CAUS: An Elasticity Controller for a Containerized Microservice

Floriment Klinaku
University of Stuttgart
Universitätsstraße 38, Germany
floriment.klinaku@informatik.uni-stuttgart.de

Markus Frank
University of Stuttgart
Universitätsstraße 38, Germany
markus.frank@informatik.uni-stuttgart.de

Steffen Becker
University of Stuttgart
Universitätsstraße 38, Germany
steffen.becker@informatik.uni-stuttgart.de

ABSTRACT

Recent trends towards microservice architectures and containers as a deployment unit raise the need for novel adaptation processes to enable elasticity for containerized microservices. Microservices facing unpredictable workloads need to react fast and match the supply as closely as possible to the demand in order to guarantee quality objectives and to keep costs at a minimum. Current state-of-the-art approaches, that react on conditions which reflect the need to scale, are either slow or lack precision in supplying the demand with the adequate capacity. Therefore, we propose a novel heuristic adaptation process which enables elasticity for a particular containerized microservice. The proposed method consists of two mechanisms that complement each other. One part reacts to changes in load intensity by scaling container instances depending on their processing capability. The other mechanism manages additional containers as a buffer to handle unpredictable workload changes. We evaluate the proposed adaptation process and discuss its effectiveness and feasibility in controlling autonomously the number of replicated containers.

KEYWORDS

cloud computing, elasticity, containers

ACM Reference Format:
Floriment Klinaku, Markus Frank, and Steffen Becker. 2018. CAUS: An Elasticity Controller for a Containerized Microservice. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3185768.3186296

1 INTRODUCTION

One of the most important characteristics of cloud computing is elasticity. Herbst et al. [4] define elasticity as "the degree" to which a system is able to adapt to changes in demand by provisioning or releasing resources autonomously. The recently adopted stack on the cloud with microservices, containers, and the orchestration of containers across multiple hosts with middlewares such as Kubernetes[1], creates a separation in concerns related to elasticity. The platform handles the scaling of nodes whenever scheduling of containers is no longer possible due to the non-availability of resources. For microservice owners, it enables a view on resources as an infinity pool and gives the possibility to scale out and in the processing containers depending on the demand for their functionality. To keep costs to the minimum and quality objectives as promised in their SLOs an adaptation process should exist which alters the number of container instances based on the demand. There exists a proposed and elaborated set of metrics that quantify and capture the quality of elasticity for systems [4, 7]. So, a theoretically perfect elastic microservice should adapt to changes in demand with instantaneous actions (speed) which provide only the necessary resources, neither more nor less (precision), to fulfill the current demand; the timeshare at which the microservice has more or less resources than needed should be as close as possible to zero percent.

In cases where demand can be predicted for coarse-grained units of days and hours, adaptation processes can spin up container instances beforehand. However, this is insufficient for microservices which are exposed to Internet workloads where unpredictable bursts in load intensity can occur. It is necessary to complement proactive methods with adaptation processes that reactively—based on conditions that reflect the need to scale—adapt the number of containerized microservices. Complementing predictive techniques with current state-of-the-art approaches on autoscaling containers reactively leads to several problems. Current adaptation processes, designed for VMs and monoliths, do not exploit the new conditions: the increased speed to instantiate containers and the constricted responsibility for a microservice which increases the accuracy of the estimated capacity. All considered approaches are either slow in matching the demand with the adequate capacity, or they are fast but they overprovision resources. Furthermore, it requires a considerable knowledge for designing scaling rules and determining scaling criteria.

Therefore, we propose a novel heuristic adaptation process which enables elasticity for a particular containerized microservice. We consider the context where a stateless microservice is facing queue-based workload: each container instance is attached as a listener to the same request queue where workload—in form of requests—arrives for processing. Predictive approaches can be used to obtain models for the arrival rate of requests so that provisioning and releasing actions take place beforehand based on the capacity that a single container promises. To complement predictive approaches, we introduce a hybrid method that consists of two parts. One part reacts to changes in load intensity by scaling container instances depending on their processing capability. The other mechanism manages additional containers as a buffer to handle unpredictable load changes. The aim is to provide an alongside mechanism that

[1] https://kubernetes.io/

complements proactive techniques that produce the base resource supply for coarse-grained time units.

