©2024 IEEE. This article has been accepted for publication in IEEE Transactions on Services Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Link to publisher page: https://ieeexplore.ieee.org/document/11144757. Citation information: DOI: 10.1109/TSC.2025.3604373

Disruption-aware Microservice Re-orchestration for Cost-efficient Multi-cloud Deployments

Marco Zambianco, Silvio Cretti, Domenico Siracusa

***Abstract*—Multi-cloud environments enable a cost-efficient scaling of cloud-native applications across geographically distributed virtual nodes with different pricing models. In this context, the resource fragmentation caused by frequent changes in the resource demands of deployed microservices, along with the allocation or termination of new and existing microservices, increases the deployment cost. Therefore, re-orchestrating deployed microservices on a cheaper configuration of multi-cloud nodes offers a practical solution to restore the cost efficiency of deployment. However, the rescheduling procedure causes frequent service interruptions due to the continuous termination and rebooting of the containerized microservices. Moreover, it may potentially interfere with and delay other deployment operations, compromising the stability of the running applications. To address this issue, we formulate a multi-objective integer linear programming problem that computes a microservice rescheduling solution capable of providing minimum deployment cost without significantly affecting the service continuity. At the same time, the proposed formulation also preserves the quality of service (QoS) requirements, including latency, expressed through microservice colocation constraints. Additionally, we present a heuristic algorithm to approximate the optimal solution, striking a balance between cost reduction and service disruption mitigation. We integrate the proposed approach as a custom plugin of the Kubernetes scheduler. Results reveal that our approach significantly reduces multi-cloud deployment costs and service disruptions compared to the default Kubernetes scheduler implementation, while ensuring QoS requirements are consistently met.**

***Index Terms*—Microservice re-orchestration, cost minimization, service disruption minimization, multi-cloud systems, Kubernetes scheduler, optimization**

I. Introduction

In recent years, the multi-cloud computing paradigm has become increasingly adopted to distribute workloads across multiple public cloud regions that offer on-demand computational resources at different prices [1]. This business model gives service providers the opportunity to dynamically lease a number of virtual nodes tailored to the deployment of their cloud-native applications with limited operational efforts [2]. In this context, ensuring an efficient utilization of the resources is a determining factor to minimize the economic cost of running containerized microservices on rented multi-cloud virtual nodes [3]. To facilitate this objective and, in general, the management of the microservices lifecycle, container orchestration tools such as Kubernetes have become the standardized approach to automatize the allocation of workloads on the available nodes based on their resource requirements [4] [5].

M. Zambianco and S. Cretti are with Fondazione Bruno Kessler, Italy. D. Siracusa is with University of Trento, Italy. Corresponding author: M. Zambianco (email: mzambianco@fbk.eu)

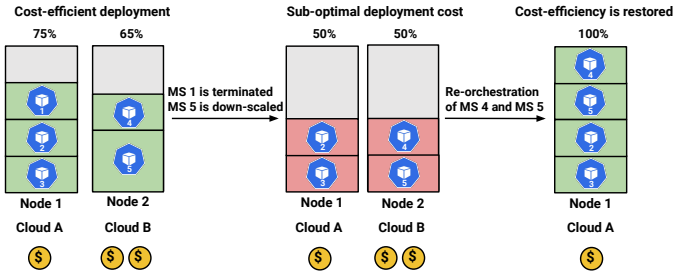


Fig. 1: Example of microservice re-orchestration restoring the deployment cost efficiency. Microservices are deployed across virtual nodes belonging to two cloud regions, with Cloud B being more expensive than Cloud A.

Leveraging these frameworks, the research activity has extended single-cloud orchestration solutions such as [6]–[9], which ignore the concept of deployment cost, to achieve cost-efficient application deployments while ensuring that QoS requirements are met. In particular, multi-cloud orchestration approaches primarily consist of scheduling algorithms that determine where to deploy microservices in order to reduce the deployment cost according to a fixed configuration of resource requirements and node availability within different geographically-distributed cloud regions [10]–[12]. At the same time, these schemes also ensure the co-location of delay-sensitive microservices within the same region to prevent QoS degradation caused by high inter-region network latency.

However, resource requirements of deployed microservices may change overtime due to various reasons including adjusting to more intensive task processing, managing seasonal spikes in user requests, accommodating the rollout of new microservice functionalities, mitigating the resource contention on overloaded nodes [13]. Similarly, resource availability on nodes can also fluctuate due to the deployment of new microservices to accommodate new applications or due to the termination of existing microservices that have completed their tasks, such as the training of machine learning models and processing of big data analytics [14]. The constant changes in resource requirements, along with the arrival and termination of microservices, progressively increase resource fragmentation, which in turn increases the demand of nodes needed to accommodate the re-configured applications [15]. As a consequence, the deployment cost-efficiency steadily deteriorates, since the initial node assignment computed for the original configuration of microservices becomes gradually suboptimal in minimizing costs.

To address this issue, re-orchestrating microservices (in

other words, the rescheduling of deployed microservices) offers a way to restore the cost-efficiency by re-allocating microservices on a cheaper configuration of multi-cloud nodes. In Fig. 1, we provide a simple example to better illustrate this aspect. In detail, the cost efficiency of the microservice deployment deteriorates due to a combination of resource downscaling and workload termination (note that the same effect is produced by combining subsequent workload downscaling/upscaling with new workload arrival/termination). In this scenario, the unused resources, that were originally reserved, can now be utilized to re-orchestrate microservices from Node 2 to Node 1, thus reducing the deployment cost as Node 2 can be terminated.

An effective and practical microservice re-orchestration solution has to account for the impact on service continuity caused by the rescheduling process. The rescheduling of deployed microservices is implemented by terminating existing containers and subsequently restarting them on the newly assigned destination nodes. This process causes service downtime which might not be tolerable for some applications since rescheduled microservices temporarily stop processing tasks. The severity of service disruption is exacerbated by the number of rescheduled microservices as the amount of pending unprocessed tasks increases. In addition, the re-orchestration of microservices also affects the system readiness to accommodate upcoming modifications to the configuration of deployed applications. In highly dynamic environments, the rescheduling procedure might conflict with the deployment operations (e.g., modification of environment variables, update of container images, interaction with monitoring and logging services, changing security policies) and may leave applications in an inconsistent and/or unresponsive state for an extended period of time [16] [17]. For this reason, it is important to complete the rescheduling procedure in a short amount of time to minimize the impact on the deployment stability. Overlooking these effects significantly reduces the benefit of potential cost savings achieved by the microservice re-orchestration, since frequent service interruptions undermine the dependability of the deployed applications. In this regard, existing multi-cloud orchestration schemes for cost minimization are designed under the assumption that microservices are scheduled for the first time, hence they are unsuited for microservice re-orchestration since they ignore the service disruption.

To fill this gap, we propose a disruption-aware re-orchestration algorithm that mitigates service disruption when rescheduling microservices. Our approach aims to minimize deployment costs in multi-cloud environments while preserving both service continuity and QoS requirements. Specifically, we design our rescheduling solution assuming that microservices are rescheduled using a rolling update strategy, which is a deployment option provided by the Kubernetes orchestrator. This mechanism reduces service downtime during rescheduling by first deploying an instance of rescheduled microservices on the new destination nodes before terminating the corresponding replicas at their original location [18]. From a resource allocation perspective, this effect requires accounting for the coexistence of the new and old microservice instances during the rescheduling phase, temporarily increasing the node resource utilization, in order to achieve an optimal rescheduling decision. We previously leveraged this rescheduling strategy to re-orchestrate at runtime microservices in order to minimize the deployment cost as described in [19]. In this work, we extend our former re-orchestration solution to also include the minimization of the number of rescheduled microservices and of the rescheduling duration. These additions allow to further mitigate the service disruption severity by avoiding unnecessary microservice migrations and overextended duration of the rescheduling procedure. Moreover, unlike the simulated environment used to assess the results of our previous work, we enhance the performance evaluation by implementing the proposed re-orchestration solution as a custom Kubernetes scheduler plugin. This allows us to benchmark the results against the default Kubernetes scheduler configuration, providing a more accurate performance assessment for practical scenarios. In detail, we summarize our contributions as follows:

- We formulate a multi-objective integer linear programming problem, based on a customized version of the Variable Cost Bin Packing Problem, to compute the optimal rescheduling strategy that minimizes i) the total economic cost associated to multi-cloud virtual nodes hosting microservices, ii) the number of rescheduled microservices and iii) the number of rescheduling slots. The first term addresses the cost minimization objective, where the second and third terms minimize service disruption. Moreover, we preserve the QoS performance by means of rescheduling constraints that enforce the co-location of communicating microservices having stringent latency requirements on nodes within the same cloud region.
- We propose a low-complexity heuristic algorithm to approximate the optimal solution using an iterative greedy approach. In detail, the designed algorithm maximizes the number of unused high-cost nodes by parallelizing the rescheduling of microservices from those nodes to the cheapest available nodes without violating colocation requirements. This strategy achieves a balanced trade-off between cost reduction, number of rescheduled microservice and rescheduling duration.
- We integrate the optimal and heuristic re-orchestration solutions as custom Kubernetes scheduler plugins in order to evaluate their performance on practical scenarios. Compared to the default Kubernetes scheduler implementation, the proposed schemes can provide higher cost reductions using a lower number of rescheduled microservices and a shorter rescheduling duration while guaranteeing a consistent fulfillment of QoS requirements.

