K8-Scalar: a workbench to compare autoscalers for container-orchestrated database clusters

Wito Delnat
Dept. of Computer Science, KU Leuven
Leuven, Belgium
wito.delnat@gmail.com

Eddy Truyen, Ansar Rafique, Dimitri Van Landuyt, Wouter Joosen
imec-DistriNet, KU Leuven
Leuven, Belgium
firstname.lastname@cs.kuleuven.be

ABSTRACT

Although a considerable amount of research exists on auto-scaling of database clusters, the design of an effective auto-scaling strategy requires fine-grained tailoring towards the specific application scenario.

This paper presents an easy-to-use and extensible workbench exemplar, named K8-Scalar (Kube-Scalar), which allows researchers to implement and evaluate different self-adaptive approaches to autoscaling container-orchestrated services. The workbench is based on Docker, a popular technology for easing the deployment of containerized software that also has been positioned as an enabler for reproducible research. The workbench also relies on a container orchestration framework: Kubernetes (K8s), the de-facto industry standard for orchestration and monitoring of elastically scalable container-based services. Finally, it integrates and extends Scalar, a generic testbed for evaluating the scalability of large-scale systems with support for evaluating the performance of autoscalers for database clusters.

The paper discusses (i) the architecture and implementation of K8-Scalar and how a particular autoscaler can be plugged in, (ii) sketches the design of a Riemann-based autoscaler for database clusters, (iii) illustrates how to design, setup and analyze a series of experiments to configure and evaluate the performance of this autoscaler for a particular database (i.e., Cassandra) and a particular workload type, and (iv) validates the effectiveness of K8-Scalar as a workbench for accurately comparing the performance of different auto-scaling strategies.

KEYWORDS

Container orchestration, Autoscaling, Experimentation exemplar

ACM Reference Format:
Wito Delnat and Eddy Truyen, Ansar Rafique, Dimitri Van Landuyt, Wouter Joosen. 2018. K8-Scalar: a workbench to compare autoscalers for container-orchestrated database clusters. In *SEAMS '18: SEAMS '18: 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3194133.3194162

1 INTRODUCTION

The heterogeneous storage requirements of cloud applications have led to the development of various highly-scalable databases commonly referred to as NoSQL. NoSQL databases are ideal candidates for cloud applications as they offer improved support for *elasticity*, the ability to expand or contract resources in order to adapt to workload changes while taking into account application-specific QoS requirements of Service-Level Agreements (SLA) [18].

In practice, elasticity is typically implemented by a reusable autoscaler component that implements a control loop [3, 14, 26] where the actual (de)provisioning of resources is achieved by either (i) horizontal scaling, i.e. adding/removing database instances or (ii) vertical scaling, i.e. adding/removing resources in an existing database instance [13]. Thus, autoscalers are configured for a particular database system (e.g., Cassandra) and a particular workload type (e.g., 95% writes-5% reads) by means of *elastic scaling policies* that analyze certain metrics to detect imminent SLA violations and then trigger appropriate actions (e.g., scale up, scale down, etc.).

Although autoscalers for database clusters [12, 46, 48] or multi-tier applications [3] have been researched, developing an effective autoscaler for databases is still an art, rather than a science. Firstly, there is no one-size-fits-all solution as autoscalers must be customized for specific databases. For example, auto-scaling makes only sense when a database cluster is load balanced such that adding a new database instance will reduce the load of an existing instance. The load balancing algorithms that are specifically designed for that purpose are database-specific, however, and therefore the autoscaler's design must take into account how fast the load balancing algorithm adapts to requested cluster reconfigurations. Secondly, adding or removing an instance should be cost-efficient in the sense that it should not require extensive (re)shuffling of data across existing and new database instances. Thirdly, detecting imminent SLA violations accurately requires multiple type of metrics to be monitored and analyzed simultaneously, so that scaling decisions are not triggered erroneously by temporary spikes in resource usage or performance metrics. Such wrong scaling decisions can be very costly and actually hurt elasticity instead of improving it.