This paper summarizes the proposal, results, and observations conducted in a master thesis [6]. It explores current alternatives and highlights their problems in the *State of the Art* section. Then, in the *Adaptation Process* section, it proposes a new adaptation process for controlling the elasticity of microservices which are in the same context as described—a microservice which is listening for requests in a particular queue and processes them with a particular logic. In *Evaluation*, we evaluate the method by measuring the achieved elasticity. Finally, we conclude the work and present future steps in the *Conclusion and Future Work* section.

2 STATE OF THE ART

Recently Al-Dhuraibi et al. [2] conducted extensive research on presenting and classifying state-of-the-art approaches for exploiting elasticity in general for cloud computing. Reactive approaches make use of a variety of techniques (queueing theory, control theory, time series analysis etc.) to provide resources autonomously when they are needed and with the right amount. We present two widely used representatives of such approaches and show their shortcomings when applied to containers. We choose Kubernetes Horizontal Pod Autoscaler[2] and Amazon EC2 Container Service Autoscaling[3] as they represent the current state-of-the-art middlewares available in controlling autonomously the number of replicated containers.

Kubernetes, as mentioned earlier, is a middleware that orchestrates containers accross multiple hosts. They offer a variety of features which enable users to run workloads as containers in a cluster of machines. With regard to scaling containers autonomously they offer a default elasticity controller called the Horizontal Pod Autoscaler (HPA). As the name suggests, the HPA enables elasticity for a microservice compound of container instances deployed to the cluster with the pod abstraction (we assume a one-to-one mapping between the pod and the container). When considering Kubernetes HPA, the owner of a microservice can select between two different feedback signals: the desired CPU utilization among processing containers or the length of the request queue.

The use of CPU utilization as the criterium to base scaling decisions in Kubernetes HPA leads to an engineering trade-off between precision and speed—two properties that characterize elasticity [6]. The HPA, implemented as a control loop, computes in every iteration[4] a ratio between the average utilization among the available containers and the specified target CPU utilization. The obtained factor multiplies the current number of containers which results either in a scale-out decision, scale-in decision or in a decision to keep the current number of containers. To avoid consecutive decisions that are in conflict (an immediate scale-in decision that follows a scale-out), the HPA uses a default cool-down period of three minutes for the next scale-out decision and one other of five minutes for the next scale-in decision.

Using HPA as a complementary method to predictive approaches leads to several considerations. Any deviation in load intensity causes the predicted number of containers for that time-interval to saturate computing resources. Having set a high target utilization, in cases where load intensity deviations are small, the algorithm will react accordingly by adding containers in small units. However, when the deviation approaches the peak, the algorithm will need several scaling steps—with a fine-granular magnitude—to match the supply with the demand. Setting a lower target CPU utilization will increase the magnitude of scaling steps, thus, reaching the target number of containers faster.

Kubernetes HPA works also with custom metrics such as the length of the request queue which suits better the given scenario. Users— microservice owners—can define a target queue length and the autoscaler will add/remove instances to keep the length equal to the target. The algorithm computes periodically the ratio of the measured length and the user defined target. The obtained ratio is multiplied with the current number of containers to obtain the new number of containers. In cases where queued events have a time-deadline, the targeted length of the queue should be set close to zero. As it is known from queuing theory [3], as soon as the rate at which the queue increases is greater than the processing rate, events will start to accumulate in the queue. Considering this, even if the difference between the rate the queue increases and the processing rate requires only a single additional container, the algorithm—because of the ratio it computes—may double, triple or multiply severalfold the number of containers.

