Dynamically meeting performance objectives for multiple services on a service mesh

Forough Shahab Samani † and Rolf Stadler†
† Dept. of Computer Science, KTH Royal Institute of Technology, Sweden
Email: {foro, stadler}@kth.se

***Abstract*—We present a framework that lets a service provider achieve end-to-end management objectives under varying load. Dynamic control actions are performed by a reinforcement learning (RL) agent. Our work includes experimentation and evaluation on a laboratory testbed where we have implemented basic information services on a service mesh supported by the Istio and Kubernetes platforms. We investigate different management objectives that include end-to-end delay bounds on service requests, throughput objectives, and service differentiation. These objectives are mapped onto reward functions that an RL agent learns to optimize, by executing control actions, namely, request routing and request blocking. We compute the control policies not on the testbed, but in a simulator, which speeds up the learning process by orders of magnitude. In our approach, the system model is learned on the testbed; it is then used to instantiate the simulator, which produces near-optimal control policies for various management objectives. The learned policies are then evaluated on the testbed using unseen load patterns.**

***Index Terms*—Performance management, reinforcement learning, service mesh, digital twin**

I. Introduction

End-to-end performance objectives for a service are difficult to achieve on a shared and virtualized infrastructure. This is because the service load often changes in an operational environment, and service platforms do not offer strict resource isolation, so that the resource consumption of various tasks running on a platform influences the service quality.

In order to continuously meet performance objectives for a service, such as bounds on delays or throughput for service requests, the management system must dynamically perform control actions that re-allocate the resources of the infrastructure. Such control actions can be taken on the physical, virtualization, or service layer, and they include horizontal and vertical scaling of compute resources, function placement, as well as request routing and request dropping.

The service abstraction we consider in this paper is a directed graph, where the nodes represent processing functions and the links communication channels. This general abstraction covers a variety of services and applications, such as a network slice on a network substrate, a service chain on a softwarized network, a micro-service based application, or a pipeline of machine-learning tasks. We choose the service-mesh abstraction in this work and apply it to micro-service based applications.

In this paper, we propose a framework for achieving end-to-end management objectives for multiple services that concurrently execute on a service mesh. We apply reinforcement learning (RL) techniques to train an agent that periodically performs control actions to reallocate resources. A management objective in this framework is expressed through the reward function in the RL setup. We develop and evaluate the framework using a laboratory testbed where we run information services on a service mesh, supported by the Istio and Kubernetes platforms [1],[2]. We investigate different management objectives that include end-to-end delay bounds on service requests, throughput objectives, and service differentiation.

Training an RL agent in an operational environment (or on a testbed in our case) is generally not feasible due to the long training time, which can extend to weeks, unless the state and action spaces of the agent are very limited. We address this issue by computing the control policies in a simulator rather than on the testbed, which speeds up the learning process by orders of magnitude for the scenarios we study. In our approach, the RL system model is learned from testbed measurements; it is then used to instantiate the simulator, which produces near-optimal control policies for various management objectives, possibly in parallel. The learned policies are then evaluated on the testbed using unseen load patterns (i.e. patterns the agent has not been trained on).

We make two contributions with this paper. First, we present an RL-based framework that computes near-optimal control policies for end-to-end performance objectives on a service graph. This framework simultaneously supports several services with different performance objectives and several types of control operations.
Second, as part of this framework, we introduce a simulator component that efficiently produces the policies. Through experimentation, we study the tradeoff of using the simulator versus learning the policies on a testbed. We find that while we lose some control effectiveness due to the inaccuracy of the system model we gain by significantly shortening the training time, which makes the approach suitable in practice. To the best of our knowledge, this paper is the first to advocate a simulation phase a part of implementing a dynamic performance management solution on a real system.

Note that, when developing and presenting our framework, we aim at simplicity, clarity, and rigorous treatment, which helps us focus on the main ideas. For this reason, we choose a small service mesh for our scenarios (which still includes key complexities of larger ones), we consider only two types of control actions, etc. Our plan is to refine and extend the framework in future work, as we lay out in the last section of

the paper.

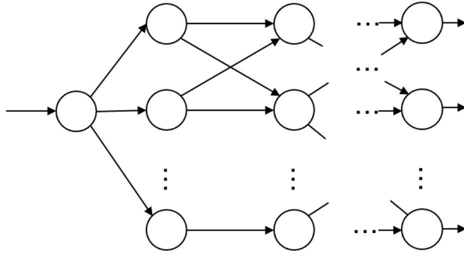
II. Problem formulation and approach

TABLE 1
Notation for formalizing the problem.

