2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 978-1-7281-9586-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/CCGrid51090.2021.00098

AI-based Resource Allocation: Reinforcement Learning for Adaptive Auto-scaling in Serverless Environments

Lucia Schuler
Karlsruhe Institute of Technology
lucia.schuler@alumni.kit.edu

Somaya Jamil
IBM Research & Development GmbH
jamilsom@de.ibm.com

Niklas Kühl
IBM Deutschland GmbH
Karlsruhe Institute of Technology
niklas.kuehl@kit.edu

***Abstract*—Serverless computing has emerged as a compelling new paradigm of cloud computing models in recent years. It promises the user services at large scale and low cost while eliminating the need for infrastructure management. On cloud provider side, flexible resource management is required to meet fluctuating demand. It can be enabled through automated provisioning and deprovisioning of resources. A common approach among both commercial and open source serverless computing platforms is workload-based auto-scaling, where a designated algorithm scales instances according to the number of incoming requests. In the recently evolving serverless framework Knative a request-based policy is proposed, where the algorithm scales resources by a configured maximum number of requests that can be processed in parallel per instance, the so-called concurrency. As we show in a baseline experiment, this predefined concurrency level can strongly influence the performance of a serverless application. However, identifying the concurrency configuration that yields the highest possible quality of service is a challenging task due to various factors, e.g. varying workload and complex infrastructure characteristics, influencing throughput and latency. While there has been considerable research into intelligent techniques for optimizing auto-scaling for virtual machine provisioning, this topic has not yet been discussed in the area of serverless computing. For this reason, we investigate the applicability of a reinforcement learning approach to request-based auto-scaling in a serverless framework. Our results show that within a limited number of iterations our proposed model learns an effective scaling policy per workload, improving the performance compared to the default auto-scaling configuration.**

***Index Terms*—serverless, auto-scaling, reinforcement learning, Knative**

I. Introduction

Driven by the advancements and proliferation of virtual machines (VMs) and container technologies, the adoption of serverless computing models has increased in recent years [1]. According to the Cloud Native Computing Foundation, serverless computing offers two main advantages to the user [2]. First, with a true and fine-grained pay-as-you-go pricing model, costs only occur when resources are actually used and not for idle VMs or containers. Second, there is no overhead for the user associated with infrastructure maintenance, such as provisioning, updating, and managing the server resources, as this is delegated to the cloud provider. This also includes flexible on-the-fly scalability which enables resources to be added or removed automatically depending on the incoming load. For providers, the auto-scaling capability provides the ability to optimize resource utilization and reduce the effort required to manage cloud-scale applications [1].

In the implementation, the scaling mechanisms differ within the serverless offerings. Some open source serverless frameworks use the resource-based Kubernetes *Horizontal Pod Autoscaler* (HPA) to drive scaling via per-instance CPU or memory utilization thresholds (e.g. Fission [3]). This, of course, makes the auto-scaling feature dependent on the fast and correct calculations of respective system components [4]. Commercially provided serverless platforms often feature workload-based scaling by providing additional resources when incoming traffic increases, e.g. AWS Lambda initializes an instance for each new request coming in until a limit is reached [5]. However, the creation of a new instance implies a certain time lag, known as *cold start*. To bypass this issue to a certain extent, a recently emerging open-source framework *Knative* supports parallel processing of up to a predefined number of concurrent requests per instance [6]. When the so-called *concurrency* is reached, *Knative Pod Autoscaler* (KPA) deploys additional pods to handle the load. Moreover, the concurrency parameter can be adjusted manually to use resources more efficiently and to adapt the auto-scaling system to individual workloads.

In the work at hand we show that, depending on the workload, different concurrency levels can influence the performance and can lead to a latency difference of up to multiple seconds. Since this can have a critical impact on the user experience in serverless computing, we propose a reinforcement learning (RL) based model to dynamically determine the optimal concurrency for an individual workload. In general, RL formalizes the idea of an agent learning effective decision-making policies through a sequence of trial-and-error interactions with its environment. Thereby, the agent evaluates the current state of the system dynamics in each iteration, and then decides on a particular action. After the action has been performed, the agent receives either positive or negative reward and consequently learns about the goodness of the respective action-state combination. As this approach does not require any prior knowledge about incoming workload and can adapt to changes at runtime, RL algorithms have been proven as valid methods in the field of VM auto-scaling techniques in research [7]. However, it has not been

978-1-7281-9586-5/21/$31.00 ©2021 IEEE
DOI 10.1109/CCGrid51090.2021.00098

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 10,2022 at 05:37:13 UTC from IEEE Xplore. Restrictions apply.

studied in a serverless environment. Therefore, we evaluate the applicability of the established RL-algorithm Q-learning to determine the concurrency level with optimized performance.