Another alternative to solve the presented problem is the use of threshold-based rules from Amazon ECS (EC2 Container Service). Amazon offers an autoscaling capability called Service Auto Scaling. It enables users to define scale-out and scale-in policies which are executed when chosen metrics exceed user specified thresholds. In both types of policies users have to provide the number of containers to add or to remove. There are two different ways in which the number of containers can be provided: as a fixed absolute value or as a percentage. Both of them entail several considerations. Using a fixed value close to one has the benefit of matching the supply precisly with the demand although with a lower speed. Setting a higher fixed value increases the speed at which resources are provided, however, it affects precisions in cases where less are needed. With a fixed percentage approach, users specify how much of the current capacity should be added or removed in a scaling step. This adds complexity because the policy can produce two corrective actions with a large difference in the number of containers added even though the magnitude of devation requires the same amount of additional containers. Both ways of providing fixed capacity affect the speed and the precision of the system's elastic property. To improve these properties, Amazon offers a feature called step scaling for Auto Scaling EC2 instances. It allows users to specify multiple scaling steps based on different thresholds. They can specify the following semantic: "if CPU utilization alarm triggers and utilization value is 60% add two tasks or if the value is 80% add four tasks". There is a lack of documentation as to whether step scaling is also offered for containers.

Current reactive approaches have the following issues when dealing with unpredictable changes in workload intensity: inability to handle unpredictable changes in workload intensity, the need to sacrifice speed for precision or vice-versa and they require a considerable knowledge for configuration. Given these issues, this paper proposes a hybrid method that reacts to changes in workload

[2]https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[3]http://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-dg.pdf
[4]default frequency: one in 30 seconds

intensity together with an extra-capacity mechanism that captures a range in the acceleration of workload intensity. Our aim is to provide an alongside mechanism that complements proactive techniques that produce the base resource supply for coarse-grained time units.

3 ADAPTATION PROCESS

We tackle the problem of enabling elasticity for a containerized microservice with an elasticity controller that has a novel adaptation process. We see the proposed method as a complementary mechanism to predictive approaches. Predictive approaches are able to predict the demand for coarse-grained timescales i.e. days and hours, by building models for reoccurring patterns. For any non-predictable deviation in workload intensity, the elasticity of the microservice is controlled with the proposed adaptation process.

To begin with, in Figure 1, we depict an overview of our elasticity controller and its adaptation process in phases of a MAPE-K control loop [8]. The adaptation process alters the number of containers of a specific microservice in a specific context. Thus, besides for operating, it needs *knowledge* that describes the context and the microservice; specifically it needs information concerning the `capacity` that a single container can provide. When the controller starts, it *monitors* periodically the `workload intensity` which reflects the need to scale out or in the processing containers. Based on the current value, the controller executes the *analysis* phase where it either increases, decreases or decides to keep the current number of replicated containers. In the *planning* phase the controller enforces scalability bounds: avoiding a scale-out above a specified maximum and a scale-in below a given minimum. Further, to avoid switching between two condradictory scale decisions, the controller uses an arbitrary silence period of three minutes between the last scale decision and a following scale-in decision. In the *Execution* phase, the controller executes platform-specific APIs to increase or decrease the number of running containers. The novelty of the elasticity controller relies on three parts which will be described next. First, we will describe the specific knowledge that characterizes the microservice for which elasticity is enabled. Then we will show how the `workload intenstiy`—on which scaling decisions are based—is constructed. Finally, we will elaborate two techniques which are used together in the *analysis* phase to obtain a propoal value for the number of containers that should exist at that particular point in time.

Knowledge: capacity. The specific context is the well-known pattern of decoupling microservices from its clients with a messaging queue. Clients push requests to a specific queue on which the microservice in scope is attached as a listener. To describe the microservice, the controller needs the `capacity` for a single instance of a microservice. The value for the `capacity`, denoted with μ, determines the maximum workload intensity that a single container can handle. In the current version of the controller we assume that work is homogenous, therefore the estimated capacity μ is for a single class of requests. The method can be extended for several queues with different request classes where for each a capacity estimation is needed.

