Deep Reinforcement Learning in Cloud Elasticity through Offline Learning and Return Based Scaling

Miltiadis Chrysopoulos
CSLAB NTUA
Athens, Greece
miltos503@gmail.com

Ioannis Konstantinou
University of Thessaly
Dept. of Informatics & Telecommunications
Lamia, Greece
ikons@uth.gr

Nectarios Koziris
CSLAB NTUA
Athens, Greece
nkoziris@cslab.ece.ntua.gr

***Abstract*—Elastic resource allocation is a desirable feature of cloud environments, and one of the main reasons for their widespread adoption. Resource elasticity allows for adaptive and real-time infrastructure scaling that can follow workload fluctuations in a cost-effective manner without sacrificing performance. Promising Machine Learning (ML) approaches employ Reinforcement Learning (RL) where an agent interacts with the cloud environment by employing actions, observing the reward of their outcome and modifying its strategy accordingly. Nevertheless, one of the main problems of RL in this setup is that to acquire a sufficient initial knowledge of the environment, the agent needs to perform numerous time-consuming and performance-degrading interactions with the cloud. In this work we design and implement RBS-CQL, a Deep-RL Kubernetes agent system to monitor and automatically scale the containers of a NoSQL application according to incoming workload. We combine training optimization techniques from contemporary literature as well as offline RL algorithms to reduce the training time. We provide empirical results that show that RBS-CQL achieves systematic improvement of more than 10% compared to its online equivalent for a given number of experiences and that it is able to extract improved decision-making policies even from data of lower quality.**

***Index Terms*—Cassandra, K8s, NoSQL, containers**

I. Introduction

Cloud native applications are one of the main focus points of enterprise software development. In 2021 about 30% of new generated digital workload is deployed on cloud native platforms and it is estimated that this figure can climb up to more than 90% by 2025 [1]. The dynamic and scalable nature of cloud native applications is a promising alternative to on-premises development because it alleviates an organization from the costs and risk of maintaining costly infrastructure to support its software operations. Moreover, scaling is limited, time consuming and even impossible in certain scenarios.

Containerization enhances the potential of cloud native applications because it offers a lightweight alternative to virtual machines that need to boot an OS instance to run. Using containerization, the components of an application can run with a significantly smaller hardware and time overhead. Capitalizing on these properties, cloud applications can be organized in components that execute atomic functions of the application, rather then running monolithic application instances. This software development paradigm is called microservices and it offers increased flexibility in terms of scaling because each component can be scaled independently.

As containerized applications become more complex and more distinct components are added to the deployment, monitoring, scaling and connecting the components of the application becomes a tedious task [2]. The rapid increase in DevOps complexity of containerized applications limited their adoption and created the need for orchestration mechanisms. Kubernetes (K8s)[1] is a software solution that solves the orchestration problem effectively, a fact that resulted in its increased adoption by enterprises since its release in 2014. Kubernetes also offers resource-based autoscaling features such as the Horizontal Pod Autoscaler (HPA) [3]. This autoscaling feature uses CPU and memory utilization thresholds to estimate the optimal number of instances to run. Most cloud services vendors offer threshold based autoscalers that increase the number of application instances as the incoming traffic increases (a brief experimental comparison of different industrial autoscalers can be found here [4]). These methods are oversimplistic and can only provide limited performance guarantees while requiring specific domain knowledge (see Section 8 of [5], where rule-based resource estimation is one out of the 6 different categories).

An alternative approach to threshold based autoscaling, is to utilize algorithms from the domain of RL (section 8.5.1 in [5] or a complete survey in [6]). RL formalizes the idea of an agent that learns effective decision-making policies by performing trial and error interactions with a system. Deep Reinforcement Learning (Deep RL, firstly introduced by DeepMind [7]) enhances the capabilities of RL by utilizing neural networks as high order non-linear function approximators. In that way, problems that are described by more complex state spaces and could not be solved by original RL algorithms, are now solvable. However, there is a key limitation to applying Deep RL to the resource auto-scaling problem. In typical Deep RL problems mentioned in literature, the number of training steps required to reach an optimal solution is in the order of millions. In ordinary auto-scaling scenarios the time it takes for an action to take effect is in the order of minutes until the system completely stabilizes and all resources are fully

[1]https://kubernetes.io/

functional, making the "scaling latency" a first-class citizen in quantifying cloud elasticity [8]. Additionally, interacting with the application in a random manner is in some cases not desirable because it can cause disruptions or lead it to destructive states [4]. Considering the above, it is evident that in order to derive a feasible solution for the auto-scaling problem the number of interactions with the system must be reduced significantly or alternatively the information gain from each interaction must be maximized.

In this work we create an agent that monitors a containerized application and dynamically scales the deployed instances and consequently the consumed computational resources. To develop this agent, we create two neural networks that are trained using online and offline Reinforcement learning algorithms. There are two main challenges that need to be tackled by the proposed solution.

a) The number of parameters needed to describe the system state can increase arbitrarily, depending on the complexity of the deployed application. Also, the parameters are not discrete. This makes tabular applications of Reinforcement Learning ineffective and consequently the utilization of a neural network is justified.

b) The time between consecutive actions is in the order of minutes. This limits the rate of data accumulation, meaning that the model must extract as much information as possible from available data points.

To address these challenges, we propose a model that consists of two neural networks. The first model is trained in an online manner, using the well-established Double Deep Q Learning algorithm [9]. The second is trained with an offline dataset using the Conservative Q Learning algorithm [10] (i.e., CQL), a state of the art Offline RL algorithm. The offline dataset consists of the training experiences that are generated by the online model during its training. Essentially, the offline model receives the data generated by the policy of the online agent and trains without further interacting with the system. To decrease the time it takes to perform the scaling actions, we deploy our application inside a distributed Kubernetes cluster to leverage its automated scheduling features. The application we chose to monitor is a Cassandra NoSQL cluster. The choice of this application is based on the fact that its performance depends on several parameters [11], a fact that highlights the effectiveness of our model.

Our man contributions are the following:

- We identify, briefly analyze and bring into the wider cloud elasticity context two state-of-the-art promising approaches from the RL domain, namely Return Based Scaling (i.e., RBS) and Conservative Q Learning (i.e., CQL) that can optimize learning with minimal interaction with the environment.
- We present RBS-CQL, an elasticity agent system that implements, employs and combines the previous techniques to scale a cloud-native K8s workload consisting of a Cassandra NoSQL cluster in a public cloud testbed.
- We test RBS-CQL at several checkpoints of the training and show that our models outperform more than 10% online training up to a certain size of dataset. We also provide empirical results that indicate that from the point onwards where online training outperforms offline training, performance gains diminish dramatically in comparison with the training steps required to achieve this improvement.

