Published in: Proceedings of the 1st IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS 2020), Washington, DC, Washington, USA, August 17-21 2020.
DOI: https://doi.org/10.1109/ACSOS49614.2020.00023

Hierarchical Scaling of Microservices in Kubernetes

Fabiana Rossi, Valeria Cardellini, Francesco Lo Presti
DICII, University of Rome Tor Vergata, Italy
{f.rossi,cardellini}@ing.uniroma2.it, lopresti@info.uniroma2.it

***Abstract*—In the last years, we have seen the increasing adoption of the microservice architectural style where applications satisfy user requests by invoking a set of independently deployable services. Software containers and orchestration tools, such as Kubernetes, have simplified the development and management of microservices. To manage containers' horizontal elasticity, Kubernetes uses a decentralized threshold-based policy that requires to set thresholds on system-oriented metrics (i.e., CPU utilization). This might not be well-suited to scale latency-sensitive applications, which need to express requirements in terms of response time. Moreover, being a fully decentralized solution, it may lead to frequent and uncoordinated application reconfigurations.**

In this paper, we present *me-kube* (Multi-level Elastic Kubernetes), a Kubernetes extension that introduces a hierarchical architecture for controlling the elasticity of microservice-based applications. At higher level, a centralized per-application component coordinates the run-time adaptation of subordinated distributed components, which, in turn, locally control the adaptation of each microservice. Then, we propose novel proactive and reactive hierarchical control policies, based on queuing theory. To show that me-kube provides general mechanisms, we also integrate reinforcement learning-based scaling policies. Using me-kube, we perform a large set of experiments, aimed to show the advantages of a hierarchical control over the default Kubernetes autoscaler.

***Index Terms*—Container; Elasticity; Hierarchical control; Kubernetes; Microservices; Self-adaptation.**

I. Introduction

Microservices is an architectural style for developing an application as a suite of autonomous and decoupled services, that communicate using synchronous or asynchronous techniques. Thanks to microservices, a monolithic application can be split into small independent units, each providing a single and well-defined functionality. Most IT companies (e.g., Amazon, Netflix, Spotify, Uber) are switching to microservices to increase their applications' efficiency and scalability in a distributed cloud environment [1]. To effectively process varying workloads and satisfy Quality of Service (QoS), applications are equipped with adaptation capabilities. Exploiting horizontal elasticity, multiple microservice replicas can be dynamically provisioned to process the incoming load in parallel.

Software containers well fit in the landscape of microservices and promise to simplify their deployment and run-time management. A container orchestration tool can automate container provisioning, management, communication, and fault-tolerance. It can be especially useful for deploying and managing complex microservice-based applications during their whole life cycle. Although several orchestration tools exist [2], Kubernetes is the most popular solution in the academic and industrial scenario. However, Kubernetes (like the others) does not provide effective policies for driving the elasticity of latency-sensitive applications. It includes the Horizontal Pod Autoscaler[1], which uses a threshold-based policy that relies on system-oriented metrics to horizontally scale applications. As such, determining a good scaling threshold is cumbersome, because it requires to identify the relation between a system metric (i.e., utilization) and an application metric (i.e., response time), as well as to know the application bottleneck (e.g., in terms of CPU or memory). To manage a microservice-based application, Kubernetes allows to create multiple Horizontal Pod Autoscaler instances, each carrying out the adaptation of a single microservice deployment. Although this decentralized approach can improve scalability, it can also negatively affect the application stability and performance due to frequent and uncoordinated reconfigurations.

In this paper, we propose *me-kube* (Multi-level Elastic Kubernetes), a Kubernetes extension that introduces a hierarchical architecture for controlling the elasticity of microservice-based applications. Me-kube aims to exploit the strengths of centralized and fully decentralized approaches and avoid their drawbacks. The main contributions are as follows.

- We propose a hierarchical architecture, where a high-level Application Manager coordinates the run-time adaptation of subordinated Microservice Managers, which locally control the elasticity of microservices.
- We design and implement me-kube, an extension of Kubernetes that introduces self-adaptation deployment capabilities through the newly designed Application and Microservice Managers components (Section IV).
- We design novel proactive and reactive hierarchical control policies based on queuing theory for scaling out/in microservice-based applications (Section V).
- Using me-kube, we experimentally compare our hierarchical control policies against fully decentralized ones, also including approaches based on reinforcement learning (Sections VI and VII). The experimental results show the advantages of our hierarchical approach with respect to the default threshold-based policy in Kubernetes. Our control policies can efficiently scale the application, so to satisfy its maximum target response time.

II. Related Work

In this section, we analyze existing policies and architectures that deal with the elasticity management of microservice-

[1]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

based applications on cloud resources. We also consider approaches for scaling single containers, because, so far, only few works have specifically targeted the elasticity issues of complex microservice applications.

Elasticity Policies. We can classify the existing deployment policies in *reactive* and *proactive*. A reactive approach determines the adaptation actions relying on the current system state. Conversely, a proactive approach defines reconfigurations relying on a prediction of the future system evolution. So far, only few existing solutions are proactive (e.g., [3], [4]). To adapt the application deployment, existing approaches mainly apply one of the following methodologies: threshold-based heuristics (e.g., [3], [5], [6]) queuing theory (e.g., [7]–[9]), and machine learning solutions (e.g., [10], [11]).

Threshold-based policies are the most popular approaches for elastically scaling containers (e.g., [3], [5], [6], [10]), also in frameworks (e.g., Kubernenets). Usually, static thresholds are exploited for planning adaptation (e.g., [3], [5]). To proactively scale microservices, Bauer et al. [3] propose Chamulteon: it predicts the CPU utilization of each application microservice and then it takes adaptation actions using a static threshold-based policy. Although policies with static thresholds are simple to design, they require a manual threshold tuning that, in general, is not a trivial task. To overcome this issue, Horovitz et al. [10] use reinforcement learning to dynamically adapt the scaling thresholds.

Queuing theory allows to predict the application performance under different conditions of load and number of replicas. So, it is often applied to drive scaling operations (e.g., [8]), also in combination with other techniques (e.g., [3], [9]). The key idea is to model the application as a queuing network system with inter-arrival times and service times having general statistical distributions (e.g., M/M/k, G/G/k). To simplify the analytical investigation, the application is considered to satisfy the Markovian property, thus leading to approximated system behavior. For example, Mao et al. [7] model a four-tier application using queuing theory. Modeling a microservice-based application through a Layered Queuing Network, Gias et al. [9] solve an optimization problem to dynamically control the number of microservice replicas and the relative container CPU share. Bauer et al. [3] and Tesauro et al. [12] combine M/M/k queuing model with threshold-based and machine learning approaches, respectively. An M/M/k model can give inaccurate estimates of response time when the inter-arrival or service time deviate significantly from the exponential distribution. However, as shown in [13] for multi-components application in the context of data stream processing, for most cases, the G/G/k model shows similar capability to the M/M/k one in estimating application performance and identifying the components to scale. To the best of our knowledge, existing solutions propose a centralized control component that takes scaling decisions using the application queuing model. Therefore, these solutions typically do not scale well in a highly distributed environment.