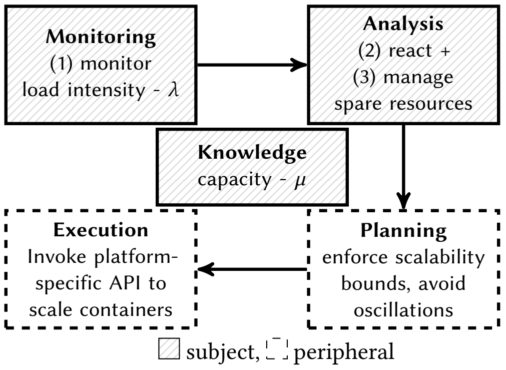


Figure 1: The MAPE-K loop of the elasticity controller

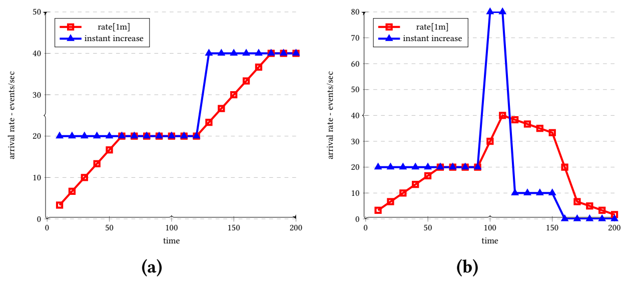


Figure 2: Two characteristics caused by averaging the incraese rate over the last minute: (a) the delay and (b) the inability to capture the magnitude of short wavelength spikes

Monitoring: load[5] intensity. For the `load intensity`, on which scaling decisions are based, we chose the rate at which the queue has increased over the last minute. We denote this value with λ. The average rate over the last minute is computed based on a time-series that contains samples of the total number of published events at a certain point in time. In our prototype implementation, we obtain the chosen signal from Prometheus by using the `rate` function[6] over a one-minute interval. We argue that this signal reflects the necessity to scale. Having provisioned resources with a predictive technique for units of hours, we aim to rely on a reactive mechanism which bases its scaling decisions based on a signal that reflects the load in fine-grained time units. By computing the rate over the last minute, the signal represents the right magnitude of trends that change in minute timescales. In cases where load spikes occur and they level off to a new value, as it is shown in Figure 2a the intensity reflects the right magnitude with around one minute delay. However, when spikes of a short wavelength occur the signal used for load intensity filters them out. As it is shown in Figure 2b, the curve marked with triangles depicts the computed instant increase, as a difference between two consecutive queue length samples. As seen, the curve marked with squares which represents the average rate over the last minute filters out the short wavelength spike that occurred.

[5]since work is constant
[6]https://prometheus.io/docs/prometheus/latest/querying/functions/

Analysis: react. When the elasticity controller obtains λ and has the knowledge on the `capacity` - μ, it calculates the appropriate number of containers to handle the current load intensity. Thus, as an output, it produces the ratio $\lceil \frac{\lambda}{\mu} \rceil$. The rationale is to keep the supply ($n * \mu$), where n is the number of replicated containers, equal through time with the demand which is reflected by the load intensity λ. Despite the rationality, there exist several impediments which affect the practicallity of a react-only mechanism. One obstacle is related to two inevetiable delays: one to reflect the change in magnitude inherited by the used load intensity, and another delay exists in starting containers which even though it has improved, it takes still a considerable amount of time in which period quality objectives can be violated. Another challenge exists in dealing with spikes that have a short wavelength and are not reflected in the used load intensity. We tackle these obstacles by providing and managing *aditional containers as spare capacity.*

Analysis: manage spare resources. The adaptation process manages the number of additional containers based on two parameters. One is the `initial number` of spare containers (s_i) that should always exist parallel to the number of containers calculated by the reactive mechanism. The other parameter is a `threshold` value which is given as a percentage and determines when the spare capacity scales out and when it scales in. When the controller starts, it provides the specified initial amount of containers as spare capacity alongside the base capacity. When the recent value of load intensity is obtained, the controller calculates the percentage with which the spare containers are contributing to the new arrival rate. If the percentage is above the specified threshold, the controller provides one additional container as a buffer. If the percentage is lower than the threshold, the controller decreases the number of spare containers towards the initial value by reducing the amount by one.

