RESEARCH

Online dynamic container rescheduling for improved application service time

Vincent Bracke*, Gillis Werrebrouck, José Santos, Tim Wauters, Filip De Turck and Bruno Volckaert

Abstract

Despite their maturity and robustness, container orchestration platforms still suffer from some limitations. One of those concerns the lack of runtime adaptability of the scheduler to the overall cluster status as i) it instantiates containers with local optimization in mind i.e. it only considers the container-specific predefined requirements which may lead to a sub-optimal overall cluster state and ii) it does not reshuffle the deployed containers at runtime based on container observed behavior and interdependencies. These limitations become even more apparent within volatile load contexts. This work proposes an autonomous and dynamic rescheduling system that aims at improving application service time by co-locating highly interdependent containers for network delay reduction. To this extend, two distinct combinatorial optimization heuristics, Simulated Annealing and Particle Swarm Optimization, are evaluated and compared on their respective effectiveness and efficiency as well as on their relative performance towards the optimal solution obtained by Integer Linear Programming. Additionally, the impact of the proposed system on application service time is validated by means of two complementary use-cases, an event-based IoT data-hub platform and a web-based e-commerce app, with an average improvement of the end-to-end service time of 21.6% and 13.1% respectively.

Keywords: Cloud Computing Services; Performance Management; Autonomic and Cognitive Management; Orchestration; Mathematical Optimization; Optimization Theories

1 Introduction

Microservice software architectures promote 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, microservice oriented architectures 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 Central Processing Unit (CPU), Random-Access Memory (RAM), disk I/O and network. It acts as an abstraction at the application layer that packages code and dependencies (application code, runtime, system libraries, settings, etc.). Multiple containers can thus simultaneously run on the same machine and share the 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. As much as they considerably improve the deployment process, containers by themselves do not make the management of applications easier. Therefore, container orchestration platforms have been developed to help manage containerized applications running on distributed clusters. Main such platforms comprise Apache Mesos [6], Docker Swarm [7] and Kubernetes (K8s)[8] among others. These platforms offer scalability, availability management, monitoring tools, networking functionalities, container orchestration, etc. for the containerized application. All those platforms schedule individual containers on the most appropriate server within a cluster based on server resource availability and container resource needs. Additional parameters are taken into account like the deployment strategy as well as predilection and aversion constraints. Nevertheless, despite their maturity and robustness, those container orchestration platforms still suffer from some limitations as they lack runtime adaptability to overall cluster sta-

*Correspondence: vincent.bracke@ugent.be
IDLab, Department of Information Technology, Ghent University - imec, Technologiepark-Zwijnaarde 126, B-9052, Ghent, Belgium
Full list of author information is available at the end of the article

tus. More specifically, main weaknesses addressed in this work are hereafter introduced:

- Container deployment requests are individually enqueued as they arrive (online scheduling); the scheduler only considers container-specific requirements in FIFO order and not as a set of constraints to be optimized, which may lead to a sub-optimal overall cluster state.
- Consequently, the scheduler does not reconfigure or adapt the distribution of containers amongst cluster servers at runtime based on the observed container behavior and interdependencies.

These limitations restrain cluster wide optimisation of resource allocation and consequently affect its performance. We therefore propose an autonomous rescheduler that periodically reassesses the distribution of containers among servers in order to optimize the cluster state and mitigate unwanted effects of evolving load within a cluster. Optimization of the cluster state depends on the goal given to the rescheduler. This goal may be expressed as an objective function to be minimized or maximized. For instance, if the goal is to minimize the infrastructure cost, the rescheduler will have to find the best distribution that allows for a minimal number of servers. In contrast, if the goal is to fairly balance load on a fixed infrastructure, the rescheduler might seek to distribute containers based on even resource consumption among servers. Lastly, if the goal is to optimize application service time (with possibly distinct Quality-of-Service (QoS) levels) in a fixed infrastructure, the rescheduler will have to find the best distribution that allows for minimal inter-server network traffic and consequently regroup containers that are strongly interdependent (i.e.: that have high network traffic among them). Indeed, containers that communicate with one another are preferably co-located on the same server for minimal network traffic as communication between different servers substantially increases overall application latency.

This article presents a self-driving rescheduling system that is capable of reallocating containers within a distributed cluster to improve overall application service time by co-locating interdependent containers based on their observed network traffic while still taking other constraints into consideration (e.g. (anti-) affinities as well as available and required resources). Firstly, both the effectiveness and efficiency of the proposed rescheduling system are demonstrated. Secondly, the impact of the rescheduling on application service time is empirically validated by means of two complementary use-cases: an event-based Internet of Things (IoT) data-hub platform and a web-based e-commerce app. To this end, both Docker (as container technology) and K8s (as container orchestration platform) have been chosen in this work for their ease of use, high maturity and dominant market position[1, 9].

The remainder of the article is organized as follows: section 2 summarizes the current state of the art in the domain, after which section 3 presents three different combinatorial optimization techniques together with an analysis and comparison of their respective 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 4 and then validated by means of two complementary use-cases. Section 5 identifies and discusses improvement and extension opportunities and, finally, section 6 provides concluding remarks.

2 Related Work

This section focuses on the current state of research in the field of resource scheduling for cloud applications rather than on metaheuristics used to solve such NP-hard combinatorial optimization problem. However sub-section 3.3 further frames the latter by means of additional references.

Resource scheduling can be described as a decision-making process, used in many manufacturing and services industries, that deals with the allocation of resources to tasks over given time periods with the goal of optimizing one or more objectives [10]. From the early days of Information Technology (IT), scheduling techniques have been intensively used and developed first on standalone computers, most commonly to minimize task completion time. With the rise of computer networks, clusters (of homogeneous computers) followed by grids (of heterogeneous devices) and more recently the cloud (offering virtualized computing resources) acted as a multiprocessor computer with distributed data sources. With a slower communication channel between processors when compared to supercomputers, task scheduling in distributed systems eventually bloomed as a specific branch of research where different optimization objectives ballooned the scheduling literature in the past decade [11]. The authors of [11] 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, Söylemez et al. [12] 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.

While the literature on resource scheduling for the data center initially focused on Virtual Machine Consolidation (VMC) mostly to optimize the performance-energy tradeoff [13–15], it progressively evolved, with the rise of 'containerization' and microservices architecture, to address the issue of scheduling for the containerized application considering various factors such as load balance-application performance tradeoff [16], heterogeneity of resources [17], classical bin-packing [18], network QoS [19] and network latency introduced by inter-microservice communication [20]. These works do however only consider the initial allocation of containers and do not consider their rescheduling.

Piraghaj et al. [21] propose a framework for energy efficient container rescheduling in cloud data centers, evaluated by means of simulation only. Furthermore, 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 [22] proposes 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 [23], 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 [24], 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 [25], the authors propose an efficient online algorithm that optimizes container placement based on resource prices, while taking inter-container traffic into consideration. Besides its theoretical nature (backed by trace-driven simulations), the presented analysis diverges from our work as i) it focuses on initial container placement and ii) it requires the presetting of the traffic demand between containers.

In [26], the authors propose NetMARKS, 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-nodes traffic by considering applications individually one at a time, possibly leading to sub-optimal overall cluster status. A comparable approach is also proposed in [27].

Lastly, Joseph and Chandrasekara [28] propose a microservice rescheduling framework, Throttling and Interaction-aware Anticorrelated Rescheduling for Microservices (TIARM), to proactively perform rescheduling activities whilst 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[1]. 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, missing opportunities for smaller intermediary adjustments that could maintain the cluster state closer to the optimum state.

- **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, which solely seeks to minimize network traffic among servers, i.e. to consolidate containers on the same server based on the data volume they exchange.

3 The dynamic rescheduling algorithm

3.1 Problem formulation: the wedding seating chart problem

Combinatorial optimization problems are usually expressed by means of concrete use-cases helping the reader to correctly apprehend the problem formulation and the associated challenges, e.g. the knapsack problem, the traveling salesman problem, the cutting stock problem. Likewise, we refer to the wedding seating chart problem as a metaphor to the optimization problem at stake in this work. Though, if most brides and grooms struggle with the complexity of this headache, it can paradoxically be summarized quite simply as: “maximizing guest satisfaction”. This can be achieved by placing guests with people they enjoy the company of and, inversely, avoid as much as possible disliking groups. What makes this problem complex lays in its combinatorial nature and the following set of constraints:

- All guests must have one seat (and, as corollary, can only be seated at one table).
- There is a limited (and possibly variable) number of seats per table.
- Possibly, some guests must, or must not, seat at the same table.
- Possibly, some guests must, or must not, seat at a specific table.

Assuming that the level of affinity among each guest can be quantified in a relationship matrix, the best possible arrangement would be obtained as the highest possible score, summing the affinities of guests sharing a table and this for each table, while violating none of the above mentioned constraints. To a certain extent, the wedding seating chart problem can thus be considered as an extension of the multi-parameter Quadratic Assignment Problem (QAP) [29], expanding it with (anti-) affinity constraints. The dynamic rescheduling of containers to servers is pretty similar to the wedding seating chart problem where guests are containers, tables are servers and mutual relationships would be the network interdependency between containers. Table capacity is represented by both RAM and CPU capacity of a server (instead of simply the number of seats of a table), with containers requiring a slice of each (and not simply 1 seat as guests would). Inter-container (anti-) affinity constraints correspond to guests having to (not) sit together at the same table, while server (anti-) affinity constraints match the (non-) assignment of guests to specific tables.

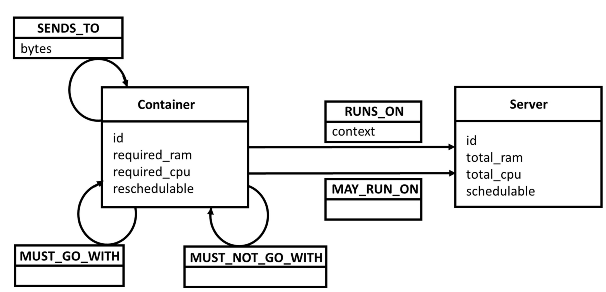

Figure 1 The labeled-property graph model for the dynamic rescheduling of containers to servers

More formally, Figure 1 illustrates by means of a labeled-property graph the data model for the problem at stake, where:

- ‘MUST_GO_WITH’ optional relationship represents container affinity constraints. Container anti-affinity constraints are represented by the optional relationship ‘MUST_NOT_GO_WITH’. As for the ‘SENDS_TO’ relationship, they describe relationships among containers with, for the latter, the measured number of sent bytes as property.

[1] 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 time.

- ‘MAY_RUN_ON’ relationship represents the server (anti-) affinity constraint. It links a container to the set of servers it possibly may run on.
- ‘RUNS_ON’ relationship specifies the hosting server for each container. When dynamically rescheduling containers, it is important to identify both, the server a container is currently running on as well as the target server the container shall be running on after the rescheduling; the recording of this information is done by means of the ‘context’ property of the relationship.

Following sub-sections propose different approaches to solve this problem firstly by means of the Integer Linear Programming (ILP) exact optimization method and afterwards by means of Simulated Annealing (SA) and Particle Swarm Optimization (PSO), two metaheuristics optimization techniques. Container rescheduling being a NP-hard problem [28], the use of metaheuristics allows to reach near optimum solution in a reasonable time.

3.2 Problem modelization by means of ILP

Table 1 introduces the variables and parameters of the model. Most of those are self-explanatory, though the three last parameters require more introductory explanation. Firstly, a_n^p allows the model to take the server (anti-) affinity constraints into account. There are three possibilities:

- 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 unavailable servers or for servers dedicated to the management of the cluster and not to application hosting.
- Server affinity is defined : in this case, this parameter equals 1 for all schedulable servers matching the defined affinity, else 0.
- Server anti-affinity is defined : in this case, this parameter equals 1 for all schedulable servers not matching the defined anti-affinity, else 0.

