A Proactive Cloud Application Auto-Scaler using Reinforcement Learning

Heimerson, Albin; Eker, Johan; Årzén, Karl-Erik

Published in:
UCC '22: Proceedings of the 15th IEEE/ACM International Conference on Utility and Cloud Computing

DOI:
10.1109/UCC56403.2022.00040

2022

A Proactive Cloud Application Auto-Scaler using Reinforcement Learning

Albin Heimerson, Johan Eker, Karl-Erik Årzén

Abstract

This work explores the use of reinforcement learning to design a proactive cloud resource auto-scaler that is able to predict usage across a distributed microservice application. The focus is on serving time-sensitive workloads, e.g., industrial automation, connected XR/VR (eXtended Reality/Virtual Reality), etc., where each job has a deadline and there is some cost associated with missing a deadline. A simple workload model, as well as a microservice application model, is presented. A reinforcement learning agent is trained to identify workloads and predict needed utilization for identified service chains. The results are compared to standard purely reactive techniques.

This work was supported by Vinnova grant ITEA3-17002 (AutoDC).

1 Introduction

Today, the use of cloud services has become an integral, and increasingly invisible part of our daily routines, e.g., helping us navigate the Internet, stream multimedia content, and buy groceries. The large public cloud providers have become major players in both shaping and enabling the way modern society function and providing services with impressive availability and reach. What we are currently witnessing is a second wave of cloud services that goes beyond web storefronts and IT systems and includes the digitalization of industrial systems. Automation and time-sensitive systems, that until recently have been confined to the factory floor and hidden from the external world, are now taking their first steps towards the cloud. Other emerging time-sensitive systems are XR/VR applications and different kinds of conversational video systems.

The digitalization of timing-sensitive automation systems holds the promise to improve efficiency and resource utilization through data-driven innovations.

However, providing end-to-end timing guarantees with state-of-the-art technologies is typically not possible. There are numerous challenges pertaining to deterministic virtualization technologies, virtualized real-time networking, etc. In addition, there are control plane issues related to how resources are allocated in a timely fashion. The hypothesis explored in this paper is that we can use reinforcement learning (RL) to implicitly model the service chain and thus be able to propagate information correctly to predict resource needs for connected cloud-based services. The focus of our work is time-sensitive services where each job is associated with a deadline, which is typically the case for, e.g., cloud-based control systems [1, 2]. To meet deadlines, there is typically a trade-off involved between over-allocating resources and thus having unused capacity on standby or allowing deadlines to be missed while scaling up.

The paper is organized as follows. Section 2 defines the problem, and Section 3 contains an overview of related work and what new ideas this article provides. Section 4 presents the application model as well as the workload model. In Section 5 the setup for the simulations is presented, as well as the setup for all the baseline agents and the RL agent. The results are presented and discussed in Section 6. Finally, Section 7 presents some concluding remarks as well as some ideas for future work.

2 Problem definition

Microservice architecture is the predominant style for modern cloud applications. The basic trait is that a single application is built from a set of small, independent services, typically built around given business objectives. Microservices are deployed and managed individually with little to no centralized coordination. An advantage compared to previous generations of cloud applications, that were designed as larger monoliths, is great

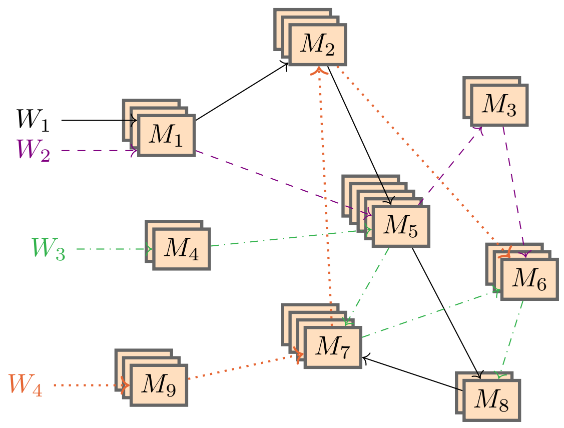

Figure 1: Four cloud applications, W_1 to W_4, that are composed of a set of microservices. While many of the applications share microservices, the call graphs are different, i.e., the paths the workloads go through differ.

simplification with regard to scaling and management. However, a downside is an increase in complexity on the application layer, since an application is composed of distributed communicating microservices, which are hard to debug, trace, etc. In addition, a microservice is often part of more than one cloud application. For example, an authentication service or an ad service may be used by several cloud applications. In many situations, there is a many-to-many relation between microservices on a single platform. This can give rise to quite complex call graphs, sometimes referred to as microservice "death star" architecture [3]. A service mesh is a layer that abstracts operations on a set of connected microservices. This allows compartmentalizing services and allows for easier handling and individual scaling of certain parts of a larger system (the mesh).

Figure 1 shows a network of connected microservices. There are four cloud applications. Each cloud application provides a workload that spans over one or more microservices. The actual call graph is typically dynamic or data-dependent.