In order to enable researchers to implement and compare various auto-scaling solutions, an integrated research workbench for implementing and testing the effectiveness of a particular auto-scaling approach for a particular database and workload is needed. Ideally, the performance of different auto-scaling approaches should be performed in an identically-configured development or testing environment in order to support a fair comparison between different approaches. The most well-known suite for evaluating the performance of databases is YCSB [11]. Although this benchmark

SEAMS '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5715-9/18/05…$15.00
https://doi.org/10.1145/3194133.3194162

suite supports different workload types, it is not designed to test the performance of advanced self-adaptive approaches such as those based on control theory [17, 42] and machine learning [34] that can take into account the history of seasonal or oscillating workloads. Many tools and approaches have been proposed to automate scalability testing [2, 10, 23, 25, 43, 45]. Scientific reproducibility of benchmarking results and generalization thereof remain however key issues.

This paper therefore presents an easy-to-use and extensible workbench exemplar, named K8-Scalar, for implementing and evaluating different self-adaptive approaches to autoscaling database clusters. It is based on Docker [15], not only because Docker is a popular tool for deploying software using fast and light-weight Linux containers [41, 47], but also because of Docker's approach to image distribution, which enables researchers to store a specific configuration of a NoSQL database instance as a portable and light-weight Docker image that can be downloaded from a local or central Docker registry. The latter advantage also helps to address the above-mentioned necessity for repeating experiments in identically configured testing environments to improve reproducible research [4]. K8-Scalar also builds on Kubernetes [31], the de-facto industry standard for orchestration and monitoring of elastically scalable container-based services.

The technical contribution of K8-Scalar is that it integrates and customizes Scalar [23], a generic platform for evaluating the scalability of large-scale distributed systems, to support evaluating the performance of a particular auto-scaling approach for a particular database and workload type. Moreover, K8-Scalar extends Kubernetes with an advanced autoscaler for database clusters, based on the Riemann event processor [39] that allows for simultaneous analysis of multiple metrics and composition of multiple event-based conditions. This Advanced Riemann-Based Autoscaler (ARBA) comes with a set of elastic scaling policies that are based on the Universal Law of Scalability [21, 22] and the Law of Little [35], and that have been implemented and evaluated in the context of a case study on using Cassandra in a Log Management-as-a-Service platform [38].

The remainder of the paper is structured as follows. Section 2 motivates the technology choices for K8-Scalar and presents background on Kubernetes. Then, Section 3 presents the architecture and implementation of K8-Scalar, focusing on how Kubernetes and Scalar have been integrated. Moreover, it discusses how to plug into the workbench a specific autoscaler. Subsequently, Section 4 illustrates some typical usage scenarios of K8-Scalar and validates the performance and effectiveness of the default Riemann-based autoscaler integrated in K8-Scalar. Section 5 concludes and presents directions for future extension of K8-Scalar.

2 BACKGROUND AND MOTIVATION FOR USED TECHNIQUES

This section motivates the selection of Kubernetes and Apache Cassandra and presents the necessary background to comprehend the remainder of this paper.

2.1 Kubernetes

Despite the clear benefits of container and light-weight virtualization technologies, containers or VMs only provide a single-node solution. Container orchestration frameworks have been designed to support the deployment and management of a distributed application as a set of containers across a cluster of nodes. *Kubernetes* [31], *Mesos* [1] and *Docker Swarm* [16] are the most advanced container orchestration frameworks. Kubernetes [6] is the most popular container orchestration framework [19, 37], and has been pushed by the Linux Foundation as an industry standard for cloud native applications [36]. Various private and public cloud providers provide commercially-supported and certified Kubernetes installations with respect to this standard [30, 36].

A Kubernetes cluster consists of a master and multiple worker nodes (see Figure 1). Workers contain scheduled Pods. The master is responsible for receiving requests, making global decisions (e.g. scheduling), and detecting and responding to cluster events [29].

