Journal of Internet Services and Applications

RESEARCH Open Access

Improving microservice-based applications with runtime placement adaptation

Adalberto R. Sampaio Jr.[1*], Julia Rubin[2], Ivan Beschastnikh[3] and Nelson S. Rosa[1]

Abstract

Microservices are a popular method to design scalable cloud-based applications. Microservice-based applications (μApps) rely on message passing for communication and to decouple each microservice, allowing the logic in each service to scale independently.

Complex μApps can contain hundreds of microservices, complicating the ability of DevOps engineers to reason about and automatically optimize the deployment. In particular, the performance and resource utilization of a μApp depends on the placement of the microservices that compose it. However, existing tools for μApps, like Kubernetes, provide minimal ability to influence the placement and utilization of a μApp deployment.

In this paper, we first identify the runtime aspects of microservice execution that impact the placement of microservices in a μApp. We then review the challenges of reconfiguring a μApp based on these aspects. Our main contribution is an adaptation mechanism, named REMaP, to manage the placement of microservices in an μApp automatically. To achieve this, REMaP uses microservice affinities and resource usage history. We evaluate our REMaP prototype and demonstrate that our solution is autonomic, lowers resource utilization, and can substantially improve μApp performance.

Keywords: Microservices, Runtime adaptation, Optimization

1 Introduction

As business logic moves into the cloud, developers need to orchestrate not just the deployment of code to cloud resources but also the distribution of this code on the cloud platform. Cloud providers offer pay-as-you-go resource elasticity, and virtually infinite resources, such as CPU, memory, disk, and network bandwidth. The management of these cloud resources, however, is a major challenge, leading to new roles like DevOps engineers. In this context, microservices have become an essential mechanism to provide the necessary deployment flexibility and at the same time take advantage of abundant resources [1].

A *microservice* is a decoupled and autonomic software, having a specific functionality in a bounded context. The decoupling and well-defined interfaces provide the ability for microservice-based applications (μApps) to scale in/out seamlessly and for developers to perform upgrades by deploying new service versions, all without halting the μApp. The decoupling also allows microservices to be developed using different programming languages.

Despite the many similarities between services and microservices [2], there are some fundamental differences, mainly related to their execution. Languages like WS-BPEL[1] describe the workflow of service compositions. By contrast, a μApp workflow is not formally specified. The μApp communication must be monitored to infer the underlying workflow.

A downside of using microservices is their management complexity. A microservice may have different behaviours while it executes, which is reflected in both the volatility of resource usage and changes in its workflow. Hence, initial μApp deployment choices, like the placement of microservices, may be sub-par later.

*Correspondence: arsj2@cin.ufpe.br
[1]Center of Informatics, UFPE, Recife, Brazil
Full list of author information is available at the end of the article

The management of μApps is performed by engineers aided by tools that provide timely data about applications (e.g., resources usage) and microservice lifecycle management (e.g., replicating microservices on demand). However, these management tools are incapable of providing crucial runtime data like the amount of data exchanged between two microservices or the resource usage history of a given microservice. As a consequence, existing tools are unable to perform management operations, like the replacement of microservices, based on actual execution data.

At runtime, microservices that compose an application can interact and exchange a significant amount of data, creating communication *affinities* [3]. We define affinity as a relation between two microservices given by the number and size of messages exchanged over time. These inter-service affinities can have a substantial impact on the performance of the μApp, depending on the placement of the microservices, e.g., microservices with high affinity placed on different hosts will have worse performance due to higher communication latency. Making this more complex is the fact that affinities change at runtime.

In addition to affinity, developers must also account for microservice's resource usage history to optimize μApp placement. For example, microservices with a history of high resource usage should not be co-located on the same host. Also, service upgrades and different workflows along the μApp execution change the resource consumption and affinities.

Existing management tools, like Kubernetes[2] and Docker Swarm[3], allow DevOps engineers to control μApps by setting resources thresholds on the microservices. The management tools use this information to decide when to scale in/out each microservice and where to place microservice replicas. At runtime, the management tools continuously compare the instantaneous resource usage of the microservices with their resource threshold. When the resource usage reaches the limit, the management tool starts the scaling action. During scale out, existing management tools select the hosts where to place the microservices replicas based on the set thresholds instead of their history of resource usage. As our experiments will show, in most cases a resource threshold is unrealistic and leads the μApp to waste cluster resources or to lose performance due to resource contention.

Management tools should be aware of runtime and historical data for μApps to avoid misplacement of microservices. Although these tools can use instantaneous data to perform scale up/down activities, this is not enough in cases where a re-arrangement of microservices is necessary. We therefore propose a new tool that uses history and runtime data to better manage microservices.

The focus of our work is on improving the management of μApps through the use of runtime data as the basis for automatic management tasks, such as placement optimization.

Next, we overview three challenges to realizing adaptation of μApps:

- **Challenge 1: Unified monitoring of μApps.** Existing management tools can collect and expose many resource metrics from executing μApps. However, each μApp uses its own monitoring stack. The diversity of monitoring options creates a semantic challenge, requiring a single unified data model.
- **Challenge 2: Finding a high-performing placement.** Microservices are usually placed using static information like the available host resources. However, this strategy risks lowering μApp performance by putting high-affinity microservices on different hosts or by co-locating microservices with high resource usage. Hence, it is necessary to find the best performing configuration that maps microservices to servers. This need leads to two sub-problems: (1) a large space of configurations: with n servers and m microservices there are m^n possible configurations; and (2) the performance of a μApp configuration changes dynamically.
- **Challenge 3: Migrating microservices.** Existing microservices management tools do not present alternatives to performing live migration of microservices between hosts. This migration is necessary to provide seamless runtime adaptation.

Our proposal is an adaptation mechanism, named REMaP (RuntimE Microservices Placement), that addresses the above three challenges and automatically changes the placement of microservices using their affinities and history of resource usage to optimize the μApp configuration. In our prior work [4], we considered *Challenge 1* by defining a model to support the evolution of μApps. In this work, we refine the ideas presented in [4] to address *Challenge 2* and *Challenge 3*. The core of this work is about addressing *Challenge 2*. We present our view on runtime adaptation of μApps and the challenges with optimally arranging microservices. Finally, in this work, we partially address *Challenge 3*.

REMaP approach overview. Our solution uses a MAPE-K based [5] adaptation manager to alter the placement of μApps at runtime autonomously. REMaP uses Models@run.time concepts and abstracts the diversity of monitoring stacks and management tools. In this way, our solution provides a unified view of the cluster and the μApps running under the adaptation manager.

REMaP groups and places microservices with high affinity on the same physical server. This strategy is in contrast with the existing static approaches that rely on information provided by engineers before μApp deployment. For instance, Kubernetes arranges microservices in a cluster based on the resources limits (max and min) and tags set by engineers. Kubernetes does not consider information about the relationship between microservices to improve the deployment of the μApp by reducing the hosts to be used and the communication latency between microservices. Hence, our adaptation manager can provision resources based on actual microservice resource utilization, avoiding resource contention/waste during μApp execution. Moreover, the co-location of microservices decreases the impact of network latency on the μApp workflow, which improves overall application performance. At the end of the adaptation, the μApp has an optimized configuration that reduces the number of hosts needed to execute the μApp and improves its performance as compared to an unoptimized deployment.

Prior work focusing on adaptation of μApps proposed to change the μApp deployment at runtime to provide runtime scaling [6] or to update a running deployment to a new version [7]. These approaches do not improve the placement of the constituent microservices nor consider the resource usage history when formulating an adaptation. These tools use instantaneous metrics gathered from the μApp, which do not reflect their real resources needs over a longer time period.

We evaluated REMaP in two scenarios, and we compared the results of an optimization algorithm based on SMT (Satisfiability Modulo Theory) [8] with a simple variation of a First-Fit algorithm [9]. We made this comparison to evaluate the feasibility of finding an optimal placement instead of an approximation, given the complexity of the problem. In the first scenario, we used the proposed mechanism to compute the adaptation of synthetic application graphs, having a different number of microservices and affinities. In this scenario, we realized adaptations that saved approximately 80% of the hosts used initially by the μApp. This evaluation shows that our approach produces better results on μApp with dense topologies. Moreover, when using SMT, the adaptation mechanism was unable to work on μApps larger than 20 microservices. Hence, although our heuristic cannot guarantee an optimal result, it can compute placements for μApps of any size.

In the second scenario, we used REMaP to adapt a reference μApp[4] running on Azure. In this scenario, we achieved a performance improvement up to 62% and saved up to 40% of hosts used in the initial deployment. Moreover, we found that a poor placement that uses the same number of hosts can decrease the overall performance of the μApp by 150%, indicating that the placement requires special care by engineers; care that our approach automates.

The rest of this paper is organized as follows. Section 2 introduces the concepts necessary to understand the rest of the paper. Section 3 discusses the challenges of general-purpose runtime adaptation of microservices. In Section 4 we discuss the specific challenges to adapt μApps by re-configuring the microservices placement at runtime. Sections 5 and 6 present the design and implementation of our solution, respectively. Section 7 presents the evaluation of the proposed approach. We contrast with related work in Section 8. Finally, Section 9 concludes and notes some avenues for future research.

2 Background

2.1 Microservices

The *microservice pattern* is an architectural style of *service-oriented computing* whose cornerstone is a high degree of decoupling [10]. A microservice is an autonomic, and loosely-coupled software having a specific functionality in a bounded context. As shown in Fig. 1, microservices (hexagons) belonging to a microservice-based application (μApp) communicate using lightweight communication protocols like HTTP. These microservices are usually deployed in containers, a lightweight alternative to traditional virtual machines [1]. The use of containers facilitates the elasticity of the μApp: only some parts, and not the entire application, need to expand and contract in response to changing workload. For instance, suppose that the microservices related to orders have a higher load than those associated with sign-ups during a Christmas sale. In this case, just the first set of elements needs to scale up to avoid bottlenecks, while the others can contract and release resources for use by other components. This behaviour is not available in monolithic applications.

Figure 1 shows the design of Sock-shop, a microservice-based reference application [11]. Sock-shop adopts good practices in μApps, like data storage and communication middleware (e.g., RabbitMQ) wrapped by microservices, and containers, which provide flexibility to the μApp. Sock-shop is composed of a hub of six microservices (front-end, orders, users, carts, catalogue, and shipping), their respective data storage, and auxiliary microservices.

The decoupling provided by microservices facilitates the maintainability of μApps by reducing the complexity of management tasks like upgrade and replication. The use of microservices typically increases the number of components that make up an application and complicates application management. To make matters worse, an essential feature of μApps is their ability to scale in/out by removing/creating replicas as necessary. This fact causes microservice instances to have a short lifetime and adds further dynamism and complexity to the deployment.

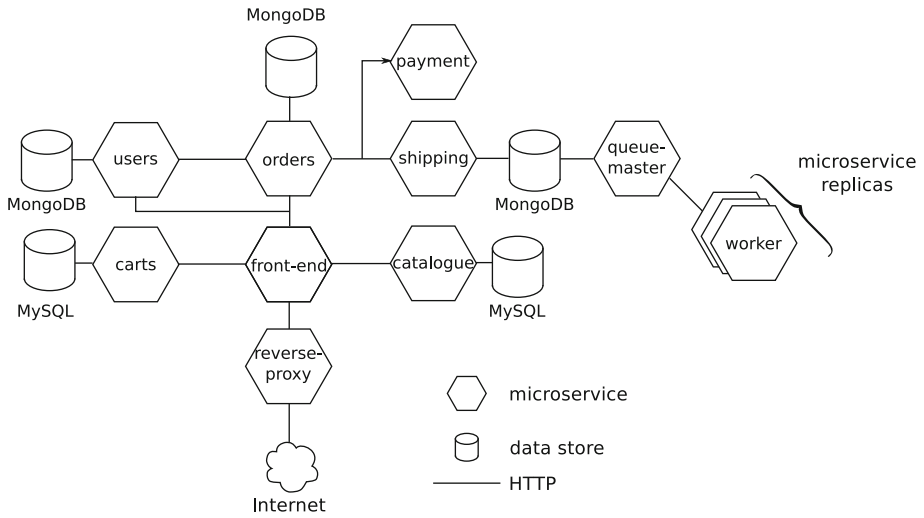

Fig. 1 Architecture of Sock-Shop a reference μApp

Despite the many similarities between services and microservices [2], there are some fundamental differences. Developers use languages like WS-BPEL [12] to define the workflow of a service-based application and use an engine to execute the workflow. By contrast, the flow of messages exchanged by microservices that compose a μApp defines the workflow. Hence, it is necessary to change at least one microservice to modify μApp's workflow, e.g., by replacing it. Consequently, more caution is needed to evolve a μApp.

μApp engineers use management tools, like Kubernetes, to control a μApp automatically. Unlike autonomic applications, whose management requires either minimal or no human intervention, microservice management tools need an engineer to guide the management tasks. These tools can automatically update and upgrade a μApp, e.g., scale in/out or roll out/back a microservice, by following the engineer's instructions. In general, engineers set the maximum and the minimum number of replicas that a μApp should have, and a resource threshold that triggers the scaling process. Moreover, the management tool automatically deploys a microservice onto the cluster by considering attributes set by the engineers at deployment time. However, this placement is not optimal and can jeopardize the execution of a μApp in some circumstances.

2.2 Autonomic computing

Autonomic computing refers to self-managing computing systems [13]. Self-management means that the system can control itself, and human intervention is not necessary for activities such as optimization, healing, protection, and configuration.

A self-managed system must monitor itself and the environment, analyze signals produced by the monitoring, and applies actions in response, perhaps by modifying itself. These steps repeat indefinitely as a control loop. IBM systematized this loop by proposing a reference model for automatic control loops called MAPE-K [5] (Fig. 2).

In MAPE-K, the ***m****onitor* collects data from the managed system and dispatches them to the ***a****nalyzer* that reasons over them or infers new information. Next, the analyzer forwards the result of the analysis to the ***p****lanner* that computes an adaptation or management plan. The ***e****xecutor* receives the plan and applies it to the managed system. Lastly, components of MAPE-K share a ***k****nowledge base* that maintain rules, properties, models and other kinds of data used to steer how to provide autonomy to the underneath system.

Szvetits and Zdun [14] and Krupitzer et al. [15] surveyed several strategies for adapting complex, heterogeneous, and dynamic systems that do not apply the adaption plan directly to the managed system. In this case, the adaptation plans are applied to models maintained at runtime (model@run.time) [16]. model@run.time abstracts the system, simplifies the adaptation process and helps to handle the diversity of underlying technologies. This model has a causal connection with the managed system such that changes to the application are reflected in the model, and vice versa [17]. The causal connection is carried out by a meta-object protocol (MOP) that maps the elements of the model into their representations in the application.

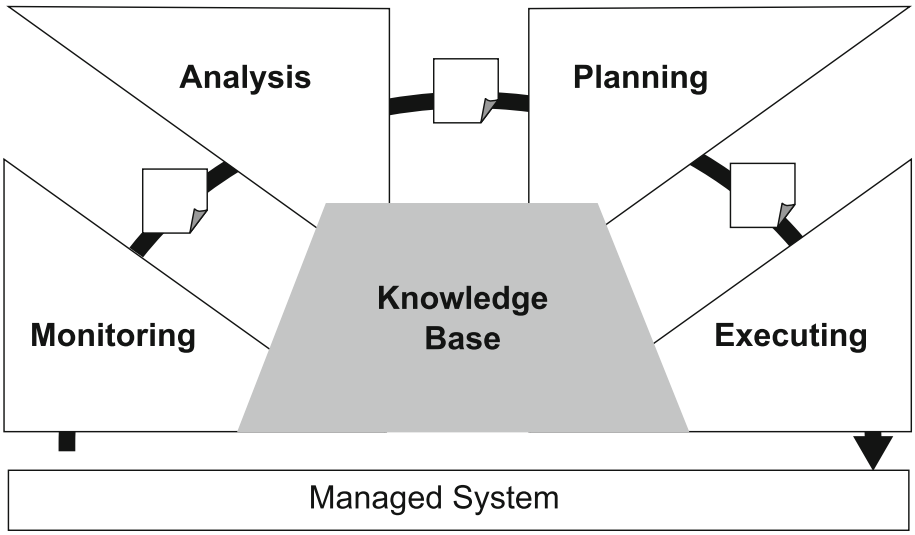

Fig. 2 MAPE-K reference model

3 Challenges of runtime adaptation of microservices

Runtime adaptation is inherent to μApps, which can evolve during execution. The decoupling promoted by the microservice architectural style allows updates and upgrades at runtime without pausing or terminating the application. Scale in/out operations usually update μApps, while roll out/back operations upgrade the microservice versions.