Secondly, l^{pq} allows the model to take inter-container affinity into account. If such an affinity is defined for 2 containers then the parameter equals 1, else 0. Finally, d^{pq} allows the model to take inter-container anti-affinity into account. If such an anti-affinity is defined for 2 containers, then the parameter equals 1, else 0.

Equation (1) defines the objective function of the model; in this case, the model seeks to minimize inter-server traffic.

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

Table 1 ILP formulation

Variable	Description
n,m	server number, from 1 to N, where N is the total amount of servers
p,q	container number, from 1 to P, where P is the total amount of containers
r_n^p	= 1 if container p runs on server n, 0 otherwise (binary decision variable)
s_n^{pq}	= 0 if containers p and q are both hosted on server n or if none of them is, else 1.
Parameter	**Description**
v_n	CPU capacity (units) of server n
x_n	RAM capacity (bytes) of server n
w^p	CPU requirement (units) of container p
y^p	RAM (bytes) requirement of container p
t^{pq}	network traffic (bytes) between containers p and q
a_n^p	=1 if container p may be instantiated on server n, else 0
l^{pq}	=1 if container p and q must be co-located, else 0 (inter-container affinity)
d^{pq}	=1 if container p and q must not be co-located, else 0 (inter-container anti-affinity)

This is however subject to the following constraints:

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

$$\sum_{n=1}^{N} r_n^p = 1, \forall p \tag{2}$$

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

$$\sum_{p=1}^{P} r_n^p w^p \leq v_n, \forall n \tag{3}$$

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

$$\sum_{p=1}^{P} r_n^p y^p \leq x_n, \forall n \tag{4}$$

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

$$\sum_{n=1}^{N} r_n^p a_n^p = 1, \forall p \tag{5}$$

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

$$1 - l^{pq} \geq r_n^p - r_n^q, \forall n, p, q \neq p \tag{6}$$

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

$$2 - d^{pq} \geq r_n^p + r_n^q, \forall n, p, q \neq p \tag{7}$$

- Eq.(8) and Eq.(9) ensure that inter-container traffic is taken into account when containers p and q are not co-located. Those two equations ensure the linearization of $s_n^{pq} = |r_n^p - r_n^q|$:

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

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

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

$$r_n^p, s_n^{pq} \in \{0, 1\} \tag{10}$$

3.3 Introduction of the selected metaheuristics

There exists a wide range of metaheuristics that are used to solve combinatorial optimisation problems. They can be classified into two main categories: trajectory methods and population-based methods. This categorization permits a clearer description of the algorithms[2]. Trajectory methods all share the property of describing a trajectory in the search space during the search process while population-based metaheuristics, on the contrary, perform search processes which describe the evolution of a set of points in the search space. Trajectory-based search algorithms include among others Tabu Search (TS)[31], Iterated Local Search (ILS)[32], Variable Neighborhood Search (VNS)[33], Greedy Randomized Adaptive Search Procedures (GRASP)[34] and Simulated Annealing (SA)[35, 36]. Examples of population-based search algorithms are Genetic Algorithms (GAs)[37], Honey-Bees Mating Optimization (HBMO)[38], Particle Swarm Optimization (PSO)[39] and Ant Colony Optimization (ACO)[40].

The benchmark realized in subsection 3.5 compares SA, a trajectory metaheurstic, with PSO, a population-based metaheuristic for solving the problem of dynamic container rescheduling. SA has been chosen as trajectory metaheuristic for its simplicity of implementation and as it has successfully been applied to a wild variety of combinatorial optimization problems [41, 42] among which Grid-Computing Scheduling [43]. PSO has been selected as population-based metaheuristic since it has relatively few parameters, exhibits a good ability of global searching, has been successfully applied to many areas [44] and, more particularly, has demonstrated good results in solving scheduling problems in distributed grid systems outperforming other population-based metaheuristics in terms of solution quality and convergence time [45, 46].

[2] Moreover, a current trend is the hybridization of methods in the direction of the integration of trajectory methods in population-based ones [30].

3.3.1 Simulated Annealing (SA)

Simulated Annealing (SA) is a probabilistic method proposed by Kirkpatrick et al. [35] and Cerny [36] for finding the global minimum of a cost function that may possess several local minima. Based on an analogy to the statistical mechanics of annealing in solids it emulates the physical process whereby a solid is slowly cooled so that when its structure is eventually "frozen", this happens at a minimum energy configuration [47].

Algorithm 1 The SA algorithm as used for the wedding seating chart problem

```
procedure ANNEALING(sol, t, stop, iter, α)
    cost_o ← COST(sol)
    while t > stop do
        for i ← 1, iter do
            sol_new ← RANDVALIDNEIGHBORSOL(sol)
            cost_n ← COST(sol_new)
            if PROBACCEPT(cost_o, cost_n, t) > RAND(0, 1) then
                sol ← sol_new
                cost_o ← cost_n
            end if
        end for
        t ← t × α
    end while
    return sol, cost_o
end procedure
```

The SA algorithm (see Algorithm 1) may be summarized as follows:

1. Generate a random solution (see sub-section 3.4.3).
2. Calculate its cost (see sub-section 3.4.1).
3. Generate a random neighboring solution (see sub-section 3.4.2).
4. Calculate the new solution's cost (see sub-section 3.4.1).
5. Compare previous and new solution costs:
    - If $cost_n < cost_o$: 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 $cost_n \geq cost_o$: 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 pa-

rameter 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 the α constant (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 [48].

6 Repeat steps 3-5 above until an acceptable solution is found or some maximum number of iterations is reached.

Based on the $cost_o$, $cost_n$ and temperature, the acceptance probability is calculated by means of Equation (11) 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:

$$a = \min(1, e^{\frac{cost_o - cost_n}{T}}) \tag{11}$$

where a is the acceptance probability, $cost_o - cost_n$ is the difference between the old 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"*[48].

3.3.2 Particle Swarm Optimization (PSO)

Proposed by Kennedy and Eberhart [39], the Particle Swarm Optimization (PSO) metaheuristic is an algorithm used to search for an optimal solution in a n-dimensional solution space. The underlying bio-inspired reasoning has been aroused by the observation of animal swarms (flocks of birds, schools of fish, etc.) moving in groups where "*individual members of the school can profit from the discoveries and previous experience of all other members of the school during the search for food*"[39].

In PSO, a particle is an individual entity that has a position (in n-dimensions) and a velocity and keeps track of its best position found so far. The movement of particles within the n-dimensional search space is thus governed by their individual velocity, current position, personal best as well as the global best position. A particle's position represents a candidate solution which value is expressed as a fitness value that is measured over an objective function and represents how good or bad a particle's position is. This mechanism progressively guides the movement of the swarm by attracting the particles to positions of high fitness. The best solution gets iteratively improved and eventually converges to a high quality solution.

The two equations which are used in PSO are velocity update (12) and position update (13) equations. These are to be modified in each iteration of PSO algorithm to converge to the optimum solution. For a n-dimensional search space, the position of the i^{th} particle of the swarm is represented by a n-dimensional vector, $Pi = (P_{i1}, P_{i2}, ..., P_{in})^T$. The velocity of this particle is represented by another n-dimensional vector $Vi = (V_{i1}, V_{i2}, ..., V_{in})^T$. The previously best visited position of the i^{th} particle is denoted as $Bi = (B_{i1}, B_{i2}, ..., B_{in})^T$. G_{best} is the index of the best particle in the swarm so far. The velocity of the i^{th} particle is updated using Eq. (12) and the position is updated using Eq. (13) where $d = 1, 2...n$ represents the dimension and $i = 1, 2, ..., s$ represents the particle index with s being the size of the swarm. Constants $c1$ and $c2$ are called cognitive and social scaling parameters respectively and $r1$, $r2$ are random numbers drawn from a uniform distribution.

$$V_{id}(t+1) = V_{id}(t) + c1r1(B_{id} - P_{id}) + c2r2(B_{gd} - P_{id}) \tag{12}$$

$$P_{id}(t+1) = P_{id}(t) + V_{id}(t+1) \tag{13}$$

The PSO algorithm proceeds as follows [49]:

1 Particles' velocities and positions are initialised randomly. For each particle, best visited position is set to current position. G_{best} references the particle with best fitness value.

2 Particles' velocities and positions are updated according to Eq. (12) and (13).

3 For each particle, if the current fitness of the particle is better than its previous best fitness value, then B_i is updated to the current position P_i.

4 G_{best} is updated if the current best fitness of the whole swarm is fitter.

5 Steps 2–4 are repeated until stopping criteria (usually a predefined number of iterations and/or a quality threshold for objective value) are met.

Shi and Eberhart [50] introduced in 1998 the concept of Inertia Weight, a constant that would provide balance between exploration and exploitation during the search process. The Inertia Weight determines the contribution rate of a particle's previous velocity to its velocity at the current time step. The resulting velocity update equation then becomes:

$$V_{id}(t{+}1) = w{*}V_{id}(t){+}c1r1(B_{id}{-}P_{id}){+}c2r2(B_{gd}{-}P_{id}) \tag{14}$$

A large Inertia Weight facilitates a global search (exploration) while a small Inertia Weight facilitates a local search (exploitation). Various strategies have been proposed to dynamically adjust the Inertia Weight during the course of the run [51]. Commonly, it is decreased linearly so that the search effort is mainly focused on exploration at initial stages and is focused more on exploitation at latter stages of the run.

In parallel, variations of PSO have been introduced to allow it to cover discrete problems; main ones being discussed and compared in [49]. Though despite the relative success of those Discrete PSO (DPSO) approaches, "*in a discrete space, when lacking continuity, the movement, the velocity and inertia ideas lose sense*"[52]. García and Moreno-Pérez [52] developed thus a new DPSO technique for discrete optimization : the Jumping Frogs Optimization (JFO) (also referred to as Jumping Particle Swarm Optimization (JPSO)). It works without these components but keeps the concept of attraction by the best positions. So instead of velocity and inertia, the authors considered a random component in the movement of particles; that now has the form of jumps. The position of the particles is updated similarly to the velocity update in the canonical PSO (see Eq. (14)), except that weights of the update equation are now interpreted as probabilities of the movement of a particle towards its attractors whereas improvement. The update equation of particles position, where $c1$, $c2$, $c3$ and $c4$ are the probability values of the movement of the particles towards their corresponding attractors, is given by:

$$P_i(t+1) = c1(P_i(t)) \oplus c2(B_i) \oplus c3(N_i) \oplus c4(G) \tag{15}$$

The result of this operation consists of making random moves (see sub-section 3.4.2) with probability $c1$, approaching moves towards the best position of the own particle B_i with probability $c2$, towards the best position of its social neighbourhood N_i with probability $c3$, or towards the best global position G with probability $c4$. Those approaching moves are not exactly similar to random moves as they require to move in the direction of another solution. To this end, the difference between two assignment schemes, the particle current position and the attractor's position, is obtained by listing all individual 'container-to-server' assignment that differ among both positions. After this, a randomly chosen possible re-assignment is performed: this corresponds to a move towards the attractor. Concretely, it consists in moving an aggregated set of containers from the server it is assigned to (within a particle's position) to the server it is assigned to in the attractor particle's position. This re-assignment is only possible if no container's anti-affinity or hosting capability constraint in the attractor's particle position gets violated by this move.

For the probability values, the unit interval [0, 1] is divided into four segments with lengths $c1$, $c2$, $c3$ and $c4 = 1-(c1+c2+c3)$. Then a random number is generated with uniform distribution in [0, 1] and based on the segment to which the resulting random number belongs, random improvement movements are applied to the position of the particle towards the corresponding attractor. The moves that do not produce improvement are rejected. JPSO has successfully been applied to various combinatorial optimization problems, outperforming classical DPSO techniques, among other when applied to the set covering problem [53], to the vehicle routing problem [54] and to the minimum labelling Steiner tree problem [55].