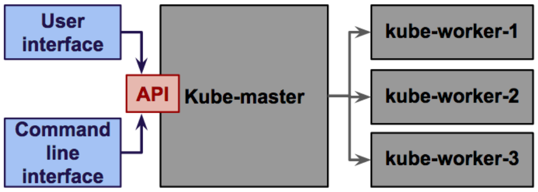

Figure 1: Architecture of Kubernetes

The following paragraphs explain the most important Kubernetes concepts.

Master and Worker nodes run on separate physical or virtual machines on which Pods can be scheduled.

Pods. A Pod is a set of co-located containers that logically belong together and therefore are always deployed together on the same node. Pods are therefore also the unit of failure. An important characteristic of Pods is their fast lifecycle. For example, a Pod will be terminated and recreated when a problem occurs instead of trying to resolve it. This results in *transient* IP addresses.

Persistent volumes. Pods and their containers are by default stateless; when a Pod dies or is stopped, any internal state is lost. Therefore, so-called persistent volumes can be attached to store the persistent state of the container on disk. Persistent volumes can be implemented by services for attaching block storage devices to guest virtual machines such as OpenStack's Cinder service, distributed file systems such as NFS, cloud services such as GCE Persistent Disk or just be a local directory on the node.

Services are an abstraction on top of a pool of replicated Pods to provide a *stable* IP address with load balancing capabilities. Kubernetes also supports an internal DNS service that enables lookup of the cluster IP address of a service based on its name that is specified in its Service configuration.

Requests, Limits and Quota. Per container and per Pod, it is possible to specify Requests, which are guaranteed available quantities of resources available (e.g. 250 milliCPUs and 500Mi of memory), and Limits that are maximum hard limits on available resources (e.g. 0.5 CPU, and 1Gi). It is also possible to specify per application or user organization soft and hard quotas for CPU, memory and disk storage. Network and disk I/O limits will be supported in the future.

StatefulSets provide a solution to complications that arise when deploying a database cluster as a set of replicated Pods. The essence of these complications is that the built-in round-robin load balancing policy behind Kubernetes' Service abstraction is in conflict with the existing load balancing and replication algorithms of database cluster software systems: putting a round-robin load balancer in front of a database cluster is not going to work because the database cluster already implements its own load balancing scheme that is based on how data is replicated across multiple database instances. To handle this incompatibility, K8s version 1.5+ offers support for the concept of StatefulSet [32], which is a higher-level configuration abstraction for automating the deployment of database clusters such that database instances are exposed to clients via a stable, unique DNS name (by relying on the internal DNS service of Kubernetes). Pods of a StatefulSet are also created and deleted in an ordered fashion - this constraint is important for master-slave database architectures such as MongoDB. Finally, persistent volumes for storing the persistent state of a database are automatically provisioned on a per Pod basis.

Horizontal Pod Autoscalers (HPAs) are Kubernetes' solution to autoscaling. HPAs are designed to hold the average CPU usage (or the average of another metric) of the Pods of a Service or StatefulSet around a certain percentage by adding or removing Pods. However, this HPA solution does not allow for simultaneous analysis of multiple metrics.

2.2 Apache Cassandra

In this paper, we select Apache Cassandra [9] as the NoSQL technology for illustrating the workings of K8-Scalar in the context of a Log Management-as-a-Service (LMaaS) application [38]. The LMaaS application offers among others a central cloud-based service for real-time aggregation and storage of log files in the cloud and offers an SLA with respect to latency of write requests and availability of data. As the workload type for this service consists of 95% write requests and 5% read requests, Cassandra has been selected as the underlying database cluster because its design is optimized for write-heavy workloads with large volumes of data and provides basic support for network partition tolerance. It has been configured with high availability and eventual consistency [5] to support the specific SLA.