In our context, an adaptation consists of (1) replacing one microservice instance with another, usually on a different host, or (2) creating new microservice replicas. Management tools execute such adaptations. However, engineers use these tools and manually guide the adaptation. For example, a tool can automatically trigger auto-scaling, but an engineer must fix both the maximum number of replicas and the resource usage that triggers the scaling. In an autonomic approach, the adaptation mechanism would automatically decide on these parameters.

The high decoupling of μApps facilitates the adaptation, although additional mechanisms are necessary to change a μApp safely. For example, while a microservice is being replaced, the new instance can become unavailable for a short time. During this time, other microservices may attempt to establish communication and will fail as the new instance is not ready.

In addition to failures during adaptation, failures can come up during normal μApp execution. Today's developers use design patterns like *circuit breaker*[5] and *retry with exponential back off* [18] to minimize the negative impact of failures on a μApp.

At runtime, another source of flaws is the stateful nature of microservices. When a new microservice must replace an old one, the management tool cannot automatically synchronize their data. In general, the critical steps to replace a microservice are (i) to instantiate the new instance and (ii) remove the old one. Therefore, when a stateful microservice is updated, a mechanism is necessary to deal with state synchronization.

Finally, a μApp is potentially multi-lingual and multi-technology, which complicates monitoring. Although monitoring tools can collect information from a μApp execution (e.g., instantaneous resource usage), behavioural aspects (e.g., resource usage history and μApp workflow) are not collected by existing tools.

3.1 Monitoring μApps

It is necessary to track the behaviour of a μApp to control it. Having this information, engineers or management tools can understand how the application works and then compute plans to improve its behaviour. Furthermore, it is possible to foresee future behaviours and apply adaptations to optimize resources to accommodate them.

The behaviour of a black-box μApp can be observed in at least the following three ways:

Resource usage The resource usage is the amount of resource used by a microservice during the μApp execution, e.g., CPU, memory, disk and network bandwidth.

Application logs μApp engineers determine events of the application to signal errors, warnings, and informational messages during its execution. Informational messages do not log warns or errors,

e.g., "[INFO] loading library xxx". Dedicated tools maintain the events in the sequence they occur, and engineers use the application logs for tracking the μApp execution.

Message information The messages exchanged by microservices, including message's source and destination, payload size, some metrics like response time, and contextual data such as HTTP headers and response codes.

However, instantaneous data is not enough to track the behaviour of a μApp. Maintaining historical data along the execution is essential. Nowadays, engineers use third-party tools to record and track the historical data of μApps, since management tools only expose instantaneous microservice information. Some existing tools are now being used for this purpose. cAdvisor[6] gathers cluster (hosts) and microservices (wrapped into containers) metrics natively; Prometheus[7] stores data collected by cAdvisor or self-stored by microservices; and Influxdb[8] stores monitored data.

Although management tools expose the microservices' execution logs, these tools cannot aggregate and use them at runtime. Consequently, aggregators are needed to organize logs, ensure their temporal order, and store them to maintain their history. Popular log aggregators include Fluentd[9] and Logstash[10]. It is also necessary to use data stores like Elasticsearch[11] and Influxdb to maintain a history of the μApp execution. Furthermore, cloud providers usually provide their private solutions such as Amazon CloudWatch[12].

None of the current management tools are aware of messages exchanged between microservices. This fact is a major drawback since messages are critical to understanding how a μApp actually works. There are few initiatives to gather and store μApps messages, like Zipkin[13] and Jaeger[14].

The broad diversity of tools to collect data and track μApp behaviour is made worse by the fact that, in general, existing tools do not follow any standard. This lack creates a semantic gap in the information they provide. For example, data metrics collected by Influxdb and Prometheus have different formats. Hence, it is difficult to analyze the behaviour of varying μApps running on the same cluster, since each one can use a different monitoring stack.

Currently, the collected data is analyzed manually by engineers, who have to retrieve and parse the gathered data, send it to visualization tools, like Kibana[15], to take some action based on what they observe. These steps make the management of μApps complex and error-prone [19].

Also, a μApp may include microservices implemented in different languages, which means that various tools are necessary to monitor the same information. Furthermore, not all languages include bindings for a specific tool, e.g., Zipkin, which means that different tools may monitor microservices belonging to the same μApp. The heterogeneity of monitoring tools is a challenger as it is necessary to deal with different semantics, data structures, and technologies.

3.2 Model at runtime

According to Blair et al. [16], the use of model@run.time simplifies the inspection and adaptation of complex and heterogeneous systems. Hence, considering the heterogeneity on monitoring μApps, model@run.time is an interesting concept to be applied in the management of these applications.

Models are used to simplify the representation of complex systems. The model is a high-level view that only exposes the relevant structure and behaviour of the system according to the model's usage intent.

A model can also be used at runtime. In this case, the model can provide a causal connection with the underlying application/system. This causal connection allows changes to be applied to the model and reflected in the application at runtime, and vice-versa. This feature facilitates the adaptation process of μApps since it is not necessary to deal with interfaces of the management tools. Hence, the model also acts as a *proxy*, abstracting and enhancing the access to the management tools' interface. Due to these features, several projects, listed in [14], use models at runtime as the primary element of the runtime adaptation of complex systems.

Also, the unified view of the system in a single artifact facilitates the maintenance of its evolution [4]. μApps are dynamic distributed systems, and their heterogeneity makes challenging to track their evolution through time. The use of model@run.time helps by (i) unifying the data in a well-defined structure, and (ii) evolving it along with the abstracted μApp. Hence, we can build up an evolution trace of the application by keeping snapshots of the model as changes are applied.

The history trace allows retrospective analysis to enhance the current state of the μApp or foresee its future state. These analyses are essential in bringing autonomy to manage μApps since autonomic mechanisms can inspect current and past states and decide what to do without human intervention.

Another advantage of using models at runtime is to plan out elaborate actions that can be applied to μApps. The model organizes data in such a way that planners can readily traverse the model, combining and deriving new information, without facing the semantic gap that appears when dealing with raw data produced by monitoring tools.

Finally, model@run.time allows safe changes to be applied to μApps. Since the model has all the information about its underlying application, it is possible to check the

changes applied to the model before consolidating them. For example, suppose that the adaption needs to move a microservice to a new host. In this case, it is necessary to check in the model if the target host has sufficient resources to accommodate the microservice (e.g., considering its resource usage history). Without such a model, this check cannot be quickly performed.

4 Optimal placement of microservices

In Section 3, we emphasized that adaptation of μApps means to change microservices to different versions by rolling them out/back, or by creating or deleting microservices instances through scaling in/out. In both cases, the adaptation relies on placing microservices into different hosts. However, to define the best placement is not an easy task.

The deployment of μApps in a cluster must take into account the required resources defined by engineers and resources available in the hosts. To configure the deployment, μApp engineers might set the minimum and maximum amount of resources the microservice needs, e.g., CPU and memory. However, there are no rules to determine these values accurately. Usually, engineers set these values based either on previous executions of the microservice or their own experience, which is subjective. Hence, it is difficult to establish what resources a microservice may need at runtime to work well.

Although management tools, like Kubernetes, allow users to set upper limits on resource usage, there is no guarantee that engineers will set these limits. And, even if engineers do set them, there are no guarantees that the chosen values are the best for all workloads for an μApp execution. Besides, assuming that the restrictions are properly configured, management tools do not impose them during the execution of a microservice, but only use these This issue is more evident in languages like Java (prior version 8)[16] and Python[17], in which the runtime cannot properly interpret these limits and can crash the μApp if the limits are reached. This unreliable approach, therefore, leads management tools to make poor deployments, which may either degrade the application performance or crash the entire μApp.

Another consequence of only setting the minimum quantity of resources is the placement of many microservices together into a single host. Co-located microservices may in aggregate demand more resources than are available on the host. This demand leads the μApp into contention and hurts performance. Meanwhile, microservices configured with minimum resource requirements drive management tools to deploy a μApp across many hosts, which may waste resources. Also, placing microservices across several hosts jeopardizes their performance due to the network latency imposed on their communication.

It is worth observing that several μApps share a single cluster and each one has different features and requirements. But, management tools are unaware of the runtime needs of microservices. At deployment time, the cluster provider tries to balance the hosts' resource usage without jeopardizing the μApp performance. However, the lack of standardization by engineers to set the microservices resource requirement complicates their placement.

Existing management tools implement several common placement strategies. These are used by the cluster provider to deal with the average demand of μApps. Next, we overview these common placement strategies:

Spread strategy. The management tool places a minimum number of microservices per host in the cluster. This strategy tries to avoid resource contention since few concurrent microservices will dispute for the same resources. However, it can lower μApp performance by adding latency to request/response messages as microservices may be deployed on different hosts. Moreover, this strategy can waste resources since some microservices may need fewer resources than what their host provides. Docker Swarm and Kubernetes adopt the spread strategy.

Bin-pack strategy. The management tool uses the minimum number of hosts to deploy a μApp. This strategy avoids the cluster resource waste. However, putting many microservices together causes contention for the same resources, dropping μApp performance drastically. This strategy is available in Docker Swarm.

Labeled strategy. In addition to the resource requirements, microservices can be annotated with attributes used to guide host selection. For example, a machine learning μApp can require being deployed on hosts with GPUs for performance reasons. Then, at deployment time, the management tool selects a host that matches the microservice labelled requirements. This strategy is usually used to customize the default management tool strategy. For example, the default strategy of Docker Swarm and Kubernetes can be customized with labels as constraints on the placement of some microservices.

Random strategy. The management tool selects a host to deploy a microservice randomly. This strategy is available in Docker Swarm.

Whatever the strategy, management tools do not use historical data to drive or enhance the placement of microservices. Existing tools select the hosts considering the instantaneous resource usage to place the microservice, and rarely try to find an optimal setting.

4.1 The case for runtime microservices placement

The facility of scaling and upgrading microservices can threaten the μApp, imposing resource contentions, network latency and wasting of cluster resources.

The choreography of microservices defines the μApp workflow, which means that it is necessary to upgrade one or more microservices to evolve the μApp behaviour. During an upgrade, the resource requirements might change, as well as the relationship between microservices might result in a new workflow.

Similarly, a new version of a microservice can come up using more or fewer resources than the old one. Management tools unaware of runtime resource usage or considering only the minimum and maximum configuration can make a poor placement choice for the new microservice.

For example, if the new microservice has a higher communication demand than the prior version, the management tool unaware of μApp communication requirements can place microservices in different places. The high communication between the two services over the network can hurt the overall performance.

Finally, the new workflow that arises after an upgrade can change the relationship between microservices. For example, microservices previously related may no longer be created, and vice-versa. The new relationship among them requires a μApp reconfiguration (new placement) to avoid the performance degradation due to resource contention and network latency. The analysis of historical data can improve the placement of μApps by providing reliable information about different features of the application, such as the actual resource usage and the messages exchanged.

4.2 Steering microservices placement

Analyzing the messages exchanged by microservices helps to understand their relationships. Related microservices usually exchange a high number of messages and/or a high amount of data. The combination of the number of messages and the amount of data gives us an idea of *affinity* between microservices. High-affinity microservices placed in different hosts can impose performance overhead on the μApp due to network latency. Therefore, related microservices should be placed together.

However, only putting high-affinity microservices together is not enough to optimize the μApp placement. It is necessary to consider the runtime resources usage of microservices to achieve optimal placement. Existing management tools do not take into account the historical resource usage to place microservices. Instead of considering static values set by engineers to select a better host, it is also necessary to analyze the resource usage history to choose the host that better fits the actual microservice resource requirement.

In Section 3, we discussed the use of Models@run.time to keep runtime data and help the runtime analysis of μApps. The history of resource usage allows the selection of hosts based on the actual needs of μApps, avoiding (or at least reducing) the concurrency problems mentioned before.

4.3 Placement optimization

Optimizing the placement of microservices in a cluster is a variation of the bin-packing problem [20]. In the bin-packing problem, objects of different sizes must be packed into a finite number of bins of volume V in a way that minimizes the number of bins needed. This approach is a combinatorial NP-Hard problem. In our context, the objects to be packed are the *microservices* and the bins are *hosts* of the cluster.

Unlike the classical bin-packing problem, the placement of microservices in a cluster cannot consider only one dimension, but the microservices affinities and their resources usage, e.g., CPU, memory, disk, and so on. Therefore, our problem is a multi-dimensional variation of bin-packing [21] that is exponentially harder to solve. The formal statement of the microservice placement problem is stated as follows:

Given a set of hosts $H_1, H_2, \cdots, H_m$ and a set of μApps $P_1, P_2, \cdots$, where P_i is a set of n microservices $m_{P_i,1}, m_{P_i,2}, \cdots m_{P_i,n}$ linked by the affinity function $A : m_{P_i} \rightarrow m_{P_i}$. Find an integer number of hosts H and a H-partition $H_1 \cup \cdots \cup H_B$ of the set $\{1, \cdots, n\}$ such that the $\bigcup$ of the multi-attributes microservices $m_{P_i,j}$ fits on H_k for all $k = 1, \cdots, B$, $i = 1, \cdots, P$, and $j = 1, \cdots, n_{P_i}$. A solution is optimal if it has minimal B and maximum affinity score for all H_k.

The multi-dimensional bin-packing problem adopted in the cluster domain is well understood [22]. However, the complexity of computing an optimal result in a reasonable time for big instances prevents its use at runtime.

There are several approaches surveyed in [20, 22] to compute this optimization in an offline way. At runtime, the best strategies are approximations calculated through heuristics and evolutionary algorithms to achieve a *quasi*-optimal solution.

4.4 Moving microservices at runtime

To optimally place microservices in a cluster, it is first necessary to know how to move them at runtime. Not all microservices can be moved into a new placement, e.g., stateful and labelled microservices.

Further, a stateful microservice is a kind of data source used by μApps. Usually, μApps outsource their data to dedicated storage services provided by cluster infrastructure, which are out of the scope of management tools. If the μApp has a data store (e.g., SGBDs and NoSQL databases) and it is moved to a new placement,

management tools are unable to seamlessly migrate their data to the new destination, which leads to inconsistencies in the state of the μApp.

Existing management tools have simple primitives used to move a microservice across different hosts. However, due to a limitation of existing operating systems and frameworks, it is not possible to live-migrate processes (microservices) between machines. As a workaround, the movement of a microservice can be emulated by a three-step sequence:

1. Instantiate the microservice replica at the new location,
2. Wait for the microservice to become ready, i.e., load its libraries and be able to handle requests, and
3. Remove the microservice replica from the previous location.

Management tools usually have built-in primitives to help to implement these steps. The primitives try to avoid faults during the migration. To achieve a safe migration, μApps should be implemented using patterns such as *circuit breaker* and *retry with exponential back-off* (see Section 3).

5 Design

To bring autonomy to μApp management, we propose a MAPE-K [5] based adaptation manager, named REMaP (RuntimE Microservices Placement), to autonomously adapt μApps at runtime. Runtime adaptation requires three main steps: to monitor the system under management, to make a decision based on monitored data, and to execute an adaptation plan taking into account the decision. Figure 3 overviews our solution.

5.1 Design overview

Upgrade events of the μApp trigger the adaptation process. The adaptation repeats indefinitely at regular time intervals. When the code of one or more microservices changes, i.e., developers push new code into the repository, the *Adaptation Manager* listens for this event and starts the adaptation. The push command should be labelled with the time interval in which REMaP waits between the adaptations.

The first step of the adaptation is to collect data about the μApp. The *Monitor* inspects the μApp through the *Monitoring Adapters*. The adapters abstract away different monitoring technologies to gather useful historical data, such as resources usage and microservices interactions (μApp workflow). The *Monitor* gathers data according to the time interval set in the adaptation bootstrap (e.g., if the time interval is 10 minutes, the *Monitor* gathers all data generated in the last 10 min.) REMaP takes collected data and populates the application *Model*. The *Model* organizes the data for the μApp, building up its topology, filling up each representation of the microservices in the model with their historical data, such as the average of CPU and Memory consumption in the time interval, and linking the microservices according to the messages exchanged.

