# Automatic management of cloud applications with use of Proximal Policy Optimization

Włodzimierz Funika$^{1[0000-0003-3321-7348]}$, Paweł Koperek$^{1[0000-0003-3613-2390]}$, and Jacek Kitowski$^{1,2[0000-0003-3902-8310]}$

[1] AGH, Faculty of Computer Science, Electronics and Telecommunication, Dept. of Computer Science, al. Mickiewicza 30, 30-059, Kraków, Poland
[2] AGH, ACC CYFRONET AGH, ul. Nawojki 11, 30-950, Kraków, Poland

email:funika@agh.edu.pl, pkoperek@gmail.com, kito@agh.edu.pl

**Abstract.** Reinforcement learning is a very active field of research with many practical applications. Success in many cases is driven by combining it with Deep Learning. In this paper we present the results of our attempt to use modern advancements in this area for automated management of resources used to host distributed software. We describe the use of an autonomous agent that employs a policy trained with use of Proximal Policy Optimization algorithm. The agent is managing a cloud infrastructure used to process a sample workload. We present the design and architecture of a complete autonomous management system and explain how the management policy was trained. Finally, we compare the performance to the traditional automatic management approach exploited in AWS stack and discuss feasibility to use the presented approach in other scenarios.

**Keywords:** reinforcement learning, cloud resources, automatic management, proximal policy optimization

## 1 Introduction

Leveraging the cloud computing infrastructures is one of the currently dominating trends in the design of modern software systems. The main advantages of this approach include high availability, increased security and flexibility in resource allocation. In many cases the ability to adjust the amount of used resources to the actual needs is the driver of the adoption. The promise to reduce the costs of resources is very compelling. Unfortunately it requires implementing special measures. The application needs in support for adding or removing more cores, RAM, hard drives (*vertical scaling*) or more virtual or physical machines (*horizontal scaling*) without breaking the core functionality. Furthermore one needs to develop a policy under which the resources will be added or removed. In certain scenarios, where the environment renders strong and stable seasonal behavior patterns, the configurations and resources can be approximated by human experience in advance. Nevertheless, in many other cases elasticity can only be enabled by automatic scaling, which we can define as *a dynamic process [...] that adapts software configurations [...] and hardware resources provisioning [...] on-demand, according to the time-varying environmental conditions* [1].

Reinforcement learning techniques have been known for a long time [2, 3]. Until recently they were mostly applicable to relatively simple problems, where observing the whole environment was easy and the number of possible actions was small. The new advancements in the area allow to tackle domains which are much more complicated, like computer games [4], control of robots [5] or the game of Go [6]. The state-of-the-art results can be obtained thanks to application of Deep Learning, e.g. in form of Deep Q Learning [7], Asynchronous Actor-Critic Agents (A3C) [8] or more recently Proximal Policy Optimization [9]. Those methods enable learning complex behaviors by directly observing an environment and interacting with it through pre-defined actions. In some cases this approach allowed to achieve results surpassing human decisions based performance.

Such successes encourage experimenting with applying Deep Reinforcement Learning (*DRL*) to other domains. One particular area, where this technique might deliver a lot of benefits, is the auto-scaling of applications deployed to compute clouds. The infrastructure used to host the application becomes the environment in which the automatic agent operates. The state of the application becomes the state which the agent alters and the operations which can be executed through a cloud vendor API become the agent's actions. Available measurements and metrics are well defined: usually the technology choices help to define which elements of the system should be observed and how. There is a variety of available monitoring software which can be used. In this type of task there are usually well defined goals (e.g. reducing request latency, average CPU load, memory consumption or monetary cost of hosting). Such an optimization goal can be translated into a reward function, which can be used by the agent as a feedback mechanism. The agent can discover management policies without any prior knowledge, by conducting experiments in the underlying infrastructure through a process of trials and errors.

The main disadvantage of the described approach is the cost of creating of a DRL policy. The policy needs to go through multiple iterations where it interacts with the automatically scaled system and observes its reactions. Unfortunately, since the decisions made at the beginning might be quite random, there is a substantial risk of destabilizing the observed application and making it unusable by the end users. Such a situation is unacceptable in a production system, what suggests that creating a special, duplicate training environment might be helpful. Unfortunately, this introduces additional resource requirements and therefore increases the overall cost. This situation might be mitigated by modelling the auto-scaled application and simulating it in an artificial environment. This enables replaying the observed workloads multiple times and much faster than they happened in reality. Since DRL usually can benefit from increasing the number of training iterations, the simulation-based approach can help to create more efficient policies.