The paper is structured as follows: Section II gives the reader the necessary background regarding RL and NoSQL, Section III describes the selected optimizations from the RL domain and explains their fit for our case, Section IV presents a detailed experimental evaluation of our approach whereas Sections VI and VII conclude our work.

II. Preliminaries

The aim of this section is to present a very brief background regarding the system(s) tackled in this work and also the theoretical foundations used to model the elasticity work. Subsection II-A gives a brief overview of K8s and Kassandra. Subsection II-B gives the necessary brief information regarding reinforcement learning.

A. *Containerization and NoSQL*

Virtualization comes with a significant hardware overhead, since every VM needs to have its dedicated kernel which results in increased resource usage. Containers alleviate this problem by providing a "lightweight" alternative for running isolated applications, while sharing the same kernel of the host Operating System (OS).

Containerized applications have significantly reduced booting times, since there is no need to boot an operating system. This means that deploying and destroying containers on demand, is much cheaper in terms of resources and elapsed time. Kubernetes, initially launched in 2014 is a container scheduler that runs on a cluster of physical machines.

K8s introduces the notion of "pods" when it comes to resource scheduling. Pods are the smallest unit of computing that can be deployed in the K8s ecosystem. A pod is a group of containers that are always co-scheduled on the same node and share the same computational resources, filesystem and storage.

Cassandra [12] is a distributed open-source database management system. It is a NoSQL solution and was initially developed by Facebook to support the Inbox Search feature until 2010. A Cassandra cluster works with a Peer to Peer (P2P) architecture, meaning that every node is connected to all other nodes. Also, every node is aware of the data distribution on all other nodes. As a result, every node is capable of serving clients and perform all database operations offering scalability and eliminating single points of failure.

K8ssandra[2] is a K8s operator that packages an entire Cassandra deployment (storage nodes, monitoring, API nodes, etc.) into a cloud native offering. K8ssandra pods consist of three containers. A Cassandra container that runs the Cassandra image, a logger container that stores log information

[2]https://k8ssandra.io/

about Cassandra operation within the pod and is useful for troubleshooting and an init-container that handles configuration settings as the pod is initialized.

B. Reinforcement Learning

The goal of RL is to maximize the overall rewards an agent can gain by interacting with an environment. The maximization infers the need for an optimality criterion. More specifically, we must define how the agent evaluates the rewards in order to modify its behavior. There are three approaches that are most commonly used for this purpose:

- The *finite-horizon* model, where the agent is expected to maximize the expected rewards for a finite number of steps h: $E[\sum_{t=0}^{h} r_t]$. This approach is suitable when the agent can only perform a fixed number of steps in the environment, for example in a game that ends after a fixed number of turns. For this approach we can either opt for a fixed or receding horizon. Fixed horizon means that the agent does not have a stationary policy. On the first step it chooses the h-step optimal action, on the second step it chooses the (h-1)-step optional action and so on. In the receding horizon case, the agent has a fixed policy and it just alters the h number of forward steps it will look to decide on an action. This approach can be problematic since it limits the number of steps ahead the agent must consider and this information is not always known beforehand.
- Another approach is the average-reward model, in which the agent is supposed to maximize the long-term average reward: $lim_{h\to\infty}E(\frac{1}{h}\sum_{t=0}^{h} r_t)$. Again, this approach is problematic because it does not distinguish between a policy that prioritizes short term large rewards to and agent that prioritizes long term larger rewards.
- To avoid these two problematic cases, an alternative form of reward model is used the infinite-horizon discounted model. The equation to maximize is (where $0 \leq \gamma \leq 1$): $E(\sum_{t=0}^{\to\infty} \gamma^t r_t)$. In this case, the long-term rewards are considered but they are discounted by a factor γ for each additional step it takes to receive them. The discount factor serves more than one purposes. First of all, it is effectively setting a frame of effective rewards. The smaller the discount factor is, the faster future rewards diminish to zero and are not affecting the result. It also conveniently ensures that the sum will converge to a finite number given that the rewards are themselves a finite number. This mathematical tractability is the dominant reason this type of reward is has received the most attention [13].

Markov Decision Process (MDP): Markov Models are stochastic models created to describe non deterministic processes [14]. Markov models are described by a number of states and a transition probability between states. One of the properties that make them widely adopted, is that they are memoryless, meaning that the behavior of the system only depends on the current state. If our problem can be transformed into a Markov Model, then by observing the current state we have sufficient information to decide on the next states. An *MDP* is an extension of the Markov chain, where actions are added to each state and rewards for executing these actions. These processes are suitable to describe RL problems because they are expressive enough to describe the process of an agent taking deliberate actions and not only transition to states stochastically and also receive rewards. Using these models, we can calculate optimal policies for agents.

An MDP consists of the following:

- a set of states $s \in S$
- a set of actions $\alpha \in A$
- a reward function $R : S \times A \to \mathbb{R}$
- a transition function $T : S \times A \times S \to [0, 1]$

Before discussing algorithms for finding MDP models, we will first explore techniques for finding an optimal policy π^*, given that we already have the MDP model available. We call *optimal value* of a state, the expected infinite discounted reward the agent will gain if it starts from this state and executes the optimal policy: $V^*_{(S)} = maxE(\sum_{t=0}^{\to\infty} \gamma^t r_t)$ This optimal value is unique and is the solution to the *Bellman optimality equations* [15]:

$$V^*_{(S)} = max(R(s, \alpha)) + \gamma \sum_{s} T(s, \alpha, s')V^*(s')) \quad (1)$$

$$\pi^*_{(S)} = argmax(R(s, \alpha)) + \gamma \sum_{s} T(s, \alpha, s')V^*(s')) \quad (2)$$

The optimal values can be calculated using the *value iteration algorithm* [14]. At every step of the algorithm, we iterate over all states and all actions and calculate the next estimation about function V. Then we update our next step estimation as the max calculated value for each state. This algorithm is shown to converge to the optimal policy [15]. There is no obvious termination criterion of the algorithm. It has been proven however, that if the difference between two successive value functions is less than ϵ then value of the greedy policy differs from the optimal value by no more than $\frac{2\epsilon\gamma}{1-\gamma}$. This provides an effective stopping criterion for the algorithm. By greedy policy we mean taking the max value at each iteration as an update for the value function. It is apparent that the policy can be arbitrarily close to the optimal, depending on the value of ϵ we choose.