The remainder of the paper is organized as follows. Section II covers the related work. In Section III, we outline the considered system model. Section IV presents the optimal microservice re-orchestration solution and a heuristic algorithm for approximating it. In Section V, we analyze the obtained results. Finally, we draw the conclusion in Section VI.

II. Related work

The advent of multi-cloud computing has increased the complexity of orchestration schemes due to the distributed nature of such system. An overview of the main challenges and proposed solutions in the context of multi-cloud orchestration can be found in [20] and in [21]. Given the focus on cost minimization, we compare our work with respect to microservice orchestration solutions that propose scheduling strategies to reduce deployment costs while balancing resource utilization and service performance.

The authors of [22] propose a graph clustering method for microservice placement in public clouds to optimize node resource usage and bandwidth, resulting in reduction of the associated deployment cost. The authors of [23] introduces a scheduling algorithm for Kubernetes to jointly enhance resource utilization while also improving load balance among the nodes. Their approach achieve lower deployment costs and decrease resource overruns. The authors of [24] proposes a microservice placement scheme, based on an improved version of genetic algorithm, that accounts for the sharing mechanism used by microservice instances to access common libraries in order to lower their resource usage and thus reducing their deployment cost. While these works offer different strategies for minimizing deployment costs by essentially improving the resource utilization efficiency of microservices, they are static approaches. Microservices are scheduled by assuming a fixed configuration of resource requirements and resource availability on the nodes. As a result, the performance of such solutions in scenarios where the increasing resource fragmentation deteriorates the deployment cost efficiency remains uncovered. In contrast, we specifically focus the latter problem and we design an optimization-based rescheduling strategy to efficiently repack existing workloads on the cheapest multi-cloud node configurations based on the current resource availability.

The authors of [25] develop an orchestration framework to efficiently scale resources assigned to data-intensive applications deployed across multiple geo-distributed nodes. Their scaling strategy improves the resource utilization efficiency of microservices, limiting the resource fragmentation and consequently the amount of multi-cloud nodes. Similarly, the authors [26] proposes a scheduling scheme that combines microservice scaling and allocation to optimally minimize the number of required virtual nodes based on the current resource consumption, which is dynamically adjusted based on the application load. The authors of [27] propose a microservice scheduling framework, based on a genetic algorithm, to relocate microservice on multi-cloud clusters in order to minimize the deployment cost and service latency. The authors of [28] introduce a dynamic resource management scheme that migrates deployed workloads based on their job execution time using an unsupervised learning approach. Their solution improve data center utilization, reducing the deployment cost by lowering energy expenditure. The authors of [29] present a rescheduling strategy to improve application service time by co-locating interdependent microservices on the same node in order to mitigate network latency while avoiding resource overloading. The proposed multi-objective problem formulation minimizes both inter-server network traffic and CPU throttling on nodes under varying loads. The authors of [30] propose an automated resource management system that increases the resource utilization of cloud-native applications while preserving their QoS performance. In particular, their approach leverage control theory to design an adaptive controller for each application that continuously right-sizes the resources assigned to each microservice based on performance feedback. These solutions address the limitations of static orchestration by rescheduling microservices to improve resource utilization, taking into account factors such as resource balance, QoS requirements, and costs. However, they lack strategies to mitigate service interruptions caused by frequent microservice reallocation. In contrast, our approach minimizes service disruptions during rescheduling by combining the usage of a rolling update deployment strategy with the minimization of the number of rescheduled microservices and rescheduling duration, ensuring cost-efficient deployments with minimal impact on service continuity.

The authors of [31] and [32] propose heuristic-based microservice rescheduling schemes to automatically co-locate microservices having high affinity values in terms of number of exchanged data packets in order to further improve the cost mitigation. The authors of [33] a scalable cluster scheduling mechanism built as Kubernetes extension plugin that adaptively balances the resource allocation of best-effort and latency-sensitive workloads. The aforementioned schemes enforce co-location requirements by imposing affinity rules between microservices, acting as soft constraints which might not be satisfied at runtime. In the context of microservice rescheduling, this approach performs poorly, since the co-locations requirements satisfied by the current deployment configuration could be possibly violated once the rescheduling phase is completed. For this reason, we first design an optimization problem where co-location requirements work as hard constraints to limit the possible rescheduling options, ensuring an optimal rescheduling configuration that reduce costs and guarantee QoS performance. This strategy is also embedded within the heuristic algorithm that approximates the optimal solution to prioritize QoS fulfillment.

Finally, in our previous work [19], we proposed a re-orchestration algorithm that uses an optimization and heuristic approach to reschedule microservices via a rolling-update deployment strategy. The goal was to minimize deployment costs while mitigating service disruption and preserving microservice colocation requirements. However, the approach assumed a predefined number of rescheduled microservices and sequential rescheduling, which introduced limitations. Specifically, fixing the number of microservices to reschedule can lead to an unnecessarily high number of microservice migrations. Additionally, in large-scale deployments, the sequential nature of rescheduling can result in prohibitively long durations as the number of rescheduled microservices increases. These issues can significantly impact service continuity. To address these shortcomings, in this work, we link the severity of service disruption with an increase in deployment cost. This enhancement enables the computation

of a rescheduling solution that automatically determines the minimum number of microservices that can be simultaneously repacked on a cheaper configuration of multi-cloud nodes. Consequently, the deployment cost can be reduced without causing a disproportionate service disruption.

III. System model

We introduce the main elements of the considered model that is composed by the multi-cloud architecture and by the microservice re-orchestrator module.

A. Multi-cloud microservice deployment model

We consider R geographically-distributed node clusters, each one owned and administrated by a different cloud provider (e.g., AWS, Microsoft Azure, Google Cloud) that offers computational resources in the form of virtual nodes. We indicate the set of virtual nodes hosted by each individual multi-cloud cluster as $N_0, ..., N_{R-1}$ whereas we represent the set of all virtual nodes as $N = \bigcup_{i=0}^{R-1} N_i$. Each node provides a maximum of $C_n = \{c_n^{(cpu)}, c_n^{(ram)}\}$ computational resources expressed as CPU resources and RAM resources, respectively. Due to the geo-distribution of the cloud infrastructure, virtual nodes belonging to different clusters experience some non-negligible latency denoted as $D_{n,n'} > 0$, $n \in N_i$, $n' \in N_{j \neq i}$. Conversely, we assume that intra-cluster network latency can be approximated as zero, thus $D_{n,n'} = 0, n, n' \in N_i$. The various nodes can be rented by some service provider at fixed price of $p_n C_n$, with p_n indicating the price per unit of resources, to deploy microservice-based applications encompassing different use cases such as e-commerce platforms, content streaming services, machine learning tasks. We remark that the price associated to each node is associated with the leasing of the whole instance, hence it is constant regardless of the amount of computational resource utilized by allocated microservices (i.e., 10% utilization rate of the node resources has the same cost of 90% utilization rate). We indicate the set of deployed applications as A and the set of microservices composing each application as $M_a, a \in A$. Moreover, we define the set of all microservices as $M = \bigcup_{a \in A} M_a$. Each microservice requires a minimum of $r_{a,m} = \{r_{a,m}^{(cpu)}, r_{a,m}^{(ram)}\}$ resources in order to be scheduled on a given node. We represent the assignment of microservices on the various virtual nodes using the indicator variable $s_{a,m,n} = 1$ if $m \in M_a, a \in A$ is allocated on node $n \in N$, $s_{a,m,n} = 0$ otherwise.

Moreover, to prevent the degradation of QoS performance caused by high inter-region network latency, microservices may be subject to some locality constraints which enforce their colocation on virtual nodes within the same cloud region. In detail, we refer to as $d_{a,m,m'}$ the maximum tolerable network latency between communicating microservices $m, m' \in M_a$ of application $a \in A$. Then, we express the location feasibility for each pair of communicating microservices by defining the indicator variable $\ell_{a,m,m'}^{(n,n')} = 1$ if $d_{a,m,m'} \geq D_{n,n'} \quad \forall m, m' \in M_a, \forall n, n' \in N$, $\ell_{a,m,m'}^{(n,n')} = 0$ otherwise. In other words, $\ell_{a,m,m'}^{(n,n')} = 1$ indicates that the additional network latency $D_{n,n'}$ that affects microservices m and m' on virtual nodes n and n' is below the maximum threshold $d_{a,m,m'}$ and thus the allocation of related microservices satisfies the QoS requirements. Based on this notation, we analytically represent a feasible deployment configuration as

$$S_A = \{s_{a,m,n} : \sum_{a' \in A} \sum_{m' \in M_a} s_{a',m',n'} r_{a',m'} \leq C_n \quad \wedge$$
$$s_{a,m',n'} = 1 \quad \text{if} \quad \ell_{a,m,m'}^{(n,n')} = 1, \forall m' \in M_a, n' \in N\} \quad (1)$$

Essentially, from the above definition, a feasible microservice deployment must not exceed the maximum capacity of the virtual nodes and, at the same time, must satisfy the colocation constraints.

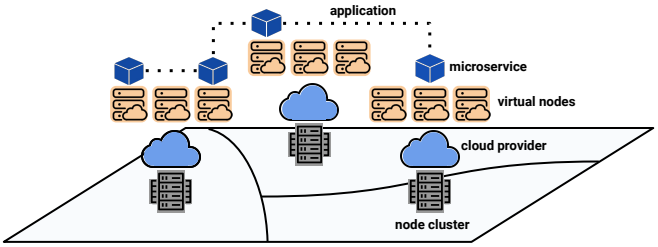


Fig. 2: Multi-cloud system model. Microservices of each application are deployed on various virtual nodes belonging to different geographically-distributed cloud providers.