The `threshold` (thr) percentage value determines when the pool of additional containers increases. Conceptually, the chosen value for the threshold classifies load intensity increases into two possible behaviors: the one in which the load intensity rappidly approaches the peak load and requires provisioning of resources beforehand in order to meet quality objectives, and, the other in which the load intensity increases steadily for which the `initial number` of additional containers suffices. For the first possible behavior the controller increases the pool of additional containers by one. At the moment when computing a decision the controller is not able to predict if load intensity will continue to increase or drop, thus it decides to add the minimum unit possible to anticipate future needs. Since the reactive technique will also create containers to handle the observed load, the controller sacrifices the quality of elasticity in favor of providing resources beforehand and meeting quality objectives. When load intensity increases gradually and does not exceed the specified threshold, the `initial number` of spare containers (s_i) serves the purpose of assisting the current capacity while the number of containers settles from one value to the other .

To summarize, Equation 1 shows the function for calculating (n', s') with n' being the number of containers produced by the reactive part and s' the amount of spare containers in every iteration of the controller. As seen, the amount is a function of the observed load intensity λ, the given processing capacity μ, the current number of containers n and the current number of spare containers s. In cases where the current number of spare containers - s, are contributing with their capacity more than the specified percentage threshold - thr, then $s' = (s + 1)$. If their contribution is lower than the specified threshold, the number of spare containers is scaled down by one towards the specified initial amount of spare containers - s_i.

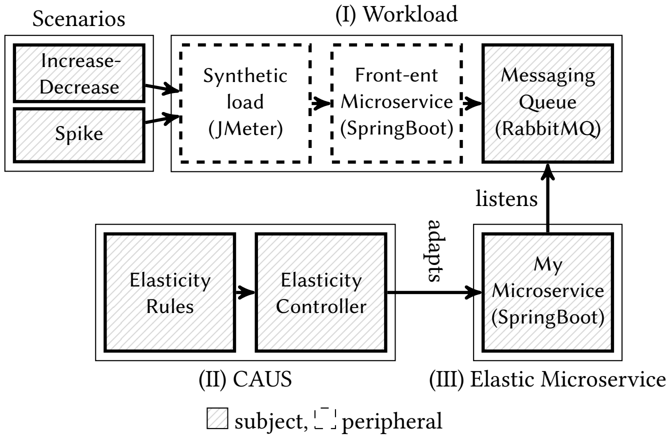


Figure 3: Experimental Setup

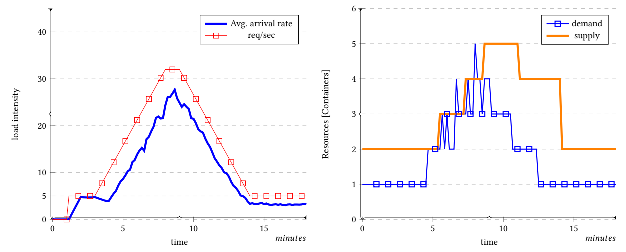
$$f(\lambda, \mu, n, s) = \begin{cases} \Big(\lceil \frac{\lambda}{\mu} \rceil, (s+1)\Big), & \text{if } \lambda \geq \mu * (n + thr * s) \\ \Big(\lceil \frac{\lambda}{\mu} \rceil, min(s_i, s-1)\Big), & \text{if } \lambda < \mu * (n + thr * s) \end{cases} \tag{1}$$