To support high-performance writes of large volumes of data, Cassandra uses an in-memory write-back cache technology, called Memtable [8] . It will batch write requests and eventually *flush* them to persistent storage. These flushes occur periodically or when the Memtable's memory limit is achieved. Before a Memtable is flushed, a new one is created. This design choice makes the *available CPU power* as the primary performance bottleneck of a single database instance. This knowledge about Cassandra will be used as the basic assumption to appropriately configure the ARBA autoscaler.

3 ARCHITECTURE AND IMPLEMENTATION OF K8-SCALAR

This section presents the architecture and implementation of the K8-Scalar workbench and illustrates how a specific autoscaler can be plugged into the workbench.

3.1 Architecture and implementation

Figure 2 presents an overview of the architecture. It consists of five components: (i) a Kubernetes cluster consisting of multiple worker nodes, which must be installed on top of a network of machines that are appropriately configured to support a performance testing environment, (ii) the Heapster service [27] – the default monitoring solution of Kubernetes, (iii) a running database cluster, (iv) a particular autoscaler that is to be tested for the database and (v) an experiment-controller to measure the performance of the autoscaler in the context of a specific workload type. Not only the database instances, but also the autoscaler and experiment-controller components are deployed as Pods on the running Kubernetes cluster. This is because the latter components may also need to be replicated depending on the size of the generated workload.

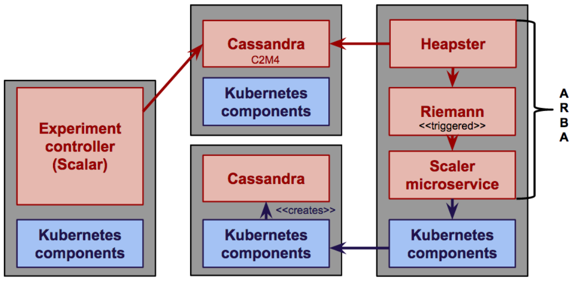

Figure 2: Architecture of the K8-Scalar exemplar

The current implementation of this architecture adopts the following technologies for each of these architectural components: (i) we have used the universal `kubeadm` command-line interface [33] for setting up Kubernetes clusters in a private OpenStack cloud; (ii) the Heapster service can be easily installed as a so-called add-on service; (iii) the StatefulSet configurations for deploying a Cassandra cluster for the experiments in Section 4 are available at K8-Scalar's GitHub repository [24]; (iv) the default ARBA autoscaler implements the MAPE-K reference model [26]: the monitoring function is based on Heapster, the Analysis and Planning functions are implemented on top of the Riemann event processor [39] – a powerful stream processing domain-specific language (DSL) on top of Clojure – and the Executor function is implemented by a separate Scaler service that offers a REST-based interface for specifying scaling commands in a portable manner; (v) the experiment-controller is based on Scalar [23], a fully-distributed, extensible load testing tool, implemented in Java, that can generate high workloads of requests using multiple coordinated nodes.

Scalar supports collection and analysis of absolute request latencies as well as an analysis of the scalability of the tested system with respect to the Universal Law of Scalability [21, 22]. It can be easily extended for testing a specific distributed system (i.e., a Cassandra database cluster) by implementing specific user types as Java classes (i.e, a `CassandraWriteUser` and a `CassandraReadUser` class), by specifying the desired distribution of the user types (e.g 95% `CassandraWriteUser` and 5% `CassandraReadUser`, and by a configuring a specific fluctuating or seasonal workload profile. The latter workload profile can be created from a generic template (see Listing 1) that defines an experiment as consisting of multiple runs, where each run consists of a constant low load phase, a ramp-up phase, a peak load phase and a ramp down phase. Per run, the duration of each phase and the maximum number of concurrent users for the peak load phase should be configured. Although Scalar has not been designed with system elasticity tests in mind, this template is still sufficiently expressive for testing various elasticity scenarios with dynamically fluctuating workload volumes. Note that Scalar only measures response latencies during the peak load phase and ramps down to the minimal peak load. Listing 1 shows how to circumvent this behavior in the ramp-down phase for a linearly increasing workload profile with seasonal peak workloads.

