A multiobjective metaheuristic-based container consolidation model for cloud application performance improvement

Vincent Bracke [1*], José Santos [1], Tim Wauters [1], Filip De Turck [1], Bruno Volckaert [1]

[1]IDLab, Department of Information Technology, Ghent University - imec, Technologiepark-Zwijnaarde 126, Ghent, B-9052, Belgium.

*Corresponding author(s). E-mail(s): vb.ugent@gmail.com;

Abstract

This work describes an approach to enhance container orchestration platforms with an autonomous and dynamic rescheduling system that aims at improving application service time by co-locating highly interdependent containers for network delay reduction. Unreasonable container consolidation may however lead to host CPU saturation, in turn impairing the service time. The multiobjective approach proposed in this work aims to improve application service-time by minimizing both inter-server network traffic and CPU throttling on overloaded servers. To this extent, the Simulated Annealing combinatorial optimization heuristic is used and compared on its relative performance towards the optimal solution obtained by Mathematical Programming. Additionally, the impact of the proposed system is validated on a Kubernetes cluster hosting three concurrent applications, and this under varying load scenarios. The proposed rescheduling system systematically i) improves the application service-time (up to 27.2% from our experiments) and ii) surpasses the improvement reached by the Kubernetes descheduler.

Keywords: Cloud Computing Services, Kubernetes, Container Scheduling, Mathematical Optimization, Autonomic and Cognitive Management, Performance Management

1 Introduction

Microservices software architecture promotes application development as a set of distributed small and independent services, each one delivering a specific set of functionalities. Appropriate service decomposition leverages on loose coupling for the inter-relationships while ensuring high cohesion of purpose. When combined with *containerization* technologies for their deployment and execution, microservices oriented applications offer unprecedented agility at both design and run time. da Silva et al. [1] define containers as a technology that provides OS-level virtualization to isolate processes and specifies system usage limits for resources such as CPU, RAM, disk I/O and network. It acts as an abstraction at the application layer that packages code and dependencies together. Multiple containers can thus simultaneously run on the same machine and share the Operating System (OS) kernel with other containers, each running as isolated processes in user space. Docker[2] is one of the most commonly used container engines but alternatives exist such as LXC/LXD[3], Podman[4], containerd[5], etc. Container orchestration platforms (e.g. Apache Mesos [6], Docker Swarm [7], Kubernetes (K8s) [8]) have been developed to help manage containerized applications running on distributed clusters. These platforms offer scalability, availability management, monitoring tools, networking functionalities, container orchestration, etc. for the containerized application. Nevertheless, despite their uptake, those container orchestration platforms still suffer from some limitations as they lack runtime adaptability which restrains cluster wide optimisation of resource allocation and consequently affects overall cluster performance. Indeed, based on their resource requests and affinity constraints, containers are scheduled to the best fitting node(s) at deployment time. However, the scheduler does not adapt the distribution of containers amongst servers at runtime based on the observed container behavior and interdependencies. Yet, the cluster state is prone to evolve over time: application workload evolves, new containers are deployed, others are removed, etc. possibly degrading the then best assignment. An interesting vision in this regard are "self-driving" systems which measure, analyze and control themselves in an automated manner, reacting to changes in the environment (e.g., demand), while exploiting existing flexibilities to optimize themselves [9].

We therefore proposed in [10] an autonomous rescheduler that periodically reassesses the distribution of containers among servers in order to optimize application service time by consolidating interdependent containers. Indeed, co-locating inter-communicating containers on the same server substantially reduces network traffic and consequently improves overall application service time. However, consolidating to the extreme may lead to server saturation and consequently disserve the initial intent of the rescheduling; therefore in addition to minimization of inter-server network traffic, minimization of CPU overload is also desirable.

This article presents a self-driving rescheduling system capable of reallocating containers within a distributed cluster to improve overall application service time by minimizing both inter-server network traffic and CPU overload while still taking

other constraints into consideration (e.g. affinity rules, resource requirements).

The remainder of the article is organized as follows. Section 2 highlights the motivation of this work. Section 3 summarizes the current state of the art in the domain, and section 4 presents the multiobjective optimisation model together with an analysis of the implemented metaheuristic's effectiveness and efficiency. Based on this, a concrete implementation of the container rescheduling system by means of a control-loop architecture is proposed in section 5 and then validated by means of three complementary applications undergoing varying workload levels. Section 6 identifies and discusses improvement and extension opportunities and, finally, Section 7 provides concluding remarks.

2 Motivation based on experimental study

A single-objective approach for the rescheduling of containers was proposed in [10], solely focusing on inter-server network traffic minimization. While allowing for significant improvement of application service time, we identified the need to extend the model in order to cope with extreme consolidation cases where most of the workload would be concentrated on a select few servers of the cluster. In such a situation, the rescheduling would ultimately disserve the initial goal as the saturated servers would not be able to respond as efficiently as when they were less heavily loaded.

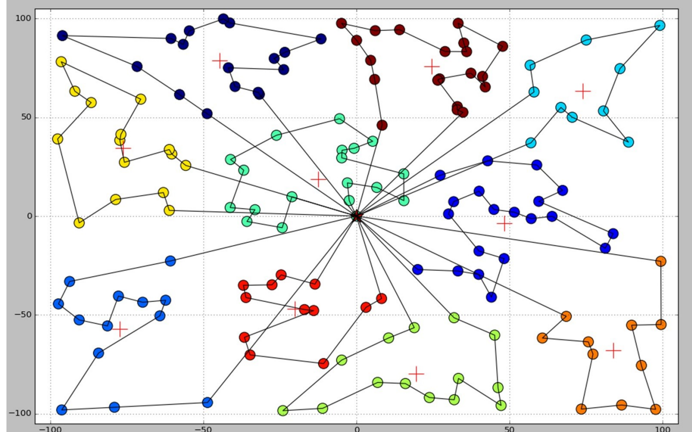

Fig. 1 The optimized route plan for 10 vans having to distribute 150 packets as output of the VRP application

We propose to verify the validity of this intuition by means of an abridged implementation of the Vehicle Routing Problem (VRP)[1], first introduced by Dantzig and Ramser [12], where 'V' vans, initially located at a central depot, have to visit 'L' different locations. The goal is to minimize the distance for each van while ensuring fair

[1]The VRP generalizes the Traveling Salesman Problem (TSP)[11]

workload distribution among vans. Last but not least, each location must be visited. The application receives as input the list of coordinates for all locations as well as the distance between all locations as a square matrix. It returns the optimized route plan for each van as illustrated in Figure 1.

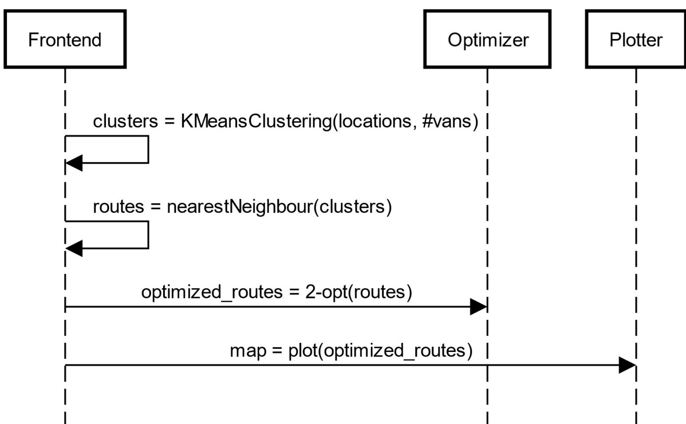

Fig. 2 The sequence diagram of the VRP application

The application is composed of three microservices which interact as illustrated by the sequence diagram in Figure 2 hereafter further described :

- The **Frontend** microservice is called by the end-user to compute the best itinerary for each van. It starts by partitioning the 'L' locations into 'V' clusters via the KMeans clustering technique (first introduced by Lloyd [13]). Then, for each cluster, it computes the itinerary by means of the Nearest Neighbour algorithm. However, due to its greedy nature, this algorithm may miss shorter routes. Therefore, the Frontend microservice calls the Optimizer microservice to improve, where possible, the generated routes.
- The **Optimizer** microservice implements the 2-Opt technique, introduced by Croes [14], which is a local search algorithm that allows for improving initial route solution[2].
- The **Plotter** microservice is subsequentally called by the Frontend microservice with the optimized routes to plot. It returns the visual representation of the solution (as illustrated in Figure 1).

The specifications of the test environments, used for the various experiments reported in this article, are described in table 2.

The experiment[3] emulates 300 clients, each executing a sequence of 5 consecutive calls to the Frontend microservice for '10 vans - 150 locations' VRP resolution. For baseline purposes, an initial run is executed with all microservices hosted on the same node and a second run with each microservice hosted on distinct nodes. Main outcome

[2]More advanced alternatives are presented in [15]
[3]inspired by [16]

Table 1 Hardware and Software description of the test environment used in this work

Server Specifications		Section
OS	Ubuntu 20.04 LTS	2, 4 & 5
CPU	2 Quad core Intel® E5520 (2.2GHz) 2 Hexa core Intel® E5645 (2.4GHz)	4 2 & 5
RAM	12GB 24GB	4 2 & 5
HDD	160GB 250GB	4 2 & 5
NIC	1Gb Intel® 82576	2, 4 & 5
Environment		**Section**
# servers	1 + 3 1 1 + 6	2 4 5
K8s	v1.25.5	2 & 5
Applications		**Section**
MP solver	IBM® ILOG® CPLEX® v22.1.0	4
Rescheduler	JDK 11.0.18 Vertx.x™ 4.4.4	4 & 5
VRP app	Python 3.8.2	2 & 5
Obelisk app	JDK 11.0.18 Vertx.x™ 4.4.4	5
Google Online Boutique app	Golang 1.21.1 C# SDK 8.0.100 Node.js 20.7.0 Python 3.10.8	5

from those two runs is illustrated in Figure 3 where an average response time of 20.45 seconds and a total experiment time of 94 seconds are observed when the microservices run on different nodes. When the microservices are all hosted on the same node those metrics approximately double with an average response time of 37.64 seconds and a total experiment time of 195 seconds. Additionally the predictability decreases with higher variations of the response time in the latter run (the shaded area around the line covers 95% of the observations). One could however intuitively consider that those better results are obtained at the cost of more resource usage as the first scenario only implies 1 server instead of 3 and consequently conclude that in absence of linearity it is not worth the money.

A further analysis of the CPU consumption however indicates a substantially different outcome as illustrated by Figure 4 and Table 2 where the single server scenario consumes 924 more CPU seconds than the scenario with 3 distinct hosting servers ($\approx$ 25% more). This results from the level of CPU load: once a server's CPU is fully used by one process, the other processes start to queue and wait to access it, which reduces their throughput. The average level of CPU load for the last minute is obtained by

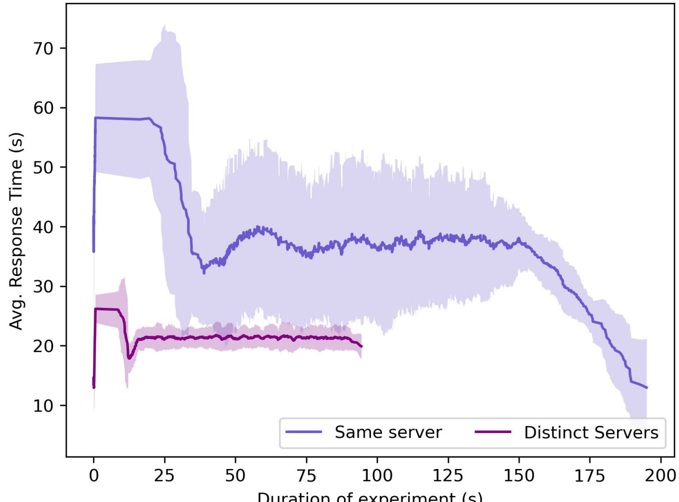

Fig. 3 Comparison of the response time and overall experiment execution time of the VRP application with the microservices hosted on the same server (in blue) and on 3 distinct servers (in purple)

dividing the 'load1' linux metric by the number of available cores. In the single server scenario (Figure 4 (a)) the CPU is directly overloaded[4] and remains so until the end of the experiment (with a load average of approximately 160%, which means that on average 0.6 processes are blocked and waiting), whereas on the distributed scenario (Figure 4 (b)), only node2 (hosting the Optimizer) gets (directly) overloaded and remains so until the end of the experiment (with the peak at approximately 160% at the end of the experiment) while the 2 other nodes are never overloaded: the Frontend (on node1) and the Plotter (on node3) are not limited in their access to CPU.

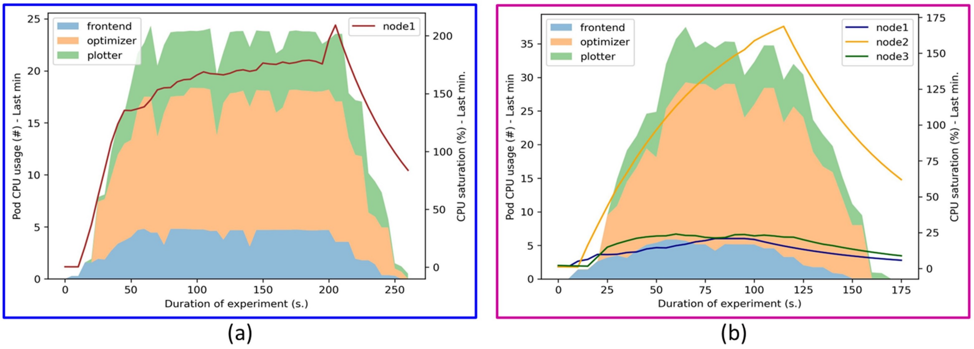

Fig. 4 Comparison of the CPU consumption and load of the VRP application with the microservices hosted on the same server (a) and on 3 distinct servers (b)

[4] as 'load1' reports the average load for the last minute

Interestingly though, the average response time (see Figure 3) of the consolidated scenario starts to decrease for the last 40 seconds of the experiment and gets even lower than the average response time of the distributed scenario. With the clients progressively ending their calls, the load proportionally decreases with a break-even point (at 185 secs. approximately) where the effect of the last requests on the CPU load gets compensated by the absence of network traffic between nodes.

