730282](https://doi.org/10.5220/0005880802730282)
30. Lou, J.G., Fu, Q., Yang, S., Xu, Y., Li, J.: Mining invariants from console logs for system problem detection. In: *USENIX Annual Technical Conference* (2010)
31. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I.: Detecting large-scale system problems by mining console logs. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pp. 117–132. ACM (2009)
32. Wang, C.: Ebat: online methods for detecting utility cloud anomalies. In: *Proceedings of the 6th Middleware Doctoral Symposium*, p. 4. ACM (2009)
33. Kang, H., Chen, H., Jiang, G.: Peerwatch: a fault detection and diagnosis tool for virtualized consolidation systems. In: *Proceedings of the 7th international conference on Autonomic computing*, pp. 119–128. ACM (2010)
34. Ye, K., Jiang, X., Huang, D., Chen, J., Wang, B.: Live migration of multiple virtual machines with resource reservation in cloud computing environments. In: *IEEE International Conference on Cloud Computing, CLOUD*, Washington, DC, pp. 267–274 (2011)
35. Deshpande, U., Wang, X., Gopalan, K.: Live gang migration of virtual machines. In: *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, pp. 135–146. ACM (2011)
36. Sun, G., Liao, D., Anand, V., Zhao, D., Yu, H.: A new technique for efficient live migration of multiple virtual machines. *Futur. Gener. Comput. Syst.* **55**, 74–86 (2016)
37. Song, X., Shi, J., Liu, R., Yang, J., Chen, H.: Parallelizing live migration of virtual machines. In: *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Houston, TX, USA, March 16–17*, pp. 85–96 (2013)
38. Liu, H., He, B.: Vmbuddies: coordinating live migration of multi-tier applications in cloud environments. *IEEE Trans. Parallel Distrib. Syst.* **26**(4), 1192–1205 (2015)
39. Berthier, N., Rutten, E., Depalma, N., Gueye, S.: Designing autonomic management systems by using reactive control techniques. *IEEE Trans. Softw. Eng.* (99), 1–1 (2015)
40. Deshpande, U., Keahey, K.: Traffic-sensitive live migration of virtual machines. In: *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*, pp. 51–60 (2015)
41. Rybina, K., Patni, A., Schill, A.: Analysing the migration time of live migration of multiple virtual machines. In: *Proceedings of the 4th International Conference on Cloud Computing and Services Science*, pp. 590–597 (2014)
42. Dong, D., Herbert, J.: Precise VM placement algorithm supported by data analytic service. In: *CLOSER*, pp. 463–468 (2013)
43. Fang, W., Liang, X., Li, S., Chiaraviglio, L., Xiong, N.: Vmplanner: optimizing virtual machine placement and traffic flow routing to reduce network power costs in cloud data centers. *Comput. Netw.* **57**(1), 179–196 (2013)
44. Diallo, M.H., August, M., Hallman, R., Kline, M., Au, H., Beach, V.: Nomad: A framework for developing mission-critical cloud-based applications. In: *10th International Conference on Availability, Reliability and Security, ARES 2015, Toulouse, France, August 24–27, 2015*, pp. 660–669 (2015)
45. Diallo, M.H., August, M.A., Hallman, R.A., Kline, M., Au, H., Beach, V.: Callforfire: A mission-critical cloud-based application built using the nomad framework. In: *Twentieth International Conference on Financial Cryptography and Data Security: 4th Workshop on Encrypted Computing and Applied Homomorphic Cryptography, Christ Church, Barbados, February 22–26* (2016)



**Mamadou H. Diallo** is a Scientist at the United States Space and Naval Warfare Systems Center Pacific in San Diego, California. He is a Science and Technology Manager and currently leading a number of Department of Defense research and development projects. He is also a PhD candidate in Computer Science at the University of California, Irvine, USA, in his final year. He has multiple publications in well-respected conferences, workshops, and book

chapters, spanning across various areas including data security and privacy in cloud computing, cryptography, security requirements, trust management in decentralized applications, dependability in software architecture, policy languages, and middleware. He is a lead or co-inventor on multiple patent applications.



**Michael August** is a Computer Scientist in the Enterprise Communications and Networks division at the U.S. Space and Naval Warfare Systems Center Pacific in San Diego, California, USA. He researches emerging technologies within the cloud and mobility areas for application to U.S. Navy problems.



**Roger Hallman** is a Research Scientist with SPAWAR Systems Center Pacific's Cybersecurity Science & Technology Branch. His research interests include security and privacy in the IoT domain, fully homomorphic encryption, autonomous vehicles, and artificial intelligence. In addition to his research, Mr. Hallman is chairing the 1st International Workshop on Multimedia Privacy and Security at CCS 2017 and is a co-chair of the Workshop on Innovative Cyber-

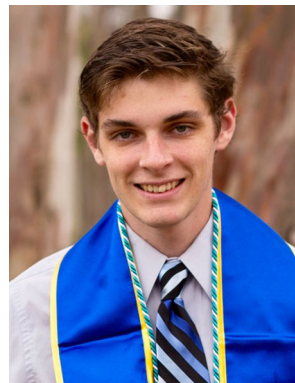
security and Privacy for Internet of Things and Enterprise Security at IoTBDS 2018.



**Megan Kline** is a Mathematician and Cybersecurity Researcher working at SPAWAR Systems Center, Pacific in San Diego. Her areas of research include decision support for security technologies, risk metrics for Android applications, fully homomorphic encryption, and security for SCADA and Industrial Control Systems.



**Scott M. Slayback** is a new employee at the Space and Naval Warfare Systems Center Pacific in San Diego, California. He holds an M.S. degree in Computer Science from the Naval Postgraduate School. He contributes to a number of Department of Defense research projects, including a project that focuses on data security in the cloud. He has extensive experience with various programming languages and computer networking.



**Christopher Graves** is a Computer Scientist at SPAWAR Systems Center Pacific, San Diego. His areas of research interest include autonomies of cloud systems used for security and energy management. He is currently the Publicity and Web chair of the 1st International Workshop on Autonomics and Cloud Security.