3.4 Commonalities in the design and implementation of both metaheuristics

3.4.1 The cost function

For performance reasons, the cost function for a specific context (called *assignment* in the SA algorithm and *Particle* in the PSO algorithm) is only executed once at initialization phase. Afterwards, only the delta caused by a move is added to or subtracted from the context cost. This allows for constant time complexity to update the cost of a context instead of $O(c^2)$ where c is the number of containers. Assuming *ContainerC* is moved from *ServerA* to *ServerB*, the delta is then obtained as the difference between the sum of bytes exchanged between *ContainerC* and the other containers running on *ServerA* and the sum of bytes exchanged between *ContainerC* and the other containers running on *ServerB*.

3.4.2 The chain of affinities and the random move function

SA and PSO both make use of a random move function that generates a new solution differing from the current one by one element. This is achieved by randomly selecting a reschedulable container and moving it onto another server that can host it. To this end, both the required memory and processing power of the container need to be known as well as the remaining memory and processing power of each server. Additionally, in order to keep the cluster state compliant with (anti-) affinity constraints, the complete process requires additional steps to be executed. More specifically, if a server affinity constraint is defined for a container, it will only be able to move to servers matching this constraint and, conversely, if a server anti-affinity constraint is defined for a container the servers matching this anti-affinity constraint will not be allowed to host the container. Likewise, if a container affinity constraint is defined for a container, containers matching this affinity will move along with it and inversely if a container anti-affinity constraint is defined for a container it will not be allowed to move to a server hosting containers matching that constraint. The interpretation of those constraints is summarized in Table 2.

Table 2 Interpretation of constraints (colour code from Figure 2)

	Server	Container
Affinity	Must run on	Must be co-located
Anti-Affinity	Must not run on	Must not be co-located
N/A	May run on	May be co-located
CPU/RAM	Remaining	Required

Due to the cumulative nature of those constraints, container scheduling may rapidly become complex. Therefore the 5-steps process, hereafter summarized and exemplified in Figure 2, has been designed to carefully identify the 'targetable' servers for a container to be moved onto:

1 **The identification of the container's chain of affinities:** starting from the chosen container to move, a Directed Acyclic Graph (DAG) representing the container (anti-) affinities is recursively constructed. The first step consists in moving upward the affinities paths until reaching vertices with no predecessor, called the roots hereafter. Then, for each of those roots, the chain is constructed downward until all vertices with no successor are found, hereafter called the leaves. While going downward from roots to leaves, all anti-affinity direct predecessors are identified for all vertices. In the Figure 2 example, starting from container γ (assumed to be the random container to move - see blue circle (1)) the procedure will identify after two affinity upward hops the root vertex, i.e. the container α as it does not have any affinity predecessor. Moving downward, it will identify container β on an affinity edge (plain green line) as well as container η on an anti-affinity edge (dotted red curve) to container β at the first downward hop. At the second downward hop, both containers γ and θ are identified on an affinity and anti-affinity edge respectively. Lastly, container δ is identified on an affinity edge at hop 3. The process stops here as all leaves have been identified. Importantly, the (anti-) affinity definition is assumed to be both acyclic and non adversarial.

2 **The aggregation of the container's chain of affinities:** the full chain of container affinities is then aggregated as a virtual set of containers. Continuing with the example in Figure 2, the virtually aggregated container represents the set composed of containers α, β, γ and δ where the quantity of memory and processing power required by the set is obtained by summing respective requirements for all member containers of the set (550MB = 100MB + 150MB + 200MB + 100MB and 0.7CPU = 0.1CPU + 0.2CPU + 0.3CPU + 0.1CPU). The set has no container affinity relationship to external containers as the entire chain belongs to the set. Conversely, the container anti-affinities to and from any member of the set do not belong to it and stay unchanged. When server affinities are defined for a container, the most restrictive subset dominates, i.e. server affinities of container α being less restrictive than β, the aggregated container inherits server affinities of the latter. At this stage of the process, servers 1,2,3,4,5 and 8 are possible hosting candidates. If there is only one hosting candidate, it means that no alternative exists; the process stops here as the aggregated container is not movable. With unchanged constraints, the now constructed chain of affinities remains fixed and consequently does not need to be re-computed at each random move request.

3 **The servers anti-affinities filtering:** the servers matching anti-affinities of the aggregated container are removed from the selection. In the example, server 5 is thus removed from potential candidates (server 6 was already not part of them). If no alternative exists the process stops here as the aggregated container is not movable in the current cluster context. With unchanged constraints, the now constructed set of potential candidate servers remains fixed and consequently does not need to be re-computed at each random move request.

4 **The containers anti-affinities filtering:** the servers hosting the containers matching anti-affinities (predecessor or successor) are removed from the selection. In the example only server 3 is removed from potential candidates (server 6 was already not part of them). If no alternative exists the process stops here as the aggregated container is not movable in current cluster context.

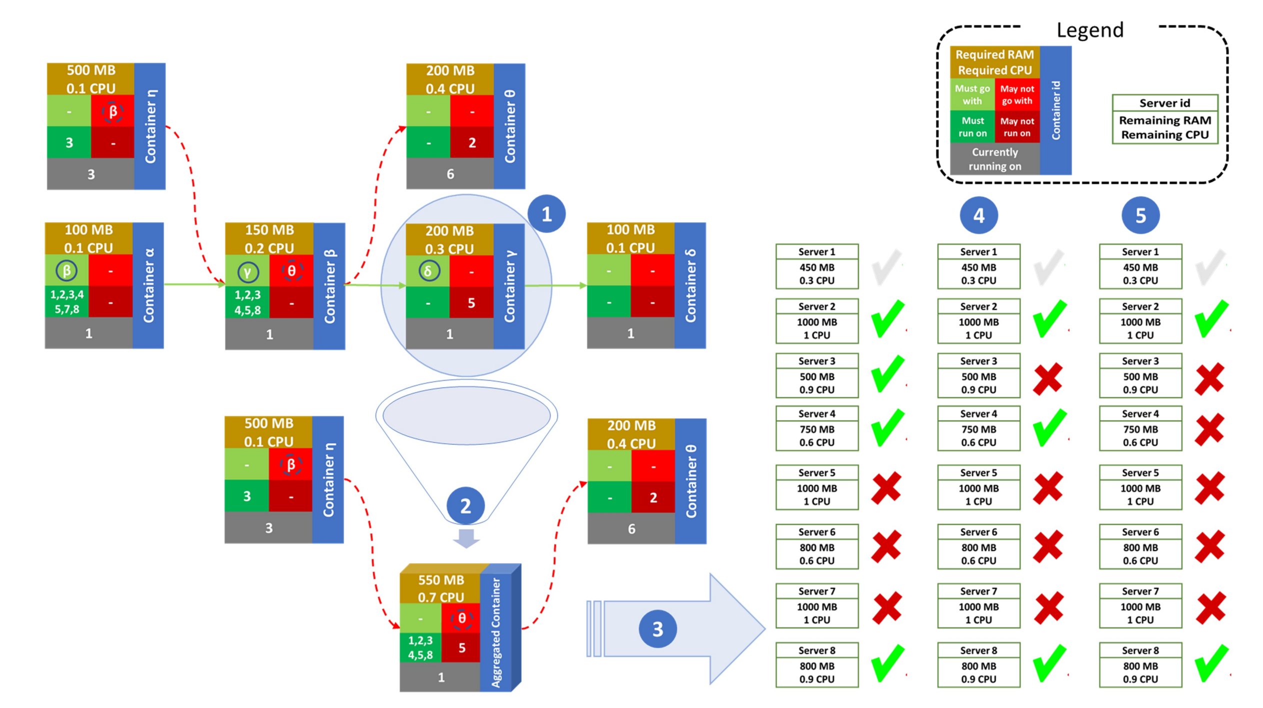

Figure 2 Constraints based identification of candidate target servers for a container and its chain of affinities

5 **The servers remaining capabilities filtering:** lastly the resource requirements are compared to the resources available. Servers with not enough resources are thus removed from the selection which is the case of server 4, in the example, that while still having enough memory capacity, is lacking processing power. If no alternative exists the process stops here as the aggregated container is not movable in the current cluster context.

At the end of the process, a list of targetable servers is obtained. If the list only contains the server currently hosting the aggregated set of containers, it means that it cannot be moved in the current cluster context and, otherwise, the aggregated container is assigned to the best possible server (other than the current host) according to the cost function, i.e. to the server that allows for the minimal cost. It is worth mentioning that steps 1 to 3 are only performed once at algorithm initialization phase since the information collected in those steps does not evolve while performing moves. Only steps 4 and 5 need to be repeated for each move.

3.4.3 The initial random reshuffling

In order to optimally explore the search space, SA and PSO both start from a randomized initial solution. This random allocation is executed as follows:

1 **Irreducible baseline assignment:** this initial step is executed only once and consists in assigning all non-reschedulable aggregated sets of containers. So, all aggregated sets of containers that have only one possible hosting candidate at the end of step 2 of the process presented in subsection 3.4.2 are assigned to that specific host. This leaves the cluster in its minimal common assignment scheme.

2 **Prioritization:** the list of targetable servers is then computed for all reschedulable aggregated sets of containers that need to be re-assigned (see steps 3 to 5 of the process introduced in subsection 3.4.2) and those are ordered by increasing number of targetable servers.

3 **Assignment:** all reschedulable aggregated sets of containers having the smallest number of targetable servers are then assigned, randomly selected one by one, to one of their candidate hosting servers. If no assignment solution remains for a specific aggregated set of containers, then the process restarts at step 1. Once all top priority aggregated sets of containers have been assigned, the process re-executes step 2 with the remaining aggregated sets of containers.

This process is ensured to eventually terminate as there is at least one possible assignment solution that can meet all constraints, i.e. the previous cluster state.

The execution time of this process though will be highly dependent on the restrictiveness and number of constraints. If execution time is deemed critical, a possible alternative can be achieved by returning the previous cluster state after a predefined delay or number of unsuccessful trials as this solution is part of the set of possible solutions; though this may impact the quality of the solution found by the metaheuristic.

3.5 Heuristics benchmarking

3.5.1 Description of the simulation tests and environment

In order to select the most appropriate algorithm for the implementation of the wedding seating chart problem applied to the dynamic rescheduling of containers within a cluster of servers, their respective efficiency and effectiveness are compared by means of four distinct scenarios: S, M, L and XL ordered by increasing size, as illustrated in Table 3 where column:

- **'#C'** defines the number of containers for each scenario,
- **'#S'** defines the number of servers for each scenario,
- **'#SA'** defines the number of server affinity for each scenario. If a server affinity 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.
- **'#SAA'** defines the number of server anti-affinity for each scenario. If a server anti-affinity 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.
- **'#CA'** defines the number of container affinity for each scenario. If a container affinity 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 for each scenario. If a container anti-affinity 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.

Table 3 The four distinct scenarios used for the algorithm selection

	#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

Additionally, those four scenarios have been tested on two different topologies:

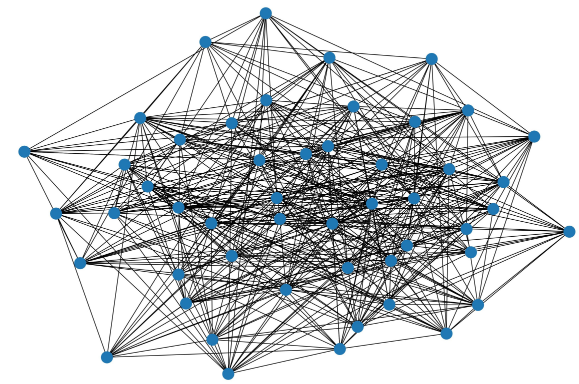

Figure 3 Network connections among containers for the 'S' scenario within the dense topology

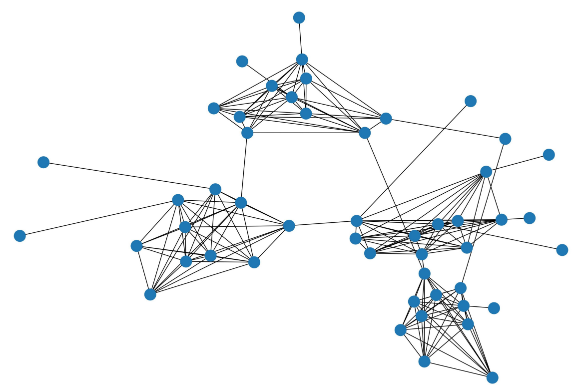

Figure 4 Network connections among containers for the 'S' scenario within the cluster topology

- An unstructured **dense** mesh topology were containers send network traffic to randomly selected containers, as illustrated in Figure 3. This kind of topology, also known as the 'death star' topology, is typically encountered in the context of complex applications split among multiple microservices and hosted onto a dedicated cluster of servers. Examples of such implementation concern, among other, heavy e-commerce applications like the Netflix video streaming platform [56] and the Amazon.com retail website [57].
- A split mesh topology were containers are **clustered** in smaller meshes, possibly exchanging network traffic among them through a limited number of their containers, as illustrated in Figure 4. This kind of topology is typically encountered in the context of multi-tenant clusters of servers where multiple applications co-exist and depict limited interactions among them; most of the network traffic remaining circumscribed to each individual cluster of microservices.