In this paper we present a novel auto-scaling software system which leverages the simulation of a compute cloud. We present the Semantic-Based Automatic Monitoring and Management (SAMM) system [10] extended by a decision making component, which employs a DRL method for training - the Proximal Policy Optimization. SAMM observes a cloud-based application, passes its observations (metric measurements and information about the scheduled jobs) to the policy. In case the policy decides to execute an action, that information is used by SAMM to adjust the environment by e.g. creating or removing resources through cloud API function invocations. Before being deployed to the live system, the policy is trained in a simulator, what enables us to avoid potentially very costly bad decisions made during the training phase. To the best of our knowledge, this is the first attempt to use the PPO algorithm to control resources of a distributed application deployed in a real-world cloud computing environment. We present a complete and functioning implementation and share the results of experiments.

The paper is organized as follows: in Section 2 we present related work, Section 3 describes the design and architecture of the environment and the training of the decision policy. Section 4 provides details details of the experiments and the environment in which they were being run. Section 5 summarizes our research and outlines next steps.

## 2 Related Work

Managing an infrastructure in a way, which minimizes the monetary cost while maintaining the business requirements, sometimes defined as Quality-of-Service (*QoS*) objectives, is a very complex subject. There have been numerous attempts to tackle this problem, which differ in many ways. One possible taxonomy has been proposed by the authors of [1], where the following features were chosen to classify auto-scaling systems:

- *self-awareness* - the ability to acquire and maintain knowledge about the system's state. It can be implemented in the form of: *stimulus awareness*, *time awareness*, *goal awareness*, *interaction awareness*, *meta awareness.*
- *self-adaptivity* - the ability to change own behavior depending on the requirements. We include the variants of *self-healing*, *self-configuring*, *self-optimizing*, *self-protecting*.
- *architectural patterns* - the structure of the auto-scaling process which includes the definition of interactions between components and specification of modules. The dominant approaches are: *Feedback loop* [11], *Observe-Decide-Act* (ODA) [12], *Monitor-Analysis-Plan-Execute* (MAPE) [13]
- *QoS modeling* - the control primitives which are used to change the environment and the model which connects QoS metrics to the control primitives. There are different types of models: static (the relationship between metrics and how resources are reallocated is defined before the system begins to work) [14], dynamic (resource allocation is determined based on statistical analysis of workload traces) [15], semi-dynamic [16], based on machine learning [17] or simulation [18].

- *granularity of control* - specification of what is the basic entity to control: virtual machine [19], container [20], application [21].
- *decision making* - definition of the process which produces a control decision, including:
  - the mechanism of reasoning and searching for the decisions
  - objectives and their representations
  - which control primitives need to be changed.

  There are numerous different approaches, most notably:
  - rule-based control [22, 23] - classic approach in which the scaling actions are executed when some pre-defined conditions occur, e.g. when the average CPU load reaches 0.9 a new VM is added to the pool of resources;
  - control theory based - the process of making a decision to change the pool of resources uses mechanisms described by the control theory, e.g. [24];
  - search based optimization - the possible decisions are treated as a part of a large but finite search space. The process of choosing among them becomes a search problem. There were many different attempts which have used this approach: [25, 26, 17]. The first attempts to exploit Deep Reinforcement Learning [27] also fit in this category.

The system presented in this paper can be classified as a *self-optimizing* and *interaction-aware*, with a *feedback-loop* based architecture. It operates on the container granularity level. The decision making process is based on the Deep Reinforcement Learning (PPO algorithm) approach. To our best knowledge our contribution is a first example of the complete system which implements such an approach in a widely available public cloud environment.