Current standard platforms provide reactive scaling mechanisms that dynamically add or remove capacity to match demands. This can be done by measuring the usage of resources and inspecting queue lengths, etc. There will be a trade-off between the ability to serve a rapid increase in workload and keeping unused allocated resources on standby. The tighter the resource allocation, the more cost-efficient the system.

If we again consider the system in Figure 1 and how reactive scaling would work for it. The workload W_1 enters at M_1 and causes further calls to M_2, M_5, M_8, and M_7. If there is a rapid increase in demand, a reactive auto-scaler would simply detect an increase in utilization at each of the involved microservices, one at a time. This means that the scaling of M_7 does not occur until the traffic reaches it, although it could have easily been predicted it if the call graph was known ahead of time, leading to an unnecessary and unwanted delay in scaling.

Instead of explicitly modelling the call graph, and all the intricacies that come with that, we explore how well an RL agent can learn an implicit model given minimal data from the microservices. Some implicit representation of the underlying call graph will need to be learned in order to be able to act proactively, but this will be embedded in the agent's internal neural networks. This will effectively enable the policy to create feed-forward connections between the load of one microservice to the desired state of another based on how it estimates the workload behavior. In the above example, this would mean that when workload W_1 increases and the capacity of microservices M_2, M_5, M_8, and M_7 can increase proactively. For time-sensitive workloads where the individual jobs have real-time constraints, like the ones described above, we can then avoid having spare capacity on standby and still minimize missed deadlines.

We do not assume knowledge about the workloads and do not try to predict the incoming loads, nor do we have any information on the call graphs, i.e., the paths that the workloads take. Instead, we use RL to train an agent to proactively scale microservices when we can observe changes in the workload on the system.

3 Related Works

Many recent works on cloud optimization look at some trade-off between maximizing utilization while honoring performance requirements, for example, latency bounds specified in some service level agreement (SLA). The fundamental trade-off deals with the cost of resources vs the value of providing a service at the right level.

(author?) [1] did automatic scaling and admission control of time-sensitive service chains using control-based techniques and network calculus. The focus of that work was to determine the lowest upper bound on computing resources to support workload variations without missing deadlines.

Examples of previous work on automatic resource allocation using reinforcement learning are found in [4, 5, 6]. The general problem addressed by these teams is that of controlling and scaling a single microservice by adding or removing single instances. By contrast, the work in this paper aims at scaling capacity proactively on a service mesh and allowing for arbitrarily sized scaling increments. We target workloads with real-time constraints and deadlines, requiring the auto-scaler to be fast and responsive.

(author?) [5] proposed a proactive scaling algorithm using a linear regressor to forecast future requests and using this to estimate future utilization. We specifically avoid estimating future workload given the inherent uncertainties and instead rely on the proactive scaling enabled by learning the workload paths in the service mesh. **(author?)** [7] presented a scaling approach using reinforcement learning and focusing on fast responses. However, they only act on a single machine and do not aim at any proactive scaling.

More traditional workload scheduling is also a related area of interest. The work found in [8, 9, 10] aims to solve a different type of task or job scheduling in a cloud setting using reinforcement learning. They model tasks as directed graphs of jobs to be processed in a given order. The graph, or the workload path, is thus known. In this paper where we aim to learn the nature of the different workloads by training, which will allow for a more adaptive algorithm that can handle changes over time, as there might be in a real environment.

(author?) [11] are looking at network routing using reinforcement learning, which is a slightly different problem, but they do strive for fast interaction similar to the work in this paper.

One common theme in the related work is to heavily reduce the size of both action and state space to simplify learning. This is done using a variety of methods, such as limiting the action space to only add or remove a fixed number of instances, as well as using longer steps. The latter leads to solutions that work best on stationary workloads and typically fail to address jobs with deadlines. Furthermore, to the best of our knowledge, previous work requires knowledge about the structure of the mesh and information about the call graph.

Our contribution is creating a reinforcement learning agent acting on a short timescale and taking the whole service chain into account to be able to do proactive scaling in the chain. We look at a chain of services in some microservice application, rather than only scaling a single service at the time. We also consider the latency, for example from boot time, when adding new resources.

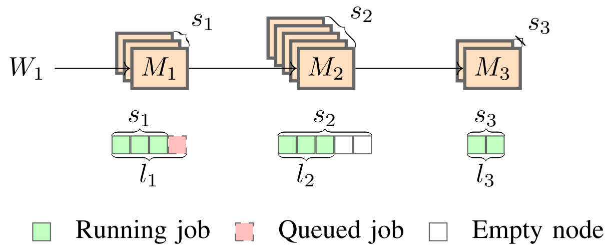

Figure 2: A single workload over 3 microservices M_i. Each has a scale s_i that can be controlled and a current load l_i denoting how many jobs are currently being processed or waiting to be processed. The workload W_1 has a path $P_1 = [1, 2, 3]$ associated with it, an execution time for each microservice on the path $t_{j,i}$ and a deadline T_{d_j} for each job's total allowed execution time.