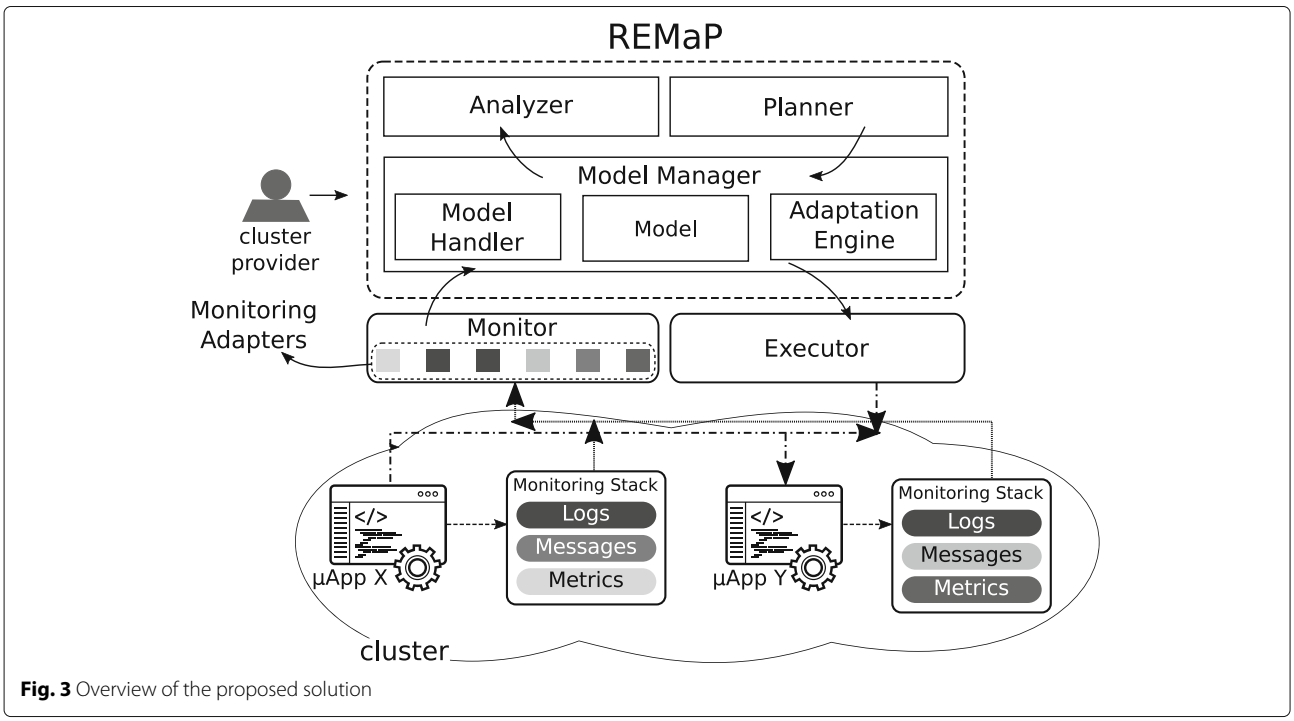

Fig. 3 Overview of the proposed solution

Next step is the model analysis. The *Analyzer* inspects the model looking for interactions between microservices to compute their affinities. The *Analyzer* stores the affinities in the model, in such way that the *Planner* can access them.

Once the affinities are available, the *Planner* can compute the adaptation plan. The *Planner* uses the affinities and resource usage stored in the model to calculate a new placement plan (adaptation plan) for the microservices.

In this scenario, the adaptation means the optimization of the microservice placement. The optimization relies on two dimensions: microservices' affinities and resource usage history. Considering these dimensions, the *Planner* computes a deployment plan to reduce the resource usage and communication latency while minimizing the impact on the application performance.

Traditional approaches to the Bin-Packing problem try to minimize the number of bins used considering the number of items available and their respective values (see Section 4). In contrast, the optimization of microservice placement also includes the concept of affinity, so that the value of an item (microservice) varies according to the other items in the bin where it is assigned. This feature increases the complexity of the problem. Hence, in addition to the minimization of resource usage, we also aim to maximize the affinity score of the selected hosts, i.e., placing the maximum number of highly-related microservices together.

Finally, the *Adaptation Manager* applies the adaptation on the *Model* and checks the consistency of changes before consolidating them in the running μApp. The *Adaptation Manager* forwards to the *Executor* the changes that do not violate the model.

The *Executor* consolidates the changes in the μApp by translating the actions defined in the adaptation plan and applied to the model into invocations to the management tool API.

The rest of this section details all of the components introduced above.

5.2 Model

The *Model* shown in Fig. 4 abstracts and allows the inspection and analysis of μApps at runtime. As mentioned in Section 3.2, we use Models@run.time concepts to make viable the use of a unique artifact and reduce the semantic gap between technologies used to monitor μApps.

The *Model* abstracts essential elements of the μApps in a cluster. This model is inspired by the evolution model proposed in [4] and includes the concept of affinity. In this work, the model acts like a *facade* to simplify and unify the interfaces provided by different monitoring tools.

Class *Microservice* models a microservice and includes the name and an indication whether the microservice is stateful or not. Class *MicroserviceInstance* is a specialization of class *Microservice* and represents a microservice instance, i.e., a μApp includes different kinds of microservices and each type of microservice can have multiple replicas (microservice instance). A microservice instance has the total number of messages and data exchanged by a microservice replica.

Class *Message* models the communication between μApps, and represents the edges in the μApp graph. Every message has a unique id, response time, timestamp, and size. The set of messages describes the workflow of the μApp.

Class *Affinity* models the communication between two different microservices (not their replicas) considering the number of messages and amount of data exchanged. The affinity has a degree that represents the strength of the relationship between two microservices.

Class *Cluster* abstracts the management tool used and maintains the hosts available in the cluster. In turn, each class *Host* has a set of microservice instances.

Finally, hosts and microservice instances have *Resources* attributes that maintain the information of the usage history, e.g., CPU and memory mean usages (history) of hosts and microservices, and resources limits. Figure 5 shows how a μApp is represented at runtime by our model, and it highlights the μApp architecture (application view) and the μApp deployment (cluster view).

The cluster view models how the μApp is deployed across several hosts and the use of resources by hosts and microservices. Moreover, this view shows the communication topology and messages exchanged by the microservices. The cluster view is volatile as microservices replicas frequently come up and go at runtime.

The application view models the architecture of μApps running in a cluster and highlights the microservices that make up the application. This view also shows affinity links between microservices; this view is more stable than the cluster view since microservice upgrades are less often than microservices scaling.

The separation of application and cluster concerns creates a more expressive model. It allows the analysis and adaptation actions to be performed individually on the μApp or cluster without needing to inspect the whole model. For example, the *Adaptation Manager* can use a fresh configuration to compute the adaptation plan (cluster view), while the *Executor* uses cluster information to guarantee that only safe changes will be applied on the μApp (*application view*).

5.3 Monitoring

The *Monitoring* component, shown in Fig. 3, is designed to gather heterogeneous data from the cluster and unify it in a technology-agnostic API to populate the *Model*. The cluster data is collected discretely to avoid flooding

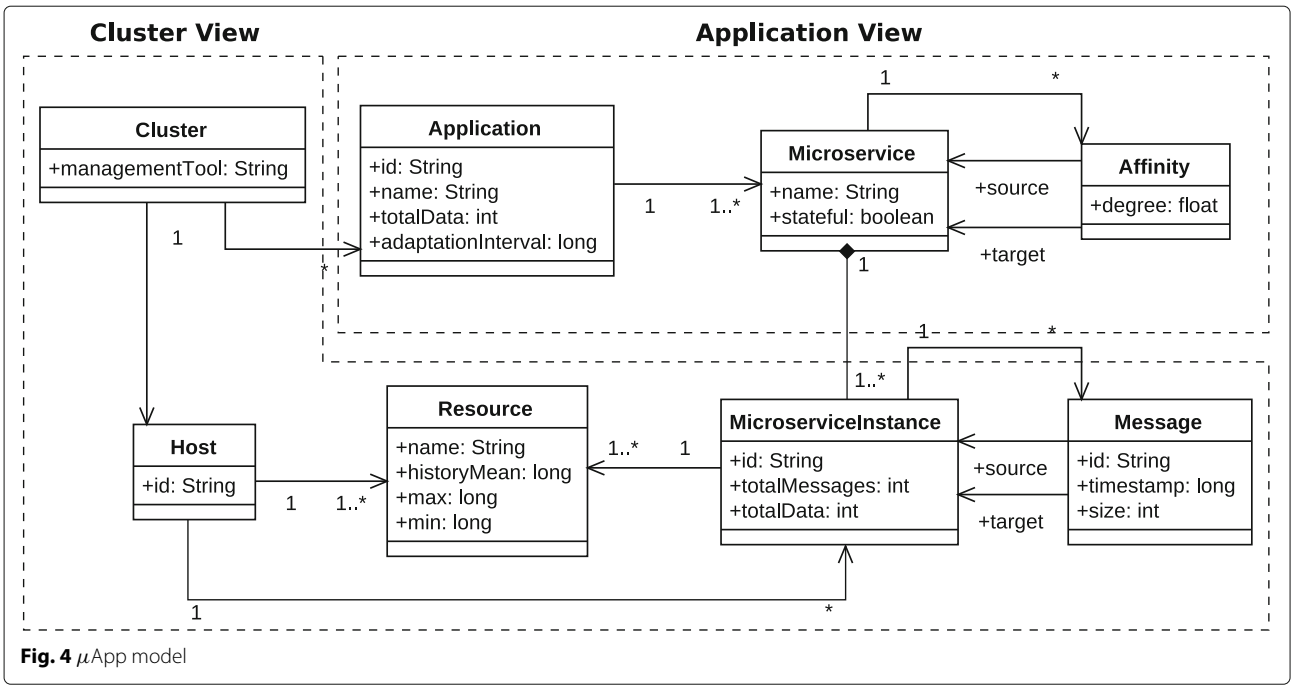

Fig. 4 μApp model

the *Adaptation Manager* and consequently triggering too many adaptations.

Component *Monitoring* provides a standard way to retrieve data independent of monitoring technologies used in the cluster. This component is passive (only returns data in response to a request) and aggregates all data necessary to compute the adaptation plan. In practice, the *Model Manager* used information monitored to update the runtime model.

Every μApp has a monitoring stack to collect data. Usually, the monitoring stack collects three different kinds of data: resources usage, execution logs, and exchanged messages. The monitor maintains a global view of the environment by gathering information from the management tool being used (e.g., Kubernetes or Docker Swarm), host and microservice resource usages, and events generated by DevOps operations, such as updates to the code repository and microservices deployment. The numeric data, usually related to resources usage, are aggregated as the average of instantaneous values measured from the μApps components. For instance, this average can represent the history of CPU usage of a microservice in a given time interval.

Monitoring tools collect data and store them in different data stores continuously. The monitor abstracts these data stores by using clients responsible for retrieving and transforming these data into agnostic structures used for populating the model.

Finally, the monitoring provides data according to various aspects such as resources metrics, messages, logs, and events. For each of these, the monitoring component allows sampling of the data over a given period.

5.4 Analyzer

The *Analyzer* component, shown in Fig. 3, is designed to process the model at runtime by looking for affinities between microservices. We define *affinity* between two microservices using the number of messages and the amount of data exchanged between them. We use a ratio (weight) in the affinity calculation to steer the *Analyzer* execution. The data exchanged by the microservices are not equally distributed among all exchanged messages. It may exist μApp workflows with few messages and a large amount of data and vice-versa. Hence, a high ratio value (ratio $>$ 0.5) leads the *Analyzer* to value the number of messages over the amount of data. A small value (ratio $<$ 0.5) does the opposite, and a ratio of 0.5 balance the two attributes equally.

The *Analyzer* component detects unexpected behaviours not perceived by application engineers. Different analyzers can be implemented and plugged into the adaptation mechanism, depending on the analysis needed. In this work, we design an *Affinity* analyzer that detects affinities between microservices. As mentioned in Section 4, the affinity is used to optimize the μApp placement.

The analyzer checks the messages exchanged by the μApp (stored in the model) and calculates the affinity between two microservices. This calculation uses the number and size of messages to determine the bi-directional affinity. We define the affinity between two

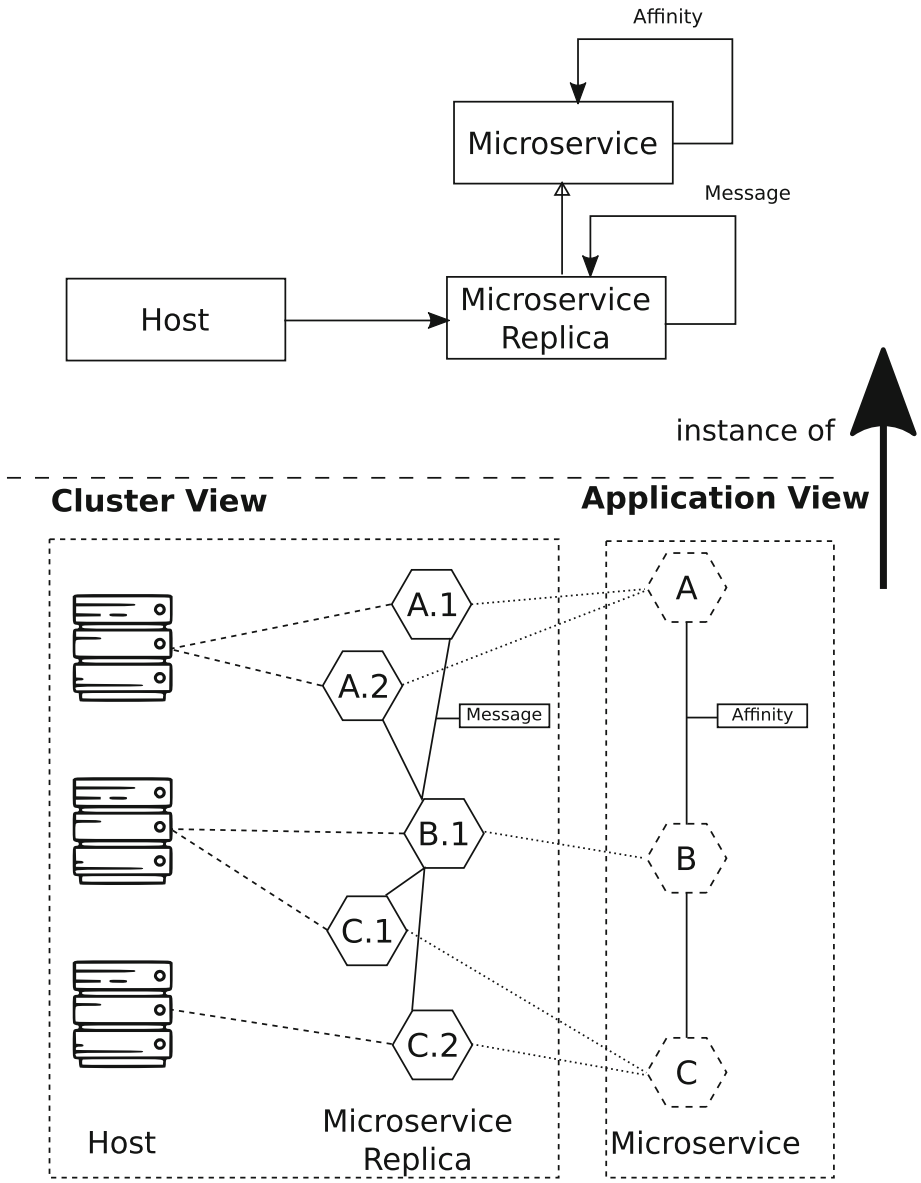

Fig. 5 Instantiation of the model

types of microservices a and b as $A_{a,b}$ and calculate it as follows:

$$A_{a,b} = \frac{m_{a,b}}{m} \times w + \frac{d_{a,b}}{d} \times (1 - w), \quad (1)$$

where,

- m is the total of messages exchanged by all microservices,
- $m_{a,b}$ is the number of messages exchanged between microservices a and b,
- d is the total amount of data exchanged by all microservices,
- $d_{a,b}$ is the amount of data exchanged between microservices a and b,
- w is the weight, such that $\{w \in \mathbb{R} \mid 0 \leq w \leq 1\}$, used to define which variable is the most important to compute the affinity, i.e., number of messages or amount of data exchanged.

The analyzer calculates the affinities between all microservice instances and dispatches them to the planner via the model manager. Besides, the analyzer aggregates affinities of microservices instances, taking into account their types and populates the model with the computed affinities.

