J Supercomput (2018) 74:2956–2983
https://doi.org/10.1007/s11227-018-2345-2

Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications

Carlos Guerrero[1] · **Isaac Lera**[1] · **Carlos Juiz**[1]

Published online: 2 April 2018

Abstract An approach to optimize the deployment of microservices-based applications using containers in multi-cloud architectures is presented. The optimization objectives are three: cloud service cost, network latency among microservices, and time to start a new microservice when a provider becomes unavailable. The decision variables are: the scale level of the microservices; their allocation in the virtual machines; the provider and virtual machine type selection; and the number of virtual machines. The experiments compare the optimization results between a Greedy First-Fit and a Non-dominated Sorting Genetic Algorithm II (NSGA-II). NSGA-II with a two-point crossover operator and three mutation operators obtained an overall improvement of 300% in regard to the greedy algorithm.

Keywords Microservices · Cloud computing · Container orchestration · Genetic algorithm · Multi-objective optimization

1 Introduction

Microservices have become a new development pattern widely adopted during the last years. Microservices-based applications are designed as a set of independent, small and modular services, called microservices, which implement a single task and communi-

✉ Carlos Guerrero
carlos.guerrero@uib.es

Isaac Lera
isaac.lera@uib.es

Carlos Juiz
cjuiz@uib.es

[1] Computer Science Department, Balearic Islands University, 07122 Palma, Spain

cate between them using messages. The requirements of the applications are achieved by the composition of a set of microservices [4]. The benefits of microservices-based applications are making easier to ship and update applications, allowing independent updating and redeployment of parts of the application, facilitating closer development and operation teams, allowing continuous release cycles, simplifying the orchestration of applications across heterogeneous cloud data centers, and so on [14]. These benefits have promoted a general movement to migrate application to this development pattern, even in important companies, such as Netflix [51], or Amazon [41].

The popularity of microservices has been also favored by the emergence of containers and their most popular implementation, Docker [38]. Containers are an operating system level virtualization technology which offers independent execution environments and isolated file systems. Container images are incremental and layered file systems which only include the additional files and libraries that are not in the underlying operating system, resulting in light-weighted instances reducing the overhead of the virtualization [38]. Containers are suitable elements to encapsulate, isolate, and scale microservices, resulting in one of their most important enabling technologies. For example, Balalaie et al. [4] reported the lessons learned from the migration of a commercial mobile back end to a microservices architecture deployed with containers.

Early adopters of containers and microservices deployed the applications in single cloud architectures [29]. However, tools for the deployment of containers in multi-cloud have recently emerged, such as Kubernetes Containership [7,17] or HashiCorp Nomad [8,25]. By the use of these tools, the deployment of microservice applications has resulted in a faster and easier task. For example, Sousa et al. [49] presented a framework for automated deployment of microservices applications in multi-cloud environments with containers and created a e-commerce application to validate it.

The deployment of microservices-based applications in multi-cloud architectures runs the risk of increasing the application execution time. For instance, if two interoperated microservices are allocated in different providers, the network delay between them is increased compared to their allocation in the same provider. In contrast, benefits in terms of cost can be achieved by allocating the microservices in the cheapest provider, or also benefits in terms of repair times, by replicating the microservices in several providers. Resource management techniques need to be proposed addressing and balancing those opposite optimization objectives.

Resource management is used to meet end users' requirements through the mapping of tasks to resources. Resource management is a complex problem in cloud environments due to high scale levels, heterogeneous nature of the components, and high variation of unpredicted workloads [6]. Resource management and optimization are problems that, to the best of our knowledge, have been only partially studied in the field of multi-cloud and containers. For example, Hoenisch et al. [27] studied the deployment of traditional applications, but they did not address microservices-based ones. Moreover, current container management tools, such as Kubernetes, Docker Swarm, or Nomad, do not implement optimization policies, or they are basic ones, remaining ample room for improvement [9].

We address the resource optimization of multi-cloud container orchestration for containers and microservices. We minimize three metrics: total monetary cost of the deployment; the increase of the application execution time due to the distribution of

microservices among the cloud providers, measured in terms of network latency among containers with inter-related microservices; and, finally, the microservice repair time, measured in terms of the time that elapses until a second instance of a microservice is available when one cloud provider or one VM instance fails. The optimization is achieved by managing, orchestrating, and scaling containers and VMs. We study two alternatives to solve our multi-objective optimization: a stochastic solution based on a greedy algorithm and an evolutionary approach based on a genetic algorithm, the Non-dominated Sorting Genetic Algorithm II or NSGA-II.

Our main contributions are: (i) a multi-objective approach to improve the orchestration of containers encapsulating microservices in multi-cloud systems; (ii) the study of the Greedy First-Fit and NSGA-II to address the optimization problem; (iii) the experimental evaluation of both algorithms.

The paper is organized as follows: The introduction is completed with the explanation of a motivational example in Sect. 1.1; Sect. 2 presents related research works; Sect. 3 describes our approach and formalizes the domain problem and the optimization objectives; Sect. 4 describes the implementation of the optimization based on a genetic algorithm; Sect. 5 describes the implementation of the optimization based on a greedy algorithm; Sect. 6 describes the experiments conducted to evaluate our approach and the results of both optimization algorithms. Additionally, an analysis of the evolution of the solutions in the genetic approach is also presented in this section. Finally, the conclusion and future works are explained in the last section.

1.1 Motivational scenario

Consider a system administrator that needs to deploy an application which is created from the interoperation of three microservices. Each of the microservices is encapsulated in a container image, and the microservice is deployed by creating instances of the containers in a VM. The system administrator disposes of three cloud providers where the VMs with the containers can be deployed, just by selecting the desired VMs characteristics (resources and cost) among the VM templates offered by each provider.

The system administrator has two management tools: one for the orchestration of the microservices and a second one for the VM management. Real examples of those tools are OpenNebula [52], for the automatization and management of VMs in multi-cloud providers, and HashiCorp Nomad [45], for the container orchestration. These tools allow the system administrator to easily deploy the application in any of the three cloud providers, first by deciding the number and type of the VMs to create and second by deciding the scale level of the containers and the VMs where they are allocated in.

The system administrator is concerned with the risk of increasing the application execution time due to the additional network latency among microservices allocated in different providers. However, he is also interested in the reduction of deployment cost and in the improvement of the application availability by considering several cloud providers. The solution is easily affordable if only one of those objectives is considered. For example, the application availability is easily maximized by creating instances of the three microservices (or their containers) in two providers, but this solution will also increase the deployment cost. In contrast, the VM and container

allocation plan is not so simple if the three objectives are considered together, even for the small architecture size of the example.

An external pluggable solution is required to be able to manage concurrently the VM and container management tools, since those tools are not coordinated between them. By gathering information from both components, an optimization process can be conducted, resulting in a VM and container orchestration plan that can be returned to the management tools in order to change the allocation of both VMs and containers. Thus, the decision-making process is automated and, for example, the deployment cost can be reduced without damaging the application performance or availability.

We have designed a pluggable approach whose inputs are the system characteristics and the output the VM and containers allocations and scale levels. Thus, our solution can be included in a real scenario by integrating the inputs and outputs of our approach with both the VM and container managers, for example, OpenNebula and HashiCorp Nomad.

2 Related work

There are important open problems in the field of multi-cloud resource management and brokering [19]. Traditionally, resource management has been addressed separately for multi-cloud and for container orchestration. To the best of our knowledge, this is the first work dealing with container orchestration in multi-cloud architectures for the specific case of deploying microservices inside the containers. The related research is presented in two main parts: multi-cloud resource management and container orchestration. Table 1 shows the main features of each work and clarifies the differences among previous studies and our approach.

First, we present studies in the field of heterogeneous multi-cloud scheduling problem. For example, Tordsson et al. [52] optimized cost and performance in heterogeneous multi-cloud architectures using linear programming which manages the allocation of virtual machines (VMs) in physical machines (PMs). Pietrabissa et al. [46] proposed an improved Cloud Management Broker to maximize the revenue over the time. Their solution was based on Markov decision process modeling, reinforced with learning techniques. Fard et al. [13] proposed a task scheduling process to improve completion time and monetary cost in multi-cloud environments based on game theory, the VCG reverse auction mechanism. Heilig et al. [26] proposed a genetic algorithm to satisfy cloud user requirements for multi-cloud deployment of tasks. Frincu and Craciun presented a multi-objective algorithm to achieve high availability and fault-tolerance and to reduce cost by keeping maximized the resource usages by optimizing the allocation of tasks in physical machines (PM). They implemented the solution with a genetic algorithm and compared the results with the traditional round-robin strategy [16]. Li et al. [32] proposed a strategy allocating VMs in multi-tenant data centers with the objective of reducing the network diameter and the resource fragmentation based on the Multiple Knapsack Problem (LP-MKP). Amato et al. [3] explored the use of NSGA-II for the scheduling of tasks in cloud providers to optimize time, cost, availability, and reliability. Panda et al. [42] compared the results of three task scheduling algorithms across multiple PaaS (Platform as a Service) cloud environ-