Table 2 Comparison of the number of CPU seconds used by the microservices when hosted on 1 or 3 servers

CPU sec. (Area)	**1 Server**	**3 Servers**
Frontend	885	530
Optimizer	2536	2373
Plotter	1137	731
Total	**4558**	**3634**

This example emphasizes the need to limit containers consolidation by seeking to balance CPU load among nodes to minimize overloading. This does not abrogate the single-objective model introduced in [10] but rather highlights the need to let it evolve to a multiobjective model with the combined effect of i) consolidating network interrelated containers on the same node and ii) evicting non network interrelated containers to other nodes.

3 Related Work

While the literature on resource scheduling for the data center initially focused on Virtual Machine Consolidation (VMC) mostly to optimize the performance-energy tradeoff [17–19], it progressively evolved, with the rise of 'containerization' and microservice architectures, to address the issue of scheduling for the containerized application considering various factors such as load balance & application performance tradeoff [20], heterogeneity of resources [21], classical bin-packing [22], network Quality-of-Service (QoS) [23] and network latency introduced by inter-microservice communication [24]. These works do however only consider the initial allocation of containers and do not consider their rescheduling.

The authors of [25] report that online schedulers are much less common in the literature than offline schedulers, making the development of effective online scheduling challenging. Furthermore, the same authors also state that imprecision of input data (e.g. execution time and resource needs) represents another challenge as it negatively impacts the scheduling performance. In their systematic literature review on challenges and solution directions for microservice architectures (MSAs), Söylemez et al. [26] identify service orchestration as one of the nine main categories of challenges. More specifically, the authors state that the challenges for service orchestration relate among other to dynamic and automated orchestration and scheduling, stating that it is a challenging issue to perform the necessary adjustments according to the usage

of resources over time. In addition, the authors report that reducing total traffic cost and delay are important criteria for scheduling as misguided scheduling directly affects the availability and reliability of the system.

Santos et al. [27] propose a network-aware framework for the popular K8s platform, named Diktyo, that determines the placement of dependent microservices by focusing on the application's end-to-end latency reduction. This framework (now partially open-sourced at the K8s sig-scheduling community[5]) however diverges from our work as i) it focuses on initial network-aware container placement, ii) it requires the specification by the application developer of minimum traffic demands and maximum network costs between dependent containers, and iii) it needs the specification of the infrastructure topology as a Custom Resource (CR) in K8s. The same divergences are observed for the comparable (but only evaluated by means of simulation) approach proposed in [28], that aims at optimizing container placement based on resource prices, while taking inter-container traffic into consideration.

Piraghaj et al. [29] propose a framework for energy efficient container rescheduling in cloud data centers, evaluated by means of simulation only. However, the only perspective of optimizing the energy consumption, while being relevant for data center owners, expels other important aspects like the quality of service and user-experience as the proposed system does not have any knowledge of the applications running inside the containers.

Rattihalli [30] propose a two stages approach where containers are first instantiated in a so-called 'little cluster' for profiling before being instantiated on the so-called 'big cluster'. This approach assumes overestimated resource requirements that can be fine-tuned during the profiling stage before final scheduling, which represent an overhead for containers with appropriately defined requirements. Another drawback of this approach resides in the assumption of stable load over time. A solution to this latter drawback is proposed in [31], where the authors propose a self-adaptative K8s cloud controller that continuously updates an internal performance model of each service and uses it to determine the kind of resources needed by a service, as well as to predict potential contention on shared resources, and (re-)deploys services accordingly. However, it still requires an initial profiling stage, the assessment phase, in a dedicated environment.

The container rescheduling framework introduced by Rodriguez and Buyya [32] does not actively monitor resource consumption to initiate rescheduling, but rather only reacts upon appearance of unschedulable containers in the pending queue by evicting moveable containers from their server if i) the moveable containers can be rescheduled on another server and ii) by evicting the moveable containers, the server has enough resources to host an unschedulable container. In contrast to this reactive approach, our rescheduling system proactively monitors resource consumption to periodically improve container assignment. In [33], the authors propose NetMARKS,

[5] https://github.com/kubernetes-sigs/scheduler-plugins/blob/master/pkg/networkaware/README.md

a K8s scheduler extender that uses information collected by Istio Service Mesh to schedule pods based on current network metrics in order to save inter-node network bandwidth and to reduce the application response delay. However, NetMARKS minimizes inter-node traffic by considering applications individually one at a time, possibly leading to sub-optimal overall cluster status. A comparable approach is also proposed in [34].

Lastly, Joseph and Chandrasekaran [35] propose a microservice rescheduling framework, Throttling and Interaction-aware Anticorrelated Rescheduling for Microservices (TIARM), to proactively perform rescheduling activities while ensuring timely service responses. The framework incorporates a component that performs periodic monitoring and triggers rescheduling activities based on threshold-based rules to reduce microservice response time. The rescheduling phase first selects the containers for migration based on a multicriteria decision making method and terminates them. The containers are then redeployed onto nodes selected by a multiobjective strategy. While sharing the objective and exhibiting technical similarities with our work, both approaches diverge on the fundamental aspect of container and node selection strategy. More specifically:

- **Container selection**: TIARM only reschedules containers running on overloaded servers. Containers to be evicted are identified using a weighted linear combination of the CPU throttling level and the interaction factor: the proposed system prefers to move containers with the least interactions with other containers on the current node[6]. Besides the fact that the threshold used to assess server overloading is statically defined and consequently prone to inefficiency, TIARM only reschedules containers when this limit is reached on one or more servers, potentially missing opportunities for smaller intermediary adjustments that could bring the cluster state closer to the optimum.
- **Server selection**: the server selection module of TIARM seeks to maximize the anticorrelation between the microservice container and the server resource vectors. The underlying rationale justifying this approach can be summarized as follows: the performance of workloads often depends on other workloads running on the same server. Workloads with positively correlated resource utilization (e.g. all heavily CPU-bound) running on the same server offer more risks of overutilization. Coupling microservice containers with complementary resource demands can improve the resource utilization of the server and thus improve QoS values. In contrast, the system proposed in this article considers server resources as constraints rather than as part of the optimisation objective.

Lastly, the resizing module of TIARM adapts the deployment configuration by increasing the 'CPU limit' for throttled containers; while technically ensuring a lower throttling rate, the choice for this technique seems questionable as it leads to unsupervised higher resource consumption.

[6] The intuition is that highly interacting containers spend more time in communication when placed across different nodes, thereby leading to a degradation in the observed response times.

4 The dynamic rescheduling algorithm

4.1 Problem formulation and modelization

The combinatorial optimization problem of dynamically rescheduling containers to servers is an application of the multiobjective Quadratic Assignment Problem (QAP) where a set of 'p' containers need to be optimally assigned to 'n' servers. The problem at stake in this work is to assign all containers to servers with the goal of i) minimizing the sum of the inter-server network traffic and ii) minimizing server overload. Intuitively, the model encourages consolidation of containers with high data flows among them while discouraging over-consolidation. For a given cluster of 'N' servers, the cluster network traffic cost function ('cntc'), already introduced and discussed in [10], is obtained by Eq.(1) where r_n^{pq} equals 0 if containers 'p' and 'q' are both hosted on server 'n' or if none of them is, else 1 and t^{pq} is the amount (in bytes) of network traffic exchanged between containers 'p' and 'q'. The function thus returns the sum of bytes exchanged among the cluster servers.

$$cntc = \sum_{n=1}^{N} r_n^{pq} t^{pq}, \forall p, q \neq p \tag{1}$$

Regarding the server overload cost function, some more introductory details are hereafter provided. As introduced in section 2, the 'load1' Linux metric provides the run-queue length: the number of processes that are running plus the number that are waiting (queued) to run. Dividing this number by the number of cores provides the averaged server load (for the last minute). Consequently the CPU saturation point ('sp') is achieved at a CPU load of 1.0 (100%)[7]. We define the overload as the load exceeding the 'sp'.

For a given cluster of 'N' servers, the average server overload ('aso') is obtained by Eq.(2) where l_n represents the load of server 'n'.

$$aso = \frac{\sum_{n=1}^{N} max(0, l_n - sp)}{N} \tag{2}$$

The cluster's overload cost ('coc') is obtained by Eq.(3).

$$coc = \sum_{n=1}^{N} max(0, l_n - (sp + aso)) \tag{3}$$

Minimizing this value ensures :

- a cost of zero for not overloaded clusters.
- that the most overloaded server cannot be less overloaded: this results in balancing overload among servers instead of (possibly) topping all cluster overload on one

[7]Sysadmins usually define a more conservative value (typically around 0.7) to provide some headroom to the system.

single server (in the worst case). The underlying rationale for this choice comes from the non linearity of the relation between CPU utilization and application performance [36].

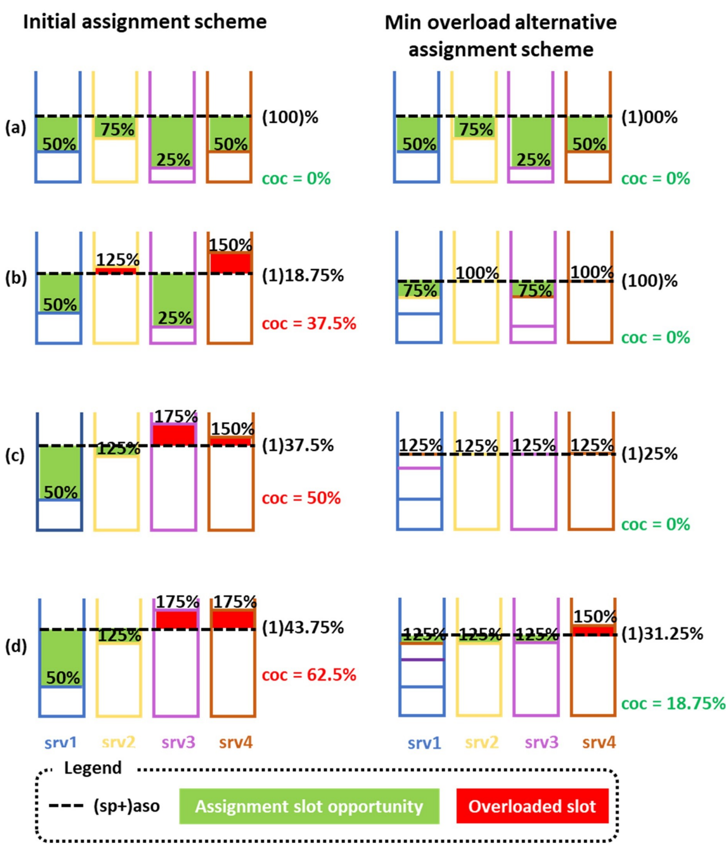

Fig. 5 Distinct assignment schemes illustrating the 'coc' calculation and minimization

To exemplify what has just been stated, let us assume a cluster of four servers, a 'sp' value of 100%, and a fixed CPU load of 25% per hosted container. Figure 5 illustrates, by means of four distinct assignment schemes, how the value of the 'coc' is computed as well as how it can possibly be minimized:

- case a): there is no load over the 'sp+aso' threshold in the initial assignment scheme, resulting in a 'coc' value of zero. As this cost cannot be further minimized and to ease the understanding of the illustration, the same scheme is reused in the second column. However, any other assignment scheme with a 'coc' value of zero (e.g. all servers with a load of 50%) is as legitimate.

- case b): servers 2 and 4 exhibit a load above the 'sp+aso' threshold ($100 + \frac{0+25+0+50}{4} = 118.75$), resulting in a 'coc' value of $0 + (125 - 118.75) + 0 + (150 - 118.75) = 37.5$. Minimizing this cost comes to assign the overloading slots (red boxes) to the absorption slots (green boxes), which is illustrated by the minimum overload assignment scheme of the second column. Concretely, one of the containers of srv2 is now assigned to srv1 and two containers of srv4 are now assigned to srv3 bringing both the 'aso' and 'coc' value to zero. Here again alternative assignment schemes with a 'coc' value of zero are all as legitimate (e.g. srv1=100%, srv2=100%, srv3=50% and srv4=100%).
- case c): this case diverges from previous one only for srv3 which now exhibits a load of 175% instead of 25% for the initial assignment scheme leading to higher 'aso' and 'coc' values. Reaching a 'coc' value of zero remains however possible when all servers exhibit a load of 125%. This case further highlights the dynamics behind the 'coc' minimization. Indeed, the proposed minimum overload alternative reaches a 'coc' value of zero while the 'aso' value is of 25%. This comes from the intrinsically immutable nature of the average server load ('asl') of 125% $= \frac{50+125+175+150}{4} = \frac{125+125+125+125}{4}$; while overload can possibly be reduced, load cannot be. Therefore a 'coc' value of zero should not be interpreted as the absence of load exceeding the 'sp' threshold but rather the absence of load exceeding the 'sp+aso' threshold.
- case d): further expands case c by adding 25% of load to srv4. In this case a perfect load balance is not achievable anymore, resulting in a 'coc' value greater than zero. More specifically, given a certain cluster load, minimizing the 'coc' value certifies that no other assignment scheme allows for a lower maximum server load when this load exceeds the 'sp+aso' threshold.