Specifically, we implement a cloud-based framework upon which two consecutive experiments are conducted. First, we perform an analysis to examine performance variations of different workload profiles under different auto-scaling configurations. We demonstrate the dependence of throughput and latency on the concurrency level and indicate the potential for improvement through adaptive scaling settings. Using these results, we enhance the framework with an intelligent RL-based logic to evaluate the ability of a self-learning algorithm for effective decision making in a serverless framework. As we show in a second experiment, our proposed model is able to learn in limited time an appropriate scaling policy without prior knowledge of the incoming workload, resulting in an increased performance compared to the framework's default auto-scaling settings.

The remainder of the work is organized as follows. Section II introduces the serverless platform Knative and the theory of Q-learning. Section III reviews related work in both serverless frameworks and cloud-based auto-scaling techniques. Section IV gives an overview of the underlying experimental setup of the work, based on which section V presents the tests on the impact of different concurrency limits. Using these findings, section VI proposes a Q-learning model to adapt the concurrency limit on-the-fly. Section VII concludes the paper with remarks on limitations and possible future work.

II. Background

To allow for a common understanding of the application domain and used techniques, we first provide an overview of the functionality of Knative and its auto-scaling feature. Further, we introduce the theoretical foundations of the Q-learning algorithm which is applied in the second experiment.

A. Knative Serverless Platform

As an open-source serverless platform, Knative provides a set of Kubernetes-based middleware components to support deploying and serving of serverless applications, including the capability to automatically scale resources on demand [6].

The auto-scaling function is implemented by different serving components, described by the request flow in Fig. 1 based on Knative v0.12. If a service revision is scaled to zero, i.e. the service deployment is reduced to a replica of null operating pods, the ingress gateway forwards incoming requests first to the activator [6]. The activator then reports the information to the autoscaler, which instructs the revision's deployment to scale-up appropriately. Further, it buffers the requests until the user pods of the revision become available, which can cause cold-start costs in terms of latency, as the requests are blocked for the corresponding time. In comparison, if a minimum of one replica is maintained active, the activator is bypassed and the traffic can flow directly to the user pod.

When the requests reach the pod, they are channeled by the *queue-proxy* container and, subsequently, processed in the *user-container*. The queue-proxy only allows a certain number of requests to enter the user-container simultaneously, and queues the requests if necessary. The amount of parallel processed requests is specified by the *concurrency* parameter configured for a particular revision. Depending on which concurrency is set in the revision, the queue-proxy will only allow a corresponding number of requests to be processed by the user-container simultaneously, queuing them if necessary.

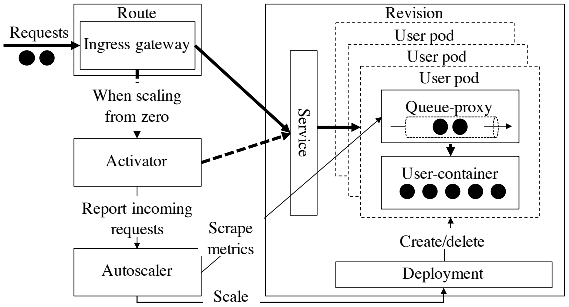

Fig. 1: Illustration of the request flow in Knative v0.12

By default, the value is set to a concurrency target of 100, defining how many parallel requests are preferred per user-container at a given time. However, the user can restrict the number of concurrent requests by specifying a value between 0 and 1000 for the *concurrency limit*.[1] Further, each queue-proxy measures the incoming load, reporting the average concurrency and requests per second on a separate port. The metrics of all queue-proxy containers are scraped by the autoscaler component, which then decides how many new pods will be added or removed to keep the desired concurrency level.

B. Q-learning

RL refers to a collection of trial-and-error methods in which an agent is trained to make good decisions by interacting with his environment and receiving positive or negative feedback in form of rewards for a respective action. A popular RL algorithm is the model-free Q-learning.

Q-learning stepwise trains an approximator $Q_\theta(s, a)$ of the optimal action-value function Q^*. $Q_\theta(s, a)$ specifies the cumulated reward the agent can expect when starting in a state s, taking an action a, and then acting according to the optimal policy forever after. By observing the actual reward in each iteration, the optimization of the Q-function is performed incrementally per step t:

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a)] \quad (1)$$

α describes the learning rate, i.e. to what extent newly observed information overrides old information and γ a discount factor that serves to balance between the current and future reward. As RL is a trial-and-error method, during training, the agent has to choose between the exploration of a new action and the exploitation of the current best option [9]. In research,

[1] A value of 0 allows unlimited concurrent requests (no scaling) [8].

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 10,2022 at 05:37:13 UTC from IEEE Xplore. Restrictions apply.

this is often implemented with an ϵ-greedy strategy, where ϵ defines the probability of exploration that usually decreases as the learning process advances [10], [11]. With a probability of $1 - \epsilon$, the agent selects based on the optimal policy and chooses the action that maximizes the expected return from starting in s, i.e. the action with the highest Q-value:

$$a^*(s) = \arg\max_a Q^*(s, a) \tag{2}$$

In the basic algorithm, the Q-values for each state-action combination are stored in a lookup table, the so-called *Q-table*, indexed by states and actions. The tabular representation of the agent's knowledge serves as a basis for decision-making during the entire learning episode.