B. Disruption-aware microservice re-orchestration model

We consider a multi-cloud environment where the resource requirements and/or the number of microservices deployed change over time. The service provider continuously monitors the cost-efficiency of the deployment and triggers a microservice re-orchestration at any point during the deployment lifecycle to restore cost optimality. Exampless of re-orchestration trigger conditionsinclude low resource utilization on some nodes, deployment cost exceeding a maximum cost budget, and availabilitylity of less expensive virtual nodes in a different cloud region. The re-orchestrator module is responsible for computing a new feasible deployment configuration that reallocates microservices onto a more cost-effective set of virtual nodes. Nodes left empty after the rescheduling phase are terminated, thus reducing overall costs. We model the execution of the rescheduling phase as a sequence of time slots in which one or multiple microservices are reallocated on the new destination nodes. In particular, indicating with T the set of time slots, the maximum number of time slots available is $|T| = |\bigcup_{a \in A} M_a|$, which corresponds to the limit scenario where all deployed microservices are individually rescheduled in a different time slot. The rescheduling phase affects the service continuity due to the inability to process tasks during the time period in which the container of a rescheduled microservice is restarted on the new destination node. Moreover, the duration of the rescheduling process can also affect applications' stability and interfere with the deployment operations, causing delays and further service interruptions. Although temporarily, this effect hinders the overall service performance as a part of the deployed applications is

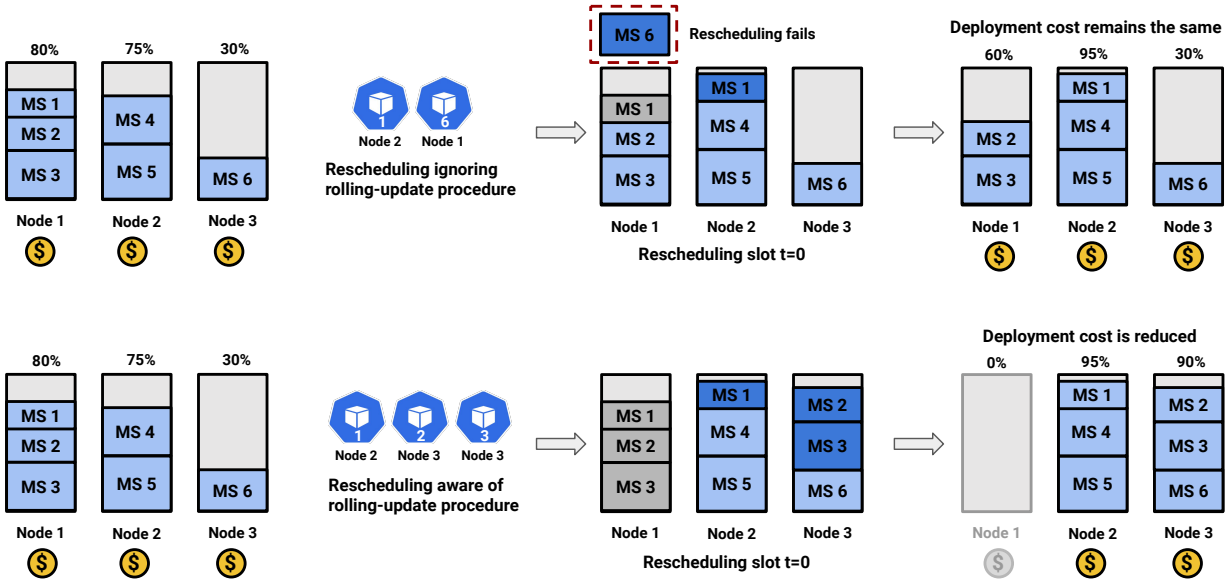


Fig. 3: Example showing the effect of rolling-update deployment strategy on deployment cost. Upper picture shows a microservice re-orchestration that does not account for the rolling-update procedure: the old instance of microservice 1 (gray color) prevents the rescheduling of microservice 6, denying the minimization of the deployment cost. Bottom picture shows a microservice re-orchestration that accounts for the rolling-update procedure: microservices 1,2 and 3 are successfully rescheduled (dark blue) on the designated nodes, leading to a reduced deployment cost through the termination of Node 1.

not fully operational. For example, time-sensitive applications may fail to process some tasks within the time deadline or machine learning pipelines may lose some training progress due to unresponsive microservices. Consequently, in addition to the final goal of restoring the cost-efficiency of deployment while preserving the co-location constraints, the computed microservice re-orchestration is also disruption-aware as it mitigates the level of service disruption generated during the rescheduling phase. In detail, we assume that the minimization of service disruption can be achieved by a rescheduling of microservices that satisfies three objectives.

- **Rolling-update redeployment strategy**: microservice rescheduling is operationally executed by first redeploying each microservice instance on the new node and, only once the rebooting process is completed, the old instance is terminated. This mechanism allows for reducing service interruptions as most tasks are processed by the old microservice instance while the new instance is restarting.
- **Minimization of the number of rescheduled microservices**: the number of rescheduled microservices is minimized to prevent unnecessary migrations that do not have benefits in terms of cost reduction, but can still affect service continuity due to delays in tasks execution.
- **Minimization of rescheduling phase duration**: the number of rescheduling slots is minimized to reduce possible conflicts with the deployment operations, possibly affecting the stability of applications and impairing their functionalities.

We acknowledge that the level of service disruption also depends on the functionalities and availability of each microservice within an application. For example, rescheduling a microservice that works as the front end of a web application is more disruptive than a microservice that handles the automation of e-commerce platform advertisement emails, as the former needs to frequently handle new user requests and dispatch them to other microservices. Similarly, rescheduling a microservice with multiple replicas has a negligible impact on the service continuity, as its workload is distributed across the other existing copies. Accounting for these effects increases the microservice re-orchestration complexity as they render the impact on the service performance more challenging to quantify, hence we leave them for a future analysis.

C. Impact of rolling-update deployment strategy on cost

The combination of a rolling-up update deployment strategy with minimization of the number of rescheduled microservices and rescheduling slots requires proper modeling of their dynamics to optimally minimize the deployment cost. During the rolling-update process, both old and rescheduled microservice instances are temporarily active on the original and destination nodes, respectively. The higher resource usage on the nodes increases the resource contention, that hampers the effectiveness of cost minimization by making the selection of which microservices to reschedule less obvious. In Fig. 3, we illustrate an example of microservice rescheduling to show how ignoring the rolling-update procedure can impact the deployment cost. The top picture shows a re-orchestration scheme that ignores the rolling-update rescheduling dynamic, hence it assumes that microservices can be instantaneously rescheduled on the destination nodes. Given the initial deployment configuration, the rescheduling of Microservice 1 on Node 2 and Microservice 6 on Node 1 would lower

the deployment cost and would provide minimum service disruption as two microservices are rescheduled within the same rescheduling slot. However, such a solution fails in practice, since the old instance of Microservice 1 is still active and prevents Microservice 6 from being rescheduled on Node 1. As a result, Microservice 6 remains allocated on Node 3, resulting in the same deployment cost as in the initial deployment configuration. Differently, the bottom picture shows a re-orchestration scheme that accounts for the rolling-update rescheduling dynamic when computing the microservice rescheduling. In this scenario, microservice 1, 2, and 3 are now rescheduled on Node 2 and Node 3. This solution overcomes the resource contention caused by the old microservice instances and successfully reduces the deployment cost through the termination of Node 1.

Given the impact of the rolling-update mechanism on cost performance, it is important to model this dynamic to design an effective re-orchestration scheme that optimally minimizes the deployment cost. To address this, the next section introduces an optimization problem that incorporates the effects of the rolling-update process on node resource utilization, ensuring that the proposed microservice rescheduling can be executed in practice.

IV. Problem formulation

We present the multi-objective optimization problem that computes the optimal microservice rescheduling solution, minimizing costs while mitigating service disruption and preserving QoS requirements. Moreover, we also propose a heuristic algorithm to approximate the optimal formulation.