Minimizing both the 'cntc' (Eq.(1)) and the 'coc' (Eq.(3)) objective functions is our multiobjective model's goal. Table 3 introduces the variables and parameters of the model. Most of those are self-explanatory, though the three last parameters still require introductory explanation as the model is constrained by:

1. server (anti-) affinity: for containers that must (not) run on a server member of a specific subset. This feature is modeled through the a_n^p parameter. There are three possibilities:
   (a) No server (anti-) affinity is defined for container p: this parameter equals 1 for all schedulable servers, else 0. A server could indeed be (temporarily) unschedulable: this may be the case for instance for (temporarily) unavailable servers or for servers dedicated to the management of the cluster and not to container hosting.
   (b) Server affinity is defined : in this case, this parameter equals 1 for all schedulable servers matching the defined (set of) affinity(ies), else 0.
   (c) Server anti-affinity is defined : in this case, this parameter equals 1 for all schedulable servers not matching the defined (set of) anti-affinity(ies), else 0.
2. inter-container affinity: for containers that must run together on the same server. This feature is modeled through the parameter f^{pq}. If such an affinity is defined for 2 containers then the parameter equals 1, else 0.

3. inter-container anti-affinity: for containers that must not run on the same server. This feature is modeled through the parameter v^{pq}. If such an anti-affinity is defined for 2 containers, then the parameter equals 1, else 0.

Table 3 Problem modelization

Variable	Description
n,m	server id, from 1 to N, where N is the total amount of servers
p,q	container id, from 1 to P, where P is the total amount of containers
s_n^p	= 1 if container p is scheduled on server n, 0 otherwise
r_n^{pq}	= 0 if containers p & q are hosted on server n or if none of them is, else 1
Parameter	**Description**
w_n	CPU capacity (units) for server n
x^p	CPU (units) required by container p
y_n	RAM capacity (bytes) for server n
z^p	RAM (bytes) required by container p
sp	the defined CPU saturation point (%)
l^p	the CPU load contribution (%) of container p
t^{pq}	network traffic (bytes) between containers p and q
a_n^p	=1 if container p may be assigned to server n, else 0
f^{pq}	=1 if container p and q must be co-located, else 0 (inter-container affinity)
v^{pq}	=1 if container p and q must not be co-located, else 0 (inter-container aversion)

All in all, the model is subject to the following constraints:

- Each container must be instantiated on one and only one server:

$$\sum_{n=1}^{N} s_n^p = 1, \forall p \tag{4}$$

- Each server must be able to provide the CPU capacity required for each hosted container:

$$\sum_{p=1}^{P} s_n^p x^p \leq w_n, \forall n \tag{5}$$

- Each server must be able to provide the RAM capacity required for each hosted container:

$$\sum_{p=1}^{P} s_n^p z^p \leq y_n, \forall n \tag{6}$$

- Container assignment cannot violate server (anti-) affinity constraints:

$$\sum_{n=1}^{N} s_n^p a_n^p = 1, \forall p \tag{7}$$

- Two containers must be co-located if defined by an inter-container affinity constraint:

$$1 - f^{pq} \geq s_n^p - s_n^q, \forall n, p, q \neq p \tag{8}$$

- Two containers must not be co-located if defined by an inter-container anti-affinity constraint:

$$2 - v^{pq} \geq s_n^p + s_n^q, \forall n, p, q \neq p \tag{9}$$

- Eq.(10) and Eq.(11) ensure that inter-container traffic is taken into account when containers p and q are not co-located. Those two equations permit the linearization of $r_n^{pq} = |s_n^p - s_n^q|$:

$$r_n^{pq} \geq s_n^p - s_n^q, \forall n, p, q \neq p \tag{10}$$

$$r_n^{pq} \geq s_n^q - s_n^p, \forall n, p, q \neq p \tag{11}$$

- Lastly, variables r_n^p and s_n^{pq} are defined as binary variables:

$$s_n^p, r_n^{pq} \in \{0, 1\} \tag{12}$$

4.2 Problem solving

The most common approach to solve a multiobjective optimization problem is to combine its multiple objectives functions into one single-objective scalar function[37]. This approach is known as the weighted-sum method. In more detail, the weighted-sum method minimizes a positively weighted sum of the objectives that represents a new optimization problem with a unique objective function [38]:

$$\min \sum_{i=1}^{I} \omega_i \cdot \eta_i \cdot f_i(x) \tag{13}$$

where :

$$\sum_{i=1}^{I} \omega_i = 1$$

$$\omega_i > 0, \forall i$$

$$\eta_i = \frac{1}{max_i - min_i}, \forall i$$

The ω_i parameter represents the weight assigned to function 'i' while the η_i parameter represents its normalization factor (used to annihilate the differences in magnitude among the functions and obtain a common scale). This normalization function has been chosen as it provides the best possible normalization results by normalizing the objective functions by the true intervals of their variation over the Pareto optimal set [39]. Transposing Eq.(13) to the container rescheduling problem as modeled in previous sub-section, gives :

$$\min(\omega_{cntc} \cdot \eta_{ctnc} \cdot cntc + \omega_{coc} \cdot \eta_{coc} \cdot coc) \tag{14}$$

Main advantages of the weighted-sum method are its relative ease of implementation and interpretation as well as the guarantee to obtain a strict Pareto Optimal solution. However, there are two shortcomings to this approach [38, 40]:

- While the relative value of the weights generally reflects the importance of the objectives, it is not always possible to define weight vectors to reach a specific portion of the Pareto curve: it is often observed that the solutions are grouped in certain parts of the Pareto front, while some (possibly significant) portions are not reached.
- Non-convex parts of the Pareto set cannot be reached by minimizing convex combinations of the objective functions.

Container rescheduling being a NP-hard problem [35], the use of a metaheuristic to solve it allows to reach near optimum solution in a reasonable time. To this extent, we expanded the Simulated Annealing (SA) metaheuristic implementation introduced in [10] with the 'coc' function and scalarized both objective functions[8] as prescribed by the weighted-sum method. SA[9] is a probabilistic method proposed by Kirkpatrick et al. [41] and Cerny [42] for finding the global minimum of a cost function that may possess several local minima. This trajectory-based metaheuristic has successfully been applied to a wild variety of combinatorial optimization problems [43, 44] among which Grid-Computing Scheduling [45]. The SA algorithm may be summarized as follows:

1. Generate a random solution.
2. Calculate its cost.
3. Generate a random neighboring solution.
4. Calculate the new solution's cost.
5. Compare previous and new solution costs:
   - If $cost_{new} < cost_{prev}$: move to the new solution as it is better (i.e.: getting closer to an optimum). "Moving" to a new solution happens by saving it as the incumbent solution for next iteration.
   - If $cost_{new} \geq cost_{prev}$: maybe move to the new solution. Most of the time, the algorithm will eschew moving to a worse solution, however it sometimes elects to keep the worse solution in order to avoid being trapped in a local minimum. To decide, the algorithm calculates the 'acceptance probability' and then compares it to a randomly generated number in the interval [0;1]: if the acceptance probability is larger than the random number, the algorithm moves to the new solution. The explanation so far leaves out an important parameter called the temperature (as the algorithm is inspired by a method of heating and cooling metals). The temperature decreases with the iterations of the algorithm; it usually is started at 1.0 and decreased at the end of each iteration by multiplying it by a constant called α (typically between 0.8 and 0.99). Furthermore, SA performs better when the 'neighbor-cost-compare-move' process is carried about many times (typically between 100 and 1000) at each temperature[46].
6. Repeat steps 3-5 above until an acceptable solution is found or some maximum number of iterations is reached.

[8] each with a weight of 0.5

[9] A more detailed justification for the selection of this specific metha-heuristic is described in [10]. Note however that the model described in this work is (meta-) heuristic/algorithm agnostic; while SA provides satisfactory results, we do not pretend it to be the most efficient or effective technique. In fact, comparing all possible techniques is considered out-of-scope for this work.

Based on the $cost_{prev}$, $cost_{new}$ and temperature, the acceptance probability is calculated by means of Eq.(15) and can be seen as a recommendation on whether or not to jump to the new solution. The equation typically used for the acceptance probability is:

$$a = \min(1, e^{\frac{cost_{prev} - cost_{new}}{T}}) \quad (15)$$

where a is the acceptance probability, $cost_{prev} - cost_{new}$ is the difference between the previous cost and the new one and T is the temperature. This equation helps to move from a random solution to one with a very low cost as the acceptance probability:

- is always > 1 when the new solution is better than the old one. Since a probability cannot exceed 100%, we use $a = 1$ in this case.
- gets smaller as the new solution gets worse than the old one.
- gets smaller as the temperature decreases.

The algorithm is thus *"more likely to accept 'slightly-bad' jumps than 'really-bad' jumps, and is more likely to accept them early on, when the temperature is high"*[46].

In order to evaluate the fitness of the adapted SA implementation to the problem at stake, we simulated three distinct clusters : one without any overload, one with half of the servers being overloaded and a last scenario with all servers overloaded. Three distinct clusters of size 'S'[10] have been randomly generated for those simulations (see Table 5 and associated explanations (sub-section 4.3) for more details). The Pareto front has been approximated by lexicographic optimization on 100 equidistant points in the [min(overload);max(overload)] interval. Those successive optimization runs have been realized using the CPLEX® Mathematical Programming (MP) solver[11]. Lastly, the weighted sum method has also been implemented using the same CPLEX® solver as a yardstick for the evaluation of the metaheuristic effectiveness.

Table 4 Effectiveness and efficiency comparison of both the MP and the SA metaheuristic weighted-sum implementations

Overloaded servers		**MP** %	**MP** Secs	**SA** %	**SA** STD	**SA** Secs
no	cntc	67.75	41.42	69.31	1.56	0.15
	coc	0		0	0	
half	cntc	64.18	4905	63.92	3.07	0.13
	coc	7.11		15.22	7.17	
all	cntc	87.01	30.72	87.59	3.47	0.13
	coc	59.12		61.36	2.83	

Figure 6 illustrates the result of these three simulations (further described by Table 4):

- the cluster with **no** overloaded servers reaches a new assignment scheme with 67.75% of the initial inter-servers network traffic for the MP implementation and an average

[10] 50 containers on 5 servers

[11] See Table 2 for a complete description of the test environment specifications

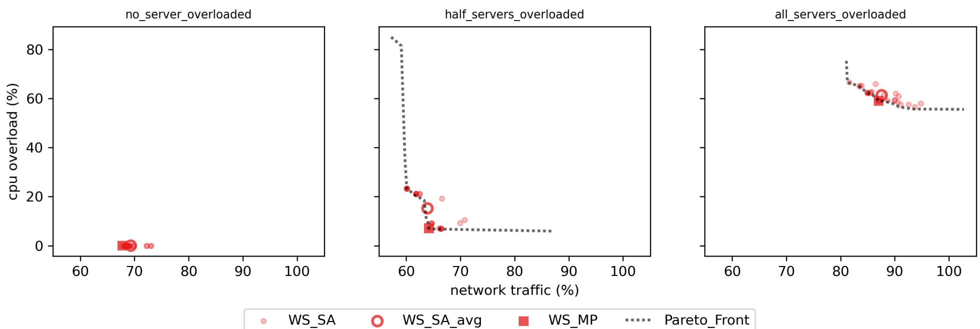

Fig. 6 Effectiveness of the SA implementation : location on the Pareto front and comparison with the MP solver exact optimum

of 69.31% for the 25 distinct runs of the SA metaheuristic. Both implementations manage to keep the 'coc' value at 0.0 (which explains the absence of Pareto Front in Figure 6). While both implementations reach comparable effectiveness, their efficiency greatly diverges as it takes 41.42 seconds for the MP implementation to complete and only an average of 0.15 seconds for the SA metaheuristic.

- the cluster with **half** of the servers overloaded reaches a new assignment scheme with 64.18% of the initial inter-servers network traffic for the MP implementation and an average of 63.92% for the 25 distinct runs of the SA metaheuristic. However the optimal assignment scheme computed by the MP implementation reduces the 'coc' value to 7.11% while the 25 runs of the SA metaheuristic only achieve 15.22% on average (with relatively high standard deviation of 7.17%). In this case the MP implementation surpasses the SA metaheuristic with a substantially more effective optimization of the 'coc' value. Nevertheless, the MP implementation takes 4905 seconds to complete while the SA metaheuristic only takes an average of 0.13 seconds.
- the cluster with **all** servers overloaded reaches a new assignment scheme with 87.01% of the initial inter-servers network traffic for the MP implementation and an average of 87.59% for the 25 distinct runs of the SA metaheuristic. Additionally, the computed optimal assignment scheme reduces the 'coc' value at 59.12% for the MP implementation and at 61.36% on average for the 25 runs of the SA metaheuristic. While both implementations reach comparable effectiveness, their efficiency greatly diverges as it takes 30.72 seconds for the MP implementation to complete and only an average of 0.13 seconds for the SA metaheuristic.

Main take-aways from this tests set are hereafter summarized:

- with both functions equally weighted, the weighted-sum optimum offers a satisfactory trade-off between both functions.
- due to its stochastic nature, the metaheuristic alleviates the effect of the second drawback listed herein-above: indeed non-convex parts of the Pareto front are occasionally reached (second scenario in Figure 6).

- for the (limited) tests set, the metaheuristic implementation solutions satisfactorily neighbour the exact MP solutions.
- the MP implementation suffers from long and unpredictable duration execution time. This disqualifying (in)efficiency makes it no realistic alternative to the short and predictable execution time of the SA implementation.

Further validating the effectiveness of the SA implementation on bigger cluster sizes turns out to be unrealistic due to the unreasonable time required by the MP implementation[12]. Nevertheless this first set of reassuring observations allows for confident extrapolation to bigger cluster sizes however with some likely degradation of the relative effectiveness on more complex setups (as observed on single objective optimization in [10]).