In the field of *machine learning*, reinforcement learning (RL) is a special technique by which an agent can learn how to make good decisions through a sequence of interactions with the environment. Most of the works consider the classic model-free RL algorithms (e.g., [12], [14], [15]), which however suffer from slow convergence rate. To tackle this issue, in [11] we proposed a model-based RL solution that exploits what is known (or can be estimated) about the system dynamics to control the elasticity of containers. Recently, RL policies have been exploited to manage complex systems also in a fully decentralized manner [15].

Control Architectures. Most of the existing solutions adopt centralized controllers to carry out the application adaptation, e.g., [5], [8], [9]. As described in [16] and elaborated in Section III, different architectural patterns can be used in practice to decentralize the self-adaptation functionalities. The most common one is the master-worker decentralization pattern, where decentralized workers only deal with monitoring and reconfiguration enactment (e.g., [4], [11], [17], [18]). Although a centralized master can more easily find better adaptation strategies, it can suffer from limited scalability, especially when applications are deployed in a large-scale environment. To improve scalability and reliability, several fully decentralized architectures have been proposed, e.g., [6], [15]. In such a case, the lack of coordination among the decentralized control components can cause frequent reconfigurations, which can cause instability and adversely affect the application performance. Moreover, designing efficient decentralized control policies is not trivial.

Differently from the above approaches and our previous work [18], we propose me-kube, which relies on a hierarchical control architecture to adapt the elasticity of microservice applications. It aims to take the best of centralized and fully decentralized solutions, thus improving performance and scalability without compromising stability. A hierarchical architecture to control the elasticity was first proposed in [19] but in the different field of distributed data stream processing applications. However, in that work the hierarchical policy relies on a simple token-bucket global policy to limit the number of reconfigurations, which shows some hindrance to effectively control the overall application response time. Within me-kube, we design novel proactive and reactive hierarchical policies based on queuing theory. Exploiting the application performance model, they estimate the microservices response time and accordingly take scaling decisions. Using me-kube, we can also integrate fully decentralized control policies and compare the performance with that obtained by Kubernetes' Horizontal Pod Autoscaler.

III. System Architecture

A. Problem Definition

The microservice architecture is an architectural style that structures the application as a collection of loosely coupled and distributed services. Since the application workload usually changes over time, the number of microservices replicas should be accordingly scaled at run-time so to obtain desirable performance avoiding resource wastage. Multiple microservice replicas can process a subset of the incoming requests in parallel, thus reducing the load per replica and, in turn, the processing latency. In this work, we consider latency-sensitive applications that expose requirements on a target average response time that should not be exceeded (i.e., $R_{\max}$).

To manage and adapt the application deployment, we need an external controller that provides self-adaptation mechanisms and coordinates the microservices scaling actions. The adaptation control loop can be organized following the well established principles of the Monitor, Analyze, Plan and Execute (MAPE) architectural style. In the following, we describe some patterns for decentralizing the MAPE control loop, aiming to identify the most suitable approach to control microservice applications in large-scale environments.

B. Architectural Options for Decentralized Control

The MAPE control loop is composed by four main components: Monitor, Analyze, Plan, and Execute. The *Monitor* component collects data about the application and the execution environment. By analyzing monitoring data, the *Analyze* component determines whether the application deployment should be changed. If an updated is needed, using a specific control policy, the *Plan* component identifies the adaptation action to perform. Ultimately, the *Execution* component implements the deployment changes. A single centralized MAPE control loop may introduces a single point of failure and a bottleneck for scalability, especially when the control system is in charge of a large number of dynamic entities scattered in a distributed environment. To motivate our choice, we briefly review the main features of the most relevant patterns used to decentralize the self-adaptation functions [16].

The *coordinated control pattern* is a fully decentralized solution where multiple peer MAPE loops operate in parallel to manage the system adaptation. Each MAPE component coordinates its operations with the corresponding component of the other peer loops. This pattern improves scalability when the peer coupling is reduced. However, the lack of logically centralized coordination may introduce too frequent and uncoordinated decisions and it is not easy to design a fully decentralized policy that rapidly converges to a globally optimal solution.

The *master-worker control pattern* decentralizes only the execution of the Monitor and Execute components, relieving the burden from the centralized master. In a single loop iteration, the master component collects the monitoring data from the workers, analyzes them, and dispatches the adaptation actions to the decentralized executors. The centralized master allows to more easily design the self-adaptation policies and computes globally optimal reconfiguration strategies. Nevertheless, it may easily become the system bottleneck when it has to manage and plan scaling actions in a large-scale system.

The *hierarchical control pattern* structures the adaptation logic as a hierarchy of (usually complete) MAPE control loops. Each layer works at a different abstraction level, improving scalability without compromising stability. At the lower levels, the MAPE control loops adapt parts of the system under their direct control. The higher-level coordinator takes advantage

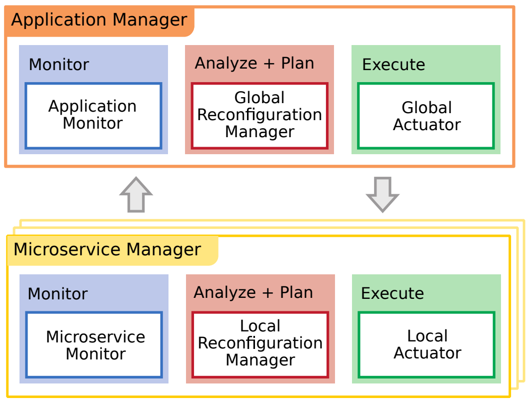


Fig. 1: Architecture of me-kube.

of a broader system view of the system to meet global performance requirements.

We believe that this latter pattern is well suited for adapting the deployment of microservice-based applications. It allows to rule the complexity by decentralizing the low-level adaptation policy, while, at the same time, exploiting the benefit of lightweight higher-level coordination elements. Moreover, a hierarchical architecture can scale well in the face of a high number of microservices, because of the distribution and clear separation of concerns.

IV. Hierarchical Control in Kubernetes

A. Kubernetes

Kubernetes is an open-source orchestration tool that simplifies the deployment, management, and execution of containerized applications. It allows to replicate containers encapsulated in a pod for improving resource usage, load distribution, and fault-tolerance. A pod is the smallest deployment unit in Kubernetes; containers within a pod are co-located and scaled as an atomic entity. Kubernetes can run multiple instances (or replicas) of a pod using a *ReplicaSet*. A ReplicaSet ensures that a given number of pods are up and running. To simplify the deployment of applications, Kubernetes includes the *Deployment* controller. Exposing a higher level abstraction, it simplifies the ReplicaSets update. To expose pods as network services, i.e., to make them accessible from external nodes, Kubernetes uses a *Service*, an abstraction that defines a logical set of pods and a policy by which to access them. The Deployment and Service objects are an easy way to design microservices in Kubernetes. Kubernetes includes a Horizontal Pod Autoscaler to automatically scale pods in a Deployment; nevertheless, it does not provide an approach to coordinate the scaling operations of multi-components application (further details are analyzed in Section VI-B). To fill this gap, we introduce a two-layered MAPE control loop in Kubernetes.