A. Optimal re-orchestration solution

We model the cost minimization problem as a variation of Variable Cost Bin Packing problem where we extend the original formulation to include the minimization of service disruption and the fulfillment of QoS. We analytically model the rescheduling of microservices on the available virtual nodes introducing the binary variable $x^t_{a,m,n} = 1$, which indicates that microservice m of application a is rescheduled on node n at time slot t, $x^t_{a,m,n} = 0$ otherwise. Similarly, we introduce three optimization variables $y_n, v_{a,m}$ and z_t to evaluate the deployment cost, the number of rescheduled microservices, and the number of rescheduling slots, respectively. In detail, the binary variable $y_n = 1$ indicates that virtual node $n \in N$ is hosting any of the deployed microservices, whereas $y_n = 0$ indicates that node resources are not used by any microservice and therefore it can be terminated to reduce the deployment cost.The binary variable $v_{a,m} = 1$ indicates that microservice $m \in M_a$ of application $a \in A$ is rescheduled on a different virtual node, whereas $v_{a,m} = 0$ otherwise. Note that $v_{a,m}$ can be considered as a simplified version of the variable $x^t_{a,m,n}$ ignoring the destination node and time slot information and it is used to streamline the formulation of the objective function. Finally, the binary variable $z_t = 1$ indicates that time slot t is dedicated to reschedule at least one of the deployed microservices. Using these variables, we formulate the following optimization problem.

$$\min_{x,y,z} \sum_{n\in N} p_n C_n \cdot y_n + \alpha\Big(\sum_{a\in A}\sum_{m\in M_a} v_{a,m} + \sum_{t\in T} z_t\Big) \tag{2}$$

Subject to

$$\sum_{a\in A}\sum_{m\in M_a}\sum_{t\in T} r_{a,m} x^t_{a,m,n} \le y_n \Delta C_n \quad \forall n \in N \tag{3}$$

$$\sum_{t\in T} x^t_{a,m,n} + \sum_{t\in T} x^t_{a,m',n'} \le 1 + \ell^{(n,n')}_{a,m,m'} \quad \forall m, m' \in M_a, \forall a \in A, \forall n, n' \in N \tag{4}$$

$$\sum_{t'\in T_{t'<t}} \Big\{ \sum_{n'\in N} \Big[\sum_{a\in A}\sum_{m\in M_a} s_{a,m,n'}\Big(x^{t'}_{a,m,n} r_{a,m}\Big) - \sum_{a'\in A}\sum_{m'\neq m\in M_{a'}} s_{a',m',n}\Big(x^{t'}_{a',m',n'} r_{a',m'}\Big)\Big]\Big\} \le \Delta C_n \quad \forall n \in N, \forall t \in T \tag{5}$$

$$v_{a,m} = s_{a,m,n} - \sum_{t\in T} x^t_{a,m,n} \quad \forall a \in A, \forall m \in M, \forall n \in N \tag{6}$$

$$z_t \le \sum_{m\in M_a}\sum_{n\in N} x^t_{a,m,n} s_{a,m,n} \le z_t \cdot T \quad \forall a \in A, \forall t \in T \tag{7}$$

$$\sum_{t\in T}\sum_{n\in N} x^t_{a,m,n} = 1 \quad \forall m \in M_a, \forall a \in A \tag{8}$$

$$z_t \ge z_{t+1} \quad \forall t \in T \tag{9}$$

$$y_n, x^t_{a,m,n}, v_{a,m}, z_t \in \{0,1\} \quad \forall a \in A, \forall m \in M_a, \forall n \in N, \forall t \in T \tag{10}$$

The objective function (2) includes three components. The first addend computes the total deployment cost as a summation of the costs associated to each active multi-cloud virtual node. The second and third addends quantify the level of service disruption produced by the rescheduling process in the form of a penalty that depends on the total number of rescheduled microservices and the total number of needed rescheduling slots, respectively. The impact of these values can be modulated by tuning the parameter $\alpha \in [0, 1]$, that we refer to as disruption cost. Intuitively, higher values of α promote a microservice rescheduling solution that prioritizes service continuity over cost minimization.

Problem constraints, beside ensuring the solution feasibility in terms of resource occupation, implement the rolling update deployment strategy and enforce colocation requirements. In detail, the constraint (3) ensures that the new resource occupation of the deployment does not exceed the assigned node capacity, where $\Delta C_n = C_n - \sum_{a\in A}\sum_{m\in M} s_{a,m,n} r_{a,m}, \forall n \in N$ corresponds to the available capacity on each virtual node according to the initial deployment configuration S_A. Moreover, it links the cost optimization variable y_n with the rescheduling optimization variable $x^t_{a,m,n}$ by ensuring that

empty nodes cannot accommodate microservices. Constraint (4) enforces colocation constraints by guaranteeing that a given microservice pair m, m' cannot be allocated on nodes n and n' in different cloud regions if $\ell^{(n,n')}_{a,m,m'} = 0$. Constraint (5) accounts for the coexistence of the old and new microservice instance m rescheduled on a different node $n' \neq n$ on time slot t due to the rolling-update deployment strategy. This is achieved in practice by ensuring that the variation in resource occupation in every node n never exceeds the related residual capacity ΔC_n in each rescheduling slot $0, .., t', ..t$. Specifically, the first term inside the square brackets computes the total amount of resources consumed by microservices that were rescheduled on node n during any of the previous rescheduling slots $0, ..., t'$. In contrast, the second term computes the total amount of resources freed up on node n as a result of microservices being rescheduled to other nodes during time slots $0, ..., t'$. Note that the resources of microservices rescheduled on the same node (i.e., when $n = n'$) are not included, as they are in fact not rescheduled in practice. Constraint (6) ensures that a microservice is considered as rescheduled only if the destination node indicated by $x^t_{a,m,n}$ differs from the original node assignment indicated by the initial deployment configuration $s_{a,m,n}$. Constraint (7) bounds the number of rescheduling slots up to T which is the total number of microservices currently deployed and guarantees that multiple microservices can be rescheduled within the same time slot t. Constraints (8) and (9) guarantee that every microservice is uniquely assigned to a single node only and that the number of employed rescheduling slots are contiguous, respectively. Finally, constraint (10) expresses the integer nature of the problem.

Intuitively, the computational complexity of the proposed problem is NP-Hard as it can be reduced in polynomial time to the formulation of the classical Bin Packing problem [34]. The latter represents the corner scenario where all microservices are rescheduled at the same time to minimize the number of equally-priced virtual nodes belonging to a single cloud region. Therefore, due to its poor scaling capabilities, the resolution time is prohibitive for large-scale deployment scenarios composed by a high amount of microservices deployed across many multi-cloud virtual nodes. For this reason, in the next section, we lower the computation complexity by designing a heuristic approach for re-orchestrating microservices.

Algorithm 1 Heuristic

Input: $s_{a,m,n}, p_n, C_n, \Delta C_n, \ell^{(n,n')}_{a,m,m'}, r_{a,m}$
Output: $x^t_{a,m,n}$
Initialize $x^t_{a,m,n} = 0 \quad \forall m \in M_a, \forall a \in A, \forall n \in N, \forall t \in T$
Initialize $y_n = 1 \quad \forall n \in N$
Initialize rescheduling time slot $t = 0$
Initialize $\Delta C^{\text{roll}}_n = \Delta C_n$
Initialize $P_{\text{prev}} = +\infty$ and $P_{\text{current}} = \sum_{n \in N} p_n C_n$
while $P_{\text{current}} < P_{\text{prev}}$ **do**
    Initialize $L_N(n) = [0, ..., n, ..., N-1 : n \in N]$
    Sort L_N in ascending order of price $p_n C_n$
    for each $r \in R$ **do**
        Sort $L_{N_r}(n)$ in descending order of ΔC_n
    end for
    Initialize $\bar{L}_N(n) = [N-1, ..., n, ..., 0 : n \in N]$
    while $L_N(n)$ is not empty **do**
        Set n_{min} = $L_N(0)$
        Remove n_{min} from L_N and $\bar{L}_N$
        for each $\bar{n} \in \bar{L}_N(n)$ **do**
            Set n_{max} = $\bar{L}_N(\bar{n})$
            for each $m \in M$ allocated on n_{max} **do**
                Identify $a \in A$ such that $m \in M_a$
                if $\Delta C^{\text{roll}}_{n_{min}} \geq r_{a,m}$ **and** $\ell^{(n_{min}, n_{max})}_{a,m,m'} = 1, \quad \forall m' \in M_a$ **then**
                    Update $\Delta C^{\text{roll}}_{n_{min}} \leftarrow \Delta C^{\text{roll}}_{n_{min}} - r_{a,m}$
                    Set $x_{a,m,n_{min}} = 1$
                end if
            end for
        end for
    end while
    for each $n \in N$ **do**
        if $m \in M$ allocated on n is rescheduled **then**
            Identify $a \in A$ such that $m \in M_a$
            Update $\Delta C^{\text{roll}}_n \leftarrow \Delta C^{\text{roll}}_n + r_{a,m}$
        end if
        if $\Delta C^{\text{roll}}_n = C_n$ **then**
            Set $y_n = 0$
            Set $P_{\text{current}} \leftarrow P_{\text{current}} - p_n C_n$
        end if
    end for
    Update $t \leftarrow t + 1$
end while

B. Heuristic re-orchestration algorithm

The proposed heuristic algorithm strikes a balanced trade-off between the various optimization objectives. This approach eliminates the need for fine-tuning additional parameters that would otherwise depend on the specific microservice deployment configuration and infrastructure, making it easier to apply effectively in real-world scenarios. The general idea is to decompose the computation of the various objectives (cost, number of rescheduled microservices, number of rescheduling slots) while still ensuring that their minimization does not conflict with one another. To reach this goal, as highlighted in the previous example in Fig. 3, it is important to mitigate the resource contention during the rescheduling process caused by the rolling-update mechanism as it could deny the rescheduling of some microservices, preventing the achievement of a more cost-efficient node configuration.

Neglecting this aspect may unnecessarily prolong the rescheduling time and increase the number of rescheduled microservices, resulting in more severe service disruption. Specifically, the successful repacking of multiple microservices on the same destination node may become feasible only after multiple time slots where some microservices allocated on the target destination are preemptively migrated to release the required resources. This dynamic may further propagate to each preemptively rescheduled microservice that also requires a suitable amount of resource on the destination nodes.