Learning an Optimal Policy: In the previous paragraph when we started describing an MDP and how to calculate the optimal value function of a model, we assumed that we already had the model parameters available to us. This is not always the case though. In many problems we do not have prior knowledge of the model that describes the system the agent interacts with. Moreover, even if we do have such knowledge, it is not always desirable to provide this knowledge to the agent because it inserts some form of bias based on our perception of this system. It is preferable that the agent is able to learn the dynamics of the system on its own and still be able to find an optimal policy.

The agent interacts with the environment and observes the states these actions lead to and the rewards it receives.

This is the only means by which the agent can observe and consequently gain information about the environment. There are two approaches to reach an optimal policy [13]:

- *Model Free Systems*: Learn the action controller without learning a model of the environment.
- *Model Based Systems*: Learn a model of the environment and then derive the controller.

The question of which approach is better still remains a bone of contention in the academic community. The only thing that is certain is that both approaches have been used to provide optimal solutions for different problems. For the scope of this work, we will attempt to solve the proposed problem using model free systems and specifically Q Learning.

Q Learning is a widely adopted method that has been able to solve many RL problems due to the ease of implementation compared to other methods. To understand Q Learning we must define the function Q^* as a function of states and actions. Q represents the expected discounted reinforcement of taking action α in state s and then continuing to make optimal options of action for the successive states. Considering the above, $V^*_{(s)}$ is the value of s assuming we perform the optimal action from step one so essentially $V^*(\,s) \,= max_\alpha Q^*(\,s, \alpha)$. Using this equation along with Equation 1 the Q function can be written recursively as:

$$Q^*(s,\alpha) = R(s,\alpha) + \gamma \sum_{s \in S} T(s,\alpha,s') max_{a'} Q^*(s',\alpha')$$

$Q^*(s,\alpha)$ also provides us with the optimal policy $\pi^*_{(s)}$ as $\pi^*_{(s)} = argmaxQ^*(s,\alpha)$ using Equation 2 and substituting Q for V. The recursive definition of Q and the fact that it provides an explicit way of deciding on an action on each step allows us to estimate Q values online and also use them to define the optimal policy, by taking the maximum Q for the current state at each step. The Q Learning rule is:

$$Q(s,\alpha) = Q(s,\alpha) + \alpha(r + \gamma max_{\alpha'} Q(s',\alpha') - Q(s,\alpha))$$

where $\langle s,\alpha,r,s' \rangle$ is an experience tuple. It is proven that if every action is executed an infinite number of times in each state and α is decayed appropriately then the Q values will converge with probability 1 to Q^*.

Deep Reinforcement Learning: RL algorithms have existed more than two decades in academic literature. Nevertheless, their applications were limited. The main problem is that every reinforcement learning algorithm needs at some point an approximator function to estimate the value function of the policy distribution of the actions it performs. Explicitly defining such a function is impossible and using simple approximators, like linear ones, greatly impacts performance and limits the scope of the problems it can be applied to. Deep neural networks provided an effective non linear approximator that is able to fit to very high dimensionality non-linear functions given enough samples and time to converge. The increase in attention neural networks have received since 2010 has also revitalized RL applications that leverage this new powerful tool to learn the estimators they need.

Deep Q Learning (DQN): As we mentioned in the definition of Q Learning, this algorithm attempts to find the sequence of actions that maximize total discounted rewards by trying to estimate the Q for any given pair of state-action and then make greedy choices by opting for the action that has the highest Q value every time. It is apparent, that the better the estimation of the Q values, the closer the action sequence to the optimal will be. Attempting to solve a problem using the definition of Q Learning is inefficient, because according to the original algorithm every sequence is evaluated independently and no form of generalization can take place. To enable generalization, a parameterized Q function is used $Q(s,\alpha;\theta) \approx Q^*(s,\alpha)$ where θ is a set of trainable parameters. In deep Q Learning the approximator is a neural network that attempts to estimate optimal Q values for all action sequences using the same function.

DQN models are trained using tuples of $\langle s,\alpha,r,s' \rangle$ (state, action, reward, next state). The difference in implementation compared to the original algorithm and other RL implementations is that the tuples are not used for training in the same order as the agent observes them. In DQL a structure known as *replay memory* is used. The replay memory is a fixed size buffer that stores the N latest tuples observed by the agent. At every step first an observation of the state is made. Then an action is chosen based on the observed state. Finally, the state that the action leads to is observed and the reward is calculated. The new tuple is stored in the memory and then m random tuples are chosen from the memory to train the agent. The process is then repeated.

Using a replay memory instead of just using the samples as they are received for training provides three important advantages. The first is that this method of training leads to more efficient sample utilization. Every sample is almost certainly used more than once for weight updates. Combined with the fact that neural networks demand small learning rates to converge, it is very likely that using a training tuple only once to update the network weight is not sufficient to gain all the information possible from this training sample. This method increased the chances that a tuple will be used more than once so that more information can be extracted from the sample. The second advantage is that this training technique breaks the strong correlations between successive samples. This is important because randomized samples reduce the variance of the updates, which leads to faster convergence. Additionally, learning on-policy is prone to get stuck in local minima. The reason is that at every step the network makes a choice based on its parameters and then trains based on the choice it made. If the choice is locally optimal, the network is going to repeat the choice and ignore other options that potentially lead to greater overall reward [16].

Double Deep Q Learning (DDQN): The original Q Learning algorithm is known to overestimate the Q values of the model. This phenomenon is not necessarily harmful for the algorithm performance and the resulting policy. A common exploration technique called optimism in the face of uncertainty, is based on this idea. Every unexplored Q value is assigned a high numeric value, so that the algorithm is incentivized to sufficiently explore the state space before making greedy locally optimal actions. Moreover, if the overestimation of the values is uniform then the dynamics of the action preferences are preserved, leading to the optimal policy. In this work [9], however, it is claimed that in several applications of DQN the overestimation is not uniform and it indeed harms performance.

To alleviate the overestimation, the authors propose to decouple the action selection from the action evaluation. The two networks of the agent are called online and target network accordingly. The online network is updated normally using the training samples. The target network is a lagging copy of the online network, meaning that every N steps, the parameters of the online network are copied to the target network. The greedy policy or action selection is evaluated using the online network. Then the target network is used to estimate the Q value of the state action pair and the weight update of the online network is performed using this estimation. The Q estimation rule for DDQN is shown below:

$$L^{DDQN} = R_{t+1} + \gamma Q(S_{t+1}, argmax_{\alpha} Q(S_t, \alpha; \theta_t); \theta'_t)$$

where θ_t are the parameters of the online network and θ'_t the parameters of the target network.