B. Hierarchical Architecture

Figure 1 illustrates the hierarchical architecture of me-kube, which includes multiple decentralized components that implement local MAPE loops. At the lower level, the Microservice

Manager is a per-service distributed entity that runs a MAPE control loop equipped with a local policy. At the higher level, the Application Manager is a centralized MAPE control loop, that coordinates the adaptation of the whole application through a global reconfiguration policy.

Each *Microservice Manager* exploits its limited local view of the system and proposes scaling actions for the controlled microservice . The Microservice Monitor represents the MAPE Monitor components; it collects metrics for the controlled microservice (i.e., response time and input data rate) and also publishes metrics related to any microservice communicating with the controlled one; to exchange such information, a message queue system is used (i.e., RabbitMQ). The Local Reconfiguration Manager analyzes the monitoring information as well as the Application Manager requests to determine if any local reconfiguration action is needed. The available actions are scale-in and scale-out, which reduce and increase the number of microservice replicas, respectively. When the Microservice Manager defines the adaptation actions, it sends a microservice adaptation proposal to the higher layer manager. To reduce communication overhead, the proposal can include a list of multiple scaling actions, each enriched with a cost and a gain. At the higher level, the *Application Manager* coordinates the adaptation of the overall microservice-based application. First, it monitors the application performance. Then, the Global Reconfiguration Manager analyzes this information and can issue reconfiguration requests to the lower level Microservice Managers, which operate in parallel. As soon as it receives the deployment adaptation proposals, the Application Manager uses its global reconfiguration policy to determine the scaling actions to perform. Then the Global Actuator communicates these decisions to the Microservice Local Actuators, which execute them using the standard Kubernetes APIs.

Being loosely coupled with the Kubernetes architecture, our components are general enough and can be easily integrated in other orchestration tools. The modularity of the control loops also allows us to equip them with different scaling policies.

V. Hierarchical Scaling Policy

To scale microservice-based applications at run-time, we model them using open queuing network systems. We consider arbitrary application topologies that do not include the fork-join pattern, which, however, is not commonly adopted in microservice architectures [1]. Anyway, the proposed solution can be easily extended to include it.

A. Performance Model

Given the incoming data rate, our goal is to estimate the microservice-based application response time as we change the replication degree of each microservice.

We model a microservice i as a $M/M/k_i$ queue, where k_i is the number microservice replicas (as also done, e.g., in [3], [12]). We also denote the microservice input data rate as λ_i and its service rate as μ_i. With such variables, we can use well-established results from queuing theory to characterize the microservice and plan reconfiguration actions. We assume each microservice to operate in a stationary condition while determining its scaling.

We use the random variable T_i to denote the response time of a request to microservice i; it represents the time interval between the request arrival and its full processing. In a steady state, the average response time $E[T_i]$ is a decreasing and convex function in k_i. It includes two contributions: the expected queuing delay $E[Q_i](M/M/k_i)$ and the expected processing time $\frac{1}{\mu_i}$:

$$E[T_i](k_i) = E[Q_i](M/M/k_i) + \frac{1}{\mu_i} \qquad (1)$$

By applying the Erlang's delay formula, we can compute $E[Q_i](M/M/k_i)$ as:

$$E[Q_i](M/M/k_i) = \begin{cases} \frac{\pi_0 (k_i\rho_i)^{k_i}}{k_i!(1-\rho_i)^2\mu_i k_i} & \rho_i < 1 \\ \infty & \text{otherwise} \end{cases} \qquad (2)$$

where $\rho_i = \frac{\lambda_i}{k_i\mu_i}$ is the resource utilization of microservice i and π_0 is a normalization term defined as follows:

$$\pi_0 = \left[\sum_{l=0}^{k_i-1} \frac{(k_i\rho_i)^l}{l!} + \frac{(k_i\rho_i)^{k_i}}{k_i!(1-\rho_i)} \right]^{-1} \qquad (3)$$

The average response time of the entire application, $E[T]$, can be computed as the weighted sum of its microservices' response time $E[T_i]$:

$$E[T](\mathbf{k}) = \frac{1}{\lambda_0} \sum_{i=1}^{N} \lambda_i E[T_i](k_i) \qquad (4)$$

where λ_0 is the request arrival rate to the application and $\mathbf{k}$ is a deployment configuration vector $(k_1, k_2, \ldots, k_N)$ containing the number of replicas k_i for each application microservice $i = \{1, 2, \ldots, N\}$, with N the total number of microservices.

Resorting on this performance model, we design a scaling policy to adapt the application deployment at run-time.

B. Hierarchical Policy

Our goal is to adapt the application deployment so to not exceed the target application response time R_{max} (i.e., $E[T] < R_{\text{max}}$). To this end, we conveniently change the number of replicas for each application microservice. When the application response time exceeds its target value, i.e., $E[T] \geq R_{\text{max}}$, we need to identify and scale-out the bottleneck microservices. On the other hand, if the application response time is below the target value, we want to decrease the number of replicas in order to reduce resource wastage (thus improving resource utilization). We formalize the microservice deployment problem as follows:

$$\begin{aligned} \min_{\mathbf{k}} \quad & \sum_{i=1}^{N} k_i \\ \text{s.t.} \quad & E[T](\mathbf{k}) < R_{\text{max}} \end{aligned} \qquad (5)$$

We design a hierarchical policy to solve the deployment problem (5) in a decentralized manner. At the higher level, the Application Manager (i) monitors and analyzes the application performance, (ii) may request reconfiguration proposals to the Microservice Managers, and (iii) accordingly updates the application deployment using a global policy. The global policy takes into account the reconfiguration actions proposed by the decentralized local policies. Each proposal includes the reconfiguration action as well as a *score*, representing the benefit of applying the adaptation action according to local policy perspective. At the lower level, the Microservice Manager local policy uses the performance model to determine the number of microservice replicas; for each new configuration, it computes the reconfiguration score representing the microservice response time variation resulting by enacting the proposed deployment for the microservice. Since the Application Manager drives the deployment reconfiguration, it explicitly requests to the Microservice Manager either scale-out or scale-in reconfiguration proposal. In turn, to optimize the interaction between the two managers, the Microservice Manager returns a set of different reconfiguration proposals.

Microservice Manager Policy. The Microservice Manager local policy implements the Analyze and Plan steps of the decentralized MAPE loop. Its main goal is to propose microservice reconfigurations for the centralized Application Manager. The local policy of each Microservice Manager uses the monitored information to estimate the microservice response time as its replication degree k_i changes. Equation (1) is used to compute response time as a function of k_i, λ_i and μ_i. The microservice input data rate λ_i and its service rate μ_i are provided by the monitoring component of the Microservice Manager. The local policy can be executed in a reactive or proactive mode, according to the provided λ_i value. To proactively change the microservice deployment, we consider the integration of a time series forecasting tool, as discussed later. The local policy computes the scaling reconfiguration proposal as reported in Algorithm 1.

