Pós-Graduação em Ciência da Computação

Adalberto Ribeiro Sampaio Junior

Runtime Adaptation of Microservices

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife

2018

Adalberto Ribeiro Sampaio Junior

Runtime Adaptation of Microservices

Ph.D. Thesis submitted by **Adalberto Ribeiro Sampaio Junior** in partial fulfillment of the requeriments for the degree of Doctor of Pholosophy on Graduated Studies in Computer Science of Centre of Informatics of Federal University of Pernambuco.

Concentration Area: Distributed Systems

Supervisor: Nelson Souto Rosa
Co-supervisor: Ivan Beschastnikh

Recife
2018

Catalogação na fonte
Bibliotecária Elaine Freitas CRB 4-1790

S192r Sampaio Junior, Adalberto Ribeiro
Runtime Adaptation of Microservices / Adalberto Ribeiro Sampaio Junior . – 2018.
134 f.: fig., tab.

Orientador: Nelson Souto Rosa
Tese (Doutorado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação. Recife, 2018.
Inclui referências e apêndice.

1. Sistemas distribuídos. 2. Microserviços. 3. Computação autonômica. I. Rosa, Nelson Souto (orientador) II. Título.

004.36 CDD (22. ed.) UFPE-MEI 2018-130

Adalberto Ribeiro Sampaio Junior

Runtime Adaptation of Microservices

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

Aprovado em: 09/11/2018.

_____

Orientador: Prof. Dr. Nelson Souto Rosa

BANCA EXAMINADORA

_____

Prof. Dr. Paulo Romero Martins Maciel
Centro de Informática/ UFPE

_____

Prof. Dr. Ricardo Massa Ferreira Lima
Centro de Informática / UFPE

_____

Prof. Dr. Vinícius Cardoso Garcia
Centro de Informática / UFPE

_____

Prof. Dr. José Neuman de Souza
Departamento de Computação / UFC

_____

Prof. Dr. Nabor das Chagas Mendonça
Programa de Pós-Graduação em Informática Aplicada / UNIFOR

ABSTRACT

The architectural style of Microservices is an approach that uses small pieces of software, each one with a single responsibility and well-defined boundaries, integrated with lightweight and general purpose communication protocols to build an application. The decoupling promoted by microservice usage makes continuous delivery cheaper and safer to be applied in comparison to other architectural styles thus allowing a microservice-based application (μApp) to be continuously updated and upgraded at runtime. For this reason, many companies have adopted microservices to facilitate the development and maintenance of their applications. However, high decoupling and a large number of microservices and technologies adopted, make it difficult to control a μApp. Despite the Microservice's architectural style relying on tools to automatically manage the deployment and execution of μApps, these tools are not aware of the application's behaviour. Therefore, most decisions are made manually by engineers, that analyze application logs, metrics, messages and take actions in response to triggers. This characteristic makes it difficult to make optimal decisions at runtime (i.e., optimizing the placement of microservices in the cluster). This thesis proposes an approach to bring autonomy to the microservice management tools by automatically evaluating the μApp's behaviour, allowing alterations to be made with minimum intervention. To achieve that, we present REMaP, a MAPE-K based framework that inspects and adapts μApp in a cluster through a model at run-time. This model abstracts several technologies and semantics of μApps cohesively, allowing decisions to be computed without the supervision of engineers. To show the feasibility of this autonomic approach, we used REMaP to optimize the placement of microservices at runtime by autonomously monitoring the μApp's behaviour, computing a (quasi-) optimal placement and re-configuring the μApp at runtime. Our approach allowed us to determine that it was possible to save up to 85% of servers used in the deployment of μApp by maintaining and, in some cases improving, its overall performance.

Key-words: Microservices. Autonomic Computing. Models at runtime. Placement Optimization.

RESUMO

O estilo arquitetural de Microserviços é uma abordagem que usa pequenas peças de software, cada uma com uma única responsabilidade e limites bem definidos, integradas sobre um protocolo de comunicação leve e de propósito geral para construir uma aplicação. O desacoplamento promovido pelo uso de microsserviços faz com que a entrega contínua seja segura e barata de ser aplicada, ao contrário de outros estilos arquiteturais. Assim uma aplicação baseada em microsserviços (μApp) pode ser constantemente atualizada em tempo de execução. Por esta razão, muitas companhias têm adotado microsserviços para facilitar o desenvolvimento e manutenção de suas aplicações. Entretanto, a alto nível de desacoplamento, e consequentemente, o grande número de microerviços e tecnologias adotadas, faz com que o controle de uma μApp não seja fácil. Apesar do estilo arquitetural de microsserviços depender de ferramentas para gerenciar automaticamente a implantação e execução de μApps, essas ferramentas não são cientes do comportamento das aplicações. Dessa forma, a maioria das decisões são feitas manualmente por engenheiros que analisam os logs, métricas e mensagens da aplicação e tomam ações em resposta à algum gatilho. Está característica dificulta que decisões ótimas sejam aplicadas em tempo de execução como, por exemplo, otimizar o arranjo dos microservices no cluster. Neste trabalho nós propomos uma abordagem para trazer autonomia às ferramentas de gerenciamento de microsserviços, avaliando automaticamente o comportamento da μApp e alterando a aplicação com a mínima intervenção de engenheiros. Para alcançar isso, nós apresentamos REMaP, um framework baseado na arquitetura MAPE-K, que inspeciona e adapta μApps em um cluster usando um modelo em tempo de execução. Este modelo abstrai várias tecnologias e semânticas das μApps de forma coesa, permitindo que decisões sejam calculadas sem a supervisão dos engenheiros. Para mostrar a factibilidade desta abordagem autonômica, nós usamos o REMaP para otimizar o arranjo de uma μApp. Através de um monitoramento autônomo do comportamento do μApp, REMaP calcula um arranjo *quasi*-ótimo, e reconfigura a μApp em tempo de execução. Com nossa abordagem constatamos que é possível economizar até 85% dos servidores usados na implantação inicial de μApps mantendo, e em alguns casos melhorando, a performance geral da aplicação.

Palavras-chaves: Microsserviços. Computação Autonônimca. Modelo em tempo de execução. Arranjo Ideal.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

In this chapter we introduce our work. We present the context of microservices and their unique characteristics, which motivate our desire to work with runtime adaptation on microservice-based applications (μApps). This chapter also presents the research's challenges on carrying out runtime adaptation on μApps and how other initiatives have partially handled this issue. Next, we highlight our proposal and list our contributions in adapting μApps at runtime.

1.1 CONTEXT AND MOTIVATION

As business logic moves into the cloud (JAMSHIDI et al., 2018), 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 a virtually infinite amount of resources, in which *Microservice architectural style* has become an essential mechanism in order to take advantage of these features (NEWMAN, 2015).

Architectural style tells us how to organize the code. It is the highest level of granularity and also specifies high level modules of an application as well as how they interact with each other. On the other hand, architectural patterns solve problems related to the architectural style (e.g., the number of tiers a client-server architecture has - MVC).

In the Microservice architectural style, microservice-based applications (μApps) are build up by integrating many pieces of software, known as microservices, over a lightweight communication protocol.

A *microservice* (LEWIS; FOWLER, 2014) is a decoupled autonomic software that has a specific function in a bounded context. It is intended role is to split the logic of an application into several smaller pieces of software, each with a single and specific role, integrated with lightweight general purpose communication protocols (e.g., HTTP).

Despite the many similarities between services and microservices (ZIMMERMANN, 2016), there are some fundamental differences between them, mainly related to their execution. Languages like WS-BPEL (JORDAN; EVDEMON, 2007) describe service compositions workflow while a microservice-based application (μApp) workflow is not formally specified. The μApp communication must be monitored to infer the underlying workflow and to change its behaviour it is necessary to upgrade the application, deploying different microservices.

The flexibility of μApps made microservices the favourite architectural style for deploying complex application logic. Companies such as Amazon and Netflix have hundreds

of different microservices in their applications. Amazon[1] uses approximately 120 microservices to generate one page whereas Uber[2] manages 1000s of microservices and Netflix[3] has more than 600 microservices in its application.

The decoupling and well-defined interfaces provide the ability for μApps to scale in-/out seamlessly and allowing developers to perform upgrades by deploying new service versions without halting the μApp. In addition to this, decoupling allows microservices to be developed by using different technology stacks or by changing it along the μApp's lifespan. Development teams can use or add new technologies to better fulfill the application's requirements, improving its security and reliability through frequent updates (e.g. applying bug fixes on the chosen stack), resulting in frequent component deployments.

A downside, however, of using microservices is managing their deployments. Microservices that compose an application can interact and exchange a significant amount of data at runtime, creating communications between them *affinities* (CHEN et al., 2013). These inter-service affinities can have a substantial impact on the performance of μApps depending on the placement of the microservices. High-affinity microservices, for example, will have a worse performance due to higher communication latency when placed on different hosts. Another fact which makes this scenario worse is the change in affinities at runtime as consequence of the frequent upgrades in a μApp.

Along with affinities, developers must also account for the microservice's resource usage in order to optimize the μApp's performance. Microservices with high resource usage, for example, should not be placed on the same host.

Existing management tools, such as Kubernetes[4], allow development-operations (DevOps) engineers to control μApps by setting resource thresholds on the microservices. Management tools uses 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 a given limit, the management tool starts the scaling action. Existing management tools select in which hosts to place the microservices replicas during scale out, based on the set thresholds instead of their real resource usage. In most cases, the threshold is unrealistic and leads the μApp to waste cluster resources or to lose performance due to resource contention.

μApps management is automatized by tools that monitor and analyze their behavior as well as apply adaptation actions in response to specific conditions at runtime. These tools are fundamental in handling μApps, especially those with a significant amount of microservices; however, they are not fully autonomous. They are unaware of information about the μApp context, making human intervention necessary in some management

[1] <http://highscalability.com/amazon-architecture>
[2] <https://www.youtube.com/watch?v=BT9sUm5M77y>
[3] <https://www.infoq.com/presentations/migration-cloud-native>
[4] <https://kubernetes.io/>

activities such as monitoring and analysis.

During a μApp's execution, management tools are only aware of the current snapshot of the application's resource usage and few static configurations, set by the engineers during the μApp's deployment. Behavioural information such as messages - exchanged by microservices -, their logs and resource usage history is not considered. Hence, to carry out a sophisticated and graceful management at runtime, engineers need to analyze these different sorts of data and manually apply an adaptation action in response to the results.

The lack of behavioural awareness comes from the heterogeneity of the microservices, which make it difficult to use a general purpose approach to inspect μApps at runtime. Different languages used in the development of microservices impose a semantic gap in collecting runtime data generated by the μApp. A single μApp may have different monitors to gather resource usage metrics from microservices in different languages, for example; in this case each tool has its own data format and semantic. It is the role of the engineer to handle these differences when analyzing the data. To gather other sorts of data, such as the logs and messages, additional monitoring tools are necessary; different from those used to collect resource metrics.

To make matters worse, frequent updates and upgrades to μApps make their behaviour change continuously; leaving engineers to face dynamic behavior despite all the challenges to monitor and analyze the heterogeneous data generated. This mutable behaviour forces the continuous monitoring and analysis of μApps in order to avoid flaws at runtime; solving them as fast as possible if they come up.

These varied and mutable characteristics overwhelm the engineers - imposing a high cost in managing the μApps- making it challenging to make decisions in order to control the μApp and consequently making management slow, error-prone and with poor results (HOFF, 2014; KILLALEA, 2016; SINGLETON, 2016)

An example of poor μApp management is placing microservices in a cluster. To deploy a μApp engineers might set a minimum amount of resources that the microservices need to bootstrap. At runtime, on the other hand, the resource usage history in some microservices in the μApp have different values from the ones set by the engineers; management tools, however, are not aware of this runtime behavior.

1.2 RESEARCH CHALLENGES

μApps have a dynamic behaviour. The high decoupling among their microservices allows engineers to apply updates and upgrades easily without the need to halt the whole μApp; making frequent updates and upgrades in a short time span common. Compared to other architectural styles in general, a μApp can change many times throughout a day, to which it is defined as an *inherent runtime adaptive system.*

μApp engineers apply adaptations mostly manually, underusing the adaptive potential

of these applications. In general, engineers observe the application log and metric charts looking for a situation that needs to be changed in the running μApp. They then set triggers to update the application (e.g. scale in/out a set of microservices) or manually apply an upgrade (e.g. roll out/back a microservice version). Hence, changes to μApps rely on the management of their microservices. Adaptations are mostly structural and must deal with the placement of the microservices onto the cluster.

Manual intervention on μApps is a consequence of the challenges for monitoring and deciding over the data generated by the μApp. The heterogeneity of the tools used for gathering data, the lack of a well-defined structure for the generated data, is the primary challenge on automatizing the adaptation in μApps.

Moreover, managing microservices in a cluster is a hard task. In order to place a microservice in a cluster, it is necessary to match the requirements of the microservice to the features available in a cluster node. It is a classic optimization problem of multi-dimensional bin-packing that it is hard to be applied at runtime due to its NP-Hard complexity. Management tools usually have generic strategies to place microservices in the cluster, but this placement it is not optimal, it wastes resources and threatens the μApp's performance.

In this context, the main objective of this thesis is:

To automatize the adaptation of μApps, improving the placement of microservices at runtime.

Along to this work, we explain that runtime adaptation of μApps relies on to instantiate (new) microservices somewhere in the cluster. Hence, we aiming to handle the challenges for improving the placement of microservices an runtime. We reach the improvement by optimizing the location of microservices based on their behavior – history resources usage and messages exchanged (μApp workflow).

To bring automation to adapt μApps, we must cope with three activities based on MAPE-K architecture (IBM, 2005):

1. to observe the application (monitoring);
2. to decide what to do (analysis and planning); and,
3. to perform an action (execution).

However, there are some challenges for carrying out these activities in the μApp domain. Next, we overview these challenges:

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 create a semantic challenge, which requires a single unified data model in order to be solved.

Challenge 2: Finding a high-performing placement. Microservices are usually placed using static information such as 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 leads to two sub-problems:

1. An ample 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 microservice management tools do not expose any means to perform live migration of microservices between hosts. Live migration is necessary to provide seamless runtime adaptation.

These challenges raise the question:

Can we optimize the placement of microservices by carrying out autonomous management based on the behavioural data of the μApps?

However,

How to bring autonomy to μApp management despite the several challenges in monitoring heterogeneous microservices?

Therefore, in this thesis we answer this question by introducing REMaP (**R**untim**E** **M**icroservices **P**lacement) an autonomous adaptation manager for μApps with minimum human intervention. The autonomous management observes μApps at runtime and smartly places microservices onto the cluster, thus improving performance and decreasing the waste of resources by μApps.

1.3 PARTIAL SOLUTIONS

Few papers aim to bring autonomy for the management of μApps. Florio and Nitto (2016) propose an independent mechanism, named Gru, to change the μApp deployment at runtime to provide runtime scaling. Gru auto-scales microservices based on CPU and memory usage as well as microservice response time. However, this approach is technologically locked and only works on Docker[5]. Moreover, it is out of the scope of Gru to inspect runtime behaviour of microservices (i.e. μApp workflow) to change the μApp.

[5] <https://www.docker.com>

Rajagopalan and Jamjoom (2015) propose App-Bisect – an autonomous mechanism to update a running deployment to a different version. App-Bisect monitors the μApp along their execution, tracking the upgrades of microservices. When App-Bisect detects a poor performance of the μApp after an upgrade, it reverts some microservices in the application to earlier versions thus solving the performance issue. This approach aims to improve the performance by reverting the configuration of the μApp. Again, this approach also does not analyze the behaviour of the μApp neither the microservices interactions in order to change it.

1.4 OUR PROPOSAL

To handle the challenges of optimizing the placement of microservices, we are proposing an adaptation manager structured as a control loop feedback – in which the performance is centered on the use of runtime models and verification as well as optimization strategies – to make practical runtime adaptations to the μApps.

We abstract the heterogeneity of microservices domain in a model at runtime that maintains information related to the μApps behaviour. The model is the knowledge source used by the control loop in the monitoring, analysis, planning and adaptation activities.

To show the feasibility of our approach, we use the execution environment to compute a quasi-optimal placement of a μApp based on its runtime behaviour. This computed placement aims to decrease the number of hosts used by the μApp and can, in specific cases, improve the performance of the μApp.

The focus of our work it to carry out a model-driven runtime adaptation on μApps. To demonstrate our adaptation approach, we should decrease the μApp resource usage by automatically reconfiguring the placement of microservices at runtime, based on online monitoring of the μApp, without compromising μApp performance. We do so by using an adaptation mechanism that solves the challenges above and automatically changes the placement of microservices by using their affinities and resource usage. Our solution uses a MAPE-K based (IBM, 2005) adaptation manager to upgrade the placement of μApps at runtime autonomously. The adaptation manager uses Models@run.time (BLAIR; BENCOMO; FRANCE, 2009) concepts to abstracts the diversity of monitoring stacks and management tools and guide the adaptation. In doing so our solution provides a unified view of the cluster and the μApps running under the adaptation manager.

The adaptation manager groups and places microservices with high affinity on the same physical server – this strategy contrasts with existing static approaches, which rely on information provided by engineers before μApp deployment. Hence, our adaptation manager can provide 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 is in an optimized configuration, which reduce the number of hosts needed to execute the μApp and can improve its performance in comparison to a static configuration.

We evaluated our adaptation mechanism in two scenarios, using three strategies: one heuristic-based and two based on SAT-Solvers (BIERE et al., 2009). 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 conducted adaptations that saved up to 85% of the hosts initially used by the μApps. This evaluation shows that our approach produces better results on μApps with a dense topology graph. Moreover, strategies chosen to optimize the placement by using SAT-Solvers were unable to work in large μApps (with more than 20 microservices); hence – despite the fact that our heuristic cannot guarantee an optimal result – it was able to compute new placements for any size of μApp.

In the second scenario we used the proposed mechanism to adapt a benchmark μApp[6] running on Azure. In this scenario we achieved a performance improvement of 3% and saved 20% 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 4x, indicating that the placement requires special care by engineers – which our approach automates.

1.5 SUMMARY OF CONTRIBUTIONS

This thesis makes several contributions to runtime adaptation of μApps. Next, we list these main contributions.

The definition of runtime adaptation of μApps : This thesis systematizes what the adaptation of μApps is, as well as the challenges of applying it at runtime.

The use of models for supporting the evolution of microservices : The number of technologies used to build up a μApp; the frequency of μApp changes – due to updates and upgrades – as well as the number of microservices and microservice replicas; make it complex to track their evolution. Therefore, we contribute to a discussion about how models can be used to support the evolution of μApps and propose two models to adapt microservices.

The use of affinities to drive runtime adaptation on μApps : In addition to resources available in a cluster, the behaviour of a μApp can affect its performance. Therefore, we propose the concept of affinities to define the degree of relationship between two microservices; by co-locating high related microservices, we can improve the overall performance of the application.

[6] <https://microservices-demo.github.io>

Strategy to rearrange microservices at runtime : to reconfigure microservices in a cluster – to optimize the μApp performance – and the resource usage in an NP-Hard task, in many cases, cannot be computed in a reasonable amount of time. Therefore, we use the concept of affinities during the adaptation to improve the placement of microservices to achieve (*quasi*-)optimal configurations.

A MAPE-K based adaptation mechanism for adapting μApps at runtime : Adaptation mechanism for μApps usually rely on metrics to change their configuration. In this thesis, we contribute by presenting an adaptation mechanism that relies on a runtime model of the μApp to guide its adaptation. Instead of observing only metrics such as CPU and memory usage, our approach observes the behaviour of the μApp through the interaction among their microservices, which allows more effective adaptations at runtime.

1.6 THESIS ORGANIZATION

The following thesis is organized as follows. Chapter 2 presents the basic concepts used throughout the thesis, namely: adaptive software; models and Models@run.time in runtime adaptation; and microservices.

Chapter 3 discusses the rationale and general overview of our approach; presents the challenges on runtime evolution of μApps; introduces our solution and our approach to support μApp evolution by using models and discusses the challenges on the placement of μApps. Chapter 4 shows the design and implementation of REMaP. It initiates with an overview of REMaP– describing the model used by REMaP during the adaptation of μApps– and presents details of the design and implementation of MAPE-K based components: *Monitoring, Analyzer, Planning, Execution.*

Chapter 5 presents the evaluation of our solution and results. Section 5.3.1 we present the results of the mock evaluation and in Section 5.3.2 we present the results of the empirical evaluation. Chapter 6 is an overview of the related work, positioning our work in five categories: supporting microservices evolution, Models@run.time on services domain, runtime adaptation of μApps, usage of models in microservice domain, and placement on clouds.

Finally, presents a summary of the results obtained; discusses our contributions; shows the limitations of our approach and how we aim to extend this work discussing future works.

2 BASIC CONCEPTS

In this chapter we present the primary concepts used throughout the rest of this thesis. We begin by describing the ideas of autonomic computing and adaptive software, highlighting the reference architecture MAPE-K – to develop adaptive software – and the main strategies to deploy MAPE-K in order to manage an underneath system. Moreover, we describe how the use of models facilitate system adaptation. Next we present the notion of models and their use at runtime. Furthermore, we present the motivation behind microservices; their fundamental concepts and how microservices are comparable to services. To conclude, we also present management tools for microservice-based applications (μApps) and the challenges in correctly placing microservices in the cluster.

2.1 ADAPTIVE SOFTWARE

Adaptive software are those capable of change in order to satisfy requirements. The adaptation is achieved, in most cases, through autonomic computing (KEPHART; CHESS, 2003). Autonomic computing refers to self-managing computing systems (HUEBSCHER; MCCANN, 2008) – the system controls itself. In self-managed systems human intervention is not necessary in activities such as optimization, healing, protection, and configuration. The system management is achieved by changing (adapting) some of its structural or behavioral aspects in response to internal or external stimuli.

A self-managed (or self-adaptive) system must monitor it's own signals – as well as the environment's – as well as analyze and apply actions in response to them – perhaps by modifying itself. These steps repeat themselves indefinitely as a control loop. Four facts (KELING; DALMAU; ROOSE, 2012) usually motivate changes in the system:

1. *Changes of environment's conditions*: When the infrastructure, where the system is deployed, changes due to updates or failures the system must adapted in order to deal with them without completely stopping;

2. *Changes of application requirements*: When application requirements change, in resilience or performance, the application must be updated in order to satisfy the new requirements.

3. *Detection of bugs*: When bugs are detected, the application must be updated to fix them, thus avoiding possible outages.

4. *Application evolution*: When a new version of a component is developed the application is updated by replacing the current version of the component with a new one.

These adaptations can be corrective, evolutionary or reactive (KELING; DALMAU; ROOSE, 2012). A corrective adaption is performed when a problem is diagnosed in the environment or system itself; an evolutionary adaption is performed to update a system in order to better satisfy the requirements of the system or to use new implementations or technologies; and a reactive adaption is carried out to respond to a specific monitored event.

Adaptation performed in the application can be structural or behavioural (KELING; DALMAU; ROOSE, 2012). As its name suggests, a structural adaption changes the structure of the application by adding, removing, replacing and reconnecting components. Meanwhile, a behavioural adaption changes the behaviour of the system by adding/removing functionalities or changing some configuration parameter.

In both cases, the structure of the adaption process consists of a control loop. The most relevant examples of control loops are Rainbow (GARLAN et al., 2004) and MAPE-K (KEPHART; CHESS, 2003). IBM systematized MAPE-K as a reference model for automatic control loops (IBM, 2005). MAPE-K organizes the adaptation process in four phases: monitoring, analysis, planning, and execution.

2.1.1 Adaptation Loop

Autonomous mechanisms carry out adaptations in a system by sensing signals from the environment and/or from the system itself; the autonomous mechanism decides what to do; and acts over the managed system. This sequence of steps repeat indefinitely as a loop named control loop or adaptation loop.

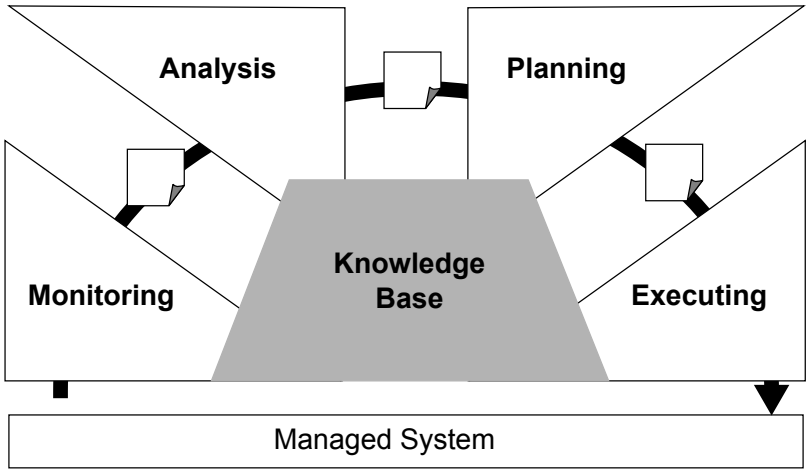
In self-adaptive systems, the adaptation loop is part of the system, whereas, in supervised adaptive systems, the process to diagnose and select an adaption is placed in a third-party element.

In MAPE-K framework, the adaption loop is made up by four stages as shown in FIG. 1: ***m****onitoring*, ***a****nalysis*, ***p****lanning* and ***e****xecution*. Salehie and Tahvildari (2009) define these activities as follows:

- *Monitoring*: The monitoring stage is responsible for collecting and correlating data from *sensors* and converting them into behavioural patterns and symptoms, which can be done through event correlation and threshold checking. MAPE-K senses a managed system through sensors, and sends the collected data to a *monitor*. The monitor then aggregates the received data as symptoms that should be analyzed.

- *Analysis*: This activity is responsible for the analysis of the symptoms – provided by the monitoring stage and the system history – in order to detect when a change is required. If a change is required, the *analyzer* generates a change request and passes it to the *planner*.

Figure 1 – MAPE-K framework for developing adaptation loops.



Source: (IBM, 2005)

- *Planning*: The planning activity determines what needs to be changed and what is the best to carry it out. In MAPE-K, the *planner* creates/selects a procedure to apply a desired adaptation in the managed resource and passes it to the *executor* as an adaptation (change) plan.
- *Execution*: The execution stage is responsible for applying the actions determined in the planning stage. It includes managing non-primitive actions through predefined workflows, or mapping actions, to what is provided by actuators and their underlying dynamic adaptation mechanisms. The *actuator* allows the *executor* to perform actions to change the managed resource.

Lastly, MAPE-K activities of share a ***k**nowledge base* that maintains rules, properties, models and other kinds of data, used to steer how to provide autonomy to the underlying system.

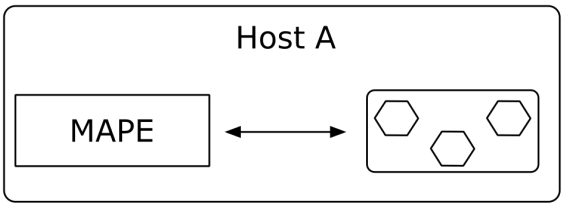
2.1.2 MAPE-K Deployment Configurations

In the adaptation of distributed systems, several deployment configurations can be applied on MAPE-K to better compute the adaptation plan without degrading the performance or threatening the execution of its managed system. Weyns et al. (2013) categorizes the deployment of MAPE-K architectures into local, remote, centralized and decentralized. Next, we present these deployments patterns and their main variations.

In a *local deployment*, FIG. 2, all components of MAPE-K run on the same host as the managed application. An advantage of this deployment is that no messages are exchanged between monitored microservices and the MAPE-K instance. The disadvantage is that both the application and the control-loop contend for the same local machine resources –

such as CPU and memory. Another issue with a local deployment is that MAPE-K lacks a global view of the application if it's components are deployed across several hosts.

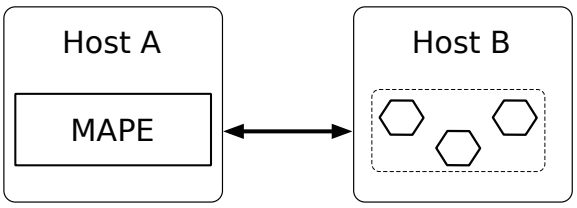
Figure 2 – MAPE-K local deployment.



Source: Adapted from (WEYNS et al., 2013)

In a *remote deployment*, as shown in FIG. 3, components that implement MAPE-K activities and the managed application run in different hosts making it necessary for monitored messages and adaptation actions to traverse the network, which incurs latency. If this latency is sufficiently high, it can jeopardize the time taken to apply an action in response to a violation and may be unacceptable. The advantage of a remote deployment is that MAPE-K can construct a global view of all components that compose the application.

Figure 3 – MAPE-K remote deployment.



Source: Adapted from (WEYNS et al., 2013)

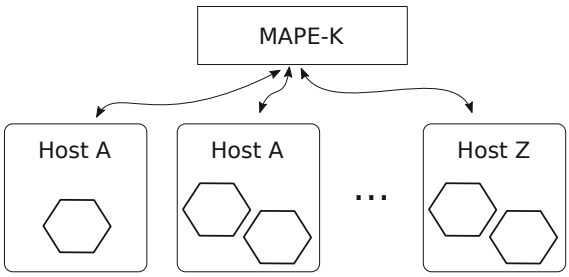
In a *centralized deployment*, as shown in FIG. 4, all the components that implement the MAPE-K activities are in a single host. A key disadvantage of centralization is that it introduces a single point of failure; leaving potential for performance degradation once each stage of MAPE-K competes for the same resources.

It is also possible to deploy a remote centralized MAPE-K instance. However, if the application is deployed across several hosts, the MAKE-K instance may be local to only a few application components and remote to others. In this hybrid approach, in case of frequently updated components, there is no guarantee that the host location where MAPE-K is deployed will be the same as the one where the application components are running.

A *decentralized deployment* can be divided into two cases:

1. **Several instances of MAPE-K**: several MAPE-K instances distributed across several hosts; and

Figure 4 – MAPE-K centralized remote deployment.

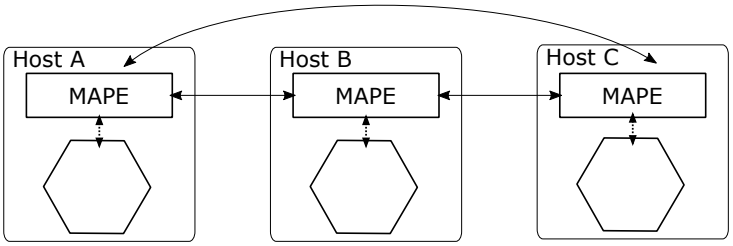


Source: Adapted from (WEYNS et al., 2013)

2. **Singe instance of MAPE-K**: a single MAPE-K instance with MAPE-K components distributed throughout different hots.

In the first deployment (**Several instances of MAPE-K**), each instance of MAPE-K receives specific events from an application. The number of instances and what they monitor depends on the partitioning strategy defined by a domain expert. To analyze the application, it is necessary to combine the results of individual instances of MAPE-K. As a consequence, this approach may require a high number of messages to achieve a global result;

Figure 5 – MAPE-K decentralized local deployment.



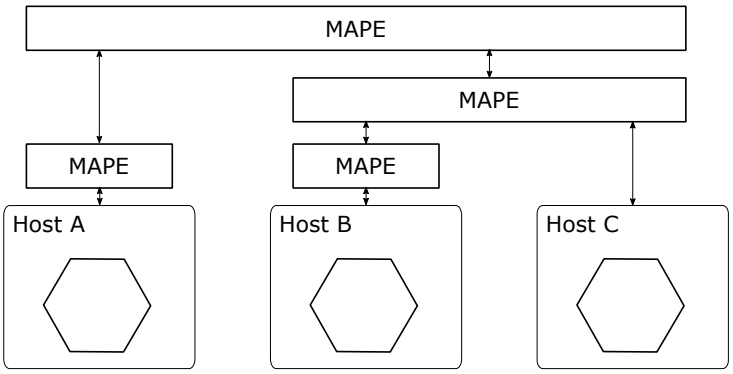
Source: Adapted from (WEYNS et al., 2013)

Every instance of MAPE-K can be deployed locally to an application (FIG. 5). If the distributed application is deployed across several hosts, then each host can has one or more instances of MAPE-K that must be coordinated to act globally on the application. In order to coordinate these actions, it is necessary to route the messages from a MAPE-K instance to another and to combine individual results into a single global outcome.

Despite the necessary coordination, this kind of deployment has the advantage of providing partial application monitoring even during severe network failures, such as network partitions. During said failure event, a MAPE-K instance can monitor and act on its local application component without needing a global view.

An alternative to the decentralized local deployment is a decentralized remote deployment (FIG. 6). The advantages are as before: remote deployments do not contend with

Figure 6 – Decentralized remote with many remote MAPE-K instances – each one in its own host.



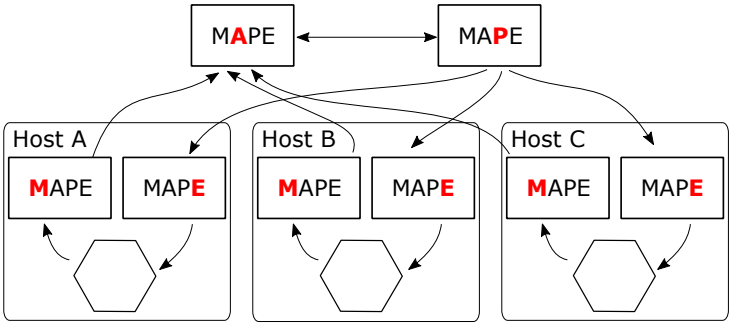
Source: Adapted from (WEYNS et al., 2013)

application components for resources and a decentralized deployment is more fault tolerant. However, unlike the decentralized local deployment, this strategy cannot guarantee property checking during network failures.

In the **Single MAPE-K with distributed components**, each component of MAPE-K can be deployed into a different host. According to (WEYNS; ANDERSSON, 2013), different combinations are possible.

For example, in the master/slave pattern (WEYNS; ANDERSSON, 2013) (FIG. 7), a monitor and an executor are local to every component of the application. Monitored events are sent to a remote Analyzer and Planner. In this deployment, the application and parts of MAPE-K do not compete for resources. Meanwhile, the local Executor removes the latency by avoiding the remote request to remote actuators.

Figure 7 – Decentralized remote: a single MAPE-K distributing each component in different hosts.



Source: Adapted from (WEYNS et al., 2013)

2.2 MODELS@RUN.TIME

Model Driven Engineering (MDE) (SCHMIDT, 2006) is an approach used in Software Engineering for specification, construction, and system maintenance. MDE aims to systematize the use of models so that they can be used for different objects along all activities in the software development process – including software execution. Models are therefore used not only during the specification and development phases or for documentation and code generation purposes, but also as a live element to maintain the state of the system (software and its environment) – keeping its structure and behaviour, as well as for guiding the system execution.

A model is a high-level representation of the system in which technical details are abstracted in favour of domain-specific and technology agnostic information. This approach separates concerns between the technology used for the system implementation and its business logic (PARVIAINEN et al., 2009), facilitating the development and maintenance of the modelled system.