Table 1 Related work summary

References	Architecture	Management	Workload	Resources	Solution	Optimization
[52]	Multi-cloud	Allocation	VM	Data center	Linear programming	Performance, cost and load bal.
[46]	Multi-cloud	Scheduling	Task	Cloud prov.	Markov process	Revenue
[13]	Multi-cloud	Scheduling	Task	Cloud prov.	Reverse auction	Monetary cost and makespan
[26]	Multi-cloud	Scheduling	Task	VM	Biased random-key	Resource usage
[16]	Multi-cloud	Scheduling	Task	PM	Genetic algorithm	Avail., cost and resource usage
[32]	Multi-cloud	Allocation	VM	PM	Multiple Kanapsack (LP-MKP)	Network dist. and resource frag.
[3]	Multi-cloud	Scheduling	Task	Cloud prov.	NSGA-II	Availability, cost, time and reliability
[42]	Multi-cloud	Scheduling	Task	Cloud prov.	Smooth-based	Makespan
[43]	Multi-cloud	Scheduling	Task	Cloud prov.	Stochastic	SLA
[31]	Multi-cloud	Scheduling	Service	VM	GA	Monetary cost
[35]	Multi-cloud	Scheduling	VM	Cloud prov.	Cost functions	Monetary cost and performance
[33]	Cloud	Allocation	Container	PM	Cost functions	Load balanc. and resource usage
[1]	Cloud	Horizontal scaling	Container	PM	Stochastic	Provisioning and cost

Table 1 continued

References	Architecture	Management	Workload	Resources	Solution	Optimization
[10]	Cloud	Horizontal scaling	Container	PM	Own algorithm	Performance
[47]	Cloud	Allocation	Container	PM	Greedy algorithm	Energy
[20]	Cloud	Allocation	Container	PM	Linear prog.	Cost
[28]	Cloud	Scheduling	Task	Container	Ant colony and greedy alg.	Performance
[50]	Cloud	Allocation	Container	PM	Fuzzy inference	Performance
[21]	Cloud	Container allocation	Container	VM	NSGA-II	Latency, network distance, load balancing, and resource usage
[40]	Cloud	Alloc. and VM scaling	Container	VM	Linear prog. and greedy alg.	Deployment time and cost
[27]	Multi-cloud	VM selection, Container allocation, and VM scaling	Container and VM	VM and cloud prov.	NSGA-II and linear prog.	Cost, deployment time and free resources
[Our work]	Multi-cloud	VM selection, Container allocation, VM scaling, and microservices scaling	Container and VM	VM and cloud prov.	NSGA-II and greedy alg.	Cost, repair time & microservices net. latency

ments: Minimum Completion Cloud (MCC), Median Max (MEMAX), and Cloud MinMax Normalization (CMMN). The same authors presented a second work where the SLA in multi-cloud is improved with the use of one and two stochastic phase scheduling algorithms [43]. Legillon et al. [31] proposed a genetic algorithm for placing services into heterogeneous VMs from different IaaS providers. The evolutionary approach is compared with the use of a Mixed-Integer Programming (MIP) solver.

Second, the problem of optimization for container orchestration is a very recent topic. Nonetheless, there are some studies along the literature but limited to cases with only one cloud provider or data center. Li et al. [33] proposed an optimal minimum migration algorithm to reduce the number of container migrations inside a data center while the balanced load and the usage ratios are improved. Adam et al. [1] proposed a self-scaling container orchestrator to reduce over-provisioning in a PaaS architecture. Alfonso et al. [10] created an auto-scaled manager for virtual computer clusters of containers on bare-metal machines for scientific workloads. Piraghaj et al. studied several strategies for container consolidation [47]. They analyzed the impact of straightforward and greedy allocation policies in data center metrics. Guan et al. [20] proposed a container-based resource allocator based on the use of Linear Programming algorithms to minimize the deployment cost in data centers and to scale the containers automatically and horizontally when the workload is increased. Kaewkasi and Chuenmuneewong [28] proposed an ant colony optimization for task scheduling for Docker containers, by maximizing the resource usages. Their approach was also compared with the results obtained with a greedy algorithm. Tao et al. [50] studied the use of fuzzy inference systems in container allocation problem in physical machines. Their solution was compared with a traditional round-robin technique. We previously studied a twofold optimization of container orchestration [21], but we studied the deployment of the container in physical machines instead of VMs, and we did not deal with multiple cloud providers.

The works of Nardelli et al. [40] and Hoenisch et al. [27] are the most similar ones to our approach but with significant differences. Nardelli did not considered multi-cloud environment neither scalability of the containers/VMs. Hoenisch included the use of multi-cloud and scalability in their model, but they only considered a scalable number of VMs and not for the containers. None of both works considered the use of microservices, so network delays due to the interactions between the containers were not included in their models. Our optimization objectives are also different to theirs one.

We have decided to validate our approach by studying and comparing two optimization algorithms: NSGA-II and Greedy First-Fit. Both alternatives are representative and common optimizers in other studies [23]. There are several works that use the comparison between genetic approaches and greedy algorithm to validate their works [59].

3 Problem statement

The most representative elements in our domain are cloud providers, VM types, VM instances, containers, and microservices. Their characterization parameters are sum-

Table 2 Summary of domain elements and characterization parameters

Parameter	Description
$prov_n$	The cloud provider with identification n
$L_{n,n'}$	Latency time between providers' data centers
$vmt_{n,i}$	VM type with identification i in cloud provider n
$R_{n,i}^{avail}$	Available computation resources in virtual machine type i
C_{total}	Total cost of the cloud services
$C_{n,i}$	User cost for the virtual machine type i
$vmi_{n,i}^{j}$	Instance of a virtual machine of type i in provider n
$ms_{x,y}$	Microservice with identification y for application x
$msi_{x,y}^{z}$	Instance z for a microservice identified as y for the application x
$R_{x,y}^{req}$	Computational resources required by a running container of microservice y
$R_{x,y}^{str}$	Computational resources required by a stored container of microservice y
$MS_{x,y}^{cons}$	The set of microservices consumed by the microservice y
$T_{x,y}^{start}$	Time to start a container of the microservice y
$T_{x,y}^{dwnld}$	Time to download the container image of microservice y
T^{repair}	Total repair time for all the microservices in the systems
$T_{vm}^{repair}[msi_{x,y}^{z}]$	Total repair time for the microservice instance z in case of a VM failure
$T_{provider}^{repair}[msi_{x,y}^{z}]$	Total repair time for the microservice instance z in case of a provider failure
$U[vmi_{n,i}^{j}]$	Resource usage of the VM instance j of type i in provider n
$alloc[msi_{x,y}^{z}]$	Function that returns the allocation of the microservice instance z
$store[msi_{x,y}^{z}]$	Function that returns the VM where the microservice instance z is stored

marized in Table 2, and an illustrative example of our domain model is explained in Sect. 3.2.

We consider a set of cloud providers $prov_n$ hosting VMs and characterized by the communication time or latency overhead among their data centers. This is modeled as a matrix of size $n \times n$ where the element $L_{n,n'}$ represents the latency time between the data centers of providers $prov_n$ and $prov_{n'}$.

The VM characteristics are defined by the VM types $vmt_{n,i}$ with the tuple $< R_{n,i}^{avail}, C_{n,i} >$, where $R_{n,i}^{avail}$ represents the number of virtual CPUs of an instance of $vmt_{n,i}$ and $C_{n,i}$ is the characteristics of the considered billing model. In order to simplify the model and the notation, our approach only considers single representations for the computational resource capacities and for the billing model. Both cases can easily be extended using, for example, a tuple for each resource element in the case of the resource capacities (e.g., the number of virtual CPUs, the main memory size, the storage size, or the input/output bandwidth) and additional cost characteristics in the case of the billing model.

Cloud users create the VMs by instancing the VM types. A VM instance is represented as $vmi_{n,i}^{j}$, and it inherits the characteristics of the VM types.

The user applications are composed of microservices $ms_{x,y}$. These microservices are related in terms of consumption. A microservice $ms_{x,y}$ consumes the microservice $ms_{x,y'}$ when the former needs the results of the latter to perform its tasks.

Our domain considers that microservices are encapsulated and deployed in containers. Each container only includes the instance of one microservice. We indistinctly use the terms and concepts of microservice instance and container because their differences do not influence in our domain definition. The deployment, or horizontal scaling up, of microservices is as easy as starting a new container. If the container image is not stored in the local storage of the virtual machine, it is downloaded from a repository.

A microservice is characterized by a tuple $<R^{req}_{x,y}, R^{str}_{x,y}, MS^{cons}_{x,y}, T^{start}_{x,y}, T^{dwnld}_{x,y}>$, where $R^{req}_{x,y}$ indicates the resource consumption of a running container; $R^{str}_{x,y}$ is the resource consumption of a stored container image; $T^{start}_{x,y}$ is the time a container takes to start; $T^{dwnld}_{x,y}$ is the time to download the container image that encapsulates the microservice; and $MS^{cons}_{x,y}$ is the set of consumed microservices.

The instance of a microservice $ms_{x,y}$ is represented as $msi^{z}_{x,y}$. The container instances are allocated in VM instances, and it is represented with the function $alloc{:}|msi^{z}_{x,y}| \rightarrow |vmi^{j}_{n,i}|$. The same representation is used for function $store$ to indicate that a container image is downloaded but not running in a VM instance. A stored container image consumes less resources and has a smaller starting time than a running container, since it is not downloaded [24].

3.1 Optimization objectives definition

We propose optimizing: (i) the cost of the cloud services for the cloud user; (ii) the latency overhead due to the allocation of the related microservices in different cloud providers or VM instances; (iii) and, finally, the time that any microservice is unavailable due to a failure of one VM, or one cloud provider. We propose achieving this optimization by managing: (a) the VM types and consequently the cloud providers; (b) the number of VM instances for each type; (c) the scale level of each microservices; and (d) the container allocations in the VMs.