When the Application Manager requires scale-out proposals, the local policy first evaluates the feasibility of any reconfiguration. An unfeasible reconfiguration is thrown if the Application Manager limits the maximum number of replicas below those guaranteeing the microservice utilization $\lambda_i/\mu_i < 1$ (lines 7-10). According to Equation (2), if $k_i < \lfloor \frac{\lambda_i}{\mu_i} \rfloor + 1$, $E[T_i](k_i)$ becomes infinitely large, leading $E[T]$ to infinity as well. If feasible reconfigurations are requested (denoted by S_i^{feas}), the policy estimates the microservice response time with the minimum number of replicas preserving feasibility. Equation (1) is used to this end (line 11). Afterwards, it starts collecting the reconfiguration proposals, by iteratively adding one replica at a time up to the maximum value $K_{\max}$ (lines 12-16). Adding a replica decreases the microservice response time $E[T_i]$ by a factor denoted by δ_i, which represents the reconfiguration score. The reconfiguration proposal, denoted by S_i^{props} in Algorithm 1 and containing the list of $\{k_i, \delta_{k_i}\}$ pairs, is sent to the Application Manager for evaluation.

When a scale-in operation is requested, the policy starts from the current number of microservice replicas, $k_i = \bar{k}_i$, and defines the reconfiguration proposals by reducing k_i by one as long as possible (i.e., $k_i > \lfloor \frac{\lambda_i}{\mu_i} \rfloor + 1$). Removing a replica increases the microservice response time $E[T_i]$ by δ_i; also in this case, it represents the reconfiguration score (lines 28-32).

Algorithm 1 Microservice Manager i Local Policy

```
1: function COMPUTESCALEOUTPROPOSAL($K_{\max}$)
2:     Input: $K_{\max}$, max replicas
3:     Monitor: $\bar{k}_i$, current microservice replicas
4:     Monitor: $\lambda_i$, input data rate
5:     Monitor: $\mu_i$, service rate
6:     Output: $S_i = \{S_i^{\text{feas}}, S_i^{\text{props}}\}$
7:     $k_i \leftarrow \max(\bar{k}_i, \lfloor \frac{\lambda_i}{\mu_i} \rfloor + 1)$
8:     if $k_i > K_{\max}$ then
9:         return UNFEASIBLE
10:    end if
11:    $S_i^{\text{feas}} \leftarrow \{k_i, \lambda_i E[T_i](k_i)\}$          ▷ Computed using (1)
12:    while $k_{\text{i}} < K_{\max}$ do
13:        $k_i \leftarrow k_i + 1$
14:        $\delta_{k_i} \leftarrow \lambda_i \cdot [E[T_i](k_{\text{i}} - 1) - E[T_i](k_{\text{i}})]$
15:        $S_i^{\text{props}}$.push($\{k_i, \delta_{k_i}\}$)
16:    end while
17:    return $\{S_i^{\text{feas}}, S_i^{\text{props}}\}$
18: end function
19:
20: function COMPUTESCALEINPROPOSAL( )
21:    Monitor: $\bar{k}_i$, current microservice replicas
22:    Monitor: $\lambda_i$, input data rate
23:    Monitor: $\mu_i$, service rate
24:    Output: $S_i = \{S_i^{\text{curr}}, S_i^{\text{props}}\}$
25:    $S_i^{\text{curr}} \leftarrow \{\bar{k}_i, \lambda_i E[T_i](\bar{k}_i)\}$          ▷ Computed using (1)
26:    $k_{\text{i}} \leftarrow \bar{k}_i$
27:    $k_{\text{i,min}} \leftarrow \lfloor \frac{\lambda_i}{\mu_i} \rfloor + 1$
28:    while $k_{\text{i}} > k_{\text{i,min}}$ do
29:        $k_i \leftarrow k_i - 1$
30:        $\delta_{k_i} \leftarrow \lambda_i \cdot [E[T_i](k_{\text{i}}) - E[T_i](k_{\text{i}} + 1)]$
31:        $S_i^{\text{props}}$.push($\{k_i, \delta_{k_i}\}$)
32:    end while
33:    return $\{S_i^{\text{curr}}, S_i^{\text{props}}\}$
34: end function
```

Application Manager Policy. The Application Manager global policy implements the Analyze and Plan steps of the centralized MAPE loop. Its main goal is to satisfy the application performance by conveniently accepting the reconfiguration actions proposed by the decentralized Microservice Managers. Intuitively, if the application response time exceeds the target value, i.e., $E[T] > R_{\text{max}}$, the Application Manager scales out the bottleneck microservices until the performance requirement is satisfied. On the other hand, when the application response time is below the target value, the Application Manager evaluates whether resources can be reclaimed (i.e., it evaluates scale-in operations).

Relying on a complete view of the system, the global policy drives the application adaptation at run-time as reported in Algorithm 2. First, it evaluates scale-out operations; then, if no scaling action is required, it evaluates scale-in operations.

To evaluate scale-out actions, the global policies retrieves the reconfiguration proposal from each microservice. Using the microservice response time, it estimates the average application response time as a weighted sum of the microservice response time. If the application response time exceeds R_{max}, it increases the replication degree of the microservice that leads to the largest estimated reduction of $E[T]$ (i.e., with the highest score). This process continues until either $E[T] < R_{\text{max}}$ or all reconfigurations are evaluated (lines 18-23).

Conversely, if the current application deployment satisfies the target response time, no scale-out operation is performed and scale-in actions are evaluated. The scale-in algorithm starts from the current configuration and evaluates reconfigurations only if $E[T]$ is below a scale-in bound referred as $R_{\text{s-in}}$. In such a case, the global policy reduces the number of used resources as much as possible. First, it identifies the microservice with the lowest score, i.e., that leads to the smallest increment of $E[T]$. This process continues until $E[T]$ is below $R_{\text{s-in}}$ or all reconfiguration proposals are evaluated (lines 36-41).