III. Related Work

To the best of our knowledge, the applicability of RL-based technology to optimize auto-scaling capabilities in serverless environments has not been investigated. However, considering the areas of serverless and intelligent auto-scaling separately, a large body of knowledge is available, summarized in the following subsections.

A. Serverless computing

With the growing number of serverless computing offerings, there has been an increasing interest of the academic community in comparing different solutions, with scalability being one of the key elements of evaluation [4]. In multiple works, different propriety serverless platforms were benchmarked, including their ability to scale, focusing on Amazon Lambda, Microsoft Azure Functions [12], along with Google Cloud Functions [13] and IBM Cloud Functions [14]. Similar studies have been carried out in the area of open-source serverless frameworks, with greater attention paid to the auto-scaling capabilities. Mohanty et al. [15] evaluated Fission, Kubeless, and OpenFaaS and concluded that Kubeless provides the most consistent performance in terms of response time. Another comparison of both qualitative and quantitative features of Kubeless, OpenFaas, Apache Openwhisk, and Knative, comes to the same conclusion, albeit generally indicating the limited user control over custom Quality of Service requirements [16]. These studies solely consider the default auto-scaler Kubernetes HPA. Possible adjustments to the auto-scaling mechanism itself are not further examined. Li et al. [4] propose a more concrete distinction between resource-based and workload-based scaling policies. The authors compare the performance of different workload scenarios using the tuning capability of concurrency levels in Knative and clearly suggest further investigation of the applicability of this auto-scaling capability, which further motivates this research.

B. Auto-scaling

As elasticity is one of the main characteristics of the increasing adaption of cloud computing, the automatic, on-demand provisioning of cloud resources have been the subject of intensive research in recent years [17]. We discuss related work under two aspects: first, the underlying theories on which auto-scaling is built with a focus on RL, and second, the entities being scaled.

To classify numerous techniques at the algorithmic level, different taxonomies were proposed, where the predominant categories are threshold-based rules, queuing theory and RL [7], [17]. In the former, scaling decisions are made on predefined thresholds and are most popular among public cloud providers, e.g. Amazon ECS [18]. Despite the simplistic implementation, identifying suitable thresholds requires expert knowledge [17], or explicit application understanding [19]. Queuing theory has been used to mathematically model applications [7]. As they usually impose a stationary system, the models are less reactive towards changes [7].

In contrast, RL offers an interesting approach through online learning of the most suitable scaling action and without the need for any a-priori knowledge [7]. Many authors have therefore investigated the applicability of model-free RL algorithms, such as Q-learning, in recent years [10]. Dutreihl et al. [19] show that although Q-learning based VM controlling requires an extensive learning phase and adequate system integration, it can lead to significant performance improvements compared to threshold-based auto-scaling, since thresholds are often set too tightly while seeking for the optimal resource allocation. To combine the advantages of both, Q-learning itself can be used to automatically adapt thresholds to a specific application [20].

In terms of the entity being scaled, RL has been mostly applied to policies for VM allocation, e.g. in [21]. With the emergence of container-based applications, this field has become a greater focus of research [10]. In both areas, the scope of action is concentrated mainly on horizontal (scale-out/-in) [20], vertical scaling (scale-up/-down) [22], or the combination of both [10]. However, little research has been done in areas that extend the classic auto-scaling problem of VM or container configuration.

As a novel approach we investigate the applicability of Q-learning to request-based auto-scaling in a serverless environment. Differently from the existing work on direct vertical or horizontal scaling with RL, we propose a model that learns an effective scaling policy by adapting the level of concurrent requests per container instance to a specific workload.

IV. Approach

To investigate different concurrency configurations, a flexible Kubernetes-based framework is designed which can be extended by an intelligent RL-based logic. In this section, we present the experimental setup including the cloud architecture, specification of the utilized workload and standard process flow. This section provides the foundation for both the first experiment assessing the impact of concurrency changes and the second experiment evaluating RL-based auto-scaling.

A. Cloud Architecture

The overall architecture of our experiment is illustrated in Fig. 2. To test the auto-scaling capabilities in an isolated environment, we set up two separate Kubernetes clusters, using

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 10,2022 at 05:37:13 UTC from IEEE Xplore. Restrictions apply.

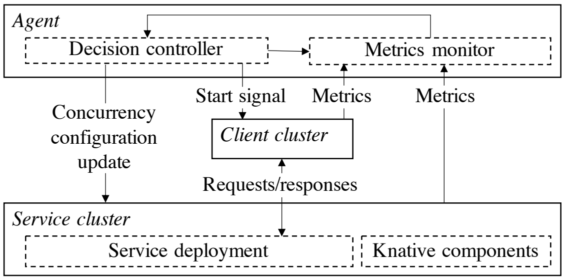

Fig. 2: Architectural setup including the information flow between the three components agent, client, and service cluster

IBM Cloud Kubernetes Service (IKS). On the *service cluster*, the sample service used for the experiments is deployed. The cluster is designed to provide sufficient capacity to host all Knative components and avoid performance limitations (9 nodes, 16 vCPU, 64 GB memory). The *client cluster* is responsible for sending requests to the service cluster to generate load (one node, 16 vCPU, 64 GB memory). The *agent* manages the activities on both clusters, including the configuration updates of the sample service based on collected metrics, and coordinates the process flow of the experiment, taking the role of an IKS user.