The cloud service cost objective is measured as the sum of the cost of each VM instance. The cost of the VM instances depends on the VM type. Our billing model is defined as a linear function of the VM utilization, i.e., the cost is calculated as the percentage of the total price proportional to the usage factor of the resources in each VM instance:

$$C_{total} = \sum_{vmi^{j}_{n,i}}^{j} C_{n,i} \times U\left[vmi^{j}_{n,i}\right] \tag{1}$$

Resource usage of a VM instance, $U[vmi^{j}_{n,i}]$, is calculated as the sum of the resources consumed by the running container instances—first summation—and the stored container images—second summation—it allocates:

$$U\left[vmi_{n,i}^{j}\right] = \sum_{alloc\left[msi_{x,y}^{z}\right]=vmi_{n,i}^{j}} R_{x,y}^{req} + \sum_{store\left[msi_{x,y}^{z}\right]=vmi_{n,i}^{j}} R_{x,y}^{str} \tag{2}$$

The latency overhead objective is calculated, for a given microservice, as the sum of the minimum distance with each instance of its consumed microservices. The latency between two microservices is 0 when the consumed microservice is running in the same VM instance than the consumer, and $L_{n,n'}$ when they are in different VM. The value of $L_{n,n'}$ is defined by the network distance between the physical nodes that allocate the VMs, i.e., the value would depend on the provider they are allocated and whether the VMs are in the same provider or not. The total latency overhead for all the microservices can be formulated as:

$$\begin{aligned} &L_{total} = \sum_{msi_{x,y}^{z}}^{z} \left[\sum_{ms_{x,y'} \in MS_{x,y}^{cons}} min(L_{n,n'}) \right] \\ &as\ alloc\left[msi_{x,y}^{z}\right] = prov_n \wedge prov_{n'} = alloc\left[msi_{x,y'}^{z'}\right] \forall\, msi_{x,y'}^{z'} \in MS_{x,y}^{cons} \end{aligned} \tag{3}$$

Finally, the repair time objective is the time to start a new instance of a microservice for two cases: a provider failure or a VM failure. This repair time is 0 when other microservice instances are running in other providers (or VM instances). If there is not any other running instance, the repair time is the container starting time when at least another provider (or VM) stores the container image of the microservice. The worst case is when none running or stored containers are available in other providers (or VMs), because the downloading time is added to the starting time. This can be formalized as:

$$T^{repair} = \sum_{msi_{x,y)}^{z}}^{z} \left[T_{vm}^{repair}\left[msi_{x,y}^{z}\right] + T_{provider}^{repair}\left[msi_{x,y}^{z}\right] \right] \tag{4}$$

$$T_{vm}^{repair}\left[msi_{x,y}^{z}\right] = \begin{cases} 0.0, & if\ \ \exists\, msi_{x,y}^{z} \wedge msi_{x,y}^{z'} \\ & |\ alloc\left[msi_{x,y}^{z}\right] \neq alloc\left[msi_{x,y}^{z'}\right] \\ T_{x,y}^{start}, & if\ \ \exists\, msi_{x,y}^{z} \wedge msi_{x,y}^{z'} \\ & |\ alloc\left[msi_{x,y}^{z}\right] \neq store\left[msi_{x,y}^{z'}\right] \\ T_{x,y}^{start} + T_{x,y}^{dwnld}, & otherwise \end{cases} \tag{5}$$

The $T_{provider}^{repair}$ function is formulated equally to the T_{vm}^{repair} one, but considering the allocation in the cloud providers instead of in the VM instances.

Therefore, our three-objective optimization problem can be formulated as

To determine:

$$|vmt_{n,i}| \wedge |vmi^{j}_{n,i}| \wedge |msi^{z}_{x,y}| \wedge alloc() \wedge store() \tag{6}$$

By minimizing:

$$C_{total} \wedge L_{total} \wedge T^{repair} \tag{7}$$

Subject to the constraints:

$$size\left(|msi^{z}_{x,y}|\right) > 1 \quad \forall\, ms_{x,y} \tag{8}$$

$$U[vmi_{j}(n,i)] < R^{avail}_{n,i} \quad \forall\, vmi^{j}_{n,i} \tag{9}$$

The solutions are constrained to consider that at least one instance of each microservice is running and that the sum of the computational resources consumed by the microservices allocated in a VM is smaller than the computational capacity of the VM.

This optimization problem is an NP-complete problem because all the possible scale levels, allocations, and VM types combinations should be evaluated to find the optimal solution. A metaheuristic method needs to be used to address the problem [54].

3.2 Model example

We include the modeling of an example to clarify the nomenclature and design of the architecture. Figure 1 presents a case with an application that is formed by three microservices. The characterization parameters of the microservices (resource consumptions and starting and downloading times) are included in the nodes of the application graph. Additionally, the system is formed by two cloud providers. The first one offers three VM templates, represented by the array of square boxes, which includes the VM characteristics (cost and resource capacity). The second provider offers two VM templates. The instances of the VMs are represented with rounded squares, and an arrow indicates their VM types. Each rounded square includes the identifiers of the microservices instances it allocates.

Under these conditions, the execution of the application starts requesting the unique instance of $ms_{1,1}$ in the first provider. This first microservice requests the closest instance of the second microservice, also allocated in $prov_1$ but in a different VM. Thus, the network latency between $msi^{1}_{1,1}$ and $msi^{1}_{1,2}$ is the intra-latency of the first provider ($L_{1,1} = 0.1$). Finally, the second microservice requests one of the instances of $ms_{1,3}$, in the second provider, accumulating an additional network latency of $L_{1,2} = 2.5$. Thus, the total network latency of this VM and container allocation plan is, considering Eq. 3, $L_{total} = 0.1 + 2.5 = 2.6$.

For the case of the deployment cost, we first calculate the resource usage of each VM instance by the summation of the resources required by the microservices in a VM and divided by its available resources, considering Eq. 2. For example, $vmi^{1}_{2,2}$ contains two microservice instances, $ms^{2}_{1,2}$ and $ms^{2}_{1,3}$, resulting in $0.3+0.4 = 0.7$ consumption

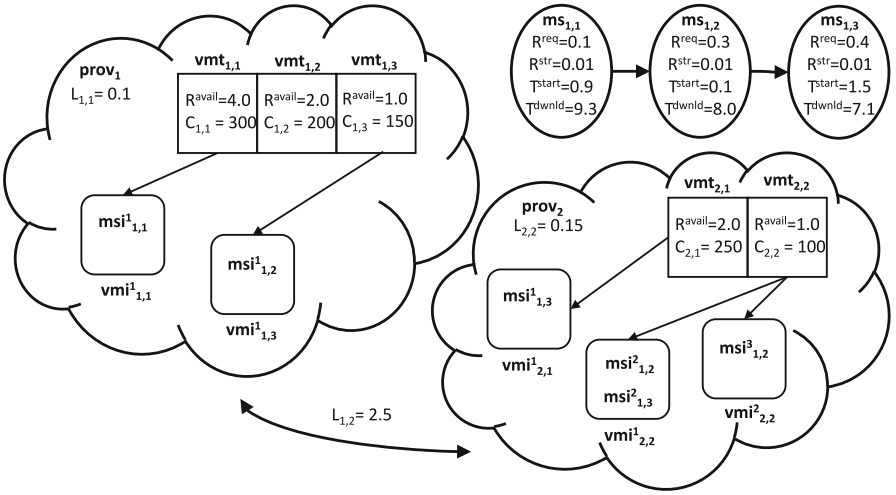

Fig. 1 Graphical representation of a model example with an application of 3 microservices and 2 cloud providers with 3 and 2 VM templates, and 5 VM instances

of the resources. The resource usage of the VM is then $\frac{0.7}{1.0} = 0.7$. Consequently, the cost of this VM instance is calculated with Eq. 1, resulting in $100 \times 0.7 = 70$. The total cost of the configuration is calculated as $C_{total} = \frac{0.1}{4.0} \times 300 + \frac{0.3}{1.0} \times 150 + \frac{0.4}{2.0} \times 250 + \frac{0.7}{1.0} \times 100 + \frac{0.3}{1.0} \times 100 = 277.5$.

Finally, the provider repair time, $T^{repair}_{provider}$, is 0.0 when there are instances of the same microservices in at least two cloud providers. In the same way, the VM repair time, T^{repair}_{vm}, is 0.0 if al least two VMs have instances of a microservice. Otherwise, the repair time is the time of starting a container instance and, if there is not any stored container in other providers/VMs, also the downloading time. For the sake of clarity, the example does not consider any stored container. In this way, $ms_{1,1}$ only has one instance and consequently its VM repair time is $T^{start}_{1,1} + T^{dwnld}_{1,1} = 0.9 + 9.3 = 10.2$, and the same value for the provider repair time. Microservice $ms_{1,2}$ has instances in more than one VM and in more than one provider, consequently $T^{repair}_{vm}[ms_{1,2}] = T^{repair}_{provider}[ms_{1,2}] = 0.0$. Finally, the third microservice has instances in more than one VM, but all its VMs are allocated in the same provider. Consequently, $T^{repair}_{vm}[ms_{1,3}] = 0.0$ but $T^{repair}_{provider}[ms_{1,3}] = 1.5 + 7.1 = 8.6$. To summarize, the total repair time is $T^{repair} = 10.2 + 10.2 + 8.6 = 29.0$.

4 Genetic algorithm proposal

Computational intelligence (CI) is a common solution for resource management problems in the field of cloud resource management [23]. Typically, techniques included in CI are evolutionary algorithms, ant colony optimization, particle swarm optimization, firefly algorithms, artificial neural networks, or fuzzy systems between others [58]. The No-Free Lunch Theorem [55] predicted that none of these techniques is clearly

superior to any other in general terms. Consequently, it is necessary to study the applicability of these techniques in each new particular problem.