Algorithm 2 Application Manager Global Policy

```
1:  function GLOBALPOLICY( )
2:      Monitor: $\lambda_0$, application input data rate
3:      $\mathbf{k} \leftarrow$ EvaluateScaleOut($\lambda_0$)
4:      if deploymentNotUpdated($\mathbf{k}$) then
5:          EvaluateScaleIn($\lambda_0$)
6:      end if
7:  end function
8:
9:  function EVALUATESCALEOUT($\lambda_0$)
10:     Output: $\mathbf{k} = (k_1, k_2, \ldots, k_N)$, application deployment
11:     $S \leftarrow [\,], \mathbf{k} \leftarrow [\,]$
12:     $E[T](\mathbf{k}) \leftarrow 0$          ▷ Estimated application response time
13:     for all $i \leftarrow 1, \cdots, N$ do
14:         $\{S_i^{\text{feas}}, S[i]\} \leftarrow$ computeScaleOutProposal($K_{\max}$)
15:         $(\mathbf{k}[i], t_i) \leftarrow S_i^{\text{feas}}$
16:         $E[T](\mathbf{k}) \leftarrow E[T](\mathbf{k}) + \frac{t_i}{\lambda_0}$
17:     end for
18:     while $E[T](\mathbf{k}) \geq R_{\text{max}}$ and ($S$ is not empty) do
19:         $j \leftarrow \arg\max_i S[i]$.fetch()          ▷ find $i$ with max $\delta_i$
20:         $(k_j, \delta_j) \leftarrow S[j]$.pop()
21:         $E[T](\mathbf{k}) \leftarrow E[T](\mathbf{k}) - \frac{\delta_j}{\lambda_0}$
22:         $\mathbf{k}[j] \leftarrow k_j$
23:     end while
24:     return $\mathbf{k}$
25: end function
26:
27: function EVALUATESCALEIN($\lambda_0$)
28:     Output: $\mathbf{k} = (k_1, k_2, \cdots, k_N)$, application deployment
29:     $S \leftarrow [\,], \mathbf{k} \leftarrow [\,]$
30:     $E[T](\mathbf{k}) \leftarrow 0$          ▷ Estimated application response time
31:     for all $i \leftarrow 1, \cdots, N$ do
32:         $\{S_i^{\text{curr}}, S[i]\} \leftarrow$ computeScaleInProposal()
33:         $(\mathbf{k}[i], t_i) \leftarrow S_i^{\text{curr}}$
34:         $E[T](\mathbf{k}) \leftarrow E[T](\mathbf{k}) + \frac{t_i}{\lambda_0}$
35:     end for
36:     while $E[T](\mathbf{k}) < R_{\text{s-in}}$ and ($S$ is not empty) do
37:         $j \leftarrow \arg\min_i S[i]$.fetch()          ▷ find $i$ with min $\delta_i$
38:         $(k_j, \delta_j) \leftarrow S[j]$.pop()
39:         $E[T](\mathbf{k}) \leftarrow E[T](\mathbf{k}) + \frac{\delta_j}{\lambda_0}$
40:         $\mathbf{k}[j] \leftarrow k_j$
41:     end while
42:     return $\mathbf{k}$
43: end function
```

Reactive and Proactive Decision Making. The proposed solution can be used to change the application deployment in a reactive or proactive manner. The two approaches can be obtained by changing the way the monitoring component provides the application and microservice input data rate (i.e., λ_0 and λ_i, respectively). In the reactive approach, the Application and Microservice Managers monitor the incoming data rates, which are then used in Algorithms 1 and 2.

In the proactive approach, each Microservice Manager forecasts the data rate values using an AutoRegressive Integrated Moving Average (ARIMA) model. We select ARIMA as it is able to estimate the trend even from a few points. In ARIMA, the future value of a variable is assumed to be a linear function of several past observations and random errors [20]. The AR-part of ARIMA indicates that the evolving variable of interest is regressed on its own prior observations. The MA-part indicates that the regression error is a linear combination of the error terms that occurred in the past. The I-part indicates that the data values have been replaced with the difference between their values and the previous values. The purpose of these features is to create a model that fits the data as well as possible. Non-seasonal ARIMA models are generally denoted ARIMA(p,d,q), where parameters p, d, and q are non-negative integers: p is the order of the autoregressive model, d is the degree of differencing, and q is the order of the moving-average model. The ARIMA parameters should be determined considering the workload characteristics.

VI. Fully Decentralized Scaling Policies

In this section, we present fully decentralized elasticity policies against which we will compare our hierarchical solution. First, we consider our model-free and model-based RL solutions [11]. Then, we describe the default threshold-based scaling solution implemented in Kubernetes.

A. Reinforcement Learning-based Policies

A RL agent learns at run-time how to make good adaptation actions through a sequence of interactions with the environment. One of the main challenges in RL is to find a good trade-off between the exploration and exploitation phases. To minimize the obtained cost, a RL agent must prefer actions that it found to be effective in the past (exploitation). However, to discover such actions, it has to explore new actions (exploration). Differently from model-free RL solutions such as Q-learning, a model-based approach exploits what is known (or can be estimated) about the system dynamics to take decisions and speed-up the learning phase. In [11], we proposed a model-based RL policy that uses an empirical model of the system dynamics to control the elasticity of single containers.

To drive the microservice application elasticity, we define multiple, decentralized, and autonomous RL agents, each controlling a single microservice i. In a single control loop iteration, first, the RL agent determines the microservice state and selects the adaptation action to be performed. Then, according to the monitored application and cluster-oriented metrics (i.e., response time and resource utilization), the RL agent updates the Q-function. The Q-function consists of $Q(s, a)$ terms, which represent the expected long-term cost that follows the execution of action a in state s. The execution of a in s leads to the transition in a new state (i.e., s') and to the

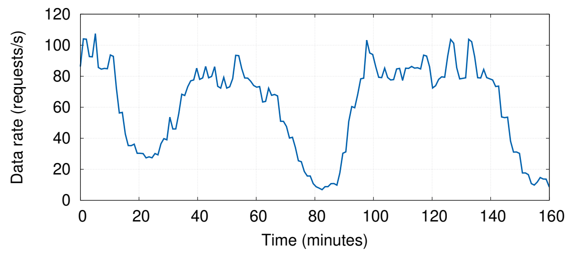


Fig. 2: Workload used for the reference application.

payment of an immediate cost. We define the immediate cost $c(s,a,s')$ as $c(s,a,s') = w_{\text{perf}} \cdot c_{\text{perf}} + w_{\text{res}} \cdot c_{\text{res}} + w_{\text{adp}} \cdot c_{\text{adp}}$, where w_{adp}, w_{perf} and w_{res}, with $w_{\text{adp}} + w_{\text{perf}} + w_{\text{res}} = 1$, are non-negative weights that represent the relative importance of each cost term. The performance penalty c_{perf} is paid whenever the average response time of microservice i exceeds its target value $R_{\text{max,i}}$. The resource cost c_{res} is proportional to the number of microservice replicas. The adaptation cost c_{adp} captures the cost to perform a scaling operation. The immediate cost $c(s,a,s')$ contributes to update the Q-function. Q-learning updates it using a simple weighted average, while the model-based approach directly applies the Bellman equation. Further details on our RL policies can be found in [11].

B. Horizontal Pod Autoscaler

Kubernetes includes the Horizontal Pod Autoscaler (HPA), which employs a threshold-based policy that relies on CPU utilization to horizontally scale applications. Since an application runs in Kubernetes through multiple deployment controllers, one for each microservice, to obtain elasticity we should create multiple HPA instances, each one carrying out the adaptation of a single microservice deployment. First, HPA monitors the CPU utilization of the microservice pods and then scales the number of pods according to the ratio between the target value and the observed value of CPU utilization. Each autoscaler takes scaling decisions in a fully decentralized and uncoordinated manner.