The frequency of synchronization between target and online network is not specific and in reality, is a tunable hyperparameter. Small values degrade the model to a simple DQN agent and introduce overestimation. Large values slow down learning because the updated Q values are not known to the evaluator for longer time-frames. According to state-of-the-art implementations, it seems that a reasonable decision is to synchronize the weights after every episode. DDQN indeed solves the overestimation problem and leads to higher performance in most tasks and for that reason DDQN has become the default implementation for solving problems using the Q Learning algorithm.

III. Optimizing Reinforcement Learning through Scaling and Offline Learning

In this Section we present the proposed optimizations to enhance the learning process in a typical RL-based elasticity system. We take into account the practical problems that arise in such a system, and we propose two different approaches to alleviate them, namely Return Based Scaling and Offline Learning.

A. *Return Based Scaling (RBS)*

Scaling issues in Reinforcement Learning models is a tedious task but also a necessary one because when errors scales vary across different stages of training, it can hinder or obstruct the convergence of the model. Especially in model-free algorithms, where the agent has to accurately estimate the value function that describes the underlying dynamics of the problem, scaling issues are even more severe.

There are various factors that can affect the error scales during training and each one of them can be detrimental for the model convergence. The most common is the reward function. Every Q value is the discounted sum of the current and future rewards the agent expects to accumulate. The greater the variance of the rewards is, the greater the disparity of the Q values can be during training. This can lead to error scales that vary in many orders of magnitude due to the cumulative nature of the Q values. Even if the reward function does not display high variance, it is possible that during an update the estimated and observed Q value vary greatly, leading to a high numeric value of error. Using such a value for an update can distort the weights convergence because neural networks are smooth function approximators. This phenomenon is more likely to happen during the early stages of training, where the agent explores the state space and it possible that it had not acted optimally around a specific part of the state space until that point. It is also possible to happen when at some point in the training the agent discovers new possibilities for higher rewards that became available after the policy started to change due to the training. Finally, the discount factor also greatly affects the arithmetic values of Q values. Trying different discount factors for the same problem may demand different scaling of the rewards, adding to the struggles, reinforcement learning practitioners face when they try to parameterize their models.

To overcome the aforementioned problems, reinforcement learning practitioners resorted to empirical solutions. These solutions may be efficient depending on the dynamics of a specific problem and the distribution of the reward function but they are not widely applicable. Some examples are reward clipping and reward or return normalization [9], [17]. These methods can effectively address the scaling problem but they are problematic because they hide certain aspects of the dynamics of the problem, that are expressed by the variance in the values of the reward function. As a result, these methods can make it impossible for an agent to reach the optimal policy due to the distortion introduced. Tom et al. [18] propose an alternative approach, to the scaling problem that preserves the dynamics of the problem while alleviating the numerical fluctuations between updates of the Q values and consequently the weights. They propose to apply the scaling directly in the temporal difference, meaning the input of the loss function of the neural network. The scaling factor is adaptive and is updated during each training step. The scaling problem of the updates is more noticeable during the early stages of the training because as the model approaches convergence, errors should approach 0 asymptotically. The derivation of the scaling factor is $\delta = R_t + \gamma V'_{t+1} - V_t$ where R_t is the reward observed at step t and V_t the estimated Q value of the model at step t. To find an approximation of the scaling factor we must estimate $V[\delta]$. At the early stages of training, we can assume that the rewards are independent from the Q values since the agent performs random actions almost always. Using this assumption, we can express $V[\delta]$ as:

$$\begin{aligned}V[\delta] &= V[R] + V[\gamma(V' - V)] + V[(1-\gamma)V] \\ &= V[R] + \gamma^2 V[V' - V] + V[\gamma]E[(V' - V)^2] + \\ &(1-\gamma^2)V[V] + V[\gamma]E[V^2]\end{aligned}$$

At every step Q values are updated with the rule $Q = R + \gamma Q'$ and Q values estimate the overall gain of the agent. It is reasonable to substitute V for G and for the one step difference $G' - G = R + (1-\gamma)G$. Also $G = \sum_t \gamma^t R_t$, so $E[G] \approx (1\gamma)E[R]$. Using these equations and substituting in $V[\delta]$ we can write: $V[\delta] = V[R] + (1-\gamma)^2 V[G] + V[\gamma]E[G^2]$. The term $(1-\gamma)^2 V[G]$ can be neglected because it is dominated by the other two. Also, when γ is constant or when the training is continuous and not divided in episodes, the term $V[\gamma]E[G^2]$ is also neglected leading to $V[\delta] = V[R]$. The scaling factor is σ where $\sigma^2 = V[\delta]$. The difference between this method is that $V[\delta]$ is calculated online at every step of the training. The authors also account for some edge cases that can occur during training. The most notable is the case where batch training is used and the variance of the rewards of the batch is greater than the overall variance.To avoid detrimental updates the scaling factor must be altered to $\sigma^2 = \max(V[\delta], V[\delta_{batch}])$

B. *Offline Reinforcement Learning (CQL)*

Offline Reinforcement Learning is a lucrative research field that has been drawing increasing attention over the latest years. The reason is that in theory, offline reinforcement learning can leverage the immense datasets that exist and effectively train agents based on these static datasets without further interaction. Currently, reinforcement learning is an active learning process, where the agent performs an action observes the results and then reiterates. This approach has limited applicability because first of all the quantities of data that can be generated are limited compared to offline training. State-of-the-art models of machine learning owe a major part of their success to the immense amounts of the training datasets they are presented. Except for the dataset limitations, interactions with the environment can be costly and/or catastrophic in several applications such as robotics or medical applications. For the reasons stated above, effectively applying offline reinforcement learning is a key challenge for the adoption of reinforcement learning in real-world environments.

The problem of most value based off-policy offline Reinforcement Learning methods is that they display poor performance in reality. The main reasons of failure are over-fitting and out-of-distribution actions (OOD). These problems usually manifest themselves as erroneous overestimations of the value function at certain states. More specifically, the problem lies in the fact that the Bellman optimization algorithm [15] tries to sample actions from the learned policy that is created as the model is trained but the Q values can only be trained on values sampled from the policy that generated the offline dataset. Since the algorithm is created to use the learned policy, it often leads to OOD actions. When these actions have erroneously high values, they lead to overestimations. Typical offline Reinforcement Learning applications mitigate this effect by restraining the algorithm from opting for unobserved states. These attempts however result in over-restrictive policies that limit the performance of the agent during testing.