The Knative resources are installed on the service cluster (version v0.12), including the serving components explained in Section II-A, which control the state of the deployed sample service and enable auto-scaling of additional pods on demand. Using the trial-and-error method of RL in the second experiment, we update the concurrency configuration of the service in each iteration, creating a new revision each time. Incoming requests are routed by default to the most recent revision with the newest concurrency update.

To comprehensively test the auto-scaling capability, we activated the scale-to-zero functionality in the autoscaler's configmap, which requires a cold start in each iteration. We further increased the replica number of ingress gateways, which handle load balancing, to bypass performance issues and to focus exclusively on the auto-scaling functionalities.

B. Workload

Serverless computing is used for a variety of applications, accompanied by different resource requirements. For example, the processing of video material or highly-parallel analytical workloads, demand considerable memory and computing power. Other applications, such as chained API compositions or chatbots, tend to be less compute-intensive but may require longer execution or response time.

To investigate the concurrency impact of many different workloads, we generate a synthetic, stable workload profile simulating serverless applications. We use Knative's example *Autoscale-go* application for this purpose, which allows different parameters to be passed with the request to test incremental variations of the workload characteristics and thus emulate varying CPU- and memory- intensive workloads [23].

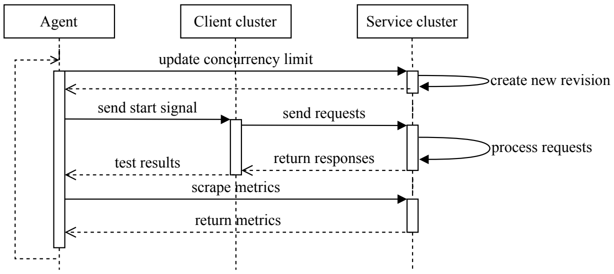

Fig. 3: Process flow of one test iteration

The three application parameters are *bloat*, *prime* and *sleep*, wherein the first is used to specify the number of megabytes to be allocated and the second to calculate the prime numbers up to the given number, to create either memory- or compute-intensive loads. The sleep parameter pauses the request for the corresponding number of milliseconds, as in applications with certain waiting times.

C. Process Flow

The basic process flow of one iteration is illustrated in Fig. 3. In each iteration the agent sends a concurrency update to the service cluster, which accordingly creates a new revision with the respective concurrency limit. When the service update is complete, the agent sends the start signal to the client cluster, which begins issuing parallel requests against the service cluster. To simulate a large number of user requests at the same time, we use the HTTP load testing tool *Vegeta*, which features sending HTTP requests at a constant rate. In the experiment, 500 requests are sent simultaneously over a period of 30s to ensure sufficient demand for scaling and sufficient time to provide additional instances. After the last response is received, Vegeta outputs a report of the test results, including information on latency distribution of requests, average throughput and success ratio of responses. The performance measures are then stored by the agent. Additionally, the agent crawls metrics from the Knative monitoring components, exposed via a Prometheus-based HTTP API within the cluster, to get further information about resource usage at cluster, node, pod and container level. Using this data, the concurrency update is chosen to proceed to the next iteration.

V. Baseline Experiment

To determine the implications of varying concurrency limits, we first conduct a baseline experiment comparing different workloads on their relative performance.

A. Design

As outlined in the previous section, we use the application parameters *bloat*, *prime* and *sleep* to simulate varying workload characteristics. Starting with a no-operation workload where no parameters are passed, the memory allocation and CPU load were gradually increased for each new experiment. The step size of the memory allocating parameter was aligned with the memory buckets commonly used for the standard

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 10,2022 at 05:37:13 UTC from IEEE Xplore. Restrictions apply.

pricing model of serverless platforms. To simulate compute-intensive and longer-lasting requests, different prime and sleep parameters were chosen correspondingly. The detailed values are specified in Table I.

Per profile, we run performance tests for varying concurrency levels according to the process flow described in Fig. 3. Theoretically, the concurrency limit can take any value between 0 and 1000. To keep the experiments computationally feasible, we proceed in steps of 20, starting at a concurrency limit of 10 and ending at 310. As stated in related literature, we focus on latency and throughput as key performance measures of serverless applications [4]. Average throughput is defined by requests per second (RPS), mean latency refers to the average time in seconds taken to return the request response. To cover tail latency, we include the 95th percentile latency as an additional metric. Furthermore, each test is repeated ten times to compensate for outliers or other fluctuations, before the concurrency is updated to the next limit.