3.2 Extensibility of K8-Scalar

In order to replace the default ARBA autoscaler with another autoscaler, it is important to understand that the stable parts of the current implementation are Kubernetes and Heapster. Heapster can store collected metrics however into different backends, which are referred to as sinks. Heapster currently supports 16 different sink types, including the Riemann sink [28]. To plug-in another autoscaler, the autoscaler should thus be compatible with one of these sink types.

Listing 1: Workload profiles are configured in the `experiment.properties` config file of Scalar

```
## LOAD PROFILE
#
#                               ____
#                              /    |______...
#            ____             /
#           /    |___________/
#          /
#     ____/
#
# The number of concurrent users for peak load for the entire cluster. For each
# number in this comma separated list, a new run is started.
user_peak_load=10,100,90,90,200,190,190
user_warmup_duration=0
user_ramp_up_duration=360
user_peak_duration=360
user_ramp_down_duration=0
user_cooldown_duration=0
user_wait_inbetween_runs=0
```

The REST-based interface of the Scaler service of ARBA also aims to offer a portable specification of scaling actions for any type of application and any type of container orchestration framework. As such, we expect that large parts of the Scaler service can be reused.

Finally, Scalar can be easily applied to other databases. To implement a particular workload type for a new database, one needs to implement appropriate `User` and `Request` classes. Moreover, Scalar itself can be easily extended with additional plugins [23]. It is of course also possible to use another benchmarking tool if that tool is more appropriate for a particular experiment.

4 ILLUSTRATION AND VALIDATION

The section illustrates typical development and operational activities of K8-Scalar and also validates the operation of the workbench for a Cassandra database. All experiments are performed in a private OpenStack cloud. The physical machines of this cloud come with 2.60 GHz Intel Xeon E5-2660 processors and 128GB DDR3 memory. Furthermore, all virtual machines (VMs) are identical: Ubuntu 16.04 instances with 4 CPU cores and 8 GB of memory. All Pods in the Cassandra StatefulSet have equal Requests and Limits of 2 CPUs, and 4 GB of memory. Note that as each VM has 4 CPUs and 8GB of memory, the Cassandra Pod can maximally use 50% of a VM's resources. The Cassandra cluster is configured with the *Murmur3Partitioner*, the *SimpleStrategy* as the replication strategy, a consistency level of *ONE* and a replication factor of one [9][1].

We propose eight consecutive steps of using K8-Scalar for fine-tuning the ARBA autoscaler to *a specific database* (i.e., in the LMaaS case study, Cassandra), a *specific workload type* (i.e., a write-heavy workload with 95% writes and 5% reads) and a *specific workload profile* (i.e., a linearly increasing request rate):

(1) Setup a Kubernetes cluster with the Heapster monitoring service.
(2) Setup the Cassandra database cluster with one instance
(3) Implement the desired workload type in Scalar and store it as a container image in a registry.
(4) Deploy the experiment controller with the new image of Scalar
(5) Using Scalar, configure and run the linearly increasing workload profile with the single database instance for determining an appropriate allocation of resources to support a service level objective (SLO) for that single database instance, and a mapping from SLO violation events to critical points for request rate and resource usage (i.e., CPU utilization, disk usage, network usage).
(6) Implement and build a container image of ARBA that is extended with a set of elastic scaling policies and configured with the observed critical points for resource usage.
(7) Deploy the ARBA autoscaler with a particular scaling policy
(8) Test the scaling policy by re-running the linearly increasing workload profile in Scalar and determine the number of SLO violations of the Cassandra cluster by off-line analysis of the metrics gathered by Scalar (i.e., absolute request latencies, mean and percentiles of request latencies).
(9) Repeat steps 5, 6 and 7 until an elastic scaling policy has been found that performs well.