### 2.1 Reinforcement Learning

Reinforcement Learning (RL) [2, 28] is a machine learning paradigm which focuses on discovering a policy for autonomous agents, which take actions within a specific environment. The goal of the policy is to maximize a specific reward whose value can be presented to the agent with a delay. The RL approach differs from *supervised learning*: training the agent is based on the fact that knowledge is gathered in a trial and error process, where the agent interacts with the environment and observes results of its actions. There is no *supervising entity*, which would be capable of providing feedback on whether certain actions are better than others. RL is also different from unsupervised learning: it focuses on maximizing the reward signal instead of finding the structure hidden in collections of unlabeled data.

There are multiple variants of Reinforcement Learning algorithms:

- *Online* and *offline*: the former approach assumes the agent's policy is updated based on the most recent data, after every step (e.g. every monitoring and management iteration), in the latter one - after a full *episode* (i.e. after interactions cease, the environment needs to restart and a complete reward is given).

– *Model-based* and *model-free*: the former approach assumes an explicit model of the environment (state transitions and reward estimations) are created, in the latter it is assumed a decision can be based only on a sample of information about transitions.

The *model-free* approach has become very popular recently thanks to combining it with Deep Learning and creating so called *Deep Reinforcement Learning* (DRL). Using this technique allowed to create autonomous agents which are capable of achieving human-level performance across many domains.

Policy gradient methods are an approach to DRL, which is believed to render good experimental results. The training process improves a vector of the policy parameters $\Theta$ based on the gradient of some estimated scalar performance objective $J(\Theta)$ in respect to the policy parameters. These methods seek to maximize performance (measured as a reward obtained from interactions with the environment) and as such they change the parameters according to an iterative process in which the changes to parameters approximate a *gradient ascent* in $J$:

$$\Theta_{k+1} = \Theta_k + \alpha \nabla_{\Theta} J(\Theta_k) \tag{1}$$

where $\Theta_k$ denotes policy's parameters in the $k$-th iteration of the training process.

There are many variants of policy gradient optimization methods, however in this paper we focus on the *Proximal Policy Optimization* (PPO) [9].

The algorithm aims to compute a parameter update at each step, that on the one hand minimizes the cost function, while at the same time ensures the difference to the previous policy to be relatively small. This is achieved by modifying the objective in such a way that it ensures the updates to parameters are not too big. The objective is therefore defined by the following function:

$$J(\Theta) = L^{CLIP}(\Theta) = \mathbb{E}_t \left[ min(r_t(\Theta)A_t, clip(r_t(\Theta), 1-\epsilon, 1+\epsilon)A_t) \right] \tag{2}$$

where $\mathbb{E}_t$ denotes calculating average over a batch of samples at timestamp $t$, $A_t$ is an estimator of the advantage function which helps to evaluate which action is the most beneficial in a given state. $r_t$ marks probability ratio $r_t(\Theta) = \frac{\pi_{\Theta}(a_t|s_t)}{\pi_{\Theta_{old}}(a_t|s_t)}$ in which $\pi_{\Theta}(a_t|s_t)$ denotes the probability of taking an action $a$ in state $s$ by a stochastic policy and $\Theta_{old}$ are the policy parameters before the update. The $clip(r_t(\Theta), 1-\epsilon, 1+\epsilon)$ function keeps the value of $r_t(\Theta)$ within some specified limits (*clips* it at the end of the range) and $\epsilon$ is a hyperparameter with a typical value between 0.1 and 0.3.

In our previous research [29] we experimentally verified that PPO gave the best empirical results in automated resources management among the policy gradient methods. Therefore have chosen this algorithm for experiments presented in this paper.

## 3 Proposed System Design

In this section we present the design of an experimental system, which allows to employ DRL to create automatic management agents. These agents are meant to manage resources of a distributed application deployed in a cloud infrastructure. The system attempts to adapt itself to ever-changing environment conditions by continuously improving the DL model used as an agent's decision policy. To avoid the costs of poor decisions while training a policy, a simulation is used as a training environment.

We begin with presenting the components of the system's architecture and their responsibilities. In the following section we discuss details of how we trained the policy used to make resource management decisions.

### 3.1 Architecture

In our research we focused on utilizing some well known and tested frameworks to reduce the amount of time needed to implement our ideas. As a foundation we have chosen SAMM [10], a prototype monitoring system enabling experimenting with automatic management of computer resources. It allows to easily extend its monitoring capabilities by new types of resources to observe and integrate with different technologies and new algorithms and observe their impact on the observed system.