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.

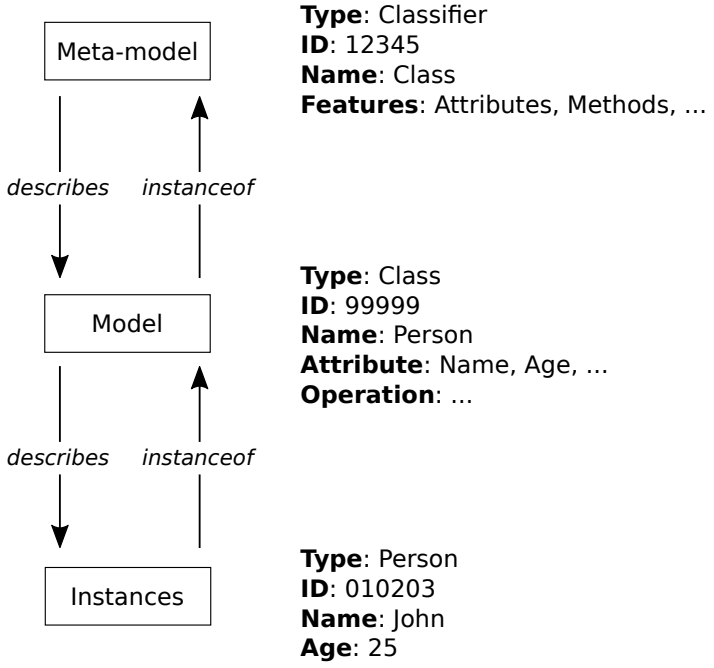
A manually created model (*user-defined model*) is usually drawn up by the application engineer in a top-down approach. The user observes aspects of the domain, removing useless information – given the context and purpose of the model –, and organizes all information respecting the model rules. User-defined models usually help the construction (NÚÑEZ-VALDEZ et al., 2017) and integration (YU et al., 2015) of systems – automatic code generation and interoperability – or for controlling the behaviour (SAMPAIO JR.; COSTA; CLARKE, 2013) – maintaining control policies to guide the system execution.

An automatically created model, on the other hand, (*emergent model*) is drawn up in a bottom-up approach. The model emerges from metrics, execution traces and several other system signals. Dedicated tools – specialized for a domain – collect all data and organize them into the model.

A model is drawn up by using its meta-model (see FIG. 8), which is the abstraction for building a model and has elements that say what can be expressed and how they are structured in the model. The characteristics of a domain are captured into the meta-model allowing it to be used for instantiating different models of the same domain. Like an object that is an instance of a class in the object-oriented paradigm, a model is an instance of a meta-model in MDE.

The meta-model is usually defined for a specific domain. The domain specialist determines which are the main elements and their meanings as well as how these elements should be organized in the model in order to best represent a particular domain. All this information is then compiled into the meta-model, guiding it on how to draw up the model.

Figure 8 – Relationship between meta-model, model, and real world instances.



The *abstract syntax* and *static semantics* are strict rules in the meta-model that define and guide the model creation. The *abstract syntax* defines the elements available to represent the model and its relationships. In the Java meta-model, for instance, class; attributes and methods, as well as the relationship that a class has to the attributes and methods – are all defined in its abstract syntax. In turn, the *static semantics* describes the meaning of the elements in the meta-model; such as constraints and rules. In this example, the static semantics defines that attributes maintain the state of a class and the methods provide the class behaviour.

The model is usually platform independent (Platform independent model- PIM) rather than specific for a given platform (Platform specific Model - PSM). The domain specialist defines the meta-model based on the high-level structure and behaviour of the system and hides technological details about both implementation and behaviour. This decoupling between technologies and semantics in the model is explored at runtime for controlling complex systems. Complex systems such as μApp (SAMPAIO JR. et al., 2017) and Cyber-physical systems (CPS) (SAMPAIO JR.; COSTA; CLARKE, 2013) involve several technologies and the model is used to abstract them and expose only the elements of interest for controlling the application.

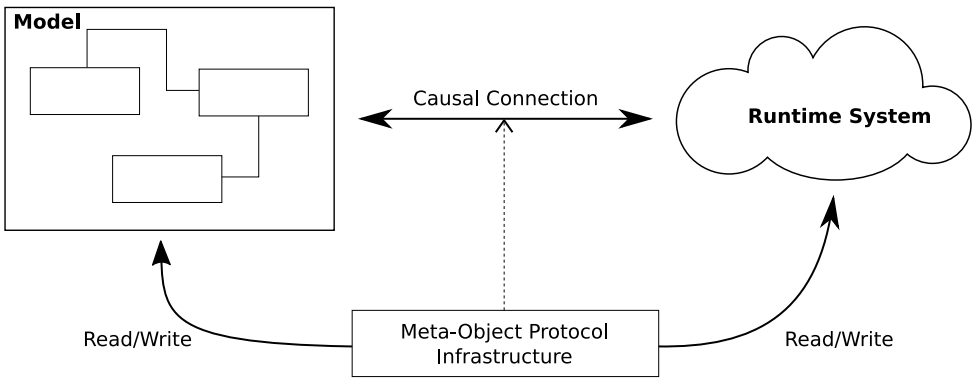
When a model is used at runtime (Models@Run.time) (BLAIR; BENCOMO; FRANCE, 2009), it is causally connected (MAES, 1987) with its underlying system as shown in FIG. 9. Hence, the model reflects the systems state (structure and behaviour), making changes to the system and vice-versa whenever the model changes. This feature facilitates the adaptation process of μApps since it is not necessary to deal with management tool interfaces. 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 by (SZVETITS; ZDUN, 2013) use models at runtime as the primary element for runtime adaptation in complex systems.

The causal connection between model and system, as well as the formal definition of a model by its meta-model, facilitates the autonomous control and management of the systems. The well-defined syntax and semantics of meta-models allow that automatic mechanism to analyze and plan changes in the system without human intervention and without dealing with technical details. Moreover, the causal connection makes data gathering and executed actions transparent for management systems. The model serves as a facade for the system providing reading and writing interfaces manipulate the system at runtime.

The manipulation of the system through its model is achieved by using the meta-object protocol (MOP) (KICZALES; RIVIERES; BOBROW, 1991), originally designed to be an object-oriented interface for modifying languages at runtime. However, this concept was expanded by MDE, so that MOP is now used to alter a model at runtime as well. As shown in FIG. 9, MOP links all elements in the model with the concrete elements of the system and maps the model (PIM) into platform-specific constructions at runtime. It synchronizes the model elements with concrete elements of the system, abstracting technological details and hiding the challenges in order to gather data and set changes from/to underlying system.

Figure 9 – Causal connection.



2.3 ADAPTATION AND MODELS@RUN.TIME

Oreizy, Medvidovic and Taylor (2008) and Vogel and Giese (2010) claim the need for runtime models to facilitate system adaptation. Runtime models abstract complex and heterogeneous systems providing adaptation managers with a unified view. Adaptation managers have more flexibility therefore to compute adaptation plans and to apply the adaptation on the managed system.

Szvetits and Zdun (2013) and Krupitzer et al. (2014) surveyed several strategies for adapting complex, heterogeneous, 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 (Models@run.time) (BLAIR; BENCOMO; FRANCE, 2009).

Models@run.time abstracts the system, simplifies the adaptation process and helps handle the diversity of underlying technologies by providing a management interface for system. This interface comes up due to a feature in the model that exposes and combines data gathered from sensors – as well as actions provided by the actuators – onto a unified interface, thus allowing analyzers to read system's state and executors to change it.

Moreover, Models@run.time has a causal connection with the managed system in a way that changes to the application are reflected in the model and vice-versa (MAES, 1987). 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.

2.4 MICROSERVICES

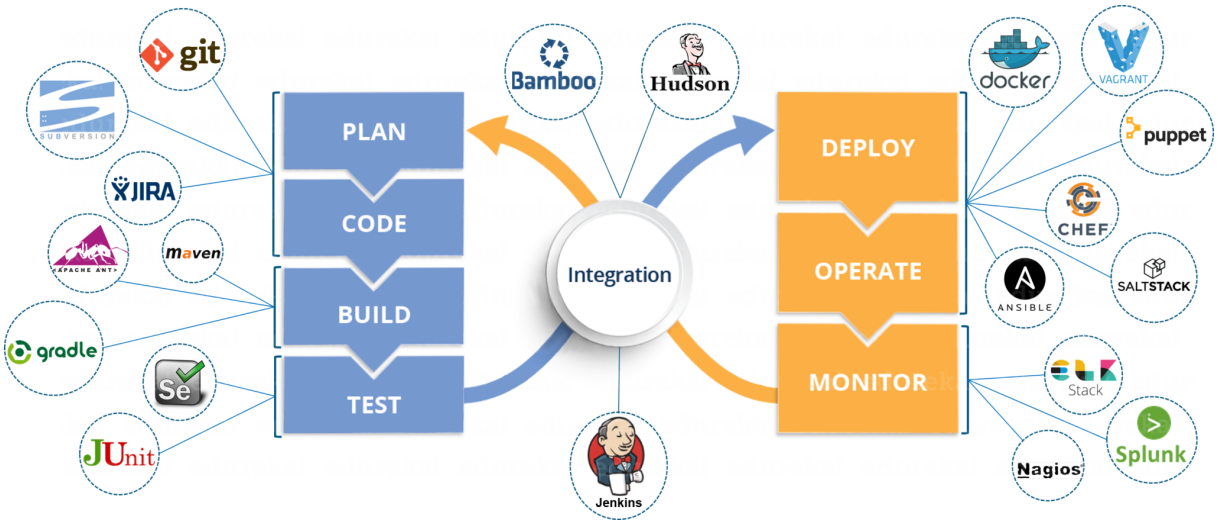
Microservice is an architectural style that promotes a *high-decoupling* of components to facilitate the evolution of applications. Its usage is a consequence of the evolution of agile methods (e.g. DevOps) and technologies used to develop and deliver software (e.g. containers). Despite the similar name (microservices and services) from traditional SOA (OASIS, 2006), there are several differences between them – mainly in regards to their runtime behaviour and evolution. Management tools are used to control the evolution of microservice-based applications (μApps) through scaling in/out and rolling out/back microservices as well as dealing with the placement of microservices into the cluster. Next, we better describe the concepts behind microservices.

2.4.1 DevOps

DevOps – Development and Operations – is a method that puts together the development and operational phases of a company, usually handled by two different people or departments (BALALAIE; HEYDARNOORI; JAMSHIDI, 2016). However, DevOps aims to improve the efficiency of software development by blurring the lines between these phases. In summary, DevOps is the practice of bringing agility and optimization by unifying into a team all phases of the software development and maintenance.

According to DevOps (BALALAIE et al., 2018), a single group of engineers is responsible for gathering requirements to develop, test, make deployments,monitor and gather feedback from the customers and the software itself. DevOps relies on continuous integration and monitoring, resulting in frequent changes to the software. Therefore, *high-decoupled* software is the key to apply DevOps.

Figure 10 – DevOps operations and some examples of the technologies used in each operation.



Source: <www.edureka.co>

To correctly apply the DevOps, it is necessary to automatize the life-cycle phases, as shown in FIG. 10. Test, integration, deployment and monitoring phases are carried out automatically, allowing engineers to focus on planning and coding the application. To achieve this high automation degree, it is necessary continually monitor several aspects of the software. All these data are used as feedback and tools, allowing the engineers to evolve the application fast and continuously.

However, this in-depth monitoring has a cost. The heterogeneity and amount of data collected eventually overwhelms the DevOps engineer, making it difficult to analyze all the information generated and plan an optimal strategy to act in response.

2.4.2 Containers

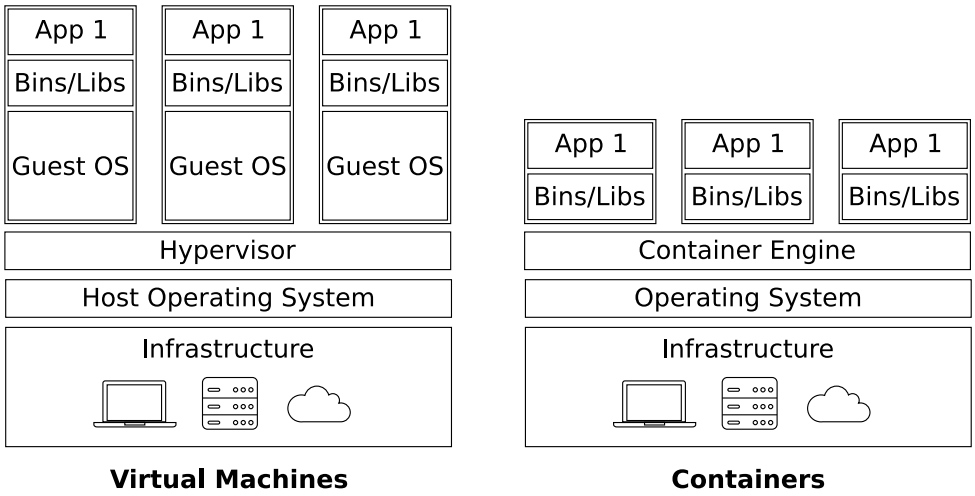
A container is an executable operating system-level abstraction that works as an alternative to virtual-machines (VMs). It packages the code and dependencies together, as shown in FIG. 11. Multiple containers can run on the same machine sharing the OS kernel and file system, each running as an isolated process in the user space. Different from virtual machines, containers take up less space and resources (e.g., RAM), and start to run almost instantly (BERNSTEIN, 2014).

The long time for provisioning VMs is a constraint to scale-out applications that use this deployment technology. This fact leads developers to create monolithic applications deployed into single VMs. Hence, when the application needs to scale, the developer scales up the resources of the VM, such as CPU and memory, instead of scaling-out more replicas of the application or its components. Hence, this approach leads to a waste of resources

on behalf of the applications.

In general, when an application needs to scale, it happens due to a bottleneck in some components and not in the whole application. When developers scale up a VM, all components of the application get extra resources to use, even if they do not need them. Hence, there is an additional cost to scale up a VM with more resources than necessary. The monolithic architecture of the application avoids the scale up/out of individual parts of the application.

Figure 11 – Virtual machines versus Containers



Moreover, the monolithic architecture imposes that upgrades should replace the whole application. Since there are several components involved in upgrading the application, component developers have to agree about the changes to be applied in the application such as: which technologies to adopt and when to apply the changes. Hence, upgrading the application becomes costly and can take a long time to be applied.

As an alternative to deal with these challenges, developers have begun to replace VMs with containers. The small footprint of containers makes their provisioning faster than VMs (SEO et al., 2014; FELTER et al., 2015) and facilitates scaling out the application. Therefore, developers started to split applications and deploying their components one per container into a cluster. This characteristic allows scaling out and upgrading each component frequently and individually – avoiding the misuse of resources, time, and money related to monolithic deployment into VMs.

2.4.3 Services and Microservices

The need for a high decoupled application to better apply DevOps as well as the popular use of containers originated a new wave in software development named *microservices*.

A microservice is an autonomous piece of software with well-defined boundaries used to build up applications. In microservice, the prefix “micro” is not directly related to its size, but to the number of operations it provides; a microservice should have a single

responsibility. A microservices-based application (μApp), shown in FIG. 12, uses many microservices; (dozen[1], hundreds[2], or more[3]) integrated over a lightweight language agnostic protocol to build up a complex system (LEWIS; FOWLER, 2014). Such a large number of distributed components usually makes the management and maintenance of a μApp difficult. A μApp is usually built up by several multi-lingual microservices such as Java, Go and Node. JS the most common, integrated through lightweights protocols like HTTP.

The high decoupling of microservices and small container footprints allow parts of the μApp to scale in/out on demand, which improves the resource usage of the application in a cluster. Moreover, this approach enables to split the development of applications in small teams, each one responsible for a single microservice. Hence, upgrades of μApp do not need to change the whole application, resulting in fast and frequent changes if needed. These characteristics made the microservice architectural style a standard for building continuously deployed systems.

The microservice definition is very similar to a service in service-oriented paradigm (PAPAZOGLOU, 2003). In fact, microservices and services are conceptually the same and are built up by grouping several microservices/services. However, microservices and services have several practical differences, such as: their development, execution, and maintenance.

Usually, services are big pieces of software, a.k.a. Monoliths, that group several tasks related to one activity of the application. An activity is a use case of the application, and the tasks are the operations performed by different methods to achieve the use case objective. For example, in e-commerce, paying using a credit card is an application activity provided by a service. This activity is built up by several tasks, such as: checking the existence of the customer, checking the credit card balance, authorizing payment and notify the bank of the order.

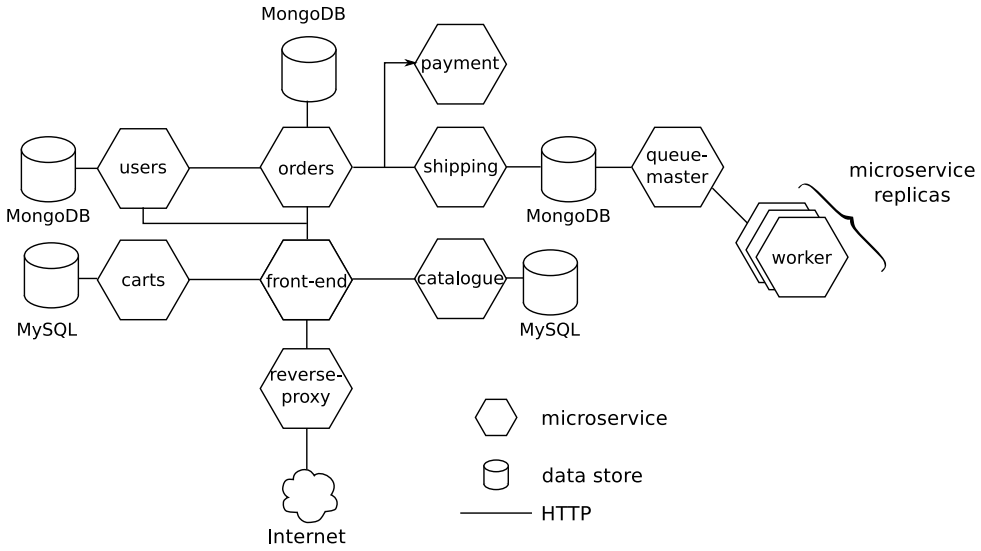
Moreover, in many cases, the service is public and developed to perform an activity generic enough to outsource it to other companies. However, when the company needs a particular activity, it creates a private service.

Developers build up the application by integrating several services, private or public, by using coordination languages such as WS-BPEL (JORDAN; EVDEMON, 2007), that orders the workflow of the application (BICHLER; LIN, 2006). Hence, developers do not need to deal with the code of the application – only its high-level specification. For example, if a service needs to be altered, the developer needs to change the address of the current service while the language interpreter performs the steps necessary to use the new service from said point.

The seamless integration of services is possible due to the use of enterprise service bus (ESB) (PAPAZOGLOU, 2003). ESB provides several mechanisms to integrate the services of an application, such as service registration and discovery; messages routing; business

[1] <https://microservices-demo.github.io/>
[2] <http://highscalability.com/amazon-architecture>
[3] <http://tecnoblog.netflix.com>

Figure 12 – Example of μApp.



Source: <https://microservices-demo.github.io/>

rules; messages filtering; service fail-over; message transformations; security and many others. This approach forces the application to rely on a single component (single point of failure) in the application architecture which can threaten the application execution if it is not working correctly. However, service engineers are free on not to concern about several non-functional requirements during a service development.

On the other hand, microservices are small pieces of software that provide single and well-defined functionalities. Unlike services, a microservice does not implement an activity by itself, but rather a group of microservices each provide a task. Given the credit card example mentioned before, the activity of processing a payment is made by several microservices each one providing a single task.

The integration of microservices relies on the concept of *smart endpoints and dump pipes* (LEWIS; FOWLER, 2014). The microservice (smart endpoints) receives a request and applies a logic as an appropriate answer to produce a response. To reach it, the application uses simple built-in RESTis protocols rather than WS-BPEL. The microservice itself is responsible for choreographing the μApp's behaviour. The integration of microservices is achieved over a lightweight message bus (dump pipes) – RabbitMQ[4] or ZeroMQ[5] – that does not do much more than provide a reliable mesh connecting the endpoints by routing raw messages without any manipulation.

Moreover, microservices have some degree of autonomy and are implemented by knowing how to locate and communicate with other microservices. This characteristic means that changes in the μApp's workflow are made by replacing microservices with new versions. In reality, the low coupling between microservices allows them to be changed without

[4] <https://www.rabbitmq.com/>
[5] <http://zeromq.org/>

affecting others.

Ideally, microservices should be stateless. Stateful microservices decrease the flexibility of μApps, locking their placement to data location. Furthermore, microservice management tools do not have mechanisms to deal with data migration and data replication when scaling in/out microservices or new microservice version deployment. To avoid data locking, μApps regularly use data stores provided by cloud providers, such as Amazon Simple Storage Service[6], instead of managing their data store.

2.4.4 Microservices Management Tools

The high coupling between microservices and containers has led to the use of single management tools to control container's life-cycle and microservices. Cloud Foundry[7], Docker Swarm[8] and Kubernetes[9] are widely used to manage containers, and consequently microservices (BERNSTEIN, 2014).

The management tools do not handle microservices directly, in fact the microservices must be wrapped into containers or other technology specific abstraction, e.g, Pods in Kubernetes. Along to this document we assume that all microservices are individually wrapped as required by the management tool, so that microservices and containers (or Pods) can be used interchangeable in this document. It is worth observing that Kubernetes is by far the most popular, being the management tool made available by cloud providers such as Google, Microsoft and Amazon.

Management tools are responsible for dealing with auto-scaling and deployment of microservices, as well as providing registering and discovering services and some security to μApp developers. Management tools collect instantaneous information off microservice execution – resource usage and microservices logs (BERNSTEIN, 2014). However, behavioural information such as message exchanges and resource history is not provided by management tools. Despite some automation, application developers are responsible for microservice management, such as defining the number of replicas and triggering scale in/out actions.

When an engineer configures a microservice to be deployed, some parameters used by the management tool at runtime are set. In general, engineers set the max and min amount of resources required by the microservice, max and min number of replicas available at runtime, and thresholds used to trigger scaling in/out and to request/release host resources. At runtime, management tools collect instantaneous snapshots of resource usage and compare them with the values set by engineers,scaling the microservice or requesting/releasing resources.

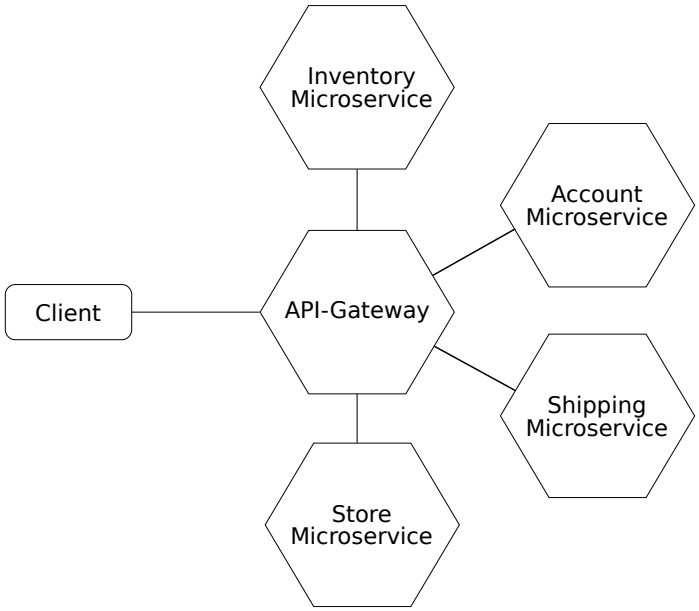
[6] <https://aws.amazon.com/documentation/s3/>
[7] <https://www.cloudfoundry.org/>
[8] <https://docs.docker.com/swarm/overview/>
[9] <https://kubernetes.io>

In addition to the mentioned limits and thresholds, management tools allow microservices to be tagged. These tags can be used at runtime to map microservices to specific hosts according to annotations in the microservice and hosts. Moreover, management tools provide these tags at runtime through API for third-party tools to use.

2.4.5 μApps Architectures

μApps may be integrated by using different integration styles, API-Gateway FIG. 13 and Point-to-Point FIG. 14 are the most recommended (NEWMAN, 2015). The FIG. 13 and FIG. 14 do not represent any real application, they are just examples of how would be a topology of an μApp if it uses one of the integration styles mentioned before.

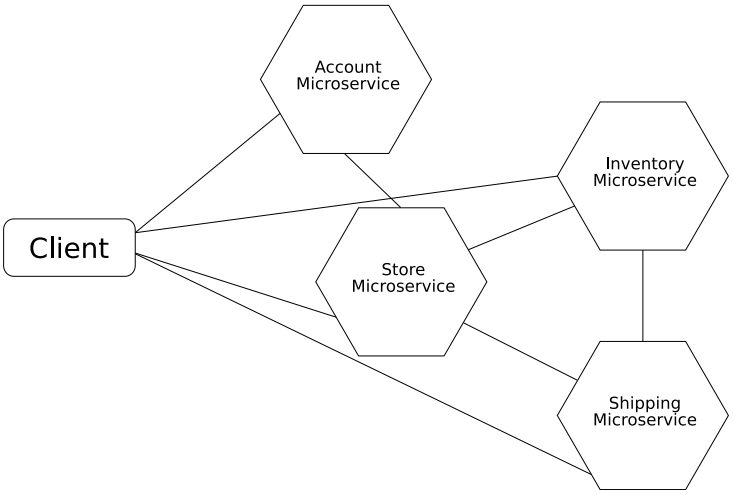
Figure 13 – API-Gateway Example.



The API-Gateway provides a single, unified API entry point across one or more internal APIs. The microservice that implements the API-Gateway typically implements some reliability patterns such as circuit breaker and retry with exponential back-off, and security mechanisms, like authentication, as well. The API-Gateway prevents internal concerns from being exposed to external clients and adds a layer of security to microservices. However, the main drawback of API-Gateway is that it can become a limiting factor and even a single point of failure of a μApp.

The Point-to-Point style relies on the use of messaging middleware. In this style there is not a unified entry point, leaving each microservice to expose and deal with security and message throttling by itself. However, the reliability of delivering messages relies on the messaging middleware, such as RabbitMQ, that handles undelivered messages. A drawback of using Point-to-Point style is the duplication of common functionalities across the microservices making the μApp implementation more complicated.

Figure 14 – Point-to-Point Example.



2.4.6 Microservices Placement

The 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, which is not an easy task.

The deployment of μApps in a cluster must take into account the required resources defined by engineers as well as the 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. Engineers usually set these values based either on previous executions of the microservice or their own experience, which is subjective; making it difficult to establish what resources a microservice may need at runtime to work well.

Moreover, engineers usually do not set a maximum resource usage and the placement is guided only by the minimum. This unbounded approach leads management tools to poor μApp deployments, which causes a negative impact on the application performance and causes a waste of resources.

Another consequence of only setting the minimum quantity of resources is the placement of many microservices together into a single host. Co-located microservices, however, can start to demand more resources than available on the host. This competition leads the μApp to contention, dropping performance. Meanwhile, microservices configured with minimum resource requirements drive management tools to deploy a μApp across many hosts, potentially wasting resources and jeopardizing their performance due to network latency imposed on their communication.

Existing management tools implement several common placement strategies, 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 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, avoiding 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 tagged 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.

Finally, most of management tools provide abstractions to force microservices to be co-located during the deployment. However, this abstractions are static, they cannot change automatically at runtime according to variations of the μApp behavior. This co-locations should be set by the engineers that, not necessarily, have all information needed to set them usefully.

2.5 CONCLUDING REMARKS

In this chapter, we initially presented the central concepts of autonomous computing and adaptive systems focusing on MAPE-K architecture. We showed the consequences of using different deployment configuration to deploy MAPE-K. We also presented the notion of model and Models@run.time, and how Models@run.time can be used on adaptive systems. Finally, we presented the primary motivations behind microservices along with a discussion about the differences to the traditional idea of service in service-oriented computing.

3 REMAP - RATIONALE AND GENERAL OVERVIEW

In this chapter, we present the challenges of evolving μApps at runtime and highlight the critical challenges of adapting μApps at runtime. We highlight the monitoring and placement of microservices on a cluster as the critical challenges to be handled in adapting μApps. Next, we present an overview of our proposal to perform runtime adaptation on μApps. Moreover, we describe the concept of model evolution and how it is used to guide the adaptation of μApps. Finally, we characterize the problem microservice placement at runtime and how it affects the μApps.

3.1 CHALLENGES ON RUNTIME EVOLUTION OF μAPPS

The decoupling promoted by the Microservice architectural style facilitates applying changes to the μApp– allowing engineers to replace or scale microservices without collateral effects on the application structure. Engineers do not need to halt the application to apply change. Therefore, we define μApps as an inherited adaptive application.

However, high decoupling in μApps is a two-sided sword for their maintenance. On one hand, decoupling allows engineers fixing the μApp or adopt new technologies smoothly. On the other hand, this ease in changing microservices may threat the application's behaviour if applied in a careless fashion – when engineers take into consideration the past behaviour of the μApp (e.g., its workflow or resource usage), the adaptation may lead the to a poor performance on behalf of the μApp.

Changes on μApps rely basically on Microservice placement. Hence, in order to decide *when* and *why* to change the μApp, it is necessary to observe different aspects of the application. However, due to the heterogeneity of μApps, monitoring is not a simple task. Furthermore, the dynamics of a μApp can affect or be affected according to where the microservices are placed. Next, we expand the discussion of the challenges in monitoring and placing microservices at runtime.

3.1.1 Challenges in Monitoring

Despite the microservice style facilitating *how* the changes in μApps occur, it makes it difficult to identify *what* and *when* it is necessary to adapt the application. Hence, it is essential to observe the behaviour of the μApp.

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

Resource usage The resource usage is the amount of resource used by a microservice while the μApp executes – CPU, memory, disk and network bandwidth.

Application logs μApp engineers determine application events to signal errors, warnings and informational messages throughout its execution. Informational messages do not log warnings or errors, but it informs what is happening throughout the application execution (e.g."[INFO] loading library xxx"). Dedicated tools maintain the events in the sequence they occur and allow the engineers to use the application logs for tracking the μApp execution.

Message information Messages exchanged by microservices, including: message sources and destinations; payload size; some metrics, such as response time; and contextual data, such as HTTP headers and response codes.

Due to the large number of microservices in a μApp– different types of microservices and their replicas building up the μApp– its execution generates a huge amount of heterogeneous data, which makes it challenging to monitor the μApp. To make matters worse, the diversity of technologies used in the microservice domain makes it difficult to get a unified view of the μApp at runtime.

A μApp is potentially multi-lingual and multi-technological, which means that various tools are necessary to monitor the same information across different microservices in the μApp. 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 of standard creates a semantic gap in the provided information. For example, data metrics collected by Influxdb[1] and Prometheus[2] have different formats. Therefore, it is difficult to observe the dynamics of μApps running on the same cluster, since each one can use a different monitoring stack.

Furthermore, not all languages include bindings for a specific tool, e.g., Zipkin[3], which means that different tools may monitor microservices belonging to the same μApp. The heterogeneity of monitoring tools is a challenge as there is a need to deal with different semantics, data structures and technologies.

Although monitoring tools can collect information from a μApp execution (e.g., instantaneous resource usage), they are unable to collect behavioural aspects such as resource usage history and μApp workflow. Instantaneous data alone 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 historical data of μApps, since management tools only expose instantaneous microservice information. Some existing tools are now being used for this purpose: cAdvisor[4] gathers cluster (hosts) and microservices

[1] <https://www.influxdata.com/>
[2] <https://prometheus.io/>
[3] <https://zipkin.io>
[4] <https://github.com/google/cadvisor>

(wrapped into containers) metrics natively; Prometheus stores data collected by cAdvisor or self-stored by microservices; and Influxdb stores monitored data.

Moreover, 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, such as Zipkin[5] and Jaeger[6].

Despite the fact that management tools expose microservices' execution logs, these tools cannot aggregate and use them at runtime. As a result, aggregators are needed in order to organize logs as well as ensure their temporal order and store them to maintain their history. Popular log aggregators include Fluentd[7] and Logstash[8]. It is also necessary to use data stores like Elasticsearch[9] and Influxdb, to maintain a history of the μApp execution. Furthermore, cloud providers habitually provide their own private solutions such as Amazon CloudWatch[10].

Hence, to monitor μApps and track their behaviour at runtime, it is necessary to use several tools to gather signals of different aspects of μApp behaviours such as the microservices' resource usage and data workflow.

By having all 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 use this information as input for an artificial intelligence mechanism which anticipates future behavior based on past information and applies preemptive adaptations on the system.

Engineers are currently manually analyzing the collected data – retrieving, parsing and sending them to visualization tools (e.g., Kibana[11]) – and taking action based on what is observed. These steps make the management of μApps complex and error-prone (WARD; BARKER, 2014).

As a consequence, the use of autonomous tools is made necessary in order to improve μApp adaptation, releasing engineers from decision making, makes the whole process faster and more reliable. However, the challenges in adapting μApps are not restricted to monitoring them – it relies on placing microservices somewhere. Next, we discuss the details of this challenge.

3.1.2 Challenges in Placement

The high decoupling provided by microservices facilitates the scaling and upgrading of μApps, promoting their evolution. In our context, the evolution of μApps consists of its

[5] <https://zipkin.io>
[6] <https://uber.github.io/jaeger>
[7] <https://www.fluentd.org>
[8] <https://www.elastic.co/products/logstash>
[9] <https://www.elastic.co/products/elasticsearch>
[10] <https://aws.amazon.com/cloudwatch>
[11] <https://www.elastic.com/products/kibana>

adaptation by (1) replacing one microservice instance with another, usually on a different host, or (2) creating new microservice replicas. Management tools execute such adaptations. Hence, we classify μApp adaptation in two types:

1. *Updates*: The microservice implementation do not change, only their instances are reconfigured, e.g., scaling in/out;

2. *Upgrades*: The microservice implementation and/or application structure change, e.g., rolling out/back a microservice version.

The evolution of μApps is a natural activity in the microservices domain, and as mentioned in Section 2.4, it is the only way to alter the workflow of the μApps. Hence, engineers use management tools to carry out the adaptation.

These tools are currently unaware of microservice runtime behaviour, and the decision to place a microservice somewhere is based exclusively on static values set by engineers. In most cases, these values do not reflect the real behaviour of the μApp, resulting in a poor placement.

For example, a tool can automatically trigger the auto-scaling, but an engineer must fix both the maximum number of replicas as well as the resource usage that triggers the scaling. In an autonomic approach, the adaptation mechanism would automatically decide these parameters.

In another example, a new version of microservice "A" has a higher communication demand with microservice "B" than in its prior version. The management tool is unaware of μApp communication requirements – it knows nothing about messages exchanged between microservices. Hence, it put these microservices in different places. However, the high communication between the two services over the network can hurt the overall performance due to network latency. In this case, it would be better to co-locate these both microservices in the same host.