4 Cloud application model

A cloud application commonly consists of a set of communicating microservices, each made to do one job. Incoming jobs will enter at some microservice, and after being processed there, they are either passed on to another microservice for further processing or they are done and can return. Different types of jobs can enter at different points and take different paths through the microservices.

We model this as a graph of microservices M_i where different workloads W_j generate jobs traversing the graph in different ways, as in Fig. 1.

We only simulate simple paths over a single microservice at a time, though the RL agent is not restricted to this and should also be able to learn good strategies for microservice graphs with, e.g., parallel paths.

To define the environment and how it works, we look at the smaller environment in Fig. 2, a single workload with three microservices.

Each microservice M_i has an arrival queue where incoming jobs are placed, this has a max length of q_{max} after which jobs are dropped. The number of running instances on M_i is s_i, and this can be changed inside the interval $[s_{min}, s_{max}]$ by booting or closing instances. Booting a new instance takes T_{boot} time, while closing takes T_{close} time. The total number of jobs on M_i is l_i, representing both jobs being worked on and jobs in the queue. The utilization of M_i is then the load over available resources, $u_i = l_i/s_i$.

Each workload W_j has a path P_j containing the order in which the microservices should be visited by corresponding jobs and a deadline D_j which is the total time allowed for a job to traverse the path. A job from workload W_j has a processing duration $t_{j,i}$ that depends on what type of workload it is, and which microservice it is currently on.

The environment is discretized to evolve in steps of Δt seconds, and in each step, the instances work off Δt time from their assigned jobs.

There is a value V_j to finishing a job of type W_j before the deadline, and a cost C_i to running instances of type M_i.

To earn as much value as possible, we want to maximize the finished jobs while minimizing the running instances.

5 Implementation details

Here we present the parameters and implementation details for the simulation model, as well as for the different algorithms we ran on the simulation model. The code used for this paper, as well as some instructions to help run it, is publicly available on Github[1].

5.1 Simulation model

Not all parametrizations of this type of environment will lend themselves to the problem we set out to solve. We specifically want to look at environments that, like Fig. 1, has chains where it could be possible to scale quicker if scaling is done with respect to the full set of microservices.

Taking a simple scenario with a single workload as in Fig. 2, the entry point M_1 will not be able to scale proactively without making guesses about random changes in arrivals, which is not something we look at in this work. But for $M_{i>1}$, it is possible to start scaling before the workload reach the microservice, and how much is gained depends on the difference between the duration of the work on previous microservices and the time to boot a new instance. To fully boot instance $M_{i>1}$ proactively would require that

$$\sum_{k=1}^{i-1} t_{1,k} \geq T_{boot} \text{ for } i \in [2,3]. \tag{1}$$

As long as there is no buildup of jobs and the microservices can immediately process the jobs, the roundtrip times for the jobs will then be $t_{1,1} + t_{1,2} + t_{1,3}$ steps, while if the job ever has to wait for either M_2 or M_3 to scale up, it will take at least $t_{1,1} + t_{1,2} + t_{1,3} + T_{boot}$ steps to finish. If the workload have a deadline D_1 that jobs should meet, we then want that

$$t_{1,1} + t_{1,2} + t_{1,3} + T_{boot} > D_1 \tag{2}$$

to make sure that proactive scaling is required in order to avoid buffers on M_2 and M_3. To also make sure it is actually possible to hit deadlines we want that the total work time is less than the deadline.

$$\sum_i t_{1,i} < D_1 \tag{3}$$

Selecting $t_{j,i} = T_{boot} = 1$ will fulfill (1) for M_2 and M_3 allowing them to scale proactively. Then we have $3 < D_1 < 4$ from (2,3), so choosing e.g. $D_1 = 3.5$ will make any reactive approach have to choose between either keeping a large enough buffer on all microservices or miss deadlines as the workload fluctuates. A proactive approach on the other hand could scale M_2 and M_3 based on the current state of microservices before them in order to hit deadlines while not keeping a buffer anywhere but on the first microservice.

Given that T_{close} does not affect any important dynamics of the environment, it is simply set to zero.

The costs of instances and values of jobs were selected so that the value of a job should be able to cover booting and running costs for instances working on it, with an additional margin to make it worth hitting deadlines in almost all cases. Without this, the RL agent would likely have found the most value in just scaling down as much as possible.

We create a synthetic workload that is simply designed to generate a load that does a random walk over integers in a constrained range. This could for example be a set of control applications turning on and off randomly, adding a constant load while they are on. So assuming we currently have an arrival rate of x new jobs every step, then the workload will with a probability $p = 0.1$ pick a new random arrival rate in the range $[\max(x_{min}, x-1), \min(x_{max}, x+1)]$, and otherwise the workload is kept as is. Here we set $[x_{min}, x_{max}] = [0, 3]$. The cost is set to $C_i = 1$ and the value for finished jobs is $V_j = 16$.