In our use case, SAMM is used to integrate other elements of the system together to form a feedback loop:

- Periodically polls measurements which describe the current state of the system (e.g. the average CPU utilization in the computation cluster, amount of used memory etc),
- Aggregates measurements into metrics used by the decision policy,
- Communicates with the *Policy Evaluation Service*. Provides the current state of the system in a form of metric values and retrives decisions,
- Executes decisions through the cloud vendor API (e.g. Amazon Web Services API) taking into account environment constraints (e.g. warm-up and cool-down periods).

The raw measurements are being collected with use of the Graphite monitoring tool [30]. Every machine executing the computations is expected to automatically start sending frequent reports through that tool as soon as it starts operating. The measurements can be reported at different intervals, e.g. CPU and memory statistics are sent every 10 seconds while the number of running virtual machines (VM) is reported once per minute. To introduce a coherent view of the environment, Graphite aggregates the values within a common interval, which in our case is set to one minute.

The role of *Policy Evaluation Service* is straightforward: it evaluates the state of the observed system with the use of the policy trained using the Proximal Policy Optimization. There are three possible results of the evaluation: *do nothing* (according to the policy a proper amount of resources is being used, there is no need to change the environment), *add another VM* (there are not enough resources in the current state of the system), *remove a VM* (there are too many machines used, one of the currently running ones should be stopped).

The decisions of the agent are not always translated directly to changes of the environment. Taking an action is always subject to the environment constraints: starting or stopping a virtual machine takes some time. In order to observe the effects of the previous interaction, we need to wait for some time, i.e. wait for a *warm-up* (starting a new VM) or *cool-down* (stopping a VM) of the system. We might also need to simply wait until the previous request gets fulfilled or handle a request failure.

The presented system does not make many assumptions about the workload for which the infrastructure is being controlled. It is required though that: a) the work can be split into multiple, independent tasks, b) there is a concept of a queue which can be monitored to check how many tasks are waiting for processing, c) the VMs used for processing can be stopped, the tasks which would be processed by them would be retried, d) executing the same job multiple times does not carry a risk of system malfunction.

It is assumed that the process which generates workload is being run on a machine which is not subject to automatic scaling. In order to fulfill monitoring requirements it might be necessary to instrument that machine and the software package which is responsible for generating tasks.

The complete diagram of the architecture, including the relationships of the discussed components, is presented in Fig. 1.

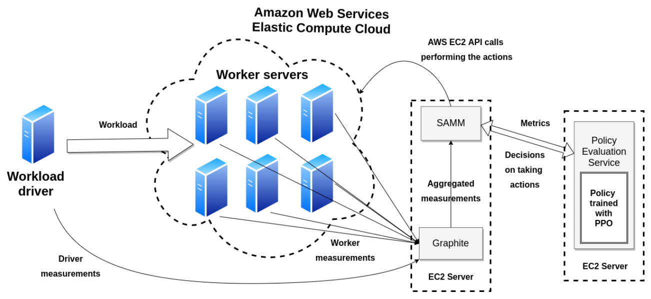


Fig. 1: Components of the discussed system. Arrows denote interactions between them.

### 3.2 Training the Policy

Autonomous management systems face a serious risk of introducing unnecessary costs while they are being trained. If they would start experimenting with executing the actions with regard to a real live application, they could significantly degrade its performance. This would in turn lead to business losses. A common solution to such a problem is to simulate the cloud resources [31]. Thanks to this approach, we can train the model in a safe, isolated environment, where even catastrophic events have no real consequences. There is also a number of other advantages:

- The computational cost of the simulation is orders of magnitude lower than running the actual system. We can potentially parallelize this process and evaluate multiple agents in parallel.
- The time in the simulation can be easily controlled. In a relatively short amount of time we can expose the agent to events from within a long period of time.
- Results are repeatable: if we need to replicate the training of a policy, we can rerun the simulation with the same set of parameters and we can expect the simulator to behave exactly in the same way.
- We can safely try to fine-tune the policy by changing the training algorithm parameters and re-running the simulation.