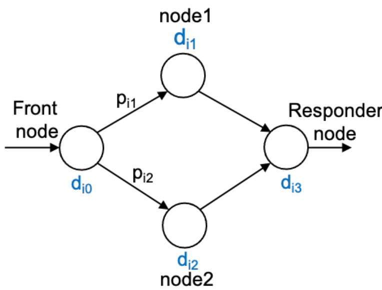
Concept	Notation
Service index	i
Node index	j
Offered load of service i	l_i
Carried load of service i	$l_i^c = l_i(1 - b_i)$
Utility of service i	u_i
Response time objective	O_i
Response time	d_i
Routing weight of service i towards node j	p_{ij}
Blocking rate	b_i

We consider application services built from microservice components, whereby each component performs a unique function. A service request traverses a path on a directed graph whose nodes offer microservices. Figure 1(a) shows a directed graph with a general topology of a microservice architecture, which we call a service mesh. A service is realized as a contiguous subgraph on the service mesh.

While the framework we present in this paper applies to the general service mesh of Figure 1(a), we focus the discussion and experimentation on the smaller configuration shown in Figure 1(b). An incoming service request from a client is processed first by the front node, before being routed to one of the two processing nodes. The processing result is sent to the responder node which sends it to the client.



(a) Directed graph of a service mesh



(b) Use case in this work

Fig. 1. (a) General service mesh; each processing node runs one or more microservices; links are communication channels for service requests. (b) Service mesh of the use case; d_{ij} is the processing delay of a request of service i on node j; p_{ij} is the routing weight of request i towards node j.

We consider two services S_i with different resource requirements on the processing nodes. In the configuration in Figure 1(b), both processing nodes can process requests from both services. We express the service quality in terms of end-to-end delay d_i of a service request, the carried load of a service l_i^c (which we also call the throughput of the service), or the utility u_i generated by a service.

In order to control the service quality, our framework includes control actions. In this work, we consider request routing expressed by $p_{ij} \in [0, 1]$ which indicates the fraction of requests of service i routed to processing node j. In addition, we consider blocking service requests at the front node at the fraction $b_i \in [0, 1]$.

Central to this work are *management objectives*, which capture the end-to-end performance objectives for the services on a given service mesh. These objectives include client requirements, as well as provider priorities. To present and evaluate our framework, we focus on the following three management objectives. See Table 1 for notation.

Management Objective 1 (MO1): the response time of a request of service i is upper bounded by O_i and the overall carried load is maximized:

$$\text{maximize} \sum_i l_i^c \text{ while } d_i < O_i \tag{1}$$

Management Objective 2 (MO2): the response time of a request of service i is upper bounded by O_i and the sum of service utilities is maximized:

$$\text{maximize} \sum_i u_i \text{ while } d_i < O_i \tag{2}$$

Management Objective 3 (MO3): the response time of a request of service i is upper bounded by O_i, the carried load l_i^c of service i is maximized, while service k is prevented from starving (i.e., by specifying a lower threshold l_{min} for service k):

$$\text{maximize } l_i^c \text{ while } d_i < O_i \text{ and } l_k^c > l_{min} \; i \neq k \tag{3}$$

For the management objective M1, M2, or M3, we solve the following problem for the configuration in Figure 1(b). Given the offered load of service i, and the response time d_i, we need to find the control parameters p_{ij} and b_i. We obtain these parameters through reinforcement learning where they represent the optimal policy.

Following the reinforcement learning approach, we formalize the above problem as a Markov Decision Process (MDP), which is a well-understood model for sequential decision making [3][4]. The elements of this formalization for the scenario in Figure 1(b) are:

State space: The state at time t includes the response time and offered load of the services running on the service mesh: $s_t \in \{(d_{i,t}, l_{i,t}) \in \mathbb{R}^{2m} | \; i = 1, ..., m\} = \mathcal{S}$, where m is the number of services.

Action/control space: The action at time t is a vector of the control parameters: $a_t \in \{b_{i,t}, p_{ij,t}| \ i = 1, ..., m, j = 1, 2\} = \mathcal{A}$.