A detailed tutorial on how to perform these steps is presented in the GitHub repository of K8-Scalar [24]. In the two following illustration and validation subsections, we present experimental results obtained by executing steps 5 to 7 with respect to the Cassandra database and a linearly increasing workload profile with 95% writes and 5% reads. As explained in Section 2, this type of workload will have CPU as primary performance bottleneck.

4.1 Illustration

4.1.1 Mapping high-level SLOs to resource metrics. As containers can be configured with guaranteed reserved resources and maximum limits of available resources (see Section 2), a potential benefit of container orchestrated services is that service-level objectives (SLOs) in terms of desired throughput or response latency can be mapped to resource allocation policies on the one hand, and container resource usage metrics can be directly mapped to SLO violation events on the other hand [44]. However such mappings must take into account the specific database technology and application scenario and should be based on a proven queuing or capacity

[1] For reasons of simplicity, this illustration does not include a setup with dynamically provisioned persistent volumes.

theory. Crucial to preventing SLO violations using resource-level metrics only is thus performing an experiment to find a correlation between QoS metrics that correspond with SLO violations on the one hand, to low-level resource metrics on the other hand. In this experiment, we assume that the SLA of the LMaaS service specifies the following SLO with respect to request latency:

SLO1: the `0.95th` percentile of the write request latencies must be lower than `x=150 ms`.

Our assumption is that for a workload with a linearly increasing request rate, the average request latency remains below a constant until the request rate crosses a certain *critical point* (or threshold value) of *RR* requests per sec. After this critical point, the latency will increase asymptotically. This assumption is based on the universal law of scalability [22], which defines a model of relative capacity and that can be combined with the law of Little [35] to arrive at a quadratic expression for request latency in terms of the request rate. This leads to the following relation between latency and request rate:

Critical request rate for SLO1: the `0.95th` percentile of write request latencies $\leq$ 150 ms $\leftrightarrow$ write request rate $\leq$ *RR* requests/sec.

Thus the goal of the experiment is to determine the critical point *RR* for the request rate and the critical point *CP* for resource usage. In order to determine the critical point *RR* for the request rate, we have built around Scalar a `stress.sh` script to precisely control the request rate of a particular workload profile in function of time. More specifically, the command shown in Listing 2 instructs Scalar to collect request latency metrics during 400 seconds per run. The workload starts at a run with 50 requests per second and increases up to 1500 requests/s with an increment of 50 requests/s. For these arguments, the experiment will thus consist of 29 runs and last 11600 seconds.

Listing 2: Usage of `stress.sh`

```
kubectl exec experiment-controller -- bash
bin/stress.sh --duration 400 50:1500:50
```

The experimental setup (see Figure 3) consists of three nodes in a Kubernetes cluster. The experiment-controller Pod uses Scalar to expose the write-heavy linearly increasing workload to a Cassandra StatefulSet with one Cassandra instance. As CPU resources are primarily stressed by this workload, we analyze the CPU usage metrics of the Cassandra instance to determine the critical point *CP* for CPU usage. These metrics are collected by Heapster and displayed with Grafana [20] as milliCPU seconds (also referred to as millicores).

The latency graph (see Figure 4) shows that the average latency stays below `x=150 ms` until a request rate of about `y=1050` requests per second. This critical point of the request rate is reached after two hours and thirteen minutes since the start of the experiment. The critical point for CPU can then be retrieved from the Heapster metrics of the Cassandra Pod at the above mentioned time (see Figure 5). At this point in time, the measured CPU usage amounts to 0.67∗2000 millicores. In terms of relative CPU usage with respect to the Pod's CPU limit, 67% is therefore the critical point.

Note that if other types of resources (e.g. available disk space, disk I/O bandwidth) form a primary or secondary scalability bottleneck, a similar experiment should be performed that aims to determine the critical points for other Heapster metrics (e.g. `filesystem/available` or `disk/io_write_bytes_rate`) [27].