4 EVALUATION

To evaluate the proposed adaptation process, we constructed the experimental setup shown in Figure 3. The experimental setup consists of three parts: (I) components that are used to create the demand for the targeted microservice, (II) the implemented prototype of the adaptation process - CAUS (**C**ustom **AU**to**S**caler)[7], and (III) the microservice which listens for events in a particular queue and its number of replicated containers is altered with the presented adaptation process The goal of evaluation is to measure the elasticity that the microservice achieves when its number of containers is altered with the presented method. Furthermore, we observe if the elasticity controller allows the microservice to achieve its quality objectives. Since the proposed adaptation process provisions extra containers to anticipate future needs, we compare the results of each test against a corresponding baseline which represents an ideal policy of over-provisioning with minimal containers to withstand changes in the constructed arrival patterns.

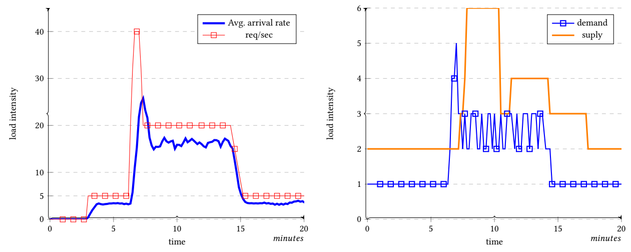
We chose to show two out of three test scenarios that were presented in [6]. A test scenario represents a unique demand pattern and simulates a deviation from the predicted load intensity. For each test, we generate synthetic load with Apache JMeter (version 3.1). We use the Throughput Shaping Timer plugin to construct the

[7]https://github.com/klinakuf/caus



(a) Synthetic load (b) Demand vs. Supply

Figure 4: Increase-Decrease scenario



(a) Synthetic load (b) Demand vs. Supply

Figure 5: Spike scenario

load patterns which differentiate the test cases. In every test case, JMeter—based on the rate that is expressed in the load pattern—invokes HTTP requests towards a helper front-end microservice (FEM). We provision enough FEMs for the designed demand. After a FEM receives a request, it creates an event and pushes it to the messaging queue. Specifically, events are pushed to the *SimpleRun* queue where they wait to be fetched and processed by the microservice.

On the other side, the targeted microservice is attached as a listener to the SimpleRun queue. To enable elasticity for the microservice, the elasticity controller requires knowledge of the processing capacity. We stress test the microservice—pertaining the number of containers at one—to obtain the capacity for a single replica. After several heuristics tests, we obtain the capacity, which follows a poisson distribution, with a mean of eight events per second. Further, we conduct a scalability analysis to guarantee the linear relationship between the number of containers and their altogether capacity. The capacity of eight events per second is given as an input in the elasticity rules.

To assess the achieved elasticity, we follow the proposed method in [4] and [9]. We evaluate elasticity in *precision* and *timeshare*. The precision shows the difference between the number of containers needed (the demand) and the number of containers provided (the supply). The timeshare illustrates the percentage of time the microservice spent in an over-provisioned or under-provisioned state. During a test run, we obtain the curve for the demand in containers by observing the queue growth rate in ten-seconds intervals; depending on the growth rate and the capacity, we obtain the number of containers needed at that interval in time. To obtain the supply curve, we observe the number of consumers for the SimpleRun queue in RabbitMQ. Based on these two, we obtain elasticity quality measures such as precision and timeshare.

Tests. Both tests start from a base load intensity of 5 events per seconds. The load intensity deviates from the base load by a factor of 6 in the first test, and by a factor of 8 in the other. The motivation for choosing the following patterns is twofold. On one hand, they represent possible patterns from a business point of view and on the other because they have different acceleration they create different test cases to validate the adaptation process. In both tests we provide elsaticity rules which have the same configuration: an initial amount of one additional container together with a 50% threshold to increase the pool of additional containers and a capacity of 8 $\frac{evt}{sec}$. Initially, in both tests there are two containers running: one of them is assumed to be provisioned for the predicted demand and the other is the specified additional buffer in the elasticity rules. Both tests last around 20 minutes in which the simulated deviation occurs.