Following this observation, we adopt a greedy iterative approach that in each rescheduling slot exploits the resource available on the various nodes to progressively repack microservices from the highest cost node to the cheapest node, thus maximizing the cost reduction. This strategy prevents the aforementioned effects caused by resource contention, as microservice are rescheduled only on nodes that can actually accommodate the required resources in a given slot without the need to preemptively rescheduling other microservices. Moreover, the selection of the cheapest available nodes as destination node in combination with the strict utilization of the available resources in each slot also limits the number of rescheduled microservices, as the latter are selected with the goal of improving the resource utilization of the least expensive nodes. We provide the pseudocode of the designed heuristic scheme in Algorithm 1. In detail, the algorithm takes as input the various parameters defining the deployment configuration and the multi-cloud infrastructure and provides as output the rescheduling of microservices on the available nodes. Lines 3-7 initialize the various parameters needed for the computation: ΔC_n^{roll} tracks the resource availability in each node accounting for the rolling update deployment logic, P_{prev} and P_{current} corresponds to the deployment cost before and after the rescheduling procedure in the time slot t, respectively. Lines 9-14 compute the list of nodes $L_N(n)$ sorted by increasing price, where equally priced nodes (i.e., nodes within the same cloud region) are sorted in decreasing order of resource occupation. Lines 16-17 select the cheapest and most loaded node, denoted as n_{min}, which is the element in the first position in $L_N(n)$, and remove it from the list. Lines 20-24 try to progressively reschedule microservices allocated on the most expensive and least loaded node, denoted as n_{max} and corresponding to the element in the last position in $L_N(n)$, to node n_{min} within the same time slot t. In particular, the rescheduling of microservice m occurs only if there are enough resources available (both for CPU and RAM resources) at the destination node n_{min}, where the resource occupation ΔC_n^{roll} is updated taking into account the coexistence of old and new microservice instances. Moreover, m is rescheduled only if it does not have colocation requirements in the scenario where n_{min} and n_{max} belong to different cloud regions. Once $L_N(n)$ is empty, it means that there are no resource available in time slot t to reschedule more microservices because of the rolling update procedure saturating the resources. Consequently, lines 30-36 release the resources occupied by the old microservice instances that have been rescheduled and update the deployment cost $P_{current}$ by subtracting the cost of the resulting empty nodes. Finally, the whole procedure 8-40 is repeated as long as the rescheduling of the microservice in each time slot provides a reduction in deployment cost, otherwise it ends. Intuitively, the proposed scheme enforces a coordinated microservice rescheduling to improve the resource utilization efficiency on the cheapest nodes by progressively migrating microservices originally allocated on most expensive nodes. At the same time, to minimize the number of rescheduled microservices, it prioritizes the re-orchestration of microservices from nodes with moderate resource consumption to nodes with higher resource consumption.

The computational complexity of the proposed approach can be computed as follows. The sorting of nodes based on price and resource occupation has complexity $O(N \cdot \log N)$. The selection of microservices to reschedule has complexity $O(M)$ as it is performed by evaluating microservices as they are assigned to the ordered list of nodes (e.g., microservices allocated to the most expensive nodes that have the highest amount of resources available are checked first). Similarly, resource occupation on the various nodes is computed by iterating over nodes and removing the resources of the rescheduled microservices, thus it has complexity $O(N)$. Overall, one iteration of the algorithm has complexity $O(M \cdot N \cdot \log N) + O(N)$, which can be further simplified asymptotically to $O(M \cdot N \cdot \log N)$.

V. Performance evaluation

A. Simulation setup

We evaluated the performance of the **Optimal** and **Heuristic** solutions with respect to the Kubernetes scheduler. The latter has become the de-facto standardized orchestrator tool to manage containerized applications, and thus offers a reliable benchmark to assess the quality of the proposed re-orchestration scheme. To provide a fair comparison with our scheme, we configure Kubernetes scheduler to heuristically achieve the same objectives of cost minimization, service continuity, and QoS requirements preservation when computing the microservice rescheduling. An overview of the functionalities and configuration parameters of the Kubernetes scheduler can be found in [35]. In our scenario, we minimize the deployment cost configuring the *MostAllocated* and *NodeAffinity* scheduling options to prioritize the repacking of microservices on the cheapest heavy-loaded virtual nodes. We promote the co-location of microservices within the same cloud region using the *PodAffinity* option, which acts as a soft constraint to incentivize the allocation of microservices with high affinity values within the same region. Finally, to better highlight the impact on the rescheduling logic when mitigating service disruption, we consider two scheduler configurations which employ the same options mentioned beforehand except for the microservice deployment strategy. The first configuration employs the microservice deployment strategy *Recreate*, which executes the microservice rescheduling by first terminating the old instance on the currently assigned node and then restarting the microservice container on the new destination node. The second configuration employs the microservice deployment strategy *RollingUpdate*. Consequently, configuring the scheduler using the *Recreate* deployment strategy affects more severely the service continuity, as microservices are not active during the rescheduling process. Based on these Kubernetes scheduler configurations, we consider two rescheduling algorithms to use as baseline:

- **Recreate (RC)**: this scheme employs Kubernetes scheduler configuration using the deployment strategy *Recreate* to reschedule all microservices on the cheapest cloud regions in order to minimize the deployment cost. This strategy ignores the impact of service disruption during the rescheduling phase for the sake of a better repacking

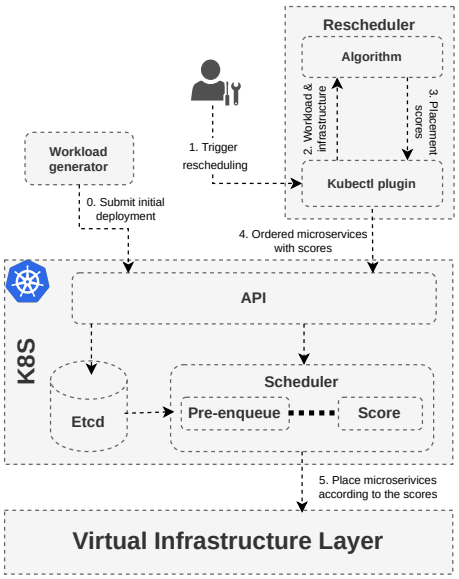


Fig. 4: Software architecture of the emulation environment

of microservices which is ultimately aimed at maximizing the reduction of the deployment cost.

- **Rolling-update (RU)**: this scheme employs Kubernetes scheduler configuration using the deployment strategy *RollingUpdate* to reschedule only microservices from the most expensive cloud regions in order to minimize the deployment cost. This strategy provides a disruption-aware rescheduling logic that reduces the deployment cost by rescheduling with the rolling update strategy only microservices that can potentially provide the highest cost savings.

We quantitatively evaluate the performance of our approach with respect to the benchmark schemes by computing the following metrics:

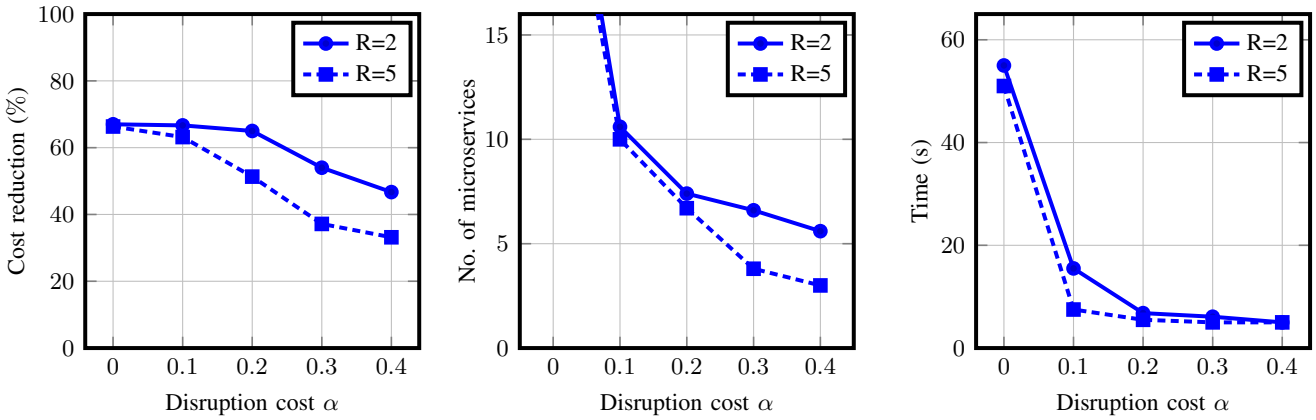
- **Deployment cost**: this metric measures the cost efficiency of the new microservice configuration on the available nodes after the rescheduling phase and is represented in the form of cost reduction with respect to the initial deployment cost.
- **Rescheduled microservices**: this metric contributes to the evaluation of service disruption and measures the number of rescheduled microservices required to achieve the new deployment configuration.
- **Rescheduling duration**: this metric contributes to the evaluation of service disruption and measures the time required to execute microservice rescheduling (step 5 in Fig. 4).
- **QoS fulfillment rate**: this metric assesses the effectiveness of the new deployment configuration in meeting QoS requirements by computing the fraction of satisfied colocation constraints after the microservice rescheduling,

We implemented a custom emulation environment that leverages Kubernetes orchestrator to manage the microservice rescheduling using a combination of Go and Python programming languages. In particular, we employed PySCIPOpt library to solve (2)-(7) using SCIP optimizer, which is a non-commercial solver for mixed integer programming [36]. To simulate various configurations of virtual nodes, microservices and resource requirements, we employed the software toolkit Kubernetes Without Kubelet (KWOK) [37]. This tool allows setting up fake virtual nodes, behaving like real ones, as well as to fully simulate the life-cycle of fake pods (pods are the atomic unit that can be managed by Kubernetes and can be considered as a microservice from a logical perspective). The advantage of this architectural design choice ensures a very low memory footprint that can be handled by a single physical machine. Nonetheless, since original control plane functionalities of Kubernetes, including the scheduler implementation, are preserved, we can reliably analyze the performance of our re-orchestration scheme without compromising the accuracy of results, ensuring their applicability in real-world scenarios.