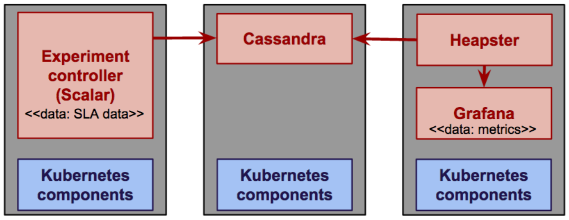

Figure 3: Experimental setup for determining the mapping between critical points of QoS metrics and critical points of resource usage

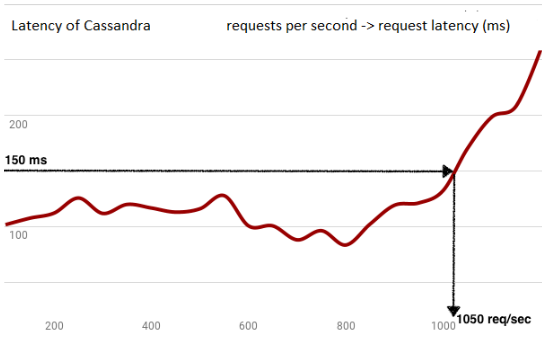

Figure 4: Request latency metrics (in ms) per request rate

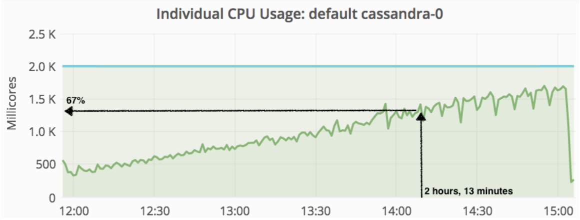

Figure 5: Grafana plot of the CPU usage of the Cassandra Pod (in millicores)

4.2 Validation

This section demonstrates the usefulness of K8-Scalar by validating that (i) ARBA is able to correctly execute a configured autoscaling strategy and (ii) K8-Scalar allows implementing and comparing autoscaling strategies that can simultaneously analyze multiple metrics.

4.2.1 Validation of reactive scaling. We validate that autoscaling strategies can be correctly executed in K8-Scalar. More specifically, we validate that when the CPU usage of a database instance passes the critical point of 67%, a database instance is added in a timely fashion and the higher CPU usage of the first instance will gradually be divided across the two instances. To validate this hypothesis, we study Heapster's generated CPU usage graph of both Cassandra instances. The experimental setup is very similar to what is shown in Figure 2 of Section 3. Again, a linearly increasing workload is exercised by means of the `stress.sh` script (see Listing 2). In addition, ARBA is configured with a simple strategy: the scaling threshold is chosen a bit higher than the observed *CP* of 67% in order to ensure that additional database instances are not created unnecessarily:

Strategy 1: *If CPU_usage >= 70%, then add one database instance*

Figure 6 shows the result of this experiment. These two graphs show the CPU usage of the existing and new database instances in function of the time. It is possible to conclude that a workload on a Cassandra StatefulSet that is higher than 70%, results in the creation of an additional instance after approximately 3 minutes (which includes the time for pulling the Cassandra image from a Docker registry and starting the Cassandra Pod). However, the autoscaling strategy itself does not perform well as Figure 6 shows that the CPU usage of the first instance, after a temporary drop, surpasses the critical point during a considerable time interval of the experiment. However, Cassandra’s decentralized architecture supports almost linear scalability for write-heavy workloads [7]. As such, the only possible explanation is that the addition of a second database has been performed too late for the linearly increasing workload profile.