System model: This model provides the new state when specific action is taken in a given state:

$$(d_{i,t+1}, l_{i,t+1}) = f(d_{i,t}, l_{i,t}, b_{i,t}, p_{ij,t}| \ j = 1, 2) \quad i = 1, ..., m \tag{4}$$

Since we train the policy on a known load pattern, the system model we must learn can be simplified, as we argue in Section IV.

Reward function: The reward function expresses the management objective in the reinforcement learning context. The reward functions of all three management objectives are detailed in Section IV.

Learning an effective or close-to-optimal policy on a real system is often infeasible due to the time that it takes for the system to reach a new stable state after an action. On our testbed, training a reinforcement learning agent can require days or even weeks as we show in Section V. For this reason, our framework includes two systems. The first is a lab testbed, on which the system model is learned. The second is a simulation system, which is instantiated with the system model and on which the optimal policy is learned. The learned policy is then transferred to the testbed where it can be evaluated [5].

III. Testbed and Request Generation

Our testbed at KTH includes a server cluster connected through a Gigabit Ethernet switch. The cluster contains nine Dell PowerEdge R715 2U servers, each with 64 GB RAM, two 12-core AMD Opteron processors, a 500 GB hard disk, and four 1 Gb network interfaces. The tenth machine is a Dell PowerEdge R630 2U with 256 GB RAM, two 12-core Intel Xeon E5-2680 processors, two 1.2 TB hard disks, and twelve 1 Gb network interfaces. All machines run Ubuntu Server 18.04.6 64 bits and their clocks are synchronized through NTP [6]. The orchestration layer and the service mesh are realized using Kubernetes (K8) [2] and Istio [1].

On top of the Istio service mesh, we implemented two information services, which we call service 1 and service 2. Both services, upon receiving request with a key, return a corresponding data item from a database. The difference between the services is the structure of data items and the size of the databases.

Figure 2 shows the implementation of the services on the testbed following the configuration given in Figure 1(b). The front node and the responder node in Figure 1(b) are realized as a single node in Figure 2. All nodes are implemented in Python [7] using Flask [8]. The front node provides the web user interface and the other two nodes provide the content for the information service. Each processing node is implemented as a Kubernetes pod [9].

We implemented a load generator in order to emulate a client population. It is driven by a stochastic process that creates a stream of service requests. We realize two load patterns with this generator.

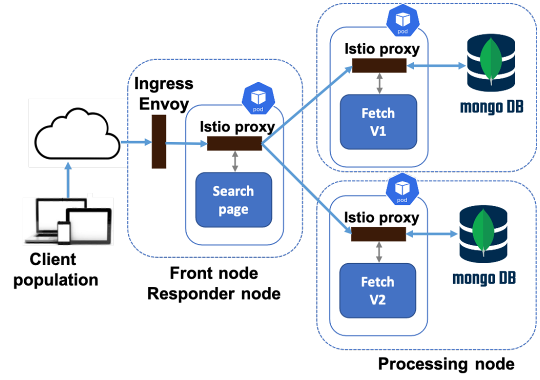


Fig. 2. The architecture of the microservice-based application deployed over the service mesh. See Figure 1(b).

The *random load pattern* produces a stream of requests at the rate of $l_i(t) \sim \mathbb{U}_{\{5,10,15,20\}}$ requests/second. It changes at every time step to a value drawn uniformly at random from $\{5, 10, 15, 20\}$. A time step is 5 seconds on the testbed.

The *sinusoidal load pattern* produces a stream of requests at the rate of $l_i(t) = 12.5 + 7.5 \times \sin(\frac{2\pi}{T}t + \phi)$ requests/second.

IV. Scenarios and Evaluation Set Up

We study three scenarios that help us evaluate our approach to efficiently learn effective policies on the testbed for different management objectives. At the core of each scenario is a management objective that captures the intention of the service provider and a reward function that reflects the management objective in the reinforcement learning framework.

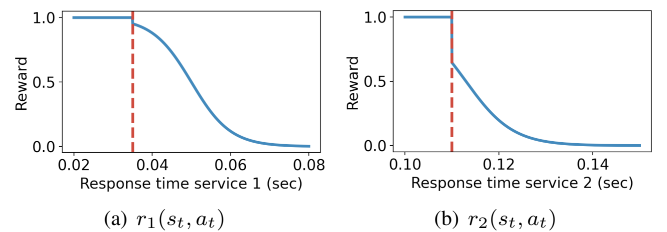
Scenario 1: We run the two services (i.e., $m = 2$) under management objective 1, where the RL agent attempts to maximize the joint throughput while enforcing constraints on the response times under changing load. The management objective MO1

$$\text{maximize } (l_1^c + l_2^c) \text{ while } d_1 < O_1 \text{ and } d_2 < O_2$$

is implemented through the reward function

$$r(s_t, a_t) = l_1^c \times r_1(s_t, a_t) + l_2^c \times r_2(s_t, a_t)$$

whereby r_1 and r_2, depicted in Figure 3 are functions based on $tanh$, capturing the reward of the average response time during time step t for service 1 and service 2, respectively.