In Fig. 4, we provide an overview of the system architecture used to execute the experiments along with the typical workflow, that consists of the following steps:

1) The `Workload generator` provides a feasible deployment configuration S_A consisting of a set of microservice-based applications with predefined resource requirements. This configuration is deployed using the default Kubernetes orchestrator (0.)
2) The `Service provider` triggers the rescheduling of microservices due to a low resource utilization efficiency of the current microservice configuration (1.)
3) A custom `Kubectl plugin`, which is the component allowing to communicate with Kubernetes control plane, collects the status of the virtual infrastructure and the workload deployed and invokes the `Algorithm` module (2.)
4) The `Algorithm` module implements the Optimal and Heuristic schemes and outputs the new placement for each microservice as well as the related rescheduling time slot in which to execute the node reassignment. This information is sent back to the `Kubectl plugin` in the form of placements and ranking scores for each microservice (3.)
5) The `K8s Scheduler` receives the ordered list of microservices to be redeployed. Two custom plugins have been added to the `K8s Scheduler` at the extension points `Pre-Enqueue` and `Score`: the first one allows to order the microservices according to what requested by the `Algorithm`, while the second one applies the scores (4.).
6) Finally, the planned rescheduling of microservices is executed on the selected nodes by the `K8s Scheduler` (5.).

We generate the virtual infrastructure as virtual nodes of equal capacity C_n grouped in R differently-priced cloud regions having a linearly increasing cost per unit of resources (for example, assuming R=3, each region would have $p_n = 1$ for region 1, $p_n = 2$ for region 2 and $p_n = 3$ for



(a) Deployment cost (b) Rescheduled microservices (c) Rescheduling duration

Fig. 5: Optimal microservice rescheduling computed with different values of disruption cost α when the number of cloud region is $R = 2$ and $R = 5$. The number of microservices is $M = 32$. The number of nodes is $N = 10$.

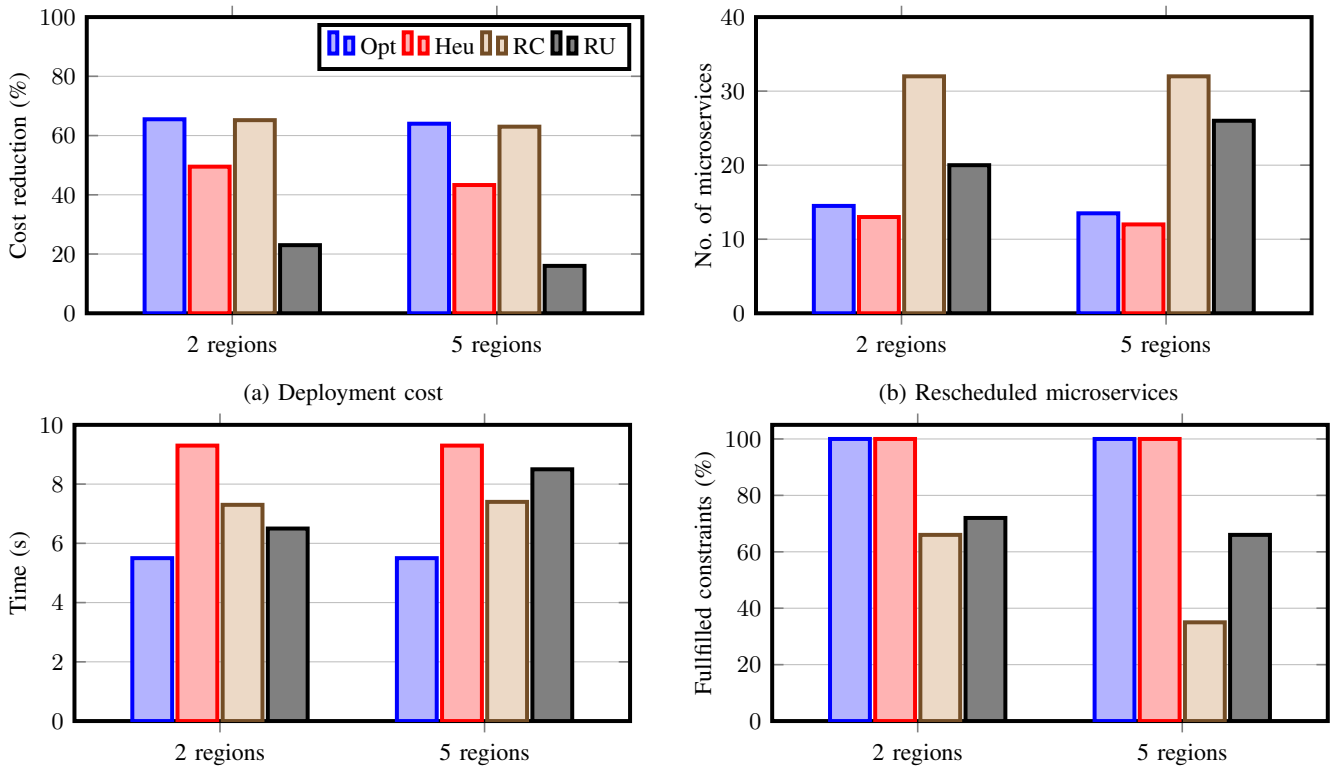
TABLE I: Simulation parameters

Number of applications	{8, 25}
Microservices per application	4
Number of cloud regions	{2, 5}
Inter-region network latency	50ms
Number of virtual nodes	{10, 27}
Normalized cost per multi-cloud node	$\{1, 2, ..., 5\}$
Number of vCPU	8 cores
Available RAM	32 GB
Microservice CPU request	$[0.5, 2.5]$ vCPU
Microservice RAM request	$[0.2, 0.3]$ GB

region 3). The feasible deployment of microservice S_A is computed following three main steps. First, we randomly generated the resource requirements for each microservice according to a uniform distribution of $[0.5, 2.5]$ for CPU resource and $[0.2, 0.3]$ for RAM resources. We employed these values by analyzing Alibaba cluster traces, which are a publicly available collection of statistics of real microservices measured on the Alibaba cloud provider infrastructure [38]. Then, to artificially produce a fragmented resource utilization reflecting a deployment configuration where the microservice had various modifications to the resource requirements as well as multiple arrival/termination processes, we randomly allocated the microservices among the various virtual nodes enforcing an average resource occupation within $50\% - 70\%$ of the maximum node capacity. Finally, to ensure that all QoS requirements were met prior to the re-orchestration process, we retrospectively defined the co-location constraints $\ell_{a,m,m'}^{(n,n')}$ for the deployed microservices. These requirements were computed by randomly sampling a maximum latency within the range of $[20\text{ms}, 100\text{ms}]$. The lower bound represents latency-critical communication flows, while the upper bound corresponds to latency-agnostic flows. We resume the main configuration parameters in Table I. We generated 50 feasible deployment configuration S_A for each combination of number of microservices and virtual nodes grouped in different cloud regions and we averaged the results obtained with the various schemes.

B. Optimal solution analysis

In Fig. 5, we initially discuss the behavior of the optimal scheme for different values of the disruption cost α in order to better highlight the general relationship between deployment cost and service disruption. We consider a scenario with $A = 8$ applications of $M = 4$ microservices each allocated on $N = 10$ virtual nodes equally distributed in $R = 2$ region and $R = 5$ regions, respectively. In detail, in Fig. 5a, the increase of the service disruption cost decreases ther cost minimization gain. This trend stems from an increase in the penalty magnitude applied to the objective function (2) when rescheduling microservices across multiple time slots. This dynamic incentives a rescheduling solution that greatly limits the number of reallocated microservices and that employs few rescheduling time slots, thus decreasing the rescheduling duration, as shown in Fig. 5b and in Fig. 5c, respectively. This effect is more exacerbated when the number of cloud regions is $R = 5$ compared to $R = 2$ since the more challenging QoS requirements restricts the possible rescheduling options. As a matter of fact, the repacking of microservices in cheaper cloud regions is exclusively performed only if the related nodes have enough resources to fully accommodate microservices with co-location requirements. Although the trade-off between cost mitigation and service disruption becomes increasingly evident with higher values of the disruption cost, we observe that for $\alpha = 0.1$, the deployment cost performance are essentially comparable to the one obtained with $\alpha = 0$. The latter provides a rescheduling solution that ignores the impact of rescheduled microservices and rescheduling duration on the service continuity. Therefore, for the considered scenario, $\alpha = 0.1$ provides the most efficient configuration for the optimal scheme that minimizes the service disruption without deteriorating the cost minimization performance.