4.3 Time-budget based improvement

After the validation of the appropriateness of the weighted-sum SA metaheuristic to the problem at stake, remains the evaluation of its scalability to bigger cluster setups. To this end, four distinct cluster sizes (S, M, L and XL) have been defined as described by Table 5 where column:

- **'#C'** defines the number of containers.
- **'#S'** defines the number of servers.
- **'#SA'** defines the number of server affinity constraints. If such a constraint is defined for a container, it may only be assigned to a randomly defined set of 25% of the servers. This ratio has been arbitrarily defined and is deemed to represent cases where a container's requirements are only fulfilled by a limited subset of servers.
- **'#SAA'** defines the number of server anti-affinity constraints. If such a constraint is defined for a container, it may only be assigned to a randomly defined set of 75% of the servers. This ratio has been arbitrarily defined and is deemed to represent cases where a container's requirements cannot be met by all servers.
- **'#CA'** defines the number of container affinity constraints. If such a constraint is defined for a container towards an other container, it may only be assigned to the server hosting the other container.
- **'#CAA'** defines the number of container anti-affinity constraints. If such a constraint is defined for a container towards an other container, it cannot be assigned to the server hosting the other container.
- **'%NT'** defines the percentage of containers each container sends network traffic to.

For all simulations:

- 80% of the containers are reschedulable.
- Half of the servers offer 8 CPUs and 64GB of RAM each and the other half of the servers offer 4 CPUs and 32GB of RAM each. In each setup, one server is flagged as non-schedulable (to simulate a typical cluster Master Node).

[12] For the 'M' scenario (see Table 5), CPLEX crashes after 4.3h (out-of-memory on a 12GB RAM server). At this stage, it still reports a 'Gap' value of 25.81% in the minimalization of the 'cntc' function (first step of the algorithm required to compute the normalizing factor).

Table 5 Composition details of the four distinct scenario sizes

	#C	#S	#SA	#SAA	#CA	#CAA	%NT
S	50	5	5	5	5	5	20
M	100	10	10	10	10	10	10
L	500	50	50	50	50	50	2
XL	1500	150	150	150	150	150	1

- Individual required RAM/CPU is randomly assigned (within servers max capacity boundaries to allow hosting).
- Inter-container network traffic is randomly assigned on a scale from 1 to 5.
- Half of the servers are overloaded.

As a preliminary remark, it is worth noticing that the achieved effectiveness rates in the performance analysis that follows must not be interpreted through their absolute value, intrinsically depending on the test-cases parameters and constraints, but rather through their relative evolution among the proposed methods. The absolute value of the observed efficiency rates however are only dependent of the scenario sizes and can therefore reasonably be extrapolated and interpreted as representative for other cluster parameters and constraints (with limited possible variations depending on the used hardware).

Table 6 Decomposition of the average processing time (in seconds) of the WS_SA implementation for each scenario size

	S	M	L	XL
cntc_min	0.026	0.242	88.46	6056
cntc_max	0.025	0.259	89.94	5040
coc_min	0.020	0.162	45.16	3724
coc_max	0.021	0.159	39.40	3104
ws	0.033	0.277	109.3	8688
TOTAL	0.125	1.099	372.3	26612

The results from the first 25 test-runs executed for each scenario size already highlight a severe scalability limitation for the '**WS_SA**' implementation with an average completion time of 372.3 seconds ($6.2m$) for scenario 'L' and 26612 seconds ($\approx 7.5h$) for scenario 'XL'. Decomposing those timings (as presented in Table 6) enlightens the origin of the issue: while being ideal in terms of results, the normalization schema used (see Eq. (13)) is computationally too expensive.

To overcome this issue, a possibility may be to execute the four single objective optimization (finding 'cntc_min','cntc_max','coc_min' and 'coc_max') in parallel before the execution of the weighted-sum. However, this alternative is not retained as i) the wall-clock-time can only be approximately divided by two (assuming a server able to run those four threads in parallel) and ii) the resource usage remains unchanged. Two more efficient alternatives are instead hereafter proposed:

Table 7 Effectiveness and efficiency comparison of 4 variations of the SA weighted-sum implementation

Size	SA variation	f()	%	STD	Secs
S	WS_SA	cntc	79.5	4.9	0.12
		coc	44.7	10.5	
	WS_SA_BN	cntc	95.0	6.1	0.04
		coc	28.1	6.6	
	WS_SA_PN	cntc	79.9	5.4	0.04
		coc	45.5	10.9	
	WS_SA_PN_TB	cntc	75.4	3.2	1.0
		coc	39.7	5.5	
M	WS_SA	cntc	65.0	3.2	1.1
		coc	13.7	4.9	
	WS_SA_BN	cntc	80.5	6.5	0.31
		coc	10.5	3.7	
	WS_SA_PN	cntc	64.0	2.6	0.32
		coc	13.4	3.9	
	WS_SA_PN_TB	cntc	59.3	1.8	10.0
		coc	10.9	2.8	
L	WS_SA	cntc	52.5	1.2	372
		coc	21.4	0.4	
	WS_SA_BN	cntc	59.7	2.1	114
		coc	20.1	0.1	
	WS_SA_PN	cntc	55.9	1.7	117
		coc	20.3	0.2	
	WS_SA_PN_TB	cntc	55.9	1.7	60
		coc	20.3	0.2	
XL	WS_SA	cntc	60.2	0.9	26612
		coc	19.2	0.2	
	WS_SA_BN	cntc	74.4	1.3	9237
		coc	18.0	0.1	
	WS_SA_PN	cntc	63.9	1.0	8952
		coc	18.5	0.2	
	WS_SA_PN_TB	cntc	70.6	1.0	644
		coc	19.0	0.1	

1. the '**WS_SA_BN**' implementation: in this basic normalization schema the value of 'cntc_min' and 'coc_min' is 0 while the value of 'cntc_max' and 'coc_max' are the 'cntc' and 'coc' value of the initial assignment scheme respectively. This implementation normalizes the objective functions by their respective magnitude at the initial point. As reported in Table 7, if this method allows to approximately divide the computation time (and cost) by 3, it however substantially impairs the optimization result (see also Figure (7)). This is explained by the fact that the initial point may provide very poor representation of the function behaviour at optimality.

2. the '**WS_SA_PN**' implementation: at initial point, the value of 'cntc_min' and 'cntc_max' is the 'cntc' value of the initial assignment scheme. The same holds for the 'coc' parameters which then take the 'coc' value. The value of η_{cntc} and η_{coc} is artificially kept to 1 as long as 'cntc_min' equals 'cntc_max' and 'coc_min' equals 'coc_max' respectively. After each iteration within the metaheuristic, 'cntc_min' and 'coc_min' are updated with the 'cntc' and 'coc' value respectively if smaller. Conversely, 'cntc_max' and 'coc_max' are updated with the 'cntc' and 'coc' value respectively if larger. The cost of the best assignment scheme found so far is updated with the new normalization factors before being compared to the cost of the assignment scheme from the latest iteration. This progressive approach combines the best of both worlds: it reaches comparable effectiveness as with the '**WS_SA**' implementation while achieving similar efficiency as with the '**WS_SA_BN**' implementation.

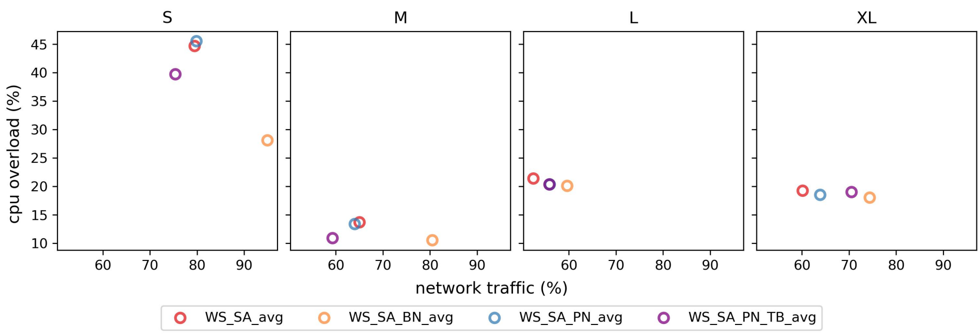

Fig. 7 Effectiveness of the SA implementation with ideal (red), basic (orange), progressive (blue) and time-budget based progressive (purple) normalization schemas

Taking a step back from those successive performance increments allows to introduce the 'WS_SA_PN_TB' implementation that extends the 'WS_SA_PN' implementation with the concept of time-budget. More specifically:

- the solution for the 'S' and 'M' scenario is obtained in a very short time span. Performing multiple successive runs within a longer but still relatively short delay allows to take the best solution out of those iterative internal runs. For instance, solving the 'S' and 'M' scenarios in 1 and 10 seconds respectively allows in both case to surpass the effectiveness of the 'WS_SA' implementation. This is illustrated in Figure (7) and detailed in Table 7.
- the delay required to obtain the solution for the 'L' and 'XL' scenarios may however still be estimated as relatively too long: limiting it to a given number of seconds is a desirable feature for most operational environments. However, simply stopping the metaheuristic at deadline does not allow to make best use of the imparted delay. This is due to the shape of the progress curve of the SA metaheuristic, as illustrated in

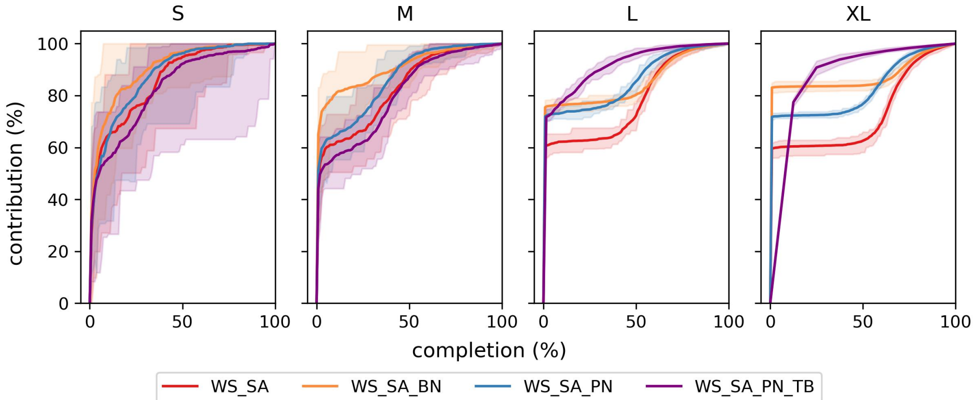

Fig. 8 Progress evolution of the SA implementation with ideal (red), basic (orange), progressive (blue) and time-budget based progressive (purple) normalization schemas

Figure (8), that (for the 'L' and 'XL' scenarios) rapidly reaches a plateau with lower progression for approximately 40-50% of the iterations before 'restarting' again[13]. For instance if the metaheuristic is only given a third of the total amount of time it would require, it would stagnate most of the time with few progress. Therefore we propose a 'fast-forward' mechanism that intends to make best use of the imparted time where the metaheuristic derives, after each search iteration, the remaining number of iterations it still can perform (based on average time taken by previous iterations and remaining number of iterations). The metaheuristic then skips the intermediary iterations and directly jumps to the iteration it should be by updating the 'Temperature' parameter of the SA algorithm. To avoid recurrent micro-adjustments, a minimum jump of 10 iterations is required[14] and can only be performed if there are more than ten iterations remaining. When given a 60 seconds delay for the 'L' scenario, this 'fast-forward' mechanism allows to reach the exact same optimum solution as for the 'WS_SA_PN' while being given half of the time required (fixed 60 seconds vs 117 in average for full completion), as illustrated in Figure (7) and detailed in Table 7. Additionally, when given a 600 seconds delay for the 'XL' scenario, this 'fast-forward' mechanism impacts the effectiveness of the 'cntc' value (70.6% vs 63.9%); it slightly improves the 'coc' value (19% vs 20.3%) but most importantly allows for a completion time of $\approx 7\%$ of the normally required delay (fixed 600 seconds vs 8952 in average for full completion).

As concluding remark, this performance analysis of the 'WS_SA_PN_TB' implementation does not aim at exhaustiveness: the provided delays have been arbitrarily defined such as to exemplify the possibilities offered by the concept of this time-budget based strategy that allows for i) an improved effectiveness for scenario sizes with

[13]This suggests that the size of the search-space negatively influences the performance of the exploration phase with only little progress observed in the progressive transition to the exploitation phase.
[14]This value has been empirically defined and can be adapted according to cluster specificity.

insignificant time-to-complete and ii) a significant impact on efficiency for scenario sizes with considerable time-to-complete with a limited impact on effectiveness.

5 Validating the rescheduling system in a container orchestration platform

This section aims at validating the 'WS_SA_PN_TB' implementation presented in Section 4 within a K8s cluster. K8s has been selected as container orchestration platform for its prominent market position and proven track record [47]. Originally developed by Google and currently being maintained by the Cloud Native Computing Foundation (CNCF), K8s is an open-source container orchestration system for automated deployment, scaling and management of containerized applications.

This section first introduces the scheduling and descheduling mechanisms of K8s, as well as the 'Pod QoS Classes' mechanism used by K8s to manage resource constraints[15], after which the dynamic rescheduling system, embedding the 'WS_SA_PN_TB' implementation, is presented. Lastly, the impact on application service time of the system under varying loads is tested and evaluated on 3 distinct applications.