Finally, in addition to synchronous communication through REST APIs, the microservice architecture commonly uses asynchronous communication through Pub-Sub protocols. REMaP can compute an optimization for μApps using async communication since, like data stores, the messaging middleware (e.g., RabbitMQ) is wrapped into a container. In this case, the analyzer can identify which microservices have a high communication rate and may co-locate them with the middleware. However, if the μApp outsources the messaging middleware, REMaP cannot correctly calculate microservices affinities, and consequently, no placement optimization may be applied.

5.5 Planner

The *Planner* component decides how to apply the adaptation to the μApp. Similarly to the analyzer, different planners can be implemented and plugged into the adaptation mechanism, depending on the adaptation strategy.

We propose two *Planners* to compute the placement of microservices during an adaptation: the *Heuristic-based Affinity Planner* (HBA) and the *Optimal Affinity Planner* (OA). Both planners compute a new placement for μApps by reducing this problem to a multi-objective bin-packing problem. As this problem is NP-Hard, we know that for large μApps an optimal approach is infeasible. Hence, we implement the heuristic version (HBA) to achieve approximate solutions for large μApps and the optimal version (OA) to achieve an optimal solution for small μApps.

Both planners access the model to obtain information about the resource usage history and affinities between microservices to compute the adaptation plan. It is worth noting that the planners do not use instantaneous values of the metrics. Instead, they use historical data maintained in the model. This approach provides more reliable limits (max and min) on the resource needs of each microservice.

Finally, both planners can handle stateful microservices and data stores, since the data stores are also wrapped into microservices. However, REMaP cannot handle data sync across different hosts after migrating a stateful microservice. Hence, the migration of stateful microservices may lead the μApp into an inconsistent state.

5.5.1 Heuristic-based Affinity Planner (HBA)

The heuristic planner (HBA) reorganizes the placement of microservices that make up a μApp in a cluster. The planner computes how to rearrange the microservices so that microservices with high affinity are co-located, while microservices' resource usage and availability of resources at the host are taken into account. This planner is inspired by the First-Fit [9] approximation algorithm (see Algorithm 1) to compute the list of movements necessary to reconfigure the μApp.

Algorithm 1 iterates over the affinities and tries to co-locate the microservices associated with them. For each pair of microservices (m_i, m_j) linked by an affinity, the algorithm attempts to place m_j onto the host of m_i (H_i). If H_i does not have enough resources, the algorithm tries to put m_i onto the host of m_j (H_j). If both hosts do not have enough resources to co-locate m_i and m_j, these microservices remain at their original hosts. When a microservice is placed into a new host, it is marked as *moved* and cannot move anymore, even if it has an affinity with other microservices. In the end, a list of movements is generated containing microservice identities and their new locations.

This algorithm does not guarantee that the list of moves computed is optimal for a cluster given a set of microservices.

Algorithm 1: Variant of the First-Fit algorithm to move microservices.

```
1  moved ←[ ]
2  // affinities are in decreasing order
3  forall the a ∈ affinities do
4      // r(m) gets the microservice usage resources
5      // r(H) get the amount of free resources in host H
6      // Microservices m_i, m_j have affinity a
7      m_i ∈ H_i // m_i located at host H_i
8      m_j ∈ H_j // m_j located at host H_j
9      H_i ≠ H_j
10     hasMoved ← false
11     if r(m_i) + r(m_j) ≤ r(H_i) ∧ m_j ∉ moved then
12         H_j ← H_j − m_j
13         H_i ← H_i ∪ m_j
14         hasMoved ← true
15     end
16     else if r(m_i) + r(m_j) ≤ r(H_j) ∧ m_i ∉ moved then
17         H_i ← H_i − m_i
18         H_j ← H_j ∪ m_i
19         hasMoved ← true
20     end
21     if hasMoved then
22         moved ← moved ∪ [m_i, m_j]
23     end
24 end
```

5.5.2 Optimal Affinity Planner (OA)

Planner OA optimizes the placement of μApps. Given a list of affinities between microservices, this planner computes an optimal configuration of the microservices in a cluster. The optimization is calculated by using a SAT solver [23]. We state our placement optimization problem as follows:

Maximize:

$$\sum_{(j_i,j_k,\text{score})\in A, n\in \text{Hosts}} \text{if } \left(p\left(j_i,n\right)\wedge p\left(j_k,n\right),\text{score},0\right) \quad (2)$$

Subject to:

$$\left[\sum_{n}^{\text{Hosts}} p\left(j,n\right)\right]=1 \quad \text{for } j\in \textit{Microservices} \quad (3)$$

$$\left[\sum_{j}^{\text{Microservices}} \text{if } \left(p(j,n),M(j),0\right)\right]\leq M(n) \quad \text{for } n\in \text{Hosts} \quad (4)$$

$$\left[\sum_{j}^{\text{Microservices}} \text{if } \left(p(j,n),C(j),0\right)\right]\leq C(n) \quad \text{for } n\in \text{Hosts} \quad (5)$$

Where:

- *Microservices* is the set of microservices to be deployed,
- $p(j,i)$ is true if microservice j is placed on host i,
- $A\subset \textit{Microservices}\times \textit{Microservices}\times \mathbb{N}$ associates an affinity score to a pair of microservices,
- *Hosts* is the set of hosts available for placing microservices,
- $M(j)$ is the memory required by microservice j,
- $M(n)$ is the memory available in host n,
- $C(j)$ is the number of cores required by microservice j, and
- $C(n)$ is the number of cores available in host n.

Equation 2 defines the objective function, maximizing the sum of affinity scores of all co-located microservices j_i, j_k. This equation returns *score* if $p\left(j_i,n\right)\wedge p\left(j_k,n\right)$ evaluates to true and 0, otherwise.

Equation 3 is a constraint enforcing that each microservice is placed on precisely one host. For each microservice, the sum $p(j,n)$ over all hosts must be 1, meaning that for each microservice j, p must be true exactly once. Note that while a microservice can only be placed on one host, a host may have multiple microservices placed on it. Equations 4 and 5 enforce that each host has sufficient memory and cores for executing the microservices.

This planner tries to minimize the number of hosts used to deploy a μApp and maximize the affinity scores on each host. In practice, this algorithm places microservices with high affinity together, considering the host resources and the microservice resource usage history.

Unlike the HBA planner, the OA planner is guaranteed to find an optimal placement, i.e., oen that minimizes wasted resources.

5.6 Executor

Component *Executor*, shown in Fig. 3, is designed to apply the adaptation plan computed by the *Planner* in the μApps safely. As μApps are constantly changing, an adaptation that was computed but has not been yet applied could be unsafe to apply due to external factors. As the *Model* is causally connected, if the application is upgraded and has part of its architecture changed, the model reflects its new configuration. Hence, the *Executor* validates the change on the *Model* before applying the change to the μApp. If the change is no longer valid, then it is discarded.

The *Executor* component essentially translates high-level commands defined by the planner as low-level primitives of the management tool. Hence, an executor is necessary for each management tool.

The executor first checks whether it is possible to move a microservice from one host to another in the model or not. In some situations, the executor cannot perform this movement. For example, assume that microservices A and B have two instances, A.1 and A.2, and B.1 and B.2, respectively. Besides, A.1 and A.2 have high affinity with B.1 and B.2 as well. As mentioned in Section 5.4, the analyzer checks in the model that the microservice of type A has high affinity with B. Also, suppose that the planner has computed that A.2 should be co-located with B.2, but at runtime, A.2 has been de-allocated due to an unexpected scale-in action. Therefore, the movement to co-locate A.2 with B.2 becomes invalid.

By only using valid movements guarantees that only safe changes occur in the μApp. If the movement is valid, the executor attaches the microservice to the destination host and unsets this microservice from the source host.

Finally, the executor deals with microservice instances that come up during the adaptation process. Once the model has the microservices linked by the affinities, the executor can use this information to drive where the new replicas will be placed. For example, given two microservices A and B with high affinity. Initially, there might be only replicas A.1 and B.1. However, during adaptation to co-locate A.1 and B.1, the microservice B scale out, and a B.2 replica is generated. The *Executor* will check the model and find the affinity between A and B. First the *Executor* will attempt to co-locate B.2 with A.1, but it rechecks the model, and the host where A.1 is placed does not fit another replica of B. So, the executor will try to co-locate B.2 with another microservice such that both of them have an affinity. If such microservice does not exist, the executor maintains replica B.2 at the host where it was instantiated.

5.7 Model manager

The *Model Manager* is the core component of REMaP. It implements the causal connection between the model and instances of the microservices that compose the μApp.

In essence, the model manager triggers the adaptation by coordinating the MAPE-K elements, and by maintaining the model at runtime.

The *Model Manager* has two key elements: *Model Handler* and *Adaptation Engine* (engine). The model handler populates the model using data collected by the monitor. At runtime, the model is represented as a graph (see Fig. 5), and the model handler performs changes on this graph, e.g., the inclusion of affinities, and microservices moves.

The adaptation engine coordinates the actions of the MAPE-K components and provides an interface to add new analyzers and planners. Moreover, this engine triggers the adaptations. In this paper, the adaptation performed is the placement optimization applied to the microservices. The adaptation is triggered in timed intervals set by the μApp engineer, and each μApp has its timer. However, when the μApp is upgraded (e.g., new versions of its microservices are deployed), the timer is reset to wait for this new version generates enough data to populate the model. When the time interval is reached, the control loop is started. The analyzer calculates the affinities, updates the model, and notifies the planner. The planner uses the affinities to compute an adaptation plan. The planner sends the adaptation plan to the executor that migrates the microservices.

The causal connection has two steps. In the first step, known as *reflection*, REMaP receives data collected by the monitor and uses it to create the model. In the second step, the *reification*, REMaP consolidates the changes to the model into the executing μApp through the executor.

6 Implementation

6.1 Monitoring

We implemented the *Monitoring* component (see Fig. 3) to collect data from Influxdb[18] and Zipkin[19].

Influxdb stores resource information from the microservices and hosts. Heapster, a Kubernetes plug-in[20], collects resource usage information of microservices and hosts. It inspects microservice containers and hosts, and stores CPU and memory metrics into Influxdb.

Zipkin is a distributed tracing system to collect messages exchanged between microservices. Developers have to instrument the microservice with code snippets informing which messages Zipkin needs to capture. Once collected, Zipkin stores and makes them available via an API.

We also implemented a Kubernetes client to collect signals from the cluster and cluster configuration. Kubernetes has an API that provides cluster data such as available hosts and running microservice instances. This data is used to populate the model as illustrated in Fig. 5.

Each monitoring component wraps the underlying monitoring technology and exposes some interfaces: *MetricsInspectionInterface* provides information about resources usage, *MessagesInspectionInterface* gives information about exchanged messages, and *ClusterInspectionInterface* stores information about cluster data and organization, e.g., hosts and running microservice instances. These interfaces are combined into a *InspectionInterface* exposed by the monitoring.

Finally, the monitor also implements a listener to handle DevOps events. The listener receives events from Travis. Travis[21] is a continuous integration tool that signals when a new deployment event occurs, such as upgrading a μApp. Hence, when engineers upgrade the μApp, this listener resets the timer in the adaptation engine, as described in Section 5.7.

6.2 Analyzer

As presented in Section 5.4, the affinity analyzer retrieves information about messages exchanged by the microservices from the runtime model and calculates their affinities using Eq. 1. Our analyzer uses the EMF framework to look up the elements in the model. The EMF framework[22] has inner mechanisms to transverse the model transparently. Hence, the analyzer only looks up elements by their types, in this case, lookup for message types.

After calculating the affinities, the analyzer generates two lists of affinities. One is a sorted list that the analyzer dispatches to the planner. Another list is aggregated, by summing affinities of the same type of replicas into one affinity.

6.3 Planner

The planner computes the movements to optimally rearrange the microservices. It creates a list of moves to transfer a microservice from one host to another and passes this list to the model manager. Then, the model manager forwards the list to the executor.

As mentioned in Section 5.5, we implemented two planners: HBA and OA.

6.3.1 HBA Planner

This planner goes through each affinity generated by the analyzer and checks if it is possible to move one of the microservices, as defined in Algorithm 1. If the movement is valid, then it is stored in the *Adaptation Script*. It is worth observing that only stateless microservices can be moved. As mentioned in Sections 3 and 4, the movement of stateful microservices can raise issues on μApps execution. Hence, we decided to add this constraint to planner HBA.

To better illustrate this point, given microservices A.1 and B.1 running on hosts H.a and H.b respectively, the planner checks if A.1 and B.1 fit onto H.a. If possible, a movement is computed to move B.1 to H.a. Otherwise, the planner checks if both microservices fit on H.b.

In this case, a movement is computed to move A.1 to H.b. If both microservices neither fit on H.a nor H.b, then this affinity is discarded, and the planner tries the next affinity. The HBA planner only moves microservices to hosts where they already execute. In this example, microservices A.1 and B.1 can only be moved to hosts H.a or H.b.

After passing over all affinities, the planner forwards an *Adaptation Script* to the executor.

6.4 OA Planner

Planner OA computes the placement of microservices using an SMT solver to calculate the optimal arrangement of microservices in such a way that minimizes the number of used hosts and maximizes the affinity scores for each host.

Our implementation uses the Z3 SMT solver[23]. The optimization is modelled as a satisfiability statement [23] that can return the optimal placement to a given input. However, SMT solvers usually use brute force to compute an optimization. Hence, as the placement of microservices is an NP-Hard problem, SMT solvers are unable to calculate an optimal placement for large instances of this problem.

Planner OA transforms the SMT solver output into a list of moves like *move(microservice, source=host, target=host)*. Next, it sends the list to the model manager that forwards it to the executor.

Unlike HBA planner, OA planner can move *A.1* and *B.1* to hosts other than H.a and H.b. Furthermore, the current implementation of *planner OA* cannot differentiate stateless and stateful microservices. As OA planner must create a satisfiability formula for all attributes used to compute an adaptation plan (microservices affinities and metrics of CPU and memory), adding a dimension increases the difficulty of calculating an optimal adaptation plan exponentially, even for small μApps, e.g., less than 12 microservices. Hence, we choose not to use this constraint in the computation of the adaptation plan.

6.5 Executor

The executor has an engine that identifies the kind of adaptation plan received from the planner and executes it automatically.

Similarly to the monitor, the executor also provides wrappers to existing management tools. When the executor receives the adaptation plan, it applies the moves one-by-one on the model. If a move cannot be applied, as discussed in Section 5.6, it is discarded. Otherwise, the move is sent to the management tool wrapper that applies it to the μApp. Currently, the implementation includes wrappers for Kubernetes and Docker Swarm.

In Kubernetes, the wrapper applies the changes by updating how the microservices are attached to the hosts in the deployment description maintained at runtime by Kubernetes. Kubernetes goes through all services updating their placement attribute. Next, it executes the update, e.g., it creates a replica of the updated microservice in a new host, starts the microservice and automatically removes the old one.

Docker Swarm wrapper works similarly. However, unlike Kubernetes, Docker Swarm first removes the previous microservice and then creates a new replica in the new location.

It is worth observing that, in both wrappers, if the executor detects a failure after applying the changes, the adaptation process stops, and the change is undone in both the μApp and model.

6.6 Model manager

REMaP (see Fig. 3) wraps the Model Handler and Adaptation Engine maintains the Model at runtime, and connects all MAPE-K related components.

REMaP coordinates the MAPE-K components through its built-in messaging mechanism. The Monitor sends collected data to the Model Manager that dispatches them to the messaging mechanism. The Model Handler is responsible for building up the Model. When the Model Handler builds the model, it signals the Analyzer that must evaluate it. Next, the Analyzer signals the Planner to compute the adaptation plan, i.e., it generates the adaptation script. Finally, the adaptation script is delivered to the Executor that carries out the adaptation. All signals are received and dispatched via the messaging mechanism.

The Model Handler also uses the EMF framework to maintain the *model* at runtime. EMF abstracts the construction/traversal of the model and provides a robust notification mechanism to notify when the model changes, signalling these changes to other components. In our implementation, the *Model Handler* captures these signals to update the number of messages and total data exchanged between the microservices.

When a *message* is attached to a microservice instance, EMF signals to the Model Handler that the model was updated. The signal includes the message attributes. The EMF notification uses actuators to update the microservice instance automatically by counting the new message attached to it and the message size. Once the microservice instance is updated, EMF signals to the Model Handler that the microservice was updated. Recursively, the EMF notification mechanism updates the *application*, counting the total number of messages exchanged by the whole μApp and the total amount of data exchanged.