In addition to the model with a single workload chain presented above, the model in Fig. 3 with two workloads taking different paths over some shared and some individual microservices is also used. The parameters for this

[1] `https://github.com/albheim/ServiceMeshControl/tree/dml-icc_2022`

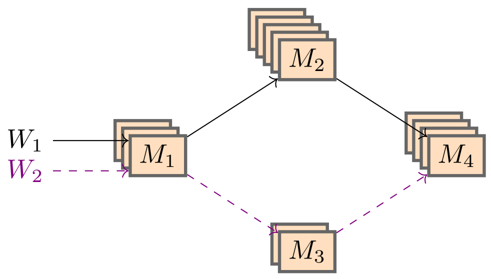

Figure 3: An environment with two workload chains traversing four microservices, some shared and some individual.

model were chosen with the same reasoning as above, and are the same in all cases except for the workload parameters. Now we have W_1 with $[x_{min}, x_{max}] = [0, 4]$, and W_2 with $[x_{min}, x_{max}] = [0, 2]$, both still using $p = 0.1$.

These environments are difficult from a learning perspective in a few different ways. First, it is not fully observable, meaning though we can see the utilization for a microservice, that does not tell us how much work there is left on the current jobs. This makes it harder to understand the value of states, since the same state could have different values based on a hidden internal state. There are also delays, e.g., when booting new instances it takes time between the decision to scale before the state actually changes and there is a delay between a change in the state that will lead to a job passing through and the job actually passing through and generating a value. There is a tendency towards scaling down since it gives an immediate response, while how many jobs get through is a more delayed signal. This results in a local optimum of scaling down everything as much as possible, where any single action only gives extra cost, and to get any reward multiple services have to scale up and do it for long enough for any actual job to get through and generate value. This tendency seemed much stronger when the environment allowed scaling down to zero instances, and instead setting the minimum scale to one made it much less prevalent. Since part of the problem is also that all jobs accumulating in queues will likely miss their deadline, this will make it harder to find some reward again when scaling up. For this reason, the maximum for the queues is set rather low, to allow for a small buffer while not letting too much accumulate.

5.2 Reinforcement learning agent

We use a version of Soft Actor-Critic (SAC) [12, 13], an off-policy model-free deep RL algorithm. It was selected mainly for the sample efficiency and robustness to its hyperparameters, though other algorithms could very well have sufficed. The SAC algorithm we use is an implementation from `ReinforcementLearning.jl` [14], corresponding to the updated SAC algorithm in [13]. This means there is an automatic tuning of the entropy parameter α to match a given target entropy $\mathcal{H}$, simplifying parameter tuning.

The neural networks for both policy and value function are a few densely connected layers with a common activation function. The policy network outputs 2 continuous values for each action dimension, creating a normal distribution from which to sample the actions. Since the action is to choose the desired scale of the microservices, the action space is discrete

$$\mathbf{a}_t \in [s_{min}, s_{min} + 1, \ldots, s_{max}]^N$$

and the action produced by the policy has to be converted by rounding to the nearest valid integer.

The state used is the current scale s_i and the utilization u_i for each microservice. This is then stacked with the previous state to give the agent more information, both since the state is not fully observable, and also to allow for information about changes in state to be incorporated in the decision. This can help in learning when there are delays in the environment, so that the agent can easier connect what action led to what result. The full state the agent sees is then

$$\begin{aligned}\mathbf{s}_t = [&s_1(t), u_1(t), \ldots, s_N(t), u_N(t),\\ &s_1(t-1), u_1(t-1), \ldots, s_N(t-1), u_N(t-1)]\end{aligned}$$

where $N = |\boldsymbol{M}|$ is the number of microservices in the application and t is the current step. The state is also normalized, letting the neural network operate on values of similar scales, which helps with learning.

The reward is defined based on the value of completed jobs, i.e. no value for missed deadlines, and the cost of running instances. The number of jobs that finished within the deadline during this step is denoted $f_j(t)$, giving

$$r_t = \sum_i^N C_i s_i(t) - \sum_j^{|W|} V_j f_j(t). \tag{4}$$

The reward is scaled by a factor r_{scale}, where finding a good value is a common strategy to improve the learning of the value network.

Table 1: Hyperparameters for Soft Actor-Critic

Parameter	Value
optimizer	`ADAM`
learning rate for optimizer	0.003
activation function	`elu`
reward scaling r_{scale}	0.5
discount factor γ	0.93
target smoothing τ	0.01
target entropy $\mathcal{H}$	$2.5 \cdot N$
value net hidden layers	4
value net hidden units	120
policy net hidden layers	2
policy net hidden units	20
minibatch size B	100
alpha learning rate	$5 \cdot 10^{-5}$
target entropy	7.5
update frequency	8
replay buffer size	$5 \cdot 10^5$
start algorithm	random action
steps with start algorithm	50000

Though SAC is supposed to be robust to hyperparameters, some tuning still played an important role in making a successful agent. Table 1 contains the parameters used for the experiments and are similar for both environments, only the target entropy depends on N to reflect that the differently sized action spaces should have different entropy.