B. Results

We structure the analysis of the baseline experiment results in three parts. First, we examine the behavior of the individual workload profiles under different concurrency configurations. Second, we focus on the relation of the target variables throughput and latency. Finally, we analyze further metrics about resource utilization on container and pod level.

As described above, we conducted the experiment for different combinations of the three parameters to simulate possible use cases. Table I gives an overview of the outcomes with the concurrency limit that lead to the optimal test result in terms of one of the performance measures. Due to the numerous uncontrollable factors that influence the performance of the cluster, each result forms a snapshot in time. The respective workload configuration is described by the three columns on the left. Taking all tests into account, the smallest possible concurrency of 10 is the most common configuration that resulted in the best performance across all three indicators. Interestingly, this does not correspond to the default setting of the KPA where a target concurrency value of 100 is preferred [24]. In particular, workloads that consume memory exclusively perform better with fewer parallel requests per pod instance, e.g tests #II, #IX, #XVI and #XVII. Similar observations are made for workloads with additional low CPU usage, i.e. lower *prime* parameter, as in tests #III and #X. Deviations can be observed when the requests pause for a certain time. These workloads result in higher throughput and lower mean and tail latency when a higher concurrency is chosen, e.g tests #VII, #VIII and #XIV.

Depending on the workload, the distance between the optimal configuration and the second best concurrency can be very small, which becomes more evident when analyzing a single test in detail. Fig. 4 shows the result of test #VII, which is examined representatively. Although the individual measurement points fluctuate, clear trends are identified in the average values. A significant increase in throughput can be observed when the concurrency limit is raised to 70. This setting also yields the lowest value for mean latency, differing from the second-best value at concurrency 50 by only 80 milliseconds. The distance becomes more critical when considering the tail latency of the 95^{th} percentile, where a request takes more than 740 milliseconds on average longer to receive a response when compared to the most effective configuration. At a concurrency of 10, the difference amounts to almost 3 seconds, further underlining the performance variations caused by the different settings. The greatest slowdown in tail latency in this test occurs at a concurrency of 310 with more than 3.7 seconds.

TABLE I: Concurrency Performance Tests

Test	Workload Profile			Conc. Limit Yielding Best Perf.		
#	*bloat**	*prime*	*sleep**	*thrghpt*	*mean lat.*	*95th lat.*
I	-	-	-	50	50	70
II	128	-	-	30	30	30
III	128	1000	-	10	10	10
IV	128	10.000	-	30	30	10
V	128	100.000	-	10	10	10
VI	128	1000	1000	110	110	150
VII	128	10.000	1000	70	70	70
VIII	128	100.000	1000	110	110	110
IX	256	-	-	10	10	10
X	256	1000	-	10	10	10
XI	256	10.000	-	10	10	10
XII	256	100.000	-	10	10	10
XIII	256	1000	1000	50	50	50
XIV	256	10.000	1000	110	110	110
XV	256	100.000	1000	10	10	30
XVI	512	-	-	10	10	10
XVII	1024	-	-	30	30	30

* *bloat* is defined in MB and *sleep* in milliseconds.

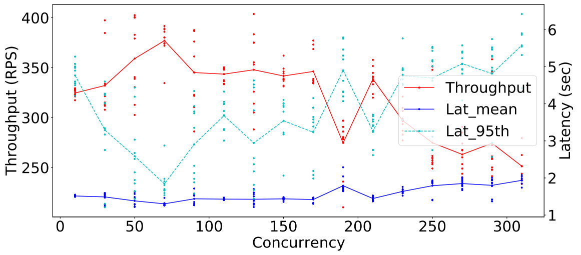

Fig. 4: Performance in baseline experiment (workload #VII)

Besides, the overall performance decreases strongly when the concurrency limit exceeds a level of 210. This tendency can be found across the majority of tests, indicating that due to the high simultaneous processing of many requests, only a limited amount of resources are available for a single request. Further observations show that with increasing memory utilization, i.e. the bloat parameter, performance tends to drop at lower concurrency limits. In some cases, additionally, the success ratio strongly declines. For example in test #XVI, from a concurrency of 170 onwards, more than 10% of the requests received non-successful responses. In test #XVII accordingly, this output can be observed from a concurrency of 90 onwards.

Focusing on the target metrics, the tests show that adjusting the concurrency limit to an appropriate setting can yield significant improvements in throughput and latency. Furthermore, an inverse behavior of the measures can be observed within the

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 10,2022 at 05:37:13 UTC from IEEE Xplore. Restrictions apply.

tests. For the previously considered test #VII, the results indicate a significant negative correlation of throughput and mean latency of -0.989, and a similar correlation for throughput and 95^{th}-percentile latency of -0.916.[2] This strong negative relationship between the metrics is found across all tests, with significant correlation coefficients ranging from -0.995 to -0.748.[3] Subsequently, an improvement in throughput usually results in a lower and more favorable latency. This finding implies that there is no need to make trade-offs between different target metrics when adjusting the concurrency. Instead, the problem can be reduced to one objective metric, representing the others.