A promising recent approach that tackles the afforementioned problems is Conservative Q Learning (CQL) presented in [10]. In our work we have integrated this approach to our cloud elasticity agent. We now give a brief theoretical explanation of the reasons why it can help in the learing process.

The aim of CQL is to estimate the value function $V^\pi(s)$ of a target policy $\pi(s)$ given a static dataset D that is generated by a behavior policy $\pi_\beta(\alpha \mid s)$. Ideally, the target policy is identical to the optimal policy. To learn in a conservative manner, an additional term is added to the minimization equation alongside the standard Bellman objective that is used in online Reinforcement Learning. The intuition behind this additional term is that since the values of the dataset D are generating by the behavior policy $\pi_\beta(\alpha \mid s)$, actions that are more likely according to this policy to be overestimated and so this additional term acts as a penalty for these actions. The objective function of conservative Q- Learning is provided by the equation:

$$\begin{aligned}Q^{k+1} = \min_Q \alpha \cdot (&E_{s\sim D, \alpha\sim\mu(\alpha|s)}[Q(s,\alpha)] - \\ &E_{s\sim D, \alpha\sim\pi_\beta(\alpha|s)}[Q(s,\alpha)]) + \frac{1}{2}L\end{aligned} \tag{3}$$

where L is the standard Bellman objective function and $\mu(\alpha \mid s)$ is the desired distribution action-states after training. The authors that propose the solution prove that for $\mu = \pi$ the resulting estimation of the value function and the Q values satisfies the restriction $\hat{V}^\pi(s) < V^\pi(s) \forall s \in D$, meaning that every Q value that can be estimated using the static dataset, is bounded by the actual value of the Value function, so overestimation is eliminated. The constant α is a hyperparameter of the optimization problem. In reality this constant needs to be sufficiently big for a dataset of fixed size. In other words, the larger the size of the dataset, the smaller α can be. Asymptotically, for a large enough dataset α can take very small numeric values and the objective function is dominated by the Bellman objective term.

Observing Equation 3 we notice that the minimization involves *a priori* knowledge of the distribution $\mu(\alpha \mid s)$. However, μ is a part of the training process and after a sufficient number of training steps we want $\mu = \pi$. Since μ is a part of the optimization problem we can include a maximization over μ in the conservative-learning term so that at every iteration the objective function is

$$\begin{aligned}Q^{k+1} = \min_Q \max_\mu \alpha \cdot (&E_{s\sim D, \alpha\sim\mu(\alpha|s)}[Q(s,\alpha)] - \\ &E_{s\sim D, \alpha\sim\pi_\beta(\alpha|s)}[Q(s,\alpha)]) + \frac{1}{2}L + R(\mu)\end{aligned}$$

where $R(\mu)$ is a regularizer term. A reasonable choice of $R(\mu)$ is the Kullback-Liebler divergence (KL). KL-divergence $D_{KL}(P\|Q)$ is a type of statistical distance that expresses the additional surprise or uncertainty introduced because of our choice to use as a model a distribution Q when the actual distribution is P. In our case $P = \mu$ and Q is a prior distribution of action-states. When the distribution of actions is almost uniform at every state, then the maximization over μ results in a soft-max of the Q-values at any given state and the objective function is transformed to:

$$Q^{k+1} = \min_{Q} \alpha \cdot E_{s\sim D}(\log \sum_{\alpha} exp(Q(s,\alpha)) - E_{s\sim D, \alpha\sim\pi_\beta(\alpha|s)}[Q(s,\alpha)]) + \frac{1}{2}L \tag{4}$$

Transforming equation 4 to a Loss function that can be used to calculate gradients for a neural network is straightforward:

$$L = \alpha \cdot E_{s\sim D}(\log \sum_{\alpha} exp(Q(s,\alpha)) - E_{s\sim D, \alpha\sim\pi_\beta(\alpha|s)}[Q(s,\alpha)]) + \frac{1}{2}L$$

At every step of the training we randomly pick a subset of the dataset and calculate L and then update the parameters of the network using the chosen optimizer, following [10].

IV. Evaluation

In this section we describe the way we coordinated the different components used to perform our experiments. We also present a detailed experimental evaluation of RBS-CQL managing a NoSQL cluster deployment in a public cloud setup. We used a K8ssandra deployment that runs inside a distributed Kubernetes cluster. The clients that generated the traffic load ran an instance of the YCSB [19] service, a typical workload generator for NoSQL database clusters, and a remote script monitored the number and nature of the generated requests. For the collection of metrics, we used a Prometheus instance that was deployed inside the K8s cluster as well. The VMs were hosted in the Okeanos [20] public cloud environment.

A. Experiment Setup

The Kubernetes cluster consists of 10 VMs, one of them acted as a Master Node and the rest as Worker Nodes. Each of the worker nodes had 4GB of RAM, 30GB of storage space and 2 virtual CPU cores. The master node had 8GB of RAM, 30GB of storage space and 4 virtual CPU cores. From the 30GB of available storage of every node, 15GB were allocated as a virtual disk and provided to the Kubernetes cluster as a Persistent Volume. Every worker node had an instance of the Kubernetes local volume static provisioner running on it. Its role is to manage the PersistentVolume lifecycle for pre-allocated disks by detecting and creating PVs for each local disk on the host, and cleaning up the disks when released. Every K8ssandra node needs at least 2GB of RAM to operate flawlessly. For this reason, only one K8ssandra node could run at a time per worker node. The resource limits that performed best in our setup were to use 2GB of RAM and 1 CPU core per K8ssandra node. We also allowed the nodes to exceed RAM usage to a margin of 0.5GB. This allowed the K8ssandra cluster to perform scaling operations even under head traffic and high percentage of resource utilization. Finally, we deployed 3 instances of Stargate nodes to coordinate incoming requests.

The role of the client generating the queries against our database was carried out by the YCSB framework. YCSB [19] is a benchmark tool written in Java that can generate traffic for several database systems. The workloads can be configured in terms of target loads (in requests per second), the number of operations to be executed, time limits of total execution time and the percentage of reads and writes among others. In essence, YCSB sets the arrival rate λ in queuing theory terms. Throughout our experiments we observed that $\lambda = \mu$ where μ defines the system's service rate or the system's observed throughput (the message queue size was not altered). To generate the traffic needed we created 4 additional VMs with the YCSB tool installed in them. The generated traffic was monitored by a remote script that sent commands to the VMs to execute partitions of the total workload and ensured that the total workload was evenly distributed among client machines.