(a) Deployment cost

(b) Rescheduled microservices

(c) Rescheduling duration

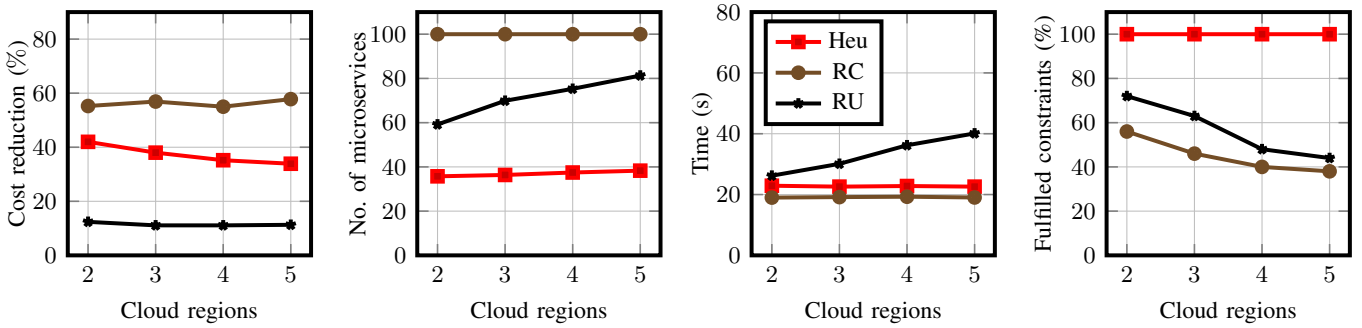
(d) QoS fulfillment rate

Fig. 6: Microservice rescheduling performance obtained by the Optimal (computed with $\alpha = 0.1$), Heuristic, RC and RU schemes when the number of cloud regions is $R = 2$ and $R = 5$. The number of microservices is $M = 32$. The number of nodes is $N = 10$.

C. Heuristic solution analysis

In Fig. 6, we extend the performance analysis by including the Heuristic scheme and we compare the results obtained with respect to the benchmark schemes RC and RU. In Fig. 6a, we observe that the heuristic scheme outperforms the RU scheme, which employs a scheduler configuration that tries to actively mitigate service disruption, in terms of cost performance. On the other hand, the RC scheme provides a cost reduction which is comparable to the Optimal scheme and outperforms the Heuristic solution by some margin. However, such performance is achieved by totally ignoring the service continuity degradation. Indeed, RC scheme employs the deployment strategy *Recreate* in combination with a complete rescheduling of all microservices. Therefore, by redeploying all the applications from scratch, it can achieve better repacking performance since nodes are initially empty and resources can be used more efficiently. The downside of this approach, as shown in Fig. 6b, produces the highest number of rescheduled microservices that are first terminated before being reallocated to the new destination node. Conversely, the heuristic scheme reschedules a significantly lower number of microservices, which is comparable to the optimal solution, by selectively repacking only microservices that offer the largest cost reduction based on the available resources in the current rescheduling time slot. This strategy also reduces the number of rescheduled microservices compared to RU scheme, which ignores the increase in resource contention caused by the coexistence of old and new microservice instances within the same rescheduling slot. Overlooking this effect produces an uncoordinated rescheduling of microservices that does not only degrade cost performance but also extends the rescheduling duration as depicted in Fig. 6c. Furthermore, as shown in Fig. 6d, the heuristic scheme fully preserves the QoS performance as it achieves the maximum QoS fulfillment rate like the optimal solution. In this regard, both RC and RU achieve poor results, which worsen in the scenario with $R = 5$. The heuristic approach constrains the rescheduling of microservices with colocation requirements within the same cloud region of origin. Conversely, RC and RU try to simultaneously achieve cost efficiency and QoS preservation at the same time by combining the contribution of $NodeAffinity$ and $PodAffinity$ scores, resulting in a poor trade-off of both metrics. This strategy performs the worst with RC as the latter can potentially violate a higher amount of QoS requirements since it reschedules more microservices compared to RU.

We also provide the results obtained on a larger virtual infrastructure and microservice deployment configuration in order to showcase performance on a more realistic multi-cloud scenario. Due to the high computational complexity of the optimal solution, we only show the results of the heuristic scheme, whose lower complexity ensures its applicability in larger multi-cloud computing environments. In Fig. 7, we show the overall performance achieved by the heuristic scheme for



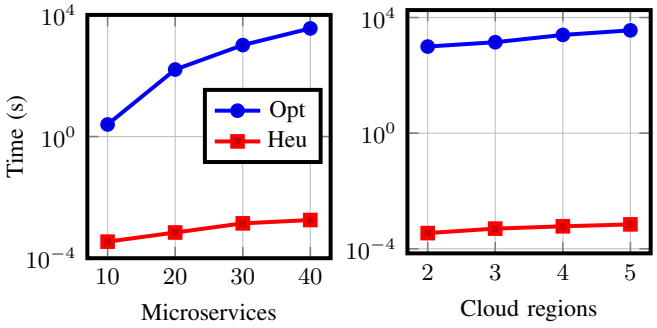
(a) Deployment cost (b) Rescheduled microservices (c) Rescheduling duration (d) QoS fulfillment rate

Fig. 7: Microservice rescheduling performance obtained with the Heuristic, RC and RU schemes as the number of cloud region increases from $R = 2$ to $R = 5$. The number of microservices is $M = 100$. The total number of nodes is $N = 27$.

an increasing number of multi-cloud regions starting from $R = 2$ up to $R = 5$ when total number of applications is $A = 25$ (i.e., $M = 100$ microservices) and the total number of nodes $N = 27$. In general, the same observations made for the scenario with $M = 32$ can be extended also for this case. The heuristic scheme ensures the most efficient trade-off in terms of cost reduction, service disruption, and QoS preservation. In particular, as shown in Fig. 7a, the cost minimization performance achieved by the heuristic scheme decreases with the number of regions. This is caused by a more restrictive rescheduling of microservices, which is needed to avoid the separation of colocated microservices in different cloud regions. As shown in Fig. 7d, this conservative approach allows the heuristic scheme to maximize the QoS fulfillment rate regardless of the number of cloud regions. This is not the case for the RU and RC schemes which perform visibly worse than the scenario with $M = 32$ microservices. Furthermore, the larger configuration of microservices and nodes also amplifies the resource contention issue that impacts more severely the rescheduling duration of the RU scheme, as shown in Fig. 7c. In contrast, it moderately affects the heuristic scheme, confirming its capability to effectively parallelize the microservice rescheduling process. In addition, as shown in Fig. 7b, the heuristic scheme reschedules the same number of microservices as the number of regions increases. Indeed, it repacks microservices based on a combination of resource availability and cost reduction gains that automatically bound the number of rescheduling opportunities. In contrast, the RU scheme repacks microservices based only on cost metrics, and thus their number depends on the number of regions. As a result, from a service disruption standpoint, the RU produces progressively poorer performance compared to the heuristic as the number of multi-cloud region increases.

D. Computational complexity analysis

In Fig. 8, we discuss the computational time performance of the optimal and heuristic schemes to provide some insights about their feasibility in real-world scenarios. Specifically, in Fig. 8a, we compare the scalability performance of the optimal and heuristic schemes by showing the time required to compute a rescheduling solution as the number of microservices increases from $M = 10$ to $M = 40$. Although the scenario complexity is rather modest, the convergence time of the optimal solution rises quickly, making it a prohibitive option for large and highly dynamic microservice deployments subject to frequent re-orchestrations. In contrast, the heuristic solution proves to be a viable alternative, as it requires significantly less time and scales effectively with the growing number of microservices. Similarly, in Fig. 8b, we compare the resolution time as the number of cloud regions increases. Higher number of cloud regions renders the QoS fulfillment more challenging due to the higher number of colocation requirements, prolonging the resolution time. Overall, this configuration parameter has a moderate impact on the scalability performance of both schemes. Nonetheless, the heuristic solution is way less affected than the optimal scheme, which still suffers from a visible performance degradation. This result showcases the computational efficiency of the proposed heuristic approach and promotes its applicability on practical multi-cloud environments.



(a) Resolution time for $R = 5$. (b) Resolution time for $M = 40$.

Fig. 8: Computational performance of the Optimal and Heuristic schemes obtained with different configuration of microservices and cloud regions. The number of nodes is $N = 10$.

VI. Conclusion

We considered the problem of cost minimization in multi-cloud environments, that consists of differently-priced and geographically distributed virtual nodes offered by different

cloud providers. In this context, the variable resource requirements and lifecycle dynamism typical of microservice deployments progressively deteriorates the resource utilization efficiency, leading to higher deployment costs. To address this issue, we designed a re-orchestration scheme that reschedules deployed microservices in order to reduce the resource fragmentation, thereby reducing the deployment cost. At the same time, the scheme mitigates service disruptions caused by the rescheduling process, which temporarily affect applications dependability, and preserves microservice colocation requirements within the same cloud region. In particular, we enhanced service continuity by combining the use of a rolling-update deployment strategy with the minimization of the number of rescheduled microservices and rescheduling duration. We analytically incorporated the service disruption model within the deployment cost model and we formulated a multi-objective linear integer problem to compute the optimal rescheduling solution satisfying the aforementioned objectives. We approximated the optimal solution computation by designing a heuristic scheme based on a greedy iterative approach that ensures a balanced trade-off between cost minimization and service disruption. We evaluated the performance of the optimal and heuristic schemes by integrating both approaches as a custom Kubernetes scheduler plugin. We compared the results against the default Kubernetes scheduler that was configured to achieve the same optimization objectives as of our solution. The results show that our approach provides the best compromise between cost minimization, service disruption mitigation and QoS preservation compared to the benchmark schemes.

ACKNOWLEDGEMENTS

This work has received funding from the EU Horizon Europe R&I Programme under Grant Agreement no. 101070473 (FLUIDOS).

REFERENCES

[1] J. Hong, T. Dreibholz, J. A. Schenkel, and J. A. Hu, "An overview of multi-cloud computing," in *Web, Artificial Intelligence and Network Applications*. Springer, 2019, pp. 1055–1068.