Since the neural networks used are quite small, the RL agents could be trained and evaluated on single CPUs.

5.3 Baseline algorithms

5.3.1 Reactive autoscaler

One very simple strategy is to look at the average utilization u_i over all machines for microservice i and if this is larger than some u_{max} a machine is added, if it is lower than some u_{min} a machine is removed. This only looks at utilization and can only scale by one machine per step, but it manages well with reasonably chosen limits and is very simple to implement and understand. For these experiments $(u_{min}, u_{max}) = (0.5, 0.8)$ is used.

5.3.2 Kubernetes autoscaler

The Kubernetes horizontal pod autoscaler (HPA) is similar to the basic reactive scaler in that it is only reactive and scales based on the desired utilization value.

The desired scaling for microservice i is calculated as

$$D_i(t) = \begin{cases} \left\lceil s_i(t)\frac{u_i(t)}{u_{target}} \right\rceil & \text{if } \left|1 - \frac{u_i(t)}{u_{target}}\right| > \epsilon \\ D_i(t-1) & \text{else} \end{cases}$$

where $\epsilon = 0.1$ and $u_{target} = 0.8$ is used. This means that the scaler only acts if the relative change is large enough, and will help to avoid jitter in the signal. The next step is that the actual action it takes is the maximum of all the previous desired actions over a window of time, so

$$a_i(t) = \max_{\tau \in [0,W]} D_i(t - \tau)$$

where $W = 60$s was used as the size for the window. This means scaling down will happen gradually, reducing the impact of fluctuations in the metrics and making sure to rather keep a few more instances running than have too few for the workload.

The Kubernetes scaler is implemented according to what default parameters we could find for HPA at the time of the experiments.

5.3.3 Oracle autoscaler

The name oracle is to indicate that it has knowledge not available to the other algorithms. The extra knowledge is still constrained to what was deemed interesting for the proactive scaling mechanism in order to create something that is as close to optimal as possible for the proactive part.

So the first microservice in a job chain will have to be reactive, and the strategy is to always keep one extra instance running per job type.

For a microservice later in the job chain, scaling of the service can start at some point after the job arrives with the goal that the microservice will finish scaling at the same point the job arrives at it.

This assumes full knowledge of the utilization on each microservice and what type each job is. It also assumes full knowledge of all microservice chains for jobs as well as delays between difference microservices in the chain.

It will only work well under specific circumstances, but since these are fulfilled in the environments presented here, it is used as a benchmark of how well it is possible to scale proactively.

This is most likely not optimal in regard to the loss function defined here, but will be as close as we can get with a relatively simple algorithm.

6 Results

The results from the two environments in Fig. 2 and 3 are presented here. Some additional simulations were made to verify that the RL agent could still perform well outside these examples that were specifically constructed for this, though we choose to focus on presenting these cases since they are simple to reason about while providing the complexity needed to examine how well the RL agent can learn the desired traits. Other cases that were tested had for example more complex workload paths with varying execution times, where proactive scaling was not always possible, or simply a larger value for the jobs compared to the cost of an instance.

The simulations are run over a longer time than shown here to verify that the RL agent has stabilized and does not hit a phase where catastrophic forgetting happens, a well-known phenomenon for neural networks learning in a sequential manner. For clarity, we only show 100 days of data in the plots. This was deemed enough to both see a little of the initial learning phase and the later phase where the policy is optimized.

All data presented is sampled and smoothed to make it possible to visualize what is going on. The simulation environment logs data every minute, which is then processed through a moving average over 1000 data points and downsampled by a factor of 1000 for nicer plotting.

The random seeds used for the presented experiments are different from the ones used in the hyperparameter optimization. This ensures that the agent was not overfitting to the specific seed.

Starting with the simple environment in Fig. 2, the reward in Fig. 4 show how the RL agent quickly learns to achieve a competitive score, and manages to keep a relatively stable performance throughout the simulation. The few spikes are due to continuously learning and exploring, something that could be turned off. But continuously learning could provide additional benefits for the algorithms, especially when applied to more complex environments, since being able to adapt to changing conditions in the environment can allow for more specialized control.

The scale and utilization for the microservices are displayed in Fig. 5 where the average arrival rate for the jobs are 1.5 jobs/s, so a proactive strategy should be able to keep M_2 and M_3 at those levels, while M_1 will likely need a small buffer. Neither strategy keeps that low on any of the microservices, but the oracle scaler is close on M_1 and M_2, where proactive scaling can be done, with

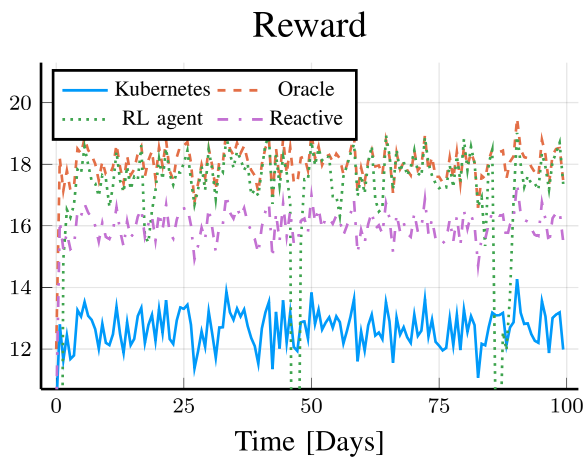

Figure 4: Reward for the different scaling strategies on the single workload chain environment from Fig. 2. Dips in RL agent performance is due to continually learning and exploring.

the RL agent not far behind.