(a) $r_1(s_t, a_t)$ (b) $r_2(s_t, a_t)$

Fig. 3. The components of the reward functions for MO1 and MO2.

Scenario 2: We run the two services (i.e., $m = 2$) under management objective 2, where the RL agent attempts to

maximize the sum of the utility functions u_1 and u_2 while enforcing constraints on the response times under changing load. The management objective MO2 for this case is

$$\text{maximize } (u_1 + u_2) \text{ while } d_1 < O_1 \text{ and } d_2 < O_2$$

In this scenario, the service provider gives priority to service 2 over service 1 by setting $u_2 = 5 \times l_2^c$ and $u_1 = l_1^c$. This objective is achieved by maximizing the reward function

$$r(s_t, a_t) = l_1^c \times r_1(s_t, a_t) + 5 \times l_2^c \times r_2(s_t, a_t)$$

where r_1 and r_2 are defined above.

Scenario 3: We run the two services (i.e., $m = 2$) under management objective 3, where the RL agent attempts to maximize the throughput of service 2 while enforcing constraints on the response time of service 2 and guaranteeing a minimum throughput of l_1^{min} to service 1 under changing load. The management objective MO3:

$$\text{maximize } l_2^c \text{ while } d_2 < O_2 \text{ and } l_1^c > l_1^{min}$$

Like in scenario 2, the service provider gives priority to service 2 over service 1 but prevents service 1 from starving. This objective is achieved by maximizing the reward function

$$r(s_t, a_t) = l_2^c \times \big(r_3(s_t, a_t) + r_2(s_t, a_t)\big)$$

where r_3 is a function based on $tanh$, capturing the reward of the carried load during time step t for each service.

Common to all scenarios is the reinforcement learning model. The state of the system at time t is

$$s_t = (d_{1,t}, d_{2,t}, l_{1,t}, l_{2,t})$$

The action the agent takes at time t is

$$a_t = (p_{11,t}, p_{21,t}, b_{1,t}, b_{2,t})$$

We discretize the action space, which facilitates estimating the highest achievable performance of the system. We discretize every component of the action space by allowing six values so that

$$\mathcal{A} = \{[0, 0, 0, 0], [0, 0, 0, 0.2], [0, 0, 0, 0.4], ..., [1.0, 1.0, 1.0, 1.0]\}$$

System model: We learn the system model through testbed observation and supervised learning. Measurements on the testbed suggest that the delay at time step $t + 1$ does not depend on the delay at time step t. Therefore, the system model in Equation 4 simplifies to

$$(d_{1,t+1}, d_{2,t+1}) = f(l_{1,t}, l_{2,t}, p_{11,t}, p_{21,t}, b_{1,t}, b_{2,t}) \quad (5)$$

Optimal policy: To obtain the optimal policy for a given state, we compute the rewards of all possible actions using the system model. We call the highest reward the *optimal reward*, and the actions giving this reward determine the *optimal policy*.

Evaluation metric: The metric measures to which extent the management objective is met in a specific scenario. We define it as the average of the obtained reward divided by the optimal reward and call it ANR (Average Normalized Reward):

$$ANR = \frac{1}{T} \sum_{t=1}^{T} \frac{\text{obtained reward(t)}}{\text{optimal reward(t)}} \quad (6)$$

The value of this metric is between 0 and 1. 1 means the management objective is always met. Sometimes, we measure the performance of the agent for specific time step t and call this metric Normalized Reward (NR) at time t.

Learning the system model: We run both services on the testbed under varying load, whereby at each time step t the load for both services changes independently, taking values uniformly at random from the set $\{5, 10, 15, 20\}$ requests per second. Each time step is 5 seconds. In addition, actions are taken randomly from the action space $\mathcal{A}$. We run the system for $45\,343$ time steps and collect the data $(d_1(t), d_2(t), l_1(t), l_2(t), p_{11}(t), p_{21}(t), b_1(t), b_2(t))$ as a trace.