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).
- Individual required RAM/CPU is randomly assigned (within servers max capacity boundaries to allow hosting). However, the total required capacity in terms of RAM/CPU is 60% of the cluster wide available RAM/CPU capacity.
- Inter-container network traffic is randomly assigned on a scale from 1 to 5.

All experiments have been conducted on a server equipped with 2 Hexacore Intel® E5645 (2.4GHz) CPUs and 288GB of RAM and running with the Linux Operating System *Ubuntu 18.04 LTS*. IBM® ILOG® CPLEX® Optimizer v22.1.0 has been used as ILP solver while both heuristics have been developed in Java (JDK 17.0.2).

3.5.2 Effectiveness and efficiency comparison

Both heuristics:

- have been evaluated on eight distinct test-cases; the four different sizes being tested across both topologies. Each distinct test-case has been run twenty-five times to reduce the impact of the inherently stochastic behaviour of the heuristics on the conclusions of the benchmark. ILP test-cases were only executed once since this technique ensures the optimum solution.
- are evaluated and compared throughout their effectiveness, which is measured by the cost of the best solution found as well as their efficiency, which is measured by the time taken to perform a run.

The outcome, illustrated in Figure 5 and reported in Table 4, is hereafter further discussed :

- **Efficiency:**
  - For each test-case, both heuristics take comparable time to complete. This is due to the parameters that have been used: instead of stopping the heuristics after a given amount of consecutive iterations with limited or no improvement, we simply limit it by a total number of iterations that is based on the search-space size. More concretely, the SA implementation defines an initial temperature value of 1, an α value of 0.9 and the annealing stops when temperature is smaller than 0.00001. Those annealing parameters ensure for a constant 111 temperature reductions. The inner loop, specifying the number of iterations at a given temperature varies with the size of the search-space and consists of the number of containers multiplied by the number of servers divided by 25. Those parameters generate thus 1110, 4440, 111000 and 999000 permutations for the S, M, L and XL scenarios respectively. The PSO implementation defines the number of particles as the number of servers, each particle iterating C times, where C equals the number of containers. Those parameters generate thus 250, 1000, 25000 and 225000 permutations for the S, M, L and XL scenarios respectively. Those parameters ensure comparable efficiency for both heuristics : while SA performs more iterations, PSO must compute the difference between a particle's position and the attractor's position at each non-random move. SA and PSO parameters optimization has already been researched and discussed in the literature (e.g. [58] for PSO and [59] for SA) and is therefore considered as out-of-scope for this work; instead those empirical values have been retained as they allow for a fair comparison of the effectiveness of each heuristic.
  - Depending on the order of magnitude of the actual cluster to be rescheduled, one

[3] (gap = 31.83%)

Table 4 Measured efficiency and effectiveness for the different test-cases

Topology	Size	Algo	Effectiveness		Efficiency
			AVG	AVG/ILP	AVG
Dense	S	SA*	16.74%	90.95%	0.034s
		PSO*	16.63%	90.39%	0.034s
		ILP	18.4%	100%	1.417s
	M	SA	21.56%	86.98%	0.313s
		PSO	21.76%	87.81%	0.307s
		ILP	24.78%	100%	2.03d
	L	SA	15.26%	(99.8%)	80.9s
		PSO	15.03%	(98.29%)	82.8s
		ILP[3]	(15.3%)	100%	(74.9d)
	XL	SA	10.23%	-	7233s
		PSO	9.92%	-	7249s
		ILP	-	-	-
Clustered	S	SA*	43%	94.59%	0.032s
		PSO*	42.49%	93.48%	0.03s
		ILP	45.45%	100%	1.13s
	M	SA	46.23%	90.15%	0.267s
		PSO	46.27%	90.24%	0.243s
		ILP	51.28%	100%	2.16s
	L	SA	49.09%	89.14%	73.8s
		PSO	48.51%	88.08%	71s
		ILP	55.07%	100%	6.4d
	XL	SA	49.49%	-	4985s
		PSO	49.6%	-	4658s
		ILP	-	-	-

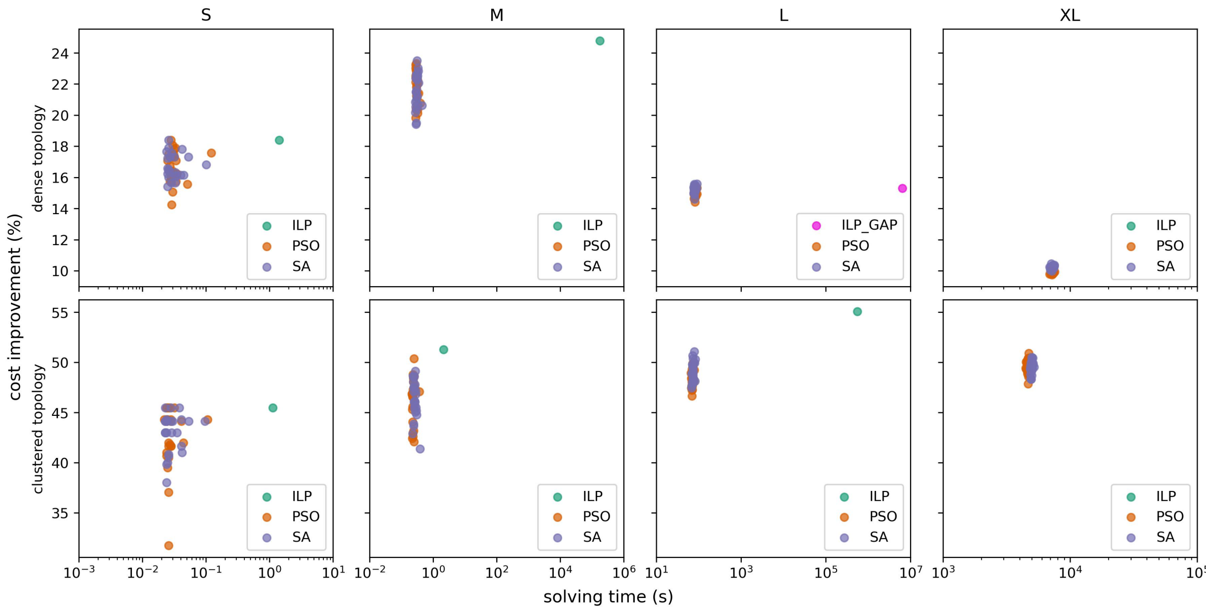

Figure 5 Efficiency and effectiveness comparison

could however retain other parameters that would better meet a specific trade-off between efficiency and effectiveness. For instance, if the supervised cluster size doesn't exceed by far the M scenario, more iterations could reasonably be performed for a possibly higher effectiveness within acceptable execution time boundaries. Inversely, one could consider that the time taken by the heuristics for the L and XL scenarios are not acceptable and therefore reduce the number of iterations (possibly at the cost of a lower effectiveness).

- The PSO heuristic is on average more efficient than SA for all scenarios of the clustered topology as well as for scenario M of the dense topology. However, the difference in time is relatively limited.
- With the search-space size defined as $C * S$, where C is the number of containers and S the number of servers, a quadratic time complexity is observed for both heuristics.
- Due to its disqualifying inefficiency (except for the 'S' scenarios as well as scenario 'M' of the Clustered topology), ILP should more be interpreted as a yardstick for the comparison of both heuristics relative effectiveness. For the 'L' scenario of the Dense topology, the ILP solver was not able to provide an optimum value since it ran out of memory[4] after 74.9 days. For both 'XL' scenarios, an out-of-memory crash happens at modelling time (during variables creation)[5].

- **Effectiveness:**
  - As mentioned in previous bullet, for each test-case, both heuristics take about the same amount of time to complete. SA tries 111/25 times more permutations than PSO, however, being a population based heuristics, PSO has the advantage of searching at different locations of the search-space in parallel. Interestingly, not only does it result in comparable efficiency but also in comparable effectiveness for both heuristics.
  - The topology significantly affects effectiveness. This is observed for all scenario sizes.
  - The heuristics relative effectiveness averages around 90% when compared to ILP optimum (see Table 4) and exhibits negative correlation with scenario size: this involves that, despite the increasing improvement ratio, the gap between ILP optimum and heuristics best solution increases with scenario size. As previously stated, the ILP solver crashed af-

[4] ilog.cplex.CpxException: CPLEX Error 1001: Out of memory.

[5] java.lang.OutOfMemoryError: Java heap space - reported by the program used to feed CPLEX with the variables and constraints.

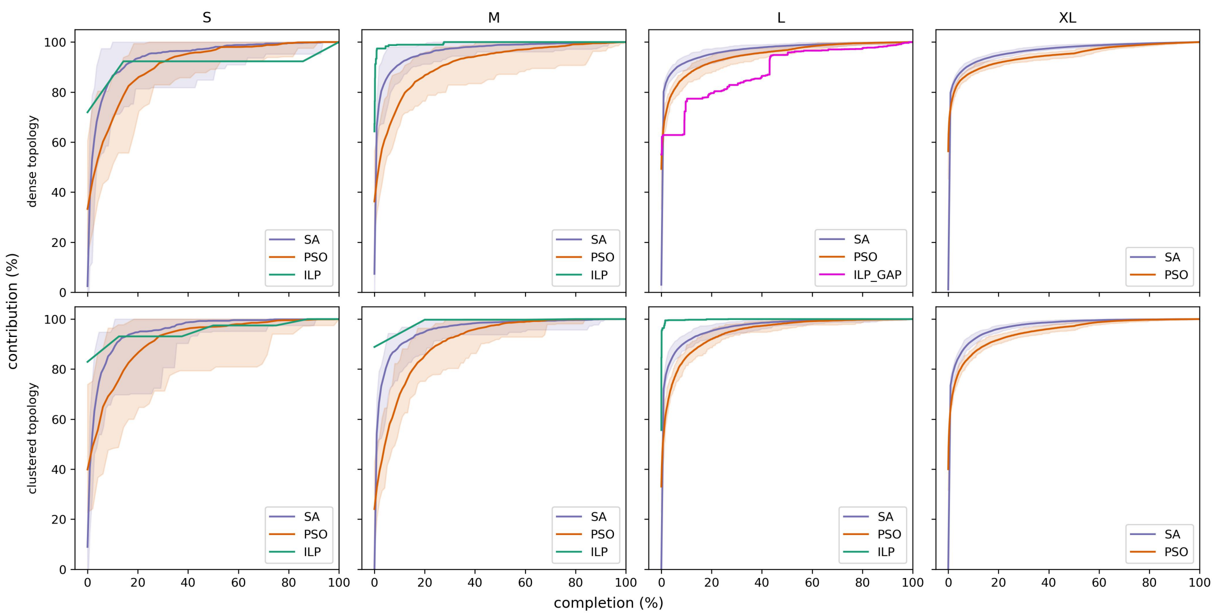

Figure 6 Comparison of the relative progress of cost improvement along the iterations

ter 74.9 days for the ‘L’ scenario of the Dense topology. At crash-time though the best solution found was the 15.3%, however with an optimality gap value of 31.83%. The reported solution is thus probably not the optimum; this is confirmed when comparing it with SA and PSO that were able to improve the cost by 15.58% and 15.42% respectively.