Genetic algorithms (GA) are metaheuristics included in the evolutionary approaches for solving optimization problems. Non-dominated Sorting Genetic Algorithm II (NSGA-II) is a common GA implementation for the case of multi-objective problems [12]. It has resulted as a suitable algorithm in the optimization of resource management for VM allocation [2], data replica placement [22], federated clouds [30], or software composition [15]. The use of any of the other CI techniques remains as an open research problem for future works because, as we have just explained, a particular study of each CI technique needs to be performed for each new addressed problem.

The following subsections explain the details of the adaptation of each part of the GA to our optimization problem. Additionally, we include a general structure of NSGA-II in Algorithm 1 for a better understanding of the following subsections. The algorithm is a transcription of the original algorithm presented by Deb et al. [12] and we have included our parametrization values, such as population size, number of generations, and mutation probability.

4.1 Chromosome representation

Solutions are called individuals in GAs, and they are represented using a string or array notation which is called chromosome. Our solutions need to represent the number, type, and provider selection of the VM instances and the scale level and allocation of the microservices. The number of microservices is fixed and determined by the application to deploy. The number of instances of each microservice and the number of VM instances are variable. Therefore, we use two structures to represent each solution: the *vmChromoseme* to represent the VM instances and the *msChromosome* to represent the number and allocation of the microservices instances. Figure 2 shows an example of two individual representations for an application with 3 microservices and 5 and 4 VM instances, respectively.

The *vmChromosome* is a variable size array of integers. Each position of the array represents a VM instance, and the integer value of the array position represents the VM type. The allocation of the VM in a cloud provider is implicitly represented with the selection of the VM type, as far as it is associated with only one cloud provider. The *msChromosome* is a bi-dimensional array where the rows correspond to the fixed number of microservices and the columns correspond to the variable number of VM machine instances, equal to the size of the *vmChromosome*. Each position of the bi-dimensional array is an integer value which indicates the number of microservices instances in the VM. Value 0 means that the microservice image is stored in the VM, but it is not running. Value -1 means that the microservice is neither running nor stored.

4.2 Crossover and mutation operator

GAs are based on the evolution of a set of solutions, also called population, through the algorithm iterations, also called generations. The best solutions of a generation

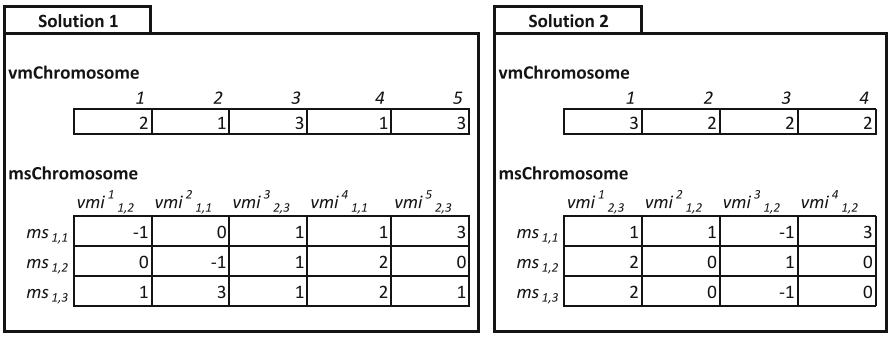

Fig. 2 Example of a chromosome representation for two solutions

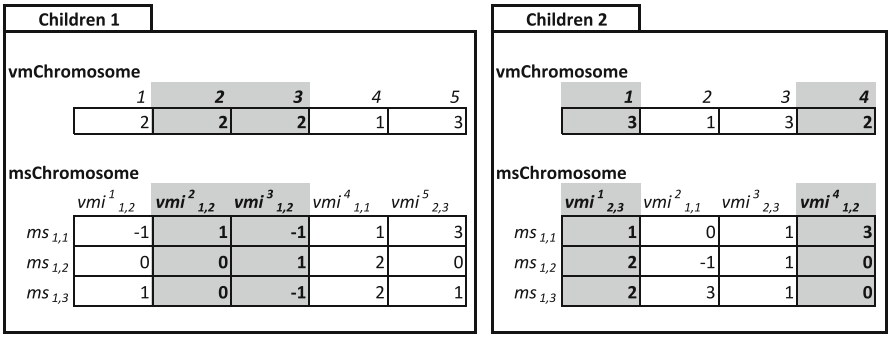

Fig. 3 Example of the offspring obtained with crossover point values of 2 and 4 for the previous solution examples

are combined using crossover operations to create the next population or offspring. Additionally, some solutions in the offspring are modified with the mutation operators.

Although crossover operators are commonly used, they tend to cause building blocks disruption and consequently a poor optimization process [11]. We conducted a previous exploratory phase in which three operators were tested: one-point, two-point, and uniform crossover. The results showed that the two-point crossover operator obtained the best performance. The uniform operator was damaged by the own nature of our solution representation, since a mix of genes (a VM with their allocated containers) from each of the parents is very disruptive for a container allocation planning. In contrast, the two-point crossover replaced just a set of intermediate genes in the chromosome, allowing that the arbitrary order of the genes did not influence in the matting operation, unlike to the one-point crossover.

Therefore, we finally adopted a two-point crossover operator to generate two new solutions from two parent solutions [39]. It consists of the generation of two random numbers which split the chromosome into three pieces. The first and third pieces of one parent solution are combined with the second piece of the other parent to generate a new individual. A second individual is generated by combining the opposite pieces. In our particular case, the same two random numbers are used to split the *vmChromosome* and the columns of the *msChromosome*. Figure 3 shows the results of applying the

Algorithm 1 Multi-objective optimization algorithm [12]

```
1: procedure NSGA- II
2:     populationSize ← 200
3:     generationNumber ← 250
4:     mutationProb ← 0.25
5:     P_t ← generateRandomPopulation(populationSize)
6:     fitness ← calculateFitness(P_t)
7:     fronts ← calculateFronts(P_t, fitness)
8:     distances ← calculateCrowding(P_t, fronts, fitness)
9:     for i < generationNumber do
10:        P_off = ∅
11:        for j < populationSize do
12:            father1, father2 ← binaryTournamentSelect(P_t, fronts, distances)
13:            child1, child2 ← crossover(father1, father2)
14:            if random() < mutationProb then mutate(child1),mutate(child2)
15:            P_off = P_off ∪ {child1, child2}
16:        P_off = P_off ∪ P_t
17:        fitness ← calculateFitness(P_t)
18:        fronts ← calculateFronts(P_off, fitness)
19:        distances ← calculateCrowding(P_off, fronts, fitness)
20:        P_off = orderElements(P_off, fronts, distances)
21:        P_t = P_off[1..populationSize] #the best half
22:    Solution = fronts[1] #the Pareto front
```

crossover to the solutions of Fig. 2 when the random cutting points are 2 and 4. Gray background corresponds to the data from solution 2 and white background to solution 1.

The mutation operators are included in GAs to avoid local minimums and to cover a wider solution search space. Three mutation operators are considered in our approach: *Add VM*, a new VM is included in the solution with random microservices allocated in it; *Del VM*, a VM and all the containers instances it allocates are deleted; *Change Containers*, the containers allocated in a VM are randomly changed.

4.3 Fitness function, selection operator, and offspring generation

The fitness function measures the quality of the solutions, and it is generally calculated with the objective functions. For multi-objective optimizations, several values are considered and the comparison of the fitness values is performed using the dominance concept. A solution s_1 dominates another solution s_2 if the fitness values of s_1 are better than the values of s_2 for all the objectives. In the same way, a solution s'_1 non-dominates another solution s'_2 if the fitness values of s'_1 are better for a subset of objectives and worse for the others. Consequently, the Pareto optimal front is defined as the set of solutions that are non-dominated by other solutions. The other solutions are classified iteratively in fronts or sets of non-dominated solutions. Each solution in a front optimizes one or more objectives, but not all of them, compared to the other solutions in the same front. Therefore, the *best* solutions are classified in the first front, also called Pareto optimal front, the second *best* solutions in the second front, and so on.

Additionally, NSGA-II establishes a mechanism to classify the solutions inside a front, the crowding distance. This metric represents the density of solutions surrounding a given one by considering the average distance of all the objectives in relation with the closest neighbors [12]. NSGA-II considers a solution with higher quality when it does not have surrounding solutions.

NSGA-II sorts the solutions in front levels, and all the solutions in the same front level are ordered using the crowding distance (lines 17–20 in Algorithm 1). Once all the solutions are sorted, it applies a binary tournament selection operator over the sorted elements (lines 11–12): Two solutions are randomly selected from the population, and the *best* one, in terms of front level and crowding distance, is finally selected.

4.4 Genetic algorithm parametrization

Once the GA is defined and implemented, the execution needs to be parametrized. This parametrization defines the generation of the initial population, number of solutions—population size—the number of iterations to execute—number of generations—and the mutation probability.

We conducted previous testing to study the suitable values for each parameter. These parameters are generally established by the exploration of ranges of values and selecting the cases of the smallest sizes that obtain suitable optimized results [18].

The number of individuals in the population was varied from 25 to 300 in steps of 25. We observed that 200 solutions were enough to cover a wide range of solutions and that experiments with higher population sizes did not increase the solution space. The ending condition was initially fixed at 50 generations, and it was increased in steps of 50. We observed that after 200 generations, the values of the objectives were stabilized. We finally set the ending condition in 250 generations to have a certain flexibility. The mutation probability was determined by exploring a range of values between 0.05 and 0.5, in steps of 0.05, and observing that for values above 0.25 there were not significant improvements. To summarize, we finally established a population size of 200 solutions, a mutation probability of 0.25 and 250 generations.