In our setup all of the monitoring was performed on the server-side. The metrics were periodically scraped by the Prometheus instance and stored in its database. Then the script that executed the monitoring agent, performed PromQL queries against the database over HTTP to collect the metrics. After that the metrics were normalized and formatted as a numpy array and finally provided as input to the decision-making module.

B. Results

For the training of our models, in an approach similar to [21], [22], we used 17 parameters to describe the state of the cluster:

- The size of the cluster
- The average 98^{th} percentile latency measured by the stargate nodes
- The average 99^{th} percentile latency measured by the stargate nodes
- The average 999^{th} percentile latency measured by the stargate nodes
- The throughput or requests per second measured
- The throughput measured at the previous decision step
- The total free memory of the cluster as a percentage of the total available memory
- The total cached memory of the cluster as a percentage of the total available memory
- The average CPU utilization of the cluster
- The minimum CPU utilization of the cluster
- The maximum CPU utilization of the cluster
- The average CPU that is idle in the cluster
- The average CPU time spent waiting for IO
- The average IOPS in the cluster
- The average disk read throughput of the nodes in the cluster

- The average disk write throughput of the nodes in the cluster
- The percentage of reads in the incoming load

All the metrics are scraped by Prometheus every 10 seconds and the measurements are averaged over a 5 minutes interval. The reward function used to evaluate every state is:

$$R = 0.01 * throughput - (VMs - B)$$

where B is the minimum cluster size. Following the observations in [23], the selected reward function instructs the RL agent to optimize the cluster throughput while keeping the cluster size (and therefore the cloud cost) as small as possible to accommodate the incoming workload traffic. The rationale behind this approach is that the observed throughput in essence consists of the respective "profit" of a cloud service (e.g., by charging clients on a per-query basis) whereas the cluster size consists of the respective service "cost". Therefore, the goal is to maximize profit while minimizing cost. In this work we do not focus on the reward function selection. For a detailed discussion on this matter please refer to [23].

Regarding the network architecture of the online agent, we used a fully connected network with 2 hidden layers. The first hidden consisted of 48 nodes and the second layer consisted of 24 nodes. The replay memory buffer was set to store the last 300 experiences and the weights were updated with a learning rate of $\alpha = 0.001$. The discount factor was set to $\gamma = 0.99$. To avoid weight updates that could lead to the divergence of the system, we used the return-based scaling (RBS) technique described in the previous section to normalize the loss at every update step according to the running variance of the rewards of the past experiences. The training starts with a replay memory of 300 random experiences. The agent then performs 500 annealing steps, with epsilon decaying from 1 to 0.1 linearly over the course of the 500 steps. We preserve a small epsilon value over the rest of the training to preserve the potential of the agent to explore higher reward states at later stages of the training. The batch size for the training after every decision was set to 32 randomly sampled memories from the replay buffer.

The agent at every step observes the current state of the cluster and chooses between 3 actions: Increment the cluster size by 1, decrement the cluster size by 1 or do nothing. In the cases where the size of the cluster changed, the agent periodically observed the state of the cluster as described by the variable *status.cassandraOperatorProgress.* When the value of this variable was set from "Updating" to "Ready" the agent waited for 5 minutes and then collected the metrics from Prometheus to perform the next action. When the cluster remained unchanged, the agent waited for 2 minutes and then collected the metrics to perform the next action. This fact showcases the difficulty in creating an initial training set. In order to collect, for instance 300 experiences for a small initial training set the agent requires around 15 hours, whereas for 3300 experiences it requires around 165 hours or 7 days where the cluster is stressed with the training load affecting its normal operation (see Table I).

For the network architecture of the offline agent, we again used a fully connected network with 2 hidden layers. Given the fact that we had access to an offline dataset and training can be performed in a matter of minutes or hours, we had greater freedom to tune the hyperparameters of the model. For every checkpoint that we compare our agents, we use a different number of nodes for the hidden layers to optimize the performance of the model. Also, the hyperparameter a of the CQL loss function ranges from 5 for the smallest dataset to 1 for the largest dataset.

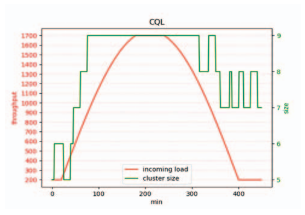
Finally, we utilized an additional optimization that is described in literature as *initial value offset* [18] Although the effectiveness of this method is not proven theoretically, empirical results show that in certain cases it speeds up convergence dramatically. The intuition behind this method is that the output layer of the algorithm in Deep Q Learning tries to estimate the numeric value of the Q function for a pair of state-action. When the nature of the reward function is such that it is significantly offset from 0, then initializing the biases of the output layer to 0 as per usual, can be problematic. The reason is that the agent will spend a great amount of training time to increment the biases to reach the order of magnitude of the Q function, given that the Q values are discounted sums of the observed reward values, before it can start to effectively learn the dynamics of the problem. To overcome this delay in learning we can initialize biases with an estimation of the mean value of the overall gain of the agent E[G] based on some initial statistics. Our experiments have shown that this bias initialization indeed helps the training to take off sooner than the zero initialization approach.

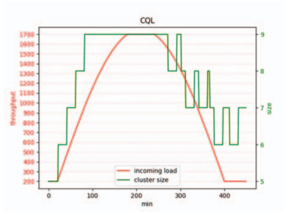
We now present the comparative performance of the online (i.e., Double Deep Q-Learning DDQN) and our system (RBS-CQL) at specific training checkpoints.

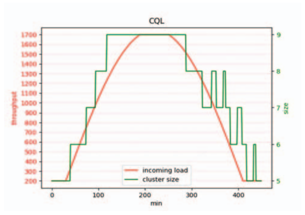
In Figures 1a and 1e we compare the two approaches using a minimal initial dataset of 300 experiences. Observing the behavior of the two models it is apparent that the RBS-CQL agent can already extract some knowledge from the minimal dataset of 300 observations about the dynamics of the problem (Fig. 1a). On the contrary, the DDQN agent has only learned that higher cluster sizes can potentially lead to higher rewards (Fig. 1e).

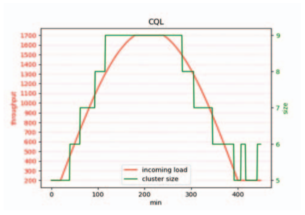
Next, in Figures 1b, 1f, 1c and 1g we compare the performance of the two models after the DDQN agent has completed the annealing steps, meaning the part of the training that the agent performs mainly exploratory actions. Again, in Figs. 1b and 1c we can observe that RBS-CQL is able to scale the cluster more drastically. On the other hand, The DDQN agent is still biased toward higher cluster size states (Figs. 1f and 1g).