Differently from all the policies we have so far presented, HPA relies on cluster-oriented metrics. Therefore, it might not be well suited to scale latency-sensitive applications, because its usage requires the application administrator to identify the relation between a system-oriented metric (i.e., CPU utilization) and an application-oriented metric (i.e., response time).

VII. Experimental Results

In this section, we use me-kube, our extension of Kubernetes, to evaluate the proposed elasticity policies in a real distributed environment.

A. Experimental Setting

We deploy me-kube on a cluster of 5 virtual machines of the Google Cloud Platform; each virtual machine has 2 vCPUs and 7.5 GB of RAM (type: n1-standard-2). We consider an application that consists of a pipeline of 5 microservices: an API gateway, 3 asynchronous workers (i.e., service1, service2, service3), and a publisher that sends the request response to a message queuing system (i.e., RabbitMQ). As observed in [21], applications with a sequential dependency graph, as the one we consider, are the most common case. HTTP requests from users arrive to the application API gateway, which timestamps and forwards them to the next service1 microservice. Service1 asynchronously processes the incoming request; the processing result is then forwarded to the next service2 microservice. After processing, service2 invokes service3. In turn, service3 sends the response back to the API gateway, which computes the request response time and forwards it to the publisher. The three workers have different processing time per request. To estimate their service rate, we considered each service in isolation: while keeping its parallelism fixed to one, we increased its incoming request rate and monitored the number of requests processed per unit of time. In such a way, we identified their service rate μ_i, with $i = \{1,2,3\}$, as 35, 20, and 30 requests/s, respectively. Service2 is the application bottleneck. The application receives a varying number of requests, represented in Figure 2; it follows the workload of a real distributed application [22], accordingly amplified and accelerated so to further stress the application resource demand. The application exposes a target response time $R_{\text{max}} = 550$ ms, with the scale-in threshold $R_{\text{s-in}} = 400$ ms. The maximum parallelism degree K_{max} is set equal to 10 for each worker, and to 3 for the API gateway and the publisher, being the latter lightweight components that only forward data.

To dynamically scale the microservice-based application at run-time and highlight the benefits of a hierarchical policy, we equip me-kube with our novel queuing-based elasticity policy. To forecast the input rate without relying on a a-priori training set, each manager uses an ARIMA(0,1,1) model, so it is a basic exponential smoothing model. In general, exponential functions are used to assign exponentially decreasing weights over time. We also present the results achieved by the fully decentralized policies described in Section VI by disabling the Application Manager. For the RL policies, each Microservice Manager runs a RL agent that performs scaling actions in isolated manner. We parametrize the RL-based approaches as in [11]. We consider the set of weights $w_{\text{perf}} = 0.90$, $w_{\text{res}} = 0.09$, $w_{\text{adp}} = 0.01$ which correspond to weigh most the microservice response time and to consider adaptation costs as negligible. To set the target response time $R_{\text{max,i}}$ with $i = \{1,2,3\}$, we performed some preliminary experiments. First, we feed the application with a number of requests so that no service was overloaded (i.e., utilization below 70%). Keeping the parallelism fixed, we monitored the response time of the single services and the whole application. With this information, we observed how much time was spent at service i when the application response time was close to R_{max} and accordingly set $R_{\text{max,i}}$ to that value, specifically 150 ms for service1 and service3, and 200 ms for service2. The Application Manager and the fully decentralized policies evaluate scaling decisions every 60 seconds. For sake of comparison, we evaluate also the default HPA of Kubernetes, adding as many autoscaler instances as application microservices.

B. Elasticity Policies Evaluation

We summarize the experimental results in Figures 3-5. We can readily observe that the application deployment significantly varies under the different policies. During the first minutes, all scaling policies violate the target response time; as we can see from Figure 2, this can be explained by observing that the workload starts right away with a peak of requests that progressively then decreases around the 20th minute.

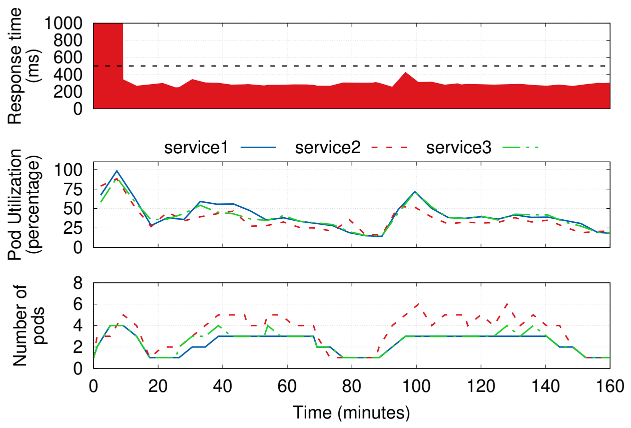
Figure 3 shows the application performance when the hierarchical policy drives the application elasticity in proactive and reactive manner. Although the two approaches have similar performance, the proactive policy results in slightly better application response time. As soon as the data forecasting module can predict the incoming workload, the hierarchical policy can successfully scale the application microservices. Differently from the reactive approach, the proactive policy almost halves the response time violations in the first part of the experiment. The benefit of forecasting is also clear around minute 100, where the hierarchical policy can anticipate the steep load increment and scale-out the bottleneck microservice, i.e., service2. As a consequence, the proactive hierarchical policy meets the application response time requirement during almost all the experiment duration: R_{max} is exceeded only 5.56% of time (and registers 286.81 ms as median response time). The reactive policy takes scaling decision by analyzing the current value of incoming data rate, so it cannot anticipate future dynamics. Furthermore, to perform scale-out operations, Kubernetes gradually creates new instances that are not immediately up and running. As a consequence, the reactive hierarchical scaling policy registers a slightly higher number of R_{max} violations (i.e., 11.11%). In general, the proactive and reactive hierarchical policies deploy a rather high number of microservices' replicas (on average, 7.75 and 7.93, respectively) with an average pods CPU utilization of about 40%. This depends on the estimate of the microservice service rate μ_i, $\forall i \in \{1 \dots N\}$ and on Equation (2), which conservatively approximates the real application behavior. Nevertheless, we can readily see that the scaling policy deploys a number of replicas that follows the application workload (see Figure 2).

To show the accuracy of the prediction algorithm, Figure 6 reports the difference between the real value and the predicted value of the application data rate as well as the absolute error (defined as $|x - x'|$, where x' is the prediction and x the real value). As shown in Figure 6, the absolute error is rather small except when the data rate changes very quickly; in this experiment, the median prediction error is 7 requests/s with an average error of 8.56 requests/s.