The evolution of μApps eventually changes the requirements of their microservices. For example, the upgrade of the μApp either makes some microservices require more or fewer resources at runtime, or breaks or adds relationships between microservices which change the workflow of the application.

The management tool, which controls the microservices inside the cluster, should decide where to place the microservices as a consequence of the adaptation. Different management tools each have their strategy to place microservices but none of them consider runtime and history data to make a placement decision. Due to the amount of data generated by the μApps and its heterogeneity, the management tools discard the past resource usage and not considers the microservices interactions during its execution. Usually, these tools only consider the current resource usage in order to decide where to place the microservice after the adaptation.

Nevertheless, the ease of change in microservices can threaten the μApp. The careless placement of microservices can lead the application to get undesired behaviours such as resource contention or communication latency.

Resource contention arises when microservices with high resource usage are placed together into a cluster node. Due to the lack of information about past resource usage by the microservice, the management tool may place microservices with high resource usages together. These tools are only aware of the instantaneous state of the applications; therefore it cannot infer any information about past microservice behaviour. Two microservices could, for example, have a high resource usage in the morning but low one in the afternoon. If an adaptation is made in the afternoon, the management tool may place both together causing the μApp to drop its performance the next morning due to both microservices competing for resources in their host.

Similarly, the lack of information about the workflow of a μApp endangers its execution. Management tools are not aware of the μApp's workflow – all the messages exchanged by microservices are transparent to the tool. Hence, tools can place high related microservices separated into different hosts, which lowers the μApp's performance due to the network latency between the microservices.

However, dealing with placement is not easy. Placing a microservice in a cluster based on different constraints is an NP-Hard optimization problem. This characteristic makes it difficult (impossible in some cases) to optimize μApp placement at runtime. It is not easy to have a complete view of the μApps in order to decide when to change the application. Therefore, in this thesis, we deal with monitoring and μApp placement problems by proposing REMaP (**R**untim**E** **M**icroservices **P**lacement), an adaptation mechanism to improve the placement of μApps based on its runtime behaviour, past resource usage and its data workflow.

Management tools should be aware of runtime information in μApps to avoid the aforementioned drawbacks, however they are not. Therefore, we propose the use of REMaP, which uses runtime data to improve microservice placement. The use of runtime data can make better μApp placement by providing information about different features, such as the actual resource usage and message exchange.

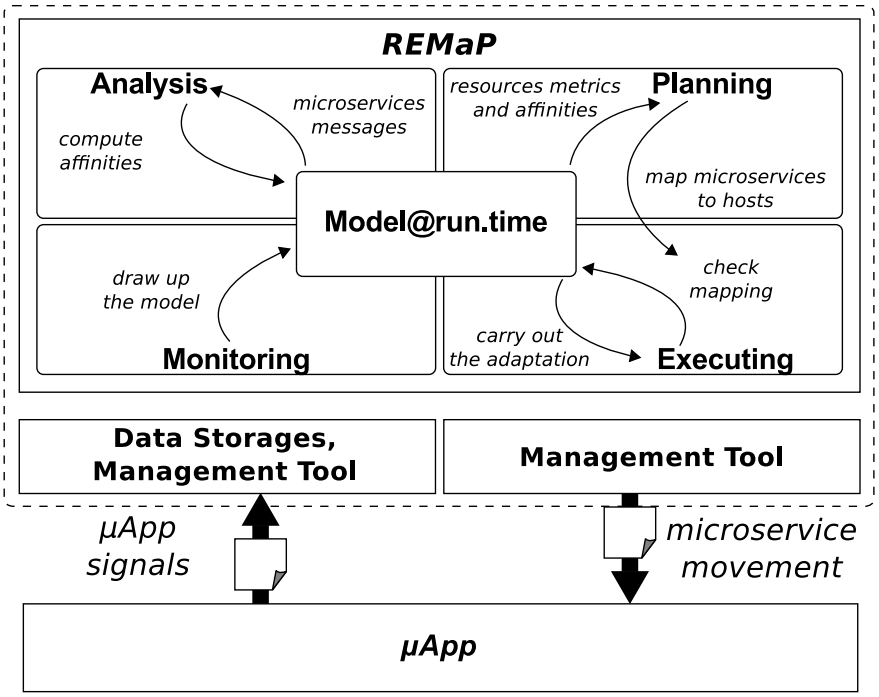
Next sections present an overview of REMaP, the use of models to support μApps evolution, and how to deal with microservices placement at runtime.

3.2 PROPOSED SOLUTION - OVERVIEW

REMaP (**R**untim**E** **M**icroservices **P**lacement) is an autonomous mechanism for adapting μApps in a cluster at runtime whose architecture, shown in FIG. 15, is based on MAPE-K and was designed to handle seveal μApps in a cluster. REMaP automatizes the adaptation of μApps and improves their placement. The proposed solution uses Models@run.time

as the knowledge source. REMaP uses Models@run.time to unify the heterogeneity of technologies used to monitoring μApps and to guide the adaptation, validating changes before applying them and sharing data and partial computations across the different components within REMaP.

Figure 15 – REMaP's conceptual architecture.



The application management in general relies on two steps that repeat indefinitely in a control-loop: (i) to watch for application signals; and, (ii) to act in response to what is sensed. The outcome of these activities is the adaptation of the managed application – the application changes along its execution.

In the **monitoring activity**, REMaP gets a complete view of the μApp by unifying into the model different data, such as resource metrics and exchanged messages. REMaP uses an evolution model to unify the heterogeneous data generated by the μApp's execution. This model provides a well-defined structure and semantics of the collected data, abstracting technological specificities existing in the Microservice domain.

In the **analysis activity**, REMaP analyzes the model by detecting affinities between microservices based on the messages exchanged, and keeps them in the model. REMaP uses the μApp behaviour to guide the placement of microservices at runtime. We observed that a careless placement of microservices, without considering its behaviour (workflow), jeopardizes the performance of the whole μApp. Hence, REMaP uses microservice affinities to steer where to place microservices during the adaptation considering its workflow.

In the **planning activity**, REMaP plans the adaptation by selecting microservices with high affinity and maps them to a host. REMaP aims to improve the placement

of μApps by co-locating high affinity microservices together into a host, respecting the microservices resource usage as well as the resources available in the host. REMaP's strategy uses the μApp's behaviour (microservice workflow and usage history resources) to compute and optimize the placement of microservices. Due to the complexity of optimizing large μApps at runtime using optimization algorithms, REMaP uses a heuristic to improve the placement of large μApps timely.

In the **execution activity**, REMaP uses Models@run.time to place the microservices in the new locations safely. During the μApp's execution, external factors, such as variation on the number of requests from clients, might result in a reconfiguration of the μApp by scaling in/out their microservices or scaling up/down microservices' hosts. Hence, REMaP's execution component use the model to check the current state of the μApp before applying the adaptation.

REMaP use models during the adaptation process. As mentioned in Section 3.1, the heterogeneity and complexity of μApps is a challenge in the evolution of μApps. The next section describes our vision for supporting μApp evolution by using models.

3.3 EVOLUTION MODEL

Understanding a single microservice may be straightforward, but μApps often contain dozens of inter-dependent microservices that continuously change. Monitoring and logging stacks for microservices like ELK-stack[12], is essential to understanding the microservice execution in a μApp, despite the critics from Section 3.1. M Unfortunately, logs produced by such stacks contain low-level information for a *single* deployment. Reconciling the view of the deployed version of the system with the historical view of changes being introduced requires interpretation by the engineer.

For example, a log may record a failing REST invocation against a particular URL but it is up to the engineer to determine if this invocation was introduced in a recent change and requires fixing or if it indicates an undesirable dependency that should be eliminated. Furthermore, non-trivial tasks require piecing together logged information from multiple sources such as multiple systems logs, container infrastructure data, real-time communication messages and so on. Collecting and analyzing such information in the context of an evolving system relies on non-trivial knowledge and effort.

We identified several evolution-related maintenance tasks that are challenging for microservice engineers. Supporting these and similar tasks is the focus of our work.

Next, we overview the tasks and briefly outline the challenges they entail.

- **Evaluating changing deployment trade-offs.** Microservices offer extensive deployment flexibility. For example, two microservices can be co-located as two con-

[12] <https://logz.io/learn/complete-guide-elk-stack/>

tainers on the same physical machine, as two containers in one VM or as two VMs on the same physical machine. A poor deployment choice can increase cost and hurt performance, scalability and fault tolerance. Furthermore, these decisions must be re-evaluated as the μApp evolves. Developers nowadays change the deployment by trial and error without a systematic strategy and having a poor tool support.

- **Checking for upgrade consistency.** Microservices are developed and evolve independently, yet the μApp must remain coherent and functional. Determining compatibility and consistency between microservice versions is a continuous challenge for developers. Now, developers manually identify microservice dependencies by evaluating dependency through code inspection.

- **Identifying architectural improvements.** An evolving μApp will experience software architectural corrosion, such as a decrease in cohesion and increase in coupling between related services. Nowadays, detecting such architectural problems and evolving microservice architectures are manual processes that require complete knowledge of microservice inter-dependencies.

We propose an approach for combining structural, deployment, and runtime information about evolving microservices in one coherent space, which we refer to as *microservice evolution model* (SAMPAIO JR. et al., 2017). By aggregating and analyzing information in the model, we aim to provide insights about the system, assisting μApp developers with maintenance and evolution tasks.

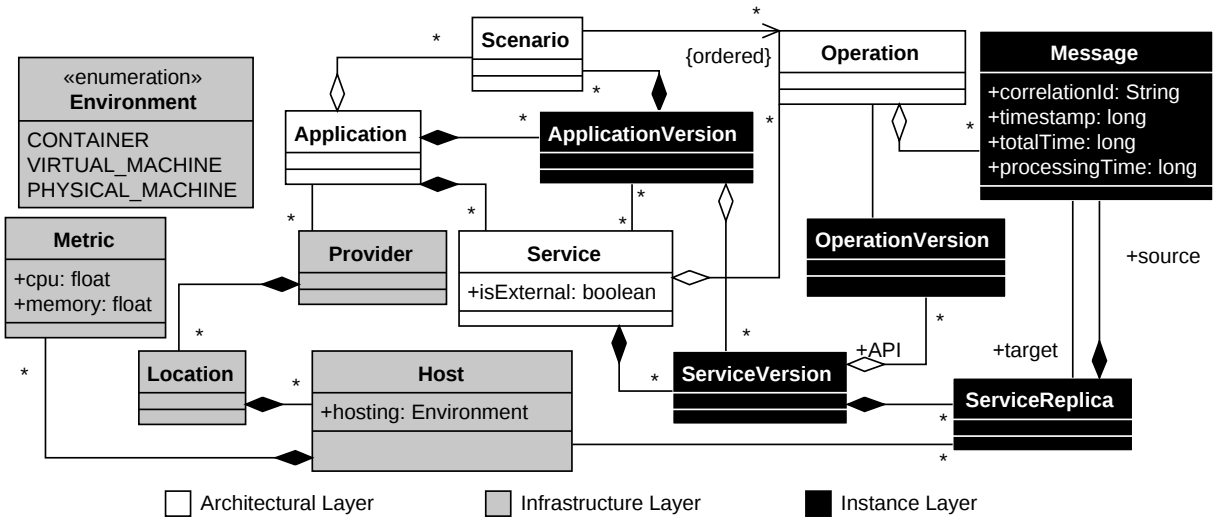
3.3.1 The Model

We propose a model for microservices and their evolution in FIG. 16 inspired in the object model from Kubernetes[13] and Docker Swarm[14]. This model is divided into three layers: the *Architectural* layer (unshaded elements) captures the topology of a μApp; The *Instance* layer (black elements) captures information about service replicas and upgrades and the flow of messages of the μApp– this layer links the topology outlined in the *Architectural* layer with deployed microservice instances; and the *Infrastructure* layer (gray elements) captures deployment parameters. These three layers together hide different technologies and provide a global view of the application, organizing different aspects such as resources usage and data workflow.

Next, we describe how the model hides the complexity of a μApp by unifying different aspects of the application into a single artifact. To make description easier, we will use a simple μApp.

[13] <https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/>
[14] <https://docs.docker.com/engine/api/latest/>

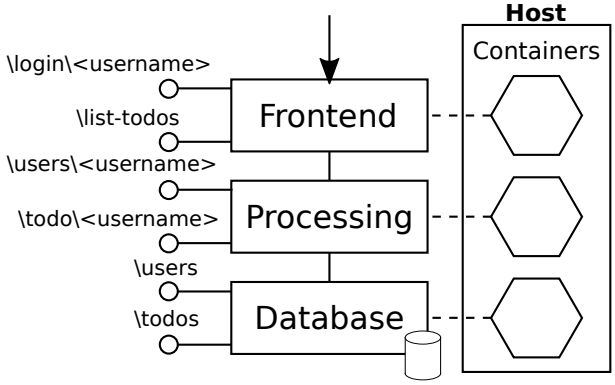
Figure 16 – Service evolution model.



Source: (SAMPAIO JR. et al., 2017)

As our running example, we use a simplified version of an open-source *ToDo* μApp application[15] as shown in FIG. 17, which consists of three microservices: *Frontend*, *Processing* and *Database*. Each microservice is deployed in its own container. *Frontend* allows new users to log in (via the `\login\<username>` operation) and, for already logged-in users, to retrieve the list of their to-do items (`\list-todos\<username>`). *Frontend* communicates with the *Processing* microservice to obtain information about a specific user (`\users\<username>`) and retrieve all to-do lists of a specific user from the database (`\todos\<username>`). The database access is managed by the *Database* microservice that provides access to the list of all users (`\users`) and all to-do items (`\todos`).

Figure 17 – ToDo application architecture



Source: (SAMPAIO JR. et al., 2017)

Next section presents how to populate the model.

[15] <https://github.com/h4xr/todo>

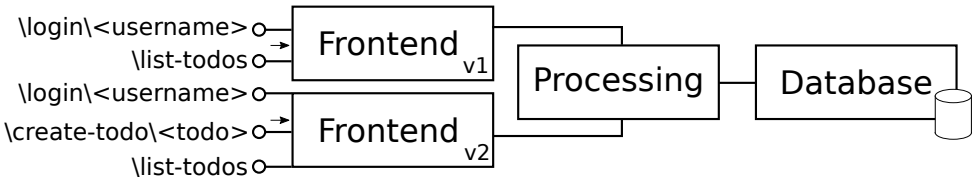
3.3.2 Populating the Model

A μApp is represented by element *Application* in FIG. 16, which consists of a set of *Services*, each one exposing a set of *Operations*. In the example shown in FIG. 17, the *Frontend* service exposes two operations: `\login\<username>` and `\list-todos\<username>`. *External* services are "black-box" services managed by third-party organizations and are marked using the *isExternal* flag.

A *Scenario* describes a high-level use case of the application and is carried out by an ordered list of operations executed by the microservices. In the ToDo application, such scenarios include users logging into the application and retrieving their to-dos. The login scenario is carried out by the *Frontend* `\login\<username>` operation followed by the *Processing* `\users\<username>` operation and *Database* `\users` operation. A scenario specifies the allowed order of operations, helping to detect faulty behaviours. For example, there is not a scenario in which the *Processing* `\todo\<username>` operation precedes the *Frontend* `\login\<username>` operation.

The *ServiceVersion* and *OperationVersion* elements keep track of changes in services and their interfaces. Any Microservice upgrade creates a new *ServiceVersion* element. In addition, if the upgrade involves an operation change, a new *OperationVersion* element is created and attached to the new *ServiceVersion* element. For example, adding the `\create-todo\<todo>` operation in the *Frontend* microservice, as shown in FIG. 18, will create new *ServiceVersion* and *Operation* instances.

Figure 18 – Two versions of *Frontend*, differing in their supported operations.



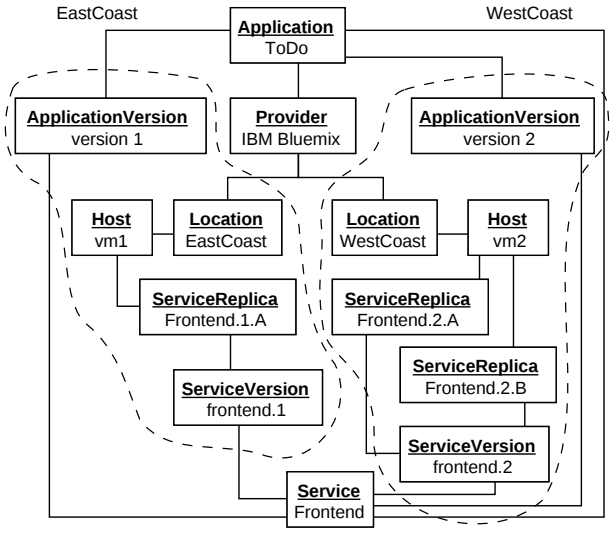
Source: (SAMPAIO JR. et al., 2017)

In Canary releases[16], multiple services and multiple versions of the same service can run in parallel to each other as part of the same application. The *ApplicationVersion* element, groups all service versions in a particular configuration. A sequence of *ApplicationVersions* captures the evolution of an application over time.

To model scale-in and out of services, multiple identical instances of a service version are represented by the *ServiceReplica* element. *ServiceReplicas* are *Hosted* by containers or by physical and virtual machines, depending on the *Environment* made available by the cloud *Provider*. Cloud providers such as Amazon AWS, Microsoft Azure, IBM Bluemix and Google Cloud Platform offer several hosting environments.

[16] <https://martinfowler.com/bliki/CanaryRelease.html>

Figure 19 – An example deployment of the *Frontend* service in FIG. 17.



Source: (SAMPAIO JR. et al., 2017)

Hosts can be deployed in multiple geographic *Locations*. FIG. 19 shows a fragment of the *Frontend* service deployment model of the ToDo application. In this example: version *Frontend.1* is hosted by VM_1 on the East Coast and runs one replica: *Frontend.1.A*; while Version *Frontend.2* is hosted by VM_2 on the West Coast and runs two replicas: *Frontend.2.A* and *Frontend.2.B*. All replicas, on both coasts, correspond to different versions of the *Frontend* service.

To optimize the deployment while the application evolves, we periodically monitor and store *Metrics* related to hosts' CPU load, memory utilization, required traffic and latency from a particular area, and so on.

A core element of our model is the *Message*. Each *Message* represents a uniquely-identified call, issued by the source microservice to a particular API (i.e., *Operation*) exposed by the destination microservice. In the example shown in FIG. 17, the *Processing* microservice exposes the `\users\<username>` operation, which can be called `\login<username>` *Frontend* microservice. Each *Message* carries the timestamp of the request, total time elapsed between issuing the request and obtaining the response, and the time spent in processing the request by each downstream microservice.

Messages running the same *Scenario* are grouped via a *correlationId*. For example, when both *User A* and *User B* log into the ToDo application, they execute the same login scenario, which involves the same sequence of messages but with different correlation ids: all messages corresponding to the User A login are correlated with each other and are distinct from those of User B.

In order to generate a model that covers all aspects of a μApp and provides a complete view of the application, it is necessary to gather data from different sources, having different semantics and formats. Furthermore, there is a need to homogenize them according

to the elements in the model.

For instance, we may generate the model by using information from system logs, container infrastructure data, and messages over protocols such as HTTP. More specifically, we can extract microservice information from hosts' meta-data and configuration files, like deployment files in Kubernetes[17]. To identify operations and their association with services, we may rely on a variety of sources – when available, we can extract information from API gateways combined with service discovery tools like Zuul[18]. We may also inspect documentation tools like Swagger[19], if developers published that information. We correlate and augment the extracted information by monitoring HTTP messages between services to reveal used operations.

We may generate message elements by using distributed tracing mechanisms like Zipkin[20]. We can use *correlationIds* in HTTP requests if developers follow the Correlation Identifier pattern (NEWMAN, 2015). In case this information is missing, an alternative is to implement dynamic information flow analysis techniques to correlate input messages with the outgoing requests they trigger.

Scenario is the abstraction of a μApp worflow. A scenario begins in the gateway (frontend) of the μApp and goes through several microservices, responding to the client. The scenario can be identified by grouping requests that have the same *correlationId*. In such case, each operation serving as the first point of contact for a user will generate a new scenario.

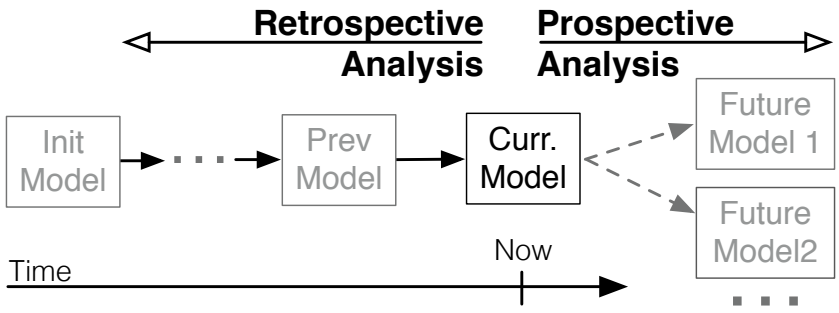
By querying cloud provider APIs, we may extract information about the properties of the provider, such as the data centre location, hosts, and so on. Interfaces provided by container orchestration systems, such as Kubernetes, can also be used to obtain notifications of new versions and recently created replicas. Performance metrics such as network throughput, CPU, memory and disk usage can be periodically collected using monitoring mechanisms such as cAdvisor[21].

3.3.3 Analysing the Model

The model captures information about an evolving μApp (FIG. 20) and may automate two kinds of analysis: the retrospective, considering current and past models; and prospective, considering current and future models. In both cases, the model can be used at runtime to steer and facilitate the process of adaptation.

[17] <https://kubernetes.io/>
[18] <https://github.com/Netflix/zuul>
[19] <http://swagger.io/>
[20] <http://zipkin.io>
[21] <https://github.com/google/cadvisor>

Figure 20 – Retrospective and prospective model analysis.



Source: (SAMPAIO JR. et al., 2017)

3.3.3.1 Retrospective Analysis

Retrospective analysis is the process the model uses for tracking previous states and behaviors of the application in order to decide what to do during the next μApp management. This analysis might be carried out by inspecting previous versions of the model, stored throughout the execution of the application, so that each version captures one change applied on the application.

For instance, individual microservices depend on each other to work – a change in a microservice might cause a failure in an entirely different dependent service. Comparing the sequence of messages in the faulty application workflow before a failure occurs, i.e., in the failing and the previous versions of the model, helps detect modified downstream service(s) involved in the workflow. Such services in the workflow are more likely to be responsible for the mistake and should be inspected first when *checking for upgrade inconsistencies*.

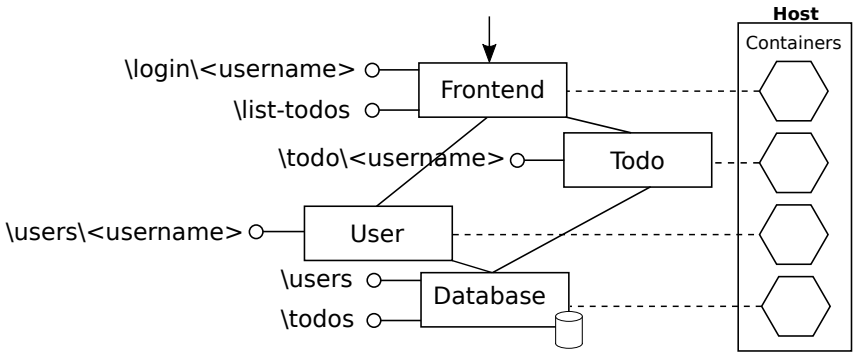
We can also use retrospective analysis to *recommend architectural improvements*. For example, we can use prior models to identify changes in μApp topology, i.e., their communication patterns. Changing coupling and cohesion of services can trigger topology re-organization, e.g., by merging interdependent services.

Another example is to use retrospective analysis to support splitting microservices. Splitting microservices whose operations exhibit different workloads ("imbalanced" microservices) can help to scale these operations more accurately. For example, in our ToDo application, once users log in, they create and modify numerous to-do items. In this case, the login endpoint is underutilized as compared to the endpoint that manages to-dos. Splitting this microservice into two separate entities, as shown in FIG. 21, makes it possible to scale up the *Todo* microservice while avoiding simultaneous scaling of the *Users* microservice.

The next example is the motivation of this thesis, and it uses retrospective analysis to suggest deployment improvements. By using retrospective analysis, we use the metric

history stored in the model, correlated with microservice information such as the interaction with other microservices and their locations, to suggest *deployment improvements*. For example, the *Frontend* and *Todo* microservices in FIG. 21 are tightly-coupled – in this case, our solution may suggest they be located close to each other. On the other hand, the *Todo* and *User* microservices do not need such proximity. Likewise, if we observe a sudden decrease in the number of user logs in a certain geographic location, we can recommend removing the replica from that location, saving money and resources.

Figure 21 – Refactored ToDo application architecture



Source: (SAMPAIO JR. et al., 2017)

3.3.3.2 Prospective Analysis

The prospective analysis uses the model as a "sandbox" for exploring the space of possible *architectural and deployment refactorings*. Several possible refactorings can be emulated as new snapshots of the model and evaluate their ability to handle the collected real-life μApp scenarios. That is, the model can assess potential improvements by replaying the traces corresponding to the scenarios from the current model in the new model. If the new model withstands a battery of tests such as replaying traces of a given time interval in the new model snapshot, we will issue a recommendation to change/refactor the μApp.

Moreover, as mentioned in Section 3.1.1, the model can be used as input for prediction tools (Artificial Intelligence) in order to foresee μApp behavior based on past behavior, thus anticipating actions to improve μApp performance and reliability.

3.3.4 Models@run.time

According to Blair *et al.* (BLAIR; BENCOMO; FRANCE, 2009), the use of Models@run.-time simplifies the inspection and adaptation of complex heterogeneous Systems. Hence, considering the heterogeneity of monitoring μApps, Models@run.time is an interesting concept to be applied in the managing these applications.

In addition to the retrospective and prospective analysis mentioned before, there is another advantage of using models at runtime – to plan out elaborate actions that can be

applied to μApps. The model organizes data in such a way that planners (see Section 2.1.1) 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, Models@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 within 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 evaluation cannot be performed quickly.

Despite the use of Models@run.time helping in some aspects of μApp adaptations, it remains a challenge to place microservices while adapting a μApp wisely. As discussed in the Section 3.1, μApp adaptation relies on placing microservices in different locations. The next section presents the importance of the optimal placement of microservices.

3.4 PLACEMENT OF MICROSERVICES

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

However, putting high-affinity microservices together is not enough to improve μApp placement. It is necessary to consider microservice runtime resource usage as well. Existing management tools do not take into account the resource usage history at runtime in order to place microservices – it can only observe instantaneous resource usage or the values set at configuration by the engineers. However, these values do not reflect the real behaviour of the microservices. Using this information to place microservice instances, can lead the μApp to contention situations.

Microservices can use more or fewer resources in specific workflows along μApp execution. Hence, we can observe peaks and lows in resource usage. If the management tool does a placement based on observing the peaks, it may place the microservices across several hosts, wasting resources. On the other hand, if the placement is based on the lows, many microservices may be co-located into a host, leading the μApp to a contention behaviour.

Although management tools, like Kubernetes, allow setting upper limits for resource usage, there is no guarantee that engineers will set these limits. And, if these limits are set, there are no guarantees that the chosen values are the best for all workloads during a μApp execution. Moreover, set limits in the management tools does not make

a microservice stay under the limit, it just tells the kernel when to start to constrain the microservice and when to kill it. This is most evident in interpreted languages, like Java (prior version 8)[22] and Python[23], in which the interpreters cannot handle the limits set on management tools properly and can crash the μApp when the limits are reached. This unbounded/unreliable approach leads management tools to make poor deployments, which may degrade application performance or crash the entire μApp.

Therefore, it is necessary to balance the relationship between microservices and their resource usage in order to improve their placement. In Section 3.3, we discuss the use of Models@run.time in keeping runtime (history) data to be used during placement computation. However, only models are not enough to handle the placement problem.

3.4.1 The Placement Problem

Microservice placement improvement is a hard task. Placing microservices in a cluster is a variation of the bin-packing problem (KORTE; VYGEN, 2006). 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 combined NP-Hard problem. In our context, the objects to be packed are the *microservices* and the bins are the cluster *hosts*.

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

Whatever the strategy, management tools do not use runtime information or history data in order to drive or enhance the placement of microservices. Existing tools select the hosts considering the minimum availability of resources to accommodate the microservice, and rarely the maximum. If these requirements are not set, any host can be a candidate to receive a microservice.

Unlike the classical bin-packing problem, the microservice placement in a cluster cannot consider only one dimension (size of microservice), but N dimensions: (i) the microservices affinities and their resources usage, e.g., (ii) CPU, (iii) memory, (iv) disk, and so on. Therefore, our problem is a multi-dimensional variation of bin-packing (CHEKURI; KHANNA, 2004) that is exponentially harder to solve.The formal statement of the microservice placement problem, guided by affinities 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 $A : m_{P_i} \rightarrow m_{P_i}$.

[22] <https://jaxenter.com/nobody-puts-java-container-129373>
[23] <https://stackoverflow.com/questions/36759132/why-does-docker-crash-on-high-memory-usage>

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 (CHRISTENSEN et al., 2016). However, the complexity of computing an optimal result in a reasonable time for large instances prevents its use at runtime.

There are several approaches surveyed by (CHRISTENSEN et al., 2016; KORTE; VYGEN, 2006) to compute this optimization in an offline way. At runtime, the best strategies are approximations, calculated through heuristics and evolving algorithms to achieve a *quasi*-optimal solution.

3.4.2 Requirements to handle μApp Placement

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 in 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 based on the blue-green deployment (CARNEIRO; SCHMELMER, 2016):

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. Moreover, 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. To deal with these faults, developers use design

patterns like *circuit breaker*[24] and *retry with the exponential back off*[25] to minimize their negative effect on a μApp.

However, stateful microservices may be a source of flaws during μApp adaptation. When a new microservice must replace an old one, the management tool cannot automatically synchronize their data. Therefore, when a stateful microservice is updated, a mechanism is necessary to deal with state synchronization. However, this approach is out of the scope of this thesis.

In next chapter, we show how REMaP handles these requirements and the strategies adopted to place microservices at runtime smartly.

3.5 CONCLUDING REMARKS

In this chapter, we discussed the main aspects of runtime adaptation in μApps. We introduced the monitoring of μApps and the placement of microservices as the main challenges in carrying out runtime adaptation on μApps. Then, we overviewed REMaP our MAPE-K based adaptation mechanism to carry out runtime adaptation on μApps. Finally, we present our approach to handle the challenges of monitoring and placing microservices. We show our approach using models to support μApp evolution and we described the placement of microservices as a multi-objective bin-packing problem.

[24] <http://microservices.io/patterns/reliability/circuit-breaker.html>
[25] <https://en.wikipedia.org/wiki/Exponential_backoff>

4 REMAP - DESIGN AND IMPLEMENTATION

While the previous chapter presented the main challenges of runtime evolution of μApps and a general overview of REMaP, this chapter presents the design choices in order to deal with the challenges mentioned before. Furthermore, this chapter details the current implementation of REMaP to handle the runtime adaptation of μApps by reconfiguring their placement at runtime.

4.1 BASIC FACTS

REMaP (**R**untim**E** **M**icroservices **P**lacement) is a MAPE-K (IBM, 2005) based adaptation manager that autonomously adapts 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. FIG. 15 overviews REMaP.

When an individual or a group of microservices are rolled out or rolled back, the *Adaptation Manager* starts executing the control loop indefinitely at regular time intervals.

The adaptation starts with the *Monitor* inspecting the μApp through the *Monitoring Adapters*. The adapters abstract different monitoring technologies to gather useful data, such as resource usage and microservice interactions (μApp workflow).

REMaP takes uniformly collected data and populates the application *Model*. The *Analyzer* inspects the model looking for interactions between microservices in order to compute their affinities. The *Model* stores information about affinities and the *Planner* accesses this information.

The *Planner* uses the affinities and resource usage stored in the model to calculate a new placement plan (adaptation plan) for the microservices. The *Adaptation Manager* applies the adaptation to the *Model* and checks the consistency of changes before consolidating them into the running μApp. The *Adaptation Manager* forwards the changes that do not violate the model to the *Executor*.

In this scenario, adaptation means the optimization of microservice placement. The optimization relies on two dimensions: microservice affinities and resource usage history. By considering these dimensions, the *Planner* computes a deployment plan to reduce wasted resources as well as improve application performance and decrease communication latency.

Unlike traditional approaches – whose only objective is to minimize the number of bins used, considering the number of items – the optimization of microservice placement

proposed also includes their affinities. Hence, in addition to the minimization of resource usage, we also aim to maximize the affinity score of the selected hosts, placing the maximum number of highly-related microservices together while minimizing the resources used.

Finally, 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.

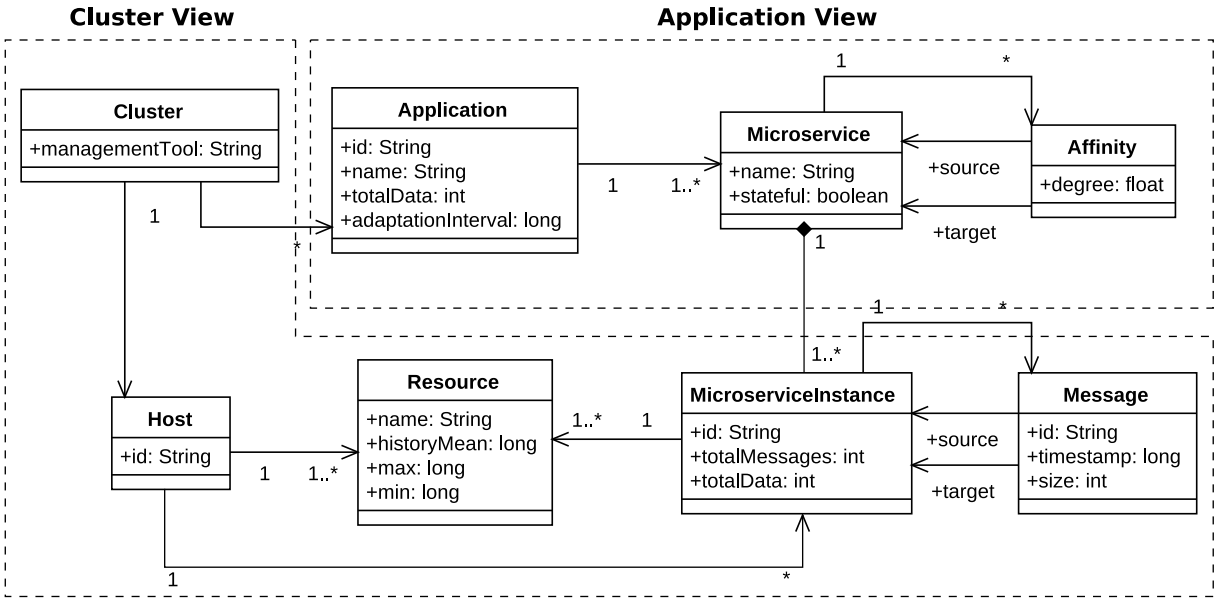
REMaP was developed in Java 1.8 and Python (Z3 binding). As management tool, we adopted Kubernetes 1.8 configured with the metric module Heapster to collect metrics from the cluster and carry out the adaptations on the μApps. The messages are collected from Zipkin 2.1, hence the μApps should be properly instrumented with the Zipkin bindings.