Based on these results, the RL agent seems to do well in scaling. It does learn a strategy that keeps similar utilization for the microservices where proactive scaling is possible, achieving a similar reward, making a case that the RL agent has learning some kind of forward connections between the microservices to do proactive scaling. In addition to being very close to the oracle for the two later microservices, it also pushes the utilization for the first microservice higher than the other strategies. This can be good but will likely have an effect on deadline misses, the other term in the reward (4).

The environment with two different workloads over different paths in the application is visualized in Fig. 6. Here we see how the oracle scaler keeps the highest utilization on the microservices where it can scale proactively, with the RL agent not far behind. The oracle scaler has information about the job distribution on each microservice that is not available to the other scalers, and should be expected to be able to keep a higher utilization because of this. The Kubernetes scaler prioritizes not scaling down over minimizing the buffer, and will as such have quite low utilization on average.

The reward for the strategies is shown Fig. 7, and we see that the RL agent has a similar reward to the oracle scaler, though a bit noisier. This is expected though, since it does not know the full state or exact microservice layout

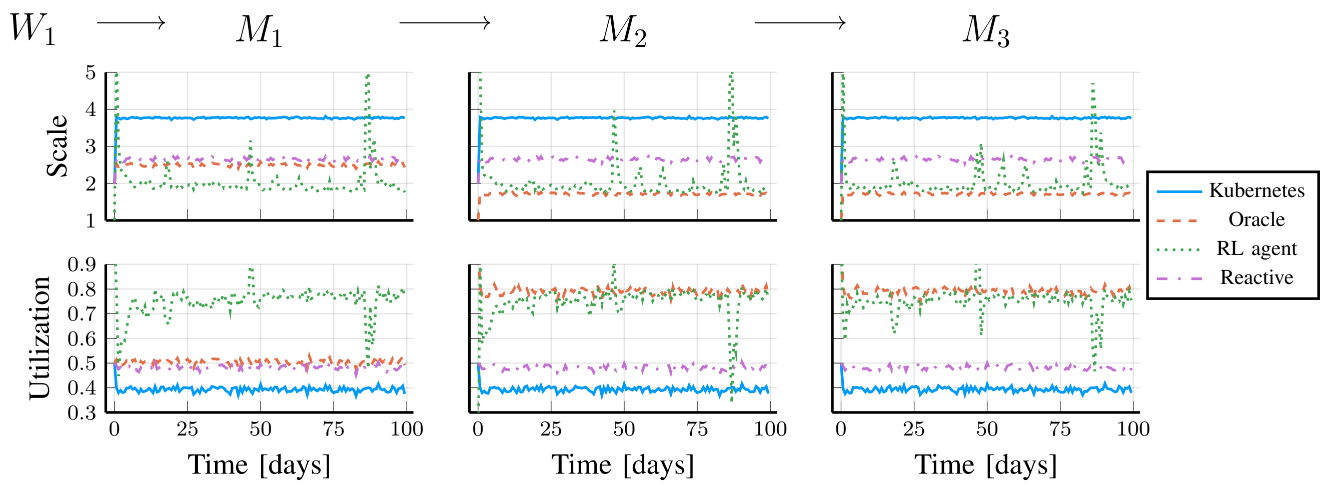

Figure 5: Scale and utilization of the three microservices in the single chain environment from Fig. 2. The oracle scaler manages to keep the highest utilization for the microservices that can be proactively scaled, but does get information not available to the other scalers. The RL agent manages to keep all three at relatively high utilization in comparison, with some noise in the scale due to continually learning and exploring.

as the oracle do, and it is also continuously learning. The Kubernetes scaler receives quite a low reward, which can be attributed to it being a very safe strategy in that it promotes keeping a large buffer over risking scaling down too low and missing a job. The environment is certainly not optimal for the Kubernetes scaler, but it was selected to be able to show the benefit of proactive scaling in the cases where that is possible.

In the bottom left of Fig. 6 we also see that neither the oracle nor the Kubernetes scaler miss a single job. This is expected since the oracle was designed to keep the lowest scale, while still guaranteeing no misses for this type of workload. The Kubernetes scaler looks at the maximally needed scaler over a period, and as such will tend to have a larger buffer in favor of missing jobs. The RL agent is still learning, as we see how the missed jobs have a trend downwards for most of the period. The reason it still misses a few jobs can be attributed to mainly two reasons. First, the objective function does not explicitly tell the agent to not miss any jobs. The reward (4) is a weighted sum of the cost of running instances and the value of finishing jobs in time, which means that there will be an optimal strategy where neither term can be improved without making the other one worse. Depending on parameters in the environment, this might very well be at a level where some jobs get dropped. The other reason is that the RL agent is stochastic, since it needs to explore to learn. Some stochasticity will likely result in unfortunate scaling decisions now and then, even if the number of occurrences is relatively low. This will result in booting or closing instances in ways that might not be optimal, and the RL agent will on average boot and close more instances than the other strategies, but this is an effect of needing exploration to accommodate the learning process.