  – For the ‘S’ scenario of the two distinct topologies, both heuristics succeed at least once in finding the optimum; SA exhibiting a slightly higher effectiveness average than PSO for the 25 different runs. While both heuristics offer very similar effectiveness average across the different sizes and topologies, SA surpasses PSO most often.

Lastly, Figure 6 illustrates the relative progress of cost improvement along the iterations for each approach. Solid lines represent the average improvement and the coloured surrounding shade indicates the dispersion around this average. It is worth mentioning that X and Y axis are expressed as percentages, allowing for a relative comparison of the different approaches. Main outcome is hereafter discussed:

- ILP’s initial solution already covers between 50% and 90% of the distance between the cost of the current context and the optimum. Inversely, the cost of the initial solution for both heuristics, being randomly generated, may even be worse than the cost of the current context.
- While ILP progresses by steps with relatively long stable phases, both heuristics continuously progress.
- Most of the contribution to the cost improvement happens at early iterations. This observation advocates for a relative stop criterion (e.g. the slope evolution) instead of an absolute stop criterion (e.g. the number of iterations), certainly for high search-space use-cases like the ‘L’ and ‘XL’ scenarios. Indeed, heuristics efficiency could be improved by a factor of 3 (i.e. stopped at 30-35 percent of current implementation run time) with a limited impact on the effectiveness.
- SA surpasses PSO in all scenarios as it exhibits a sharper slope in early iterations than PSO, reaching earlier near optimum solution. Additionally, the standard deviation for SA is smaller than for PSO, making it more predictable. In relative stop criterion implementation those two factors are determinant and plead in favor of SA.

4 Validating the rescheduling system in a container orchestration platform

This section aims at validating the algorithm presented in Section 3 within a container orchestration platform. To this end the SA implementation of the algorithm is retained. Additionally, K8s has been selected as container orchestration platform due to its prominent market position and proven track record [9]. 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, after which the dynamic rescheduling system, embedding the SA implementation, is presented. Lastly, the impact on application service time of the system is tested and evaluated by means of 2 distinct concrete use-cases.

4.1 Scheduling and descheduling containers in K8s

4.1.1 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.

4.1.2 K8s scheduler

K8s default scheduling system (KS) has static rules to schedule Pods in a cluster. Developers can specify resource requests and limits on the Pod configuration file. A resource request is the minimum amount of resources (e.g. CPU and/or RAM) required by all containers in the Pod while a resource limit is the maximum amount of resources that can be allocated for the containers in a Pod. 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 can (not) be scheduled on and Inter-pod (anti-) affinity allowing to constrain which nodes a Pod can (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 ("requiredDuringSchedulingIgnoredDuringExecution") or soft ("preferredDuringSchedulingIgnoredDuringExecution") 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[6]. The KS uses those resource and (anti-) affinity constraints in its allocation decisions.

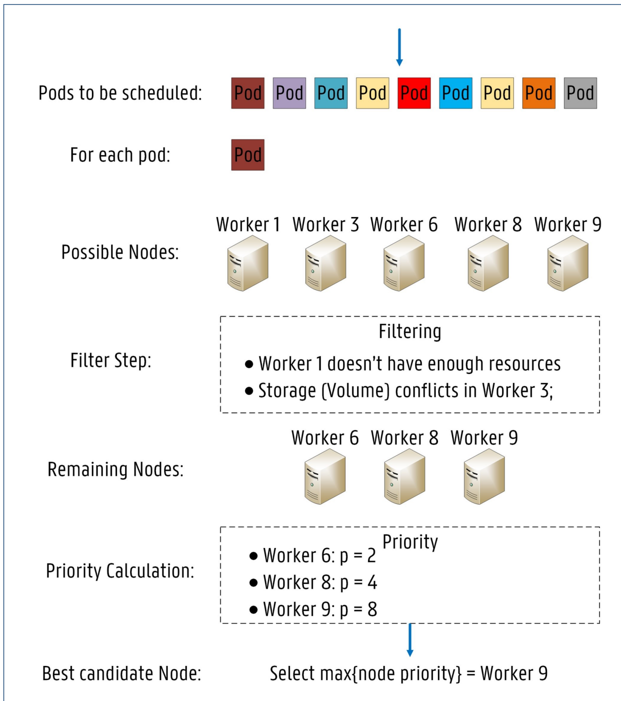

Figure 7 Sample of detailed scheduling operations of the Kube–Scheduler [60]

Every Pod that requires allocation is first added to a queue, which is monitored by the KS. As illustrated in Figure 7, the KS allocates Pods to Nodes based on a two-step procedure. The first step is to filter the

[6] 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 (see Section 5).

available Nodes based on a set of predicates 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 KS can use predicates to filter the Nodes which are suitable for the Pod that needs to be scheduled. Priority calculation is used if multiple Nodes still remain after predicate filtering. 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". 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.

4.1.3 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 scheduler's decisions are influenced by its view of a K8s cluster at that point of time when a new pod appears for scheduling [61]. 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 scheduler for their rescheduling afterwards. The K8s descheduler's policy is configurable by the definition of the following 7 settings:

- nodeSelector: limits the nodes which are processed.
- evictLocalStoragePods: allows eviction of pods with local storage.
- evictSystemCriticalPods: allows eviction of pods with any priority, including system pods.
- ignorePvcPods: set whether Persistent Volume Claim (PVC) pods should be evicted or ignored.
- maxNoOfPodsToEvictPerNode: maximum number of pods evicted from each node.
- maxNoOfPodsToEvictPerNamespace: maximum number of pods evicted from each namespace.
- evictFailedBarePods: allows eviction of pods without owner references and in failed phase.

Additionally, 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 requests and limits 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.
- RemovePodsViolatingInterPodAntiAffinity: 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.
- RemovePodsViolatingTopologySpreadConstraint: 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.

Lastly, the K8s descheduler allows for *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.

4.2 Architecture and design of the rescheduling system

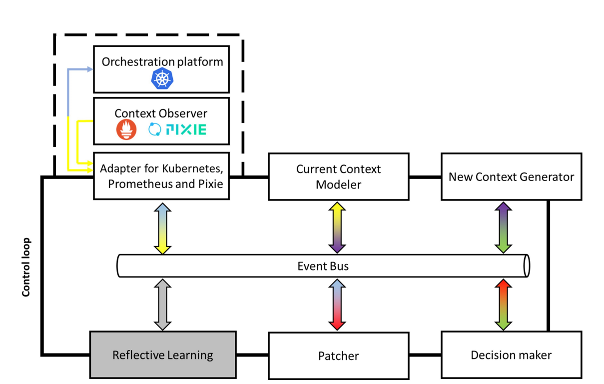

Figure 8 The self driving rescheduling control loop

The proposed dynamic container rescheduling system is designed as a closed control loop as illustrated in Figure 8. A closed control loop, also referred to as feedback loop, is a non-terminating loop that regulates the state of a dynamical system and manipulates it towards a more desirable state while minimizing any delay, overshoot, or steady-state error and ensuring a level of control stability; often with the aim to achieve a degree of optimality [62]. In K8s for instance, controllers are implemented by means of control loops that watch the state of the cluster and make changes where needed [63]. The main purpose of these controllers is to push the cluster closer to the desired state. Under stable load, the control loop should eventually stop to adapt the system it controls by converging to an optimum. The designed control loop is based on the following 6 components:

1 The **Adapter** is in charge of interfacing the dynamical system (i.e. the orchestration platform and monitoring tools) specific APIs and consequently allows the portability of the control loop to other environments. It mainly fulfills 2 functionalities:
   - **Context data fetching**: when a request for context data fetching event arrives on the *gatherClusterData* topic, the Adapter queries:
     - Prometheus to collect node characteristics (e.g. the Fully Qualified Domain Name (FQDN), the allocatable CPU and RAM capacity as well as the taints and labels) and pod characteristics (e.g. the name, the IP address, the hosting node, the resource requests (CPU and RAM) and the labels). Labels play an important role as they are used afterwards for the matching to pod and node (anti-) affinities. Furthermore, the proposed rescheduling system identifies the reschedulable pods by means of a specific label : only pods having the label 'reschedulable' with the value 'true' will be considered as potential candidates for rescheduling. Lastly, the pod resource usage (RAM and CPU) for the latest 's' seconds are also collected; the actual pod resource need is then defined as the highest value between the pod resource request value and the observed pod resource usage value. This allows to avoid considering the rescheduling of a pod to a certain node when feasible from the perspective of its resource request but not from the perspective of its observed resource usage. The value for 's' is provided in the request for context data fetching event.
     - The K8s API to collect all pod and node (anti-) affinities.
     - The PIXIE backend for network traffic metrics for the last 's' seconds. Despite the vast amount of monitoring metrics provided by the classical

K8s-Prometheus association, those are mostly limited to a system and infrastructure perspective: no real application level metrics are defined as standard that would allow to monitor network flows for instance. Of course one can always expose application metrics to Prometheus but those would remain specific and can hardly be generalized; this approach is anyway no option since it would require to rewrite all individual hosted applications to let them expose the required metrics. This issue is usually circumvented by adding a 'network sniffer' component within the cluster. Two options are then possible:

 * Embedding the sniffer container along with the application containers as a sidecar in each pod. Istio for instance uses this concept as a foundation for its service mesh. Main advantage of this approach resides in the fact that the sidecar container may be used as Transport Layer Security (TLS) termination, potentially enabling richer reporting at the cost, however, of significant unwieldiness as it requires to embed such sidecar container into every pod of the service mesh.
 * Deploying a single instance of the sniffer container on each node. This approach benefits from a more lightweight footprint while meeting the requirements of the proposed rescheduling system, i.e. collecting inter-pod network traffic volumetry. 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 [64]. 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.

- **Containers rescheduling**: when a request for patching event arrives on the *moveContainers* topic, 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 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 chosen rescheduling algorithm. It is worth mentioning that the choice of algorithm is not limited to the three optimization techniques presented in this work (ILP, SA and PSO) as the New Context Generator component launches its execution through an algorithm agnostic interface. 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 (e.g. podA is running on node1, must be moved to node2 and has an anti-affinity with podB, currently running on node2 but having to go to node3. In this case, podB will be moved first). Afterwards, the Patcher sends that ordered list to the Adapter (through the *moveContainers* topic) 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 has not been implemented in this work and would certainly justify specific research as it represents a challenge on its own (mainly to isolate the impact of the rescheduling on application service time in volatile load context).

4.3 First validation use-case : a cloud-based IoT data-hub platform

This first use-case focuses on an IoT cloud-based data-hub platform. A data-hub is a central mediation point between various data sources and data consumers [65]. With a data-hub serving as a single point of data access, users receive the means to structure and harmonize information collected from various sources; a key asset in IoT applications where data integration remains a complex challenge. There exists a large amount of data hub platforms, some of the most prominent ones being: CDP Data Hub [66], Cumulocity IoT DataHub [67], Azure IoT Hub [68], AWS IoT Core [69] and Google Cloud IoT Core [70]. IoT applications typically exhibiting volatile load patterns, this use-case may certainly be considered relevant for testing and validating the proposed dynamic rescheduling system.

4.3.1 Architecture of the cloud-based IoT data-hub

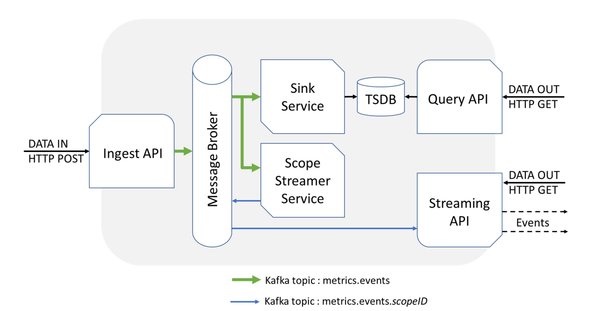

Figure 9 Architecture of the IoT data-hub: data flow from ingest side to query and streaming side

The IoT data-hub platform that has been used is a simplified version of Obelisk [71] that has the advantage of being based on widely used open-source packages, which made the implementation of this test version straight-forward. The event-based microservice architecture of the IoT data-hub is presented in Figure 9. It 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. 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[7], 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.