VI. Reinforcement Learning Experiment

The experiment described in the previous section demonstrates the impact the concurrency configuration can have on performance. Therefore, we evaluate the applicability of the model-free RL algorithm Q-learning in a second experiment to learn effective scaling policies by adjusting the concurrency limit during run time.

A. Design

The process flow is based on the procedure from section IV-C, extended with a more sophisticated logic of the agent. Instead of incrementally increasing the concurrency, the agent uses knowledge of the system environment (states) to test different concurrency updates (actions) and evaluates them by receiving scores (reward).

In each iteration, the environment is defined by the current state, which, should provide a complete description of the system dynamics including all relevant information for optimal decision making. Due to the large number of factors influencing performance, e.g. hidden cluster activities or network utilization, this is neither traceable nor computationally feasible in the used Q-learning algorithm. Therefore, we break down our state space S into three key features. We define S at time step i as the combination of the state variables $s_i = (conc_i, cpu_i, mem_i)$, where $conc_i$ depicts the concurrency limit, cpu_i is the average CPU utilization per user-container and mem_i is the average memory utilization per user-container. The selection of the features is aligned with related research, with $conc_i$ as the equivalent of the number of VMs in VM auto-scaling approaches [21], [25]. Further, cpu_i and mem_i serve as a direct source of information of the resource utilization of a respective workload and are therefore used to describe the current system state.

Since both CPU and memory utilization are continuous numbers, we discretize them into bins of equal size. In each state $s_i \in S$, we define $A(s_i)$ as the set of valid actions, where A is the set of all actions. The agent can choose between decreasing, maintaining or increasing the concurrency limit by 20, i.e. $A = \{-20, 0, 20\}$. If the agent reaches the endpoints on the concurrency scale, i.e. the minimum or maximum concurrency, the action space in this state is reduced accordingly by the non-executable action.

After each iteration, the agent receives an immediate reward according to the performance achieved through the action. In related literature, the reward is often based on the distance or ratio between the performance measure and a certain *Service Level Agreement*, such as a throughput or response time target value [20], [26]. Since there is no target level to be achieved nor prior information about the performance given in our problem definition, we define an artificial reference value ref_value as the best value obtained to date. Due to the permanent, albeit minor fluctuations in the measures, we propose a tolerance band around the reference value to avoid weighting minor non-relevant deviations. Furthermore, the results from the preliminary study have shown a highly negative correlation between throughput and latency, i.e. higher throughput usually leads to lower and therefore better latency. This relation in turn allows to focus exclusively on throughput ($thrghpt$) as one single objective. The calculation of the reward r in time step i is as follows.

$$r_i = \begin{cases} \frac{thrghpt_i}{ref\_value} & \text{if } thrghpt_i \leq \text{ref\_value} \cdot 0.95 \\ & \text{or } thrghpt_i \geq \text{ref\_value} \cdot 1.05 \\ 1 & \text{else} \end{cases}$$

Q-learning is initiated with the following parameters. A learning rate α = 0.5 is chosen to balance newly acquired and existing information, a discount factor γ = 0.9 to ensure that the agent strives for a long-term high return. To encourage the exploration of actions at the beginning of training, we implement a decaying ϵ-greedy policy starting at iteration 50 with ϵ = 1 and then slowly decrease over time by a decay factor of 0.995 per iteration. The minimum exploration probability is set to ϵ_{min} = 0.1, to allow for the detection of possible changes in the system. The knowledge the agent acquires is stored in a Q-table and updated each iteration.

To examine whether the model can effectively learn the concurrency values identified in section V as high throughput configurations, the results are analyzed representatively based on workload test #VII and #X. The former test showed high performance at a concurrency limit of 70, while the second reached the best test results at an edge concurrency of 10.

B. Results

First, we analyze the results to examine the suitability of the proposed Q-learning-based model to fine-tune the auto-scaling. Second, we evaluate the performance of the approach in terms of throughput improvements compared to Knative's default auto-scaling configuration.

Based on workload profile #X, Fig. 5 shows how the agent applies RL logic to incrementally change the concurrency and to adjust it as the training progressed. Beginning at concurrency 170, random exploration leads to a moderate decline of the concurrency limit in the first 30 iterations. This results in an improvement in throughput, captured by

[2] For all statistical tests, Pearson correlation coefficient is used with a two-sided p-value for testing non-correlation and an alpha level of .001.

[3] Except for test #I and #XIII with significant correlations of throughput and tail latency of -0.629, and throughput and mean latency of -0.677.

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 10,2022 at 05:37:13 UTC from IEEE Xplore. Restrictions apply.