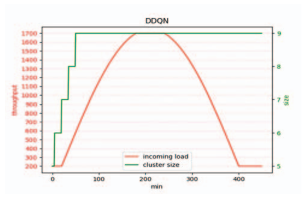
Increasing the size of the dataset from that point on-wards has shown no significant improvement for RBS-CQL and as a result we terminate the training at this point (see Figs 1d and 1h). DDQN continues to improve as it interacts with the environment but the progress slows down greatly.

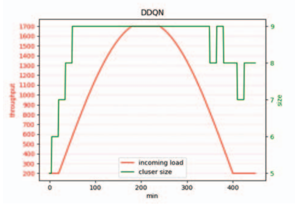

(a) RBS-CQL, minimal dataset

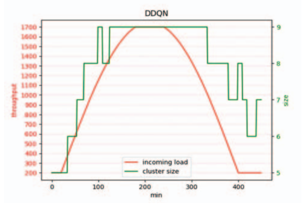

(b) RBS-CQL, small dataset


(c) RBS-CQL, medium dataset


(d) RBS-CQL, final dataset


(e) DDQN, minimal dataset


(f) DDQN, small dataset


(g) DDQN, medium dataset

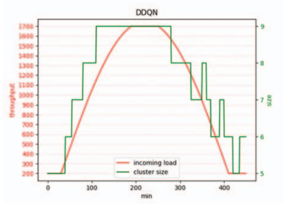
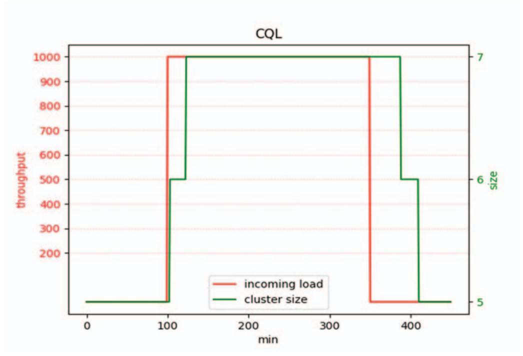
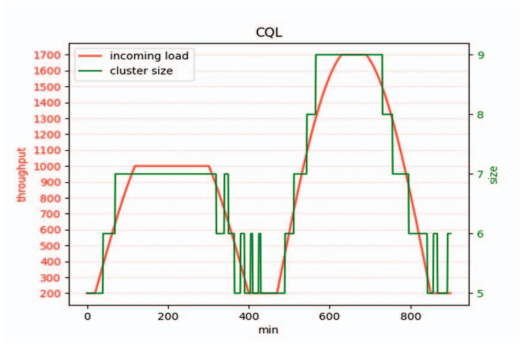

(h) DDQN, final dataset

Fig. 1: Comparison of the behavior of the online DDQN vs offline RBS-CQL for different initial experience sizes.



(a) RBS-CQL, constant unseen dataset



(b) RBS-CQL, unseen sinusoidal loads of different aptitudes

Fig. 2: RBS-CQL generalization capabilities under unseen constant and sinusoidal loads

TABLE I: **Performance comparison of RBS-CQL vs DDQN regarding total reward with various initial experience sizes.**

Dataset	DDQN reward	RBS-CQL reward	Improvement
Minimal (300 exp)	581.43	642.73	10.5%
Small (800 exp)	601.74	670.05	11.3%
Medium (1800 exp)	659.71	698.36	6.1%
Final (3300 exp)	690.42	714.62	3.5%

V. Related Work

The most common way to deal with the issue of elasticity is auto-scaling. Amazon's auto-scaling [24] for instance dynamically increases or decreases a user's resources based on thresholds applied on user cluster's specific metrics. Microsoft's Azure (Microsoft's Azure) and Celar [25] use the same technique. Yet, as shown in [22] these approaches are difficult to calibrate and optimize.

Methods from the RL domain [6] are often used for autoscaling, where multiple aspects are being tackled, from determining the correct size and type of the scaling action to solving scheduling decisions that determine the correct placement of jobs to workers. Nevertheless, none of these approaches focus on the problem of maximizing the algorithm's efficacy with minimal interaction with the environment.

The authors of [26] use a dynamic programming algorithm that tries to determine through a series of past experiences the optimal behavior for the system's nextstate. Markov Decision Processes (MDPs) and Reinforcement learning algorithms have been used in [23] to address issue, as well as an approach involving wavelets for prediction of a cloud state and resource provisioning. However, the efficiency of those approaches decreases, as the number of possible states increases. The input parameters of the system (metrics of the cluster) are continuous variables; therefore the number of discrete states can grow exponentially.

To manage this issue in [22] the authors propose an RL approach combined with decision-trees algorithms, in order to split the input parameters based on some split criteria. This approach manages to generalize over the input and to train the agent so that it can find out on its own which state parameters matter to the desired outcome and which not. Nevertheless, this approach also struggles with large space of states and also need a large dataset to show generalization capabilities.

The authors of [21] proposed a Deep Reinforcement Learning model to address the problem of elasticity in cloud native environments. The model is able to converge to a solution and provide increased rewards compared to previous approaches that did not utilize neural network. The main issue still is the fact that the number of training samples is significantly large.

The authors of [27] utilize Reinforcement learning algorithms to address the problem of adaptive auto-scaling for serverless applications. Their approach concerns the distribution of available workload to application containers and monitoring the performance of the system in terms of latency and throughput based on the concurrent requests each container has to serve. They then generate a policy using Reinforcement Learning to perform optimal distribution of workload. Their approach is similar to ours although it only applies to serverless applications. However, the states are described using only three parameters, a fact that makes the exploration space much smaller and easier to converge to a solution.

VI. Discussion

Using a containerized version of the application enhanced the training process because it allowed to accumulate an increased number of diverse experiences in the same amount of time compared to previous attempts that relied on VMs setups. The process can be accelerated further if we opt to compromise resource utilization, so more resources can be dedicated for the scaling tasks rather than executing the workloads. Nevertheless, our deployed application showed high level of resilience and was able to perform scaling operations even under severe resource pressure. This is due to the efficient scheduling algorithms of Kubernetes, that reorganize the deployed resources to ensure minimum interruptions to the deployed workloads.