5.1 Scheduling and descheduling containers in K8s

5.1.1 Workload resources

K8s consists of multiple components, also known as workload resources, to be able to offer the aforementioned services. The main workload resources used in this work, are briefly introduced below:

- **Pod:** A Pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. Pods are the smallest deployable units of computing that can be created and managed in K8s.
- **Deployment:** A Deployment provides declarative updates for Pods. A Deployment uses a description of a desired state and the Deployment controller changes the current state towards that desired state.
- **Service:** An abstract way to expose a set of Pods as a network Service. It offers two advantages: i) a permanent link for internal and external referencing to Pods (DNS) and ii) load balancing amongst the endpoint Pods.
- **Namespace:** A Namespace is a non-overlapping set of managed resources and allows for workload resource isolation within a cluster. Namespaces allow for cluster multi-tenancy or for the separation of development and production environments.

K8s clusters are composed of one Master Node and a set of Worker nodes. While the Worker nodes host the application workload resources described hereinabove, the Master node hosts most of the cluster management components. Main component of interest from this Control Plane are the scheduler and descheduler, hereafter further described.

[15] Most of the descriptive part of subsection 5.1 is directly imported from [48]

5.1.2 Pod QoS Classes

K8s classifies each Pod into a specific QoS class based on the resource (CPU and RAM exclusively) requests of the containers in that Pod, along with how those requests relate to resource limits. QoS classes are used by K8s to decide which Pods to evict from a Node experiencing resource pressure. When this eviction is due to resource pressure, only Pods exceeding resource requests are candidates for eviction. The possible QoS classes are

- **BestEffort** Pods have neither resource requests or limits on their configuration files for each of their containers. Such Pods can use node resources that are not specifically assigned to Pods in other QoS classes. These Pods are the first ones to be considered for eviction if the node comes under resource pressure.
- **Burstable** Pods have some lower-bound resource guarantees based on the request, but do not require a specific limit. If a limit is not specified, it defaults to a limit equivalent to the capacity of the Node, which allows the Pods to flexibly increase their resources if resources are available. In the event of Pod eviction due to Node resource pressure, these Pods are evicted only after all BestEffort Pods are evicted.
- **Guaranteed** Pods have the strictest resource limits and are least likely to face eviction. They are guaranteed not to be killed until they exceed their limits or there are no lower-priority Pods that can be preempted from the Node. They may not acquire resources beyond their specified limits. For every Container in the Pod, the CPU and RAM limit must equal the resource request.

5.1.3 The Kubernetes Scheduler (KS)

K8s default scheduling system (KS) has static rules to schedule Pods in a cluster. Container resource requests and limits are specified within the Pod configuration file. A resource request is the minimum amount of resources (e.g. CPU and/or RAM) required by a container while a resource limit is the maximum amount of resources that can be allocated to the container. Additionally, affinity constraints can also be specified within a Pod configuration file. The affinity feature consists of two types of affinity: Node (anti-) affinity allowing to constrain which nodes a Pod must (not) be scheduled on and Inter-pod (anti-) affinity allowing to constrain which nodes a Pod must (not) be scheduled to, based on the Pods already running on that node. If those constraints conflict or if no node satisfies the full set of constraints, then the Pod cannot be scheduled by the KS. Those 4 varieties of affinities can be defined as hard[16] or soft[17] constraints; the latter being associated with a preference weight indicating to which extent the constraint may be relaxed in case of constraints conflict for Node attribution[18]. The KS uses those resource and (anti-) affinity constraints in its allocation decisions.

Every Pod that requires allocation is first added to a queue, which is monitored by the KS. As illustrated in Figure 9, the KS allocates Pods to Nodes based on a two-step procedure. The first step is to filter the available Nodes based on a set of predicates

[16] "requiredDuringSchedulingIgnoredDuringExecution"
[17] "preferredDuringSchedulingIgnoredDuringExecution"
[18] For scoping reasons, this work only considers hard constraints. The impact of soft constraints is considered as a candidate topic for future extensions of this work.

to decide which Nodes are capable of running a specific Pod. The second step is to calculate each Node's priority, where the KS ranks each remaining Node based on the requirements. These steps are repeated for all Pods that require scheduling. The Node priority calculation is based on a set of priorities, where each remaining Node is given a score between 0 (worst fit) and 10 (perfect fit). The highest scoring Node is selected to run the Pod. If more than one Node is classified as the highest-scoring Node, then one of them is randomly chosen. When the allocation decision is made, the KS informs the API server indicating where the Pod must be scheduled. This operation is called 'Binding'.

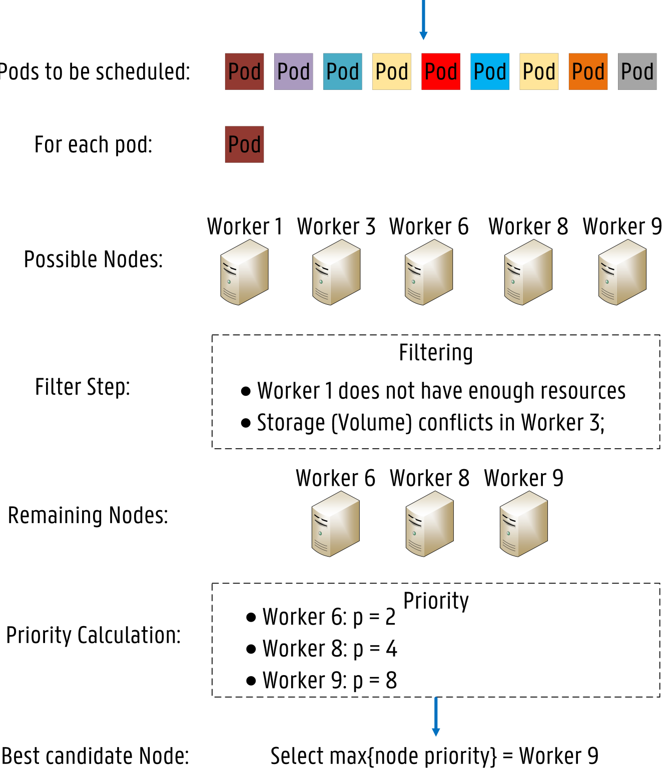

Fig. 9 Sample of detailed scheduling operations of the Kube–Scheduler [49]

It should be noted that the KS searches for a suitable Node for each Pod, one at a time. The KS does not take the remaining Pods waiting for deployment into account in the scheduling process, nor does it reschedule running Pods if the cluster state has evolved since their initial deployment. The KS statically schedules Pods one by one without considering a global view on the system. This work thus extends the KS by implementing a specific rescheduler system that works alongside the KS and focuses on rescheduling Pods with global optimization in mind while the KS only considers predefined static predicates for local optimization.

5.1.4 K8s descheduler

The KS decisions, whether or where a pod can or can not be scheduled, are guided by its configurable policy which comprises of set of predicates and priorities. The KS decisions are influenced by its view on the cluster at that point in time when a new Pod appears for scheduling [50]. As K8s clusters are very dynamic and their state changes over time, there may be desire to move already running pods to some other nodes for various reasons:

- Some nodes are under or over utilized.
- The original scheduling decision does not hold true any more, as taints or labels are added to or removed from nodes, pod/node affinity requirements are not satisfied any more.
- Some nodes failed and their pods moved to other nodes.
- New nodes are added to clusters.

Consequently, there might be several pods scheduled on less desired nodes in a cluster. The K8s descheduler, based on its policy, finds pods that can be moved and evicts them, relying on the KS for their re-scheduling afterwards. The K8s descheduler supports the following 10 strategies:

- **RemoveDuplicates**: makes sure that there is only one pod associated with a ReplicaSet, ReplicationController, StatefulSet, or Job running on the same node.
- **LowNodeUtilization**: finds nodes that are under utilized and evicts pods, if possible, from other nodes in the hope that recreation of evicted pods will be scheduled on these underutilized nodes. Currently, node resource consumption is determined by the resource requests of pods, not their actual usage.
- **HighNodeUtilization**: finds nodes that are under utilized and evicts pods from those nodes in the hope that these pods will be scheduled compactly into fewer nodes.
- **RemovePodsViolatingInterPodAntiAffi-nity**: makes sure that pods violating interpod anti-affinity are removed from nodes.
- **RemovePodsViolatingNodeAffinity**: makes sure all pods violating node affinity are eventually removed from nodes.
- **RemovePodsViolatingNodeTaints**: makes sure that pods violating NoSchedule taints on nodes are removed.
- **RemovePodsViolatingTopologySpread-Constraint**: makes sure that pods violating topology spread constraints are evicted from nodes.
- **RemovePodsHavingTooManyRestarts**: makes sure that pods having too many restarts are removed from nodes.
- **PodLifeTime**: evicts pods that are older than maxPodLifeTimeSeconds.
- **RemoveFailedPods**: evicts pods that are in failed status phase.

Those strategies can be further configured to limit the scope of the descheduler through, among others, *namespace*, *priority*, *label* and *node fit* pod filtering.

By allowing the eviction of pods not fulfilling predefined conditions, the combination of the K8s descheduler and scheduler offers thus a first option for dynamic rescheduling. However, it suffers different limitations:

- Firstly, the violating pods are descheduled from nodes that violate the constraints. Though, depending on the deployment strategy, there is no guarantee that the scheduler will optimally place new pod instances (cfr. *LowNodeUtilization* and *HighNodeUtilization* strategies).
- Secondly, K8s does not consider 'observed' pod resource consumption but instead uses the predefined pod resource requests and limits, which may be prone to approximation and consequently inefficiency.
- Additionally, K8s only supports the static definition of resources. There is no mechanism to consider the fluctuation of resource requirements over time.
- More fundamentally, the pod-centric definition of resource request and limit does not allow for inter-pod relationships consideration.

Consequently, the K8s mechanism for pod rescheduling does not offer enough efficiency and flexibility for fine-grained and observation-based rescheduling able to reshuffle pods based on actual resource consumption fluctuation over time and pods interdependencies.

5.2 Architecture and design of the rescheduling system

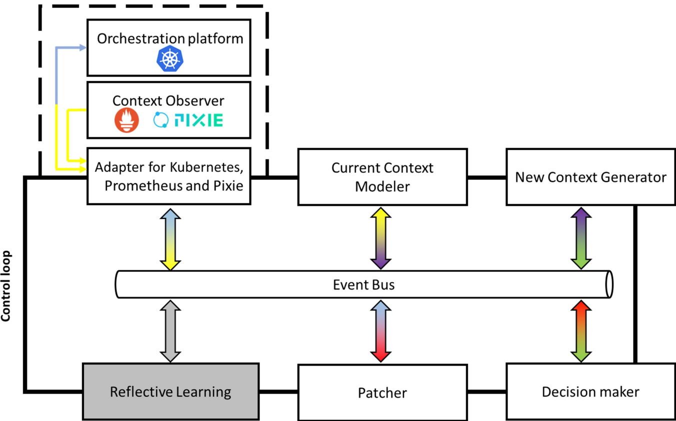

Fig. 10 The self-driving rescheduling control loop [10]

The proposed dynamic container rescheduling system, already extensively described in [10], is designed as a closed control loop whose main purpose is to push the cluster closer to the desired state[19]. As illustrated in Figure 10, it is based on the following 6 components:

[19] Under stable load, the control loop should eventually stop to adapt the system it controls by converging to an optimum.

1. The **Adapter** is in charge of interfacing the dynamical system (i.e. the orchestration platform and monitoring tools) and consequently allows the portability of the control loop to other environments. It mainly fulfills 2 functionalities:
   (a) **Context data fetching**: to periodically gather cluster context info through:
      - Prometheus for pod and node data required by the model.
      - The K8s API to collect all pod and node (anti-) affinity constraints.
      - The PIXIE backend for network traffic metrics among pods for the last 's' seconds[20].

   (b) **Container rescheduling**: when triggered, for each entry in the received ordered list of pods to reschedule, the Adapter sends to the K8s API server a strategic merge patch to update the 'nodeSelector' field of the pod's deployment manifest with the Fully Qualified Domain Name (FQDN) of the node it must be rescheduled onto. This action causes the eviction of the pod instance from its current node and the scheduling of a new instance on the target node.
2. The **Current Context Modeler** periodically queries the dynamical system (through the Adapter) and constructs the *currentContext*, a logical representation of its state.
3. The **New Context Generator** uses the generated *currentContext* to generate a *newContext* by means of the 'WS_SA_PN_TB' implementation[21]. Additionally, the New Context Generator can be configured to only consider specific namespaces, which allows to adapt the scope of reschedulable containers.
4. The **Decision maker** compares the cost from both the *currentContext* and the *newContext* and, based on decision criteria, enacts the execution of the proposed rescheduling. The decision criterion used in this work is a customizable minimum cost improvement ratio. It can however be extended or fine-tuned (e.g. only apply the rescheduling if a certain percentage of the pods to be rescheduled have not been rescheduled recently).
5. When instructed to apply the *newContext*, the **Patcher** first generates a sequence of individual pod rescheduling actions ensuring permanent respect of constraints all along the rescheduling. Afterwards, the Patcher sends that ordered list to the Adapter for sequential execution of the individual rescheduling orders.
6. The **Reflective Learning** analyzes over time the impact the rescheduling decisions had on the cluster and adapts the parameters of the Current Context Modeler, the New Context Generator and the Decision Maker components in order to continuously improve the performance of the rescheduling system. This component however has not been implemented in this work but is shortlisted for future work.

[20] Pixie is an open source observability tool for K8s applications that is contributed to by New Relic, Inc. as a CNCF sandbox project since June 2021 [51]. Pixie has been selected for its streamlined simplicity of integration, though any other network monitoring tool able to report on inter-pod network traffic can be used instead.

[21] By design any other possible optimization algorithm/metaheuristic can be used as the New Context Generator component launches its execution through an algorithm agnostic interface.

5.3 Validating the model in a K8s cluster

5.3.1 Setup definition

The experiment[22] consists of measuring the impact of the rescheduling on the application service time, under varying loads (see Table 8 for the two scenarios), of three applications concurrently running on a K8s cluster.