The first scenario is named *increase-decrease* and is depicted in Figure 4. The pattern—the red curve in Figure 4a—has a linear increase followed by a linear decrease. From a business perspective, it represents a periodic run of certain processes, such as querying the salary statement at the end of the month. Intensity increases gradually towards the peak, where it remains for several minutes before starting to drop gradually. In the second scenario—named *spike*—the pattern experiences a high-intensity spike of a short wavelength. This pattern challenges the proposed elasticity controller which cannot capture the magnitude of short wavelength spikes. As it is shown in Figure 5a, the pattern simulates a sudden increase in intensity reaching the peak after certain flash-crowd events. The second figure in both scenarios shows the relation between demand and supply; the curve marked with suqares represents the number of containers needed whereas the thick non-marked curve represents the number of containers provided.

The first scaling decision in the increase-decrease scenario, occurs around minute 5, after load intensity (Figure 4a) increases above 8 events per second. At that moment in time, the reactive part of the adaptation process produces one more container. Afterwards, the controller scales the number of containers to four and five when load intensity exceeds 16 and 24 events per second respectively. The rate never accelerates beyond the 50% threshold that is set to manage the additional containers; thus, the spare capacity never scales; it remains at one. In contrast, in the second scenario, the spike that occurs exceeds the 50% threshold value. The curve marked with squares in Figure 5a denotes the per-second growth rate averaged over the last minute. As seen, because the spike has a short wavelength, the signal used by the controller does not capture the right magnitude. When the rate increases by a factor of 8 from 5 to 40, the average arrival rate reflects only an increase by a factor of 6. However, because of high acceleration, the intensity exceeds the buffer threshold of 50%, this way the controller increases the buffer by one and together with the inferred intensity scales to six containers which is the adequate amount of resources. The extra containers stay up for the next three minutes—the cooldown period.

Summary. Results for the two tests are summarized in Table 1. The two scenarios are compared against two constructed baselines which assume policies which hypothetically know the acceleration that load intensity experiences and over-provision with the

scenario	**increase**	$baseline_1$	**spike**	$baseline_2$
$precision_O$	**0.12314**	0.10165	**0.1201**	0.29952
$precision_U$	**0.00248**	0	**0.00579**	0
$timeshare_O$	**87.6%**	99%	**73.072%**	99%
$timeshare_U$	**2.479%**	0%	**2.894%**	0%
$elasticity_{nw}$	**0.94307**		**1.76079**	

Table 1: Summary of the two test cases compared to two reactive strategies with fixed amount of extra containers

necessary additional containers to handle it. For $baseline_1$, the calculations are obtained by assuming that there is always one extra container. Similarly, for $baseline_2$, because the load intensity experiences a spike, the measures are obtained by assuming a policy that overprovisions with three additional containers throughout the test. In the first test, as it is expected, the adaptation process overprovisions during 87% of the time. The precision of overprovisioning is affected by the cooldown period where no scale-in decision occurs. This worsens the precision compared to an ideal baseline policy of having only one additional container as spare capacity throughout the time. In the spike scenario, the proposed method overprovisions containers throughout 73% of the time with a precision which is similar to the previous test. However, the precision of the second test is better than the precision of a policy, which provisions three additional containers to cope with spikes that may occur in load intensity.

The load intensity signal—the thick non-marked curve in Figure 4a and 5a—successfully captures tendencies on the rate the queue is increasing for minute timescales. Based on these trends, the elasticity controller provisions and releases containers. On the other hand, if the rate at which the queue increases experiences oscillatory behavior in shorter timescales, requests in the queue will start to build up. The current expiration deadline for the events—which measures if quality objectives are met–is 30 seconds. The portion of messages that expire with the given deadline is zero in the first test and less than 3% in the second. Given a quality objective which specifies a shorter duration for messages to expire, in presence of oscillatory behavior in shorter timescales than minutes the portion of messages that fail to meet quality objectives will increase.