Unfortunately, there is one major downside: the simulator differs from the real environment. It is very hard to include every factor which might potentially affect how the real system behaves. Part of our research is to evaluate whether this problem limits the ability of the agent to make optimal decisions, or the effect can be reduced due to generalization done by the DNN.

We have decided to train the policy using a simulated environment which has been implemented following the results of our prior research [32, 29]. The main simulation process utilizes the CloudSim Plus simulation framework [33] which has been used in a wide range of studies [34]. It is wrapped with the interface provided by the Open AI Gym framework [35]. This allows for decoupling the simulation from other elements of the system, which in turn allows to easily reuse the same environment in experiments with different algorithms. This also helps to parallelize the execution of the simulation in situations where multiple simulations need to be run simultaneously.

The agent observes the environment through the following metrics: number of running virtual machines, average CPU utilization, 90th percentile of CPU utilization, average RAM utilization, 90th percentile of RAM utilization, ratio of all the jobs waiting in the queue to all the jobs submitted, ratio of the jobs waiting in the queue submitted in the last monitoring interval to all the jobs submitted in the last monitoring interval.

The architecture of our training environment is presented in Fig. 2.

During the training, the simulated environment consisted of a single datacenter which could host up to a 1000 virtual machines. Each virtual machine could provide a 4 core processor with 16 GB of RAM, similar to *xlarge* Amazon EC2 instances. The initial number of virtual machines was pre-configured to 100. Each simulation was run until all tasks were processed.

The workload simulation was based on logs of the actual jobs executed on IBM SP2 cluster working in the Swedish Royal Institute of Technology. It consisted of 28490 batch jobs executed between October 1996 thru August 1997.

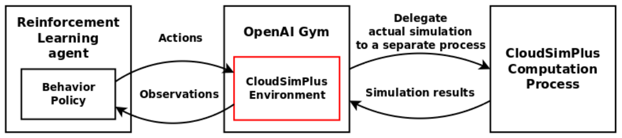

Reinforcement Learning agent
Behavior Policy
Actions
Observations
OpenAI Gym
CloudSimPlus Environment
Delegate actual simulation to a separate process
Simulation results
CloudSimPlus Computation Process

Fig. 2: Components of the training system; arrows denote interactions between them.

Jobs had varying time of execution and used different amounts of resources. The configuration of the simulated CPU cores was adjusted to the configuration of the simulation execution environment. To reduce the training time, the time in our simulation experiments was speeded up 1000 times.

The reward function was set up as the negative cost of running the infrastructure including some SLA penalties. This enabled us to formulate the training task as maximization of the objective function (minimizing the cost of the running infrastructure). The cost of running virtual machines was set to $0.2 per hour of their work. The SLA penalty was set to $0.00001 for every second of delay in task execution (e.g. waiting in the queue for execution).

As discussed in [29], the policy resulting from such a training can be applied to managing resources of workloads which are similar to the one used in simulations.

## 4 Experiments

We chose to use the *pytorch-dnn-evolution* tool [36] as an application for which the infrastructure is automatically managed. The tool implements a co-evolutionary algorithm discovering an optimal structure of a Deep Neural Network to solve a given problem [37]. This approach can be used for problems that can be solved using supervised learning, i.e. there is a training set available. In many such problems, however, due to the size of that dataset, evolutionary methods for discovering the network structure are very costly to apply. Evolution requires evaluating each candidate network by training it over this large dataset. In order to workaround this problem, we assume that by training on a small subset of the initial training dataset, neural networks can be still compared to each other.

Such an approach leads to an emergence of high number of relatively small tasks, which can be processed independently on a cluster of machines. These tasks are idempotent: they can be recalculated multiple times without a risk of corrupting the main evolutionary process. This means that each virtual machine used to conduct the training can be safely stopped at any point in time. Furthermore, the *pytorch-dnn-evolution* tool explicitly creates a queue which can be observed for monitoring the progress of the evolutionary process. Each iteration may have a different number of tasks. This allows to potentially reduce the amount of resources used and the cost of the evolution, if only we are able to design a dynamic resource allocation policy which will be able to add and remove VMs when necessary.

As a workload to which we have applied automatic scaling policies, we have chosen a simple evolutionary experiment which improves the architecture of a network which recognizes hand written numbers (the MNIST dataset [38]). The experiment consisted of executing 20 iterations of evolution over a population of 32 neural networks and 16 fitness predictors (subsets of 2000 samples from the original training set). Every network evaluation included 20 training iterations of a given fitness predictor.