The rest of this chapter presents design and implementation details of all REMaP components.

4.2 MODEL

The *Model* shown in FIG. 22 abstracts and allows the inspection and analysis of μApps at runtime. As mentioned in Section 3.3, we use Models@run.time concepts to make the use of a unique artifact viable and reduce the semantic gap between technologies used to monitor μApps.

Figure 22 – μApp model



The *Model* abstracts essential elements of the μApps in a cluster. This model is inspired by the evolution model presented in Section 3.3. Different from the prior model, which was

created to support both architectural and runtime adaptations, the current one is variation dedicated for runtime adaptions, so that it is not necessary to maintain the evolution of μApp along time, neither to maintain concerns about the location of microservices in a geographic fashion. Moreover, the current model does not use the high level definition of scenario (workflow), instead, it uses the messages to derive affinities, used to guide the adaptation. In this work, the model serves as 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 – a μApp includes different kinds of microservices and each type of microservice can have multiple replicas (microservice instance). A microservice instance contains 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 message set describes the μApp's workflow.

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 interactions between two microservices.

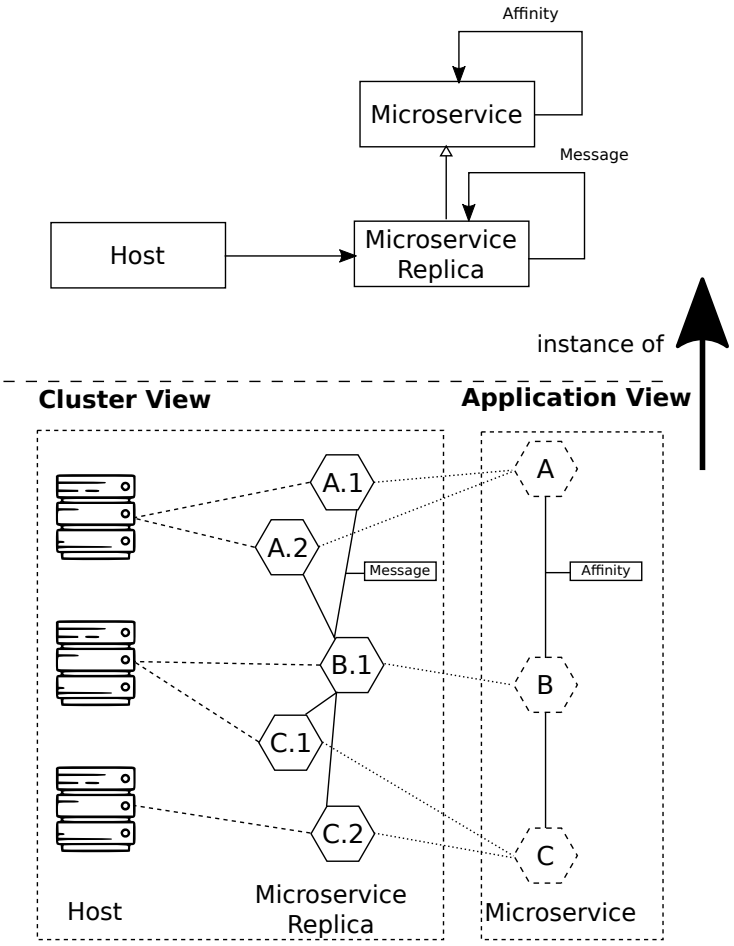
We decided to use affinities to link microservices and not their replicas because replicas are temporary and may come up/ go off many times in a short timespan due to failures or scaling process. Hence, using affinities between replicas could raise inconsistencies in adaptation time – since a link between two replicas is possible during calculation and some replicas go off, due to scaling in or failure, when the adaptation is carried out on the μApp.

Class *Cluster* abstracts the management tool, e.g., Kubernetes, 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 *Resource* attributes that maintain the information on usage history – hosts and microservices CPU and average memory usage (history) –, and resources limits, e.g., collecting the average of these metrics in a time interval from a storage like Influxdb. FIG. 23 shows how a μApp is represented at runtime by our model and highlights the μApp architecture (*Application View*) as well as the μApp deployment (*Cluster View*).

The model's cluster range is used to create the μApp as well as define 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's range is volatile as microservices replicas frequently come up and go at runtime.

Figure 23 – Instantiation of the model



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

The separation of application and cluster ranges creates a more expressive model. It allows 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* (shown in FIG. 15) 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 to the μApp (application view).

4.3 MONITORING

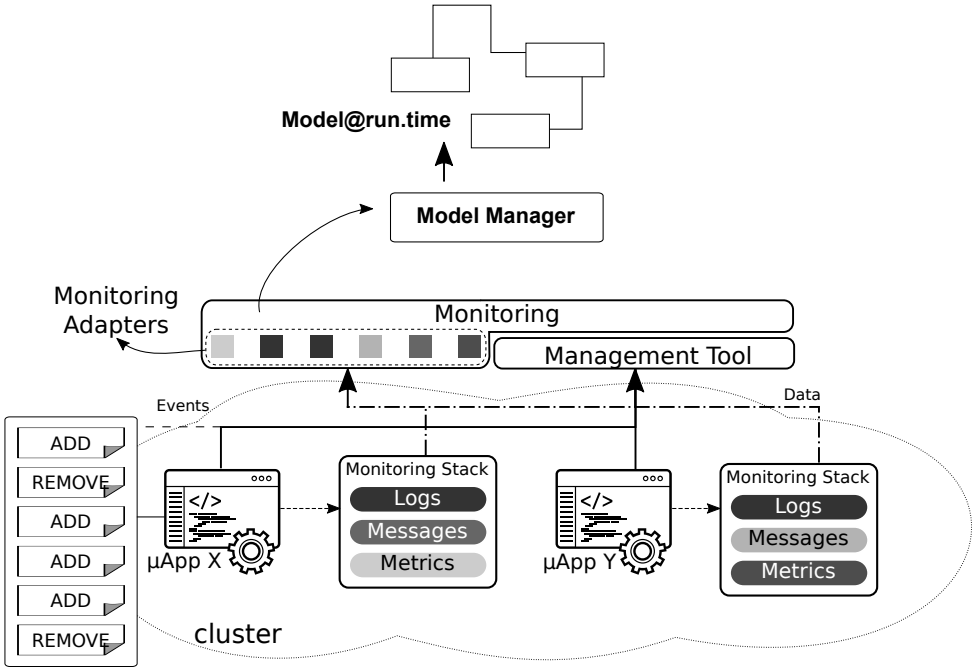
The *Monitoring* component works gathering data from a cluster when specific events are triggered. It collects data from different aspects of the μApp and transforms them into technology-agnostic data structures to populate the *Model* at runtime.

4.3.1 Design

The *Monitoring* component shown in FIG. 15, is designed for sampling heterogeneous data from the cluster. It is the first element in the REMaP's workflow and provides an unified API to inspect μApps in the cluster.

Monitoring tools in the cluster, e.g. cAdivisor, continuously collect data and store them in different data stores, such as InfluxDB, Elasticsearch, or even MySQL. For each data store technology in cluster, there is an adapter in the *Monitoring* component to transform a *technology independent operation* into a *technology specific operation* for getting data.

Figure 24 – Monitoring component.



These adapters provide a homogeneous API for heterogeneous data stores. Hence, it is possible sampling, e.g, CPU metrics, from different data stores, e.g., Influxdb and Prometheus, by using a single and homogeneous operation, e.g., `metricsHosts()`. This data transformation generates technology agnostic structures used for populating the model. Hence, the component *Monitoring* provides a standard and unified way to retrieve data from different technologies used in the cluster. FIG. 24 illustrates the interior of the *Monitoring* component.

Every μApp has a monitoring stack to collect data which usually collects three different kinds of data: resources usage, execution logs and exchanged messages. The *Monitoring* component maintains a complete view of the environment by gathering information from: the management tool in use (e.g., Kubernetes or Docker Swarm), host and microservice resource usages and events generated by DevOps operations.

Finally, the *Monitoring* component provides data according to various aspects such as resources metrics, messages, logs and events. For each of them, the monitoring component

allows sampling of the data over a given period. Especially for metrics (CPU and memory usage), the monitoring aggregates the values by calculating their average use during the sampling period, e.g., the average of CPU usage in the last hour.

The *Monitoring* component collects the cluster data in two ways (FIG. 24). The information that comes in a high volume of data – such as resources metrics, logs and exchanged microservice messages – is collected discretely while the other information – such as μApp architecture and microservices running – is collected continually. Next we describe these sampling strategies.

When the monitor works in a discrete fashion, timed events trigger the *Monitoring* component to gather the μApp data. These same events are the responsible for triggering the adaptation process. REMaP uses this time interval to gather data indefinitely – waiting between each sampling t time unit – starting the adaptation after the sampling is complete.

The engineer sets the time interval for monitoring when the μApp is upgraded (microservices are rolling out/back). The engineer informs the deployment infrastructure – through labels in the deployment command – how long REMaP should wait before the sampling date. For example, the *Monitoring* component collects data in timed intervals defined by the engineer (e.g. every 10 minutes) and starts the adaptation after sampling is complete.

When the *Monitoring* component is triggered, it retrieves data from the data store into the cluster – fetching metrics and logs from microservices as well as their exchanged messages – and transforms it into a populated the model. This strategy is used to avoid flooding the *Adaptation Manager* and consequently triggering too many adaptations.

Continuous monitoring is a strategy to maintain the model updated with the current architecture of the μApp in real time. The *Monitoring* component lists management tool events that continually come up from the cluster, such as START and STOP microservices replicas. These events are usually signaled when the μApp or cluster changes. As this information comes in a short amount of data, the *Model Manager* can handle it properly – maintaining the runtime model updated in real-time and in conformance with the current architecture of the μApp and its active replicas.

Similarly to data store adapters, the *Monitoring* component also abstracts different management tools such as Kubernetes and Docker Swarm, so that events from different events on this platforms are standardized in the same interface.

4.3.2 Implementation

We implemented the *Monitoring* component (see FIG. 15) to collect data from Influxdb[1] and Zipkin[2]. Influxdb stores resource information from the microservices and hosts. Heap-

[1] <https://docs.influxdata.com/influxdb>
[2] <http://zipkin.io>

ster, a Kubernetes (v1.8) plug-in[3], collects microservice and host resource usage information. It inspects microservice containers and hosts, and stores CPU and memory metrics into Influxdb. We choose Heapster and Influxdb because they are the default tools used in a standard Kubernetes deployment, but other tools might be used, such as Prometheus over Influxdb.

Zipkin is a distributed tracing system used to collect exchanged messages between microservices. Developers have to instrument the microservice with code snippets informing which messages Zipkin needs to capture and the amount of messages to be sampled. Code 4.1 is an example of the steps necessary to instrument a Java code using Spring Cloud Sleuth Framework[4] to use Zipkin for collecting messages exchanged by microservices. Once collected, Zipkin stores and makes the messages available via API.

Code 4.1 – Code snippet for instrumenting a Java microservice to use Zipkin through Spring Sleuth

```
class Configuration {
  @Autowired
  private SpanAccessor spanAccessor;
...
  Span span = this.spanAccessor.getCurrentSpan();
  ...
  template.header(Span.TRACE_ID_NAME, Span.toHex(span.getTraceId()));
  setHeader(template, Span.SPAN_NAME_NAME, span.getName());
  setHeader(template, Span.SPAN_ID_NAME, span.toHex(span.getSpanId()));
...
  @Bean
  Sampler customSample() {
    /*
      The follwoing Sampler would trace roughly half of all requests
    */
    return new Sampler() {
      @Override
      public boolean isSampled(Span span) {
        return Math.random() > .5;
      }
    };
  }
...
}
...
package com.example;

import org.springframework.boot.*;
import org.springframework.cloud.sleuth.zipkin.stream.*;

@EnableZipkinStreamServer
@SpringBootApplication
public class ZipkinServiceApplication {
  public static void main(String[] args) {
    SpringApplication.run(ZipkinServiceApplication.class,args);
```

[3] <https://github.com/kubernetes/heapster>
[4] <https://cloud.spring.io/spring-cloud-sleuth/>

```
36   }
37 }
```

We also implemented a Kubernetes client to collect signals from the cluster and cluster configuration. Kubernetes has an API that provides cluster data (available hosts and running microservice instances); thus, whenever the cluster suffers some change, lika a new microservice replica, the client collects this data to update the model.

The 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 – hosts and running microservice instances. These interfaces are combined into a *InspectionInterface*, Code 4.2, exposed by the monitoring.

Code 4.2 – Monitoring Interface

```
1  interface ClusterInspectionInterface {
2      // list all hosts available in the cluster
3      List hosts();
4      // returns all services (including replicas) running in the cluster
5      List services();
6      // returns all applications running in the cluster and the weight
7      // for calculate the affinities
8      Map<String, Float> application();
9      // returns the management tool being used
10     String cluster();
11 }
12
13 interface MetricsInspectionInterface {
14     Map<String, Float> metricsMicroservice()
15     // returns the metrics of all hosts in the last t seconds
16     Map<String, Float> metricsHosts(long timeInterval)
17 }
18
19 interface MessagesInspectionInterface {
20     // returns all messages from the last t seconds
21     List messages(long timeInterval)
22 }
23
24 interface InspectionInterface extends ClusterInspectionInterface,
25   MessagesInspectionInterface, MetricsInspectionInterface {
26 }
```

Finally, the monitor also implements a listener to handle DevOps events. The listener receives events from Travis Travis[5], a continuous integration tool that signals when a new deployment event occurs (e.g. upgrading a μApp). Hence, when engineers upgrade the μApp, the listener resets the timer in the adaptation engine – as described in Section 4.7.

All the data collected from different data sources is used to populate the model illustrated in FIG. 23. The *ClusterInspectionInterface* provides the data used for *Model*

[5] <https://travis-ci.org>

Manager instantiate the class that represent hosts, applications and microservices. Internally, the *Monitoring* component receives a continuous flow of events telling what is happening in the cluster – microservices and hosts being instantiated or removed – and passing them on to the model manager, to instantiate their related classes into the model.

The *MetricsInspectionInterface* provides the data used for the *Model Manager* to populate the microservice's resource consumption and the host in the model. Component *Monitoring* gathers the metrics according to their types and targets – CPU usage of microservices N – and *Model Manager* gets aggregations when they are triggered (Sectiondesign.monitoring.design). The calculated aggregation is the average of resource usage in a given interval (Section 4.3.1) – the average CPU usage in microservice N in last 10 minutes.

Finally, *MessagesInspectionInterface* provides the data used to link the microservices and to define the μApp workflow. The *Model Handler* component gets the messages exchanges by all microservices through the *Monitoring* component.

4.4 ANALYZER

The *Analyzer* component inspects the model in order to calculate the affinities between microservices. The messages exchanged by the microservices are collected and kept on the model. The *Analyzer* examines the messages and – based on the number of messages exchanged by two microservices as well as the size of the messages – calculates the degree of affinity between two microservices.

4.4.1 Design

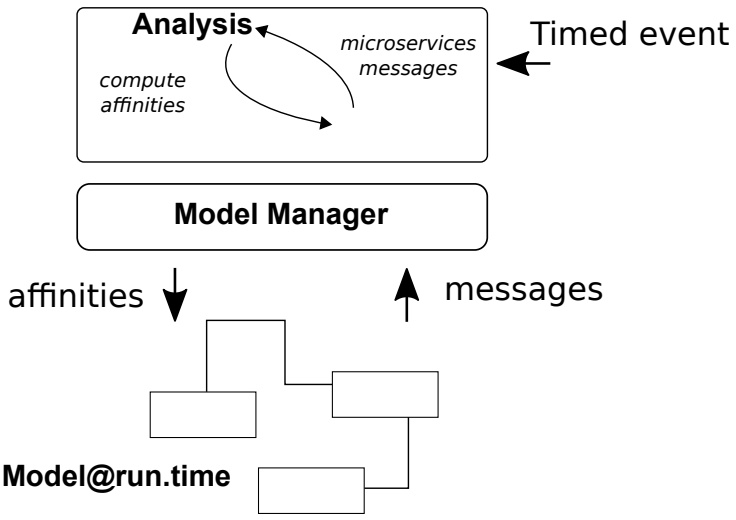
The *Analyzer* component, shown in FIG. 15, is designed to process the model at runtime by looking for affinities between microservices. We define *affinity* by using the number of messages and the amount of data exchanged between them. We use a ratio (weight) in the affinity calculation to steer the *Analyzer* execution. The data exchanged between the microservices are not equally distributed among the exchanged messages. μApp workflows with few messages and a significant amount of data, and vice-versa, may exist. Hence, a high ratio value (> 0.5) leads the *Analyzer* to evaluate the number of messages over the amount of data. A small ratio value (< 0.5) does the opposite, and a ratio of 0.5 balances the two attributes equally.

The weight is set when the REMaP is installed to monitor the cluster. The cluster provider (engineer) should set what weight REMaP will steer the adaptation.

In the general purpose of MAPE-K, the *Analyzer* component gets signals from monitoring named symptoms and, from these signals, detecting some aspect from the application or environment. In this work, we design an *Affinity* analyzer that detects affinities

between microservices. As mentioned in Section 3.4, the affinity is used to optimize the μApp placement.

Figure 25 – Analysis component.



In REMaP design, the *Monitoring* component sends timed events to the *Analyzer* component (FIG. 25) signaling it to start μApp analysis. The analyzer checks the messages exchanged by the μApp (stored in the model) and calculates the affinity between two microservice implementations (types) and not their instances (replicas). The analysis of the microservice types over their instances avoid inconsistencies when the adaptation is applied to the μApp.

During the analysis process, or even the adaptation, microservice replicas might come up or go off due to μApp dynamics. Whether an affinity is calculated based on replicas or not, it might be invalid when applied to the μApp, seeing as the replicas used to calculate affinities are not available anymore. To avoid this, we aggregate the messages between microservices by their types. Hence, if the replicas are not available, or new replicas come up, the adaptation manager can handle them based on previously calculated affinity by using their types.

The calculation of affinities uses the number and size of messages to determine the bi-directional affinity. We define the affinity between two types of microservices a and b as $A_{a,b}$ and calculate it as follows:

$$A_{a,b} = \frac{m_{a,b}}{m} \times w + \frac{d_{a,b}}{d} \times (1 - w), \tag{4.1}$$

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 or b and a,

- d is the total amount data exchanged by all microservices,
- $d_{a,b}$ is the amount of data exchanged between microservices a and b or b and a,
- 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. If $w < 0.5$ then the number of messages is more important, if $w > 0.5$ then the amount of data is more important. Otherwise, if $w = 0.5$ both information are equally important.

The analyzer calculates the affinities between all microservice instances and dispatches them to the Planner via Model Manager (FIG. 26). The analyzer aggregates affinities of microservice 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.

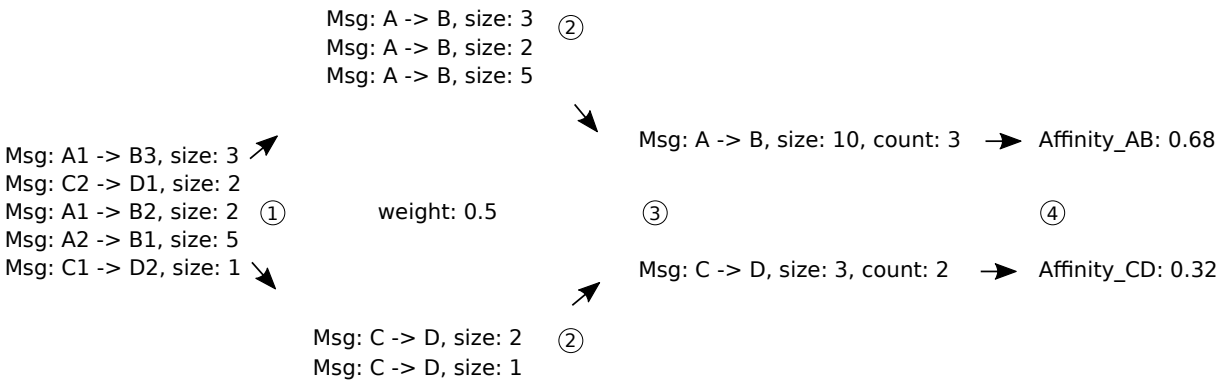
4.4.2 Implementation

As presented in Section 4.4.1, the analyzer retrieves information about messages exchanged by the microservices from the runtime model and calculates their affinities using the Equation 4.1. Our analyzer uses the EMF framework[6] to look up the elements in the model. The EMF framework has inner mechanisms to traverse the model transparently. Hence, the analyzer only looks up elements by their types; in this case, looks up message types.

The analyzer then applies a map-reduce[7] based algorithm on the massages to calculate the affinities. FIG. 26 depicts the overview of this calculation. The algorithm is performed in two steps: first, the analyzer loops through all messages, mapping their endpoints (microservices replicas) with their types (microservices types); next, the analyzer groups the messages with same endpoint types and applies the Equation 4.1 to these groups. After doing so, the affinity degree calculated is used to instantiate the *Affinity* classes (see FIG. 23). The affinity is kept sorted in the model in an ascending fashion. Finally, the *Planner* component is signaled by the Analyzer to get the affinities and calculate the adaptation plan.

[6] <https://www.eclipse.org/modeling/emf>
[7] <https://en.wikipedia.org/wiki/MapReduce>

Figure 26 – Affinities calculation.



4.5 PLANNER

The *Planner* examines the model – which has affinities between microservices and the microservice resources usage history – and resources available in the hosts. Using this information, it decides how to apply the adaptation to the μApp by computing the best hosts to place the microservices. The *Planner* adopts two strategies to compute the placement of microservices, a heuristic, which does not guarantee an optimal placement, and an algorithm that guarantees optimal placement. As a result, the *Planner* generates a list of movements in order to update the μApp by changing the placement of a microservice from one host to another.

4.5.1 Design

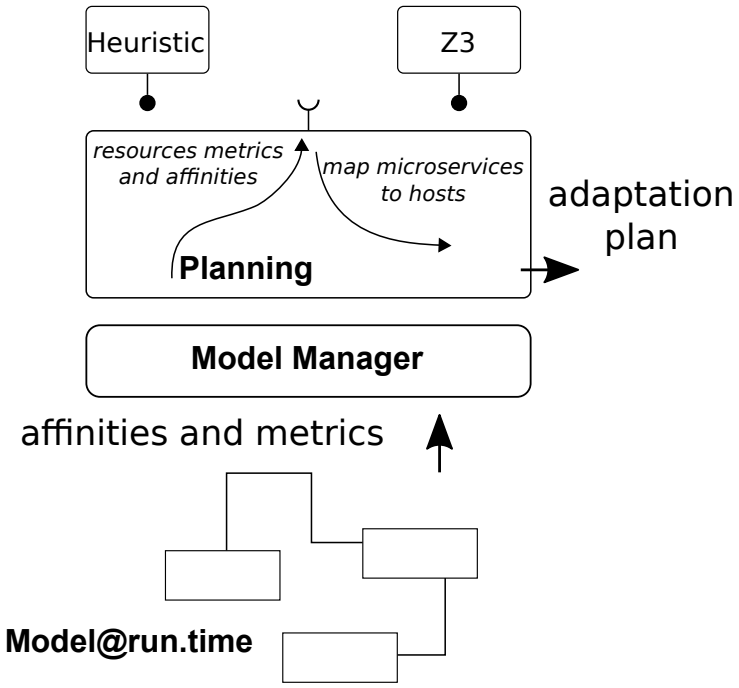
The *Planer* component (FIG. 27) uses the model at runtime to get the affinities among microservices and their resource usage history and uses this information to calculate an adaptation plan for μApps. The adaptation plan is a list of movements that the *Executor* component should carry out in the cluster, moving the microservices from a source host to a target host.

We propose three *Planners* to compute the placement of microservices during an adaptation: *Heuristic-based Affinity Planner* (HBA), *Optimal Affinity Planner* (OA) and a variation of OA, named *OAmodified Planner*. All planners compute a new placement for μApps by reducing this problem to a multi-objective bin-packing problem. Since this problem is NP-Hard, we know that an optimal approach is unfeasible for large μApps. Hence, we designed the heuristic version (HBA), to achieve quasi-optimal solutions for large μApps, and the optimal version (OA), to achieve an optimal solutions for small μApps.

The size of a μApp is related to the reference adopted such as the number of microservice types or the number of replicas running. For example: What is a big μApp? One having four types of microservices, each with a dozen replicas, or, one having ten types of microservices, each having a single replica? In this work, we agreed to classify the size

of the μApps based the number of microservice types. Hence, we consider small μApps those applications with less than 15 types of microservices. This threshold comes from the average size of the most relevant μApps listed by Aderaldo et al. (2017), which suggests μApps for benchmark Microservices Architectures.

Figure 27 – Planning Component.



Both HBA and OA planners access the model in order to obtain information about 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 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 microservices.

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.

4.5.1.1 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 in such way that microservices with high affinities are co-located, while microservice resource usage and the availability of these resources in the host are taken into account. This planner uses Algorithm 1 to compute the list of movements necessary to reconfigure the μApp.

The heuristic (FIG. 28) is a simplification of First-Fit (DOSA, 2008) approximation algorithm, it iterates over the affinities and tries to co-locate microservice instances asso-

Algorithm 1: Heuristic to move microservices.

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

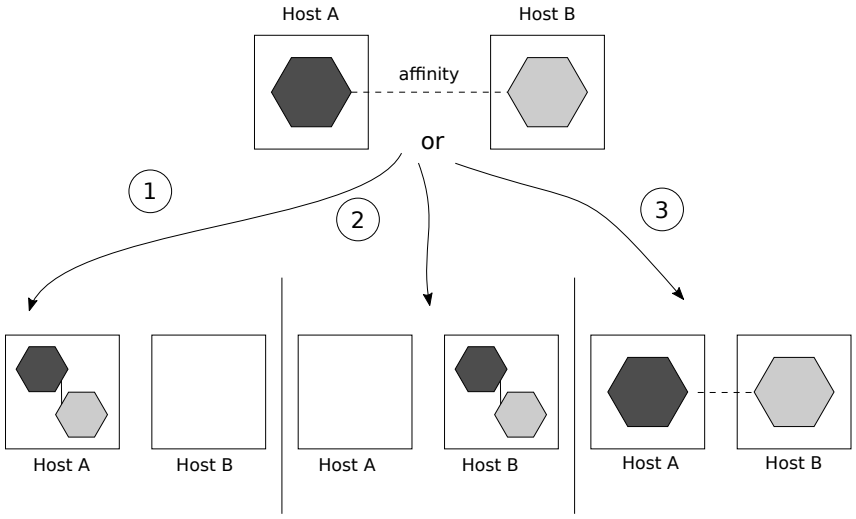
ciated with them. For each pair of types of microservices (m_i, m_j) linked by an affinity, the heuristic attempts to (1) place their instances $m_{j,l}$, onto the same host as the instances $m_{i,k}$ (H_k). If H_k does not have enough resources, (2) the algorithm tries to put $m_{i,k}$ on the same host as $m_{j,l}$ (H_l). If both hosts do not have enough resources to co-locate $m_{i,k}$ and $m_{j,l}$, (3) these microservices remain in their original hosts. When a microservice is placed on 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 heuristic does not guarantee that the list of moves computed is optimal for a cluster given a set of microservices.

4.5.1.2 Optimal Affinity Planner (OA)

OA planner optimizes the placement of μApps. Given a list of affinities between microservices, this planner computes an optimal configuration for microservices in a cluster. The optimization is calculated by using a SAT solver (BIERE et al., 2009). We were inspired

Figure 28 – Graphical representation of heuristic.



by Bayless et al. (2017) that uses a SAT Solver to deal with a similar optimization problem of VM allocation in a cluster. However, in said case, there is not the concept of affinities. Hence, we aim to demonstrate the feasibility of using SAT solvers to handle our placement based on affinities in a cluster.

We state our placement optimization problem as follows:

Maximize:

$$\sum_{(j_i,j_k,\text{score})\in A, n\in\text{Hosts}}^{} \text{if } (p(j_i,n) \wedge p(j_k,n), \text{score}, 0) \tag{4.2}$$

Subject to:

$$\Big[\sum_{n}^{\text{Hosts}} p(j,n)\Big] = 1 \quad \text{for } j \in Microservices \tag{4.3}$$

$$\Big[\sum_{j}^{\text{Microservices}} \text{if } (p(j,n), M(j), 0)\Big] \leq M(n) \quad \text{for } n \in \text{Hosts} \tag{4.4}$$

$$\Big[\sum_{j}^{\text{Microservices}} \text{if } (p(j,n), C(j), 0)\Big] \leq C(n) \quad \text{for } n \in \text{Hosts} \tag{4.5}$$

Where:

- $Microservices$ is the set of microservices to be deployed,
- $p(j,i)$ is true if microservice j is placed on host i,
- $A \subset Microservices \times Microservices \times Integer$ 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 4.2 defines the objective function, maximizing the sum of affinity scores of all co-located microservices j_i, j_k. This equation returns *score* if $p(j_i, n) \wedge p(j_k, n)$ evaluates to true and 0, otherwise.

Equation 4.3 is a constraint enforcing that each microservice instance should be placed on precisely one host, once a microservice is indivisible and cannot be placed in multiple hosts. 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. Equation 4.4 and Equation 4.5 ensure that each host has sufficient memory and cores to execute 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, The Equations 4.2 to 4.5 are used to model the optimization algorithm to place microservices replicas with high affinities together, considering host resources and microservice resource usage history.

Unlike the HBA planner, the OA planner guarantees that it finds an optimal placement – i.e. it minimizes wasted resources.

The designs of OA planner and OA-modify planner are the same and the difference is only in their implementation details.

4.5.2 Implementation

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

4.5.2.1 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, it is stored in the *Adaptation Script* (Code 4.3). This script is a movement list informing the source and destination of a microservice instance in a cluster.

Code 4.3 – Example of Adaptation Script

```
1 class Moviment {
2     String microservice;
3     String source;
4     String destination;
```

```
5       ...
6   }
7   ...
8
9   List<Moviment> adaptationScript = new LinkedList<>();
10  ...
11  adaptationScript.add(new Moviment("usvc1", "hostA", "hostB")
12  adaptationScript.add(new Moviment("usvc2", "hostB", "hostC")
13  ...
14  adaptationScript.add(new Moviment("uscvN", "hostX", "hostZ")
15  ...
```

It is worth observing that only stateless microservices can be moved. As mentioned in Section 3.4.2, stateful microservice movement can raise issues on μApps execution. Hence, we decided to add this constraint to the HBA planner.

To better illustrate this point, given microservices A.1 and B.1 running on hosts H.a and H.b, the planner checks if A.1 and B.1 fit in H.a. If possible, a movement is computed to move B.1 to H.a; otherwise, the planner checks if both microservices fit in H.b. In this case, a movement is computed to move A.1 to H.b. If neither microservices fit in 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.

4.5.2.2 OA Planner

OA planner computes the placement of microservices using an SMT solver to calculate the optimal arrangement of microservices in such a way as to minimize the number of used hosts and maximize affinity scores for each host.

Our implementation uses Z3[8], a state-of-art SAT solver developed by Microsoft. Z3 is open source, has a high performance and is widely used, which contributes to it having good documentation as a support by its community. The optimization is modelled as a satisfactory statement (BIERE et al., 2009) that can return 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, which makes it impossible for SMT solvers to handle large instances of the problem being solved in a reasonable time.

We choose Python wrapper for Z3 because they are better documented. Hence, OA Planner is a Python script that receives the elements of the model necessary to compute the placement in JSON notation and uses the available methods in the Z3 wrapper to implement equations presented in Section 4.5.1.2. The implementation of these equations uses the specific syntax of Z3 – the complete code used by OA Planner – to calculate the optimization depicted in Appendix A.

[8] <https://github.com/Z3Prover/z3>

REMaP integrates the OA Planner with the their other elements through the Java Process API[9]. *Planner OA* transforms the SMT solver output into an adaptation script, Code 4.3, like the HBA Planner. Next, it sends the list to the model manager that forwards it to the executor. Unlike the HBA planner, OA planner can move *A.1* and *B.1* to hosts other than H.a and H.b.

Finally, the implementation of OA-modified Planner is the same of OA Planner except in lines 13 and 151 in Appendixappendix-a, where statement `Optimize` is replaced by `Solver` (line 13) and the statement used to calculate optimization (line 151) is removed. This change makes the Planner try to find a solution that satisfies the constraints Equations 4.3 to 4.5, but it does not satisfy the objective of the Equation 4.2.

4.6 EXECUTOR

The *Executor* component carries out the placement computed by the *Planner* in the cluster. For each movement received from the *Planner*, the *Executor* validates it against the current version of the model – if the movement is in conformance with the current state of the model, the microservices are moved to a new place. Next, we describe the *Executor* and present how it validates a movement in order to update the μApp.

4.6.1 Design

The *Executor* Component, shown in Fig. 29, is designed to apply the adaptation plan computed by the *Planner* in μApps safely. As μApps are constantly changing, a not yet applied computed adaptation may be unsafe to apply due to external factors, such as microservice implementations that have arisen or gone (μApp upgrade) during adaptation computation.