Table 8 Volumetry of the varying load experiments

Application	Metric	Load	
		Low	High
Obelisk	msg./sec.	6	30
Online Bout.	clients	6	30
VRP	clients	6	30

While not pretending to exhaustiveness, this set of experiments aims at validating the model and demonstrating its benefits; the measured performance improvements being directly linked to the defined set of applications, the level of workload and the lab setup. Further, the stochastic nature of the used metaheuristic also induces variations among test runs in same conditions. Therefore we decide to only report on few representative rescheduling schemes. The first application is the CPU intensive VRP application described in Section 2.

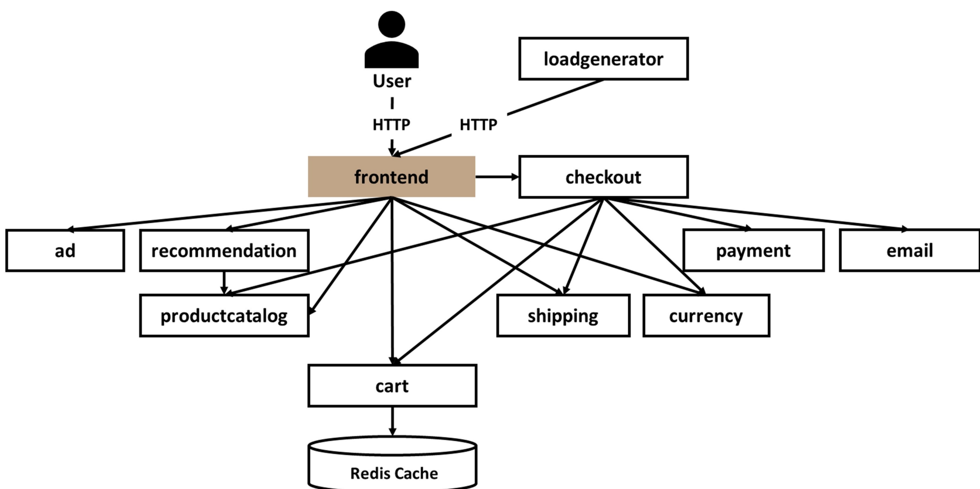

Fig. 11 Architecture diagram of the web-based e-commerce 'Online Boutique' app [10]

[22] See Table 2 for the test environment specifications

The second application is the so-called 'Online Boutique', a cloud-first microservices demo application developed and used by Google to demonstrate use of cloud technologies. Main advantage of this application resides in its ease of deployment with little to no configuration while still being representative of a typical microservices based e-commerce web-app where users can browse items, add them to the cart, and purchase them [52]. Figure 11 illustrates the architecture of the 'Online Boutique'[23]. The higher number of services and interactions makes it a valuable complement to the CPU intensive VRP application. The *loadgenerator* service has been rewritten to better accommodate the logging needs of the test as well as to allow for a more fine-grained control of the customer journey that simulates 'x' users (initially shifted by $\frac{1}{x}$ seconds) endlessly looping on this browsing script:

- Access the 'Home page' of the 'Online Boutique' by means of a HTTP GET call to the */* URI of the *frontend* service and hold the returned session-id cookie.
- Set the currency to be used for the forthcoming transactions by means of a HTTP POST call to the */setCurrency* URI of the *frontend* service with the session-id cookie embedded in the header and the selected currency as parameter.
- Repeat three times:
  - Access the product page of a randomly selected product by means of a HTTP GET call to the */products/{product-id}* URN of the *frontend* service with the session-id cookie embedded in the header. The *{product-id}* is the identifier of the selected product.
  - Add $0 < Q < 6$ occurrences of the product to the cart by means of a HTTP POST call to the */cart* URI of the *frontend* service with the session-id cookie embedded in the header and the selected product and quantity 'Q' as parameters.
- Finally, book the order by means of a HTTP POST call to the */cart/checkout* URI of the *frontend* service with the session-id cookie embedded in the header and the client details (email address, physical address and credit card details).

The last application is a simplified version of 'Obelisk' [53], an open-source[24] Internet of Things (IoT) cloud-based data-hub platform. It has the advantage of being based on widely used open-source packages, which made the implementation of this test version straight-forward. Furthermore its event-based microservice architecture, presented in Figure 12, worthfully complements the two other HTTP based applications.

Obelisk relies on Kafka as central message broker.

The *Ingest API* expects as request body a JavaScript Object Notation (JSON) array representing a batch of 1..n metric data events. Once the entire request is received, it splits the array into 'n' individual metric data events and publishes those to the *metrics.events* Kafka topic.

The *Sink Service* as well as the *Scope Streamer Service* are both subscribers of this topic and consequently consume the queued messages. As they are part of two distinct consumer groups they both receive and process messages at their own pace.

[23] More details on the scope of each individual service can be found on the official website of the app [52] as well as in our previous article [10]

[24] https://github.com/idlab-discover/obelisk

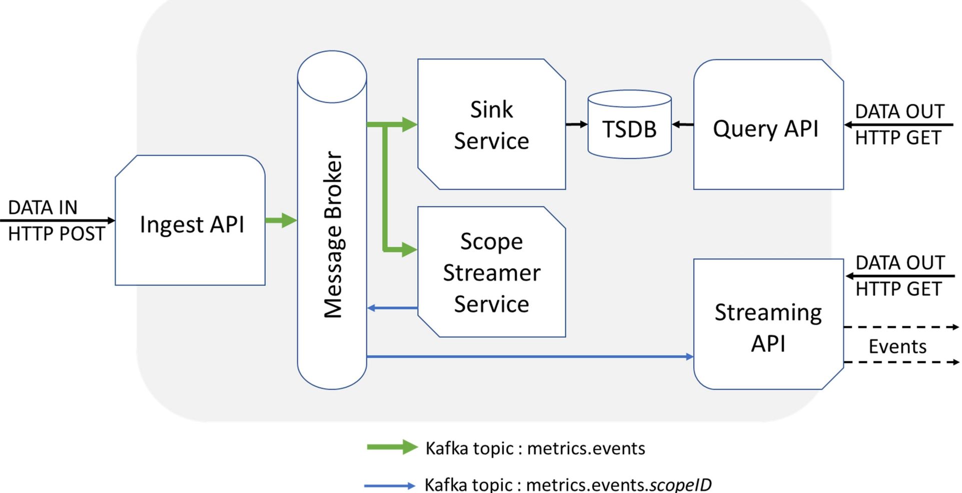

Fig. 12 Architecture of *Obelisk*, the IoT data-hub [10]

The *Sink Service* accumulates those individual metric events and performs a batch write to the Time Series DataBase (TSDB) when one of the two following conditions is met: the last batch write did happen 'x' milliseconds ago or the amount of buffered events equals to 'y'. Both 'x' and 'y' are configurable parameters of the *Sink Service*.

The *Scope Streamer Service* also consumes those individual metric events and, based on their respective scope[25], forwards them to the appropriate topic: one topic being defined per scope (*metrics.events.scope* where 'scope' is the scope name).

When a client application is willing to consume those streamed events, it calls the *Streaming API* which continuously returns the metric events it gets from the *metrics.events.scope* topic as they arrive.

Lastly, the *Query API* allows for the retrieval of historical data (with filtering and pagination mechanisms) that it fetches from the TSDB.

To ease the interpretation of the results both the *metrics.events* and the *metrics.events.test* Kafka topics are configured with the number of partitions and the replication factor equal to 1 and thus are exclusively hosted on *Kafka Broker 0* and *1* respectively. Messages are being emitted every second from 'x' simulated client devices (initially shifted by $\frac{1}{x}$ seconds) on the 'test' scope.

5.3.2 Experiment results

In order to ensure fair comparison of the results among varying loads (load volumetry is defined in Table 8), pods are initially scheduled as described in Table 10: this distribution aims at maximizing the rescheduling gain as directly inter-communicating

[25] Data isolation is internally ensured through the concept of scopes which represent logical data sets with configurable perimeter. A scope can be understood as a labelling mechanism aiming at isolating data between different contexts of use. Data access APIs for data ingestion, querying and streaming all require the scope to be mentioned.

Pods are never assigned to the same node. Further, both the *Optimizer* and *Plotter* CPU bound Pods, are assigned to the same node. Additionally, as described in Table 9, pods have all been assigned CPU limits corresponding to nodes capacity (i.e. 24)[26]; all Pods are thus assigned the K8s *Burstable* QoS with maximum flexibility.

Table 9 The Pods resource requests and limits defined for the experiments - with unreschedulable Pods flagged with a star '*'

App.	Svc.	Requests		Limits	
		CPU (#)	**RAM (GB)**	**CPU (#)**	**RAM (GB)**
Online Boutique	*RedisCache**	0.5	0.5	24	4
	Others	0.1	0.12		24
VRP	*Frontend*	1	0.5	24	24
	Optimizer	4			
	Plotter	2			
Obelisk	*TSDB**	0.5	0.25	24	1
	*Kafka0..2**	0.5	2		8
	Others	0.1	0.25		24

Both the low and high load scenarios start with the same Pod-to-Node distribution. Configurable client apps generate the specific load to the three apps during the eleven minutes of the experiment. Five minutes after the start of the experiment, the rescheduling system models the current execution context, computes the best assignment scheme and emits specific (re-)scheduling orders to the K8s scheduler in order to reflect the computed 'optimum' Pod assignment scheme. This rescheduling phase lasts overall a little bit less than one minute (between 45 and 50 seconds) for our test environment, after which the experiment continues for the remaining five minutes. Those delays ensure node stabilisation and representative results. The main outcome from the pod reassignment is illustrated in Figure 13 and further described in Table 10 and can be summarized as follows:

- The **Online Boutique** app is fully consolidated on Node2, where the unreschedulable *RedisCache* Pod is hosted, for the low and high load scenarios that both achieve approximately a 15% application service time improvement (15.45% and 17.1% respectively).
- The **VRP** app is fully consolidated on Node6 for the low load scenario, achieving a 5.8% service time improvement. While this rescheduling actively contributes to the reduction of the network traffic among nodes and does not generate any overload for Node6 (see Figure 13), the consolidation of the app on Node4, with only one Pod being reassigned instead of 3, would have achieved a comparable benefit with a lower rescheduling impact. We therefore report this point as a future possible improvement track in Section 6. For the high load scenario, Node4 is overloaded

[26]While this might at first sound counter-intuitive, in most cases, setting CPU limits do more harm than help. In fact, they are the number one cause of CPU throttling [54]. Tim Hockin, one of the K8s maintainers at Google, even suggests to never set CPU limits (https://x.com/thockin/status/1134193838841401345?s=20)

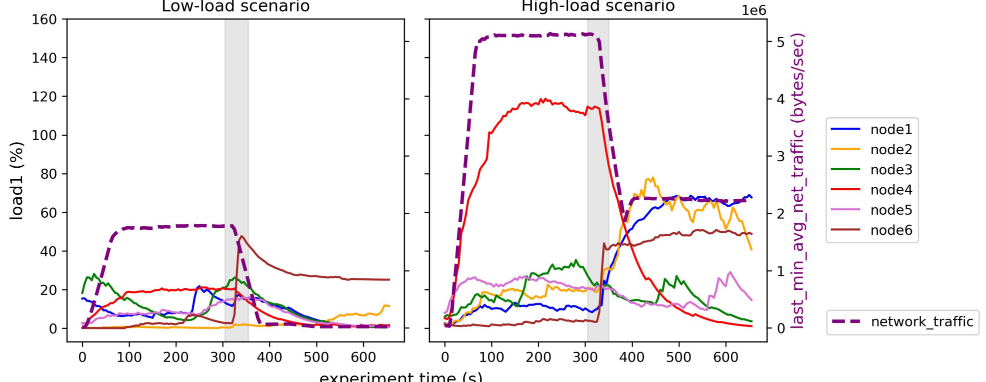

Fig. 13 Evolution of the node average load and cluster network traffic (with the effective rescheduling period indicated by the grey rectangle in the background)

before the rescheduling; both the *Frontend* and *Plotter* Pods are rescheduled on Node6 (exhibiting a load of approximately 50% after the rescheduling) while the *Optimizer* Pod is isolated on Node1 (exhibiting a load of approximately 70% after the rescheduling). Being the only Pod scheduled on that Node, allows it to exclusively use all the CPU power it needs which unleashes the application throughput that booms after the rescheduling to a 27.2% improvement.

- For both the low and high load scenarios (with a comparable service time improvement of approximately 21% (21.01% and 21.45% respectively)) and accordingly to the pre-distribution of its unreschedulable pods, the **Obelisk** app is split on Nodes 3 to 6 as such:
  - The *Query API* is hosted on Node3, already hosting the unreschedulable *TSDB* Pod, that it accesses to retrieve historical data requested by the simulated end user.
  - Both the *Scope Streamer Service* and *Streaming API* Pods are rescheduled to Node5, already hosting the *Kafka1 Broker* Pod that owns the scope specific *metrics.events.scope* topic used by the client test app. Note that the *Scope Streamer Service* Pod is rescheduled to Node5 instead of Node4 as it enriches all incoming messages with additional technical metadata (timestamps, podID, etc.): outgoing messages are thus slightly heavier than incoming ones.
  - Finally, The *Ingest API* and *Sink Service* Pods are rescheduled to Node4, already hosting the unreschedulable *Kafka0 Broker* Pod owning the *metrics.events* topic where all incoming events are pushed by the *Ingest API* Pod and consumed by the *Sink Service* Pod.