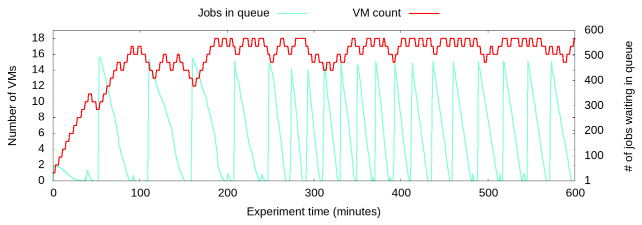
As our cloud environment we have decided to use Amazon Web Services Elastic Compute Cloud (AWS EC2) [39]. The setup consisted of up to 20 instances of *m5a.large* virtual machines from US East 2 (Ohio) region. All VMs have been executed in the same availability zone to reduce the risk of introducing random delays caused by the network communication latency. The first VM has been used to host the workload driver, together with SAMM and Graphite. The Policy Evaluation Service has been deployed to a separate machine because of its higher resource requirements. The management agent could run between 1 and 18 VMs as workers which actually performed the calculations.

To provide a comparison for the results of work of our policy, we also tried to automatically manage the set of used virtual machines with an Auto Scaling Group. This AWS EC2's feature allows to automatically start and stop VMs based on the average CPU usage of the already started machines. If that metric becomes higher than a defined threshold, a new machine is started. Similarly, if the metric drops below this threshold, one of the machines is stopped. For our workload, the value of 80% average CPU usage allowed to achieve the best results.

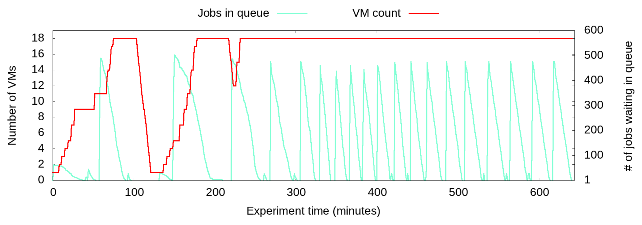
## 5 Experiment Results

Below we present the results of the automatic management of the resources with use of the policy trained using the PPO algorithm, on the one hand and the threshold-based AWS policy, on the other hand. Fig. 3 presents the number of virtual machines in the context of jobs waiting in the queue. The *step* shape of the charts comes from the fact that after performing an action through the API, the system needs to wait for 180 seconds. This is necessary to allow for startup of newly added machines or shutdown of the stopped ones.

Under the management of the PPO-trained policy, the overall experiment time was 598 minutes and the total cost of managed resources was $13.24. In the case of the threshold-based approach it was respectively 642 minutes and $14.21. The use of the first policy led to faster execution (by 42 minutes - 6.5%) and lower resources cost (by $0.97 - 6.8%) execution. For a fair cost comparison in the case of the first policy we need to include also the cost of the additional VM used to generate decisions ($0.86). This reduces the cost difference to $0.11 (0.7%). If the experiment would run for a longer period of time or more expensive resources would be used, such an additional cost would become negligible. The resources required to run the policy are constant, so they would become a very small percentage of the overall cloud resources cost.



(a) Policy trained with use of PPO



(b) Threshold-based policy

Fig. 3: Number of started VMs in context of jobs waiting in the queue.

The PPO policy was quite conservative in introducing changes and maintained a similar amount of resources most of the time. From time to time it tried to reduce the number of virtual machines after a slightly smaller number of tasks were submitted in an iteration. Those drops were quickly compensated.

The threshold-based policy was more aggressive: after the first iteration it reduced the number of used VMs to 1. Unfortunately, this leads to a slow-down of the whole processing. The final task in this iteration was being picked up subsequently by machines, which would get stopped after a couple of minutes. This delayed its execution until there was only a single machine, which would not get stopped. That single situation delayed the overall evolution process. It is worth noting that after the initial attempts to reduce the amount of resources used, that policy set the number of used resources to a maximum of 18 and maintained this number till the completion of the experiment.