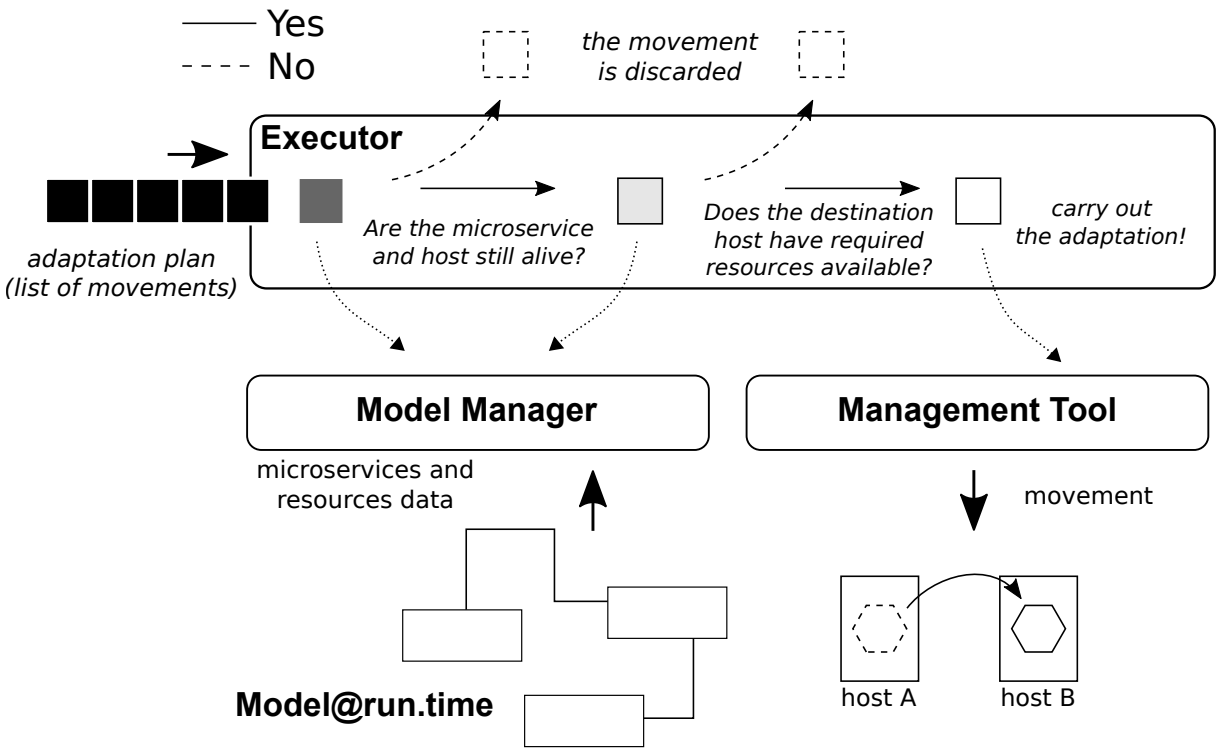
For example, assuming that microservices A and B have two instances – A.1 and A.2; and B.1 and B.2 – the analyzer checks the model to see if the microservice in type A has a high affinity with B (Section 4.4). Also, suppose that the planner has computed that A.2 should be co-located with B.2, but at runtime A.2 has been de-allocated due to an unexpected scale-in action – the movement to co-locate A.2 with B.2 becomes invalid.

If the application is upgraded and has part of its architecture changed as the *Model* is causally connected, 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.

To validate the adaptation, the *Executor* checks all movement from the adaptation script on the *Model*. For each movement, the *Executor* checks if the hosts and microservice instances are in the model and if the model hosts have resources available to fit the

[9] <https://docs.oracle.com/javase/8/docs/api/java/lang/Process.html>

Figure 29 – Executor Component.



microservice instance. The movement is only performed if, and only if, these two constraints are satisfied. If they are not, the current movement is discarded and the *Executor* tries to apply the next one in the adaptation script.

To perform a `Movement`, the component *Executor* moves a microservice to a host in the model. This high level action in the model is translated into several low level actions according to the management tool. Hence, an executor is needed for each management tool. This translation is exemplified in Table 1.

Table 1 – Translation of a REMaP movement to Kubernetes actions

REMaP	**Kubernetes**
`move(msA, host1, host2)`	1. To get microservice A in the host 1 2. To check if microservice A is in conformance with host 2 3. Instantiate a replica of microservice A into host 2 4. Remove the replica of microservice A into host 1

Source – Made by the author

Using only 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* is designed to deal with microservice instances that come up during the adaptation process. Once the model has the microservices linked by affinities, the executor can use this information to guide them 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 scales 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. However, it rechecks if the model and the host, where A.1 is placed, do not fit another replica of B. So, the executor will try to co-locate B.2 with another microservice instance so that both types have an affinity. If such microservice does not exist, the executor maintains replica B.2 at the host where it was instantiated.

4.6.2 Implementation

The component *Executor* wraps the management tool in the cluster. When the executor receives the adaptation plan, it applies the moves one-by-one in the model. If a move cannot be applied, as discussed in Section 4.6.1, it is discarded – otherwise the move is sent to the management tool wrapper, that applies it to the μApp. Implementation currently includes wrappers for the most common management tools – Kubernetes and Docker Swarm. These tools are standard in cloud vendors that provide infrastructure for running μApps.

In Kubernetes, it applies the changes by updating how the microservices are attached to the hosts in the deployment description maintained at runtime by changing the `nodeSelector` labels of the microservices (Pods). Kubernetes goes through all services updating their placement attributes. Next, it executes the update by creating a replica of the updated microservice in a new host, starts the microservice and automatically removes the old one.

The process of how Kubernetes works internally to instantiate a microservice is out of scope of this thesis. A detailed explanation about this process can be found in (HANNAFORD, 2017).

Docker Swarm wrapper works similarly, we set `labels` and `constraints` to the hosts and microservices to control the placement of microservices. 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.

4.7 MODEL MANAGER

The *Model Manager* maintains the model at runtime, drawing it up and maintaining a causal connection between the model and the μApp. The *Model Manager* coordinates other REMaP components as well.

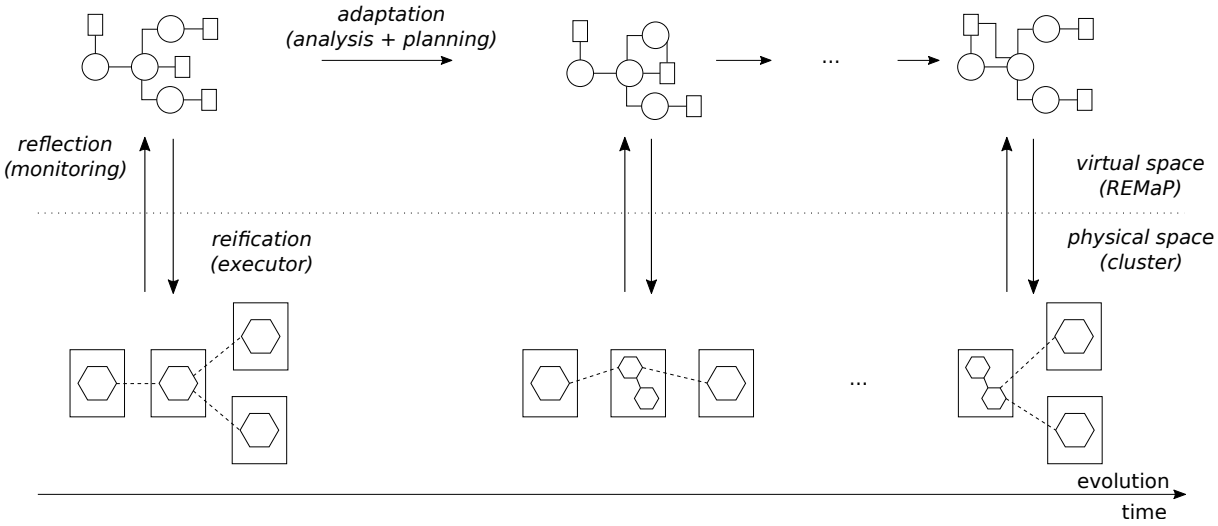
4.7.1 Design

The *Model Manager* is the core component of REMaP. It coordinates 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 MAPE-K elements and maintaining the model at runtime.

REMaP maintains a causal connection between the model and the μApp by coordinating MAPE-K implementation. Each of the components, *Monitoring*; *Analysis*; *Planning* and *Executor*, work on one aspect of the causal connection. It is the role of the *Model Manager* to coordinate these components to maintain the model and the μApp connected.

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 – *reification* – REMaP consolidates the changes to the model into the executing μApp through the executor.

Figure 30 – Evolution of μApps trough causal connection by using REMaP.



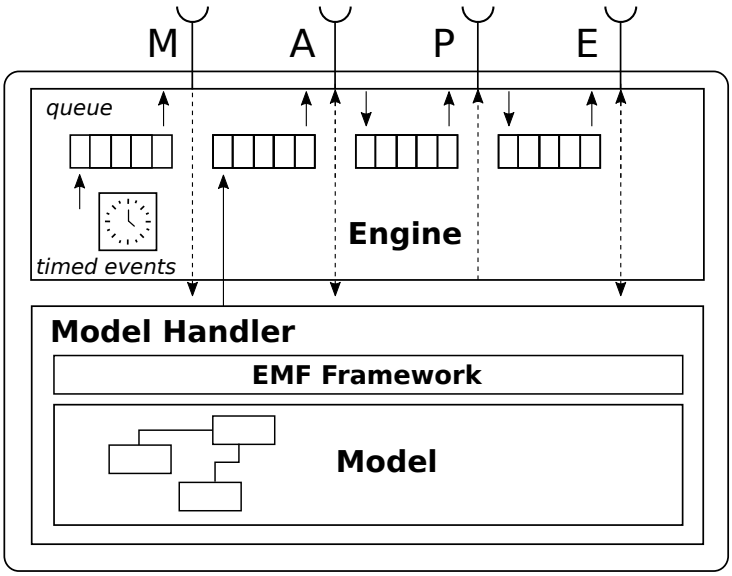
The causal connection begins with the *Monitoring* component, which gathers all data needed to build the model from the cluster. The *Model Manager* uses this data to build up the model. The model reflects the μApp configuration in a virtual space so that it is possible to inspect and reconfigure the μApp without a direct access. Moreover, any change in the μApp is automatically reflected in the model.

The components *Analysis* and *Planning* are those responsible for reconfiguring the μApp according to their specification (see Section 4.4 and Section 4.5). The reconfiguration (adaptation plan) is used by the component *Executor* to consolidate the changes in the μApp and consequently update the model.

During the adaptation, the component *Executor* checks all changes (`Movements`) against the model and updates the model if they are valid. The change into the model is translated into low-level actions and applied to the μApp; closing the causal-connection loop.

FIG. 31 depicts the *Model Manager* and its two key elements: *Model Handler* and *Adaptation Engine*. The model handler populates the model by using data structure (see FIG. 23) and performs changes to its data structure – e.g., the inclusion of affinities and microservice moves.

Figure 31 – Model Manager Component.



The adaptation engine coordinates the actions of the MAPE-K components and provides an interface to add new analyzers and planners; it also triggers the adaptations. In this work, the adaptation performed is a placement optimization applied to 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 in order to wait for this new version to generate enough data to populate the model. The control loop is started when the time interval is reached. The analyzer calculates the affinities, updates the model and notifies the planner. The planner then uses the affinities to compute an adaptation plan, sending it to the executor – that migrates the microservices.

4.7.2 Implementation

REMaP (see FIG. 15) 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 which integrates all components involved in the adaptation. The REMaP components populate and consume the messaging component. FIG. 31 depicts the internal elements of *Model Manager*.

The *Monitoring* component starts working by consuming timed events – set by the μApp engineer (see Section 4.3). The *Monitoring* component then collects the cluster

data and sends it to the *Model Manager*, which consumes and uses the data to create model classes (see Section 4.2), while maintaining the model structure at runtime. When the *Model Handler* builds the model, it notifies another queue, which is then consumed by the *Analyzer*. The *Analyzer* then consumes the notification and analyzes the model, signaling the *Planner* – in a similar way using a different queue – after it finishes. The *Planner* computes the adaptation plan, generating an adaptation script and pushing it to the *Executor*'s queue. Finally, the *Executor* consumes the script in the queue and carries out the adaptation (Section 4.6).

Queue communication was designed foreseeing the evolution of REMaP's implementation, where each component might be a different process or microservice. This is a natural evolution, since the model tends to be very large and making MAPE processing a machine might be computationally expensive.

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 as well as the total data exchanged between the microservices.

When a *message* is attached to a microservice instance, EMF signals 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 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 4.7.1. 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 μApp deployment, e.g., a custom message during a git push operation. The continuous delivery tool, Travis, handles the deployment of the microservices and notifies the *Monitor* – via a Web hook – when the building process is finished. Travis is also responsible for sending the custom messages from the git push to the *monitor* telling how long to wait between the data gathering.

4.8 CONCLUDING REMARKS

In this chapter, we discussed the design and implementation of REMaP and how it is used to improve the placement of μApps at runtime. First, we presented the model used to keep

runtime information of μApps and their primary elements. The model keeps the information collected by the component *Monitoring*. The *Monitoring* collects runtime information of several aspects of μApps at runtime. Next, we introduce the component *Analyzer* and its algorithm to calculate microservice affinities and how the messages exchanged by microservices are used for this computation. The *Analyzer* calculates the affinity degree between all tuples of microservices in the μApp based on the size and number of messages exchanged by the microservices. Finally, we presented the component *Planner* and the different strategies to calculate the placement adaptation of μApps. Than, the component *Executor* carry out the adaptation on the μApps deployed on Kubernetes and Docker Swarm. All components are integrated by the component *Adaptation Manager* which coordinates all REMaP's components and maintains the causal connection between the μApp and the model.

5 EVALUATION

In this chapter, we evaluate the impact of REMaP on the μApp adaptation considering several different scenarios. Initially, we introduce the objectives of the evaluation followed by the set up of the experiments. Finally, we present the results along with their analysis.

5.1 OBJECTIVES

As presented in Chapter 4, REMaP is a mechanism used to carry out adaptations in μApps. As REMaP manages another software, we initially designed a set of experiments to show the performance of individual components of REMaP itself. We then moved on to understand the impact of REMaP on μApps, evaluating the impact on the performance and resource usage of μApps. These experiments can be detailed as follows:

1. *Performance of REMaP components.* As presented in Section 4.1, REMaP implements the MAPE-K components and the duration of the adaptation depends on the time spent in each of these components. Hence, we aim to identify those times to better understand the individual impact of the components.

2. *Resources usage of μApps.* REMaP was developed to adapt μApps by improving their placement. This improvement usually reduces the number of hosts being used by the μApps. Hence, we aim to measure the gains made by REMaP on μApp resource usage.

3. *Performance of μApps.* As shown in Section 4.5 REMaP co-locates high-affinity microservices into a host to decrease the impact of the network latency in communication. This strategy has a potential to reduce the time spent in the communication between microservices, improving the overall performance of the μApp. Hence, we aim to measure the improvements made by the adaptation on the μApp's performance.

Next, we describe the design of the experiments.

5.2 EXPERIMENTS

We used mock and empirical evaluations to assess the objectives mentioned before. The mock evaluation relies on using an entirely controlled scenario – the cluster and μApps are modeled artificially as graphs that emulate the topologies of real μApps with different

sizes. The mock evaluation allows us to check how REMaP behaves when adapting a vast range of μApps, once we cannot use the same variety of real μApps for this purpose.

In turn, the empirical evaluation relies on a real μApp running in a cluster. The empirical evaluation allows us to check the impact of the REMaP's adaptation on a real μApp, i.e., how the saving resources impact on the μApp performance. This observation cannot be carried out on mock evaluation, once the mock μApps has no behavior at runtime.

In both evaluations the μApps are initially deployed following the *spread strategy* as shown in Section 2.4.6.

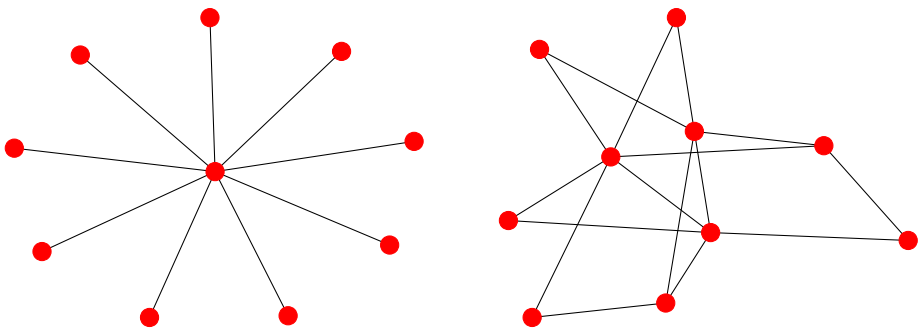
In all experiments, REMaP was executed in a dedicated machine equipped with an Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz with 16GB of RAM and running Ubuntu 16.04 LTS.

5.2.1 Mock Experiment

For the mock evaluations, we used the Barabási-Albert scale-free network model (BARABÁSI; ALBERT, 1999) to generate two artificial topologies, as shown in FIG. 32. Both topologies emulate configurations widely adopted in μApps: API-gateway and Point-to-Point (see Section 2.4.5).

Figure 32 – Topologies used in the experiments

(a) API Gateway (b) Point-to-Point



Barabási-Albert is a well-known model used to generate random network graphs. A graph of n nodes grows by attaching new nodes having m edges – usually attached 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 reproductibility, we used the library NetworkX [1] to generate complex networks. We set the randomness seed to an arbitrary number (31) and our algorithm encapsulates the Barabási-Albert model function to guarantee that all graphs generated are connected like actual μApps.

We set $m = n - 1$ and $m = 2$ to generate the API-gateways (FIG. 32a) and Point-to-point (FIG. 32b) topologies. These values guarantee that the generated graphs have similar structures even with varying nodes (microservices) and edges (affinities).

For the generated graph, we labeled the microservices and hosts (vertex of the graph) with their resource usage; for microservices and their capacity, to the host. We used the resource usage of microservices with a uniform distribution: CPU over a 1 – 500 millicores interval and memory in the rage 10 – 500 MB. Hosts have 4000 millicores and 8GB of memory. Based on experiments performed on real μApps and the analysis of deployment files of different open-source μApps available in the Internet, these values emulate an ordinary configuration of business microservices in a μApp.

The mock evaluation assesses the performance metrics, namely: *time to compute the adaptation plan*, *time to compute each MAPE-K activity*, *time to carry out an adaptation*, *number of saved hosts*, and *resource usage to plan an adaptation*. The *time to compute an adaptation plan* metric measures how long REMaP takes to calculate 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 needed to calculate the adaptation plan. We use the *impact of a component on the calculation of adaptation plan* metric to identify the impact, in percentage, of each REMaP component in the total adaptation time.

The metric *time to carry out the adaptation plan* measures how long the μApp management tool (e.g., Kubernetes) takes to carry out the adaptation plan on the μApp. The *number of saved hosts* metric shows us how many hosts we can saved in a μApp deployment after carrying out the adaptation plan computed by REMaP. Finally, the *resource usage to plan an adaptation* metric measures the CPU and memory utilization needed in order to calculate an adaptation plan. Table 2 shows a summary of the metrics evaluated in the mock experiment.

In the experiments, we used three different planners to compute the adaptation plan: *HBA*, *OA* (see Section 4.5), and a modified version of *OA*. This modified version calculates placement only considering the available resources in the host and microservices affinities, which means that it neither tries to maximize the affinities score in a host nor minimize the number of hosts to be used. Moreover, the modified *OA* and *OA* planners use Z3's bitvecs, a special 16 bits data type for integer numbers, to model data over integers with

[1] <https://networkx.github.io>

Table 2 – Mock experiments metrics

Metric	**Evaluation Objective**
Time to compute the adaptation plan	To measure REMaP's performance
Time to compute each MAPE-K activity	To identify REMaP's bottlenecks
Time to carry out the adaptation plan	To measure REMaP performance
Number of saved hosts	To measure the amount or resources saved by using REMaP

32 bits. This strategy aims to decrease the search space during the calculation since they use a brute-force based strategy.

We conducted the mock experiments three times – changing the planner used to compute the adaptation plan (HBA Planner, OA Planner, and OA-modified Planner) each time. In these experiments we first measured the *monitoring time* of each topology – 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 – the time needed in order to collect 10^5 messages using the proposed configuration, as shown in our empiric experiments. In the mock experiments we varied the number of messages exchanged by the microservices (10^1 to 10^5) and the number of microservices available (10^1 to 10^3) to see how these parameters affected affinity computation. In the second step, we measured the time needed to compute the affinities, namely *analysis time*.

Finally, in the last step we measured the *planning time* by measuring the time needed to compute adaptation plans based on the affinities calculated in the previous step. Moreover, we estimated the number of 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 measuring *execution time*, i.e., the time to move the microservice across different hosts. In this case, we measured the time to move 10^1 to 10^3 microservices using Kubernetes. We chose these numbers to emulate different real μApps, since the size of real μApps usually are not smaller than 10 microservices neither larger than 1000. We choose Kubernetes because it is a widely used management tool. Table 3 shows a summary of parameters and their values used in the mock experiments.

5.2.2 Empirical Experiment

For the empirical evaluation we used a reference μApp named Sock-Shop[2] (FIG. 33) – an open-source e-commerce μApp of a site that sells socks –; widely used to evaluate microservices (ADERALDO et al., 2017). It has 15 microservices, five of which are state-

[2] <https://microservices-demo.github.io>

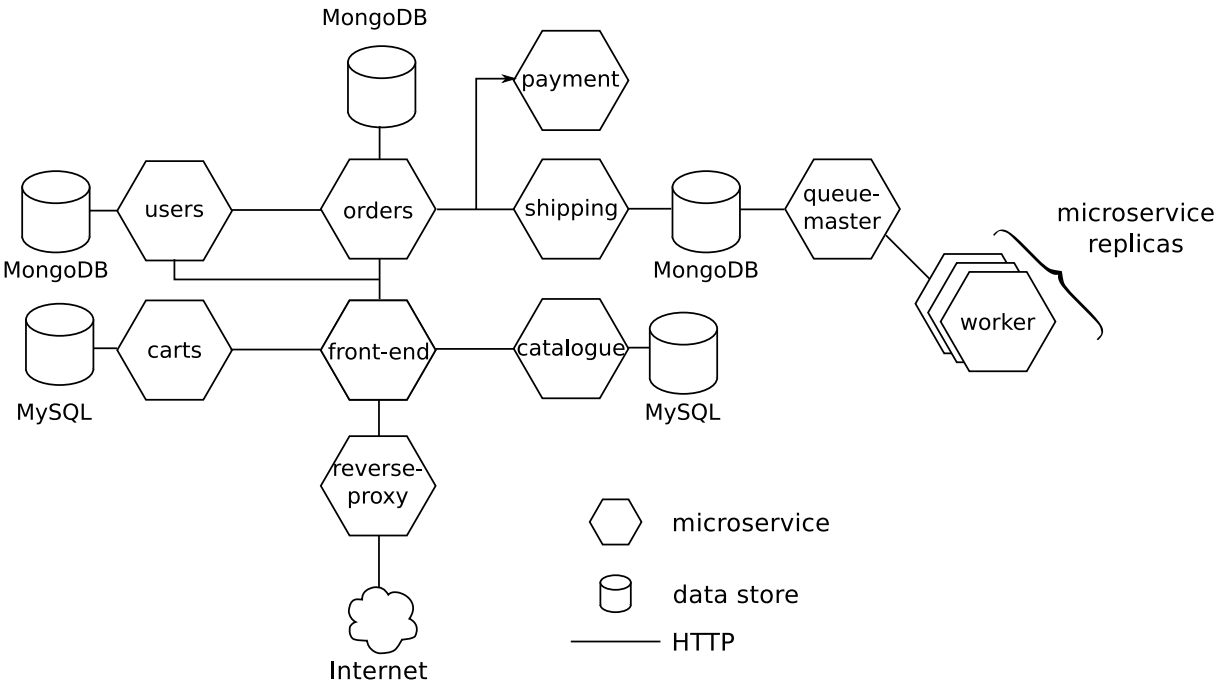
Table 3 – Parameters of the mock experiment

Parameter (Factor)	**Values (Levels)**
Planner	HBA, OA, 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

ful (databases). The microservices are implemented in Go, Java, and Node.JS, and the databases are MySQL and MongoDB.

Sock-Shop was deployed on Kubernetes one microservices per Pod and configured on top of Basic A4 VMs in Azure[3]. 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. We reach this by using the load generator[4] provided by Sock-shop's developer to evaluate the performance of the application. We reach a saturation of the front-end with 10000 messages and 100 clients.

Figure 33 – Sock-Shop architecture



Source: <https://github.com/microservices-demo/microservices-demo.github.io/>

The empirical evaluation focused on the performance metric *round trip time commu-*

[3] <https://azure.microsoft.com>
[4] <https://microservices-demo.github.io/docs/load-test.html>

nication. This metric measures how long the μApp takes to perform an activity, passing through several microservices, and returning a response to the client. Table 4 summarizes the metrics used in this evaluation.

Table 4 – Metrics of the empirical experiments

Metric	**Objective**
Round trip time communication	To measure the impact of the adaptation on the μApp
Number of saved hosts	To measure the amount or resources saved by using REMaP
Resource usage to plan the adaptation	To measure REMaP performance

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 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:

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 5.

Table 5 – Parameters of the empirical experiments

Parameter (Factor)	**Values (Levels)**
Planner to compute the placement	Kubernetes, Planner HBA, and Planner 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 instrumented, partially instrumented

5.3 RESULTS

The results show that, in general, REMaP with *HBA* planner is the best choice for μApps bigger than 20 microservices. The *OA* planner is better for ordinary μApps– smaller than 20 microservices. Finally, the adaptation proposed by REMaP on Sock-shop improves the performance of the μApp in 7%. Next, we present details of the results achieved.

5.3.1 Mock Evaluation

We organized the mock evaluation based on the metrics shown in Table 2. Next, we discuss the results of each metric evaluated.

5.3.1.1 Time to compute an adaptation plan

FIG. 34, FIG. 36, and FIG. 37 present the time to compute an adaptation plan. FIG. 34 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 (see FIG. 34). 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 Equation 4.1). 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 4.5, 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. 35a. The use of planner *OA modified* has a better performance to compute an adaptation plan than planner *OA* (see FIG. 36 and FIG. 37), 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 seconds. Furthermore, as shown in FIG. 35b, 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 microservices. In the experiments, we set a timeout of 10 minutes 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 FIG. 34, FIG. 36, and FIG. 37. 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. 38 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. 34), the executor has the longest duration as shown in Table 6. The executor takes around five seconds to move up to 100 microservices that is a reasonable size for a μApp, and up to 60 seconds to move 1000 microservices (large μApp). The total time for computing an adaptation plan takes around 0.03 and 3.06 seconds 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 seconds with 20 microservices (see FIG. 36).

When compared to the time to compute an adaptation plan using planner *HBA*, the *executor* starts to take a long time (more than 10 seconds) to reconfigure μApps with 200 microservices. This time increase linearly, and the *executor* can take up to 60 seconds to reconfigure a μApp with 1000 microservices (see FIG. 38).

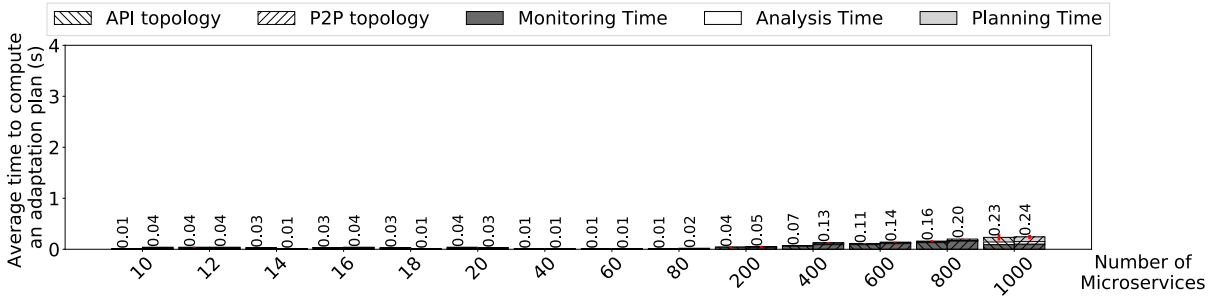
5.3.1.2 Number of hosts saved

FIG. 39 shows the number of hosts saved by the adaptation process considering various topologies and configurations (i.e. different number of microservice instances, hosts, and exchanged messages).

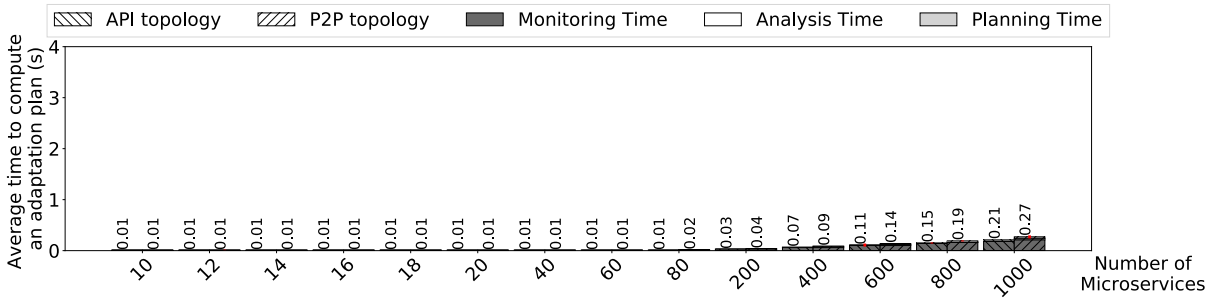
In the experiments, planner *OA* only computes optimal placements for μApps having fewer than 20 microservices. In these cases, planner *OA* can save up to 85-90% of the

Figure 34 – Time to compute an adaptation plan - HBA Planner

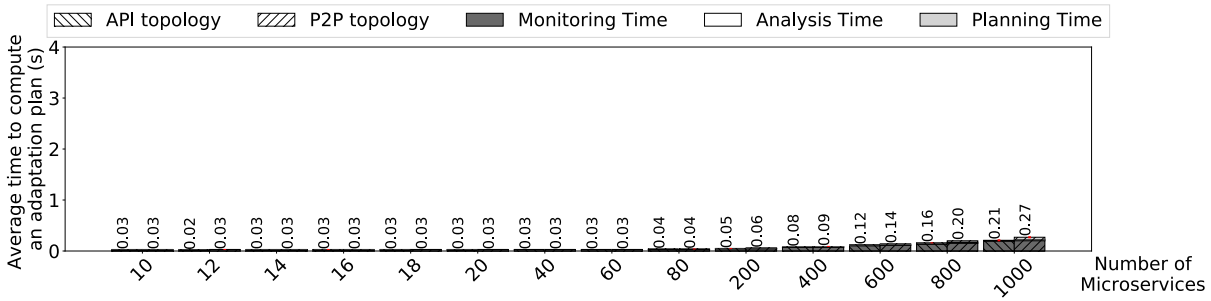
(a) 10^1 messages



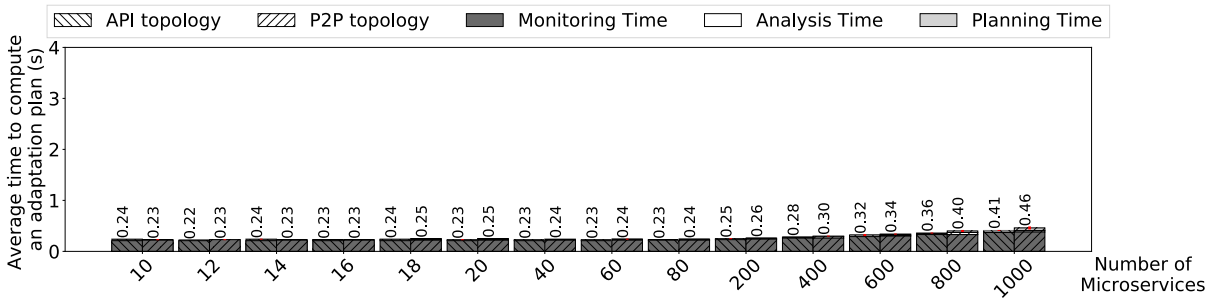
(b) 10^2 messages



(c) 10^3 messages



(d) 10^4 messages



(e) 10^5 messages

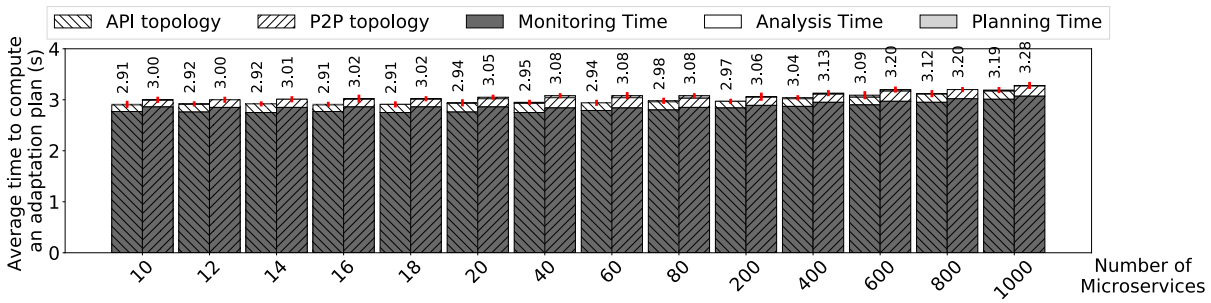
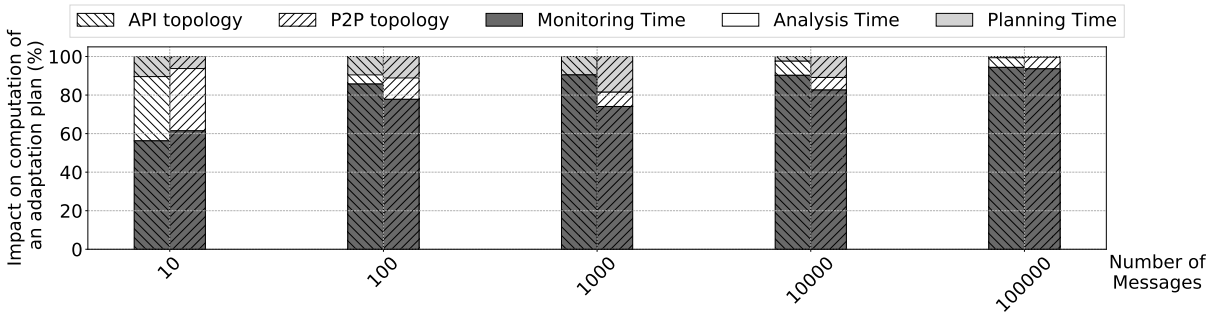
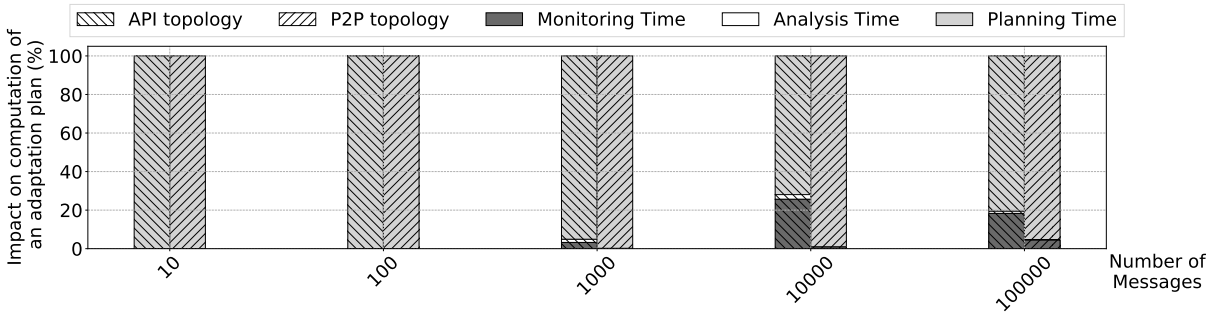


Figure 35 – Percentage of time of each activity of REMaP

(a) HBA Planner



(b) OA Planner



(c) OA-Modified Planner

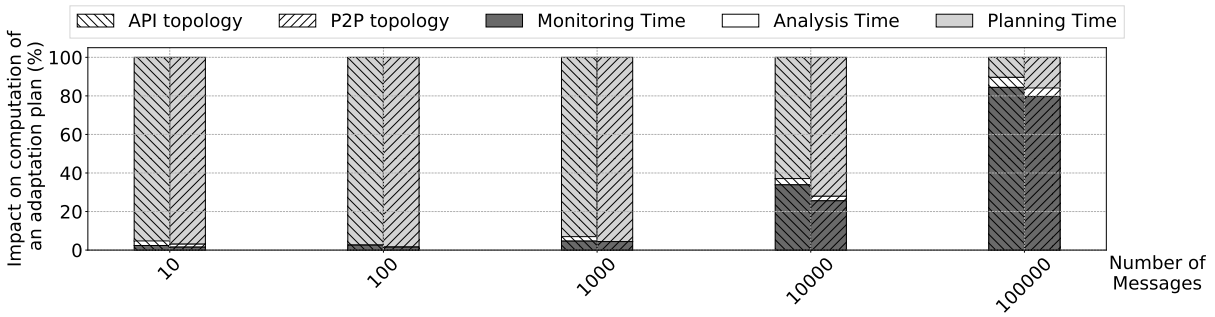
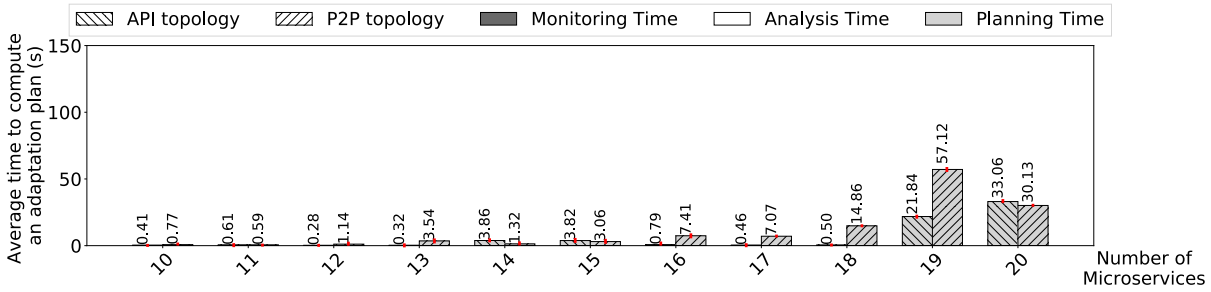


Table 6 – Time comparison for computing an adaptation plan (Planner HBA) and executing it on the cluster.