The initial population was generated by considering one instance for each microservice randomly allocated in any of the VM instances and 4 virtual machines instances, with instance types and providers assigned randomly. None of stored container images were considered in the initial population.

4.5 Solution selection

The solution of a multi-objective optimization problem is a Pareto optimal front. The orchestrator needs to define some criteria, in terms of the user preferences, to choose one solution from this front. The selection of the solution needs to consider the importance of each objective for a particular case. For example, if the most important objective of the optimization is the monetary cost, the selection should be focused on the solutions in the Pareto front with the smallest values for this objective. This selection criteria can be implemented very easily by calculating a single objective

value by weighting the n-values of the objectives proportionally to their importance, and selecting the solution with the smallest resulting value [37].

In our case, we give the same importance to all the objectives. Thus, we choose the solution with the smallest value obtained from the uniform weighted sum of the three objectives. This transformation generates a scalar fitness value by calculating the unity-based normalized value of each objective value, as

$$\theta_x = \frac{x - x_{min}}{x_{\max} - x_{min}} \tag{10}$$

and adding them by applying a weight of

$$\omega = \frac{1}{num.\,objectives} \tag{11}$$

Thus, the objective function finally results in:

$$\sum_{i}^{num.\,objectives} \omega_i \times \theta_i \times i = \frac{1}{3} \times \theta_{C_{total}} \times C_{total} + \frac{1}{3} \times \theta_{L_{total}} \times L_{total} + \frac{1}{3} \times \theta_{T^{repair}} \times T^{repair} \tag{12}$$

5 Greedy algorithm proposal

Greedy algorithms base the heuristic process of solving a problem on finding local optimums. The main idea is that global optimum can be achieved by dividing the optimization problem in smaller problems. Thus, the decisions are taken in relation to local conditions or rules, without taking into account the global vision of the system [56].

We have defined a second approach based on the use of a greedy algorithm, the Greedy First-Fit algorithm, with the main objective of comparing it with our NSGA-II approach. Greedy algorithms have been widely applied in cloud management strategies [40,47]. In fact, there are several tools that base their management policies in greedy algorithms or round-robin techniques [34]. Thus, we consider that a greedy policy is a suitable case to compare with NSGA-II. Additionally, greedy algorithms are also commonly used as the control experiment in other research studies [20,44,48].

The definition of the greedy algorithm for our domain is presented in Algorithm 2. In a first step, one instance of each microservices is processed in arbitrary order and placed in the first VM with enough resource capacities (Algorithm 2, lines 10–12). If VM instances with enough free resources are not found, a new instance with the *best characteristics* is instantiated (lines 4–7). Once the allocation of a running instance for each microservice is defined, the number of microservices instances is increased iteratively until a scaling limit. Each new instance is randomly selected and randomly allocated or stored in the system following the same strategy than in the initial phase. (lines 15–20). The final solution is the one with the smallest fitness from all the iterated solutions.

Algorithm 2 Greedy First-Fit algorithm

```
1: procedure FINDALLOCATION(ms)
2:     for vm in |vmi^j_{n,i}| do
3:         if usage(vm) + resources(ms) < capacity(vm) then return vm
4:     if not allocated then
5:         vmTypePreference = weightedSumObjectiveEvaluation(ms, |vmt_{n,i}|)
6:         bestVmType = min(vmTypePreference)
7:         return newVMInstance(bestVmType)
8: procedure GREEDY
9:     solution = ∅
10:    for ms in shuffle(|ms_{x,y}|) do
11:        vm = findAllocation(ms)
12:        solution+ = allocation(ms, vm)
13:    scalingLimit = 1000
14:    incrementalSolution = solution
15:    for 1 to scalingLimit do
16:        ms = random(|ms_{x,y}|)
17:        vm = findAllocation(ms)
18:        incrementalSolution+ = random[allocation(ms, vm) or store(ms, vm)]
19:        if fitness(solution) > fitness(incrementalSolution) then
20:            solution = incrementalSolution
21:    return solution
```

6 Experimental evaluation

A reference configuration is defined for the experimental phase. This reference configuration is alternatively varied in one of its parameters to compare both optimization approaches (Greedy First-Fit and NSGA-II) under different conditions. Additionally, an evaluation of the evolution of the solutions obtained along the iterations of NSGA-II with the reference configuration is also conducted.

The reference configuration considers 3 cloud providers: The first one offers 2 VM types and each of the other two providers 1 VM type. Table 3 shows the characteristics of the VM instances. Cost is expressed in terms of generic cost units (c). The capacity is defined in terms of number of virtual cores that cloud providers commonly offer in their VM types (one, two, four, or eight cores) multiplied by 100 to be expressed as an utilization percentage, and labeled as %cpu. We consider that the higher the number of cores in the VM is, the cheaper the cost for a core is. This cost pattern is a common situation in real cloud providers. Thus, the cost is defined considering that a quad-core VM is cheaper than two dual-cores and a dual-core is cheaper than two single cores. We include a VM type that did not follow this cost pattern: The eight-core VM is more expensive than two quad-core, four dual-core, and eight single-core machines. This last case is also common in real cloud providers as they have more expensive prices for VMs with the highest resource dimensions.

The characterization parameters of the application have been obtained by measuring the behavior of a prototype: the *Socks Shop*.[1] *Socks Shop* is a demo application developed to test the benefits of container platforms [53]. The demo also includes a

[1] https://github.com/microservices-demo.

Table 3 Characteristics of the VM templates in the three providers

VM type	Provider $prov_n$	Capacity (%cpu) $R^{avail}_{n,i}$	Cost (c) $C_{n,i}$
$vmt_{1,1}$	1	100.0	100.0
$vmt_{1,2}$	1	200.0	150.0
$vmt_{2,3}$	2	400.0	250.0
$vmt_{3,4}$	3	800.0	1000.0

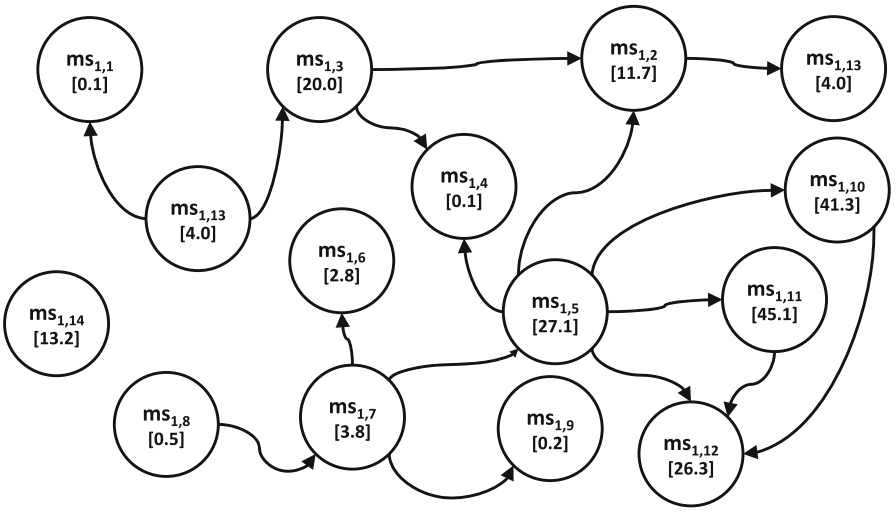

Fig. 4 Graph of the microservices stack for *Socks Shop* application. The nodes represent the microservices and the edges the consumption relationship $|consMS|_{x,y}$. The nodes are labeled with their identificator and the resource consumption $resources_{x,y}$ between square brackets, expressed as cpu utilization percentage (%cpu)

load test, which we used to measure the consumption of computational resources for each microservice by benchmarking a live demo.[2]

The application is formed by the composition of 14 microservices. Figure 4 shows the microservices (the nodes), their consumption relationships (the edges) and their CPU utilization percentages $R^{req}_{x,y}$ between square brackets, expressed as utilization percentage of a cpu core (%cpu). The times to start and download a container were also measured, and we observed that the download lasted 50 times more than starting the container. Consequently, $T^{start}_{x,y} = 1.0\ t$ and $T^{dwnld}_{x,y} = 50.0\ t$ were, respectively, established for the time to start and to download a container. The resource usages of stored container images were arbitrarily fixed in the 10% of consumption of the running instances. The network latency was arbitrarily fixed to $1.0\ t$ for VMs inside the same provider and $10.0\ t$ for VMs in different providers. All the times are expressed as generic time units (t).

[2] http://cloud.weave.works/demo.

We developed a simulator to execute the experiments. This simulator considers a synthetic input and generates a solution indicating the scale level of containers and VMs, the allocation of the containers in the VMs, and the type and cloud provider selection for each VM. The core of the simulator is the orchestrator, and our work includes its two implementations: one for the genetic algorithm and another one for the greedy algorithm. All these elements are developed with Python 2.7. This simulator corresponds to the pluggable orchestrator we explained in the motivational example in Sect. 1.1.

6.1 Solutions comparison

The reference configuration is varied in terms of VM instance costs, VM instance capacities and inter-providers latencies to study the suitability of our approach under several conditions. The variations over these parameters are performed by applying an increasingly multiplication factor from x2.0 until x5.0 in steps of 1.0.