We acknowledge that it is not fully a fair comparison. It might be possible to set the threshold in a way which will allow to avoid the slow-down described above. Using a more complex policy using the same paradigm (a multiple-threshold policy) might help to achieve even better results. Those experiments prove however that the use of the PPO-trained policy renders results on-par with other approaches. At the same time, in the new approach, one can take multiple factors into account in the decision making process (e.g. amount of free RAM), there is no need to set a fixed threshold. Furthermore, as demonstrated in this experiment, it is possible to create a generic policy which can be used also to manage resources for other workloads in which the resources are used in a similar way.

## 6 Conclusions and Further Research

In this paper we have presented a novel approach to autonomous resource management which uses recent advancements in the Deep Reinforcement Learning area. We explained how to train a cloud resource management policy using the Proximal Policy Optimization algorithm with use of a simulated cloud environment. Furthermore we demonstrated how to implement a system which enables deploying such a policy to a real cloud infrastructure - the AWS Elastic Compute Cloud. Finally, we showed that for a sample workload such a policy can manage the infrastructure in a more efficient manner comparing to a threshold-based policy. The careful examination of reasons for such a result revealed issues which led us to a different overall conclusion. Given more fine-tuning the threshold-based policy might be able to allow to achieve even better results. On the other hand, the DRL based approach offered slightly lower costs of infrastructure, while also having a number of other advantages (considering multiple decision factors, no requirement for setting the thresholds manually, re-using the policy across a range of similar applications).

Our approach to the training of the policy delivered good results. Even though we trained it by simulating a different, speeded up workload, our approach enabled to successfully manage the infrastructure for a real, sample machine-learning application. Simulations enabled us to avoid high costs of training. We were able to reduce the amount of time which was needed for a single simulation. At the same time we were able to avoid the cost of poor management decisions made by an untrained policy.

The presented system has a few limitations as well. Due to the grace period and ability to start or stop only a single VM, our policy could not react to environment changes fast enough. Furthermore, our policy was only able to efficiently handle situations, which it was exposed to in a prior training. When new jobs were being issued, the management decisions seemed reasonable, however after the workload had stopped, the number of used resources was not reduced immediately. In contrast, the threshold-based policy stopped all the machines within a couple of minutes once the evolution had stopped.

We plan to continue extending the approach discussed in this paper and mitigate the mentioned issues. We would like to extend the range of available actions in order to allow the policy to add or remove more virtual machines at once. Furthermore, we aim to modify the system to introduce a continuous policy improvement loop.

## Acknowledgement

The research presented in this paper was supported by the funds assigned to AGH University of Science and Technology by the Polish Ministry of Science and Higher Education. The experiments have been carried out on the PL-Grid infrastructure resources of ACC Cyfronet AGH and on the Amazon Web Services Elastic Compute Cloud.

## References

1. Tao Chen, Rami Bahsoon, and Xin Yao. A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. *ACM Comput. Surv.*, 51(3):61:1–61:40, June 2018.
2. R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning.* PhD thesis, 1984.
3. L. P. Kaelbling et al. Reinforcement learning: A survey. *CoRR*, cs.AI/9605103, 1996.
4. V. Mnih et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.
5. S. Gu et al. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *Proceedings 2017 IEEE International Conference on Robotics and Automation (ICRA)*, Piscataway, NJ, USA, May 2017. IEEE.
6. D. Silver et al. Mastering the game of go without human knowledge. *Nature*, 550:354–359, October 2017.
7. V. Mnih et al. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.
8. V. Mnih et al. Asynchronous methods for deep reinforcement learning. In *Proc. of The 33rd International Conference on Machine Learning*, volume 48, pages 1928–1937. PMLR, 20–22 June 2016.
9. J. Schulman et al. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.
10. Funika W. et al. Towards autonomic semantic-based management of distributed applications. *Computer Science*, 11(0):51, 2013.
11. Y. Brun et al. Engineering Self-Adaptive Systems through Feedback Loops. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 48–70. Springer, 2009.
12. H. Hoffman. *Seec: A Framework for Self-aware Management of Goals and Constraints in Computing Systems (Power-aware Computing, Accuracy-aware Computing, Adaptive Computing, Autonomic Computing).* PhD thesis, Cambridge, MA, USA, 2013. AAI0829261.
13. An architectural blueprint for autonomic computing. Technical report, IBM, June 2005.
14. N. Huber et al. Model-based self-adaptive resource allocation in virtualized environments. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 90–99, New York, NY, USA, 2011. ACM.
15. S. Kim et al. An allocation and provisioning model of science cloud for high throughput computing applications. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, pages 27:1–27:8, New York, NY, USA, 2013. ACM.
16. D. Kateb et al. Generic cloud platform multi-objective optimization leveraging models@run.time. 03 2014.
17. D. Minarolli and B. Freisleben. Distributed resource allocation to virtual machines via artificial neural networks. In *Proceedings of the 2014 22Nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '14, pages 490–499, Washington, DC, USA, 2014. IEEE Computer Society.
18. B. Wickremasinghe et al. Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications. In *2010 24th IEEE*