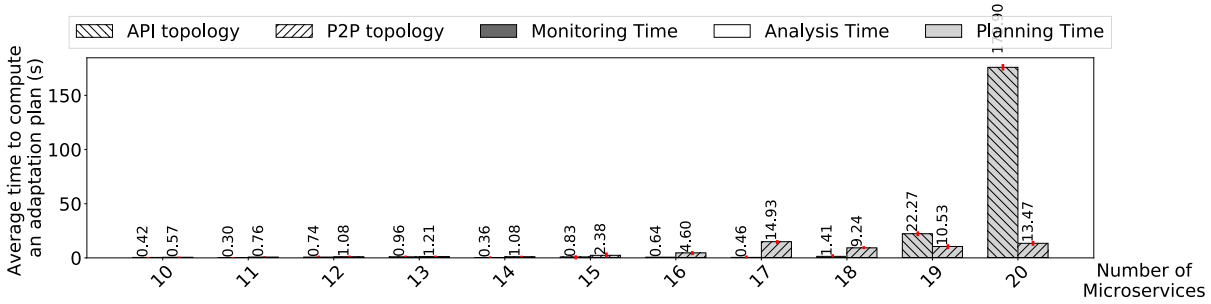
#μServices	Computing adaptation plan (s)			Executor (s)
	10^3 msgs	10^4 msgs	10^5 msgs	
200	0.05 - 0.06	0.25 - 0.26	2.97 - 3.06	5
1000	0.21 - 0.27	0.41 - 0.46	3.19 - 3.28	60

Figure 36 – Time to compute an adaptation plan - OA Planner

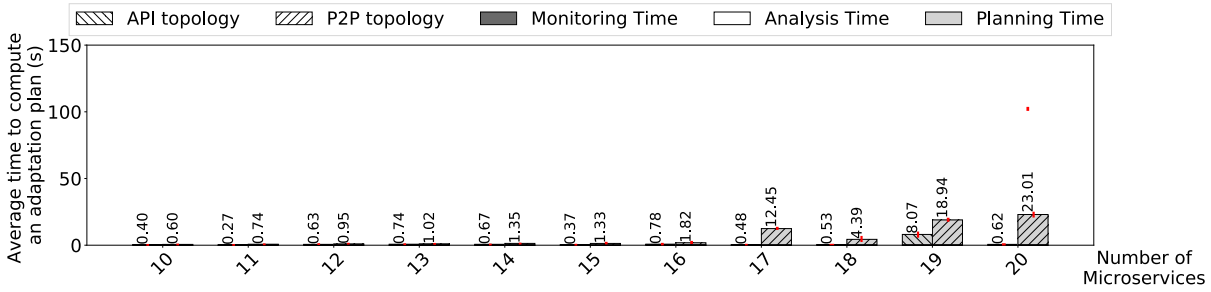
(a) 10^1 messages



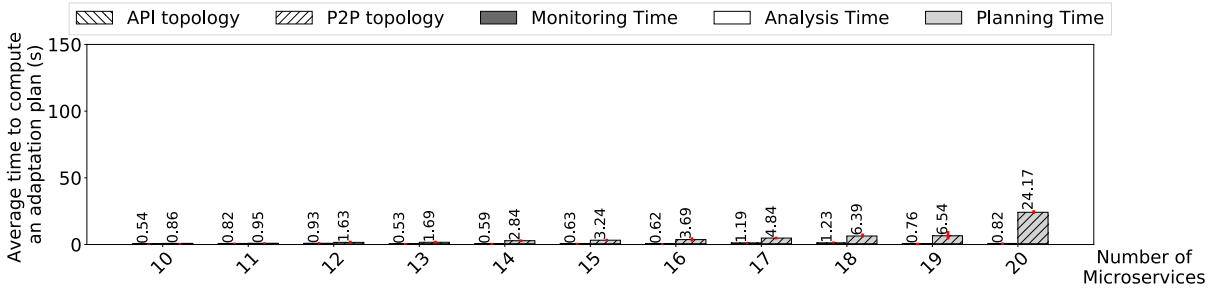
(b) 10^2 messages



(c) 10^3 messages



(d) 10^4 messages



(e) 10^5 messages

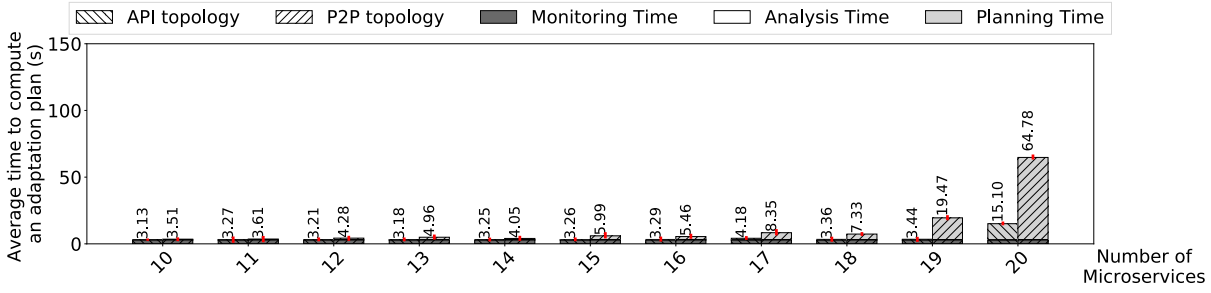
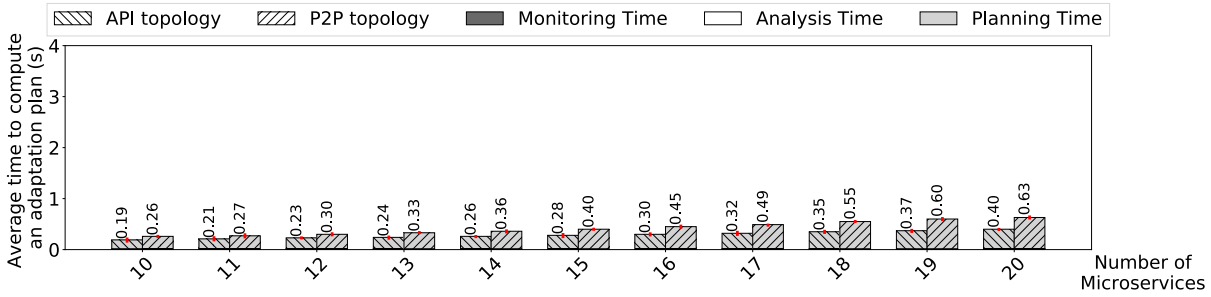
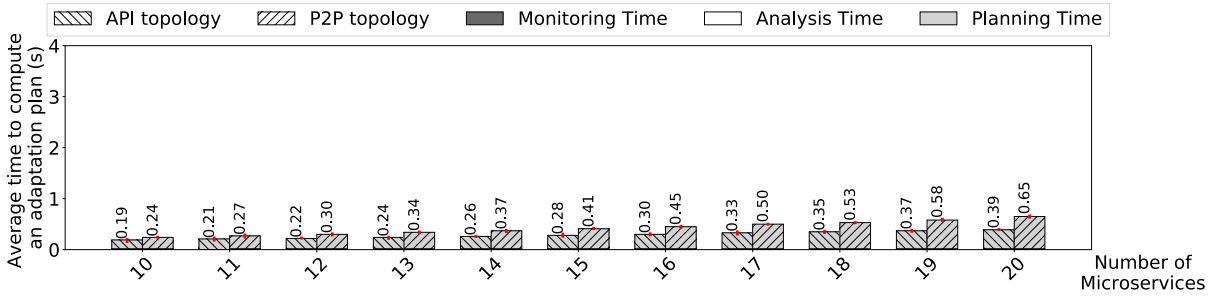


Figure 37 – Time to compute an adaptation plan - OA-modified Planner

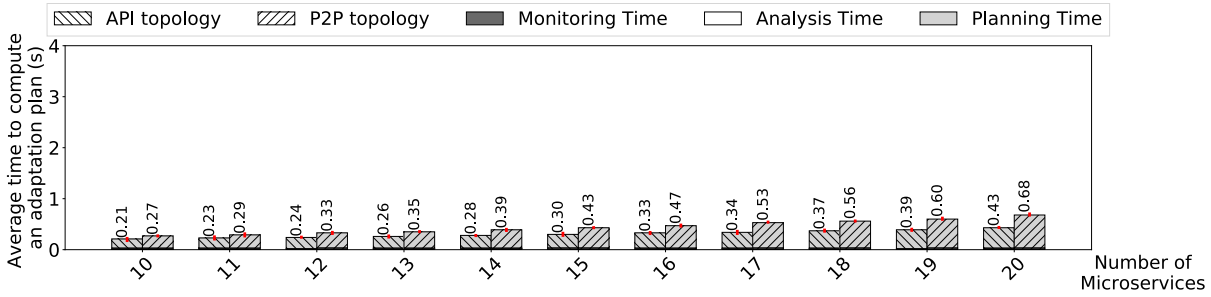
(a) 10^1 messages



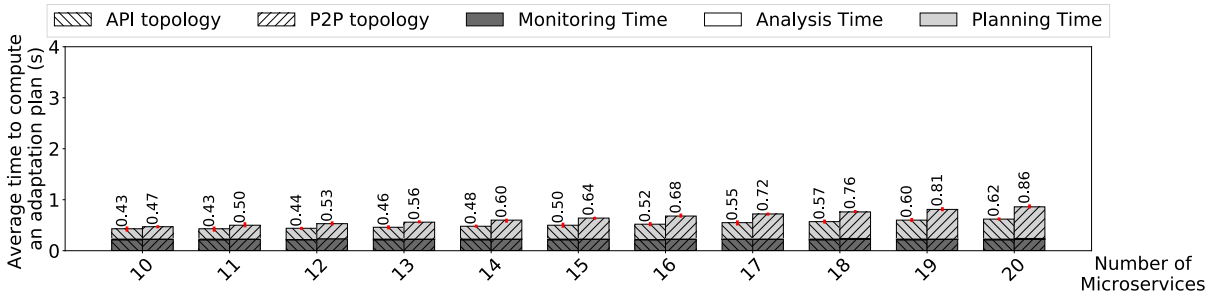
(b) 10^2 messages



(c) 10^3 messages



(d) 10^4 messages



(e) 10^5 messages

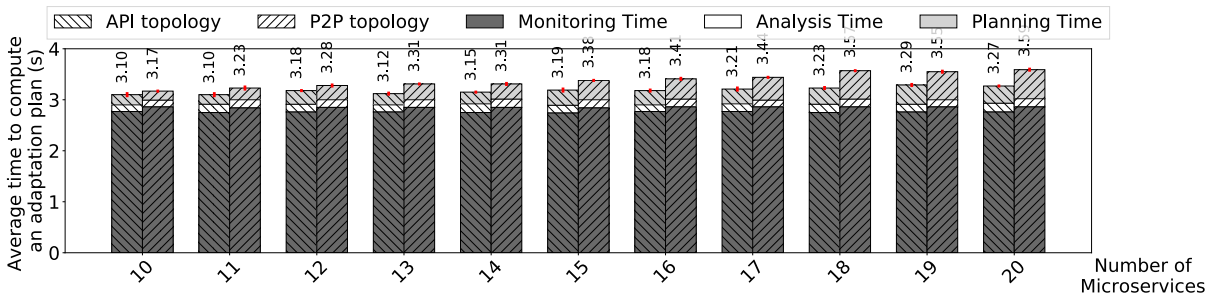


Figure 38 – Average time to move microservices using Kubernetes.

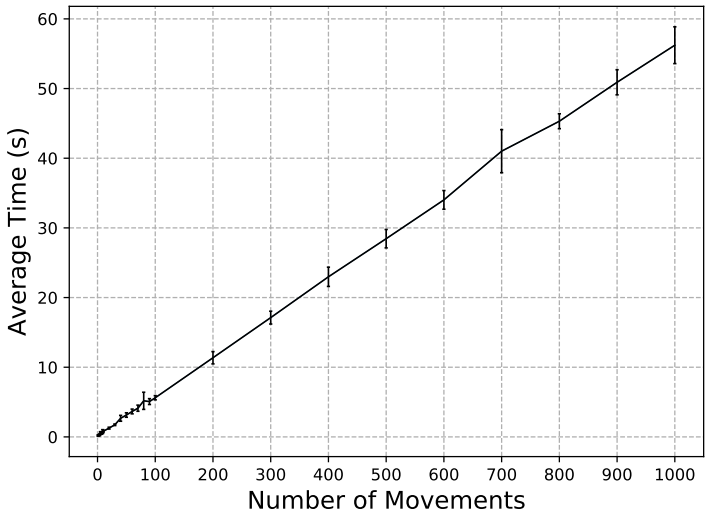
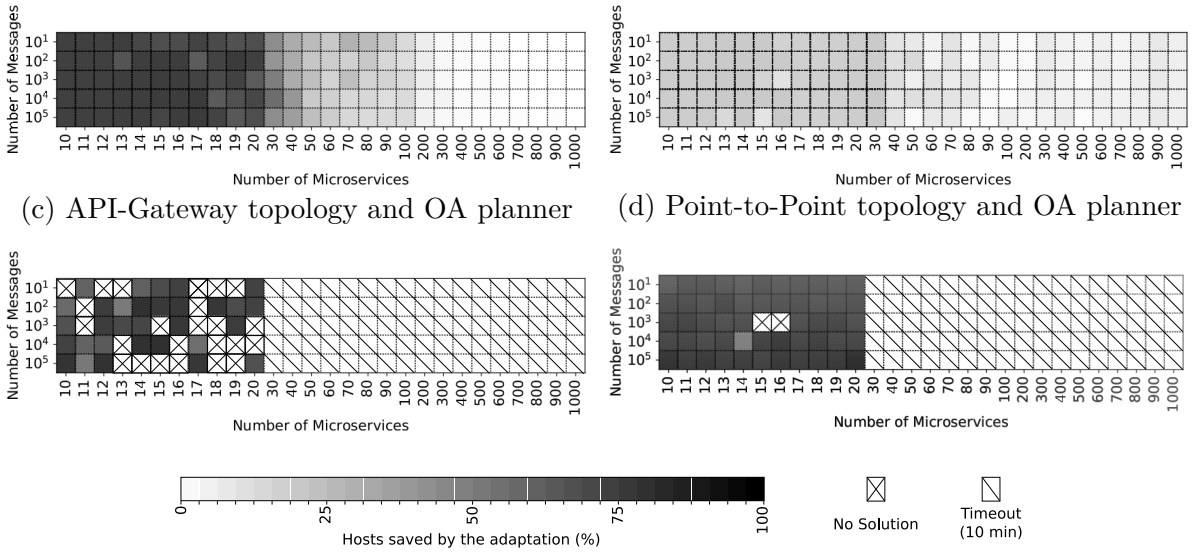


Figure 39 – Saved hosts

(a) API-Gateway topology and HBA planner (b) Point-to-Point topology and HBA planner



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.

FIG. 39 show that the topology affects the computation of the placement optimization. As shown in FIG. 39a and FIG. 39b, 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.

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. 39c. 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. 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 minutes). In FIG. 39c and FIG. 39d, 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 minutes.

FIG. 39c and FIG. 39d show that the topology affects the computation of the placement optimization. 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. FIG. 39c and FIG. 39d 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.

5.3.2 Empirical Evaluation

We organized the mock evaluation based on the metrics shown in Table 4. Next we result the results of each metric evaluated.

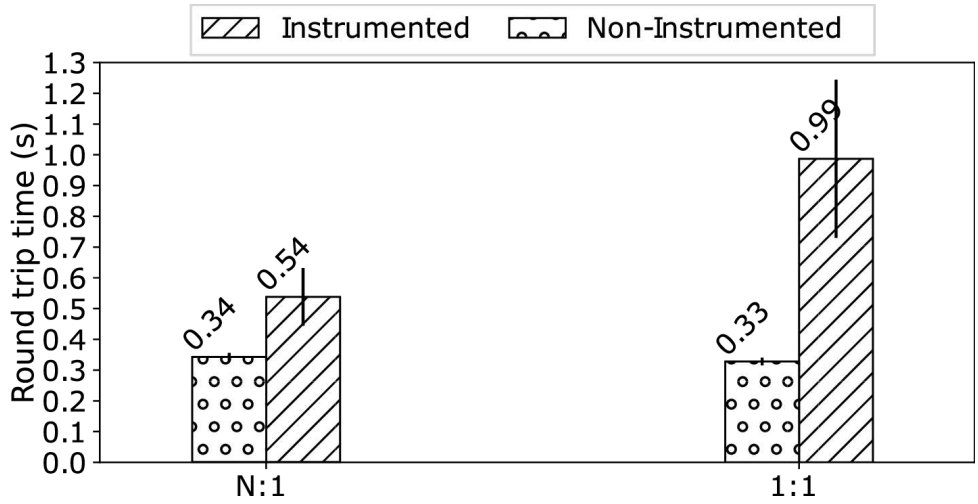
5.3.2.1 Impact on a real μApp

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

As expected, the instrumented version of Sock-shop placed without optimization has a worse performance than the non-instrumented one as shown in FIG. 40.

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 observed an overhead of approximately 58% on the RTT time when compared to the non-instrumented version (see FIG. 40). 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. 40). 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.

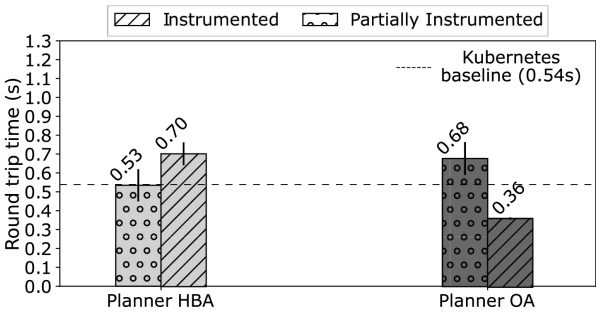
Figure 40 – Sock-shop instrumented versus non-instrumented (Kubernetes).



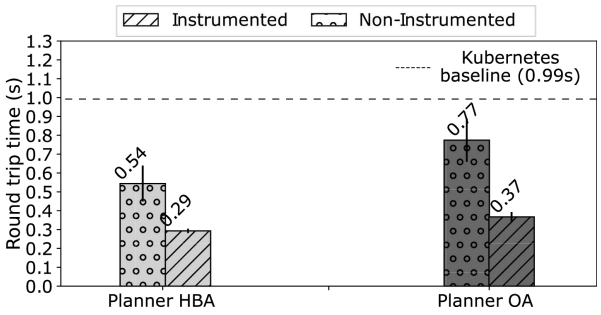
μApps are highly distributed and observing their runtime behaviour is a hard task. The instrumentation of μApps is necessary to observe and track the behaviour of the μApp at runtime, which is required for reliable management. Hence, we used the instrumented version of Sock-shop configured with Kubernetes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

Figure 41 – RTT comparison when Sock-shop is fully instrumented and non-instrumented.

(a) N:1 Deployment



(b) 1:1 Deployment



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. FIG. 41a and FIG. 41b 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.

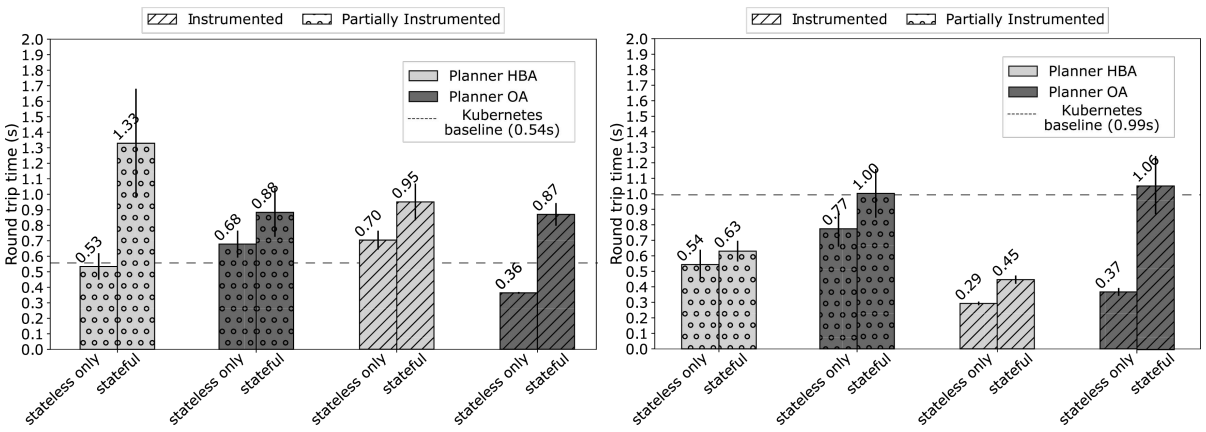
The previous experiments only optimized the μApp by migrating stateless microservices. However, in some scenarios, stateful microservices should be moved. Currently, REMaP cannot sync data after the migration. However, it is possible to evaluate the impact of co-locating I/O bound microservices. The 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. 42), 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. 42b.

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. 33), each 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.

Figure 42 – RTT comparison when optimization is applied considering the migration of stateful microservices or stateless only.



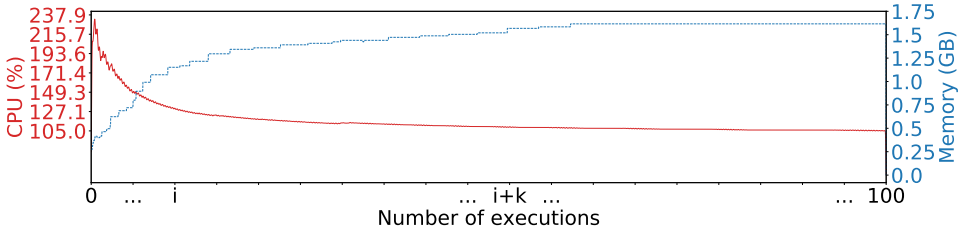
In addition to the impact of different planners on the μApp performance, we also evaluated the number of resources they can save in the cluster and compared it to the Kubernetes deployment. Table 7 shows these comparisons. We can observe that deployment in 1:1 saved more hosts. Kubernetes attaches a 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 itself until all microservices have been attached or there are no more resources available.

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

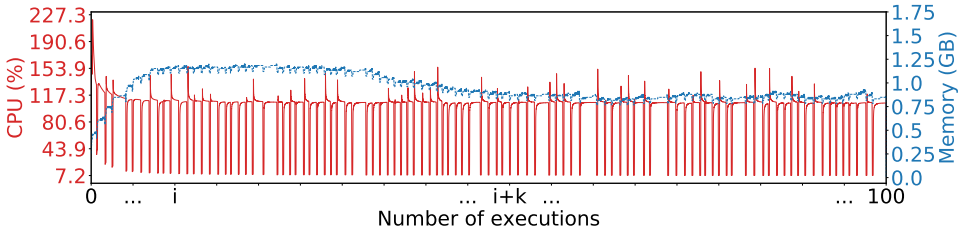
Finally, planner OA guarantees an optimal placement and improves the performance of the μApp in 1:1 deployment. However, in the N:1 deployment, the performance is worse

Figure 43 – Resource consumption

(a) OA Planner



(b) HBA Planner



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

Table 7 – (#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%)

5.3.2.2 Resource consumption of REMaP

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. 43. We did not evaluate planner *OA modified* due to its poor results for saving resources during mock experiments.

Planner HBA consumes approximately 1.65GB of memory (see FIG. 43a) and planner OA consumes 0.8GB (see FIG. 43b). 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, in contrast, is implemented in Java. According to (PEREIRA et al., 2017) Java is approximately 4.5 times more memory consuming than C++ and 2.15 times 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 an optimal solution, and its memory usage grows exponentially according to input. Planner HBA, on the other hand, has memory usage that is polynomially limited 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 5.3.1.1, 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.

In contrast, planner HBA (FIG. 43a) 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.

5.4 SUMMARY OF RESULTS

We conclude that for ordinary μApps (up to 20 microservices) the OA planner is a good choice to be used since it can minimize considerably the number of hosts used during deployment. For larger μApps, HBA planner is a better choice. OA planner modified is not useful at all. 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.

In general, the time needed to collect data of a cluster (Monitoring time) is high, making it necessary to apply optimizations to REMaP's monitoring component in order to improve this phase. Moreover, the planners are liable to receive optimizations on their

implementations.

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 undershooting, are essential.

The HBA planner may improve its results by mixing evolutionary algorithm techniques with the current implementation. The OA planner, in turn, can be optimized by refactoring data types used for encoding the problem instance. Initially, OA planner used ordinary integers and floats of 32 bits to encode the input. In this scenario, we were able to calculate adaptations for μApps with up to 12 microservices. Changing the integers to a bitvec of 16 bits (a particular data type of Z3) we improved the results by calculating adaptations for bigger μApps (20 microservices). Therefore, we believe that by making more optimizations – replacing the data types, used in the code, for optimized versions – we can use OA planner on larger μApps.

5.5 CONCLUDING REMARKS

In this chapter we presented the evaluation of REMaP. First, we presented our mock evaluation, showing the performance of REMaP in saving resources. Next, we presented the empirical evaluation of REMaP, showing the impact of REMaP on a real μApp and how our approach can improve the performance of the μApp by reorganizing the placement of microservices according to their resource consumption and their relationship in the application. Finally, we presented the results obtained. Our results show that the change in placement of microservices based on their behavior can improve the performance of a μApp while the resources usage in the cluster is decreased.

6 RELATED WORK

This chapter positions our work concerning state-of-art runtime adaptation of microservices. This thesis was born from a study to support microservice evolution. Hence, we start by settling our work concerning the evolution of software architecture. Then, we overview several works and classify them into five categories: supporting microservices evolution, Models@run.time on service domains, runtime adaptation of μApps, usage of models in microservice domain and placement on clouds. We deem that these categories are the best to classify the work done in this thesis.

6.1 SUPPORTING MICROSERVICE EVOLUTION

Evolving architecture has been researched since the notion of software architecture first emerged. An important approach in this area combines static analysis, dependency modeling and architectural evolution – proposed by Sangal et al. (2005). Our approach is close to this one, but it targets the microservice's domain, which is dynamic and requires runtime analysis.

Work on microservices. There is an increasing interest in applying techniques from software engineering (RAJAGOPALAN; JAMJOOM, 2015; HEORHIADI et al., 2016; TARVO et al., 2015), formal methods (PANDA; SAGIV; SHENKER, 2017), and self-adaptation (HASSAN; BAHSOON, 2016; TOFFETTI et al., 2015; FLORIO; NITTO, 2016) communities to microservices. Our proposal integrates these techniques and uses models at runtime to support the evolution of microservices. The use of models abstract the heterogeneity of microservice domain and bring flexibility for making decision to evolve the μApp. Moreover, the use of models help to addresses additional evolution-related maintenance tasks such as deployment and architectural refactoring.

Modeling. Dependency modelling of services is an established topic (ENSEL, 1999). Most recently, Düllmann and van Hoorn described a top-down approach to generate a μApp from a model (DÜLLMANN; HOORN, 2017). By contrast, we propose a bottom-up approach that is closer to the work of Leitner, Cito and Stöckli (2016) and Brown, Kar and Keller (2001). However, both these approaches only use network interactions between services to generate models and do not model microservice evolution.

Log analysis. The use of logs is an accessible means of monitoring and analyzing software, particularly in the cloud (OLINER; GANAPATHI; XU, 2012). The state-of-the-art log processing can reconstruct applications by analyzing high throughput and real-time data (ZIPKIN, 2017; CHOTHIA et al., 2017). Our work relies on these state-of-art tools.

Supporting microservice evolution. Version consistency has been considered for runtime reconfiguration in distributed systems (MA et al., 2011), fault-tolerant execu-

tion (HOSEK; CADAR, 2013) and in other domains. Our work, perform upgrade consistently checking the adaptation plan against the model in order to guarantee safe adaptations.

Deployment trade-offs. Existing works consider deployment trade-offs in general distributed systems (those different to μApps) (GUERRAOUI; PAVLOVIC; SEREDINSCHI, 2016). In particular, Tarvo et al. (2015) described a monitoring tool to support canary deployment and Ji and Liu (2016) presented a deployment framework that accounts for SLAs. However, we are interested in connecting evolving software engineering concerns with deployment trade-off.

6.2 MODELS@RUN.TIME ON SERVICES DOMAIN

Service-based applications are usually developed by using WS-BPEL (JORDAN; EVDEMON, 2007), a modeling language meant to orchestrate services by specifying business processes through service compositions. This characteristic, of using WS-BPEL to define the behavior of applications, makes service-based applications naturally rely on Models@run.time. This characteristic arises by the usage of WS-BPEL, so that changes to the model – created by this language – are reflected on the execution of the application. Next, we present some works that expand the modeling capacities of the WS-BPEL and bring more dynamic to the execution of their models.

(CARDELLINI et al., 2009) proposes a methodology named MOSES in order to drive the adaptation of service-oriented applications to achieve greater flexibility in facing different environments and the possibly conflicting QoS requirements of several concurrent users.

MOSES uses a model to maintain concerns of the composition manager and adaptation manager. The composition manager has as a goal the specification of the business process. The adaptation manager is to determines which concrete services should be used for each abstract services and how they should be used to meet QoS requirements in a volatile environment.

Moreover, an adaptation policy model is used to guide the runtime adaptation of the system. At execution time, a MOSES optimizer uses these models to select the best services in a given environment that satisfy the QoS policies set.

Similar to REMaP, MOSES uses the model to store information about the system as well as its environment and uses them to calculate an optimization of the system at runtime. However, MOSES guides the adaptation by using user-defined policies while REMaP performs the adaptation autonomously.

Nguyen and Colman (2010) presents an approach for Web service customization that helps to reduce the complexity of the customization process and enables the automated validation of customized services.

The service provider generates a feature model at runtime which is used as part of the service description in such a way that interested consumers can make searches on

the model looking for features. The consumer selects desired features from the model generating a feature configuration. This information is exchanged between the consumer and service provider; based on this information, the service provider generates a particular service instance binding with the service's interface.

The approach of Nguyen and Colman uses the causal connection to maintain the feature model in conformance with the executing service. Moreover, the model is used to generate bindings used at runtime to connect services. REMaP uses the causal connection to maintain the model in conformance with the running system (μApp), which is essential to guide the adaptation. Unlike Nguyen and Colman, REMaP does not generate code based on the model; instead, it derives new information from parts of the model (affinities) to be used during the adaptation.

Karastoyanova et al. (2005) addresses the problem of dynamic binding of Web services instances at runtime by extending WS-BPEL, adding support to dynamic lookup. The authors propose to move the specification of service selection from deployment time to execution time in order to standardize the selection in a cross-platform manner.

The idea behind Karastoyanova et al. proposal is to find a list of available Web services compliant with the service type associated with the business process' activity, then, to select a service from that list – according to user-defined selection criteria, such as QoS and semantics. Finally, to bind the operation of the process instance to the selected service. This service performs a task on behalf of the process instance.

Unlike REMaP, this approach uses Models@run.time to specify and bind services independent of the execution engine used to process WS-BPEL models. Microservices are neither specified nor integrated through languages as proposed by REMaP.

Ma et al. (2013) proposes a model-based approach to runtime manager of service composition which makes the administration of services more user-friendly.

Ma et al. built a system meta-model of service compositions by specifying what kinds of elements can be managed. Then, they built the access model of a service composition by defining how to manipulate the manageable elements in order to monitor and modify them. With the meta-model, they generated the synchronization engine and accessed the built model. Finally, they utilize model-based techniques to monitor, check constraints and perform control actions on the service composition based on the runtime model.

Like REMaP, this approach uses Models@run.time to monitor and check parts of the underlying system. However, REMaP neither uses the model to control the μApp nor compose the microservices at runtime – the model is used to guide the adaptation process, checking if an adaptation action is safe to be applied.

eFlow (CASATI et al., 2000) was developed to support adaptive and dynamic service compositions. The eFlow model enables the specification of the business process, that can automatically configure themselves at runtime according to the nature and type of services available on the Internet.

The dynamic change features provided by eFlow allow great flexibility in modifying service process instances and service process definitions. The engine processes the model and schedules the services to be activated according to the process definition.

FARAO (WEIGAND; HEUVEL; HIEL, 2008) is a rule-based service composition to increase the adaptability of service orchestration. FARAO supports the development of adaptable service orchestrations by providing a flexible interface to a service manager. Given a set of services to be orchestrated, the designer retrieves the interface and data descriptions from the registry. These descriptions are derived from rules to manage the data flow. The rules are extended with business rules, that steer the decisions in the orchestration. Finally, these rules are delegated to the service manager that executes them autonomously or semi-autonomously.

Like REMaP, eFlow and FARAO use the model to customize some aspects of the application execution. In general, they use the data in the model to customize the execution of the application. REMaP, on the other hand, uses the model to guide the adaptation mechanism that updates the placement of microservices and consequently changes the behaviour of the application.