REMaP uses the Adaptation Engine to handle a timed event to retrieve all messages from the model, as described in Section 5.7. After an adaptation, the Model Manager needs to wait for a while before building the model. This time is necessary because the model is constructed using

execution data collected from the μApp, e.g., resource usage. In the current implementation, DevOps engineers are responsible for setting up this time delay. This information is part of the event signalled during the μApp deployment. The continuous delivery tool, Travis, handles the deployment of the microservices and notifies the *Monitor* via a Web hook when the building process finishes.

7 Evaluation

We used mock and empirical evaluations to assess the performance of REMaP. The mock evaluation focused on the time to analyze and adapt different μApp architectures generated artificially during the experiments. In practice, this approach served to show the limits of REMaP, e.g., the size of μApp, such as the number of microservice replicas and hosts, that the Analyzer can examine, and how many hosts the Planner can save through optimizing the placement of microservices. On the other hand, the empirical approach shows the resource consumption of REMaP during its execution and its impact on an existing μApp, named Sock Shop[24]. In both cases, REMaP was executed on a dedicated machine equipped with an Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz with 16 GB of RAM and running Ubuntu 16.04 LTS.

7.1 Mock evaluation

The objective of this evaluation is to understand the impact of REMaP on μApps in an entirely controlled scenario. To assess this impact, we considered two performance metrics, namely *number of saved hosts* and *time to compute the adaptation plan*. The metric *number of saved hosts* shows us how many hosts we can save in a μApp deployment after carrying out the adaptation plan computed by the Adaptation Manager (see Fig. 3). The metric *time to compute an adaptation plan* measures how long the Adaptation Manager takes to compute the adaptation. The measurement of this metric includes three phases: monitoring, analysis, and planning. In the end, the sum of the time to compute each of these phases gives us the time to calculate the adaptation plan.

Before presenting the experiments, we first defined the μApps to be used. We generated two artificial topologies, as shown in Fig. 6, using Barabási-Albert scale-free network model [24]. Both topologies emulate configurations widely adopted in μApps: API-gateway and Point-to-Point.

Barabási-Albert is a well-known model to generate random network graphs. A graph of n nodes grows by attaching new nodes with m edges to existing nodes with high degree (preferential attachment).

Similarly to the Barabási-Albert model, the microservice architecture is used to design reliable applications by interconnecting hubs of microservices. A hub means several replicas of a given microservice and its use makes the application more reliable in such a way that failure of one replica does not compromise the entire application. A μApp usually spreads out as a consequence of splitting a microservice (old vertex) into several other microservices (new vertices). This feature motivates our use of the Barabási-Albert model to represent μApps.

For consistency and reproducibility, we used the NetworkX[25] library to generate complex networks. algorithm encapsulates the Barabási-Albert model function to guarantees that all graphs generated are connected like actual μApps.

We set $m = n - 1$ and $m = 2$ to generate the API-gateways (Fig. 6a) and Point-to-point (Fig. 6b) topologies, respectively. These values guarantee that the generated graphs have similar structures even when varying the nodes (microservices) and edges (affinities).

We used the resource usage of microservices with a uniform distribution: CPU over the interval 1–500 millicores and memory in the range 10–500 MB. Hosts have 4000 millicores and 8GB of memory.

In the experiments, we used three different planners to compute the adaption plan (see Section 5.5): *HBA*, *OA* and a modified version of *OA*. This modified version calculates a placement only considering the resources available in the hosts, which means that it neither tries to maximize the affinities score in a host nor minimizes the number of hosts to be used.

We conducted the experiments in three steps. Firstly, we measured the *monitoring time* of each topology. In this step, we gathered data stored in the monitoring stack of the mock μApp to draw up the model.

Next, we simulated several scenarios for 10 minutes varying the number of messages exchanged by the microservices $\left(10^1 \text{ to } 10^5\right)$ and the number of microservices available $\left(10^1 \text{ to } 10^3\right)$ to see how these parameters affect the computation of affinities. Initially, each microservice is running on a dedicated host, i.e., we used the spread strategy (see Section 4). In this second step, we measured the time to compute the affinities, namely *analysis time*.

Finally, in the last step, we measured the *planning time*. We measured the time to compute the adaptation plans based on the affinities calculated in the previous step. Moreover, we calculated how many hosts we can save by applying the generated adaptation plans.

The number of movements computed by a planner is not proportional to the number of microservices replicas and hosts in the model. Hence, we decided to create artificial movements to evaluate the Executor. In this case, we measured the time to move 10^1 to 10^3 microservices using Kubernetes. We choose Kubernetes because it is a widely used management tool.

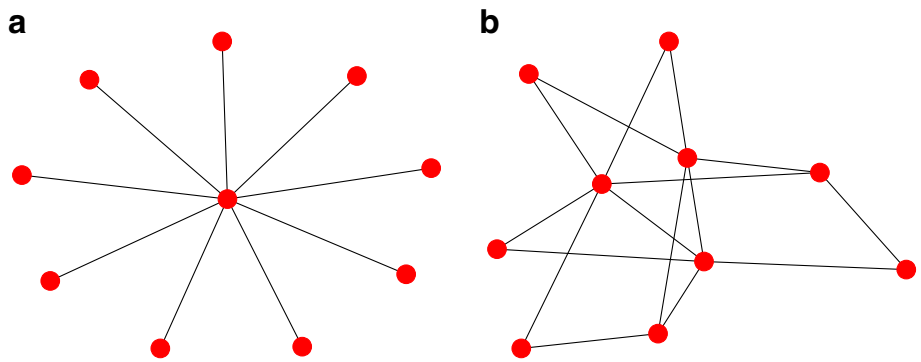

Fig. 6 Topologies used in the experiments. **a** API Gateway, **b** Point-to-Point

Table 1 shows a summary of parameters and their values used in our experiments.

Figure 7 shows the number of hosts saved by the adaptation process considering various topologies and configurations, i.e., a different number of microservice instances, hosts, and exchanged messages. Before the adaptation, all microservices were deployed using the spread strategy: one microservices per host.

In the experiments, planner *OA* only computes optimal placements for μApps having fewer than 20 microservices. In these cases, planner *OA* can save up to 85–90% of the hosts compared to the original deployment. Similarly, Planner *OA modified* only works with μApps having fewer than 20 microservices. However, the savings are worse, i.e., around 20% of hosts and in a few situations a saving of 40–45%.

Only planner *HBA* can compute placements for μApps larger than 20 microservices. In that case, planner *HBA* can save up to 85% of hosts. However, for μApps with more than 30 microservices, planner *HBA* cannot save more than 30% of hosts.

Figure 7 shows that the topology affects the computation of the placement optimization. As shown in Fig. 7a and b, the results of planners *HBA* and *OA* are different. The difference happens because planner *HBA* implements a heuristic. As the API-gateway topology is similar to a star, and edge microservices only have the affinity with one core microservice, the algorithm tries to place all edge nodes together with the core, and when the core host is at capacity, other microservices are not moved. The heuristic behaviour drastically limits the number of hosts that can be saved in dense μApps (Point-to-Point topology). In a Point-to-Point topology, each microservice could have affinities with several others. Thus, the heuristic instead of trying to group most of the microservices with a single core microservice, like in API-gateway topology, it creates several groups os microservices (hubs) in which the core microservice is that one with high affinity degree (i.e., more affinities with other microservices). Then, heuristic spreads the microservices over the cluster according to the location of microservices with a higher degree, limiting the number of hosts saved.

Table 1 Parameters of the mock experiment

Parameter (Factor)	Values (Levels)
Planner	HBA, OA, and OA Modified
Topology	API Gateway Point-to-Point
Number of exchanged messages	10^1, 10^2, 10^3, 10^4, 10^5
Number of microservices	10^1, 10^2, 10^3
Number of Microservices Movements	10^1, 10^2, 10^3

Planner *HBA* saves fewer hosts in a scenario with fewer affinities between microservices (API-Gateway topology) than in dense topologies (Point-to-Point topology). In contrast, planner *OA* saves up to 85% of hosts when optimizing Point-to-Point topologies, and up to 90% of hosts when optimizing API-Gateway topologies, as illustrated in Fig. 7c. However, in this case, we can observe that planner *OA* cannot compute an optimization for all instances of the μApp with API-Gateway topologies (like in the Point-to-Point topology).

There are cases in which the *OA* planner does not output a placement. In some cases, for μApps with fewer than 20 microservices, Z3 cannot relax the constraints defined in the optimization specification (see Section 5.5). If one of the constraints cannot be satisfied for some reason, the optimization cannot be resolved. This is not a limitation of Z3, but rather our design decision to require an all or nothing solution. The *OA* planner may also fail to produce a placement by timing out on μApps with many microservices (in our experiments, we restrict Z3 to 10 min.). In Fig. 7c and d, the squares with an 'X' are for experiments in which the constraints could not be satisfied. The squares with an '\' are for experiments in which Z3 could not finish in under 10 min.

Figure 7c and d show that the topology affects the computation of the placement optimization. Planner *OA*

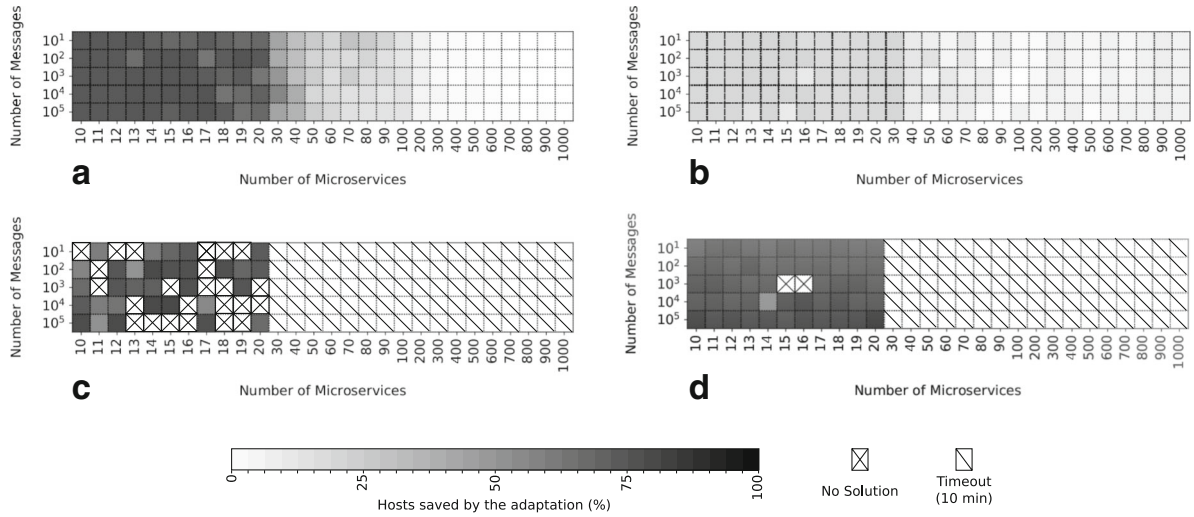

Fig. 7 The percentage of hosts saved for different configurations and planners. **a** API-Gateway topology and planner HBA, **b** Point-to-Point topology and planner HBA, **c** API-Gateway topology and planner OA, **d** Point-to-Point topology and planner OA

produces better results working on dense μApps graphs (Point-to-Point topology) than sparse ones. The planner has more data to compute better solutions as dense μApps have more affinities (links between microservices) than sparse applications (API-Gateway topology). Planner *OA modified* has no significant results and is worse than others planners: it cannot work on μApps larger than 20 microservices, neither yield results better than other planners (expect planner *HBA* applied on point-to-point topology).

Due to the exhaustion of resources to compute the placement using an SMT solver, planner *OA modified* does not work with more than 20 microservices. Conversely, planner *HBA* has not this limitation and works appropriately up to 1000 microservices. Figure 7c and d show that planner *OA modified* works better on the Point-to-Point topology and can save up to 40% of the used hosts initially. Although planner *OA* can save up to 90% of hosts when working on the API-Gateway topology, there are many situations in which this planner can not optimize μApps having more than 12 microservices. This fact happens because planner *OA modified* assumes that the current placement is already good enough for the execution, and it is not necessary to move further microservices to new places.

Figures 8, 9, 10 present the time to compute an adaptation plan. Figure 8 shows that planner *HBA* spends most of the adaptation time collecting cluster data (Monitoring Time). The monitoring step is computationally expensive because it has to collect a lot of data over the network and transform them to draw up the model.

The impact of the analysis (Analysis Time) while computing the adaptation is small when compared with the monitoring time. However, when we observe scenarios having a large number of messages (10^5), the analysis impact is more expressive. During the adaptation process, the *analyzer* traverses the model and computes the affinities (see Eq. 1). Although part of the calculation (total number of messages and total data exchanged between microservices) is already computed and kept in the model, the *analyzer* is still traversing all messages to check the microservices they link. The planning step of planner *HBA* is faster than the other planners because it uses a simple algorithm to compute the placement. As described in Section 5.5, the algorithm iterates over microservice tuples and tries to fit two associated microservices in just one of the two hosts. However, when planner *HBA* is applied on large μApps (more than 500 microservices), the planning time is within 15% and 60% of the whole time to compute an adaptation plan. This result is apparent when the planner is used in dense μApps, i.e., Point-to-Point topologies.

Similarly to planner *HBA*, the impact of the analysis while computing the adaptation using planner *OA* is small, as shown in Fig. 10. The use of planner *OA modified* has a better performance to compute an adaptation plan than planner *OA*, but planner *OA modified* cannot guarantee that the results are optimal. The use of planner *OA modified* enabled us to compute a quasi-optimal placement of up to 20 microservices and hosts in less than 4 s. Furthermore, as shown in Fig. 9, due to the brute force of the SMT solver, planner *OA*, consumes so much time computing an adaptation plan (planning time) that monitoring and analysis times become irrelevant to the total time. Finally, the NP-Hard nature of the problem makes planner *OA* unable to compute a configuration having more than 20

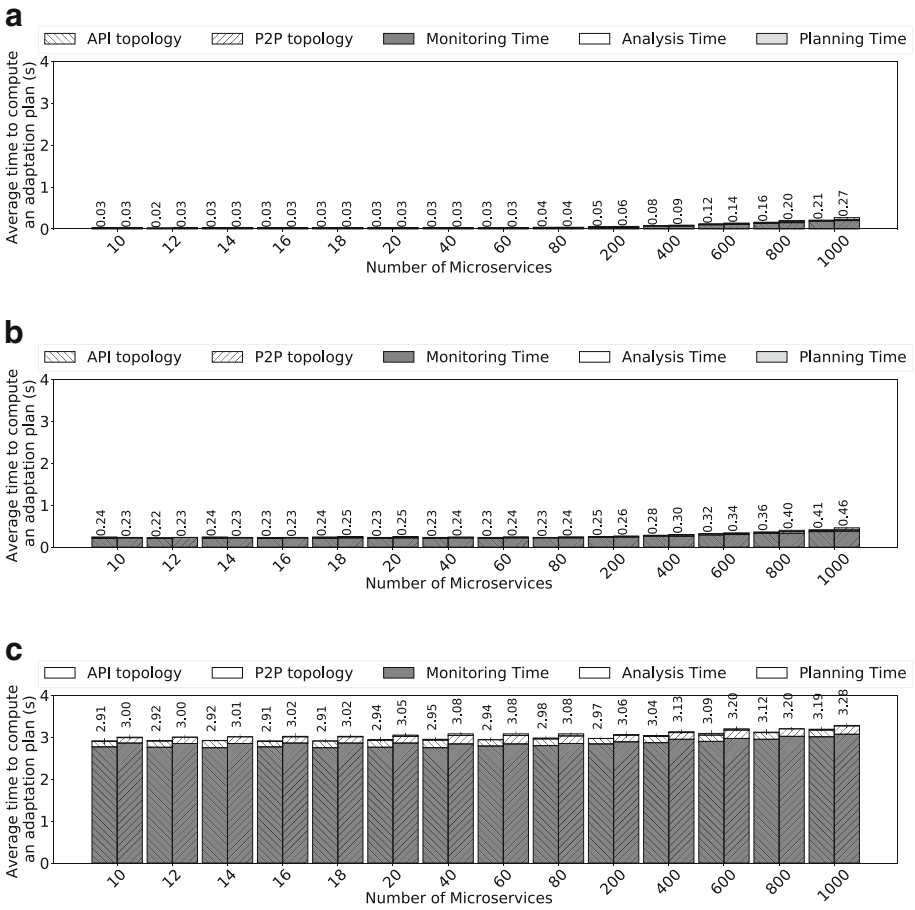

Fig. 8 Time to compute an adaptation plan - Planner HBA. **a** 10^3 messages, **b** 10^4 messages, **c** 10^5 messages

microservices. In the experiments, we set a timeout of 10 min. to compute an adaptation and in most cases, planner *OA* extrapolates this time.