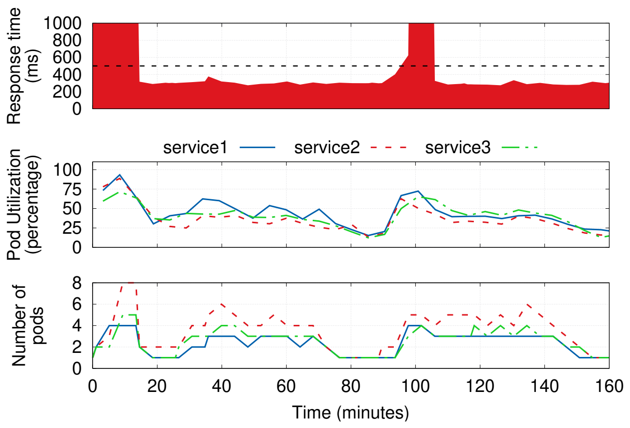
Exploiting the modular architecture of me-kube, we can easily run fully decentralized policies. RL solutions are general and flexible, requiring only to specify the desired deployment objectives. They allow to indicate *what* the user aims to obtain (through the cost function weights), instead of *how* it should be obtained. The RL agent learns the scaling policy in an automatic manner. Figure 4 shows the application performance when the model-free and model-based RL solutions are used. In particular, due to the application complexity, the model-free RL policy cannot learn a good adaptation policy within the time interval of the experiment (i.e., 160 minutes). Figure 4a clearly shows that the lack of a system model and of a coordinator leads the different RL agents to take scaling decisions that are often in contrast with one another. As a consequence, when Q-learning is used, the application response time exceeds R_{max} for 46.20% of the time. Conversely, taking advantage of the system knowledge, the model-based solution drastically reduces the number of R_{max} violations (from 46.20% to 12.03%), as shown in Figure 4b. On average, the number of application instances is higher than in Q-learning (6.37 and 5.12, respectively), and, as a consequence, resource utilization is lower (43.80% and 58.92%, respectively). This strictly follows from the weights cost configuration, for which optimizing the microservice response time is more important than saving resources. We observe that, in a multi-agent setting, the agents indirectly interfere with one another, because the reconfiguration of a microservice impacts on its communicating microservices. As a consequence, the median response time is higher than 50 ms compared to that of the proposed hierarchical policy. The lack of a central coordinator leads to unnecessary application deployment reconfigurations, e.g., as can be seen in the time interval from 40 to 60 minutes (see Figure 4b): service3 is reconfigured several times before the RL agent correctly identifies service2 as the application bottleneck that should be scaled out. This setting shows the complexity of managing microservice application in a fully decentralized manner and the challenges of mapping application requirements onto microservices requirements.

Now, we compare our deployment policies against the Kubernetes autoscaler. Its default threshold-based scaling policy is application-unaware and requires to set a threshold on the average pods CPU utilization. This is a not trivial task, especially for non-CPU intensive applications. We change the scaling threshold of HPA from 50% to 80% of pods CPU utilization. We can observe that small changes in the threshold setting does not lead to significant performance improvements. Due to space limitations, in Figure 5 we report only 50% and 80% threshold setting. Differently from the previous policies, HPA does not correctly detect the application bottleneck. Moreover, we observe that it does not immediately react to load variations; therefore, the microservices' pod CPU utilization can exceed the scaling threshold for a limited time interval, as clearly shown in Figure 5a. Changing the scaling threshold of HPA affects the average pods utilization, which decreases from 58% to 38% when setting the threshold from 80% to 50%. The average number of microservices replicas increases from 4.96 to 8.37. The number of R_{max} violations is around 14% for all the configurations. We observe that some violations are caused by the delayed scaling policy of HPA.

Discussion. We can conclude that the hierarchical policy overcomes the limitation of fully decentralized approaches, by coordinating the adaptation actions among the application mi-

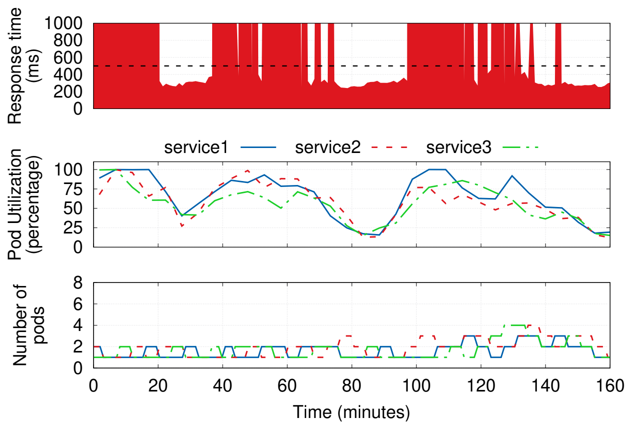


(a) Predictive Decision Making

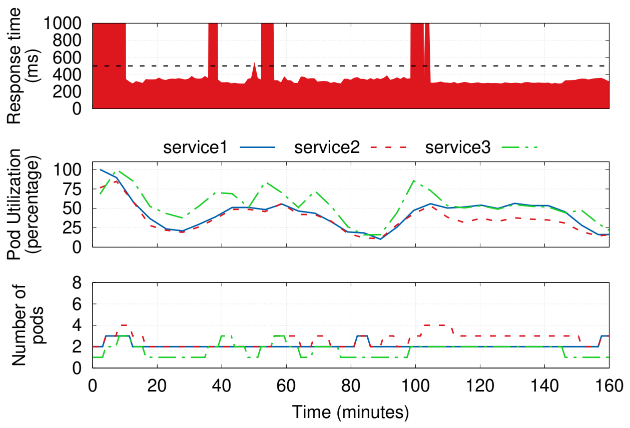


(b) Reactive Decision Making

Fig. 3: Application performance using the hierarchical scaling policy.

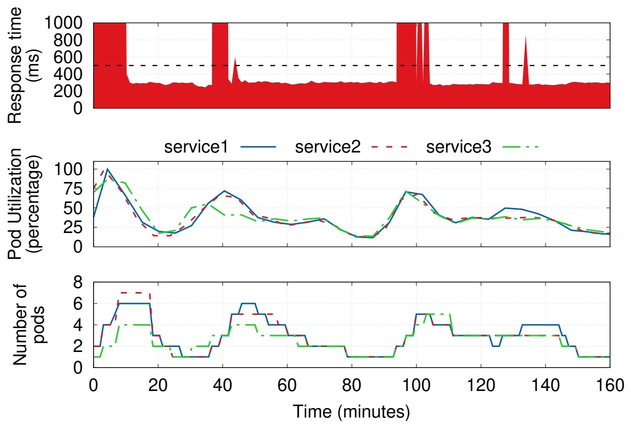


(a) Q-learning

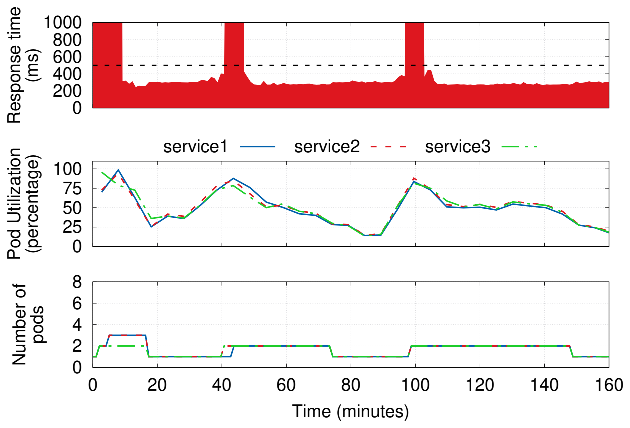


(b) Model-based

Fig. 4: Application performance using the fully decentralized RL-based scaling policies.