Taking this trace, we learn the system model using random forest regression [10][11] with input $(l_1, l_2, p_{11}, p_{21}, b_1, b_2)$ and target (d_1, d_2). The same system model is used to learn the control policies for all management objectives.

Learning the control policies: We use the Stables Baselines3 library [12] to implement the RL agent, selecting the PPO method. To train the agent, we choose a neural network of size $[64, 64]$, a batch size of 64, a distance between updates of $1\,024$ steps, the learning rate $\alpha = 0.001$, and the discount factor $\gamma = 0$. We find that the number of iterations for the agent to converge is scenario-specific and is in the order of $10\,000$.

V. Evaluation results and discussion

Training the RL agent: Control policies are learned on a simulator which acts as the environment for the RL agent. At each time step, the simulator determines the current load and executes the function of the system model (Equation 5). Note that this simulator is very different from a discrete event simulator.

When the simulation starts, the agent is initialized with a random policy. During a simulation run, we track the learning process by monitoring the reward and the ANR (Equation 6). We terminate the simulation when these metrics have sufficiently converged. We need to train the agent for each management objective separately, because a control policy is specific to an objective. However, control policies can be computed in parallel.

Evaluation of learned policies: First, we evaluate a learned policy on the simulator. Given a management objective and a related scenario, we run the scenario for both load patterns (see Section III) and measure the ANR. Since the random load pattern has been used for training the agent, we expect close to optimal performance. The sinusoidal load pattern has not been seen by the agent and allows us to assess how the policy generalizes to unseen load.

Second, we evaluate the learned policy on the testbed. We run

each scenario on the testbed for both load patterns. Like above we measure the performance of the RL agent by computing the ANR. Comparing the performance of the RL agent on the simulator with that on the testbed gives us an indication of the accuracy of the learned system model.

Evaluation Scenario 1: We run the simulator for Scenario 1. For each time step, the simulator computes the current load for both services, executes the RL agent to obtain the control vector, determines the related reward, and computes the new state. During the simulation run, we compute the ANR and produce the learning curve shown in Figure 4. We see from the curve that the agent learns an increasingly effective policy. The ANR value approaches 0.98 after some $2\,000$ time steps.

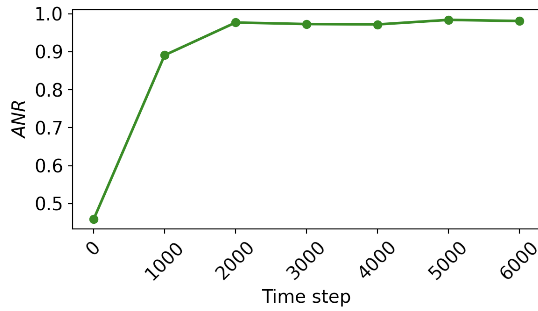
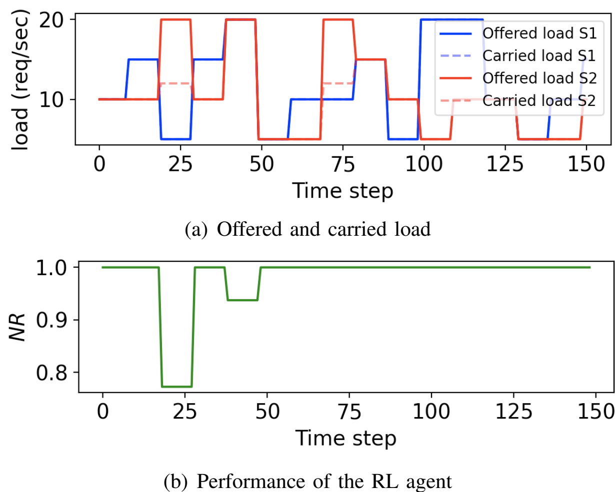


Fig. 4. The learning curve of the RL agent for performance metric ANR on the simulator running Scenario 1.

Figure 5(a) shows the offered and carried load for both services during a time interval of the 150 time steps. We observe that the agent blocks some of the requests of service 2 but none of service 1. Figure 5(b) shows the performance of the agent in NR for the same time period. We see that there are time intervals where the policy is not optimal, i.e. $NR < 1$. We explain this by the fact that the learned policy did not fully converge during the training period and by the probabilistic behavior of the PPO agent.