*International Conference on Advanced Information Networking and Applications*, pages 446–452, April 2010.
19. C. Qu et al. A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances. *CoRR*, abs/1509.05197, 2015.
20. M. A. Rodriguez et al. Containers orchestration with cost-efficient autoscaling in cloud computing environments. *CoRR*, abs/1812.00300, 2018.
21. H. Fernandez et al. Autoscaling web applications in heterogeneous cloud infrastructures. In *Proceedings of the 2014 IEEE International Conference on Cloud Engineering*, IC2E '14, pages 195–204, Washington, DC, USA, 2014.
22. P. Koperek and W. Funika. Dynamic business metrics-driven resource provisioning in cloud environments. In *Parallel Processing and Applied Mathematics*, LNCS 7204, pages 171–180. Springer Berlin Heidelberg, 2012.
23. S. Ferretti et al. Qos–aware clouds. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 321–328, July 2010.
24. A. Ashraf et al. Cramp: Cost-efficient resource allocation for multiple web applications with proactive scaling. In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pages 581–586, Dec 2012.
25. C.-Z. Xu et al. Url: A unified reinforcement learning approach for autonomic cloud management. *J. Parallel Distrib. Comput.*, 72:95–105, 02 2012.
26. P. Xiong et al. Smartsla: Cost-sensitive management of virtualized resources for cpu-bound database services. *IEEE Transactions on Parallel and Distributed Systems*, 26:1–1, 01 2014.
27. Z. Wang et al. Automated cloud provisioning on AWS using deep reinforcement learning. *CoRR*, abs/1709.04305, 2017.
28. J. Kitowski et al. Computer simulation of heuristic reinforcement learning system for nuclear plant load changes control. *Computer Physics Communications*, 18:339–352, 1979.
29. W. Funika and P. Koperek. Evaluating the use of policy gradient optimization approach for automatic cloud resource provisioning. In *Parallel Processing and Applied Mathematics*, LNCS 12043, pages 467–478, 2020.
30. Graphite Project. `https://graphiteapp.org/`. Accessed: 2019-11-28.
31. W. Rząsa. Predicting performance in a paas environment: a case study for a web application. *Computer Science*, 18(1):21, 2017.
32. W. Funika et al. Repeatable experiments in the cloud resources management domain with use of reinforcement learning. In *Cracow Grid Workshop 2018*, pages 31–32. ACC Cyfronet AGH, Kraków, 2018.
33. M. C. S. Filho et al. Cloudsim plus: A cloud computing simulation framework pursuing software engineering principles for improved modularity, extensibility and correctness. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 400–406, May 2017.
34. A. Hussain et al. Investigation of cloud scheduling algorithms for resource utilization using cloudsim. *Computing and Informatics*, 38:525–554, 07 2019.
35. G. Brockman et al. OpenAI Gym, 2016. cite arxiv:1606.01540.
36. PyTorch DNN Evolution. `https://gitlab.com/pkoperek/pytorch-dnn-evolution`. Accessed: 2019-12-01.
37. W. Funika and P. Koperek. Co-evolution of fitness predictors and deep neural networks. In *Parallel Processing and Applied Mathematics*, LNCS 10777, pages 555–564. Springer International Publishing, 2018.
38. Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010.
39. Amazon Web Services Elastic Compute Cloud. `https://aws.amazon.com/ec2/`. Accessed: 2019-12-30.