The time to compute an adaptation plan using planners *HBA*, *OA* and *OA modified* varies according to the μApp topology as shown in Figs. 8, 9, and 10. The time to compute the adaptation of API topologies is, in general, better than one to compute the P2P topologies. This fact happens because the P2P topology has more affinities (links) between the microservices which make the computation of the adaptation plan (optimization) harder.

Finally, once the adaptation mechanism computes the new placement, the executor applies the changes by moving the microservices. In the case of Kubernetes, Fig. 11 depicts the average time and its standard deviation for moving microservices by using Kubernetes, these values show that the time to move microservices scales linearly.

Analyzing the adaptation scenarios using planner *HBA* (Fig. 8), the executor has the longest duration as shown in Table 2. The executor takes around five seconds to move up to 100 microservices that is a reasonable size for a μApp, up to 60 s to move 1000 microservices (large μApp). The total time for computing an adaptation plan takes around 0.03 and 3.06 s for μApps up to 200 microservices, and 10^3 and 10^5 messages exchanged, respectively. Meanwhile, it takes 0.21 and 3.28 seconds for computing an adaptation plan for μApps up to 1000 microservices, and 10^3 and 10^5 messages exchanged, respectively.

We cannot reasonably compare the reconfiguration time of planners *OA* and *OA modified*. These planners cannot compute an adaptation plan for more than 20 microservices. Planner *OA modified* only moves a few microservices and these moves do not necessarily optimize the final placement of the μApp. In turn, planner *OA* might take more than 15 seconds on average to compute an adaptation plan for μApps with fewer than 20 microservices. In some cases, planner *OA* may take more than 150 s with 20 microservices.

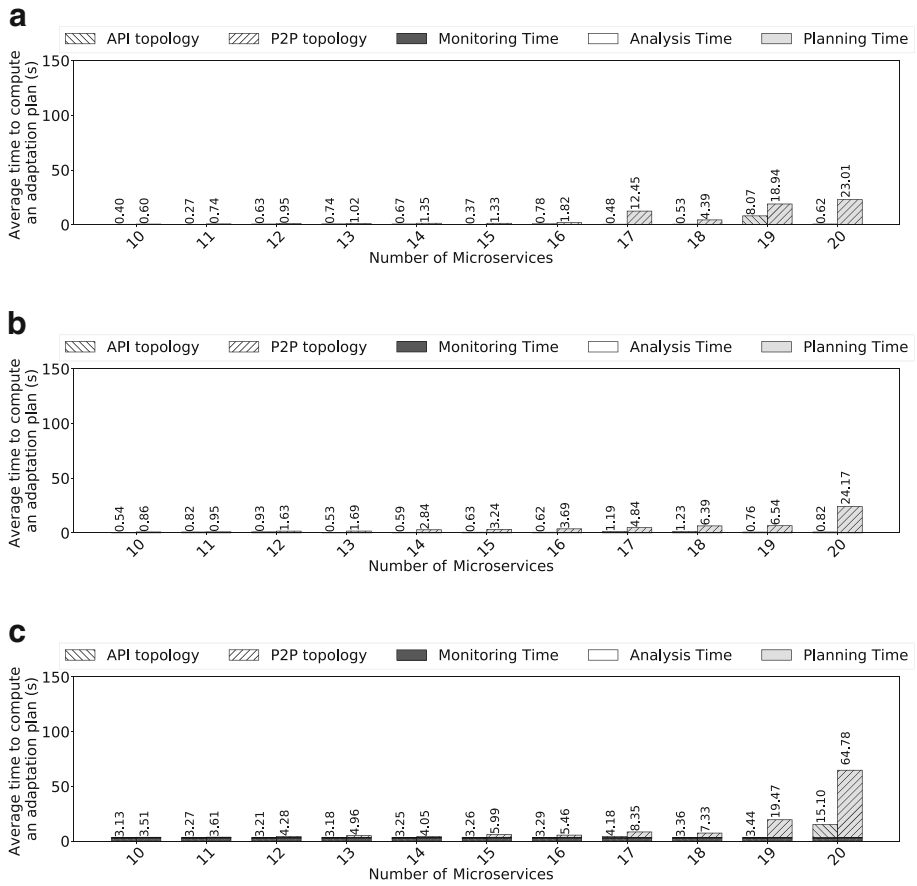

Fig. 9 Time to compute an adaptation plan - Planner OA, **a** 10^3 messages, **b** 10^4 messages, **c** 10^5 messages

When compared to the time to compute an adaptation plan using planner *HBA*, the *executor* starts to take a long time (more than 10 s) to reconfigure μApps with 200 microservices. This time increase linearly, and the *executor* can take up to 60 s to reconfigure a μApp with 1000 microservices.

7.2 Empirical evaluation

The objective of the empirical evaluation was to understand the impact of REMaP on an existing μApp, namely Sock-shop and to measure the resource consumption of REMaP. Sock-Shop is an open-source μApp of an e-commerce site that sells socks, and whose architecture is shown in Fig. 1. It has been widely adopted as a benchmark to evaluate microservices [11] and includes ten microservices and five databases. The microservices are implemented in Go, Java, and Node.JS, and the databases used are MySQL and MongoDB.

Sock-Shop was deployed on Kubernetes and configured on top of Basic A4 VMs in Azure[26]. Initially, the default Kubernetes scheduling strategy deploys each microservice. During the execution, we used REMaP to optimize the placement of the microservices by collocating some of them according to the generated adaptation plans. During the experiments, we gradually increased the number of requests to Sock-shop's front-end until it saturated and began to drop requests.

In the first experiments, we measured the *round trip time* (RTT - y-axis in Figs. 12, 13 and 14) of a request to Shock-shop. This action exercises a purchase that is the most significant flow of execution within Sock-shop and activates 9 microservices.

In all experiments, the adaptation was carried out using planners *HBA* and *OA*. We do not use planner *OA modified* due to the poor results achieved in the mock experiments. All empirical experiments have the Sock-shop deployed by the Kubernetes without any optimization as the baseline.

We conducted these experiments with two different versions of the μApp: *Sock-shop fully instrumented* and *Sock-shop partially instrumented*. In the first case, all microservices are instrumented to collect all messages exchanged. This strategy allows REMaP to build up the

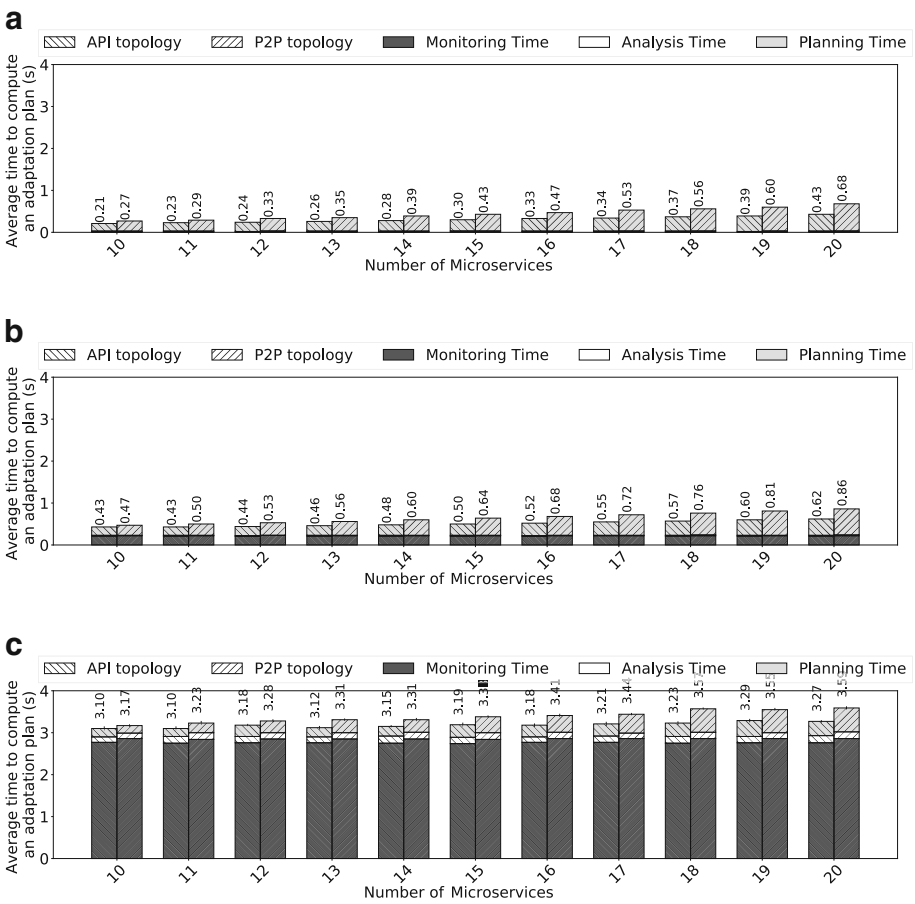

Fig. 10 Time to compute an adaptation plan - Planner OA Modified, **a** 10^3 messages, **b** 10^4 messages, **c** 10^5 messages

whole graph of the μApp. In the second case, six Java microservices are instrumented to collect their inbound and outgoing messages. In this case, REMaP builds up a partial graph of the application. Besides, we evaluated two deployment strategies:

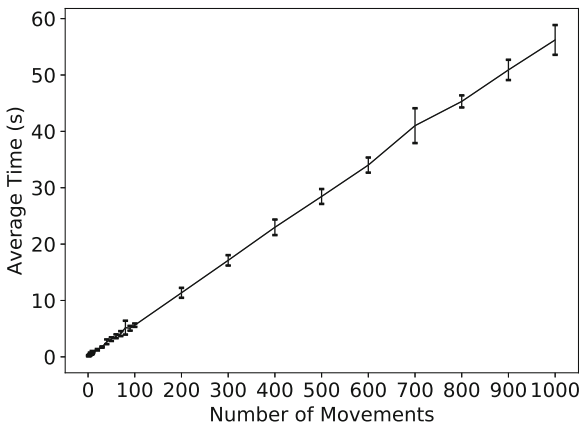

Fig. 11 Average time and its standard deviation to move microservices using Kubernetes

1. **Fully distributed (1-1):** The cluster has the same number of hosts and microservices and each microservice executes alone in a host;
2. **Partially distributed (N-1):** The cluster has 50% of hosts of the fully distributed deployment and some microservices are co-located in a host.

In both cases, Kubernetes is responsible for deciding where to put each microservice. We summarize the parameters of our experiments in Table 3.

To show the instrumentation impact on the μApp, we run an experiment comparing the Sock-shop performance. In this scenario, Sock-shop was deployed by Kubernetes, and REMaP applies no optimization on the μApp configuration.

Table 2 Time comparison for computing an adaptation plan (Planner HBA) and executing it on the cluster

#μServices	Computing adaptation plan (s)			Executor(s)
	10^3 msgs	10^4 msgs	10^5 msgs	
200	0.05 - 0.06	0.25 - 0.26	2.97 - 3.06	5
1000	0.21 - 0.27	0.41 - 0.46	3.19 - 3.28	60

As expected, the instrumented version of Sock-shop placed without optimization has the worst performance than the non-instrumented one as shown in Fig. 12.

When Sock-shop is instrumented and deployed into a small cluster, and the microservices are co-located (N:1 – N microservices per host) without any placement optimization (regular Kubernetes), we observe an overhead of approximately 58% on the RTT time when compared with the non-instrumented version (see Fig. 12). In the deployment in a large cluster without co-locations (1:1 – 1 microservice per host) and placement optimization, we observed an overhead of approximately 200% on the RTT (see Fig. 12). This high overhead is caused by the fact that all microservices are remote to each other (latency degradation) and remote to Zipkin. In this configuration, metadata are stored in Zipkin before and after each request between microservices. In N:1 deployment, there are several microservices co-located with each other and co-located with Zipkin, which reduces the latency and has a better overall performance.

μApps are highly distributed and to observe their runtime behaviour is a hard task. The instrumentation of μApps is necessary to observe and track the behaviour of the μApp at runtime, which is required for reliable management. Hence, we used the instrumented version of Sock-shop configured with Kubernetes as a baseline for the following experiments.

Although the instrumentation is necessary, it can be partial. For instance, engineers may choose to only instrument some microservices. To evaluate this scenario, we carried out an experiment to compare the performance of the fully and partially instrumented Sock-shop along with different optimization strategies. The partial instrumentation included four of ten microservices. Figure 13a and b show the results.

In the N:1 deployment, only the optimization carried out by the OA planner outperforms the results obtained with regular Kubernetes. By contrast, in the 1:1 deployment, all planners improve the μApp performance. The results show that the optimization computed for a partially instrumented μApp only improves its performance if it is fully distributed. In the case in which there are microservices co-located before the optimization, the optimization may degrade the performance of the μApp. The lack of a full application graph, due to the partial instrumentation, leads the optimization to ignore critical microservices in a workflow in such a way that the latency of these remote microservices degrades the performance of the μApp.

The previous experiments only optimized the μApp by migrating stateless microservices. However, in some scenarios, stateful microservices should be moved. Currently, REMaP cannot sync data after the migration. However, it is possible to evaluate the impact of co-locating I/O bound microservices. The next experiment evaluates the effect on the μApp by allowing the migration of stateful and stateless (together), and stateless only microservices.

In all cases (Fig. 14), the co-location of several stateful microservices degrades the performance. However, in the 1:1 deployment, planner OA migrating stateful microservices reaches the baseline and planner HBA still improves the μApp performance, even when moving stateful microservice, as shown in Fig. 14b.