Increasing the reward of a finished job compared to the cost of running an instance, will shift the optimal policy towards dropping less, and thus scaling up more. The agent has been tested on both smaller and larger job values and managed to learn well in both cases.

7 Conclusion

In the work presented, we aim at creating a proactive cloud scaler using reinforcement learning. The main objective is to achieve feed-forward scaling for service chains, such that it is possible to service time-sensitive cloud applications. A cloud model and a simulation environment are presented. With a focus on capturing traits such as delays and latencies, it comprises what is typically hard for algorithms based on sequential learning. A few

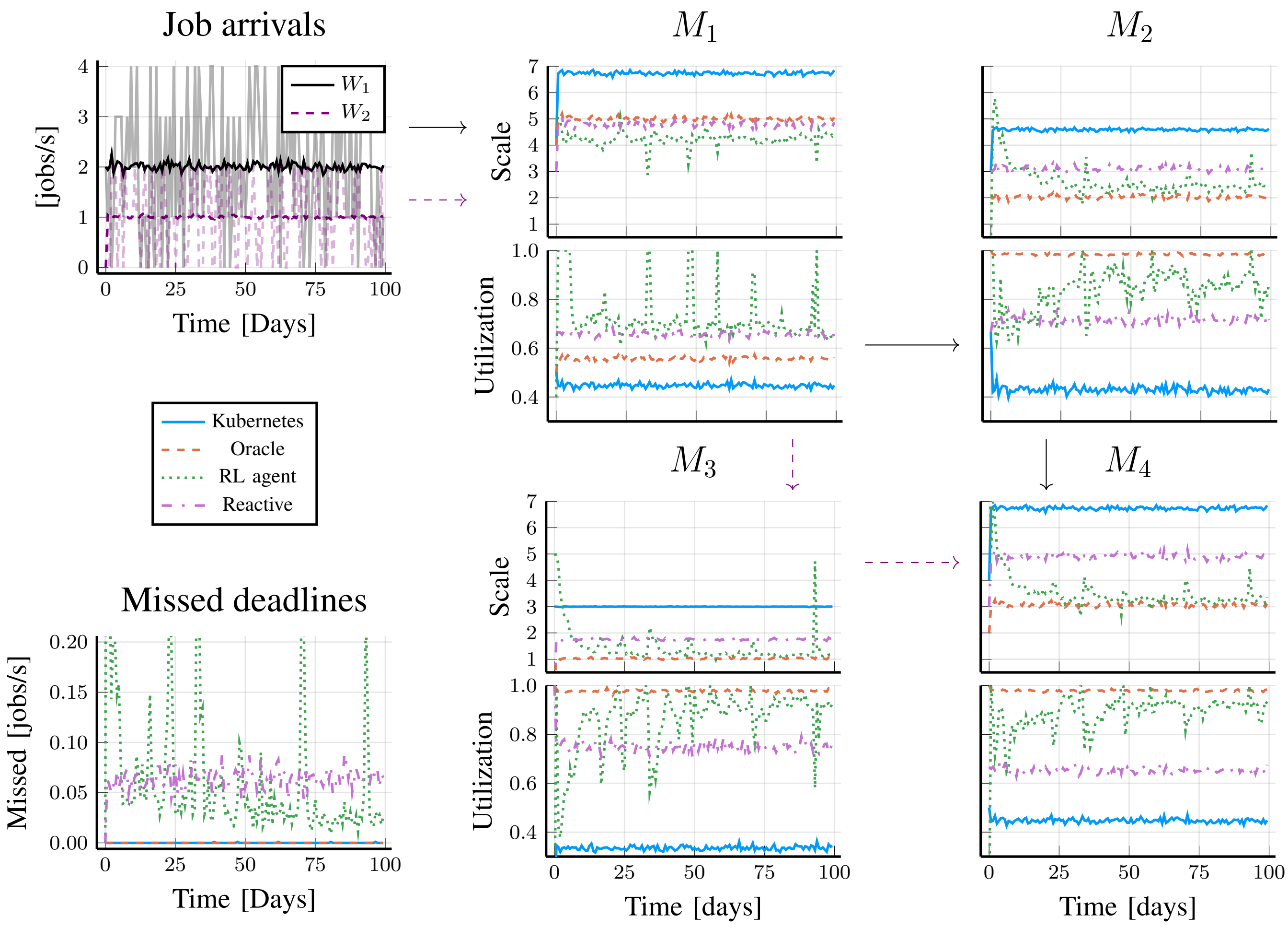

Figure 6: The double chain has two workloads (W_1 and W_2) with different average loads (top left) going through both shared microservices (M_1 and M_4) as well as individual ones (M_2 and M_3). In the bottom left, we see how many missed deadlines there are on average for the different scaling algorithms. The RL agents signal is a bit noisy due to continually learning and exploring.