5 CONCLUSION AND FUTURE WORK

After highlighting the problems of current reactive techniques with regard to elasticity, we proposed a novel adaptation process that consists of two techniques which in combination with predictive approaches improve elasticity. The adaptation process alters the number of containers based on the rate the queue has increased over the last minute. We complemented the reactive mechanism with a pool of additional containers which is altered independently depending on the acceleration the load intensity experiences. We give microservice owners the possibility to set a custom threshold, which enables them to determine under which acceleration the load approaches the peak so that the controller anticipates future needs and provisions containers before the actual load reflects it. We showed that the method with the same configuration under increased variability in the arrival patterns—that simulate possible deviations—overprovisions most of the time with a similar and satisfiable precision of less than two unneeded containers.

For future work, we envision the following directions:

Feedback control. We choose to base scaling decisions on the rate the queue increased over the last minute. If the rate at which the queue increases experiences oscillatory behavior in shorter timescales, events in the queue will still start to build up. To tackle this part, as it is proposed in [5], a combination of the proposed adaptation process with a second feedback controller is necessary.

Cost-effective combined elasticity. A recent survey on the adoption of containers [1] shows that providers run a mean of eight containers per host machine. Currently cloud providers bill per hour on VM instances. Hence it is not a reasonable choice terminating a newly-provisioned virtual machine for which providers are charged for the full hour. How does this relation stand for containers and what are the costs of keeping idling containers? Further, it is important to assess how the time to resize the cluster will affect the autoscaling of containers with the presented method.

The rigidity of μ and capacity inference algorithms. The estimated capacity - μ marks out a rigid environment. The rigidity is both internal and external. For example, if the size of the thread-pool that serves enqueued events is altered then the processing capability of the microservice is affected and a new re-estimation is required. Further, if the microservice orchestrates other microservices to process an event, any change of the other services requires a re-estimation of capacity and thus, an update of the elasticity rules. The survey [1] shows that the average lifetime of containerized applications has reduced to 2,5 days compared to 23 days with applications in VM. It is of interest to evaluate current approaches on capacity inference and their relation to the newly created conditions: the increased release frequency and the reduction in responsibilities for microservices.

Variability of parameters. In presence of more knowledge regarding the arrival pattern and workload characteristics we can establish a better understanding of spikes and their acceleration so that parameters for configuring scaling rules can be tunned: the initial amount of additional containers and the percentage at which to increase spare capacity.

REFERENCES

[1] 8 surprising facts about real Docker adoption. [n. d.]. Docker Adoption. ([n. d.]). https://www.datadoghq.com/docker-adoption/

[2] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. 2017. Elasticity in Cloud Computing: State of the Art and Research Challenges. *IEEE Transactions on Services Computing* (2017).

[3] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S. Trivedi. 1998. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications.* Wiley-Interscience, New York, NY, USA.

[4] Nikolas Roman Herbst, Samuel Kounev, and Ralf H Reussner. 2013. Elasticity in Cloud Computing: What It Is, and What It Is Not.. In *ICAC*. 23–27.

[5] Philipp K Janert. 2013. *Feedback control for computer systems.* " O'Reilly Media, Inc.".

[6] Floriment Klinaku. 2017. *Elastic Microservices.* Master's thesis. Technical University of Darmstadt.

[7] Sebastian Lehrig, Hendrik Eikerling, and Steffen Becker. 2015. Scalability, Elasticity, and Efficiency in Cloud Computing: A Systematic Literature Review of Definitions and Metrics. In *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA '15).* ACM, New York, NY, USA, 83–92. https://doi.org/10.1145/2737182.2737185

[8] Hausi A Müller, Holger M Kienle, and Ulrike Stege. 2009. Autonomic computing now you see it, now you don't. In *Software Engineering*. Springer, 32–54.

[9] A Weber. 2014. Resource Elasticity Benchmarking in Cloud Environments. *Master's Thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany* (2014).