REMaP, like other management tools, does not use I/O metrics from the μApp execution to compute the placement of microservices. As data stores are usually I/O bound, when co-located, they jeopardize the μApp performance by degrading the μApp performance up to 148% and 7% in N-1 and 1-1 deployments, respectively. Moreover, due to the architecture of Sock-shop (see Fig. 1), each

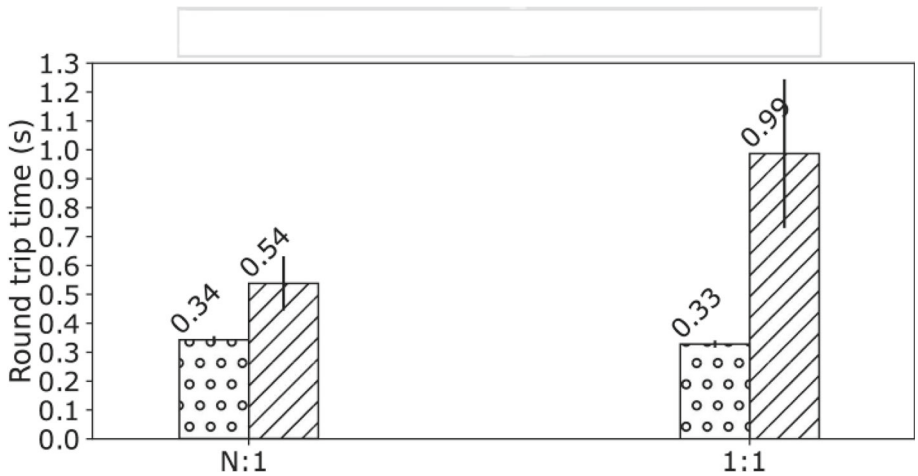

Fig. 12 Sock-shop non-instrumented (Kubernetes) versus instrumented (Kubernetes)

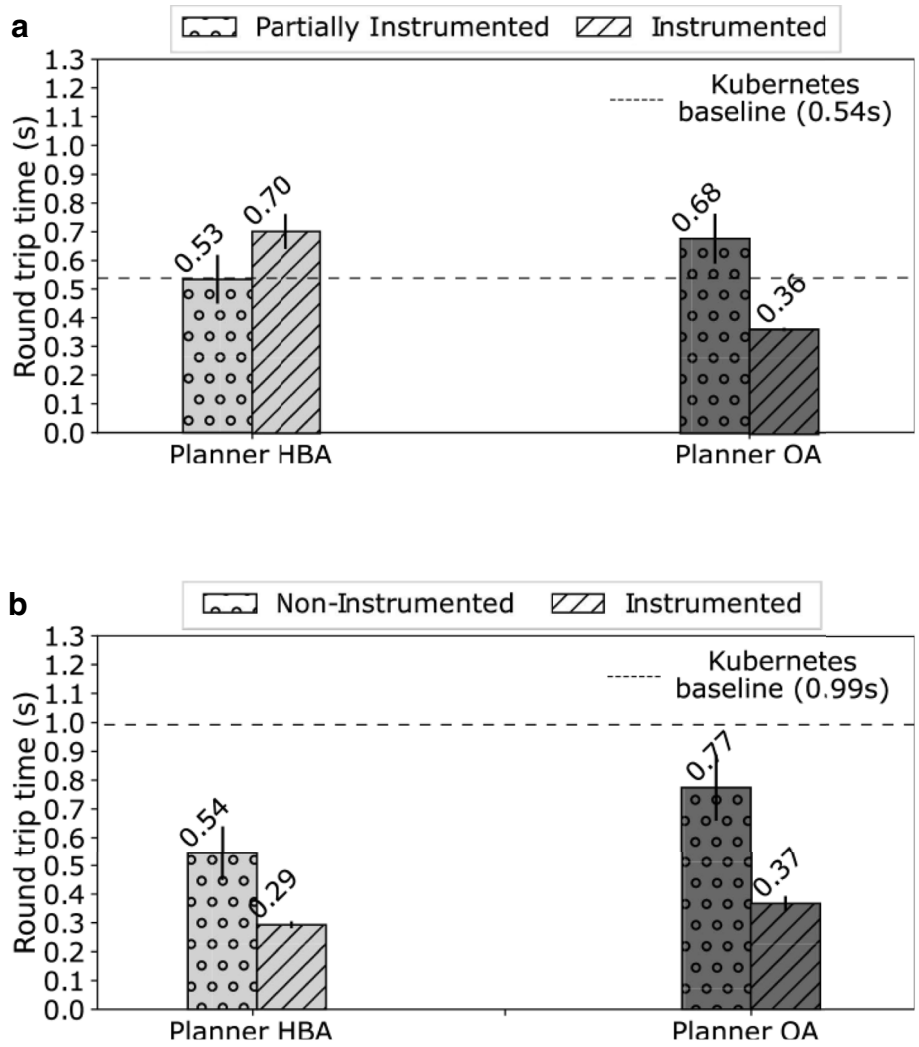

Fig. 13 RTT comparison when Sock-shop is fully instrumented and non-instrumented. **a** N:1 Deployment, **b** 1:1 Deployment

stateless microservice is co-located with its respective stateful microservice (data store). Planner OA, however, assumes that all pairs are part of a hub and the whole hub should be co-located in a single host if CPU and memory requirements are satisfied. The I/O contention degrades the overall performance of the μApp.

In addition to the impact of different planners on the μApp performance, we also evaluated the number of resources they can save in the cluster and compared against the Kubernetes deployment. Table 4 shows these comparisons. We can observe that deployment in 1:1 saved more hosts. Kubernetes attaches a microservice to a node following a variation of the First-Fit algorithm, i.e., the nodes are listed and sorted based on the number of resources available, and then each microservice is attached to the first node in the list. If there are more microservices than nodes, the process repeats until all microservices have been attached, or there are no more resources available.

Similarly to planner HBA, the Kubernetes scheduler cannot guarantee an optimal placement of microservices. Unlike Kubernetes, however, the HBA planner uses the affinities to guide the placement process. In this way, planner HBA reduces the resources used while degrading the μApp performance by 30%.

Finally, planner OA guarantees an optimal placement and improves the performance of the μApp in 1:1 deployment. However, in the N:1 deployment, the performance is worse because REMaP is unable to move cluster-dedicated components, i.e., containers used for providing health information about the cluster and microservices like InfluxDB and Zipkin. Hence, REMaP co-locates some Sock-shop microservices with these types of containers, degrading the μApp performance. As planner HBA

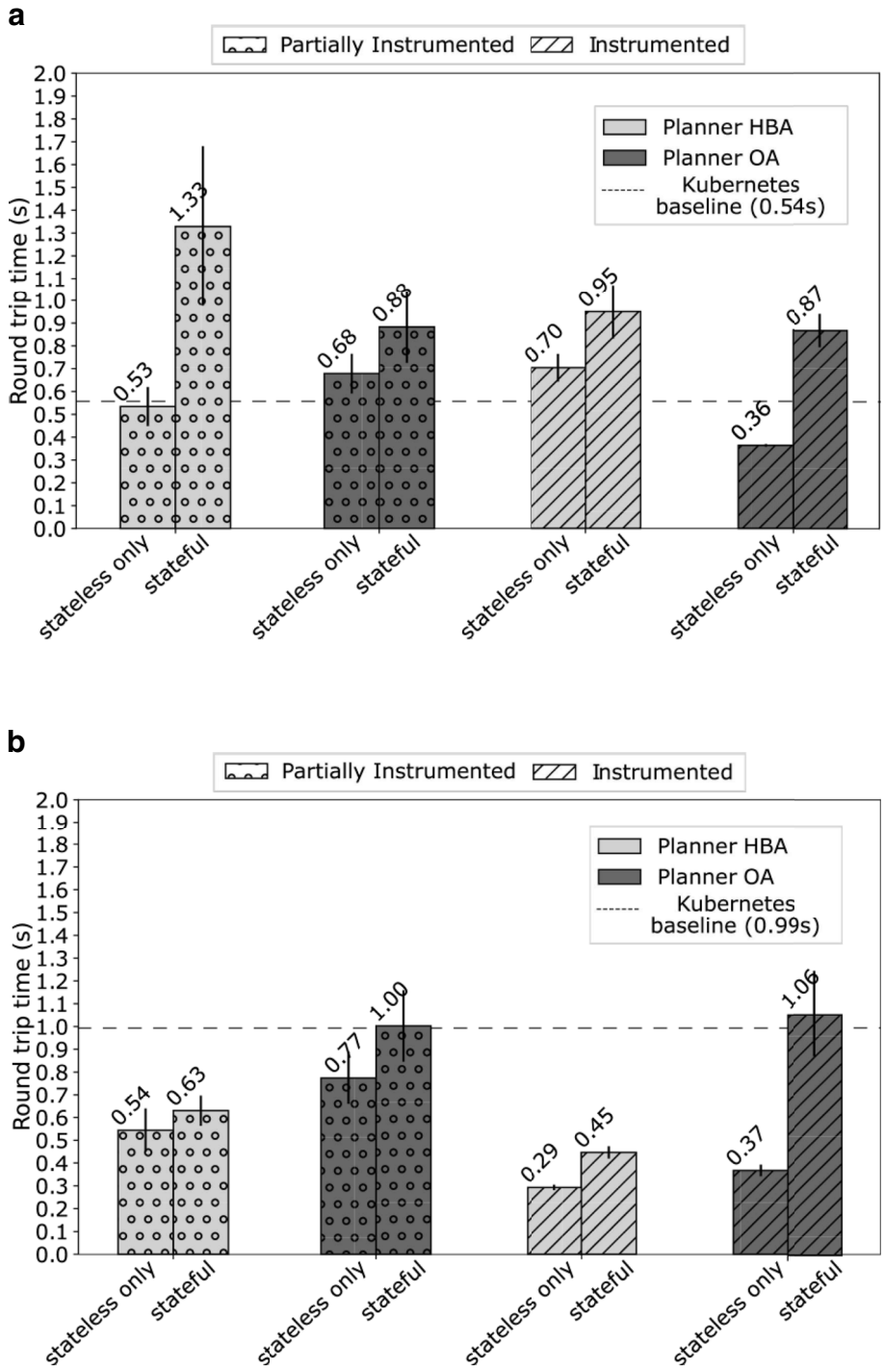

Fig. 14 RTT comparison when optimization is applied considering the migration of stateful microservices or stateless only. **a** N:1 Deployment, **b** 1:1 Deployment

Table 3 Parameters of the empirical experiment

Parameter (factor)	Values (levels)
Planner to compute placement	Kubernetes, Planners HBA/OA
Number of hosts available	7 (N:1), 15 (1:1)
Microservices able to migrate	Stateless only, stateless and stateful
Instrumentation of the μApp	Fully/partially instrumented

migrates fewer microservices than planner OA, this limitation of REMaP is less apparent.

To conclude our empirical evaluation, we measure REMaP's resource consumption when optimizing the Sock-shop placement. The resource consumption of REMaP was measured by computing the optimization of Sock-shop 100 times in a row. Although this is an unrealistic behaviour, it was useful to show how REMaP works in a high demand scenario. The results are presented in Fig. 15.

Planner HBA consumes approximately 1.65GB of memory (see Fig. 15a) and planner OA consumes 0.8GB (see Fig. 15b). For a μApp like Sock-shop, planner OA has a better memory usage due to the technologies used in the Z3 implementation. Z3 is implemented in C++ and REMaP uses a Python binding to call Z3. Planner HBA, by contrast, is implemented in Java. According to [25] Java is approximately 4.5× more memory consuming than C++ and 2.15× more memory consuming than Python. These characteristics help us to explain its current memory usage. However, for bigger μApps, planner OA can quickly run out of memory. Due to the brute-force approach used by the SMT solver: Z3 tests all possibilities when searching for a solution, and its memory usage grows exponentially according to input. Planner HBA, on the other hand, has memory usage that is polynomially bounded by the input, making the growth of its memory usage slower than planner OA.

The CPU consumption of planner OA is highly variable: a consequence of the Z3 execution. When Z3 starts to run, its CPU consumption grows quickly and remains high while the optimization is computed. As shown in Section 7.1, the calculation of an adaption for μApps up to 20 microservices can take up to one minute. Planner OA takes around 8s to compute an optimization and executes the other steps of the adaptation process in 138ms. The time to compute optimization masks the CPU usage of the other adaptation steps in the plot.

By contrast, planner HBA, shown in Fig. 15a, has a more stable CPU consumption as its optimization algorithm is less complex (asymptotically) than planner OA. This fact makes the time for computing each step of the adaptation faster than planner OA. As a consequence, the CPU is busy most of the time. Thus, planner HBA has a slightly higher CPU consumption than planner OA.

To summarize, planner OA is recommended if the μApp is fully instrumented and has fewer than 20 microservices. The average time for REMaP optimization of a μApp using planner OA is 8.5s.

Considering μApps bigger than 20 microservices, planner HBA and Kubernetes are better choices. If there are few hosts available and the μApp performance is critical, Kubernetes is preferred over REMaP. However, if resource usage is vital, REMaP with the HBA planner is the better choice. Finally, if there are many resources available, i.e., more hosts than microservices, planner HBA should be selected. The average time for REMaP to optimize a μApp using the HBA planner is 3.5s.

It is worth observing that while dozens of seconds seem to be a long time to compute and execute an adaptation plan, in a real setting this result is sufficiently fast. A μApp that is too eager to adapt is as bad as one that takes too long. As the proposed system is feedback-oriented, it is essential to take into account that actions take a while to impact the metrics. Hence, in general, mechanisms that avoid oscillations, over- and under-shooting, are essential.

Table 4 (#hosts saved by REMaP)/(original Kubernetes deployment) by each placement optimization

Kubernetes deployment		Stateful		Stateless	
		HBA	OA	HBA	OA
N:1	instrumented	2/7 (28%)	3/7 (42%)	1/7 (14%)	2/7 (28%)
	non-instrumented	2/7 (28%)	3/7 (42%)	1/7 (14%)	2/7 (28%)
1:1	instrumented	5/15 (33%)	6/15 (40%)	2/15 (13%)	3/15 (20%)
	non-instrumented	5/15 (33%)	6/15 (40%)	2/15 (13%)	3/15 (20%)

8 Related work

8.1 Runtime Adaptation of μApps

Microservices practitioners have identified the need for tools to manage μApps [1] automatically. App-Bissect [7] uses runtime information to decide when to rollback the μApp to a previous version and improves μApp performance. Existing approaches also focus on automatic scaling in/out of microservices [6, 26]. None of these approaches investigate the flexibility of μApps to explore automatic runtime placement options [27].

8.2 Microservice placement

Some initiatives focus on improving the scheduling of microservices across a cluster. Gabbrielli et al. [28] use static values of resource demands to deploy microservices, similar to commercial tools such as Kubernetes and Docker Swarm. Bhamare et al. [29] present a multi-objective algorithm to schedule microservices on a cluster taking into account function chains to optimize the execution of μApps. However, they neither monitor runtime attributes nor reconfigure the μApps.

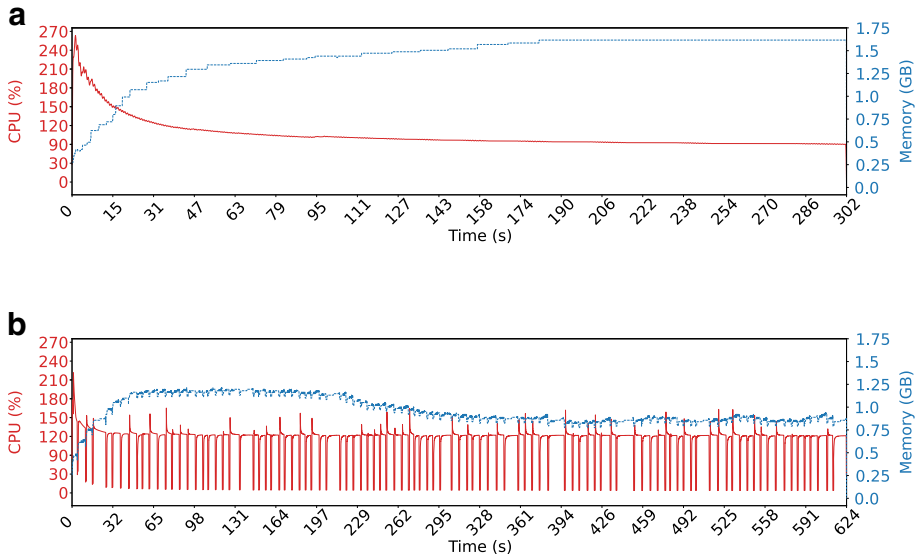

Fig. 15 CPU and memory consumption of Planner HBA and Planner OA CPU along 100 executions in sequence. **a** Planner HBA, **b** Planner OA

8.3 Allocation based on affinity

Several strategies address the allocation of VMs on clouds to improve different aspects of non-microservice-based applications, such as cost and QoS [30].

Placement and runtime migration of VMs considering their affinities have also been studied [3, 31–35]. In general, these approaches mainly group VMs based on their communication affinity to reduce the overhead imposed by communication latency.

Unlike our approach, these studies do not detect the actual usage of resources and allocate VMs based only on static resource values. Moreover, due to the monolithic nature of applications deployed in VMs, it is not common to reconfigure the VM placement to improve applications' requirements since they are not dynamic like μApps.

Finally, in addition to communication affinity, other kinds of affinities have been explored. Some examples are data affinity [36] that tries to deploy the application close to the data being used, and affinity by feature [37], in which an application is deployed according to available host configuration.

8.4 Allocation in High-Performance Computing

In high-performance computing (HPC), there are several affinity-based strategies to allocate jobs (processes, VMs or containers). The most common approaches compute the affinity of jobs and raw resources (e.g., CPU, GPU, I/O), and are not focused on inter jobs. For instance, [38] calculates the affinities between resources and jobs in such a way that an affinity is a metric of how much a resource contributes to the execution of a job. Yokoyama et al. [39] calculate the affinity based on how much competition exists between jobs for a given resource. More sophisticated affinities may consider an inter-process relationship to allocate processes to the same virtual CPU and/or virtual CPUs to the same physical CPU, avoiding cache misses along the application execution [40]. In all cases, the allocation is on the process and CPU levels, at a lower level than REMaP.

There are works in which affinities are calculated based on job communication, like REMaP. AAGA [3], CIVSched [41], and Starling [31] are some examples of this approach. In general, this prior work computes the jobs' affinity based on the bandwidth between them. The algorithms used are in a sense a variation of the First-Fit algorithm [9], which tries to fit related jobs into a node with the available resources. The REMaP's planner HBA is also inspired by the First-Fit algorithm. Finally, like REMaP, some of these algorithms [31] can reconfigure the allocation of jobs at runtime based on changes in the communication patterns of the application.