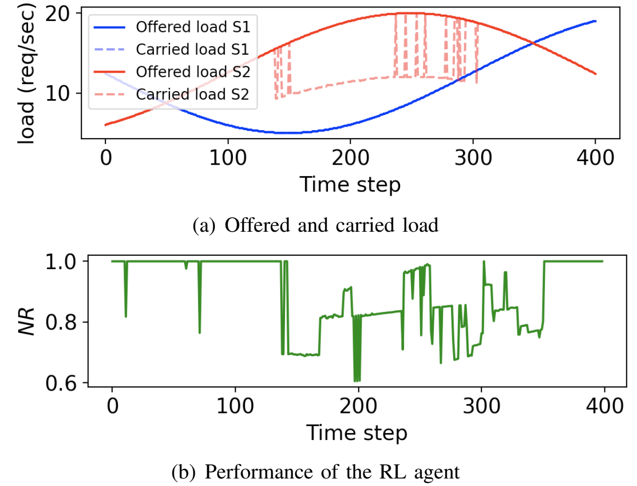


(a) Offered and carried load

(b) Performance of the RL agent

Fig. 5. (a) Offered and carried load of both services for random load in Scenario 1. (b) Performance of the RL agent in NR. Measurements taken from simulator.

Figure 6 relates to Scenario 1 under the unseen, sinusoidal load pattern during an interval of 400 time steps. This experiment allows us to evaluate to which extent the learned policy generalizes to an unseen load pattern. Similar to Figure 5, Figure 6(a) shows the offered and carried load for both services, and Figure 6(b) depicts the performance of the agent for the same time period. We observe that, as with the random load pattern, the agent sometimes blocks requests of service 2 when the system experiences high load. Also, we see that the NR value can be lower than 1 and can change significantly over time. We explain the sudden changes with the discretized action space, which limits the actions the agent can take.



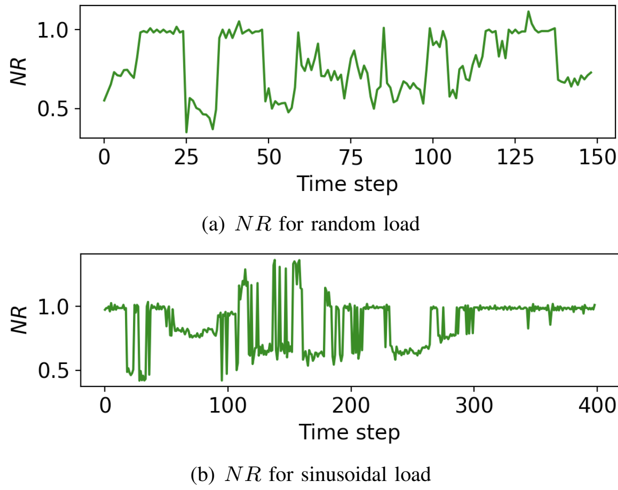
(a) Offered and carried load

(b) Performance of the RL agent

Fig. 6. (a) Offered and carried load of both services for sinusoidal load in Scenario 1. (b) Performance of the RL agent in NR. Measurements taken from simulator.

Figure 7 shows the performance of the agent on the testbed for Scenario 1. Figure 7(a) gives the NR values for a time window under the random load pattern, and Figure 7(b) shows the same metric under the sinusoidal load pattern. Surprisingly, the NR is sometimes larger than 1, which seems to contradict its definition (Equation 6). When checking the data, we find that the optimal reward is wrongly computed if this happens, because of the inaccuracy of the system model.

Due to lack of space, we do not include the detailed evaluation of Scenarios 2 and 3, which relate to management objectives 2 and 3. One notable observation is that the RL agent often takes different actions for the same load pattern when the management objective is different. An example can be seen in Figure 8, which shows the offered and carried load under the sinusoidal load pattern on the simulator for management objective 2. This objective prioritizes service 2 over service 1. We see that the agent blocks some requests of service 1 but none of service 2 between time steps 150 and 230. In contrast, Figure 6(a) shows a case where requests of service 2 are dropped but none of service 1, for the same load pattern. The reason for the difference is that management objective 1 differentiates between services according to their respective resource consumption, while management objective 2 differentiates according to their respective utility.



(a) NR for random load

(b) NR for sinusoidal load

Fig. 7. Performance of the learned policy on the testbed for the both load patterns.