Note that the solution of a multi-objective GA is a set of solutions called Pareto front. We have already explained the alternatives to select one solution from the Pareto front (Sect. 4.5). The comparison of the results is performed with the selected solution from the set. Figure 5 and Table 4 show the objective values of the selected solution for the NSGA-II (*GA*, blue/light bars) and for the Greedy First-Fit (*Greedy*, orange/dark bars). In the case of the figures, the values have been normalized between 0.0 and 1.0, and their weighted sums are also represented (labeled as *Weight.*). Each plot corresponds to an experiment where the values of the characterization parameters for the VM capacity (first column), the VM cost (second column), or the providers network latency overheads (third column) have been multiplied by the factor indicated in the plot: 2.0 (first row), 3.0 (second row), 4.0 (third row), or 5.0 (fourth row).

The weighted sum for our approach is always smaller than for the greedy solution. There are several cases where our solution shows a smaller value for the weighted sum, but higher for some single objective. Note that we are comparing with only one solution of the Pareto optimal front of our approach and, probably, the Pareto front includes other solutions with worse weighted value, but achieving a better optimization in the single objective. If we numerically measure the benefit of our approach in relation to the greedy algorithm as $\frac{greedy\ weighted\ sum}{GA\ weighted\ sum}$, the average for the 12 experiments is 4.09 (309% better). The lowest improvement is obtained in the experiment with the latency multiplied by $\times$ 3.0 that obtains a benefit of 1.68 (68% better) and the highest one in the case of multiplying the capacity by $\times$ 4.0, obtaining a benefit of 13.49 (1249% better).

Table 4 shows several cases, mainly for the greedy solution, where the optimized values of the repair time value are 0 t. Those cases correspond to solutions where all the containers have instances in at least two different providers.

6.2 Genetic algorithm evolution study

The solutions in a GA are obtained iteratively from the algorithm executions, i.e., the generations. Thus, it is also interesting to analyze the evolution of those solutions.

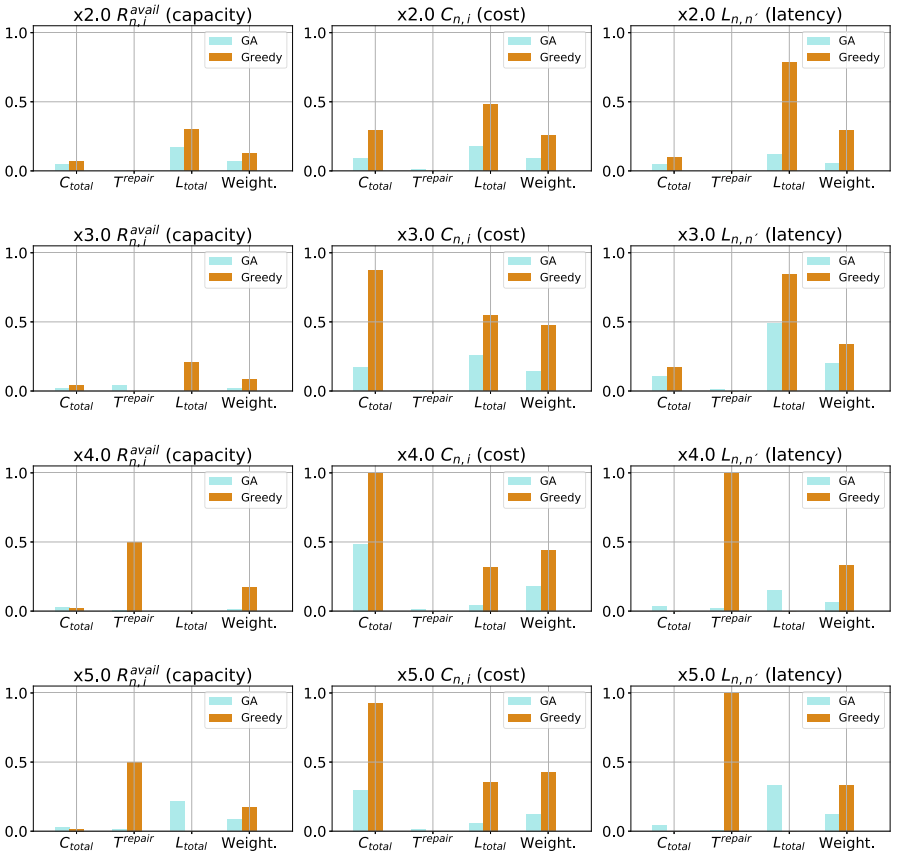

Fig. 5 Normalized objective values obtained with NSGA-II and Greedy First-Fit algorithms

Figure 6 presents the values of the selected solution—labeled as weighted and in blue/dark color—along the generations of the GA execution. Additionally, the values of the solution with the smallest value—labeled as min in red/light color and with dashed line—is also presented in the figure. Note that the values of the min series correspond to three different solutions, one for each objective. It is interesting to plot the min series since they are the lower bounds of the minimum values that can be obtained, i.e., the best possible optimized value.

Although we have used only one solution from the Pareto front in the previous section, we consider that it is important to reflect the evolution of all the Pareto front. This analysis helps us to study the coverage of the solution space and the number of generations that are needed to stabilize it. Consequently, the solutions in the Pareto front are presented in Fig. 7 for some of the generations.

The experiment obtains values of 30.0 t time units for the latency objective, 20.0 t time units for the repair, and 850.0 c cost units for the user cost. It is observed that the cost and latency are established approximately in generation 200. On the contrary, the repair time is established before generation 50. This is explained because the

Table 4 Objective values for the experiment comparison between NSGA-II and Greedy First-Fit algorithms

Parameter multipliers			Greedy			NSGA-II		
$C_{n,i}$	$R_{n,i}^{avail}$	$L_{n,n'}$	T^{repair} (t)	L_{total} (t)	C_{total} (c)	T^{repair} (t)	L_{total} (t)	C_{total} (c)
x1.0	x1.0	x1.0	1400	0	250	22	33	800
x2.0	x1.0	x1.0	0	65	4700	20	24	1600
x3.0	x1.0	x1.0	0	74	13,500	2	35	2850
x4.0	x1.0	x1.0	0	43	15,400	19	6	7600
x5.0	x1.0	x1.0	0	48	14,250	22	8	4750
x1.0	x2.0	x1.0	0	41	1300	0	23	950
x1.0	x3.0	x1.0	0	28	850	52	0	500
x1.0	x4.0	x1.0	700	0	500	12	0	700
x1.0	x5.0	x1.0	700	0	500	21	29	700
x1.0	x1.0	x2.0	0	106	1750	0	16	950
x1.0	x1.0	x3.0	0	114	2850	16	66	1850
x1.0	x1.0	x4.0	1400	0	250	24	20	750
x1.0	x1.0	x5.0	1400	0	250	4	45	850

repair time minimization is achieved easier, just by placing two instances of the same microservice in different cloud providers.

Finally, it is observed that the minimum values series (red/light lines) achieve the minimum values at the very beginning of the GA execution for the three objectives. This is explained because the minimization of one objective is much simpler and faster than the minimization of a multi-objective problem. In the experiment, they result in 8.0 t time units for the latency objective, 0.0 t time units for the repair, and 350.0 c cost units for the user cost.

Figure 7 presents the evolution of the solution search space, i.e., the objective values covered by the solutions in the Pareto front. Each point in the 3D scatter plot is the tuple with the three objective values of a solution in the Pareto front.

Initial populations have a higher concentration of solutions for a small number of values. As the first populations evolve, the solution search space is increased, covering a wider number of values, but the solutions are not still concentrated in the minimum values. Final generations show a higher number of solutions concentrated in the minimum objective values and a wider solution search space. One benefit of a GA is to have a wide range of optimized solutions to choose between them in terms of the user preferences.

7 Conclusion

We presented an optimization approach to reduce service cost, microservices repair time, and microservices network latency overhead in orchestration process of containers in a multi-cloud architecture. Our approach considered that the applications were

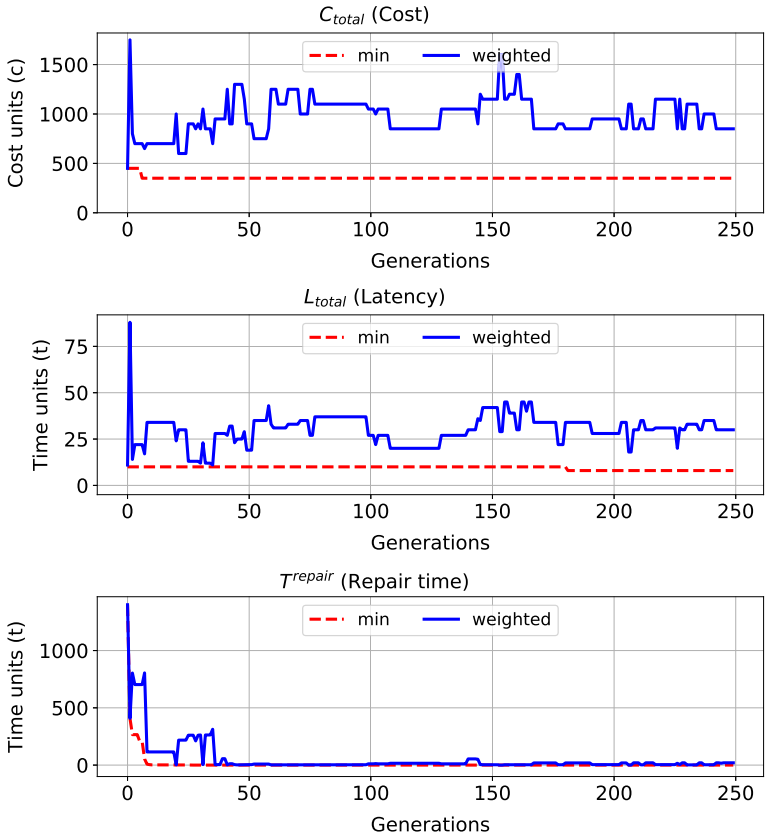

Fig. 6 Evolution of the three objective values for the solution with the smallest fitness weighted sum and for the three solutions with the smallest single objective values in each generation

formed by microservices, which were encapsulated in containers, and those containers were deployed in VMs that could be allocated in different cloud providers. The containers and the VMs were orchestrated by a central manager which was in charge of deciding the scaling level of both containers (number of container instances) and VMs, the allocation of the containers in the most suitable type of VM, and the allocation of the VM in the most suitable cloud provider. The management decision was addressed to reduce the three considered optimization objectives. We implemented the optimization with an evolutionary algorithm, NSGA-II, a common genetic algorithm for multi-objective problems. A two-point crossover and three mutation operators were considered. A Greedy First-Fit algorithm was also defined to compare our solution with.