As described in Figure 13, the rescheduling almost extinguishes the network traffic among nodes (with a reduction of 87.5%) in the low-load scenario while avoiding any overload within the cluster after consolidation. Besides the substantial reduction of

Table 10 The effect of Pods rescheduling on application service time for the low and high load scenarios (lls and hls respectively) - with unreschedulable Pods flagged with a star '*'

App.	Svc.	Node assignment			Avg. service time (ms.)					
		lls		hls	lls			hls		
		Before	After	After	Before	After	Improvement	Before	After	Improvement
Online Boutique	*Frontend*	2	2	2	447	378	**15.45%**	910	755	**17.1%**
	Currency	6	2	2						
	Ad	1	2	2						
	Prod.Cat.	1	2	2						
	Reco.	3	2	2						
	Checkout	3	2	2						
	Payment	6	2	2						
	Shipping	4	2	2						
	Email	5	2	2						
	Cart	5	2	2						
	*RedisCache**	2	2	2						
VRP	*Frontend*	5	6	6	1847	1740	**5.8%**	3257	2372	**27.2%**
	Optimizer	4	6	1						
	Plotter	4	6	6						
Obelisk	*Ingest*	1	4	4	7.56	5.97	21.01%	5.35	4.2	21.45%
	Sink	1	4	4						
	Query	2	3	3						
	ScopeStreamer	2	5	5						
	Streaming	3	5	5						
	*TSDB**	3	3	3						
	*Kafka0**	4	4	4						
	*Kafka1**	5	5	5						
	*Kafka2**	6	6	6						

network infrastructure cost this consolidation may represent at scale, our hypothesis of improved application performance gets confirmed for the three applications that all benefit from an improved service time (of 15.45%, 21.01% and 5.8% for the Online Boutique, Obelisk and VRP applications respectively) as reported in Table 10. Though, at low CPU load, the VRP application is the one benefiting the least from the rescheduling, due to its CPU bound nature.

The Pod assignment scheme after rescheduling in the high-load scenario is similar to the low-load one for the *Obelisk* and *Online Boutique* apps: though it diverges for the *VRP* app as the three Pods can no longer hold on the same node without overloading it : in fact, the combination of the *Optimizer* and *Plotter* Pods already exceeds the saturation point (they are responsible for 99% of the consumed CPU power of Node4 before the rescheduling and, by extension, of the observed 120% load; according to the 'coc' function to be minimized they should thus as much as possible not be hosted on the same node). However, the *Plotter* and *Frontend* Pods can be co-located without overloading a node and are thus both rescheduled onto Node6, with the *Optimizer* Pod being isolated on Node1. This way, the *VRP* app benefits from a service time improvement of 27.2% after the rescheduling while the two other apps still achieve approximately the same improvement as the one they achieve in the low-load scenario (17.1% vs 15.45% for the *Online Boutique* app and 21.45% vs 21.01% for the *Obelisk* app). The assignment scheme after rescheduling is again optimized according to the

model as no node experiences any overload (see Figure 13) and the network traffic among nodes is minimized. Note that in this high-load scenario, the network traffic among nodes is approximately halved (with a decrease of 56.51%): this still remarkable result is however noticeably lower than in the low-load scenario where the *VRP* app can be fully hosted on one single node.

5.3.3 Comparison with the K8s descheduler

To illustrate the differences between the rescheduling system presented in this work and the K8s descheduler, we hereby analyse its impact on the previously introduced high-load scenario and compare its performance with the rescheduling system presented in this work.

While only focusing on CPU balance, the closest configuration of the K8s descheduler is the *LowNodeUtilization* strategy which '*finds nodes that are under utilized and evicts pods, if possible, from other nodes in the hope that recreation of evicted pods will be scheduled on these underutilized nodes*' [27]. Main differences that still persist with the system introduced in this work are :

- The K8s descheduler is not observability oriented but rather configuration based: resource consumption is determined by the Pods requests, not their actual usage.[28].
- The K8s descheduler does not take into account the network traffic among nodes, nor does it take into account the load1 metric but rather their requested resources (CPU and/or RAM) and/or the total number of allocatable Pods.
- The K8s descheduler does not aim at optimizing the overall state of the cluster but rather ensures that configured node thresholds are not exceeded; when they are, it just deschedules some pods '*in the hope*'(sic) that they will be assigned to underutilized nodes.

Table 11 The Node resource requests (%) before K8s descheduling

%	Node					
	1	**2**	**3**	**4**	**5**	**6**
CPU	2.51	4.29	2.63	29.18	7.93	3.76
RAM	20.83	22.57	22.3	32.09	29.04	27

Based on the CPU resource request of each node (see Table 11), we define the CPU *overutilization* threshold at 25% resulting in defining Node4 as *overutilized*. Among the $216 (= 6^3)$ possible reassignment schemes[29], we only describe hereafter both the worst and best hypothetical reassignment schemes.

The worst reassignment scheme consists in reassigning all three Pods on Node5 as the single *VRP* app on its own already overloads the node; as illustrated in Table 12,

[27] https://github.com/kubernetes-sigs/descheduler - 17/11/2023

[28] Currently, pods request resource requirements are considered for computing node resource utilization.(...) Implementing metrics-based descheduling is currently TODO for the project. 17/11/2023 - https://github.com/kubernetes-sigs/descheduler

[29] 3 (reschedulable) Pods that may be scheduled onto 6 distinct nodes (the scheduler may indeed reassign a pod to the node it was running on prior to the descheduling)

Table 12 The impact of the worst and best K8s descheduler reassignment schemes on application service time

		Worst	Best
Rescheduling	*Optimizer*	**5**	**6**
	Plotter	**5**	**5**
	Shipping	**5**	**3**
Impact	**Online Bout.**	-93.0%	1.2%
	VRP	-26.5%	24.4%
	Obelisk	-4.4%	7.1%

this deteriorates the service time for the three applications. The best reassignment scheme consists in moving the *Plotter* Pod to Node5 (joining the *Frontend* Pod on Node5), the *Optimizer* Pod to Node6 (the least used node before rescheduling) and the *Shipping* Pod to Node3 where the *Checkout* Pod already runs. In this scenario, all three applications experience an improved service time, and more specifically:

- The *VRP* application benefits the most from this reassignment with an improvement of 24.4%, almost reaching the optimal improvement where both the *Plotter* and *Frontend* pods were also running on the same (underloaded) node. In this rescheduling though, the *Optimizer* Pod is not running alone on its node but has to share it with other Pods as well, resulting in this slightly lower improvement.
- *Obelisk* improves by 7.1%, about a third of the optimal improvement. While no *Obelisk* Pod has been rescheduled, the application service time still improves because of the relaxing of pressure on Node4, hosting the *Kafka0* broker, that is not overloaded anymore after the rescheduling.
- The *Online Boutique* service time improves by 1.2%. This improvement is explained by the dual benefit of moving the *Shipping* Pod out of the overloaded Node4 to Node3 where the *Checkout* service resides. Though this consolidation only concerns 1 (marginally used Pod) out of the 11 Pods of the application; therefore the relatively limited improvement when compared to the 17.1% of the optimal reassignment scheme.

During this test phase, multiple experiment runs have been executed, not all explicitly reported in this work; main take-aways are hereafter shortlisted: ,

- No K8s descheduler reassignment scheme did surpass the optimal reassignment scheme proposed in this work; and this for any of the three applications.
- All other K8s descheduler reassignment schemes observed were systematically between the worst<->best boundaries.
- Some reassignment schemes exhibit a higher frequency of occurrence than others, though nothing allows to predict upfront which reassignment scheme is to happen, despite the same experimental conditions amongst runs[30].

[30] For instance, we could observe the reassignment of 1,2 or 3 Pods, on 1,2 or 3 Nodes, sometimes even back on the original node (Node4).

6 Discussion and future work

While successfully meeting the service time improvement objective, the dynamic rescheduling system proposed in this work would benefit from the hereafter listed improvements and extensions:

- As observed in the low load test case (see 5.3.2) a same cost can be achieved by different rescheduling schemes, some of which require fewer reassignments and consequently generate a lower impact on the overall cluster stability during the rescheduling phase. By extension, the trade-of between the contribution of a reassignment to the cost improvement and its impact on overall cluster stability during the rescheduling phase should be further studied. Besides punctual/opportunistic tuning (limiting the number of Pods that can be rescheduled per iteration (possibly with a prioritisation mechanism), increasing the delay between patching commands emission, etc.), cooperative models represent a more comprehensive opportunity to address this key aspect.
- Batch automation of the experimental runs would allow for a more exhaustive coverage of the possible re-assignment schemes for the defined scenarios as well as for further investigating the performance and adequacy of the model for large scale scenarios and consequently for more robust conclusions.
- CPU power and network throughput awareness represents an additional direction for further investigation, since it would allow to i) alleviate the assumption of CPU power and network connectivity equivalence among cluster nodes and ii) take their exact contribution to service-time into consideration (and balance the model consequently). Incorporating energy efficiency related parameters within the model represents another desirable evolution.
- Instead of using the aggregated average resource consumption for a given period in the past, use its complete time-series (for the same given period) as a resource consumption signature for a more fine-grained consolidation.
- Based on past values, enrich the model with resource consumption prediction.
- Implementing the *Reflective Learning* component would allow the analysis of the impact the rescheduling decision has on the cluster over time and to consequently adapt the parameters of the *Current Context Modeler*, *New Context Generator* and *Decision Maker* components in order to continuously improve the performance of the rescheduling system.

7 Conclusion

This article proposes a portable dynamic rescheduling system for container orchestration platforms that aims at improving application service time by minimizing inter-container network delays and server CPU overload. To this end, a closed control loop system monitors not only resource consumption and availability but also container inter-dependency in terms of application network traffic. Periodically, the system assesses if alternative assignments may allow network traffic and CPU overload reduction. If the best alternative sufficiently improves the current assignment scheme, containers are reassigned accordingly. The multiobjective assignment problem

is modeled as a weighted-sum of both cost functions and is solved by means of the SA metaheuristic. A technique of progressive normalization of the cost functions has been developed to eliminate the otherwise too computationally expensive solving process. Additionally, a time-budget approach is proposed to increase the effectiveness of the outcome for the smaller sized clusters and to improve the computing efficiency for the larger sized clusters (at the limited cost of effectiveness). The impact, under varying load, of the proposed system on application service time is evaluated and discussed by means of three concurrent applications. In all tested scenarios the application service time is improved (up to 27.2%) and systematically surpasses the K8s descheduler results. Those promising results should, however, not be considered as the end of the story since various improvements, subject to further work, have been identified and are briefly discussed in Section 6.

Acknowledgment

José Santos is funded by the Research Foundation Flanders (FWO), grant number 1299323N.

Acronyms

API Application Programming Interface. 25, 28, 30, 31, 33

CNCF Cloud Native Computing Foundation. 23, 28
CPU Central Processing Unit. 2, 5–7, 9–11, 13, 18, 19, 24, 29, 30, 32–35, 37
CR Custom Resource. 8

FQDN Fully Qualified Domain Name. 28

GB Gigabyte. 5, 18, 32
Gb Gigabit. 5

HDD Hard-Disk Drive. 5
HTTP Hypertext Transfer Protocol. 30

I/O Input/Output. 2
IoT Internet of Things. 30, 31

JDK Java Development Kit. 5
JSON JavaScript Object Notation. 30

K8s Kubernetes. 2, 5, 8, 9, 23, 24, 26–29, 32, 35, 36, 38
KS Kubernetes Scheduler. 24–26

LTS Long-Term Support. 5

MP Mathematical Programming. 5, 16–18

NIC Network Interface Controller. 5

NP Non-deterministic Polynomial-time. 15

OS Operating System. 2, 5

QAP Quadratic Assignment Problem. 10
QoS Quality-of-Service. 7, 9, 23, 24, 32

RAM Random-Access Memory. 2, 5, 13, 18, 19, 24, 32, 35

SA Simulated Annealing. 15–23, 38
SDK Sotfware Development Kit. 5

TIARM Throttling and Interaction-aware Anticorrelated Rescheduling for Microservices. 9
TSDB Time Series DataBase. 31, 33
TSP Traveling Salesman Problem. 3

URI Uniform Resource Identifier. 30
URN Uniform Resource Name. 30

VMC Virtual Machine Consolidation. 7
VRP Vehicle Routing Problem. 3–6, 29, 30, 32, 34–36

Declarations

Funding. José Santos is funded by the Research Foundation Flanders (FWO), grant number 1299323N.

Competing interests. Filip De Turck, a listed author, is member of the editorial advisory board of this Journal. The authors declare that they have no other competing interests.

Ethical Approval. Not applicable.

Availability of data and materials. The datasets used and analysed during the current study are available from the corresponding author on reasonable request.

Authors' contributions.

Vincent Bracke substantially contributed to the conception and design of the work, to the acquisition, analysis and interpretation of data and to the creation of new software used in the work. He drafted the work and substantively revised it.

José Santos and Tim Wauters substantially contributed to the analysis and interpretation of data. They substantively revised the work.

Filip De Turck and Bruno Volckaert substantially contributed to the conception of the work, as well as to the analysis and interpretation of data. They drafted the work and substantively revised it.

All authors have approved the submitted version.

Authors biography.

Vincent Bracke is a PhD researcher in the Department of Information Technology (INTEC) at Ghent University, in collaboration with imec. He obtained the master's degree in Computer Science from Université Catholique de Louvain (UCL), Belgium, in 2006 and the master's degree in International Business and Management from ICHEC Brussels Management School, Belgium, in 2009. After several years in the private sector where he held various IT management positions, he joined in 2018 the Internet and Data Science Lab (IDLab), a research group of INTEC. His research interests include scalable and reliable software systems for IoT applications and autonomous optimization of distributed resources management.