The experiments with online Deep Reinforcement Learning algorithms highlighted the practical challenges that occur when applying these models to realistic scenarios. The most important is the fact that these models need constant and extensive interaction with the environment they monitor. This means that in order to perform successful training, the system must be configured meticulously, to avoid unexpected behaviors due to the agent's actions that may lead the system towards destructive states. The second challenge is the limitation in accumulated experiences. Typical Deep Reinforcement Learning application require millions of experiences and this may be unfeasible is realistic applications. Finally, due to the limitation in experiences, performing hyperparameter tuning is a very time consuming task.

The offline model we propose tackles all these challenges effectively. First of all, since the problem is essentially transformed to an unsupervised learning problem, we are able to perform hyperparameter tuning to derive the optimal model for the problem. Moreover, since the model is trained without any interaction with its environment, it is much less probable that after deployment it will lead the system to destructive states, if the problem is defined correctly. The offline agent is able to extract significantly more information from the provided dataset, compared to the online agent. As a result, it is able to converge to a solution much faster than the online equivalent. Finally, we observe that at every checkpoint of the training, the offline model is able to mitigate the bias of the online agent towards higher cluster size states, a fact that supports the claim that our agent is able to systematically derive a better decision-making policy than the one provided by the dataset.

Although limited to certain cases, our offline agent showed some capabilities of generalization over unseen workloads. Extensive generalization with Deep Reinforcement Learning still remains an unsolved issue. Nevertheless, these results are encouraging for further experimentation, especially with offline Reinforcement Learning techniques that tackle the generalization problem directly.

VII. Conclusion

In this work we experiment with contemporary deep reinforcement learning techniques to enhance the capabilities of an already powerful tool for containerized applications. We present RBS-CQL, an agent implemented as a cloud autoscaler that can effectively auto-scale complex applications in order to maximize resource utilization without compromising performance. Moreover, RBS-CQL is capable of discovering the dynamics of the monitored system even from very small datasets. We have identified, theoretically examined and implemented two different state-of-the art Deep-RL optimizations in the cloud elasticity domain that a) optimize algorithm performance in the case where experience information is hard to collect, as in the case of cloud elasticity and b) minimize errors during training, namely Conservative Q Learning and Return Based Scaling. We compare our solution with existing DeepRL techniques in a realistic cloud setting where the RL agents scale a NoSQL cluster and we show that in cases of limited initial knowledge it can offer a performance improvement of more than 10%.

VIII. Acknowledgement

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 883540 (PUZZLE).

References

[1] "Gartner Says Cloud Will Be the Centerpiece of New Digital Experiences." [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2021-11-10-gartner-says-cloud-will-be-the-centerpiece-of-new-digital-experiences

[2] N. C. Mendonça, C. Box, C. Manolache, and L. Ryan, "The Monolith Strikes Back: Why Istio Migrated From Microservices to a Monolithic Architecture," *IEEE Software*, vol. 38, no. 05, pp. 17–22, Sep. 2021, publisher: IEEE Computer Society.

[3] M. A. Tamiru, J. Tordsson, E. Elmroth, and G. Pierre, "An Experimental Evaluation of the Kubernetes Cluster Autoscaler in the Cloud," in *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Sep. 2020, pp. 17–24, iSSN: 2330-2186.

[4] V. Podolskiy, A. Jindal, and M. Gerndt, "IaaS Reactive Autoscaling Performance Challenges," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, Jul. 2018, pp. 954–957, iSSN: 2159-6190.

[5] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey," *ACM Computing Surveys*, vol. 51, no. 4, pp. 73:1–73:33, Apr. 2018.

[6] Y. Garí, D. A. Monge, E. Pacini, C. Mateos, and C. García Garino, "Reinforcement learning-based application Autoscaling in the Cloud: A survey," *Engineering Applications of Artificial Intelligence*, vol. 102, p. 104288, Jun. 2021.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, number: 7540 Publisher: Nature Publishing Group.

[8] N. Herbst, A. Bauer, S. Kounev, G. Oikonomou, E. V. Eyk, G. Kousiouris, A. Evangelinou, R. Krebs, T. Brecht, C. L. Abad, and A. Iosup, "Quantifying Cloud Performance and Dependability: Taxonomy, Metric Design, and Emerging Challenges," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 3, no. 4, pp. 19:1–19:36, May 2018.

[9] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[10] A. Kumar, A. Zhou, G. Tucker, and S. Levine, "Conservative q-learning for offline reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1179–1191, 2020.

[11] J. Kuhlenkamp, M. Klems, and O. Röss, "Benchmarking scalability and elasticity of distributed database systems," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1219–1230, May 2014.

[12] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[13] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[14] R. Bellman, "A Markovian Decision Process," *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679–684, 1957, publisher: Indiana University Mathematics Department.

[15] ——, "Dynamic Programming," *Science*, vol. 153, no. 3731, pp. 34–37, Jul. 1966, publisher: American Association for the Advancement of Science.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[18] T. Schaul, G. Ostrovski, I. Kemaev, and D. Borsa, "Return-based scaling: Yet another normalisation trick for deep rl," *arXiv preprint arXiv:2105.05347*, 2021.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[20] V. Koukis, C. Venetsanopoulos, and N. Koziris, "˜ okeanos: Building a cloud, cluster by cluster," *IEEE internet computing*, vol. 17, no. 3, pp. 67–71, 2013.

[21] C. Bitsakos, I. Konstantinou, and N. Koziris, "Derp: A deep reinforcement learning cloud system for elastic resource provisioning," in *2018 IEEE international conference on cloud computing technology and science (CloudCom)*. IEEE, 2018, pp. 21–29.

[22] K. Lolos, I. Konstantinou, V. Kantere, and N. Koziris, "Elastic management of cloud applications using adaptive reinforcement learning," in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 203–212.

[23] D. Tsoumakos, I. Konstantinou, C. Boumpouka, S. Sioutas, and N. Koziris, "Automated, elastic resource provisioning for nosql clusters using tiramola," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 2013, pp. 34–41.

[24] "AWS Auto Scaling." [Online]. Available: http://docs.aws.amazon.com/autoscaling/latest/userguide/as-scale-based-on-demand.html

[25] I. Giannakopoulos, N. Papailiou, C. Mantas, I. Konstantinou, D. Tsoumakos, and N. Koziris, "Celar: automated application elasticity platform," in *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 2014, pp. 23–25.

[26] R. Taft, N. El-Sayed, M. Serafini, Y. Lu, A. Aboulnaga, M. Stonebraker, R. Mayerhofer, and F. Andrade, "P-store: An elastic database system with predictive provisioning," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 205–219.

[27] L. Schuler, S. Jamil, and N. Kühl, "Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2021, pp. 804–811.