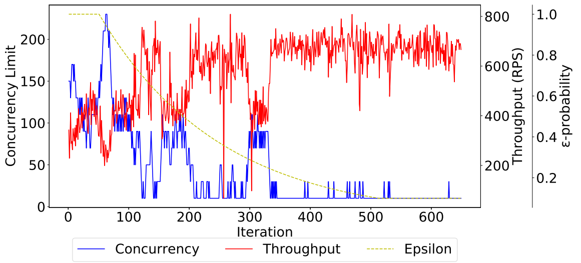

Fig. 5: Performance of Q-learning model (workload #X)

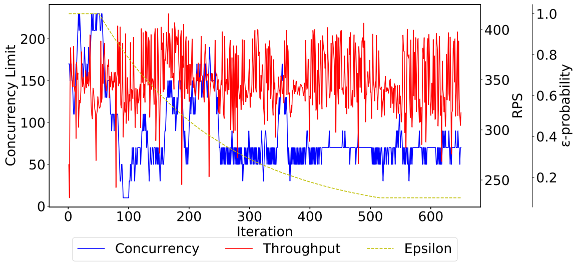

Fig. 6: Performance of Q-learning model (workload #VII)

the rewards and corresponding Q-values for each state-action combination. The most effective scaling policy of 10 parallel requests per container is first reached in iteration 121. Nevertheless, due to the ϵ-greedy strategy, exploratory actions are chosen, which might differ from the down-scaling decision and cause the agent to deviate from a good strategy. As training progresses, a trend towards performance-enhancing concurrency configurations can be observed, indicating the agent is more likely to exploit the optimal decision rather than exploring. After 330 iterations, the concurrency stabilizes at a limit of 10 parallel requests per container, implying the agent has learned the correct scaling policy, according to the results from section V. Due to the minimum $\epsilon = 0.1$, exploration still rarely occurs to ensure the agent can respond to changes in the environment.

A different learning process of the proposed Q-learning approach can be observed for workload #VII, depicted in Fig. 6. The varying concurrency curve shows the initial strategy of the agent exploring first the higher state space before proceeding with lower concurrency limits. After 250 iterations the exploitation phase outweighs and the concurrency gradually levels off. In comparison to workload #X, where the algorithm's scaling policy converges to a single concurrency limit, the configuration here fluctuates, mainly between 50 and 70, and retains this pattern. Further differences between the test results arise from the throughput metric, which shows strong fluctuations between 230 and 415 RPS across all iterations. The deviations, which also appear within one concurrency setting, considerably impair the agent's ability to evaluate suitable state-action-pairs via the reward function. Nevertheless, the agent is able to narrow down the scaling range to a limited number of values at which it identified the

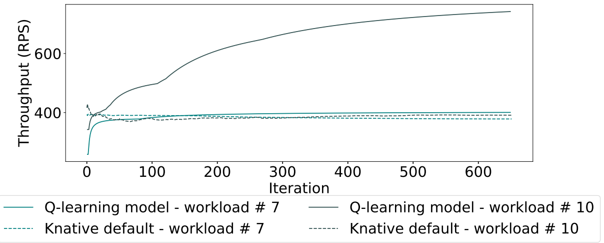

Fig. 7: Comparison of average throughput of the Q-learning model and the Knative default auto-scaling setting

best outcomes in terms of throughput, and which agrees with the result of the baseline experiment in Section V.

To evaluate the proposed scaling policies, we benchmark the average performance of the Q-learning-based approach with the static default setting. For this purpose, the same experimental setup is used as in the Q-learning test, except for the auto-scaling configuration, where the original setting of a concurrency target of 100 is applied [6].[4] Fig. 7 depicts the average throughput up to the respective iteration of the Q-learning model and the default configuration for the two considered workloads. Both result in the Q-learning model outperforming the test based on Knative's standard settings. Considering workload #VII first, the model requires approximately 150 iterations until the average performance reaches default-level. Subsequently, the throughput increases to an average of 400 RPS providing a minor advantage of 20 RPS compared to the standard system. A more significant enhancement shows workload #X. While in the first 10 iteration the default settings alternate between 350 and 440 RPS, the performance of our model is initially lower. However, with ongoing learning the average throughput improves and excels already from iteration 10 onwards. After 600 iterations, the presented Q-learning based model reaches an average throughput of 740 RPS, hence achieving more than 80% of the performance of the default setting, which stabilizes at 390 RPS on average.

To summarize the results, the proposed model learned within finite time a scaling policy that outperforms the default Knative configuration in terms of throughput, proving the Q-learning-based approach is well-feasible to refine the auto-scaling mechanism.

VII. Conclusion

With the emergence of serverless frameworks, the ability of dynamic, real-time resource provisioning to meet varying demand has become a key area of interest and has led to the development of numerous scaling mechanisms. Focusing on request-based scaling, we first investigated the impact modifying the main scaling parameter, i.e. the number of concurrent requests per instance, may have on performance. The experiments showed deviations of up to multiple seconds in the average latency as well as significant differences in throughput,

[4] Additionally, the container target percentage is set to 0.7 as in the default configmap.