Several experiments were conducted by modifying a reference configuration that considered 3 cloud providers, 4 VM templates, and one application with 14 microservices. The reference experiment was varied by changing the resource capacities of the VM templates, their cost, and the network latency among the cloud providers. Finally, 12 experiments were executed in a simulator that we developed using Python 2.7. The

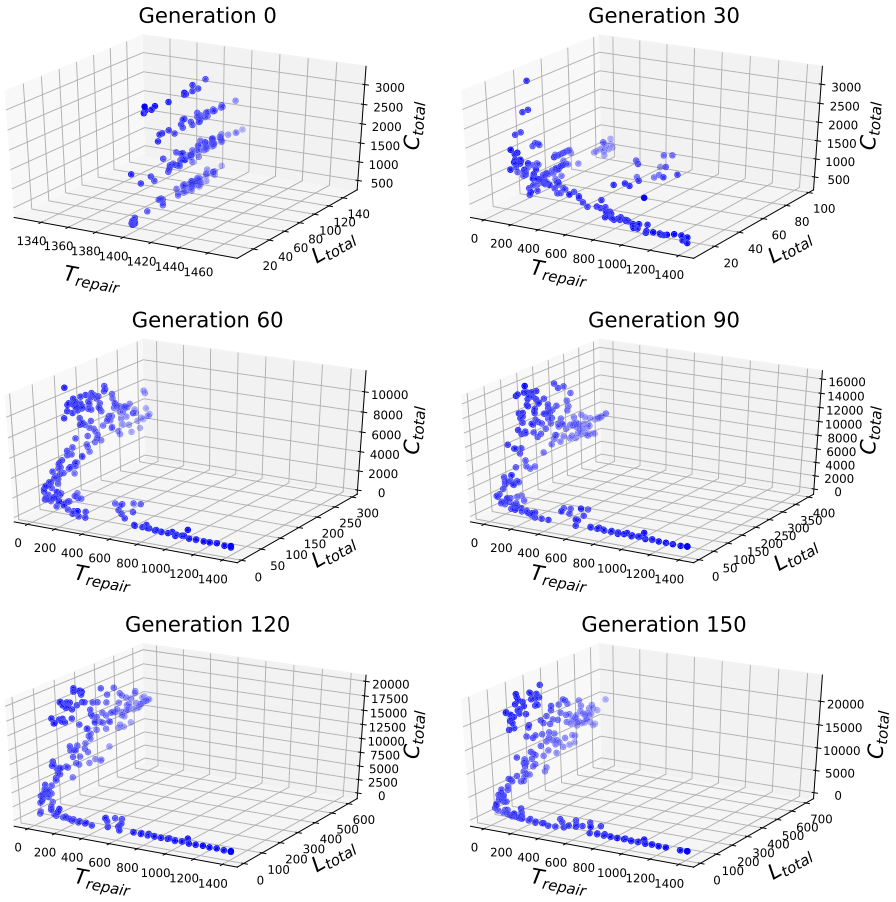

Fig. 7 Evolution of the solutions in the Pareto optimal front along six generations

simulator implemented the optimization process with both NSGA-II and the greedy algorithms, and their results were compared.

The experimental results showed that NSGA-II approach is suitable to obtain minimized objective values in just 200 generations with a population of 200 solutions. The analysis of the evolution of the Pareto front obtained with NSGA-II showed that the solution search space covered a widespread range of objective values, offering many alternatives to the orchestrator. Finally, a solution from the Pareto solution set of NSGA-II was selected by using a weighted-sum transformation, and it was compared with the results of the greedy algorithm. The solution of NSGA-II was always better than the solution of the greedy algorithm. Our approach obtained, on average, an improvement ratio of 4.09 (309% better) compared to the greedy results.

We concluded that our genetic approach was suitable to improve the orchestration of containers and VM type selection in multi-cloud environments and to self-scale the microservices and VM instances.

Three main future works emerged from the results of our paper. The first one is related with comparing the use of NSGA-II with other evolutionary algorithms, such as Firefly or Bat algorithms [57]. The second one is to study the applicability of a genetic optimization based on NSGA-II in other technologies that enable the use of multi-cloud, such as serverless computing [5]. Finally, the third research challenge is to study the adaptation of the optimization objectives of our approach to be useful for other emerging architectures that can also take profit of cloud and multi-cloud, such as fog computing [36].

Acknowledgements This research was supported by the Spanish Government (Agencia Estatal de Investigación) and the European Commission (Fondo Europeo de Desarrollo Regional) through Grant Number TIN2017-88547-P (AEI/FEDER, UE).

References

1. Adam O, Lee YC, Zomaya AY (2017) Stochastic resource provisioning for containerized multi-tier web services in clouds. IEEE Trans Parallel Distrib Syst 28(7):2060–2073. https://doi.org/10.1109/TPDS.2016.2639009
2. Adamuthe AC, Pandharpatte RM, Thampi GT (2013) Multiobjective virtual machine placement in cloud environment. In: 2013 International Conference on Cloud Ubiquitous Computing Emerging Technologies, pp 8–13 . https://doi.org/10.1109/CUBE.2013.12
3. Amato A, Martino B, Venticinque S (2013) Multi-objective genetic algorithm for multi-cloud brokering. In: Euro-Par 2013: Parallel Processing Workshops, pp 55–64
4. Balalaie A, Heydarnoori A, Jamshidi P (2016) Microservices architecture enables devops: migration to a cloud-native architecture. IEEE Softw 33(3):42–52. https://doi.org/10.1109/MS.2016.64
5. Baldini I, Castro P, Chang K, Cheng P, Fink S, Ishakian V, Mitchell N, Muthusamy V, Rabbah R, Slominski A, Suter P (2017) Serverless computing: current trends and open problems. In: Chaudhary S, Somani G, Buyya R (eds) Research Advances in Cloud Computing, Springer, Singapore, pp 1–20. https://doi.org/10.1007/978-981-10-5026-8_1
6. Bermejo B, Filiposka S, Juiz C, Gomez B, Guerrero C (2017) Improving the energy efficiency in cloud computing data centres through resource allocation techniques. In: Chaudhary S, Somani G, Buyya R (eds) Research Advances in Cloud Computing, Springer, Singapore, pp 211–236. https://doi.org/10.1007/978-981-10-5026-8_9
7. Bernstein D (2014) Containers and cloud: from lxc to docker to kubernetes. IEEE Cloud Comput 1(3):81–84. https://doi.org/10.1109/MCC.2014.51
8. Campbell M (2017) Distributed scheduler hell. USENIX Association, Singapore
9. Casalicchio E (2017) Autonomic orchestration of containers: problem definition and research challenges. In: 10th EAI International Conference on Performance Evaluation Methodologies and Tools. ACM. https://doi.org/10.4108/eai.25-10-2016.2266649
10. de Alfonso C, Calatrava A, Molt G (2017) Container-based virtual elastic clusters. J Syst Softw 127:1–11. https://doi.org/10.1016/j.jss.2017.01.007. http://www.sciencedirect.com/science/article/pii/S0164121217300146
11. de Paula LC, Soares AS, de Lima TW, Filho AR, Coelho CJ (2016) Variable selection for multivariate calibration in chemometrics: a real-world application with building blocks disruption problem. In: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion, GECCO '16 Companion. ACM, New York, NY, USA, pp 1031–1034. https://doi.org/10.1145/2908961.2931667
12. Deb K, Pratap A, Agarwal S, Meyarivan T (2002) A fast and elitist multiobjective genetic algorithm: Nsga-II. Trans Evol Comput 6(2):182–197. https://doi.org/10.1109/4235.996017
13. Fard H, Prodan R, Fahringer T (2013) A truthful dynamic workflow scheduling mechanism for commercial multicloud environments. IEEE Trans Parallel Distrib Syst 24(6):1203–1212. https://doi.org/10.1109/TPDS.2012.257
14. Fazio M, Celesti A, Ranjan R, Liu C, Chen L, Villari M (2016) Open issues in scheduling microservices in the cloud. IEEE Cloud Comput 3(5):81–88. https://doi.org/10.1109/MCC.2016.112