6.3 RUNTIME ADAPTATION OF μAPPS

Microservice practitioners already identified the need for automatic tools to manage μApps (NEWMAN, 2015). However, few works have been developed aiming for this objective. Most of the μApp adaptations are related to automatically scaling in/out microservices. Next, we present the most relevant works on this subject.

App-Bissect (RAJAGOPALAN; JAMJOOM, 2015) is a semi-automatic mechanism used to detect performance issues and apply self-healing to the application, reverting microservices to a previous version. It is a system-level application-agnostic tool that operates in the unsupervised model.

To work correctly, App-Bissect carries out a dependency analysis of a manifest file, created by the developers, which informs which are the dependencies of each microservice (another microservice and version required). With this information, App-Bissect monitors the performance of μApp execution. When it detects that the μApp performance drops after an upgrade, i.e., the deployment of new microservice version, App-Bissect rollbacks (part of) the application to the prior configuration. The rollback is applied only to the new microservice that come up as well as their dependencies, avoiding to rollback the entire μApp. The application owner limits the rollback, which defines the lower bounds, known as global restore points, where the application can rollback.

App-Bissect and REMaP address the performance of the μApp. However, App-Bissect only works on the application and does not change the placement of their microservices. However, in some situations, a better placement of a new version of a microservice could

avoid the need to rollback the μApp. Therefore, the strategies of App-Bissect and REMaP can be presented as complementary concerning μApp performance.

Gru (FLORIO; NITTO, 2016) is a decentralized autonomic manager – running over Docker engine – that monitors CPU, memory and response time in order to automatically scale out microservices.

Gru architecture is based on agents, deployed into cluster nodes in order to collect metrics such as CPU and memory usage. Each GRU-Agent implements a MAPE-K based autonomic loop that decides locally about microservices resource consumption within a node. The control is triggered periodically, with a user set time interval. The data collected by agents on each node is analyzed and combined with the analysis results of other peers in order to compute the average of all data and a partial view of the status of the system. Then, the planner decides which action to take (scale in or out) according to the partial view of the system – computed by the analyzer and defined user policies. Finally, the executor interacts with the Docker daemon to execute the choen actions.

Gru and REMaP have different implementations of the MAPE-K architecture. Gru is decentralized wheras REMaP is centralized – both with their pros and cons, as mentioned in Section 2.1.2. Currently, both Gru and REMaP are limited to carry out a single kind of adaptation – auto-scaling and smart-placement. Both were designed to allow the expansion of adaptation strategies. However, Gru is more flexible and allows extensions through policies while REMaP needs new implementations of their components plugged into the adaptation manager.

Toffetti et al. (2017) shows a case study of the self-management of cloud-based applications (microservices). The authors propose an architecture that leverages on the concepts of cloud orchestration and distributed configuration management that enable self-management of cloud-based applications.

The authors use distributed storage and leader election functionalities – commonly available in the current cloud applications – to deliver a resilient and scalable managing mechanism that provides health management and auto-scaling functionality for μApps. The key design choice in this proposal is the resilience of the management architecture. In case the self-healing and auto-scaling fail, due to crashes, they can be restarted on any node by collecting shared state information through the configuration management system.

This proposal addresses the resilience of the management tool, which is not a concern of REMaP. Moreover, Toffetti et al. uses the management tool for self-healing and auto-scaling of μApps, while the REMaP interest is on the smart placement of microservices.

JRO (GABBRIELLI et al., 2016) is a mechanism for self-reconfiguring microservices based on a description language named Jolie. In their approach, the authors can conduct an optimal deployment of an application based on the requirements annotated in its specification.

JRO addresses the problem of optimal automatic deployment of microservices based on their resource needs. The user defines the requirements of the deployment – e.g., number of microservices instances, and what can or cannot be co-located. Then, JRO retrieves the context of the environment, such as available VMs, and computes an optimal deployment plan based on the user's requirements and resources available resources. Finally, if the user agrees with the solution plan, JRO proceeds with the deployment – orchestrating how the services should be deployed; linked or removed.

Despite the fact that both JRO and REMaP aim to compute an optimal placement of μApps, their execution is different. JRO is semi-autonomous,and needs human interventions to decide the requirements of the μApp and if the optimization calculated is good enough to be applied. REMaP, on the other hand, makes this decision automatically based on the workflow history of the application. REMaP computes the affinities of the μApps and which microservices can or cannot be co-located on cloud nodes. When an optimal configuration is found it is guaranteed that the requirements are satisfied and it can be applied to the cloud.

Hassan and Bahsoon (2016) proposed a conceptual approach to use a self-adaptive mechanism in order to address design trade-offs of evolving monoliths to microservices. In their proposal, the self-adaptive mechanism should be able to handle a number of microservices as well as their non-functional requirements automatically.

Hassan and Bahsoon discuss the challenges of applying MAPE-K on microservices and suggest several ideas to overpass these challenges in each phase of the MAPE-K loop. For instance, monitoring variables that reflect the μApp and its environment limits the search space of the planning phase and intensifies the use of artificial intelligence (AI) on planning and execution.

REMaP is inspired by many suggestions given by Hassan and Bahsoon. However, REMaP does not use AI in its current implementations.

Kang et al. (2016) use optimization techniques to save energy consumption of μApps based on the analysis of resource utilization patterns of microservice requests. The authors propose a broker to forecasts future demands of the μApp requests and route the requests to container replicas at runtime – in such way that the μApp consumes minimum energy as possible.

The approach relies on the implementation of a brokering system that classifies input requests based on their resource utilization pattern. The requests are then assigned to each desirable resource in multiple server racks – i.e., containers with different resources available. The classification is made by using *k-medoid* algorithm (KAUFMAN; ROUSSEEUW, 1987) which is a well-known clustering scheme.

Similar to REMaP, Kang et al. uses the communication between microservices (containers) to optimize the behaviour of μApps. The authors use the requests to optimize the energy consumption in a cluster while REMaP uses messages between microservices to

optimize their deployment. The runtime adaptation proposed by Kang et al. merely relies on routing messages and does not change the placement of any microservices. REMaP, on the other hand, updates the arrangement of several microservices moving them across the cluster.

Table 8 summarizes the use of runtime adaptation on μApps.

Table 8 – Use of runtime adaptation on μApps.

Proposal	**Usage**	**Input**
(RAJAGOPALAN; JAMJOOM, 2015)	Self-healing	Microservices dependencies and Static files
(TOFFETTI et al., 2017)	Auto-scaling	Resources usage
(GABBRIELLI et al., 2016)	Self-reconfiguring	Resources usage
(HASSAN; BAHSOON, 2016)	Self-reconfiguring	Non-functional requirements
(KANG et al., 2016)	Energy saving	Energy consumption
REMaP	Deployment optimization	Resources usage and microservices Dependencies

Source – Made by the author

6.4 USAGE OF MODELS IN MICROSERVICE DOMAIN

The foundation of this thesis is based on the use of models to support the evolution of μApps. We use models to keep the evolution of μApps along its execution in order to carry out analysis to improve several aspects of the μApp, especially the placement of microservices on a cluster. Moreover, we leverage the use of models in the microservice domain by using it at runtime steering autonomous adaptation of μApps.

Models@run.time has been underused in the Microservice domain. μApps and their management tools might take advantage of these features to abstract away technologies used to build up and monitor a μApp, as well as a to provide a unified view to inspect the application and its environment. However, models are mostly applied at development time to the automatic generation of code (HASSAN; ALI; BAHSOON, 2017), documentation (GRANCHELLI et al., 2017), architectural analysis (DERAKHSHANMANESH; GRIEGER, 2016) and microservices integration (ZÚÑIGA-PRIETO et al., 2017).

Rademacher, Sachweh and Zündorf (2017) discusses different usages of models in Service-oriented and Microservice Architectures. The authors have identified that models have been used mostly at development/specification time, and to our knowledge, there is

no work using models at runtime in the microservice domain. Despite the lack of works of models at runtime for microservices, Rademacher, Sorgalla and Sachweh (2018) show the challenges in decomposing monolith applications into μApps. They claim that Domain-driven design (DDD) provides modeling means for software design to decompose domains in bounded contexts. Thereby, each identified into a monolith (context cluster) denotes a candidate for a microservice. However, the authors suggest that models are typically under-specified, which poses several challenges when applying a domain driven design to create the μApps. Finally, they presented selected tools of model-driven development to cope with such problems.

Development support. Most of the works on models for μApps aim to support the development of applications and offline tasks such as static analysis for architectural evolution (HASSAN; ALI; BAHSOON, 2017) and support for deployment (LEITNER; CITO; STÖCKLI, 2016).

Derakhshanmanesh and Grieger (2016) identifies how models might be used across the full software lifecycle, including runtime, to evolve the application. Derakhshanmanesh and Grieger proposes the usage of a domain-specific modelling language (DSML) and model transformations to define and evolve a microservice application at the architectural level.

Rademacher, Sachweh and Zündorf (2018) presents a model for creating μApps. The concepts adopted in this model are deduced from existing approaches to SOA modelling. Moreover, it adds frequent concepts from μApps, such as API gateways. The model was structured in different viewpoints like in REMaP. However, their viewpoints comprise the data, services, and operations of a μApp wheras REMaP comprises different abstractions of a μApp, such as nodes, applications and microservices.

Hassan, Ali and Bahsoon (2017) proposes a mechanism, named Microservice Ambients, to support the evolution of monolithic architectures for microservices at development time.

Leitner, Cito and Stöckli (2016) present an approach for model deployment cost. The model is generated by analyzing messages exchanged among microservices and creating a direct acyclic graph (DAG). The proposed algorithm, namely CostHat, is applied to the graph in order to calculate the costs generated by IO, compute, API calls, and other constant cost factors in a μApp.

Code generation. Among all offline tasks supported by models, code generation is the most used. Düllmann and Hoorn (2017) have used models for engineering microservice testbeds. The model is created by developers and generates Java code and deployment files (Maven, Docker, and Kubernetes). The microservices generated are automatically instrumented to collect metrics at runtime.

MicroArt (GRANCHELLI et al., 2017) is a tool to perform static and dynamic analysis of code repository and microservice environment (Docker). These analyses are guided by a software architect and create an architectural model of the application. The model is

used to show the application graph, the development team and the cluster where the application is deployed. Nothing about runtime metrics and messages are represented in this model. Also, this model is not used at runtime.

Sorgalla et al. (2018) proposes a collaborative model for development teams creating μApps. The model is a first-class citizen on development, not restricted only for specification but being used to generate code and integrate microservices. However, the proposed model does not update/upgrade the μApp at runtime autonomously.

Integration. Usually, the code generation is carried out to integrate microservices automatically. Zúñiga-Prieto et al. (2017) propose a model, inspired on SoaML (OMG, 2012), to integrate microservices. Developers should specify the integration, describing the integration logic and architectural impact of integration without taking into consideration the particularities of any 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.

Sorgalla (2017) presents AjiL, a domain-specific modeling language (DSML) to create μApps. The language aims to reduce the complexity of the creation of μApps by abstracting development and integration technologies in microservices. AjiL is a graphical language that facilitates the creation of applications by business experts.

Thramboulidis, Vachtsevanou and Solanos (2018) proposes a framework to build cyber-physical μApps by using models. Like REMaP, this approach uses a model to abstract the heterogeneity of the entities that compose the system. In this case, a manufacturing plant of industry 4.0 (LASI et al., 2014). Their platform abstracts these elements and developers then use the model to integrate the elements in the plant detached of their technology. Unlike REMaP, the model is not used for runtime adaptation of the system.

Table 9 highlights the primary usage of these works and compare them with REMaP.

Table 9 – Model usage on Microservice domain.

Work	**Abstraction level**	**Usage**	**Online/ Offline**
(DERAKHSHANMANESH; GRIEGER, 2016)	Architectural	Modeling the μApp Overview	Offline
(RADEMACHER; SACHWEH; ZÜNDORF, 2018)	Architectural	Modeling different μApp Viewpoints	Offline
(HASSAN; ALI; BAHSOON, 2017)	Architectural	Support evolution From monolith app To microservices	Offline
(LEITNER; CITO; STÖCKLI, 2016)	Resource Usage	Support deployment of microservices	Offline
REMaP	Architectural, Dependencies, and Resource usage	Runtime adaptation of microservices Placement	Online

Source – Made by the authors

6.5 PLACEMENT ON CLOUDS

Several strategies have been applied to the allocation of VMs on clouds in order to improve different aspects of the application; such as effectiveness, cost, and QoS (SINGH; PRAKASH, 2015). Similarly, there are several strategies on assigning containers on clouds. However, these works are not interested in the requirements of the μApps but only on system requirements. Hence, REMaP is a novelty in this context by providing a strategy to reorganize the configuration of container clusters based on the workflows of the μApps. Next, we present some of these works concerned on allocation of applications, virtual machines and containers on clouds.

6.5.1 Placement of VMs on Clouds

The placement and runtime migration of VMs based on affinity has been studied by several groups (CHEN et al., 2013; SONNEK et al., 2010; ACHARYA; MELLO, 2013; LEELIPUSHPAM; SHARMILA, 2013; PACHORKAR; INGLE, 2013; LU et al., 2014). In general, these works try to group VMs based mainly on their communication affinity in order to decrease the overhead imposed by the communication latency.

Unlike REMaP, these studies do not detect the real usage of resources, allocating the VMs based on static resource values. Moreover, due to the monolithic nature of the

applications deployed in VMs, it is not common to reconfigure the VM placement in order to improve application requirements since the monolithic application is not as dynamic as a μApp.

Close to our work, Bayless et al. (2017) uses MonoSat, a SMT solver specialized in flow problems. MonoSat optimizes the deployment of virtual clusters, based on VMs and node resources, network bandwidth and requirements of the virtual cluster. Unlike REMaP, the main problem solved by the authors is to maximize the flow of data and not the co-location of services (VMs or containers) into the cluster. The specialization of MonoSat for network flows allowed the achievement of relevant results in comparison to other SAT solvers. However, the features of MonoSat are not useful for bin-pack problem variants like ours, which lead us to use general purpose SAT solvers that are not good for handling the optimization of large cluster allocations.

Finally, in addition to the affinity based on communication, other kinds of relationships have been explored. Some examples are data affinity (JIANG et al., 2017) – which tries to deploy the application close to the data being used – and feature affinity (RAMAKRISHNAN et al., 2013), in which the application is disposed of according to available host features.

6.5.2 Placement of Containers on Clouds

Existing proposals related to the placement of microservices are not concerned with the runtime behaviour (workflow) of the μApp. None of them try to automatically detect the behaviour of the μApps (workflow) in order to reconfigure the placement of their microservices at runtime.

Reducing monetary cost. JRO (GABBRIELLI et al., 2016) uses static resource values, set by users, to deploy microservices on the cluster, similar to commercial approaches such as Kubernetes and Docker Swarm.

The user sets the requirements of CPU and memory usage, the number of replicas and the dependencies between the microservices. JRO uses this information, with data of VMs in the cluster – monetary cost, memory and CPU – to calculate the deployment of μApps minimizing the cost and satisfying the μApps needs.

Both JRO and REMaP use a multi-objective optimization strategy to place microservices in a cluster. However, JRO is not concerned with changing the μApp at runtime based on its behaviour. REMaP on the other hand continuously monitors the μApp and reconfigures its placement whenever affinities between microservices change.

Improvement in resource usage. Medel et al. (2017) propose a new Kubernetes scheduler in order to place containers on the cluster based on containers characterization – e.g., the application into the container is CPU or I/O bounded, defined by the developers. The characterization allows the scheduler to evaluate what is the best configuration to deal with the workload at a given moment.

The authors use the characterization of the applications according to to the resources they use more intensively – CPU, I/O disk, network bandwidth or memory. The characterization is made manually by the application owner, and the scheduler uses this characterization to place containers on the cluster. Medel et al. developed a schedule to balance the number of applications in each node; and/or to minimize the degradation in a machine caused by resource competition.

Medel et al. also considers that the mechanism provided by management tools, like Kubernetes and Docker Swarm, are insufficient to deal with the placement of containers (microservices). The authors use an approach similar to ours and aim to detect resource contention of microservices and decide where to place them. However, REMaP is a fully automatized tool and can change the configuration at runtime according to the μApp's behaviour changes. The scheduler proposed by Medel et al. only acts on the deployment phase of the application or when the microservice scales out. Moreover, they are not interested in using the interaction between microservices as the input of their scheduler.

Awada and Barker (2017) analyzes the container resource usage in order to group them and avoid resource contention – improving the overall performance of the applications.

The authors aim to optimize cluster resource utilization and minimize the overall execution time of tasks on clouds in different geographical regions. The approach considers requirements of containerized applications in order to achieve high throughput, high resource utilization and faster execution time. Awada and Barker's approach captures high-level resource requirements to get an updated state of multi-region cloud cluster for resource availability. Then, they merge the container representation to form new multi-container units and deploy these units on container-instances.

The approach presented by Awada and Barker has some similarities to REMaP. In both cases, the solution aims to improve the performance of μApps, decrease resource usage and avoid resource contention. However, Awada and Barker are not interested in using the runtime behaviour of μApps to update the placement of their microservices on the fly.

Kaewkasi and Chuenmuneewong (2017) uses a meta-heuristic algorithm, named Ant Colony Optimization (ACO), to improve the scheduler's optimality. The proposed algorithm spreads the application throughout containers in order to balance the overall resource usage better and lead applications to better performance in comparison to the default management tool algorithms.

The proposed algorithm helps to maximize the application performance rather than containers being fully packed into a single node. For each iteration, the algorithm tries to put tasks onto available resources in such way that it reduces the total amount of resources being used.

Filip et al. (2018) propose a scheduling algorithm for allocating microservices in a Nano Data Center (VALANCIUS et al., 2009). This approach uses primitive microservices

to compose tasks for processing a given data.

The scheduling algorithm considers a previous analysis of resource consumption of each task and analyzes the available resources currently in the cluster, aiming a balanced distribution of the tasks in the cluster. It also considers the monetary cost of the execution. First, the mechanism knows beforehand how many steps are necessary for the microservice to be executed and calculates the estimated time to perform a task. Based on the resources available in the Nano Data Centers (NaDa), the microservices are allocated to the "servers" on the edge of the network (e.g., gateways and cable modems) based on their resources availability.

The idea proposed by Filip et al. aims to schedule IoT microservices on NaDa and not consider the affinities between microservices to co-locate them in a NaDa. Despite the similarities – e.g., considering previous analysis during scheduling – the REMaP proposal is for common μApps.

(ADAM; LEE; ZOMAYA, 2017) uses a stochastic approach in order to avoid over-provisioning the resources on a cluster. Their mechanism promotes auto-scaling of microservices by applying their algorithm, named Two-stage Stochastic Programming Resource Allocator (2SPRA), on metrics such as network latency and CPU utilization.

Reduce energy consumption.GenPack (HAVET et al., 2017) uses an approach similar to garbage collectors to partition the cluster into groups of machines for different containers lifetime. According to resource usage and lifetime, containers are smartly scheduled to specific groups – co-locating containers with complementary resource requirements. The objective of GenPack is to decrease the energy consumption by the cluster.

GenPack does not know the properties of containers and workloads in advance – it relies on runtime monitoring to observe the resource usage of containers while in a "nursery". Containers run in a young generation of servers which hold short-running jobs and experience relatively high turnaround. This region is like the heap memory space in a garbage collector. Long-running jobs are migrated to the old generation, composed of more stable and energy efficient servers. GenPack runs on top of Docker Swarm.

GenPack uses meta-heuristic is based on genetic algorithms to schedule containers in the cluster. This approach does not use the communication between microservices to decide about their allocation. However, similarly to REMaP, GenPack moves microservices at runtime across different regions of the cluster. These movements are performed according to the generation of the microservices.

Reduce network impact. Guerrero, Lera and Juiz (2018) propose the use of a non-dominated sorting genetic algorithm-II (NSGA-II) to schedule microservices (containers) on a cluster.

The authors aim to achieve equal distribution of the workload along container replicas and physical machines; reduction of network overheads and better reliability. This

approach has a superior performance than Kubernetes plain-scheduler. Furthermore, this approach uses a smaller number of physical machines than Kubernetes, with improvement ratios of up to 4.8 times faster than Kubernetes. However, they are not interested in improving the performance of μApps at all.

Unlike REMaP, Guerrero, Lera and Juiz are neither interested in handling resource contention problems nor improving μApp performance by moving the microservices at runtime.

(BHAMARE et al., 2017) uses an affinity-based approach to schedule microservices in a multi-cloud scenario to decrease the traffic, generated by microservices, and the turnaround to time in order to deliver service to costumers. Unlike REMaP, this approach is neither concerned about resource contention at runtime nor in adapting the placement of microservices at runtime – due to different workflows that may come up during the μApp execution.

In our approach, we are not concerned in an optimized auto-scaling – we aim to optimize the placement of μApps after the calculation of an auto-scaling.

Like REMaP, all those approaches agree about the need to monitor and analyze runtime aspects of the microservices and containers in order to better allocate them in the cluster. However, none of them try to optimize the placement of microservices based on changes in μApp behaviour. They also do not consider the relationship of microservices (containers) and the μApp workflow to co-locate them in order to improve the overall performance of the μApp.

Table 10 summarize the strategies for smart placement of microservices (containers) in the cloud.

6.5.3 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, Lee and Katz (2011) calculate 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. (2017) calculating 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 (LI et al., 2010). 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 calculates based on job communication, like REMaP. *AAGA* (CHEN et al., 2013), *CIVSched* (GUAN et al., 2014), and *Starling* (SONNEK et al., 2010) are some examples of this approach. In general, these works compute the

Table 10 – Strategies for microservices placement.

Work	**Objective**
(GABBRIELLI et al., 2016)	Reduce monetary cost
(MEDEL et al., 2017)	Avoid resource contention
(AWADA; BARKER, 2017)	Avoid resource contention
(KAEWKASI; CHUENMUNEEWONG, 2017)	Balancing resources usage in cluster
(FILIP et al., 2018)	Balancing microservices onto cluster
(ADAM; LEE; ZOMAYA, 2017)	Reduce resource usage in the cluster
(HAVET et al., 2017)	Reduce energy consumption
(GUERRERO; LERA; JUIZ, 2018)	Reduce network overhead
(BHAMARE et al., 2017)	Reduce traffic over network
REMaP	Avoid resource contention Reduce network overhead

Source – Made by the author

jobs' affinity based on the bandwidth between them. The algorithms used are in some sense a variation of First-Fit algorithm (DOSA, 2008), which tries to fit related jobs into a node with the available resources. The REMaP's planner HBA is a simplification of the First-Fit algorithm. Finally, like REMaP, some of these algorithms (SONNEK et al., 2010) 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 in order to compute affinities. Broquedis et al. (2010) 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 like REMaP. However, *hwloc* was designed only to provide an interface to be used by another scheduler whereas REMaP is a complete solution in updating element placement in the μApp. It does so in 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.

6.6 CONCLUDING REMARKS

In this chapter we positioned our work on state-of-the-art runtime adaptation of microservices. The runtime adaptation of microservices is related to the concepts of microservice evolution and placement of microservices on cloud. We position our work in relation to these subjects. Moreover, REMaP uses Models@run.time to support the adaptation of μApps. Hence, we position our work in relation to the use of models on microservices and services adaptation.

7 CONCLUSION AND FUTURE WORK

This chapter presents the conclusion and next steps of this thesis. We show the potential of REMaP for: improving the performance, reducing resource waste and reducing the resource contention of μApps in a cluster. We highlight our contributions on using Models@run.time for μApps and runtime placement of μApps as well as discuss the limitations of our approach. Finally, we present future directions of this research.

7.1 CONCLUSION

Despite the flexibility provided by microservices, existing management tools cannot apply rich adaptations to μApps at runtime. To enrich how the adaptation is carried out on a μApp, we propose a platform-independent runtime adaptation mechanism to reconfigure the placement of microservices based on their communication affinities and runtime resources usage. To achieve it, we developed REMaP (**R**untim**E** **M**icroservices **P**lacement), an adaptation manager that monitors the μApp and uses collected data to reorganize the placement of the microservices.

At the core of our approach is a innovative optimization that changes the arrangement of microservices at runtime. REMaP is our view on how Models@run.time drives the adaptation of a μApp. Hence, in this work, we use a model to provide a unified view of the μApp and to ensure safe changes to the μApp.

The model abstracts several technologies involved in the cluster and μApp monitoring, organizing all data collected into a unified structure – providing a single view of the μApp. The model also highlights different aspects of the μApps, such as μApp workflow and topology, and allow the inference of new information about the μApp. Moreover, the adaptation manager uses the model during μApp adaptation to ensure that the adaptation will not lead the μApp to an inconsistent configuration. All adaptations are first checked on model for this, if they are safe, they are applied to the application.

REMaP is based on the MAPE-K architecture, which defines four phases to manage systems autonomously. These phases are **m**onitoring, **a**nalysis, **p**lanning and **e**xecution. There is a **k**nowledge-base, used to maintain useful information for the adaptation process, weaved to each phase.

In the *monitoring* phase, REMaP collects heterogeneous data, such as metrics and communication messages, from several sources and unifies them into a model. REMaP uses this model as a *knowledge-base* throughout the other phases of MAPE-K.

During the *analysis* phase, REMaP inspects the model looking for *affinities* between the microservices. An affinity defines a degree of relationship between two microservices and is calculated based on the number as well as size of messages exchanged. REMaP uses

the affinities in the *planning* phase. The adaptation is calculated based on: the affinities, records of resource usage and resources available in the cluster. The objective of the adaptation is to co-locate microservices with high affinity using the resource usage and resources available as constraints.

Once the adaptation is calculated, REMaP *executes* it by collocating high related microservices. The adaptation process is guided by the model, that maintains a causal connection with the μApp and avoids inconsistencies between the adaptation to be applied and the current state of the application.

In the end, the adaptation process has potential to:

1. Improve the μApp performance: Co-locating high related microservices decreases the network impact on the microservice's communication, improving the overall performance of the μApp;

2. Reduce resource wasting: Co-locating lightweight microservices avoids their execution in individual hosts; and

3. Reducing resource contention: Co-locating microservices based on their previous resource consumption avoids them being put together with heavyweight microservices. While peaks are difficult to be identified by gathering the current resource usage, information about historical resource consumption allows REMaP a better view of the behaviour of the μApp.

Finally, μApps are dynamic applications and continually change their behaviour (workflows). The dynamic performance makes the affinities vary along the μApp execution. Unlike existing approaches, whose scheduling is performed only when the application starts to execute, REMaP continuously observes the μApp's behaviour – adapting its placement whenever necessary, according to the new affinities that arise at runtime and changes in resource usage history.

7.2 SUMMARY OF CONTRIBUTIONS

The main contribution of this thesis *the management of μApps by promoting adaptation driven by Models@run.time which unify different behavioral aspects of the application.* We achieved this contribution through:

1. The definition of a service evolution model and its use at runtime to support the management of μApps by abstracting the heterogeneity of μApp environments and organizing concepts related to microservice domain; and,

2. The characterization and use of the affinity concept between microservices in order to dynamically change their placement.

Next, we discuss the contribution highlighting these two aspects studied as well as the secondary contributions that come up along the development of this thesis.

7.2.1 Service evolution model for μApps

REMaP makes intensive use of Models@run.time. We contributed by proposing a services evolution model – flexible to several situations in the management of μApps– to describe μApps. In this thesis, we presented several circumstances where the model can be used and how we vary the initial version of the services evolution model to be used in our runtime adaptation manager.

μApp management is a cumbersome task performed by μApp engineers. Different tools used in the microservice domain makes difficult a complete view of the running μApp. To achieve it, engineers must observe various aspects of the application individually, such as resource usage, communication and logs. Furthermore, a different set of non-integrated technologies are also used for each element.

Our contribution to this problem is to unify all the information in a Models@run.-time used to provide a complete view of the μApp without taking into consideration its monitoring technology. Moreover, the use of Models@run.time makes the model in conformance with the μApp throughout its execution. The causal connection between the model and the μApp in execution is essential since μApps change several times during their existence. By using a live model (Models@run.time), we can track all changes carried out on the μApp, maintaining a reliable view of its structure and behaviour.

Our solution also shows the importance and viability of using Models@run.time in the microservice domain. In particular, the proposed model includes elements that serve to characterize the microservice domain and μApps.

7.2.2 Runtime Placement of μApps

The runtime adaptation of μApps relies mostly on the scale in and out activities based on resource usage – with few works handling other types of runtime adaptation. Moreover, the adaptation on μApps uses real-time metrics or static information – set by engineers to guide the adaptation. They do not look at the execution history of the μApp neither its behaviour to take any action.

Our contribution related to this subject, is to perform runtime adaptation of microservice placement based on past data and behavioural information. Our proposal for analyzing microservice data records allows adaptation managers to make reliable decisions instead of unsafe choices based on ephemeral data (e.g. snapshot real-time metric usage) that does not represent the whole behaviour of the system.

Moreover, our approach uses the concept of *microservice affinities* to reorganize the placement of microservices. In the exploratory phase of this thesis, we observed the impact

of the placement of the μApp performance. Based on this observation, we understood how we could analyze whether microservices should be co-located or not – hence, we defined the concept of *affinity*. With this concept, we can understand the behaviour of μApps better and take more robust decisions by not considering only application metrics.

Our adaptation approach considers the behaviour of the μApp and not only metrics, leading the decision making to be more robust and dynamic. μApps are dynamic applications and their behaviour changes many times along its execution. Performing analysis on these applications by checking only "static data" such as metric snapshots provides poor feedback to the management mechanism.

Finally, the adaptation of μApps by reorganizing the placement of microservices is a novelty in this domain. Works related to this topic statically handle the arrangement. Based on a set of information, the scheduler of management tools decides to optimize the first deployment of the μApp and, in some cases, the installation of microservice replicas. The management tools are not aware of how runtime changes affect the μApp behaviour and the initial placement strategy might not be useful anymore. Our approach continuously checks the new behaviour that comes up from a μApp due to the calculation of affinities. Therefore, we can reorganize the placement of μApps as well as the μApp changes its behaviour.

7.2.3 Secondary Contributions

The main contributions of this thesis are substantiated in several secondary contributions as described in the following:

Characterization of adaptation on μApps. In the early stage of this work we investigated the meaning of *adaptation* for μApps. During research, we elaborated and crystallized this concept for the microservice domain – we systematized that adaptation for μApps stands for updates or upgrades. An update is an adaptation that changes the arrangement of the μApp in a cluster – e.g., scaling in and out or reorganizing the placement of μApps. Upgrade on the other hand, changes the implementation and/or the workflow of the μApp– e.g., rolling out and back microservice versions. Moreover, any adaptation of the μApp relies on instantiating microservices somewhere. Therefore, we contribute by specializing the general concept of adaptation for the bounded domain of microservices.

Heuristic to reorganize the μApp placement. The problem of the optimal placement of μApps is hard to solve due to its theoretical characteristics. We propose a heuristic to decide where to place high related microservices. Our algorithm is good enough for the most common scenarios evaluated in this thesis and can be reasonably used as the starting point for the development of more sophisticated algorithms.

Optimization model to reorganize μApp placement. As an attempt to place a μApp in a cluster optimally, we used the Z3 SMT solver – considering our objectives

and constraints. To our knowledge, SMT solvers are not traditionally used to solve multi-objective bin-packing problems where affinities are part of the constraints/objectives to be solved/reached. Nowadays, existing solutions only consider resource usage and available resources as input. Therefore, we contribute by creating an optimization model for Z3 that uses this new approach (metrics and affinities) to calculate an optimal placement of microservices.

Framework for creating mock μApps. The use of real μApps to validate works is a challenge. There are few μApps and open source available which are similar in size and complexity to μApps running real companies. Due to this limitation, we created a framework to emulate real μApps running on a cluster managed by a Kubernetes tool by creating artificial microservices and host configurations – attaching microservices to hosts and linking the microservices make up a mock μApp. We used this framework to generate the mock instances used in our evaluation. This minor contribution is useful for other people who aim to validate their proposals in large scale if they do not have a real μApp to validate it.

MAPE-K based adaptation manager for μApps. MAPE-K is the reference architecture on developing tools for autonomously managed systems. In this context REMaP is a innovation as it is the first adaptation tool for μApps that uses Models@run.time to guide the adaptation process. Existing tools only consider real-time information to reason and act on a μApp. REMaP on the other hand, can reason about past data and infer new information on the μApps behaviour by using Models@run.time– making it a robust tool to manage μApps autonomously.

7.3 THREATS OF VALIDITY

Next, we list the threats of validity related to REMaP.

Evaluation on a single real-world application. REMaP was evaluated by using a mock and an empirical approach. However, the empirical one was carried out on a single real μApp named Sock-shop. Even though Sock-shop is widely used as a reference μApp, it cannot pop up a wide range of situations to exercise all features of REMaP at once, thus limiting our evaluation. Moreover, the other open-sourced μApps available are in most cases incomplete, unstable, cumbersome or infrastructure dependent (different cloud providers such as Amazon AWS, Google Cloud Platform, and IBM Bluemix). These characteristics make it difficult to set up an evaluation testbed – mainly because the infrastructure dependency demands a high monetary cost in order to solve them.

Not handling replicas during the adaptation. REMaP does not cover an important aspect during the movement of microservices – handling its replicas. The migration of a container (microservice) is not like the migration of a virtual machine, which can freeze its state and execution while it is being moved and then be resumed. Containers are

processes that exist so that there is a technological limitation to carry out the migration of containers across different machines, like virtual machines. Due to this situation, the movement of a microservice relies on deleting its instance in the source and instantiating a new one in the destination. This task is under the responsibility of management tool's scheduler. Therefore, we decided not to consider replicas since it would be necessary to develop a whole new scheduler for a management tool (e.g., Kubernetes) that was out of the scope of this thesis.

Adaptation based on few quality attributes. In the current implementation, REMaP computes an adaptation guided by only two quality attributes, performance and resource consumption, leaving out other important cases, such as resiliency and security. This current strategy makes REMaP unaware of important aspects of a distributed systems, which avoid that more robust adaptations be applied on μApps. For instance, REMaP is unaware of failures frequency of the microservices. Hence, REMaP cannot decide to maintain more replicas of a given microservices of avoid long downtimes in addition to place these replicas in a optimal location.

Dumb[1] **Monitoring.** REMaP monitoring gathers data from a different data store and uses this information to populate the model. Although μApps are dynamic, much information should be inferred based on the context of the environment (e.g., time) and historical data. As we are aiming to build an end-to-end solution, we do not focus on the optimization of the process; hence we do not Implement techniques to make the monitoring smarter – using machine learning to foresee metric usage, statistical analysis to gather fewer amounts of data an so on. The use of these techniques could partially improve the overall performance of REMaP by decreasing the amount of data being collected.

REMaP is not microservice-based. Microservice style promotes easy maintenance by the developer's teams. It is not right for a single person to maintain a complex application like REMaP. Debugging an application of an ongoing project is a hard task and is even worse if it is distributed as the microservice style suggests. Hence, despite REMaP using most of the architectural patterns adopted by μApps– such as separation of concerns, well defined bounded components, asynchronous communication – its implementation does not follow the technological standards suggested for μApp development.