4.3.2 Performance evaluation

The test is conducted on a K8s v1.23 cluster with 7 nodes. The nodes are running Ubuntu 18.04.6 LTS and are equipped with 2 Quad core Intel E5520 (2.2GHz) CPUs, 12GB of RAM, a hard disk of 160GB and a gigabit network interface. *Node0* is the Master Node and hosts the K8s Control PLane while *Node1..6* are Worker Nodes and consequently are available for application hosting. 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 *2* respectively. Messages are being emitted every second from 25 simulated client devices on the 'test' scope with the content described in Listing 1.

Listing 1 JSON formatted event sent by devices to the IoT Data-Hub platform

```
{
  "device": deviceID ,
  "timestamp": timestampMillis ,
  "scope": "test",
  "metric": "temperature",
  "value": measuredTemp ,
  "location": [3.733333,51.049999],
  "elevation": 0.0
}
```

[7] 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.

Table 5 indicates the hosting node prior to and after the rescheduling. Infrastructure pods (the *TSDB* and the *Kafka Broker 0..2* instances) being flagged as not reschedulable, remain on their initial node. The API and Service pods, all initially hosted on *Node1*, on the contrary have been flagged as reschedulable and are re-assigned accordingly:

- The *Ingest API* is moved to *Node2*, where *Kafka Broker 0* is running, since this broker instance is hosting the *metrics.events* Kafka topic to which the *Ingest API* is publishing all entering metric events.
- The *Sink Service* is moved to *Node5*, where the *TSDB* instance is running. Each API and Service in the chain adding some technical extra information to messages (timestamps, podID, etc.), those messages get thus heavier leaving a pod than entering it; for a given amount of messages, the *Sink Service* consequently sends more bytes to the *TSDB* than it receives from the *Kafka Broker 0* and is therefore assigned to *Node5* rather than *Node2*.
- The *Query API* fetching data from the *TSDB*, it is moved to *Node5*, where the *TSDB* instance is running.
- The *Scope Streamer Service* is moved to *Node4* along with *Kafka Broker 2* that hosts the *metrics.events.test* topic. As previously stated, as each intermediary Service or API adds some additional technical information to all events it processes, the *Scope Streamer Service* sends more bytes to the *metrics.events.test* topic than it receives from the *metrics.events* topic. Importantly though, if another scope would have been used in parallel by another set of devices, and if the related topic of that other scope would be hosted on another broker, then the *Scope Streamer Service* would most likely have been moved to *Node2* along with *Kafka Broker 0* that hosts the *metrics.events* topic since the sum of bytes for both scopes would be higher than each individual one.
- The *Streaming API* is moved to *Node 4* where the *metrics.events.test* topic is hosted (on the *Kafka Broker 2*) as the developed client application only consumes events from the 'test' scope.

The rescheduling control loop took all in all 1820 milliseconds with 725 milliseconds for the *Current Context Modeler* to build the cluster context composed of 67 pods running on 7 nodes, 128 milliseconds for the *New Context Generator* to compute the best possible context with the SA algorithm, instantly confirms that the gain is sufficient (0 millisecond for the *Decision Maker*) and finally 967 milliseconds for the *Patcher* to request the K8s scheduler to reschedule the 5 pods.

Table 5 The Pod to Node assignation before and after rescheduling for the IoT Data-Hub use-case

Pod	Is reschedulable?	Node before rescheduling	Node after rescheduling
TSDB	NO	5	5
Kafka Broker 0	NO	2	2
Kafka Broker 1	NO	3	3
Kafka Broker 2	NO	4	4
Ingest API	YES	1	2
Sink Svc	YES	1	5
Query API	YES	1	5
Scope Streamer Svc	YES	1	4
Streaming API	YES	1	4

Lastly, Figure 10 illustrates the service time evolution before, during and after the pods rescheduling. This detailed analysis focuses solely on the streaming data flow. More specifically :

- The **i2r** labelled sub-chart represents the evolution of the One-Way Delay (OWD) spent between the publication by the *Ingest API* of an event on the *metrics.events* Kafka topic and its reception by the *Scope Streamer Service*. Table 6 indicates the averaged OWD in milliseconds for this first link before, during and after the rescheduling. An average improvement of the OWD by 30.2% is observed for this first link.
- The **r2s** labelled sub-chart represents the evolution of the OWD spent between the publication by the *Scope Streamer Service* of an event on the *metrics.events.test* Kafka topic and its reception by the *Streaming API*. Table 6 indicates the averaged OWD in milliseconds for this second link before, during and after the rescheduling. An average improvement of the OWD by 15.6% is observed for this second link.
- The **e2e** labelled sub-chart represents the evolution of the end-to-end service time spent within the platform, i.e. between the reception of an event by the *Ingest API* and the emission of the same event by the *Streaming API* to the consuming application. Table 6 indicates the averaged service time in milliseconds for the end-to-end service delivery before, during and after the rescheduling. Noticeably, an average improvement by 21.6% is observed.

If quite promising, this overall improvement remains however to temper with the impact the rescheduling generates when moving pods from one node to the other. Indeed, during the 23 seconds period of effective rescheduling, the end-to-end network latency peaks at 4 seconds for some events and exhibits an overall average of 507.54 milliseconds, two orders of magnitude

Table 6 Evolution of the average OWD and end-to-end service time before, during and after the rescheduling for the IoT Data-Hub use-case

Label	From	To	Avg_before	Avg_during	Avg_after	Improvement
i2r	Ingest_out	ScopeStreamer_in	2.157ms	7.171ms	1.506ms	30.2%
r2s	ScopeStreamer_out	Streaming_in	2.159ms	498.907ms	1.823ms	15.6%
e2e	Ingest_in	Streaming_out	4.664ms	507.539ms	3.655ms	21.6%

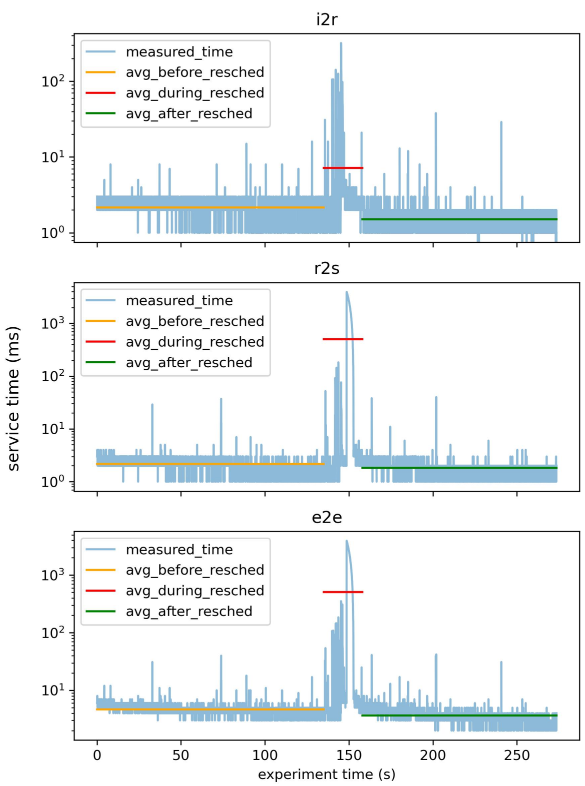

Figure 10 Evolution of the OWD and its impact on the end-to-end service time before, during and after the rescheduling for the IoT Data-Hub use-case

higher than observed in stable situation. This is mainly caused by the (re-)connection of Kafka clients to the broker being no lightweight operation.

4.4 Second validation use-case : a web-based e-commerce app

The second use-case is based on the so called 'Online Boutique', a cloud-first microservices demo application developed and used by Google to demonstrate use of technologies like K8s/GKE, Istio, Stackdriver, gRPC and OpenCensus. This application works on any K8s/GKE cluster. It's easy to deploy with little to no configuration. The application is a web-based e-commerce app where users can browse items, add them to the cart, and purchase them [72]. The web-based architecture of this use-case usefully complements the events-based architecture of the first use-case.

4.4.1 Architecture of the Online Boutique

Figure 11 illustrates the architecture of the 'Online Boutique'. Unlike the previous use-case, the services here do not communicate through a Message Broker but rather through direct HTTP calls. Furthermore, there are more services and interactions among them.

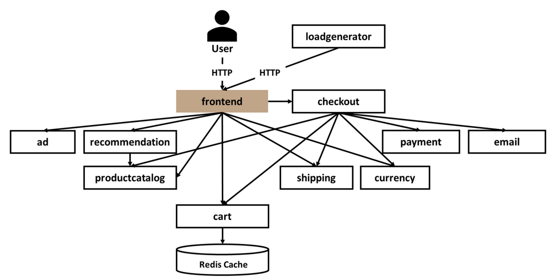

Figure 11 Architecture diagram of the web-based e-commerce 'Online Boutique' app

The components constituting the 'Online Boutique' are hereafter briefly introduced:

- The *frontend* service exposes a HTTP server to serve the website. It does not require signup/login and generates session IDs for all users automatically.
- The *cart* service stores and retrieves the user's shopping cart into/from the *Redis cache* database.
- The *Redis cache* database stores cart data.
- The *productcatalog* service provides the list of products as well as individual product details.
- The *currency* service converts one money amount to another currency. It uses real values fetched from European Central Bank.
- The *payment* service charges the given credit card info (mock) with the given amount and returns a transaction ID.
- The *shipping* service gives shipping cost estimates based on the shopping cart and ships items to the given address (mock).
- The *email* service sends users an order confirmation email (mock).

- The *checkout* service retrieves user cart, prepares order and orchestrates the payment, shipping and the email notification.
- The *recommendation* service recommends other products based on the cart content.
- The *ad* service provides text ads based on given context words.
- The *loadgenerator* service continuously sends requests imitating realistic user shopping flows to the *frontend* service.

4.4.2 Performance evaluation

The test has been conducted on the same cluster than the one used for the first use-case (see subsection 4.3.2). 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 browsing script that now simulates 2 users endlessly looping on this scenario:

- 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 as parameters. The client details consist of an email address, a physical address (street and number, zip code, city, state, country) and the credit card details (number, expiration month, expiration year and CVV).

Every second, both simulated users execute the next call in the sequence. Both users are initially shifted by 500ms.

Table 7 The Pod to Node assignation before and after rescheduling for the 'Online Boutique' use-case

Pod	Is reschedulable?	Node before rescheduling	Node after rescheduling
RedisCache	NO	6	6
Cart Svc	YES	3	6
LoadGenerator	NO	2	2
Frontend	YES	6	2
Checkout Svc	YES	3	2
Payment Svc	YES	5	2
Email Svc	YES	5	2
Recommendation Svc	YES	4	2
Ad Svc	YES	5	2
ProductCatalog Svc	YES	4	2
Shipping Svc	YES	3	2
Currency Svc	YES	6	2

Table 7 indicates the hosting node prior to and after the rescheduling. The *Redis cache* infrastructure pod being flagged as not reschedulable, remains on its initial node. Similarly, the *LoadGenerator* pod has also been flagged as not reschedulable and consequently also remains on its initial node. The remaining 10 pods are re-assigned accordingly:

- The *cart* service is moved to *Node6*, where the *Redis cache* pod resides.
- The *frontend* service is moved from *Node6* to *Node2*, hosting the *LoadGenerator* service. The minimization of the cost function progressively attracting all the 8 other reschedulable pods towards that same node.