15. Frey S, Fittkau F, Hasselbring W (2013) Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13. IEEE Press, Piscataway, NJ, USA, pp 512–521. http://dl.acm.org/citation.cfm?id=2486788.2486856
16. Frincu ME, Craciun C (2011) Multi-objective meta-heuristics for scheduling applications with high availability requirements and cost constraints in multi-cloud environments. In: 2011 Fourth IEEE International Conference on Utility and Cloud Computing, pp 267–274
17. Fundation L (2018) Kubernetes—production-grade container orchestration. https://kubernetes.io/
18. Grefenstette JJ (1986) Optimization of control parameters for genetic algorithms. IEEE Trans Syst Man Cybern 16(1):122–128. https://doi.org/10.1109/TSMC.1986.289288
19. Grozev N, Buyya R (2014) Inter-cloud architectures and application brokering: taxonomy and survey. Softw Pract Exp 44(3):369–390. https://doi.org/10.1002/spe.2168
20. Guan X, Wan X, Choi BY, Song S, Zhu J (2016) Application oriented dynamic resource allocation for data centers using docker containers. IEEE Commun Lett 21(99):1. https://doi.org/10.1109/LCOMM.2016.2644658
21. Guerrero C, Lera I, Juiz C (2018) Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. J Grid Comput 16(1):113–135. https://doi.org/10.1007/s10723-017-9419-x
22. Guerrero C, Lera I, Juiz C (2018) Migration-aware genetic optimization for mapreduce scheduling and replica placement in hadoop. J Grid Comput. https://doi.org/10.1007/s10723-018-9432-8
23. Guzek M, Bouvry P, Talbi EG (2015) A survey of evolutionary computation for resource management of processing in cloud computing. IEEE Comput Intell Mag 10(2):53–67. https://doi.org/10.1109/MCI.2015.2405351
24. Harter T, Salmon B, Liu R, Arpaci-Dusseau AC, Arpaci-Dusseau RH (2016) Slacker: fast distribution with lazy docker containers. In: 14th USENIX Conference on File and Storage Technologies (FAST 16). USENIX Association, Santa Clara, CA, pp 181–195. https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter
25. Hashicorp: Nomad by hashicorp (2018). https://www.nomadproject.io/
26. Heilig L, Lalla-Ruiz E, Voß S (2016) A cloud brokerage approach for solving the resource management problem in multi-cloud environments. Comput Ind Eng 95(C):16–26. https://doi.org/10.1016/j.cie.2016.02.015
27. Hoenisch P, Weber I, Schulte S, Zhu L, Fekete A (2015) Four-fold auto-scaling on a contemporary deployment platform using docker containers. In: Service-Oriented Computing: 13th International Conference. ICSOC 2015, Goa, India, November 16–19, 2015, Proceedings. Springer, Berlin, pp 316–323
28. Kaewkasi C, Chuenmuneewong K (2017) Improvement of container scheduling for docker using ant colony optimization. In: 2017 9th International Conference on Knowledge and Smart Technology (KST), pp 254–259. https://doi.org/10.1109/KST.2017.7886112
29. Kang H, Le M, Tao S (2016) Container and microservice driven design for cloud infrastructure devops. In: 2016 IEEE International Conference on Cloud Engineering (IC2E), pp 202–211. https://doi.org/10.1109/IC2E.2016.26
30. Kimovski D, Saurabh N, Stankovski V, Prodan R (2016) Multi-objective middleware for distributed VMI repositories in federated cloud environment. Scalable Comput Pract Exp 17(4):299–312. http://www.scpe.org/index.php/scpe/article/view/1202
31. Legillon F, Melab N, Renard D, Talbi EG (2013) Cost minimization of service deployment in a multi-cloud environment. In: 2013 IEEE Congress on Evolutionary Computation, pp 2580–2587. https://doi.org/10.1109/CEC.2013.6557880
32. Li J, Li D, Ye Y, Lu X (2015) Efficient multi-tenant virtual machine allocation in cloud data centers. Tsinghua Sci Technol 20(1):81–89. https://doi.org/10.1109/TST.2015.7040517
33. Li P, Nie H, Xu H, Dong L (2017) A minimum-aware container live migration algorithm in the cloud environment. Int J Bus Data Commun Netw 13(2):15–27. https://doi.org/10.4018/ijbdcn.2017070102
34. Lin CC, Liu P, Wu JJ (2011) Energy-aware virtual machine dynamic provision and scheduling for cloud computing. In: 2011 IEEE 4th International Conference on Cloud Computing, pp 736–737. https://doi.org/10.1109/CLOUD.2011.94
35. Lucas-Simarro JL, Moreno-Vozmediano R, Montero RS, Llorente IM (2013) Scheduling strategies for optimal service deployment across multiple clouds. Futur Gener Comput Syst 29(6):1431–

1441 . https://doi.org/10.1016/j.future.2012.01.007. http://www.sciencedirect.com/science/article/pii/S0167739X12000192
36. Mahmud R, Kotagiri R, Buyya R (2018) Fog computing: a taxonomy, survey and future directions. In: Di Martino B, Li KC, Yang L, Esposito A (eds) Internet of everything. Internet of things (technology, communications and computing), Springer, Singapore, pp 103–130. https://doi.org/10.1007/978-981-10-5861-5_5
37. Marler RT, Arora JS (2010) The weighted sum method for multi-objective optimization: new insights. Struct Multidiscip Optim 41(6):853–862. https://doi.org/10.1007/s00158-009-0460-7
38. Merkel D (2014) Docker: lightweight linux containers for consistent development and deployment. Linux J 2014(239). http://dl.acm.org/citation.cfm?id=2600239.2600241
39. Mitchell M (1998) An introduction to genetic algorithms. MIT Press, Cambridge
40. Nardelli M, Hochreiner C, Schulte S (2017) Elastic provisioning of virtual machines for container deployment. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, ICPE '17 Companion. ACM, New York, NY, USA, pp 5–10. https://doi.org/10.1145/3053600.3053602
41. Newman S (2015) Building microservices: designing fine-grained systems. O'Reilly Media Inc., Sebastopol
42. Panda SK, Jana PK (2015) Efficient task scheduling algorithms for heterogeneous multi-cloud environment. J Supercomput 71(4):1505–1533. https://doi.org/10.1007/s11227-014-1376-6
43. Panda SK, Jana PK (2017) Sla-based task scheduling algorithms for heterogeneous multi-cloud environment. J Supercomput 73(6):2730–2762. https://doi.org/10.1007/s11227-016-1952-z
44. Pascual JA, Lorido-Botrán T, Miguel-Alonso J, Lozano JA (2015) Towards a greener cloud infrastructure management using optimized placement policies. J Grid Comput 13(3):375–389. https://doi.org/10.1007/s10723-014-9312-9
45. Peinl R, Holzschuher F, Pfitzer F (2016) Docker cluster management for the cloud—survey results and own solution. J Grid Comput 14(2):265–282. https://doi.org/10.1007/s10723-016-9366-y
46. Pietrabissa A, Priscoli FD, Giorgio AD, Giuseppi A, Panfili M, Suraci V (2017) An approximate dynamic programming approach to resource management in multi-cloud scenarios. Int J Control 90(3):492–503. https://doi.org/10.1080/00207179.2016.1185802
47. Piraghaj SF, Dastjerdi AV, Calheiros RN, Buyya R (2015) A framework and algorithm for energy efficient container consolidation in cloud data centers. In: IEEE International Conference on Data Science and Data Intensive Systems, DSDIS 2015, Sydney, Australia, December 11–13, 2015, pp 368–375. https://doi.org/10.1109/DSDIS.2015.67
48. Skarlat O, Nardelli M, Schulte S, Borkowski M, Leitner P (2017) Optimized iot service placement in the fog. Serv Oriented Comput Appl 11(4):427–443. https://doi.org/10.1007/s11761-017-0219-8
49. Sousa G, Rudametkin W, Duchien L (2016) Automated setup of multi-cloud environments for microservices applications. In: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), pp 327–334. https://doi.org/10.1109/CLOUD.2016.0051
50. Tao Y, Wang X, Xu X, Chen Y (2017) Dynamic resource allocation algorithm for container-based service computing. In: 2017 IEEE 13th International Symposium on Autonomous Decentralized System (ISADS), pp 61–67. https://doi.org/10.1109/ISADS.2017.20
51. Thnes J (2015) Microservices. IEEE Softw 32(1):116–116. https://doi.org/10.1109/MS.2015.11
52. Tordsson J, Montero RS, Moreno-Vozmediano R, Llorente IM (2012) Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers. Future Gener Comput Syst 28(2):358–367. https://doi.org/10.1016/j.future.2011.07.003. http://www.sciencedirect.com/science/article/pii/S0167739X11001373
53. Weaveworks Container Solutions (2018) Socks shop—a microservices demo application. https://microservices-demo.github.io/
54. Wei G, Vasilakos AV, Zheng Y, Xiong N (2010) A game-theoretic method of fair resource allocation for cloud computing services. J Supercomput 54(2):252–269. https://doi.org/10.1007/s11227-009-0318-1
55. Wolpert DH, Macready WG (1997) No free lunch theorems for optimization. Trans Evol Comput 1(1):67–82. https://doi.org/10.1109/4235.585893
56. Wu X, Gu Y, Tao J, Li G, Jayaraman PP, Sun D, Ranjan R, Zomaya A, Han J (2016) An online greedy allocation of vms with non-increasing reservations in clouds. J Supercomput 72(2):371–390. https://doi.org/10.1007/s11227-015-1567-9

57. Yang X (2010) Firefly algorithm, stochastic test functions and design optimisation. Int J Bio Inspired Comput 2(2):78–84. https://doi.org/10.1504/IJBIC.2010.032124
58. Zhan ZH, Liu XF, Gong YJ, Zhang J, Chung HSH, Li Y (2015) Cloud computing resource scheduling and a survey of its evolutionary approaches. ACM Comput Surv 47(4):63:1–63:33. https://doi.org/10.1145/2788397
59. Ziafat H, Babamir SM (2017) A method for the optimum selection of datacenters in geographically distributed clouds. J Supercomput 73(9):4042–4081. https://doi.org/10.1007/s11227-017-1999-5