Adaptation trigger is semi-automatized. Our current implementation uses a parameterized value to trigger the adaptations – the adaptation mechanism checks the μApp at pre-determined time intervals set by a μApp engineer. This approach leads REMaP to become a semi-automated solution for adapting μApps. However, to fully automatize when the adaptation starts, an entire work is necessary in order to systematize how much data is needed to sample and analyze a given aspect of the μApp and then safely trigger the adaptation. To fully automatize adaptation triggering, further study on the smart

[1] Not smart. For example, smartphone (e.g., iPhone) and dumbphone (e.g., old fashion cell phone - the era before smartphones)

sampling of μApp data, including the use of statistical and artificial intelligence methods, is necessary in order to reach a reliable number; however, these approaches are out of the scope of this thesis.

Using EMF to keep the Models@run.time. EMF is a widely adopted Java framework used in Model Driven Development (MDE) that provides much ease for modelling and handling Models@run.time. The EMF was not developed initially to be thread safe, which makes its usage in a multi-threaded application unsafe. However, due to the ease provided by the framework, we choose it for a fast development over runtime performance. As a consequence, REMaP has a poor performance in handling the model – since its implementation is mostly single-threaded and, when multi-threaded, it suffers from an overhead, imposed by the use of synchronization primitives. This strategy means that REMaP does not use all the power of its host to populate the model and calculate the affinities. As a consequence, the model handling is a bottleneck in the adaptation process.

Non-optimized implementation of planning algorithms. Similar to the issue of processing the model in a non-optimized model, we choose to implement the planning algorithms with minimum optimization to facilitate its development. *planner OA* that uses Z3, an industrial SMT solver, to calculate the adaptation. The Z3 programming model has a steep learning curve, which makes it difficult to master its programming model and optimize an ongoing code simultaneously. Despite these problems, minor optimizations were applied, such as using data types of small sizes to reduce the search space during planning calculation. However, it is clear that there is a possibility of employing more optimizations in the Z3 model, consequently improving the results obtained in this thesis.

7.4 FUTURE WORKS

Based on the threats of the validity of this thesis, we present some future works that should be implemented to make REMaP a robust platform of runtime adaptation in μApps.

Supporting microservice evolution. It would be interesting to identify a set of desired architectural and deployment patterns and monitor their preservation as the application evolves. This monitoring can be achieved by analyzing the collected information in services, operations, exchanged messages, networking and CPU metrics, and so on. Whenever the application integrity or service quality is under risk, our solution might recommend appropriate improvements – such as replacing a microservice, splitting a microservice, merging a set of microservices, or moving microservices to different hosts.

Evaluation of new real-world applications. As mentioned before, REMaP was evaluated using a single real-world μApp. We intend to use REMaP with other μApps to check its impact on the performance of other applications. However, performing this kind of evaluation is prohibited without any sponsorship to provide a real application – used by the industry – to the faculty. Hence, we are aiming to develop a test bed

to create μApps on demand – following an architecture set by the user. The general idea is to expand the framework used to create the mock experiments for this work. The framework should automatically instantiate the model (graph) generated by using customized microservices – dummy microservices that receive fake messages (with sizes and request rate parameterized) and return a fake answer (also parameterized). Hence, we will be able to run experiments on a high scale and in a fully controlled environment.

Handling microservices replicas during adaptation. REMaP is not able to handle microservice replicas properly during the adaptation due to technological limitations. To overcome this limitation, we aim to implement a new scheduler to Kubernetes, which will work with REMaP so that REMaP to provide feedback and vice-versa. This approach will allow microservice replicas to be naturally handled by the adaptation platform (REMaP and scheduler) during an adaptation.

Smart Monitoring. We aim to trigger the adaptation without any human intervention, making it necessary to define how much data is required to make a safe analysis and plan an adaptation. Accordingly, the analysis phase should be expanded to also make a statistical analysis of the amount of data collected in order to decide the best time to trigger an adaptation. Furthermore, information can be foreseen based on historical data – we can use the machine's learning to predict future resource usage based on the history of the μApp and anticipate some changes to the μApp, such as auto-scaling or placement reconfiguration. Therefore, we are also planning to apply statistical and artificial intelligence techniques to improve the monitoring and analysis activities.

Microservice-based REMaP implementation. We intend to split REMaP into several microservices to facilitate its extension, e.g., to add new *analyzers* and *planners*. This new architecture will make REMaP more powerful so that several aspects of a μApp might be evaluated and adapted simultaneously and at runtime.

Modern tools for handling Models@run.time. Aiming to improve the performance of REMaP's workflow to calculate an adaptation, we intend to replace the framework for handling Models@run.time with more recent tools used in high-performance data analysis.

At first glance we aim to use non-structured data stores – graph- and document-oriented databases – due to the hierarchical and graph-based features of data collected. Even related databases are candidates. The use of related data stores is interesting due to the number of data relationships necessary to infer information based on the data collected from μApps.

Another alternative is to use modern and thread-safe Models@run.time frameworks, allowing multi-thread algorithms in the analysis of the model, thus improving the overall performance in REMaP's adaptation.

New planning strategies. The results obtained in this work show us that although we have achieved the objective of bringing autonomy to μApp management, some steps

must still be improved. First of all, the adaptation of μApps by updating the microservice placement is an essential issue for cluster providers. However, due to the complexity of the problem, new strategies must be applied instead of the brute force used (SMT Solver). Therefore, we should try different techniques, such as evolutionary and swarm algorithms metaheuristics in order to compute an optimal placement of microservices. Moreover, we aim to improve our heuristic in order to calculate a smart placement based on more attributes than only microservice affinities and resources usage.

Handle strongly connected sub graphs over affinities pairs. We aim to extend the adaptation strategy by improving the analysis and planner phases to find affinity hubs in μApps (strongly connected sub graphs) than simple microservices pairs. Our experiments highlight that the use of hubs by approximations algorithms may produces better results than co-location of simple pairs.

BIBLIOGRAPHY

ACHARYA, S.; MELLO, D. A. D. A taxonomy of Live Virtual Machine (Vm) Migration mechanisms in cloud computing environment. In: IEEE. *ICGCE*. [S.l.], 2013. p. 809–815.

ADAM, O.; LEE, Y. C.; ZOMAYA, A. Y. Stochastic resource provisioning for containerized multi-tier web services in clouds. *TPDS*, IEEE, v. 28, n. 7, p. 2060–2073, 2017.

ADERALDO, C. M.; MENDONÇA, N. C.; PAHL, C.; JAMSHIDI, P. Benchmark Requirements for Microservices Architecture Research. In: . [S.l.]: IEEE, 2017.

AWADA, U.; BARKER, A. Improving Resource Efficiency of Container-Instance Clusters on Clouds. In: . [S.l.: s.n.], 2017. p. 929–934.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, IEEE, v. 33, n. 3, p. 42–52, 2016.

BALALAIE, A.; HEYDARNOORI, A.; JAMSHIDI, P.; TAMBURRI, D. A.; LYNN, T. Microservices migration patterns. *Software: Practice and Experience*, Wiley Online Library, v. 48, n. 11, p. 2019–2042, 2018.

BARABÁSI, A.-L.; ALBERT, R. Emergence of scaling in random networks. *Science*, American Association for the Advancement of Science, v. 286, n. 5439, p. 509–512, 1999.

BAYLESS, S.; KODIROV, N.; BESCHASTNIKH, I.; HOOS, H. H.; HU, A. J. Scalable constraint-based virtual data center allocation. In: AAAI PRESS. *IJCAI*. [S.l.], 2017. p. 546–554.

BERNSTEIN, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, IEEE, n. 3, p. 81–84, 2014.

BHAMARE, D.; SAMAKA, M.; ERBAD, A.; JAIN, R.; GUPTA, L.; CHAN, H. A. Multi-Objective Scheduling of Micro-Services for Optimal Service Function Chains. In: IEEE. *ICC*. [S.l.], 2017. p. 1–6.

BICHLER, M.; LIN, K.-J. Service-oriented computing. *Computer*, v. 39, n. 3, p. 99–101, 2006.

BIERE, A.; BIERE, A.; HEULE, M.; MAAREN, H. van; WALSH, T. *Handbook of Satisfiability: Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009.

BLAIR, G.; BENCOMO, N.; FRANCE, R. B. Models@run.time. *Computer*, IEEE, v. 42, n. 10, 2009.

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: IEEE. *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. [S.l.], 2010. p. 180–186.

BROWN, A.; KAR, G.; KELLER, A. An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. In: *IM*. [S.l.: s.n.], 2001.

CARDELLINI, V.; CASALICCHIO, E.; GRASSI, V.; PRESTI, F. L.; MIRANDOLA, R. Qos-driven runtime adaptation of service oriented architectures. In: ACM. *SFSE*. [S.l.], 2009. p. 131–140.

CARNEIRO, C.; SCHMELMER, T. Deploying and running microservices. In: *Microservices From Day One*. [S.l.]: Springer, 2016. p. 151–174.

CASATI, F.; ILNICKI, S.; JIN, L.; KRISHNAMOORTHY, V.; SHAN, M.-C. Adaptive and dynamic service composition in eFlow. In: SPRINGER. *ICAISE*. [S.l.], 2000. p. 13–31.

CHEKURI, C.; KHANNA, S. On multidimensional packing problems. *SIAM*, SIAM, v. 33, n. 4, p. 837–851, 2004.

CHEN, J.; CHIEW, K.; YE, D.; ZHU, L.; CHEN, W. AAga: Affinity-aware grouping for allocation of virtual machines. In: IEEE. *AINA*. [S.l.], 2013. p. 235–242.

CHOTHIA, Z.; LIAGOURIS, J.; DIMITROVA, D.; ROSCOE, T. Online Reconstruction of Structural Information from Datacenter Logs. In: *Eurosys*. [S.l.: s.n.], 2017.

CHRISTENSEN, H. I.; KHAN, A.; POKUTTA, S.; TETALI, P. *Multidimensional bin packing and other related problems: A survey*. 2016.

DERAKHSHANMANESH, M.; GRIEGER, M. Model-Integrating Microservices: A vision Paper. In: *Software Engineering (Workshops)*. [S.l.: s.n.], 2016. p. 142–147.

DOSA, G. t. n. First Fit Algorithm for Bin Packing. In: _____. *Encyclopedia of Algorithms*. Boston, MA: Springer US, 2008. p. 1–5. ISBN 978-3-642-27848-8.

DÜLLMANN, T. F.; HOORN, A. van. Model-driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches. In: *ICPE*. [S.l.: s.n.], 2017.

ENSEL, C. Automated Generation of Dependency Models for Service Management. In: *OVUA*. [S.l.: s.n.], 1999.

FELTER, W.; FERREIRA, A.; RAJAMONY, R.; RUBIO, J. An updated performance comparison of virtual machines and linux containers. In: IEEE. *ISPASS*. [S.l.], 2015. p. 171–172.

FILIP, I. D.; POP, F.; SERBANESCU, C.; CHOI, C. Microservices Scheduling Model over Heterogeneous Cloud-Edge Environments as Support for Iot Applications. *IEEE Internet of Things Journal*, p. 1–1, 2018.

FLORIO, L.; NITTO, E. D. Gru: An Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures. In: IEEE. *ICAC*. [S.l.], 2016. p. 357–362.

GABBRIELLI, M.; GIALLORENZO, S.; GUIDI, C.; MAURO, J.; MONTESI, F. Self-reconfiguring microservices. In: *Theory and Practice of Formal Methods*. [S.l.]: Springer, 2016. p. 194–210.

GARLAN, D.; CHENG, S. W.; HUANG, A. C.; SCHMERL, B.; STEENKISTE, P. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, v. 37, p. 46–54, 2004.

GRANCHELLI, G.; CARDARELLI, M.; FRANCESCO, P. D.; MALAVOLTA, I.; IOVINO, L.; SALLE, A. D. Towards Recovering the Software Architecture of Microservice-Based Systems. In: IEEE. *ICSAW*. [S.l.], 2017. p. 46–53.

GUAN, B.; WU, J.; WANG, Y.; KHAN, S. CIvsched: a communication-aware inter-Vm scheduling technique for decreased network latency between co-located Vms. *IEEE transactions on cloud computing*, IEEE, n. 1, p. 1–1, 2014.

GUERRAOUI, R.; PAVLOVIC, M.; SEREDINSCHI, D.-A. Trade-offs in Replicated Systems. *Data Engineering Bulletin*, IEEE, v. 39, n. 1, p. 14–26, 2016.

GUERRERO, C.; LERA, I.; JUIZ, C. Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. *Journal of Grid Computing*, Springer, v. 16, n. 1, p. 113–135, 2018.

HANNAFORD, J. *What happens when ... Kubernetes edition!* 2017. Https://github.com/jamiehannaford/what-happens-when-k8s. Online; accessed 15 June 2018.

HASSAN, S.; ALI, N.; BAHSOON, R. Microservice Ambients: An Architectural Meta-modelling Approach for Microservice Granularity. In: IEEE. *ICSA*. [S.l.], 2017. p. 1–10.

HASSAN, S.; BAHSOON, R. Microservices and their design trade-offs: A self-adaptive roadmap. In: IEEE. *SCC*. [S.l.], 2016. p. 813–818.

HAVET, A.; SCHIAVONI, V.; FELBER, P.; COLMANT, M.; ROUVOY, R.; FETZER, C. GEnpAck: A generational scheduler for cloud data centers. *IC2E*, p. 95–104, 2017.

HEORHIADI, V.; RAJAGOPALAN, S.; JAMJOOM, H.; REITER, M. K.; SEKAR, V. Gremlin: Systematic Resilience Testing of Microservices. In: *ICDCS*. [S.l.: s.n.], 2016. p. 57–66.

HOFF, T. *Microservices - Not a free lunch.* 2014. Http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html. Online; accessed 21 December 2017.

HOSEK, P.; CADAR, C. Safe Software Updates via Multi-version Execution. In: *ICSE*. [S.l.: s.n.], 2013.

HUEBSCHER, M. C.; MCCANN, J. A. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys*, v. 40, n. 3, p. 1–28, aug 2008.

IBM. *An architectural blueprint for autonomic computing.* [S.l.], 2005. 34 p.

JAMSHIDI, P.; PAHL, C.; MENDONÇA, N. C.; LEWIS, J.; TILKOV, S. Microservices: The Journey So Far and Challenges Ahead. *Software*, IEEE, v. 35, n. 3, p. 24–35, 2018.

JI, Z.-l.; LIU, Y. A dynamic deployment method of micro service oriented to Sla. *IJCS*, v. 13, n. 6, p. 8–14, 2016.

JIANG, J.; SUN, S.; SEKAR, V.; ZHANG, H. Pytheas: Enabling Data-Driven Quality of Experience Optimization Using Group-Based Exploration-Exploitation. In: *NSDI*. [S.l.: s.n.], 2017. p. 393–406.

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. Online; accessed March 2018.

KAEWKASI, C.; CHUENMUNEEWONG, K. Improvement of container scheduling for Docker using Ant Colony Optimization. In: *254–259*. [S.l.: s.n.], 2017.

KANG, D.-K.; CHOI, G.-B.; KIM, S.-H.; HWANG, I.-S.; YOUN, C.-H. Workload-aware resource management for energy efficient heterogeneous docker containers. In: IEEE. *TENCO*. [S.l.], 2016. p. 2428–2431.

KARASTOYANOVA, D.; HOUSPANOSSIAN, A.; CILIA, M.; LEYMANN, F.; BUCHMANN, A. Extending BpeL for run time adaptability. In: IEEE. *EDOC*. [S.l.], 2005. p. 15–26.

KAUFMAN, L.; ROUSSEEUW, P. *Clustering by means of medoids*. [S.l.]: North-Holland, 1987.

KELING, D.; DALMAU, M.; ROOSE, P. A survey of Adaptation Systems. *IJIDCS*, v. 1, n. 1, p. 1–18, 2012.

KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. *Computer*, v. 36, n. 1, p. 41–50, Jan 2003.

KICZALES, G.; RIVIERES, J. D.; BOBROW, D. G. *The art of the metaobject protocol*. [S.l.]: MIT press, 1991.

KILLALEA, T. The hidden dividends of microservices. *Communications*, ACM, v. 59, n. 8, p. 42–45, jul 2016.

KORTE, B.; VYGEN, J. Bin-Packing. In: _____. *Combinatorial Optimization: Theory and Algorithms*. [S.l.]: Springer Berlin Heidelberg, 2006. p. 426–441.

KRUPITZER, C.; ROTH, F. M.; VANSYCKEL, S.; SCHIELE, G.; BECKER, C. A survey on engineering approaches for self-adaptive systems. *PMC*, Elsevier B.V., v. 17, p. 184–206, 2014.

LASI, H.; FETTKE, P.; KEMPER, H.-G.; FELD, T.; HOFFMANN, M. Industry 4.0. *BISE*, Springer, v. 6, n. 4, p. 239–242, 2014.

LEE, G.; KATZ, R. H. Heterogeneity-Aware Resource Allocation and Scheduling in the Cloud. In: *HotCloud*. [S.l.: s.n.], 2011.

LEELIPUSHPAM, P. G. J.; SHARMILA, J. Live Vm migration techniques in cloud environment—a survey. In: IEEE. *ICT*. [S.l.], 2013. p. 408–413.

LEITNER, P.; CITO, J.; STÖCKLI, E. Modelling and Managing Deployment Costs of Microservice-based Cloud Applications. In: *UCC*. [S.l.: s.n.], 2016.

LEWIS, J.; FOWLER, M. *Microservices*. 2014. Https://martinfowler.com/articles/microservices.html. Online; accessed 21 December 2017.

LI, Z.; BAI, Y.; ZHANG, H.; MA, Y. Affinity-aware dynamic pinning scheduling for virtual machines. In: IEEE. *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. [S.l.], 2010. p. 242–249.

LU, T.; STUART, M.; TANG, K.; HE, X. Clique migration: Affinity grouping of virtual machines for inter-cloud live migration. In: IEEE. *NAS*. [S.l.], 2014. p. 216–225.

MA, X.; BARESI, L.; GHEZZI, C.; MANNA, V. P. L.; LU, J. Version-consistent Dynamic Reconfiguration of Component-based Distributed Systems. In: *ESEC/FSE*. [S.l.: s.n.], 2011.

MA, Y.; LIU, X.; WU, Y.; GRACE, P. Model-based management of service composition. In: IEEE. *SOSE*. [S.l.], 2013. p. 103–112.

MAES, P. Concepts and experiments in computational reflection. *SIGPLAN*, ACM, v. 22, n. 12, p. 147–155, dec 1987.

MEDEL, V.; TOLÓN, C.; ARRONATEGUI, U.; TOLOSANA-CALASANZ, R.; BAÑARES, J. Á.; RANA, O. F. Client-Side Scheduling Based on Application Characterization on Kubernetes. In: PHAM, C.; ALTMANN, J.; BAÑARES, J. Á. (Ed.). *EGCSS*. Cham: Springer International Publishing, 2017. p. 162–176.

NEWMAN, S. *Building Microservices*. [S.l.]: O'Reilly Media, Inc., 2015. 280 p.

NGUYEN, T.; COLMAN, A. A feature-oriented approach for web service customization. In: IEEE. *Web Services (ICWS), 2010 IEEE International Conference on*. [S.l.], 2010. p. 393–400.

NÚÑEZ-VALDEZ, E. R.; GARCÍA-DÍAZ, V.; LOVELLE, J. M. C.; ACHAERANDIO, Y. S.; GONZÁLEZ-CRESPO, R. A model-driven approach to generate and deploy videogames on multiple platforms. *JAIHC*, Springer, v. 8, n. 3, p. 435–447, 2017.

OASIS. *Reference Model for Service Oriented Architecture 1.0*. 2006. Http://docs.oasis-open.org/soa-rm/v1.0/. Online; accessed May 2018.

OLINER, A.; GANAPATHI, A.; XU, W. Advances and Challenges in Log Analysis. *CACM*, v. 55, n. 2, p. 55–61, feb 2012.

OMG. *Service Oriented Architecture Modeling Language Specification*. 2012. Https://www.omg.org/spec/SoaML/. Online; accessed May 2018.

OREIZY, P.; MEDVIDOVIC, N.; TAYLOR, R. N. Runtime software adaptation: framework, approaches, and styles. In: ACM. *ICSE*. [S.l.], 2008. p. 899–910.

PACHORKAR, N.; INGLE, R. Multi-dimensional affinity aware Vm placement algorithm in cloud computing. *IJACR*, International Journal of Advanced Computer Research, v. 3, n. 4, p. 121, 2013.

PANDA, A.; SAGIV, M.; SHENKER, S. Verification in the Age of Microservices. In: *HotOS*. [S.l.: s.n.], 2017.

PAPAZOGLOU, M. P. Service -oriented computing: Concepts, characteristics and directions. *WISE*, p. 3–12, 2003.

PARVIAINEN, P.; TAKALO, J.; TEPPOLA, S.; TIHINEN, M. Model-driven development processes and practices. *VTT Technical Research Centre of Finland*, 2009.

PEREIRA, R.; COUTO, M.; RIBEIRO, F.; RUA, R.; CUNHA, J.; FERNANDES, J. P.; SARAIVA, J. 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*. New York, NY, USA: ACM, 2017. (SLE 2017), p. 256–267. ISBN 978-1-4503-5525-4.

RADEMACHER, F.; SACHWEH, S.; ZÜNDORF, A. Differences between Model-Driven Development of Service-Oriented and Microservice Architecture. In: *ICSAW*. [S.l.: s.n.], 2017. p. 38–45.

RADEMACHER, F.; SACHWEH, S.; ZÜNDORF, A. Analysis of Service-oriented Modeling Approaches for Viewpoint-specific Model-driven Development of Microservice Architecture. *arXiv preprint arXiv:1804.09946*, 2018.

RADEMACHER, F.; SORGALLA, J.; SACHWEH, S. Challenges of domain-driven microservice design: A model-driven perspective. *Software*, IEEE, v. 35, n. 3, p. 36–43, 2018.

RAJAGOPALAN, S.; JAMJOOM, H. App-Bisect: autonomous healing for microservice-based apps. In: USENIX ASSOCIATION. *HotCLoud*. [S.l.], 2015.

RAMAKRISHNAN, A.; NAQVI, S. N. Z.; BHATTI, Z. W.; PREUVENEERS, D.; BERBERS, Y. Learning deployment trade-offs for self-optimization of Internet of Things applications. In: ACM. *ICAC*. [S.l.], 2013. p. 213–224.

SALEHIE, M.; TAHVILDARI, L. Self-adaptive software: Landscape and research challenges. *TAAS*, v. 4, n. 2, p. 1–42, 2009.

SAMPAIO JR., A.; KADIYALA, H.; HU, B.; STEINBACHER, J.; ERWIN, T.; ROSA, N.; BESCHASTNIKH, I.; RUBIN, J. Supporting evolving microservices. In: IEEE. *ICSME*. [S.l.], 2017.

SAMPAIO JR., A. R.; COSTA, F. M.; CLARKE, P. A model-driven Approach to Develop and Manage Cyber-Physical Systems. In: *Models@Run.time*. [S.l.: s.n.], 2013. p. 64–75.

SANGAL, N.; JORDAN, E.; SINHA, V.; JACKSON, D. Using Dependency Models to Manage Complex Software Architecture. *SIGPLAN*, v. 40, n. 10, p. 167–176, 2005.

SCHMIDT, D. C. Model-driven engineering. *COMPUTER*, Citeseer, v. 39, n. 2, p. 25, 2006.

SEO, K.-T.; HWANG, H.-S.; MOON, I.-Y.; KWON, O.-Y.; KIM, B.-J. Performance comparison analysis of linux container and virtual machine for building cloud. *ASTL*, v. 66, n. 105-111, p. 2, 2014.

SINGH, A. N.; PRAKASH, S. Challenges and opportunities of resource allocation in cloud computing: A survey. In: IEEE. *INDIACom*. [S.l.], 2015. p. 2047–2051.

SINGLETON, A. The Economics of Microservices. *Cloud Computing*, IEEE, v. 3, n. 5, p. 16–20, 2016.

SONNEK, J.; GREENSKY, J.; REUTIMAN, R.; CHANDRA, A. Starling: Minimizing communication overhead in virtualized computing platforms using decentralized affinity-aware migration. In: IEEE. *ICPP*. [S.l.], 2010. p. 228–237.

SORGALLA, J. Ajil: A graphical modeling language for the development of microservice architectures. In: *EAM*. [S.l.: s.n.], 2017.

SORGALLA, J.; RADEMACHER, F.; SACHWEH, S.; ZÜNDORF, A. On Collaborative Model-driven Development of Microservices. *arXiv preprint arXiv:1805.01176*, 2018.

SZVETITS, M.; ZDUN, U. Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. *SSM*, dec 2013.

TARVO, A.; SWEENEY, P. F.; MITCHELL, N.; RAJAN, V.; ARNOLD, M.; BALDINI, I. CanaryAdvisor: a statistical-based tool for canary testing (demo). In: *ISSTA*. [S.l.: s.n.], 2015.

THRAMBOULIDIS, K.; VACHTSEVANOU, D. C.; SOLANOS, A. Cyber-Physical Microservices: An Iot-based Framework for Manufacturing Systems. *arXiv preprint arXiv:1801.10340*, 2018.

TOFFETTI, G.; BRUNNER, S.; BLÖCHLINGER, M.; DUDOUET, F.; EDMONDS, A. An Architecture for Self-managing Microservices. In: *AIMC*. [S.l.: s.n.], 2015.

TOFFETTI, G.; BRUNNER, S.; BLÖCHLINGER, M.; SPILLNER, J.; BOHNERT, T. M. Self-managing cloud-native applications: Design, implementation, and experience. *FGCS*, Elsevier, v. 72, p. 165–179, 2017.

VALANCIUS, V.; LAOUTARIS, N.; MASSOULIÉ, L.; DIOT, C.; RODRIGUEZ, P. Greening the internet with nano data centers. In: ACM. *ICENET*. [S.l.], 2009. p. 37–48.

VOGEL, T.; GIESE, H. Adaptation and abstract runtime models. In: ACM. *ICSE*. [S.l.], 2010. p. 39–48.

WARD, J. S.; BARKER, A. Observing the clouds: a survey and taxonomy of cloud monitoring. *JCC*, v. 3, n. 1, Dec 2014.

WEIGAND, H.; HEUVEL, W.-J. van den; HIEL, M. Rule-based service composition and service-oriented business rule management. In: CITESEER. *ReMoD*. [S.l.], 2008. p. 1–12.

WEYNS, D.; ANDERSSON, J. On the challenges of self-adaptation in systems of systems. In: ACM. *IWSESS*. [S.l.], 2013. p. 47–51.

WEYNS, D.; SCHMERL, B.; GRASSI, V.; MALEK, S.; MIRANDOLA, R.; PREHOFER, C.; WUTTKE, J.; ANDERSSON, J.; GIESE, H.; GÖSCHKA, K. M. On patterns for decentralized control in self-adaptive systems. In: *SESAS*. [S.l.]: Springer, 2013. p. 76–107.

YOKOYAMA, D.; SCHULZE, B.; KLOH, H.; BANDINI, M.; REBELLO, V. Affinity aware scheduling model of cluster nodes in private clouds. *Journal of Network and Computer Applications*, Elsevier, v. 95, p. 94–104, 2017.

YU, H.; JOSHI, P.; TALPIN, J.-P.; SHUKLA, S.; SHIRAISHI, S. The Challenge of Interoperability: Model-based Integration for Automotive Control Software. In: *DAC*. New York, NY, USA: ACM, 2015. (DAC '15), p. 58:1–58:6.

ZIMMERMANN, O. Microservices Tenets: Agile Approach to Service Development and Deployment. *CSRD*, v. 32, n. 3, p. 301–310, 2016.

ZIPKIN. 2017. Last Accessed: June 2017. Available at: <http://zipkin.io/>.

ZÚÑIGA-PRIETO, M.; INSFRAN, E.; ABRAHÃO, S.; CANO-GENOVES, C. Automation of the Incremental Integration of Microservices Architectures. In: *Complexity in Information Systems Development*. [S.l.]: Springer, 2017. p. 51–68.

APPENDIX A – Z3 OPTIMIZATION MODEL

```
# Sam Bayless, 2017
# License: CC0 and/or MIT

import os
import sys

from z3 import *
import random
import json
random.seed(0)

bitwidth = 16

#create a new Z3 solver instance.

s =  Optimize() # replace to Solver() for the planning OA-modified

# Use Optimize() because we are doing optimization below. If not using Z3 for
# optimization, this should instead be s= Solver()

class Job:

    def __init__(self, name,required_cpu,required_memory):
        self.name = name

        # IntVal(python_value) creates a Z3 constant integer value, as opposed
        # to a Z3 variable.  In some cases, Z3 will implicitly convert a python
        # value (eg, an int) into a Z3 constant, but in some cases it does not,
        # so it helps to avoid bugs if you always explicitly create python
        # constants using IntVal, BoolVal, or RealVal If you were instead
        # creating Z3 variables (rather than Z3 constants), you would use
        # Int(), Real(), or Bool()

        self.required_memory = IntVal(required_memory)
        self.required_cpu = IntVal(required_cpu)

class Node:
    def __init__(self, name, available_cpu,available_memory):
        self.name = name
        self.available_cpu = IntVal(available_cpu)
        self.available_memory = IntVal(available_memory)

    def __hash__(self):
        return hash(self.name)

    def __eq__(self, other):
        return isinstance(other, Node) and self.name == other.name

expected_runtimes=dict()
```

```
50
51 data = json.load(open('model.json'))
52
53
54 def dictToNode(d):
55     return Node(d['name'], d['cpu'], d['memory'])
56
57 def dictToJob(d):
58     return Job(d['name'], d['cpu'], d['memory'])
59
60 affinities = dict()
61 jobs = []
62 nodes = set()
63 for affinity in data:
64     j1 = dictToJob(affinity['source'])
65     j2 = dictToJob(affinity['target'])
66     jobs.append(j1)
67     jobs.append(j2)
68     assert (j1 != j2)
69     affinities[(j1,j2)] = BitVecVal(affinity['affinity'], bitwidth)
70     n1 = dictToNode(affinity['source']['host'])
71     n2 = dictToNode(affinity['target']['host'])
72
73     if n1 not in nodes:
74         nodes.add(n1)
75     if n2 not in nodes:
76         nodes.add(n2)
77
78 # The following constraints force Z3 to find a valid placement of jobs to nodes
79 # (but do not yet attempt to maximize affinity)
80 job_placements = dict()
81 for j in jobs:
82     job_placements [j]=dict()
83
84 node_placements = dict()
85 for n in nodes:
86     node_placements [n]=[]
87
88
89 for j in jobs:
90     #each job has to be placed on exactly one node
91
92     node_choices = []
93     node_choices_pb = []
94     for n in nodes:
95
96          # For each job node pair, create a Boolean variable in Z3.  If that
97          # Bool is assigned true, then we interpret it to mean that Z3 placed
98          # this job on this node.  Note: All Z3 variables (not constants) must
99          # be given a unique string name, which must be different from any
100         # other Z3 variables. In this case, this Bool variable is given the
101         # name "place_%s_on_%s"%(j.name,n.name)
102         p = Bool("place_%s_on_%s"%(j.name,n.name));
103         node_choices.append(p)
104         node_choices_pb.append((p,1))
105         node_placements[n].append((p,j))
106         job_placements[j][n] =p
```

```
107
108     #Assert that each job is placed on _exactly_ one node
109     # there are several encodings that can achieve this constraint, and you may
110     # need to play around with a few to find the one that has the best
111     # performance. Below I am using a Pseudo-Boolean encoding. But arithmetic
112     # encodings are also possible (commented out below)
113     s.add(z3.PbEq(node_choices_pb, 1)) # this not work for just one node
114     #s.add(Sum([If(b, 1, 0) for b in node_choices]) == 1)
115
116
117 # assert that, for each node, the sum of the jobs placed on that node do not
118 # exceed the available CPU this is 'hard' constraint - Z3 will refuse to find a
119 # solution at all, if there does not exist a placement that respects these
120 # constraints
121 for n in nodes:
122     placements =  node_placements[n]
123     sum_used_cpu = Sum([If(p,j.required_cpu,0) for p,j in placements])
124     s.add(sum_used_cpu<=n.available_cpu)
125     n.sum_used_cpu = sum_used_cpu
126
127 # assert that, for each node, the sum of the jobs placed on that node do not
128 # exceed the available memory
129 for n in nodes:
130     placements =  node_placements[n]
131     sum_used_memory = Sum([If(p,j.required_memory,0) for p,j in placements])
132     s.add(sum_used_memory<=n.available_memory)
133     n.sum_used_memory = sum_used_memory
134
135
136
137 # maximize the sum total affinity
138 # there are other possible ways we could set up this objective function for the
139 # affinity score.
140 affinity_score = BitVecVal(0,bitwidth)
141
142
143 for (j1, j2),val in affinities.items():
144     both_jobs_on_same_node=[]
145     for n in nodes:
146         both_jobs_on_this_node = And(job_placements[j1][n],job_placements[j2][n])
147         both_jobs_on_same_node.append(both_jobs_on_this_node)
148
149     # if both jobs are placed by Z3 on the same node, then add their affinity
150     # value to the affinity score
151     affinity_score = \
152         If(Or(both_jobs_on_same_node),affinity_score+val,affinity_score)
153
154 s.maximize(affinity_score )
155 # The objective function should be an integer (or real) that Z3 will minimize
156 # or maximize.
157
158 r =  s.check()
159
160 if r==sat:
161     # attempt to solve the instance, and return True if it could be solved
162
163     m = s.model()
```

```
164     # the model contains the actual assignments found by Z3 for each variable
165
166     # print out the objective function we are minimizing m.evaluate(x,True)
167     # extracts the sat solver's solution from the model and then .as_long()
168     # converts that solution into a python long that we can print
169     a = m.evaluate(affinity_score,model_completion=True).as_long()
170     print("Affinity score is %d" % (a))
171     assert(a>=0)
172
173     # print out the allocation found by Z3
174     print("[")
175     for j in jobs:
176         placements = job_placements[j]
177         n_found=0
178         for n,p in placements.items():
179             val = m.evaluate(p, True)
180             if val:
181                 assert(n_found==0)
182                 n_found+=1
183                 print('{"job":"%s", "host":"%s"},'%(j.name, n.name))
184
185         assert(n_found==1)
186         # sanity check: each job should be placed on exactly one node
187     print("{}]")
188     #sanity checking the cpu/ram requirements
189     for n in nodes:
190         cpu_usage = m.evaluate(n.sum_used_cpu,True).as_long()
191         available_cpu = m.evaluate(n.available_cpu,True).as_long()
192         assert(cpu_usage<=available_cpu)
193
194         memory_usage = m.evaluate(n.sum_used_memory, True).as_long()
195         available_memory = m.evaluate(n.available_memory, True).as_long()
196         assert (memory_usage <= available_memory)
197
198
199 else:
200     print("[]")
```