The rescheduling control loop took all in all 3176 milliseconds with 734 milliseconds for the *Current Context Modeler* to build the cluster context composed of 74 pods running on 7 nodes, 194 milliseconds for the *New Context Generator* to compute the best possible context with the SA algorithm, instantly confirms that the gain is sufficient (0 millisecond for the *Decision Maker*) and finally 2248 milliseconds for the *Patcher* to request the rescheduling of the 10 pods. Interestingly, the actual pod rescheduling step explains most of the difference with the first use-case where the *Patcher* needed 967 milliseconds for 5 pods to be reassigned. From this, it can be deducted that approximately 200 milliseconds are required per single patching request.

Table 8 Evolution of the average service time before, during and after the rescheduling for the 'Online Boutique' use-case

Label	Avg_before	Avg_during	Avg_after	Improvement
/	22.261ms	31.684ms	21.779ms	2.2%
/setCurrency	0.953ms	0.917ms	0.644ms	32.4%
/product	16.856ms	26.291ms	14.687ms	12.9%
/cart	4.514ms	7.436ms	3.661ms	18.9%
/cart/checkout	58.844ms	74.167ms	49.653ms	15.6%
e2e_xp	147.582ms	1708.0ms	128.286ms	13.1%

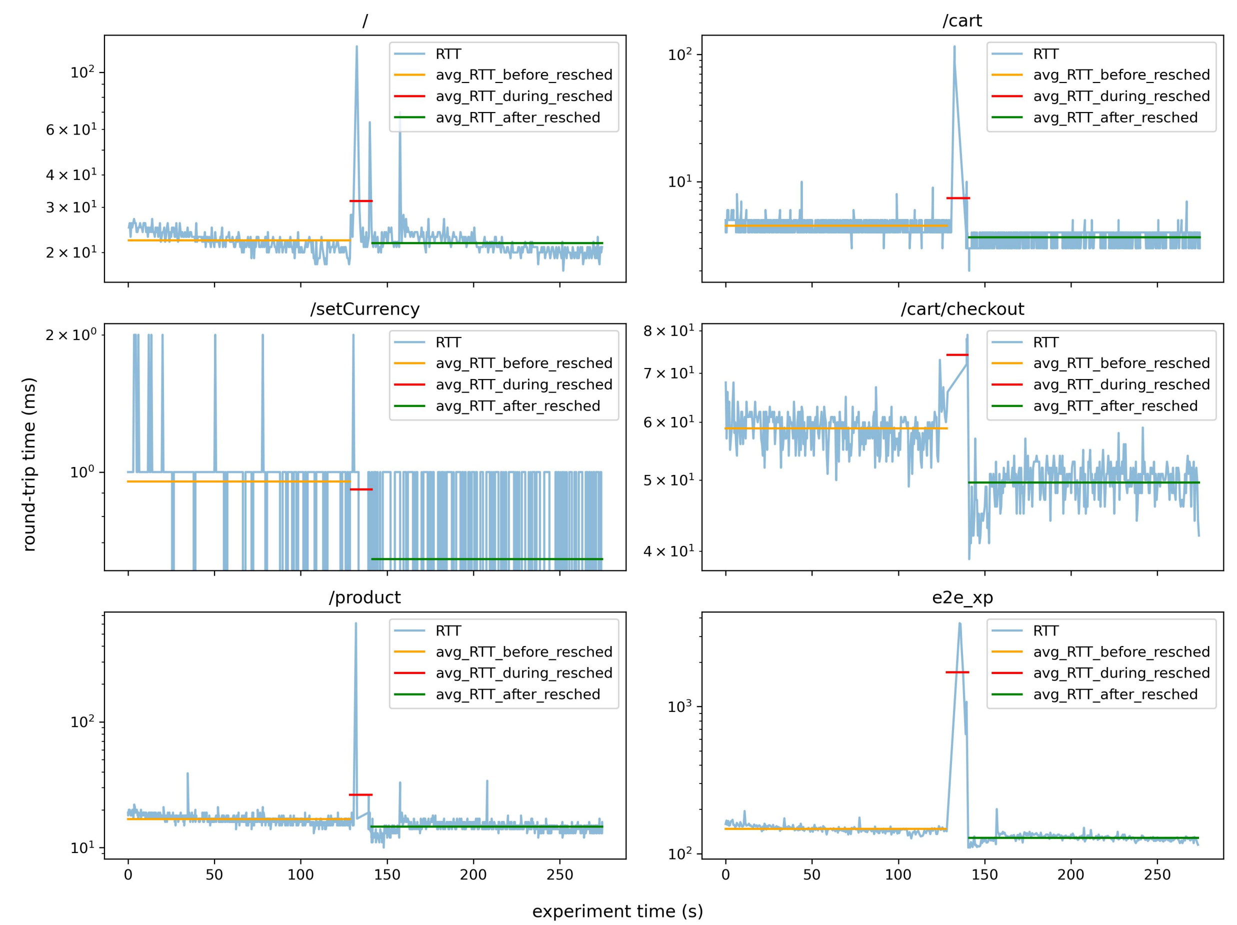

Figure 12 Evolution of the service time before, during and after the rescheduling for the 'Online Boutique' use-case

Lastly, Figure 12 illustrates the service time evolution before, during and after the pods rescheduling. More specifically :

- The **'/'** labelled sub-chart represents the evolution of the service time when accessing the 'Home page' of the 'Online Boutique' all along the experiment. Table 8 indicates the average service time in milliseconds before, during and after the rescheduling. An improvement of the average service time by 2.2% is observed. This modest improvement is mainly explained by the fact the *frontend* service locally hosts the home web-page and, consequently, does not need to call any other service.
- The **'/setCurrency'** labelled sub-chart represents the evolution of the service time when setting the currency to be used for the forthcoming transactions. An improvement of the average service time by 32.4% is observed, as mentioned in Table 8. This optimistic result must however be tempered as this service runs extremely fast exhibiting a service time of only 0, 1 or 2 milliseconds before and during the rescheduling. After the rescheduling, the service replies in 0 or 1 millisecond only. The time-granularity used for the test limits thus the interpretation of this specific result; nanoseconds precision however would not be justified for all the other services.
- The **'/product'** labelled sub-chart represents the evolution of the service time when browsing a specific product page. An improvement of the average service time by 12.9% is observed, as mentioned in Table 8.
- The **'/cart'** labelled sub-chart represents the evolution of the service time when adding a certain quantity of a product to the cart. An improvement of the average service time by 18.9% is observed, as mentioned in Table 8.
- The **'/cart/checkout'** labelled sub-chart represents the evolution of the service time spent at order checkout. An improvement of the average

service time by 15.6% is observed, as mentioned in Table 8.

- Lastly, the **'e2e_xp'** labelled sub-chart represents the evolution of the service time spent for the end-to-end customer journey in the 'Online Boutique' (cfr. *loadgenerator* scenario described hereinabove). An improvement of the average end-to-end service time by 13.1% is observed, as mentioned in Table 8.

Overall, the impact the rescheduling generates when moving pods from one node to the other still exists but is relatively less significant than it is for the first use-case. Indeed, during the 12 seconds period of effective rescheduling, the **e2e_xp** end-to-end service time peaks at 3677 milliseconds and exhibits an overall average of 1708 milliseconds, one order of magnitude higher than observed in stable situation.

5 Discussion and Future work

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

- In order to reduce the negative impact of the actual rescheduling action, different approaches should be experimented, among which limiting the number of containers that can be rescheduled per iteration (possibly with a prioritisation mechanism), increasing the delay between patching commands emission, etc.
- Constraints management should be extended to also include K8s-specific concepts like taints (other than *NoSchedule*, already covered), soft (anti-) affinities (i.e. 'preferences'), topology labels (e.g. geographic regions and zones), etc.
- Network connectivity awareness represents an additional direction for further investigation, since it would allow to not only cover centralized cloud clusters but also distributed clusters where the quality of the network link between nodes can not be assumed to be equal and constant. To this end, instead of defining s_n^{pq} as a binary variable in equation 1 it could be defined as a decimal value between 0.0 and 1.0 and would then be used as an indication of the network latency between pods, with 0.0 if containers 'p' and 'q' are both hosted on server 'n' or if none of them is, else a relative latency score. The worse the latency, the highest the score.
- Multi-objective optimization would also greatly improve the proposed system which, in its current version, does not take resource saturation (e.g. CPU throttling) into account. Concentrating containers on few servers may ultimately turn counterproductive, rather a trade-off between network delay optimization and fair load distribution is intuitively desirable.
- 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. This would certainly justify specific research on its own as isolating the impact of a rescheduling decision on application service time in volatile load context represents a certain challenge.

6 Conclusion

This article proposes a portable dynamic rescheduling system for container orchestration platforms that aims at improving application service time by minimizing network delay among containers. 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 reduction. If the best alternative sufficiently reduces it, containers are reassigned accordingly. The constrained Quadratic Assignment Problem of identifying the best alternative is solved by a metaheuristic. To this end, the effectiveness and efficiency of PSO and SA are compared and also benchmarked against an ILP approach which ensures optimum solution at the cost, though, of a disqualifying execution time. Out of this performance study, the SA metaheuristic is retained. The impact of the proposed system on application service time is evaluated and discussed by means of the *cloud-based IoT data-hub platform* and the *Online Boutique* complementary use-cases with an improvement of the end-to-end service time of 21.6% and 13.1%, respectively. 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 introduced.

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

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

Abbreviations

ACO Ant Colony Optimization. 6
API Application Programming Interface. 16–20

CNCF Cloud Native Computing Foundation. 15, 18
CPU Central Processing Unit. 1, 3–5, 9, 12, 15, 17, 19, 24
CVV Card Verification Value. 22

DPSO Discrete PSO. 8

FIFO First In, First Out. 2
FQDN Fully Qualified Domain Name. 17, 18

GAs Genetic Algorithms. 6
GKE Google Kubernetes Engine. 21
GRASP Greedy Randomized Adaptive Search Procedures. 6
gRPC Google Remote Procedure Calls. 21

HBMO Honey-Bees Mating Optimization. 6
HTTP Hypertext Transfer Protocol. 21, 22

I/O Input/Output. 1
ILP Integer Linear Programming. 1, 5, 12–14, 18, 24
ILS Iterated Local Search. 6
IoT Internet of Things. 1, 2, 19–21, 24
IP Internet Protocol. 17
IT Information Technology. 2

JDK Java Development Kit. 12
JFO Jumping Frogs Optimization. 8
JPSO Jumping Particle Swarm Optimization. 8
JSON JavaScript Object Notation. 19

K8s Kubernetes. 1–3, 14–19, 21, 24
KS Kubernetes Scheduler. 15, 16

LTS Long-Term Support. 12, 19

NP Non-deterministic Polynomial-time. 2, 5

OS Operating System. 1, 12
OWD One-Way Delay. 20, 21

PSO Particle Swarm Optimization. 1, 5–8, 10, 12–14, 18, 24
PVC Persistent Volume Claim. 16

QAP Quadratic Assignment Problem. 4, 24
QoS Quality-of-Service. 2–4

RAM Random-Access Memory. 1, 4, 5, 9, 12, 15, 17, 19

SA Simulated Annealing. 1, 5–8, 10, 12–15, 18, 20, 22, 24

TIARM Throttling and Interaction-aware Anticorrelated Rescheduling for Microservices. 3, 4
TLS Transport Layer Security. 18
TS Tabu Search. 6
TSDB Time Series DataBase. 19, 20

URI Uniform Resource Identifier. 22
URN Uniform Resource Name. 22

VMC Virtual Machine Consolidation. 3
VNS Variable Neighborhood Search. 6

Declarations

Ethical Approval

Not applicable.

Funding

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

Competing interests

The authors declare that they have no competing interests.

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.
Gillis Werrebrouck substantially contributed to the design of the work, as well as to the creation of new software used in the work. He drafted the work.
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.

Gillis Werrebrouck obtained a bachelor's degree in Multimedia and Communication Technology from Howest, Belgium in 2018 and a master's degree in Information Engineering Technology from Ghent University, Belgium in 2021. After his master's degree, he interned for the Customer Engineering team at Amazon Web Services (AWS) in Berlin, Germany. In December of 2021, he joined the Customer Engineering team at AWS full time as Software Development Engineer. His work at AWS involves the architectural design and end-to-end development of highly complex cloud applications for prominent worldclass companies.