Finally, like μApps, the HPC domain has no framework to unify several metrics of the environment to compute affinities. Broquedis et al. [42] proposes a unified interface to gather resources in an HPC cluster named *hwloc*. The idea of *hwloc* is to provide a unified view of low-level resources is a similar way to REMaP. However, *hwloc* was designed only to provide an interface that can be used by another scheduler. REMaP, in turn, is a complete solution to update the placement of the elements of a μApp, is such a way that it may use *hwloc* to draw up a more sophisticated model by using finer grained data of the physical machines in a cluster.

8.5 Models for microservices

Rademacher [43] surveyed the use of models in microservice and service-oriented architectures. Models have been used mostly at development/specification time, and to the best of our knowledge, there is no work using model@run.time in the microservice domain. Derakhshanmanesh [44] shared his vision of how models might be used across the full software lifecycle, including runtime, to evolve applications. He proposes the use of domain-specific modelling languages (DSML) and model transformations to define and evolve a microservice application at the architectural level.

Dullmann and van Hoorn [45] have used models for engineering microservice testbeds. Developers create models and generate Java code and deployment files (Maven, Docker, and Kubernetes). The microservices generated are automatically instrumented to collect metrics at runtime.

Microservices Ambients [46] uses static and dynamic analysis of application to generate a model to transform monolithic architectures into microservice-based architectures.

Zúñiga-Prieto et al. [47] propose a model, inspired by SoaML, to integrate microservices. Developers can specify the integration, describing the integration logic and architectural impact of integration, without taking into consideration the particularities of the target cloud environment. In the end, the model is used to generate skeletons of microservices, the integration logic, and scripts to automatically deploy and integrate the microservices.

MicroArt [48] makes reverse engineering of a μApp, through static and dynamic analysis of code repository and μApp deployment in a cluster, to create an architectural model. The model is refined by the μApp architecture, creating a global view of the application, but it does not include any runtime information, such as resource usage or messages exchanged. The generated model is used for multiple purposes, for example, architectural reasoning, analysis, and documentation.

Leitner et al. [49] present CostHat, an approach to model deployment financial cost. The model is generated by analyzing messages exchanged among microservices. CostHat uses the model to calculate costs due to I/O operations, processing, remote calls, and so on.

9 Conclusion and future directions

Despite the flexibility provided by microservices, existing management tools do not use valuable runtime informati on to adapt μApps. We propose REMaP, a platform-independent runtime adaptation mechanism to reconfigure the placement of microservices in μApps based on their communication affinities and resources usage. At the core of our approach is a novel optimization that changes the placement of microservices at runtime. We present our design and implementation for monitoring and constructing the runtime models of μApps and use these models to adapt the application by using the causal connection between the model and the μApp.

Our work shows that by updating the placement of μApps at runtime, we can save up to 80% of hosts being used. In some cases this causes a performance degradation for the μApp, but in some cases this improves μApp performance. We also show that it is difficult to achieve an optimal placement due to the complexity of the placement problem. However, the results obtained with our heuristic algorithm highlights the potential of approximation algorithms to improve the placement of μApps to reduce resources usage and improve performance.

Our implementation uses triggers, defined by engineers, to start the adaptation, i.e., interval-based triggers. In future work we will consider adaptation triggers that do not require human involvement. Hence, we plan to extend the proposed solution to trigger the adaptation based on statistical analysis of the μApp behaviour.

We also aim to extend the adaptation strategy by improving the analysis and planner phases to find affinity hubs in μApps rather than simple microservices tuples. Our experiments highlight that the use of hubs by approximation algorithms may produce better results than co-location of simple tuples.

This work also shows that the number of messages fetched from the μApp does not affect the placement computation. Hence, we expect to update our solution to collect the minimum data necessary to compute the adaptation. As the monitoring is a performance bottleneck for our approach, this improvement is a necessary next step. However, the number of instrumented microservices is critical for an optimal placement computation. In future work, we will develop a lighter instrumentation strategy for μApps.

The migration of microservices at runtime uses the APIs exposed by the management tools. However, in practice, these APIs are insufficient. For example, the management tools do not expose primitives to move individual microservice instances during scaling. We plan to investigate extensions to existing management tools to better support microservice migration.

Finally, in this work, we used a scale-free network model to represent mock μApps. We will study real-world μApps to validate the underlying hypothesis that power-law models, like the Barabási-Albert model, are appropriate for modeling μApps.

Looking forward, we hope that the proposed approach will be adopted by management tools to further automate μApp management using non-trivial runtime data, such as communication patterns and historical resource usage data.

Endnotes

[1] https://www.oasis-open.org/committees/wsbpel
[2] https://kubernetes.io
[3] https://docs.docker.com/engine/swarm
[4] https://microservices-demo.github.io
[5] http://microservices.io/patterns/reliability/circuit-breaker.html
[6] https://github.com/google/cadvisor
[7] https://prometheus.io
[8] https://www.influxdata.com
[9] https://www.fluentd.org
[10] https://www.elastic.co/products/logstash
[11] https://www.elastic.co/products/elasticsearch
[12] https://aws.amazon.com/cloudwatch
[13] https://zipkin.io
[14] https://uber.github.io/jaeger
[15] https://www.elastic.com/products/kibana
[16] https://jaxenter.com/nobody-puts-java-container-139373.html
[17] https://stackoverflow.com/questions/36759132/why-does-docker-crash-on-high-memory-usage
[18] https://docs.influxdata.com/influxdb
[19] http://zipkin.io
[20] https://github.com/kubernetes/heapster
[21] https://travis-ci.org
[22] https://www.eclipse.org/modeling/emf
[23] https://github.com/Z3Prover/z3
[24] https://microservices-demo.github.io
[25] https://networkx.github.io
[26] https://azure.microsoft.com

Acknowledgments
We thank Dr. Sam Bayless for sharing his expertise on SMT solvers, and for helping us with Microsoft Z3.

Funding
This work is supported by:

- CAPES Brazil – grant 88881.132774/2016-01, and
- FACEPE Brazil – grant IBPG- 0892-1.03/13.

Availability of data and materials
Please contact the author for data requests.

Authors' contributions
AS is a Ph.D. candidate and this paper presents the primary outcome of his research. AS was responsible for the identification of the problem, design the solution, and carry out the experiments shown in this paper. JR, IB, and NR participated in the design and coordination of the study. Finally, JR, IB, and NR helped to draft the manuscript. All authors read and approved the final manuscript.

Authors' information
M.Sc. AS is a Ph.D. candidate in the Centre of Informatics of Federal University of Pernambuco, Brazil. He received his M.Sc. in Computer Science from the Federal University of Goiás, Brazil.
Ph.D. JR is an Assistant Professor in the Department of Electrical and Computer Engineering at the University of British Columbia, Canada. She received her Ph.D. in Computer Science from the University of Toronto, Canada.
Ph.D. IB is an Assistant Professor in the Department of Computer Science at the University of British Columbia, Canada. He received his Ph.D. in Computer Science from the University of Washington, United States.
Ph.D. NR is an Associate Professor at Center of Informatics at the University Federal of Pernambuco, Brazil. He received his Ph.D. in Computer Science from the Federal University of Pernambuco, Brazil.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Author details
[1]Center of Informatics, UFPE, Recife, Brazil. [2]Department of Electrical and Computer Engineering, UBC, Vancouver, Canada. [3]Department of Computer Science, UBC, Vancouver, Canada.

Received: 21 April 2018 Accepted: 11 November 2018
Published online: 26 February 2019

References

1. Newman S. Building Microservices. O'Reilly Media; 2015. p. 280.
2. Zimmermann O. Microservices Tenets: Agile Approach to Service Development and Deployment. Comput Sci Res Dev. 2016;32(3):301–10.
3. Chen J, Chiew K, Ye D, Zhu L, Chen W. AAGA: Affinity-aware grouping for allocation of virtual machines. In: Advanced Information Networking and Applications (AINA). 2013. p. 235–42.
4. Sampaio Jr A, Kadiyala H, Hu B, Steinbacher J, Erwin T, Rosa N, Beschastnikh I, Rubin J. Supporting evolving microservices. In: International Conference on Software Maintenance and Evolution (ICSME) - Track NIER. 2017. p. 539–43.
5. IBM. An architectural blueprint for autonomic computing. In: IBM White Paper. IBM Corporation; 2005. p. 34.
6. Florio L, Di Nitto E. Gru: An Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures. In: Autonomic Computing (ICAC), 2016 IEEE International Conference on. 2016. p. 357–62.
7. Rajagopalan S, Jamjoom H. App-Bisect: autonomous healing for microservice-based apps. In: Conference on Hot Topics in Cloud Computing. 2015. p. 16.
8. Barrett CW, Sebastiani R, Seshia SA, Tinelli C, et al. Satisfiability modulo theories. Handb Satisfiability. 2009;185:825–85.
9. Dosa G. In: Kao M-Y, editor. First Fit Algorithm for Bin Packing. Boston: Springer; 2008. pp. 1–5.
10. Casati F. Service-oriented computing. ACM SIGWEB Newsletter. 2007;2007(Winter):1.
11. Aderaldo CM, Mendonca NC, Pahl C, Jamshidi P. Benchmark Requirements for Microservices Architecture Research. In: 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE). IEEE; 2017. p. 8–13.
12. Jordan D, Evdemon J. Web Services Business Process Execution Language Version 2.0. 2007. https://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html. Accessed Mar 2018.
13. Huebscher MC, McCann JA. A survey of autonomic computing—degrees, models, and applications. ACM Comput Surv. 2008;40(3):1–28.
14. Szvetits M, Zdun U. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. Software & Systems Modeling. 2013.
15. Krupitzer C, Roth FM, VanSyckel S, Schiele G, Becker C. A survey on engineering approaches for self-adaptive systems. Pervasive Mob Comput. 2014;17:184–206.
16. Blair G, Bencomo N, France RB. Models@run.time. Computer. 2009;42(10):22–27.
17. Maes P. Concepts and experiments in computational reflection. ACM SIGPLAN Notices. 1987;22(12):147–55.
18. Weik MH. truncated binary exponential backoff. Boston: Springer; 2001. p. 1840.

19. Ward JS, Barker A. Observing the clouds: a survey and taxonomy of cloud monitoring. J Cloud Comput. 2014;3(1). https://doi.org/10.1186/s13677-014-0024-2.
20. Korte B, Vygen J. Bin-Packing. In: Combinatorial Optimization: Theory and Algorithms. Berlin Heidelberg: Springer; 2006. p. 426–441.
21. Chekuri C, Khanna S. On multidimensional packing problems. J Comput. 2004;33(4):837–51.
22. Christensen HI, Khan A, Pokutta S, Tetali P. Multidimensional bin packing and other related problems: A survey; 2016. https://pdfs.semanticscholar.org/bbcf/4ca2524cd50fdb03b180aa5f98d2daa759ce.pdf. Accessed March 2018.
23. Biere A, Heule M, van Maaren H, Walsh T. Handbook of Satisfiability, vol. 185 of Frontiers in Artificial Intelligence and Applications. IOS Press; 2009.
24. Barabási A-L, Albert R. Emergence of scaling in random networks. science. 1999;286(5439):509–12.
25. Pereira R, Couto M, Ribeiro F, Rua R, Cunha J, Fernandes JaP, Saraiva Ja. Energy efficiency across programming languages: How do energy, time, and memory relate?. In: Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering. SLE 2017. New York: ACM; 2017. p. 256–67.
26. Toffetti G, Brunner S, Blöchlinger M, Spillner J, Bohnert TM. Self-managing cloud-native applications: Design, implementation, and experience. Futur Gener Comput Syst. 2017;72:165–79.
27. Hassan S, Bahsoon R. Microservices and their design trade-offs: A self-adaptive roadmap. In: Services Computing (SCC). 2016. p. 813–8.
28. Gabbrielli M, Giallorenzo S, Guidi C, Mauro J, Montesi F. Self-reconfiguring microservices. In: Theory and Practice of Formal Methods. Springer; 2016. p. 194–210.
29. Bhamare D, Samaka M, Erbad A, Jain R, Gupta L, Chan HA. Multi-Objective Scheduling of Micro-Services for Optimal Service Function Chains. In: International Conference on Communications (ICC). 2017. p. 1–6.
30. Singh AN, Prakash S. Challenges and opportunities of resource allocation in cloud computing: A survey. In: Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference On. 2015. p. 2047–51.
31. Sonnek J, Greensky J, Reutiman R, Chandra A. Starling: Minimizing communication overhead in virtualized computing platforms using decentralized affinity-aware migration. In: Parallel Processing (ICPP), 2010 39th International Conference On. 2010. p. 228–37.
32. Acharya S, D' Mello DA. A taxonomy of Live Virtual Machine (VM) Migration mechanisms in cloud computing environment. In: A taxonomy of Live Virtual Machine (VM) Migration mechanisms in cloud computing environment. 2013. p. 809–15.
33. Leelipushpam PGJ, Sharmila J. Live VM migration techniques in cloud environment—a survey. In: Information & Communication Technologies (ICT). 2013. p. 408–13.
34. Pachorkar N, Ingle R. Multi-dimensional affinity aware VM placement algorithm in cloud computing. Int J Adv Comput Res. 2013;3(4):121.
35. Lu T, Stuart M, Tang K, He X. Clique migration: Affinity grouping of virtual machines for inter-cloud live migration. In: Networking, Architecture, and Storage (NAS). IEEE; 2014. p. 216–25.
36. Jiang J, Sun S, Sekar V, Zhang H. Pytheas: Enabling Data-Driven Quality of Experience Optimization Using Group-Based Exploration-Exploitation. In: NSDI; 2017. p. 393–406.
37. Ramakrishnan A, Naqvi SNZ, Bhatti ZW, Preuveneers D, Berbers Y. Learning deployment trade-offs for self-optimization of Internet of Things applications. In: Proceedings of the 10th International Conference on Autonomic Computing, ICAC 2013. 2013. p. 213–24.
38. Lee G, Katz RH. Heterogeneity-aware resource allocation and scheduling in the cloud. In: HotCloud; 2011.
39. Yokoyama D, Schulze B, Kloh H, Bandini M, Rebello V. Affinity aware scheduling model of cluster nodes in private clouds. J Netw Comput Appl. 2017;95:94–104.
40. Li Z, Bai Y, Zhang H, Ma Y. Affinity-aware dynamic pinning scheduling for virtual machines. In: Cloud Computing Technology and Science (CloudCom). IEEE; 2010. p. 242–9.
41. Guan B, Wu J, Wang Y, Khan S. CIVSched: a communication-aware inter-VM scheduling technique for decreased network latency between co-located VMs. Trans Cloud Comput. 2014;2(3):320–332.
42. Broquedis F, Clet-Ortega J, Moreaud S, Furmento N, Goglin B, Mercier G, Thibault S, Namyst R. hwloc: A generic framework for managing hardware affinities in HPC applications. In: Parallel, Distributed and Network-Based Processing (PDP). 2010. p. 180–6.
43. Rademacher F, Sachweh S, Zundorf A. Differences between Model-Driven Development of Service-Oriented and Microservice Architecture. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW); 2017. p. 38–45.
44. Derakhshanmanesh M, Grieger M. Model-Integrating Microservices: A Vision Paper. In: Software Engineering (Workshops); 2016. p. 142–7.
45. Düllmann TF, van Hoorn A. Model-driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches. In: International Conference on Performance Engineering Companion. 2017. p. 171–2.
46. Hassan S, Ali N, Bahsoon R. Microservice Ambients: An Architectural Meta-modelling Approach for Microservice Granularity. In: Software Architecture (ICSA). 2017. p. 1–10.
47. Zúñiga-Prieto M, Insfran E, Abrahão S, Cano-Genoves C. Automation of the Incremental Integration of Microservices Architectures. In: Complexity in Information Systems Development. Springer; 2017. p. 51–68.
48. Granchelli G, Cardarelli M, Di Francesco P, Malavolta I, Iovino L, Di Salle A. Towards Recovering the Software Architecture of Microservice-Based Systems. In: Software Architecture Workshops (ICSAW), 2017 IEEE International Conference On; 2017. p. 46–53.
49. Leitner P, Cito J, Stöckli E. Modelling and managing deployment costs of microservice-based cloud applications. In: Proceedings of the 9th International Conference on Utility and Cloud Computing; 2016. p. 165–74.