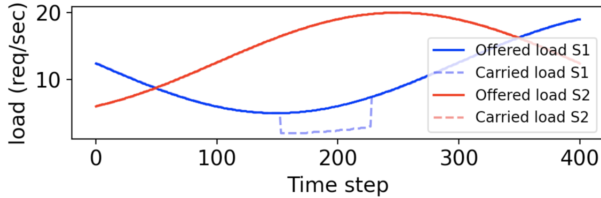


Fig. 8. The offered and carried load for Scenario 2, on the simulator for the sinusoidal load pattern.

Table 2 shows the performance of the RL agent for all three scenarios on the simulator and the testbed. We observe that the agent achieves higher performance on the load pattern it has been trained with (i.e., the random load pattern) than on unseen load (i.e., the sinusoidal load pattern). We expect such a difference in performance because the policy has been optimized for the seen load values. Second, we observe that the performance on the simulator is better than on the testbed for the same load pattern. Similarly, we expect such a difference in performance, because the policy has been computed using the learned system model. On the simulator, the system reacts to an action of the agent according to the system model. On the testbed however, the reaction of the system may differ if the system model is not accurate. Therefore, the difference in performance can be explained by the error estimating the function f in Equation 5.
Overall, our evaluation shows that the agent achieves on the testbed an average ANR value above 0.8 for unseen load. This performance can be compared to a baseline random policy, where the action parameters $(p_{11}, p_{21}, b_1, b_2)$ are selected uniformly at random from the action space, which results in an ANR value below 0.59.

We discuss the benefits and drawbacks of including a simulator in our framework. Training the agent on the simulator reduces the policy effectiveness by 0.11 ANR on average on unseen load (see Table 2). However, it significantly reduces the overall training time. For the studied scenarios, this requires some 2×10^4 measurement rounds, each lasting a time step of 5 seconds, resulting in 28 hours of monitoring time.

Hyper-parameters tuning and learning effective policy for all scenarios on the testbed needs $2\,499$ hours. However, this time can be reduced to an hour on the simulator. Therefore, the training time is reduced by a factor of at least 86. The reduction in time is achieved at the cost of reducing the policy effectiveness by 0.11 ANR.

TABLE 2
Performance of RL agent in all scenarios, running on the simulator and the testbed. Values are given in ANR. A value of 1 means optimal performance where the management objective is met at all times.

Scenario	Environment	Load pattern	ANR
1	simulation	random	0.99
1	simulation	sinusoidal	0.95
1	testbed	random	0.85
1	testbed	sinusoidal	0.90
2	simulation	random	0.96
2	simulation	sinusoidal	0.91
2	testbed	random	0.81
2	testbed	sinusoidal	0.80
3	simulation	random	0.98
3	simulation	sinusoidal	0.97
3	testbed	random	0.85
3	testbed	sinusoidal	0.80

VI. Related Work

Table 3 lists recent works that relate to the discussion in this paper. The table categorizes the works based on the applied approach, the service abstraction, the resource management function, and the environment in which the agent learns the policy. Many papers use reinforcement learning to control resource management functions [13][15][14][16]. Also, while all papers use a directed graph as service abstraction, some of them specifically consider a microservice-based service mesh, like us in this work [13][15][18][19][20][16][17]. Some works are based on implementation [18][20], while the rest use only simulation studies for evaluation [13][15][19][14][16][17]. [14] is similar to our work in the sense that the system model is learned from measurements. [15] is close to our approach since the authors consider several end-to-end performance objectives.

All works in Table 3 study one or more resource management functions, such as scaling, placement, or routing, for a particular use case. When designing a function, the authors associate a performance objective like restricting end-to-end delays for a routing function. We take a different approach with our framework: we start with an end-to-end performance objective and direct the set of resource management functions to jointly meet it. The RL agent takes a combination of actions to meet the objective, whereby one type of action corresponds to one resource management function. For instance, in the scenarios we study in this paper, an end-to-end delay objective is achieved through request routing and blocking. It could as well be achieved through scaling or through a combination of all three functions. In this sense, our approach is top-down and

TABLE 3
Related work

Paper	Control through RL	RL System model	Service abstraction	Resource management function	Simulation Implementation
[13], Rossi 2020	yes	learned from measurements	service mesh	scaling	simulation
[14], Chinchal 2018	yes	learned from measurements	cellular network	routing	simulation
[15], Schneider, 2021	yes	simulation study only	service chain	routing and placement	simulation
[16], Grag 2021	yes	simulation study only	service mesh	placement	simulation
[17], Ashok 2021	no	-	service mesh	cross-layer routing	simulation
[18], Rajib Hossein 2022	no	-	service mesh	scaling	implementation
[19], Yu 2020	no	-	service mesh	scaling	simulation
[20], Lin 2021	no	-	service mesh	orchestration	implementation
This paper	yes	learned from measurements	service mesh	routing and blocking	simulation and implementation

more general, while virtually all related work we are aware of is bottom-up and use-case specific.