José Santos obtained his M.Sc. degree in Electrical and Computers 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 and the Best Dissertation Award at NOMS 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.

Author details

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

References

1. da Silva, V.G., Kirikova, M., Alksnis, G.: Containers for Virtualization: An Overview. Applied Computer Systems **23**(1), 21–27 (2018)
2. Docker. Docker, Inc.: Docker Website. `https://www.docker.com`. [Online] (2022)
3. LXC/LXD. The Linux Foundation: LXC Website. `https://linuxcontainers.org`. [Online] (2022)
4. Podman. Red Hat, Inc.: Podman Website. `https://podman.io/`. [Online] (2022)
5. containerd. The Cloud Native Computing foundation: containerd Website. `https://containerd.io`. [Online] (2022)
6. Mesos. The Apache Software Foundation: Mesos Website. `http://mesos.apache.org`. [Online] (2022)
7. Docker. Docker, Inc.: DockerSwarm Website. `https://docs.docker.com/engine/swarm/`. [Online] (2022)
8. Kubernetes. The Cloud Native Computing Foundation: Kubernetes Website. `https://kubernetes.io`. [Online] (2022)
9. Flexera: RightScale 2019 State of the Cloud Report from Flexera. Available at `https://resources.flexera.com/web/media/documents/rightscale-2019-state-of-the-cloud-report-from-flexera.pdf` (2020/05/20) (2019)
10. Pinedo, M.L.: Scheduling, 5th edn., p. 670. Springer, New York, USA (2016). doi:10.1007/978-3-319-26580-3
11. 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). doi:10.1016/j.cosrev.2018.08.002
12. Söylemez, M., Tekinerdogan, B., Tarhan, A.K.: Challenges and solution directions of microservice architectures: A systematic literature review. Applied Sciences (2022)
13. 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. Concurrency and Computation: Practice and Experience **24**(13), 1397–1420 (2012). doi:10.1002/cpe.1867
14. Mahdhi, T., Mezni, H.: A prediction-based vm consolidation approach in iaas cloud data centers. Journal of Systems and Software **146**, 263–285 (2018). doi:10.1016/j.jss.2018.09.083
15. 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). doi:10.1002/spe.2680
16. Zhao, D., Mohamed, M., Ludwig, H.: Locality-aware scheduling for containers in cloud computing. IEEE Transactions on Cloud Computing **8**(2), 635–646 (2020). doi:10.1109/TCC.2018.2794344
17. 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). doi:10.1109/JIOT.2018.2792940
18. 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). doi:10.1145/3217871.3217876. https://doi.org/10.1145/3217871.3217876
19. 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). doi:10.1109/TPDS.2019.2929389
20. 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)
21. 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). doi:10.1109/DSDIS.2015.67
22. 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). doi:10.1109/UCC-Companion.2018.00035
23. 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). doi:10.1109/SEAA53835.2021.00029
24. 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. IGI global, Melbourne, Australia (2020). doi:10.4018/978-1-7998-2701-6.ch010
25. 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). doi:10.1109/JSAC.2019.2906745
26. 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). doi:10.1109/INFOCOM42981.2021.9488670
27. 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). doi:10.1109/CCGrid54584.2022.00102
28. 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). doi:10.1002/cpe.6290. https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.6290
29. Koopmans, T.C., Beckmann, M.: Assignment problems and the location of economic activities. Econometrica: journal of the Econometric Society, 53–76 (1957)
30. Blum, C., Roli, A.: Metaheuristics in combinatorial optimization: Overview and conceptual comparison. ACM Comput. Surv. **35**, 268–308 (2001). doi:10.1145/937503.937505
31. Glover, F.W.: Future paths for integer programming and links to artificial intelligence. Comput. Oper. Res. **13**, 533–549 (1986)
32. Baum, E.B.: Towards practical 'neural'computation for combinatorial optimization problems. In: AIP Conference Proceedings, vol. 151, pp. 53–58 (1986). American Institute of Physics
33. Mladenović, N., Hansen, P.: Variable neighborhood search. Computers & Operations Research **24**(11), 1097–1100 (1997). doi:10.1016/S0305-0548(97)00031-2
34. Feo, T.A., Resende, M.G.C.: A probabilistic heuristic for a computationally difficult set covering problem. Operations Research Letters **8**(2), 67–71 (1989). doi:10.1016/0167-6377(89)90002-3
35. Kirkpatrick, S., Gelatt, C., Vecchi, M.: Optimization by simulated annealing. Science (New York, N.Y.) **220**, 671–680 (1983). doi:10.1126/science.220.4598.671
36. Cerny, V.: Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. Journal of Optimization Theory and Applications **45**, 41–51 (1985). doi:10.1007/BF00940812
37. Holland, J.: Adaptation in natural and artificial systems. University of Michigan Press (1975)
38. Bozorg-Haddad, O., Afshar, A., Mariño, M.: Honey-bees mating optimization (hbmo) algorithm: A new heuristic approach for water resources optimization. Water Resources Management **20**, 661–680 (2006). doi:10.1007/s11269-005-9001-3
39. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proceedings

of ICNN'95 - International Conference on Neural Networks, vol. 4, pp. 1942–19484 (1995). doi:10.1109/ICNN.1995.488968

40. Dorigo, M., Birattari, M., Stutzle, T.: Ant colony optimization. IEEE Computational Intelligence Magazine **1**(4), 28–39 (2006). doi:10.1109/MCI.2006.329691
41. Koulamas, C., Antony, S., Jaen, R.: A survey of simulated annealing applications to operations research problems. Omega **22**(1), 41–56 (1994). doi:10.1016/0305-0483(94)90006-X
42. Connolly, D.T.: An improved annealing scheme for the qap. European Journal of Operational Research **46**(1), 93–100 (1990). doi:10.1016/0377-2217(90)90301-Q
43. 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). doi:10.1109/JVA.2006.44
44. Sengupta, S., Basak, S., Peters, R.A.: Particle swarm optimization: A survey of historical and recent developments with hybridization perspectives. Machine Learning and Knowledge Extraction **1**(1), 157–191 (2019). doi:10.3390/make1010010
45. Salman, A., Ahmad, I., Al-Madani, S.: Particle swarm optimization for task assignment problem. Microprocessors and Microsystems **26**(8), 363–371 (2002). doi:10.1016/S0141-9331(02)00053-4
46. Zhang, L., Chen, Y., Sun, R., Jing, S., Yang, B.: A task scheduling algorithm based on pso for grid computing. International Journal of Computational Intelligence Research **4**(1), 37–43 (2008)
47. Bertsimas, D., Tsitsiklis, J.: Simulated Annealing. Statistical Science **8**(1), 10–15 (1993). doi:10.1214/ss/1177011077
48. Ellison Geltman, K.: The Simulated Annealing Algorithm. http://katrinaeg.com/simulated-annealing.html (2014)
49. rezaee jordehi, A., Jasni, J.: Particle swarm optimisation for discrete optimisation problems: A review. Artificial Intelligence Review **43** (2014). doi:10.1007/s10462-012-9373-8
50. Shi, Y., Eberhart, R.: A modified particle swarm optimizer. In: 1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360), pp. 69–73 (1998). doi:10.1109/ICEC.1998.699146
51. Bansal, J.C., Singh, P.K., Saraswat, M., Verma, A., Jadon, S.S., Abraham, A.: Inertia weight strategies in particle swarm optimization. In: 2011 Third World Congress on Nature and Biologically Inspired Computing, pp. 633–640 (2011). doi:10.1109/NaBIC.2011.6089659
52. García, F., Moreno-Pérez, J.: Jumping frogs optimization: a new swarm method for discrete optimization. Technical report, Grupo de Computación Inteligente, Departamento de Estadística, I.O. y C., Instituto Universitario de Desarrollo Regional, University of La Laguna, Tenerife, Spain (January 2008)
53. Balaji, S., Revathi, N.: A new approach for solving set covering problem using jumping particle swarm optimization method. Natural Computing **15**, 503–517 (2015). doi:10.1007/s11047-015-9509-2
54. Gutiérrez, J., Landa-Silva, D., Moreno-Pérez, J.: Exploring feasible and infeasible regions in the vehicle routing problem with time windows using a multi-objective particle swarm optimization approach. In: Proceedings of the International Workshop on Nature Inspired Cooperatives Strategies for Optimization, NICSO, pp. 103–114 (2008). doi:10.1007/978-3-642-03211-0_9
55. Consoli, S., Moreno-Pérez, J., Darby-Dowman, K., Mladenovic, N.: Discrete particle swarm optimization for the minimum labelling steiner tree problem. Natural Computing **9**, 29–46 (2010). doi:10.1007/s11047-009-9137-9
56. Hahn, D.: A Day in the Life of a Netflix Engineer. `https://youtu.be/-mL3zT1iIKw?t=931`. AWS re:Invent 2015 - Las Vegas - Accessed: 2022-07-08 (2015)
57. Vogels, W.: Real-time graph of microservice dependencies at amazon.com in 2008. `https://twitter.com/Werner/status/741673514567143424`. CTO @ Amazon - Accessed: 2022-07-08 (2008)
58. Tewolde, G.S., Hanna, D.M., Haskell, R.E.: Enhancing performance of pso with automatic parameter tuning technique. In: 2009 IEEE Swarm Intelligence Symposium, pp. 67–73 (2009). doi:10.1109/SIS.2009.4937846
59. Park, M.-W., Kim, Y.-D.: A systematic procedure for setting parameters in simulated annealing algorithms. Computers & Operations Research **25**(3), 207–217 (1998). doi:10.1016/S0305-0548(97)00054-3
60. Santos, J., Wauters, T., Volckaert, B., De Turck, F.: Resource provisioning in fog computing: From theory to practice †. Sensors **19**, 2238 (2019). doi:10.3390/s19102238
61. SIGs, K.: Descheduler for Kubernetes. `https://github.com/kubernetes-sigs/descheduler`. Accessed: 2022-09-19 (2022)
62. Wikipedia: Control theory. `https://en.wikipedia.org/wiki/Control\_theory`. Accessed: 2022-09-20 (2022)
63. Kubernetes: Kubernetes Controllers. `https://kubernetes.io/docs/concepts/architecture/controller/`. Accessed: 2022-09-20 (2022)
64. pixielabs.ai: Pixie Overview. `https://docs.pixielabs.ai/about-pixie/what-is-pixie`. Accessed: 2022-09-22 (2022)
65. altexsoft.com: What is Data Hub: Purpose, Architecture Patterns, and Existing Solutions Overview. `https://www.altexsoft.com/blog/data-hub/`. Accessed: 2022-09-12 (2021)
66. cloudera.com: CLOUDERA DATA PLATFORM Data Hub, A comprehensive cloud-based Edge-to-AI analytics service. `https://www.cloudera.com/products/data-hub.html`. Accessed: 2022-09-12 (2022)
67. softwareag.com: Cumulocity IoT DataHub overview. `https://cumulocity.com/guides/datahub/datahub-overview/`. Accessed: 2022-09-12 (2022)
68. microsoft.com: Azure IoT Hub: Connect, monitor and manage billions of IoT assets. `https://azure.microsoft.com/en-us/services/iot-hub/`. Accessed: 2022-09-12 (2022)
69. amazon.com: AWS IoT Core: Easily and securely connect devices to the cloud. `https://aws.amazon.com/iot-core/`. Accessed: 2022-09-12 (2022)
70. google.com: Google Cloud IoT Core: A fully managed service to easily and securely connect, manage, and ingest data from globally dispersed devices. `https://cloud.google.com/iot-core`. Accessed: 2022-09-12 (2022)
71. 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). doi:10.1002/spe.2973. https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2973
72. google.com: GoogleCloudPlatform microservices-demo: Online Boutique. `https://github.com/GoogleCloudPlatform/microservices-demo`. Accessed: 2022-09-12 (2022)