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 10,2022 at 05:37:13 UTC from IEEE Xplore. Restrictions apply.

thus indicating that the concurrency configuration can affect the performance depending on the workload. To flexibly adjust the auto-scaling settings to specific requirements, we designed a RL model based on Q-learning and evaluated its applicability to learn effective scaling policies during runtime. Based on different workloads, we showed that the proposed model can adapt the concurrency appropriately without prior knowledge within limited time and outperforms the average throughput compared to the default setting of Knative.

Given these results, the presented work offers valuable contributions to both the existing work in the field of serverless frameworks and the application of RL-based auto-scaling.

- In addition to previous studies on scaling capabilities in serverless platforms, we provided a detailed analysis to reveal the performance implications of changes in the concurrency configuration.
- Furthermore, we demonstrated with our proposed model the applicability of Q-learning-based auto-scaling in the field of serverless applications.
- Additionally, the findings contribute to the ongoing development of the auto-scaling system of the Knative community project.

Nevertheless, we identified some limitations in the approach during the experiments. First, the results from Section V are based on synthetic workloads simulated by one application with varying parameters, and thus cannot be interpreted as a universally valid conclusion on the effects of real-world applications. Second, due to the focus on general applicability of Q-learning, the approach uses a rather simplistic reward function measuring exclusively the proximity to the reference value. Further refinement of the reward function may improve the efficiency of the proposed model. Similarly, the description of the system state of the RL environment could be extended by additional parameters such as memory allocation and time constraints to improve the model's accuracy.

While in this work a RL approach has been developed, which learns a certain scaling policy per workload mainly through testing different concurrency states, it remains to be analyzed to what extent the ratio of resource usage of individual components might impact the performance. Thus, a comprehensive study could be conducted to determine the combination of utilization levels that might achieve the best possible performance across all workloads. Consequently, the concurrency configuration could merely serve as a tool to bring the system into this particular state.

References

[1] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The server is dead, long live the server: Rise of serverless computing, overview of current state and future trends in research and industry," *arXiv preprint arXiv:1906.02888*, 2019.

[2] S. Allen, C. Aniszczyk, C. Arimura, and et al., "Cncf serverless whitepaper v1.0," 2018. [Online]. Available: https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf

[3] "A high-level view of the internals of fission." https://github.com/fission/fission/blob/master/Documentation/Architecture.md, accessed: 2020-03-26.

[4] J. Li, S. G. Kulkarni, K. Ramakrishnan, and D. Li, "Understanding open source serverless platforms: Design considerations and performance," in *Proceedings of the 5th International Workshop on Serverless Computing*, 2019, pp. 37–42.

[5] "Aws lambda function scaling," https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html, accessed: 2020-03-26.

[6] "Knative serving autoscaling system," https://github.com/knative/serving/blob/master/docs/scaling/SYSTEM.md, accessed: 2020-03-26.

[7] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.

[8] "Configuring knative serving autoscaling," https://docs.openshift.com/container-platform/4.2/serverless/configuring-knative-serving-autoscaling.html, accessed: 2020-03-28.

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

[10] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD).* IEEE, 2019, pp. 329–338.

[11] 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.

[12] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E).* IEEE, 2018, pp. 159–169.

[13] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 133–146.

[14] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD).* IEEE, 2018, pp. 442–450.

[15] S. K. Mohanty, G. Premsankar, M. Di Francesco *et al.*, "An evaluation of open source serverless computing frameworks." in *CloudCom*, 2018, pp. 115–120.

[16] A. Palade, A. Kazmi, and S. Clarke, "An evaluation of open source serverless computing frameworks support at the edge," in *2019 IEEE World Congress on Services (SERVICES)*, vol. 2642. IEEE, 2019, pp. 206–211.

[17] P. Singh, P. Gupta, K. Jyoti, and A. Nayyar, "Research on auto-scaling of web applications in cloud: survey, trends and future directions," *Scalable Computing: Practice and Experience*, vol. 20, no. 2, pp. 399–432, 2019.

[18] "Aws service auto scaling," https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-auto-scaling.html, accessed: 2020-03-27.

[19] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck, "From data center resource allocation to control theory and back," in *2010 IEEE 3rd international conference on cloud computing.* IEEE, 2010, pp. 410–417.

[20] S. Horovitz and Y. Arian, "Efficient cloud auto-scaling with sla objective using q-learning," in *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud).* IEEE, 2018, pp. 85–92.

[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] J. Rao, X. Bu, C.-Z. Xu, and K. Wang, "A distributed self-learning approach for elastic provisioning of virtualized cloud resources," in *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems.* IEEE, 2011, pp. 45–54.

[23] "Knative autoscale-go sample app - go," https://github.com/knative/docs/tree/master/docs/serving/samples/autoscale-go, accessed: 2020-04-04.

[24] "Configuring autoscaling," https://knative.dev/v0.12-docs/serving/configuring-autoscaling//, accessed: 2020-04-28.

[25] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656–1674, 2013.

[26] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow," in *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*, 2011, pp. 67–74.

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on November 10,2022 at 05:37:13 UTC from IEEE Xplore. Restrictions apply.