VII. Conclusions and future work

We demonstrated how end-to-end management objectives under varying load can be met through periodic control actions performed by a reinforcement learning agent. By computing near-optimal control policies on a simulator, effective control on a testbed could be achieved in several scenarios with applications on a service mesh, which suggests that the approach and framework we propose is practical. As mentioned in the introduction section, we kept the setup and scenarios simple in order to focus on the totality of the framework. Our planned work includes refinement and extension of the framework towards applicability in practice.

Acknowledgements

The authors are grateful to Andreas Johnsson, Farnaz Moradi, Jalil Taghia, Xiaou Lan, and Hannes Larsson with Ericsson Research for fruitful discussion around this work. The authors thank Kim Hammar and Xiaoxuan Wang at KTH for their helpful comments on a draft of this paper. This research has been partially supported by the Swedish Governmental Agency for Innovation Systems, VINNOVA, through project ANIARA.

References

[1] Istio community, “Simplify observability, traffic management, security, and policy with the leading service mesh,” 2017. [Online]. Available at: https://istio.io/, Accessed on: June 7, 2022.

[2] Kubernetes community, “Production-grade container orchestration,” 2014. [Online]. Available at: https://kubernetes.io/, Accessed on: June 7, 2022.

[3] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[4] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[5] K. Hammar and R. Stadler, “Intrusion prevention through optimal stopping,” *IEEE Transactions on Network and Service Management*, pp. 1–1, 2022.

[6] NTP, 2016. [Online]. Available at: http://www.ntp.org/, Accessed on: June 7, 2022.

[7] Python, “Python,” 2000. [Online]. Available at: https://www.python.org/, Accessed on: June 7, 2022.

[8] Flask community, “Flask,” [Online]. Available at: https://flask.palletsprojects.com/en/2.1.x/, Accessed on: June 7, 2022.

[9] Kubernetes community, “Pods,” 2014. [Online]. Available at: https://kubernetes.io/, Accessed on: June 7, 2022.

[10] L. Breiman, “Random Forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[11] Sklearn communiti, “sklearn.ensemble.RandomForestClassifier,” 2007. [Online]. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html, Accessed on: June 7, 2022.

[12] Stable Baselines3 community, “Ppo,” [Online]. Available at: https://www.python.org/, Accessed on: June 7, 2022.

[13] F. Rossi, V. Cardellini, and F. L. Presti, “Self-adaptive threshold-based policy for microservices elasticity,” in *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2020, pp. 1–8.

[14] S. Chinchali, P. Hu, T. Chu, M. Sharma, M. Bansal, R. Misra, M. Pavone, and S. Katti, “Cellular network traffic scheduling with deep reinforcement learning,” in *Thirty-second AAAI conference on artificial intelligence*, 2018.

[15] S. Schneider, R. Khalili, A. Manzoor, H. Qarawlus, R. Schellenberg, H. Karl, and A. Hecker, “Self-learning multi-objective service coordination using deep reinforcement learning,” *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 3829–3842, 2021.

[16] D. Garg, N. C. Narendra, and S. Tesfatsion, “Heuristic and reinforcement learning algorithms for dynamic service placement on mobile edge cloud,” *arXiv preprint arXiv:2111.00240*, 2021.

[17] S. Ashok, P. B. Godfrey, and R. Mittal, “Leveraging service meshes as a new network layer,” in *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, 2021, pp. 229–236.

[18] M. Rajib Hossen and M. A. Islam, “A lightweight workload-aware microservices autoscaling with qos assurance,” *arXiv e-prints*, pp. arXiv–2202, 2022.

[19] G. Yu, P. Chen, and Z. Zheng, “Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach,” *IEEE Transactions on Cloud Computing*, 2020.

[20] T. K.-H. Lin, “Client-centric orchestration and management of distributed applications in multi-tier clouds,” Ph.D. dissertation, University of Toronto (Canada), 2021.