José Santos obtained his M.Sc. degree in Electrical and computer engineering in July 2015 from the University of Porto, Portugal. Recently, he completed his doctoral studies at Ghent University in April 2022. He is currently a Postdoctoral Researcher in the Internet Technology and Data Science Lab (IDLab) Research Group at Ghent University - imec, Belgium. His research interests include Cloud and Fog Computing, IoT, Service Function Chaining, and Reinforcement Learning. His work has been published in more than 20 scientific publications. He received the PhD Excellence Award from imec in 2022, the Best Dissertation Award at NOMS 2023, and the FWO IBM Innovation Award 2023 based on the research conducted during his PhD about efficient orchestration strategies in Fog Computing.

Tim Wauters obtained his M.Sc. and PhD degrees in electro-technical engineering from Ghent University in 2001 and 2007 respectively. He has been working as a post-doctoral fellow of the F.W.O.-V. in the Department of Information Technology (INTEC) at Ghent University, and is now also active as a senior researcher at imec. His main research interests focus on the design and management of networked services, covering multimedia distribution, cybersecurity, big data and smart cities. His work has been published in more than 160 scientific publications.

Filip De Turck leads the network and service management research group at Ghent University, Belgium and imec. He (co-) authored over 750 peer reviewed papers and his research interests include design of efficient softwarized network and cloud systems. He is involved in several research projects with industry and academia, served as chair of the IEEE Technical Committee on Network Operations and Management (CNOM), and serves as a steering committee member of the IM, NOMS, CNSM and NetSoft conferences. Prof. Filip De Turck served as Editor-in-Chief of IEEE Transactions on Network and Service Management (TNSM), was named an IEEE Fellow in 2020, and received the IEEE ComSoc Dan Stokesberry Award in 2021.

Bruno Volckaert is professor of advanced software engineering and secure distributed systems in the Department of Information Technology at Ghent University and imec's IDLab group. His current research deals with reliable and high performance distributed software for a.o. scalable data ingestion and processing, scalable cybersecurity detection and mitigation architectures and autonomous optimization of

cloud-based applications. He has worked on over 65 national and international research projects and is author or co-author of more than 200 peer-reviewed papers published in international journals and conference proceedings.

References

[1] Silva, V.G., Kirikova, M., Alksnis, G.: Containers for Virtualization: An Overview. Applied Computer Systems **23**(1), 21–27 (2018) https://doi.org/10.2478/acss-2018-0003

[2] Docker, Inc.: Docker website. https://www.docker.com. [Online] (2023)

[3] The Linux Foundation: LXC/LXD website. https://linuxcontainers.org. [Online] (2023)

[4] Red Hat, Inc.: Podman website. https://podman.io/. [Online] (2023)

[5] The Cloud Native Computing Foundation: Containerd website. https://containerd.io. [Online] (2023)

[6] The Apache Software Foundation: Mesos website. http://mesos.apache.org. [Online] (2023)

[7] Docker, Inc.: Docker Swarm website. https://docs.docker.com/engine/swarm/. [Online] (2023)

[8] The Cloud Native Computing Foundation: Kubernetes website. https://kubernetes.io. [Online] (2023)

[9] Kalmbach, P., Zerwas, J., Babarczi, P., Blenk, A., Kellerer, W., Schmid, S.: Empowering Self-Driving Networks. In: Proceedings of the Afternoon Workshop on Self-Driving Networks. SelfDN 2018, pp. 8–14. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3229584.3229587

[10] Bracke, V., Werrebrouck, G., Santos, J.P., Wauters, T., De Turck, F., Volckaert, B.: Online Dynamic Container Rescheduling for Improved Application Service Time. Journal of Network and Systems Management **31**(4) (2023) https://doi.org/10.1007/s10922-023-09766-9

[11] Robinson, J.B.: On the Hamiltonian Game (A Traveling Salesman Problem). RAND Corporation, Santa Monica, CA (1949)

[12] Dantzig, G.B., Ramser, J.H.: The Truck Dispatching Problem. Management Science **6**, 80–91 (1959) https://doi.org/10.1287/mnsc.6.1.80

[13] Lloyd, S.: Least squares quantization in PCM. IEEE Transactions on Information

Theory **28**(2), 129–137 (1982) https://doi.org/10.1109/TIT.1982.1056489

[14] Croes, G.A.: A Method for Solving Traveling-Salesman Problems. Operations Research **6**(6), 791–812 (1958) https://doi.org/10.1287/opre.6.6.791

[15] Reinelt, G.: The Traveling Salesman: Computational Solutions for TSP Applications, 1st ed. 1994. edn. Lecture Notes in Computer Science, vol. 840. Springer, Berlin, Heidelberg (1994). https://doi.org/10.1007/3-540-48661-5

[16] Almeida, R.S.: TSP Essay. https://github.com/rsalmei/tsp-essay. Accessed: 2023-01-20 (2020)

[17] Beloglazov, A., Buyya, R.: Optimal Online Deterministic Algorithms and Adaptive Heuristics for Energy and Performance Efficient Dynamic Consolidation of Virtual Machines in Cloud Data Centers. Concurr. Comput.: Pract. Exper. **24**(13), 1397–1420 (2012) https://doi.org/10.1002/cpe.1867

[18] Mahdhi, T., Mezni, H.: A prediction-Based VM consolidation approach in IaaS Cloud Data Centers. Journal of Systems and Software **146**, 263–285 (2018) https://doi.org/10.1016/j.jss.2018.09.083

[19] Wang, J.V., Cheng, C.-T., Tse, C.K.: A thermal-aware VM consolidation mechanism with outage avoidance. Software: Practice and Experience **49**(5), 906–920 (2019) https://doi.org/10.1002/spe.2680

[20] Zhao, D., Mohamed, M., Ludwig, H.: Locality-Aware Scheduling for Containers in Cloud Computing. IEEE Transactions on Cloud Computing **8**(2), 635–646 (2020) https://doi.org/10.1109/TCC.2018.2794344

[21] Filip, I.-D., Pop, F., Serbanescu, C., Choi, C.: Microservices Scheduling Model Over Heterogeneous Cloud-Edge Environments As Support for IoT Applications. IEEE Internet of Things Journal **5**(4), 2672–2681 (2018) https://doi.org/10.1109/JIOT.2018.2792940

[22] Nanda, S., Hacker, T.J.: RACC: Resource-Aware Container Consolidation Using a Deep Learning Approach. In: Proceedings of the First Workshop on Machine Learning for Computing Systems. MLCS'18. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3217871.3217876

[23] Wen, Z., Lin, T., Yang, R., Ji, S., Ranjan, R., Romanovsky, A., Lin, C., Xu, J.: GA-Par: Dependable Microservice Orchestration Framework for Geo-Distributed Clouds. IEEE Transactions on Parallel and Distributed Systems **31**(1), 129–143 (2020) https://doi.org/10.1109/TPDS.2019.2929389

[24] Guerrero, C., Lera, I., Juiz, C.: Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications. The Journal of Supercomputing **74**(7), 2956–2983 (2018) https://doi.org/10.1007/s11227-018-2345-2

[25] Bittencourt, L.F., Goldman, A., Madeira, E.R.M., da Fonseca, N.L.S., Sakellariou, R.: Scheduling in distributed systems: A cloud computing perspective. Computer Science Review **30**, 31–54 (2018) https://doi.org/10.1016/j.cosrev.2018.08.002

[26] Söylemez, M., Tekinerdogan, B., Tarhan, A.K.: Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review. Applied Sciences (2022) https://doi.org/10.3390/app12115507

[27] Santos, J., Wang, C., Wauters, T., Turck, F.D.: Diktyo: Network-Aware Scheduling in Container-based Clouds. IEEE Transactions on Network and Service Management, 1–1 (2023) https://doi.org/10.1109/TNSM.2023.3271415

[28] Zhou, R., Li, Z., Wu, C.: An Efficient Online Placement Scheme for Cloud Container Clusters. IEEE Journal on Selected Areas in Communications **37**(5), 1046–1058 (2019) https://doi.org/10.1109/JSAC.2019.2906745

[29] Piraghaj, S.F., Dastjerdi, A.V., Calheiros, R.N., Buyya, R.: A Framework and Algorithm for Energy Efficient Container Consolidation in Cloud Data Centers. In: 2015 IEEE International Conference on Data Science and Data Intensive Systems, pp. 368–375 (2015). https://doi.org/10.1109/DSDIS.2015.67

[30] Rattihalli, G.: Exploring Potential for Resource Request Right-Sizing via Estimation and Container Migration in Apache Mesos. In: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), pp. 59–64 (2018). https://doi.org/10.1109/UCC-Companion.2018.00035

[31] Bulej, L., Bureš, T., Hnětynka, P., Khalyeyev, D.: Self-adaptive K8S Cloud Controller for Time-sensitive Applications. In: 2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 166–169 (2021). https://doi.org/10.1109/SEAA53835.2021.00029

[32] Rodriguez, M., Buyya, R.: Container orchestration with cost-efficient autoscaling in cloud computing environments. In: Handbook of Research on Multimedia Cyber Security, pp. 190–213 (2020). https://doi.org/10.4018/978-1-7998-2701-6.ch010

[33] Wojciechowski, L., Opasiak, K., Latusek, J., Wereski, M., Morales, V., Kim, T., Hong, M.: NetMARKS: Network Metrics-AwaRe Kubernetes Scheduler Powered by Service Mesh. In: IEEE INFOCOM 2021 - IEEE Conference on Computer Communications, pp. 1–9 (2021). https://doi.org/10.1109/INFOCOM42981.2021.9488670

[34] Marchese, A., Tomarchio, O.: Network-Aware Container Placement in Cloud-Edge Kubernetes Clusters. In: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 859–865 (2022). https:

//doi.org/10.1109/CCGrid54584.2022.00102

[35] Joseph, C.T., Chandrasekaran, K.: Nature-inspired resource management and dynamic rescheduling of microservices in Cloud datacenters. Concurrency and Computation: Practice and Experience **33**(17), 6290 (2021) https://doi.org/10.1002/cpe.6290

[36] Podzimek, A., Chen, L.Y.: Transforming System Load to Throughput for Consolidated Applications. In: 2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 288–292 (2013). https://doi.org/10.1109/MASCOTS.2013.37

[37] Arora, J.S.: Chapter 18 - Multi-objective Optimum Design Concepts and Methods. In: Arora, J.S. (ed.) Introduction to Optimum Design (Fourth Edition), Fourth edition edn., pp. 771–794. Academic Press, Boston (2017). https://doi.org/10.1016/B978-0-12-800806-5.00018-4

[38] Caramia, M., Dell'Olmo, P.: Multi-objective Optimization. In: Multi-objective Management in Freight Logistics: Increasing Capacity, Service Level, Sustainability, and Safety with Optimization Algorithms, pp. 21–51. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50812-8_2

[39] Grodzevich, O., Romanko, O.: Normalization and other topics in multi-objective optimization. In: Proceedings of the Fields-MITACS Industrial Problems Workshop, (2006)

[40] Emmerich, M., Deutz, A.: A tutorial on multiobjective optimization: fundamentals and evolutionary methods. Natural Computing **17** (2018) https://doi.org/10.1007/s11047-018-9685-y

[41] Kirkpatrick, S., Gelatt, C., Vecchi, M.: Optimization by Simulated Annealing. Science (New York, N.Y.) **220**, 671–680 (1983) https://doi.org/10.1126/science.220.4598.671

[42] Cerny, V.: Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. Journal of Optimization Theory and Applications **45**, 41–51 (1985) https://doi.org/10.1007/BF00940812

[43] Koulamas, C., Antony, S., Jaen, R.: A survey of simulated annealing applications to operations research problems. Omega **22**(1), 41–56 (1994) https://doi.org/10.1016/0305-0483(94)90006-X

[44] Connolly, D.T.: An improved annealing scheme for the QAP. European Journal of Operational Research **46**(1), 93–100 (1990) https://doi.org/10.1016/0377-2217(90)90301-Q

[45] Fidanova, S.: Simulated Annealing for Grid Scheduling Problem. In: IEEE

John Vincent Atanasoff 2006 International Symposium on Modern Computing (JVA'06), pp. 41–45 (2006). https://doi.org/10.1109/JVA.2006.44

[46] Ellison Geltman, K.: The Simulated Annealing Algorithm. http://katrinaeg.com/simulated-annealing.html (2014)

[47] Flexera: 2023 State of the Cloud Report. https://info.flexera.com/CM-REPORT-State-of-the-Cloud (2023)

[48] SIGs, K.: Kubernetes concepts. https://kubernetes.io/docs/concepts/. Accessed: 2023-06-22 (2023)

[49] Santos, J., Wauters, T., Volckaert, B., De Turck, F.: Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing Applications. In: 2019 IEEE Conference on Network Softwarization (NetSoft), pp. 351–359 (2019). https://doi.org/10.1109/NETSOFT.2019.8806671

[50] Kubernetes SIGs: Descheduler for Kubernetes. https://github.com/kubernetes-sigs/descheduler. Accessed: 2023-06-15 (2023)

[51] pixielabs.ai: Pixie Overview. https://docs.pixielabs.ai/about-pixie/what-is-pixie. Accessed: 2023-10-06 (2023)

[52] google.com: GoogleCloudPlatform microservices-demo: Online Boutique. https://github.com/GoogleCloudPlatform/microservices-demo. Accessed: 2023-10-10 (2023)

[53] Bracke, V., Sebrechts, M., Moons, B., Hoebeke, J., De Turck, F., Volckaert, B.: Design and evaluation of a scalable Internet of Things backend for smart ports. Software: Practice and Experience **51**(7), 1557–1579 (2021) https://doi.org/10.1002/spe.2973

[54] Yellin, N.: For the love of god, stop using CPU limits on Kubernetes (updated). https://home.robusta.dev/blog/stop-using-cpu-limits. [Online; accessed 10-August-2023] (2022)