(a) Scaling threshold set to 50% of pod utilization



(b) Scaling threshold set to 80% of pod utilization

Fig. 5: Application performance using Kubernetes' Horizontal Pod Autoscaler.

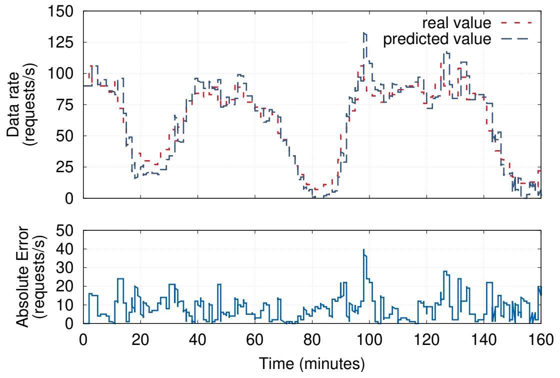


Fig. 6: Run-time prediction error using the ARIMA algorithm.

croservices. During the experiment, the proposed hierarchical scaling policy obtained better performance in terms of reduced application response time. By forecasting the incoming data rate, the hierarchical control policy also drastically reduced the number of $R_{\max}$ violations by anticipating the system dynamics. It is well known that parameterizing correctly a queuing model requires preliminary application profiling. Nevertheless, the experiments have demonstrated that an effective hierarchical policy can improve the overall performance of the system, with respect to a fully decentralized solution. We have also shown how the proposed architecture of me-kube can easily host other policies. In particular, we have evaluated the behavior of two fully decentralized RL-based scaling policies. Differently from Kubernetes' Horizontal Pod Autoscaler, the hierarchical policy and the decentralized RL-based ones take into account application-oriented metrics, thus simplifying the definition of deployment goals.

VIII. Conclusions

In this paper, we have presented me-kube, a Kubernetes extension that introduces a two-layered hierarchical control architecture for adapting microservice-based applications. At the lower level, distributed components control the adaptation of single microservices. At the higher level, a per-application component oversees the overall application reconfiguration and performance. Then, we have designed a novel hierarchical control policy, based on queuing theory, that combines load forecasting and response time estimation to proactively adapt the application deployment. Moreover, we have integrated our fully decentralized RL-based policies in me-kube. Differently from the default Kubernetes scaling solution, the proposed heuristics consider user-oriented metrics (i.e., response time). The presented prototype-based evaluation shows the benefits of the proposed heuristics as well as the importance of using the hierarchical architecture to adapt microservices deployment.

As future work, we plan to design hierarchical policies that jointly control the scaling and placement of microservice applications in a geo-distributed deployment scenario. Moreover, we want to investigate other forecasting algorithms as well as to design solutions for efficiently combining reactive and proactive policies, e.g., along different time scales.

Acknowledgment

We gratefully acknowledge the support we have received from Google with the GCP research credits program.

References

[1] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly, 2016.

[2] E. Casalicchio, "Container orchestration: A survey," in *Systems Modeling: Methodologies and Tools*. Cham: Springer, 2019, pp. 221–235.

[3] A. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, and S. Kounev, "Chamulteon: Coordinated auto-scaling of micro-services," in *Proc. of IEEE ICDCS '19*, 2019, pp. 2015–2025.

[4] M. Imdoukh, I. Ahmad, and M. Alfailakawi, "Machine learning based auto-scaling for containerized applications," *Neural Computing and Applications*, vol. 32, pp. 9745–9760, 2019.

[5] H. Khazaei, R. Ravichandiran, B. Park, H. Bannazadeh, A. Tizghadam, and A. Leon-Garcia, "Elascale: Autoscaling and monitoring as a service," in *Proc. of CASCON '17*, 2017, pp. 234–240.

[6] E. Di Nitto, L. Florio, and D. A. Tamburri, "Autonomic decentralized microservices: The *Gru* approach and its evaluation," in *Microservices: Science and Engineering*. Cham: Springer, 2020, pp. 209–248.

[7] Y. Mao, J. Oak, A. Pompili, D. Beer, T. Han, and P. Hu, "DRAPS: Dynamic and resource-aware placement scheme for Docker containers in a heterogeneous cluster," in *Proc. of IEEE IPCCC '17*, 2017, pp. 1–8.

[8] C. Barna, H. Khazaei, M. Fokaefs, and M. Litoiu, "Delivering elastic containerized cloud applications to enable DevOps," in *Proc. of SEAMS '17*, 2017, pp. 65–75.

[9] A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in *Proc. of IEEE ICDCS '19*, 2019, pp. 1994–2004.

[10] S. Horovitz and Y. Arian, "Efficient cloud auto-scaling with SLA objective using Q-learning," in *Proc. of IEEE FiCloud '18*, Aug 2018, pp. 85–92.

[11] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *Proc. of IEEE CLOUD '19*, July 2019, pp. 329–338.

[12] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A hybrid reinforcement learning approach to autonomic resource allocation," in *Proc. of IEEE ICAC '06*, 2006, pp. 65–73.

[13] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang, "DRS: Auto-scaling for real-time stream analytics," *IEEE/ACM Trans. Netw.*, vol. 25, no. 6, pp. 3338–3352, 2017.

[14] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," in *Proc. of IEEE/ACM CCGrid '17*, 2017, pp. 64–73.

[15] S. M. R. Nouri, H. Li, S. Venugopal, W. Guo, M. He, and W. Tian, "Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications," *Future Gener. Comput. Syst.*, vol. 94, pp. 765–780, 2019.

[16] D. Weyns, B. Schmerl, V. Grassi, S. Malek *et al.*, "On patterns for decentralized control in self-adaptive systems," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS. Springer, 2013, vol. 7475.

[17] M. S. de Brito, S. Hoque, T. Magedanz, R. Steinke, A. Willner, D. Nehls, O. Keils, and F. Schreiner, "A service orchestration architecture for fog-enabled infrastructures," in *Proc. of FMEC '17*, 2017, pp. 127–132.

[18] F. Rossi, V. Cardellini, F. Lo Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with Kubernetes," *Comput. Commun.*, vol. 159, pp. 161–174, 2020.

[19] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Decentralized self-adaptation for elastic data stream processing," *Future Gener. Comput. Syst.*, vol. 87, pp. 171–185, 2018.

[20] R. H. Shumway and D. S. Stoffer, "ARIMA models," in *Time Series Analysis and Its Applications: With R Examples*. Springer, 2017, pp. 75–163.

[21] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn *et al.*, "GrandSLAm: Guaranteeing SLAs for jobs in microservices execution frameworks," in *Proc. of EuroSys '19*. ACM, 2019.

[22] Z. Jerzak and H. Ziekow, "The DEBS 2015 grand challenge," in *Proc. ACM DEBS 2015*, 2015, pp. 266–268.