[2] J. Alonso, L. Orue-Echevarria, V. Casola, A. I. Torre, M. Huarte, E. Osaba, and J. L. Lobo, "Understanding the challenges and novel architectural models of multi-cloud native applications–a systematic literature review," *Journal of Cloud Computing*, vol. 12, no. 1, p. 6, 2023.

[3] G. Chatzithanasis, E. Filiopoulou, C. Michalakelis, and M. Nikolaidou, "Exploring cost-efficient bundling in a multi-cloud environment," *Simulation Modelling Practice and Theory*, vol. 111, p. 102338, 2021.

[4] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying microservice based applications with kubernetes: Experiments and lessons learned," in *2018 IEEE 11th international conference on cloud computing (CLOUD)*. IEEE, 2018, pp. 970–973.

[5] K. Senjab, S. Abbas, N. Ahmed, and A. u. R. Khan, "A survey of kubernetes scheduling algorithms," *Journal of Cloud Computing*, vol. 12, no. 1, p. 87, 2023.

[6] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, "mck8s: An orchestration platform for geo-distributed multi-cluster environments," in *2021 International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2021, pp. 1–10.

[7] Z. Wang, M. Goudarzi, J. Aryal, and R. Buyya, "Container orchestration in edge and fog computing environments for real-time iot applications," in *Computational Intelligence and Data Analytics: Proceedings of ICCIDA 2022*. Springer, 2022, pp. 1–21.

[8] G. Monaco, G. Gala, and G. Fohler, "Shared resource orchestration extensions for kubernetes to support real-time cloud containers," in *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2023, pp. 97–106.

[9] A. Alelyani, A. Datta, and G. M. Hassan, "Optimizing cloud performance: A microservice scheduling strategy for enhanced fault-tolerance, reduced network traffic, and lower latency," *IEEE Access*, 2024.

[10] S. Rac and M. Brorsson, "Cost-aware Service Placement and Scheduling in the Edge-Cloud Continuum," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 2, pp. 1–24, Jun. 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3640823

[11] L. Luo, G. Zhao, H. Xu, Z. Yu, and L. Xie, "Achieving Cost Optimization for Tenant Task Placement in Geo-Distributed Clouds," *IEEE/ACM Transactions on Networking*, vol. 32, Apr. 2024, conference Name: IEEE/ACM Transactions on Networking. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10273228

[12] T. Shi, H. Ma, G. Chen, and S. Hartmann, "Auto-Scaling Containerized Applications in Geo-Distributed Clouds," *IEEE Transactions on Services Computing*, vol. 16, no. 6, pp. 4261–4274, Nov. 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10255283/

[13] A. F. Baarzi and G. Kesidis, "Showar: Right-sizing and efficient scheduling of microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 427–441.

[14] M. Xu, L. Yang, Y. Wang, C. Gao, L. Wen, G. Xu, L. Zhang, K. Ye, and C. Xu, "Practice of alibaba cloud on elastic resource provisioning for large-scale microservices cluster," *Software: Practice and Experience*, vol. 54, no. 1, pp. 39–57, 2024.

[15] O. Larsson, C. Klein, and E. Elmroth, "The impact of directed pod eviction on kubernetes resource utilization," in *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2023, pp. 81–90.

[16] C. T. Joseph and K. Chandrasekaran, "Nature-inspired resource management and dynamic rescheduling of microservices in cloud datacenters," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 17, p. e6290, 2021.

[17] H.-C. Jang and S.-Y. Luo, "Enhancing node fault tolerance through high-availability clusters in kubernetes," in *2023 IEEE 3rd International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB)*. IEEE, 2023, pp. 30–35.

[18] V. Singh and S. K. Peddoju, "Container-based microservice architecture for cloud applications," in *2017 International Conference on Computing, Communication and Automation (ICCCA)*. IEEE, 2017, pp. 847–852.

[19] M. Zambianco, S. Cretti, and D. Siracusa, "Cost minimization in multi-cloud systems with runtime microservice re-orchestration," in *2024 27th Conference on Innovation in Clouds, Internet and Networks (ICIN)*. IEEE, 2024, pp. 65–72.

[20] O. Tomarchio, D. Calcaterra, and G. Di Modica, "Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks," *Journal of Cloud Computing*, vol. 9, pp. 1–24, 2020.

[21] B. Wang, C. Wang, Y. Song, J. Cao, X. Cui, and L. Zhang, "A survey and taxonomy on workload scheduling and resource provisioning in hybrid clouds," *Cluster Computing*, vol. 23, pp. 2809–2834, 2020.

[22] A. Aznavouridis, K. Tsakos, and E. G. Petrakis, "Micro-service placement policies for cost optimization in kubernetes," in *International conference on advanced information networking and applications*. Springer, 2022, pp. 409–420.

[23] T. Li, L. Qiu, F. Chen, H. Chen, and N. Zhou, "Carokrs: Cost-aware resource optimization kubernetes resource scheduler," in *2024 9th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*. IEEE, 2024, pp. 127–133.

[24] Z. Ding, S. Wang, and C. Jiang, "Kubernetes-oriented microservice placement with dynamic resource allocation," *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 1777–1793, 2022.

[25] F. Jiang, K. Ferriter, and C. Castillo, "A cloud-agnostic framework to enable cost-aware scheduling of applications in a multi-cloud environment," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–9.

[26] S. Wang, Z. Ding, and C. Jiang, "Elastic scheduling for microservice applications in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 98–115, 2020.

[27] Y. Aldwyan, R. O. Sinnott, and G. T. Jayaputera, "Elastic deployment of container clusters across geographically distributed cloud data centers for web applications," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 21, p. e6436, 2021.

[28] S. A. Khan, M. Abdullah, W. Iqbal, M. A. Butt, F. Bukhari, and S.-U. Hassan, "Automatic migration-enabled dynamic resource management

for containerized workload," *IEEE Systems Journal*, vol. 17, no. 2, pp. 2378–2389, 2022.
[29] V. Bracke, J. Santos, T. Wauters, F. De Turck, and B. Volckaert, "A multiobjective metaheuristic-based container consolidation model for cloud application performance improvement," *Journal of Network and Systems Management*, vol. 32, no. 3, p. 61, 2024.
[30] Y. Sfakianakis, M. Marazakis, and A. Bilas, "Skynet: Performance-driven resource management for dynamic workloads," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 2021, pp. 527–539.
[31] Z. Zhong and R. Buyya, "A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources," *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 2, pp. 1–24, 2020.
[32] A. R. Sampaio, J. Rubin, I. Beschastnikh, and N. S. Rosa, "Improving microservice-based applications with runtime placement adaptation," *Journal of Internet Services and Applications*, vol. 10, no. 1, pp. 1–30, 2019.
[33] X. Zhang, L. Li, Y. Wang, E. Chen, and L. Shou, "Zeus: Improving resource efficiency via workload colocation for massive kubernetes clusters," *IEEE Access*, vol. 9, pp. 105 192–105 204, 2021.
[34] S. Martello and P. Toth, "Lower bounds and reduction procedures for the bin packing problem," *Discrete applied mathematics*, vol. 28, no. 1, pp. 59–70, 1990.
[35] Kubernetes, "Scheduler configuration," https://kubernetes.io/docs/reference/scheduling/config/, [Online; accessed October 2023].
[36] S. Maher, M. Miltenberger, J. P. Pedroso, D. Rehfeldt, R. Schwarz, and F. Serrano, "Pyscipopt: Mathematical programming in python with the scip optimization suite," in *Mathematical Software–ICMS 2016: 5th International Conference, Berlin, Germany, July 11-14, 2016, Proceedings 5*. Springer, 2016, pp. 301–307.
[37] S. Zhang, W. Huang, and Y. Zhuang. Kubernetes without kubelet. [Online]. Available: https://kwok.sigs.k8s.io
[38] B. Everman, N. Rajendran, X. Li, and Z. Zong, "Improving the cost efficiency of large-scale cloud systems running hybrid workloads-a case study of alibaba cluster traces," *Sustainable Computing: Informatics and Systems*, vol. 30, p. 100528, 2021.



Marco Zambianco is a researcher at Fondazione Bruno Kessler in Trento, Italy. He received his Master's Degree in Telecommunication Engineering from University of Padova and his Ph.D. (with honors) in Information Technology from Politecnico di Milano in 2022. His research interests include microservice orchestration in distributed cloud computing and cybersecurity.



Silvio Cretti is a software architect at Fondazione Bruno Kessler (FBK) in Trento, Italy. He received his Master's Degree in Physics at University of Trento. Before joining FBK, he worked as software engineer, architect and technical leader for Telecom Italia group. His main technical interests are related to distributed and heterogeneous clouds, container orchestration, workload placement and cloud-native applications.



Domenico Siracusa is associate professor at the University of Trento. Previously, he was the head of the Robust and Secure Distributed Computing (RiSING) research unit at Fondazione Bruno Kessler (FBK). He received his PhD in Information Technology at Politecnico di Milano in 2012 and earned an Executive MBA in Business Innovation at the MIB Trieste School of Management in 2022. His research interests include infrastructure security and robustness, service orchestration and management, cloud and fog computing, and SDN/NFV and virtualization. Domenico authored more than 100 publications appeared in international peer reviewed journals and in major conferences on computing and networking technologies. Domenico was project manager and technical leader of relevant EU-funded projects, such as the H2020 EU-Korea DECENTER, H2020 ACINO and EIT Digital DigiFlow projects, and was involved as principal investigator, workpackage leader or contributor in other FP7/H2020, EIT digital and commercial projects.