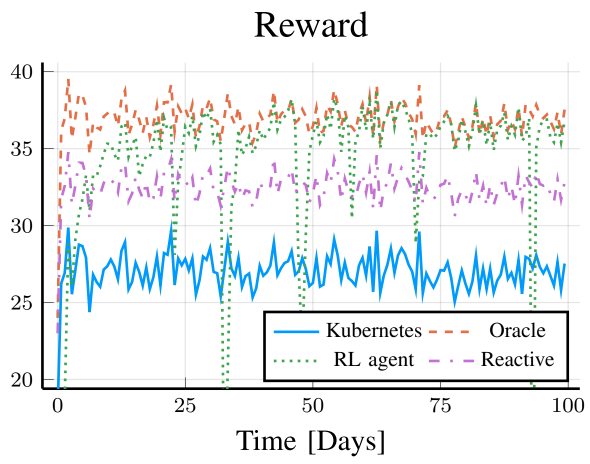

Figure 7: Reward for the different agent acting on the double workload chain environemnt from Fig. 3. Dips in RL agent performance is due to continually learning and exploring.

alternative scaling algorithms are presented to compare against, and the process for achieving successful training of the RL agent is also mentioned.

We show that the RL agent achieves a higher reward than reactive scalers, and is close to the *oracle*, when it comes to optimizing the defined objective (4). This shows that the RL agent can find good strategies, even for this type of system that are problematic for a learning-based algorithm.

We show how the RL agent is keeping utilization higher on microservices later in a service chain, indicating that the RL agent embedded some knowledge of the graph structure and learned the forward connection needed to do proactive scaling on the later microservices in the chain.

For future work, we want to implement this as a replacement for Kubernetes autoscaler with some standard mesh tools such as Istio using the Ericsson Research Datacenter. When running on real systems, it is also important that much of the training can be done beforehand on a model. So exploring offline learning based on data collected by standard algorithms to later just fine-tune the agent when running live could be an important question to look into.

References

[1] Victor Millnert, Enrico Bini, and Johan Eker. AutoSAC: Automatic scaling and admission control of forwarding graphs. *Annales des Telecommunications*, August 2017.

[2] Per Skarin. *Control over the Cloud: Offloading, Elastic Computing, and Predictive Control.* PhD thesis, Department of Automatic Control, Lund University, 2021.

[3] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18. Association for Computing Machinery, 2019.

[4] Enda Barrett, Enda Howley, and Jim Duggan. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience*, 25(12):1656–1674, 2013.

[5] Mostafa Ghobaei-Arani, Sam Jabbehdari, and Mohammad Ali Pourmina. An autonomic resource provisioning approach for service-based cloud applications: A hybrid approach. *Future Generation Computer Systems*, 78:191–210, 2018.

[6] J. V. Bibal Benifa and D. Dejey. RLPAS: Reinforcement Learning-based Proactive Auto-Scaler for Resource Provisioning in Cloud Environment. *Mobile Networks and Applications*, 24(4):1348–1363, 2019.

[7] Constantinos Bitsakos, Ioannis Konstantinou, and Nectarios Koziris. DERP: A Deep Reinforcement Learning Cloud System for Elastic Resource Provisioning. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 21–29, Nicosia, December 2018. IEEE.

[8] M. Cheng, J. Li, and S. Nazarian. DRL-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 129–134, January 2018.

[9] Tingting Dong, Fei Xue, Chuangbai Xiao, and Juntao Li. Task scheduling based on deep reinforcement learning in a cloud manufacturing environment. *Concurrency and Computation: Practice and Experience*, 32(11):e5654, 2020.

[10] Y. Wang, H. Liu, W. Zheng, Y. Xia, Y. Li, P. Chen, K. Guo, and H. Xie. Multi-Objective Workflow Scheduling With Deep-Q-Network-Based Multi-Agent Reinforcement Learning. *IEEE Access*, 7:39974–39982, 2019.

[11] Qinliang Lin, Zhibo Gong, Qiaoling Wang, and Jinlong Li. RILNET: A Reinforcement Learning Based Load Balancing Approach for Datacenter Networks. In Éric Renault, Paul Mühlethaler, and Selma Boumerdassi, editors, *Machine Learning for Networking*, Lecture Notes in Computer Science, pages 44–55, Cham, 2019. Springer International Publishing.

[12] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv:1801.01290 [cs, stat]*, August 2018.

[13] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic Algorithms and Applications. *arXiv:1812.05905 [cs, stat]*, January 2019.

[14] Jun Tian and other contributors. Reinforcementlearning.jl: A reinforcement learning package for the julia language, 2020.