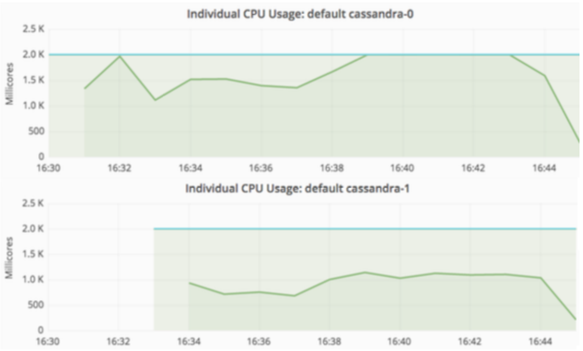

Figure 6: Results of the ARBA validation experiment

4.2.2 *Support for evaluation of complex autoscaling strategies.*

We determine whether it is possible to accurately compare the performance of more complex scaling policies that require parallel analysis of multiple metrics. For this purpose, ARBA has been configured with three Riemann-based strategies:

Strategy 2	$cpu_{usage} > CP$
Strategy 3	$[cpu_{usage} > CP - 5]$ $\vee\ disk_{usage} > 75\%$
Strategy 4	$[eliminate\_outliers(cpu_{usage})$ $> CP - 5] \vee disk_{usage} > 75\%$

Table 1: Scaling strategies for Cassandra

Each strategy becomes more complicated than the previous one. Strategy 2 changes Strategy 1, presented earlier, by triggering scaling when CPU usage reaches the observed *CP* of 67%. Strategy 3 triggers scaling faster (at *CP* − 5%) so that Cassandra’s load balancing algorithm gets the time to adapt before the request rate directed towards the first Cassandra instance surpasses the critical point *RR*. Furthermore, it also considers used disk space. Finally, Strategy 4 extends Strategy 3 by eliminating outliers in the third strategy. This is done by the `eliminate_outliers` functions, which creates a *moving-event-window* [40] stream and computes the median.

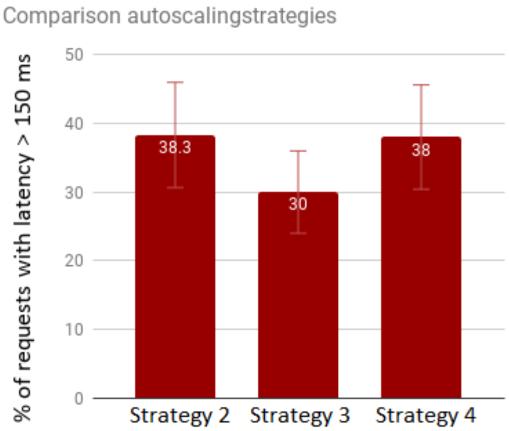

Figure 7: The same Scalar experiment is repeated multiple times. The mean and standard deviation of the ratio of requests with a latency higher than 150 ms are shown.

The results are presented in Figure 7 that shows the percentage of requests of which the latency is higher than the latency target of SLO1 (i.e., 150 ms). In order to meet SLO1, this percentage should be below or equal to 5%. As expected, strategy 3 performs better than strategy 2 by triggering scaling faster. Strategy 4 reacts slower than strategy 3 because the outlier elimination algorithm delays scaling. Somewhat unexpected, across all strategies, 25% to 45% of the requests have a latency higher than 150 ms and therefore none of strategies are effective in guaranteeing SLO1. This suggests that an effective MAPE-K auto-scaling strategy needs to include additional elements such as workload volume prediction [34] in the Analysis function and an accurate projection of the needed number of Cassandra instances in the Planning function.

5 CONCLUSION

This paper presented the K8-Scalar exemplar, a workbench for implementing and evaluating autoscalers for container-orchestrated services. It leverages state-of-the-art containerization and monitoring techniques and integrates and extends Scalar, which provides state-of-the-art coordination techniques for conducting large-scale and distributed scalability tests for any type of user behavior and workload type. The paper has illustrated and validated the operation of K8-Scalar in the context of auto-scaling NoSQL clusters, an active area of research. This has shown that K8-Scalar allows to correctly implement different variants of a Riemann-based autoscaler, which is included in the workbench, and to evaluate these variants swiftly using on-line resource monitoring and off-line request latency analysis. Finally, future work includes (i) creating a tool for automated experimentation and interpolation of mappings between critical points of SLOs and resource usage for different sizes of the same database cluster, (ii) improving Scalar's workload profiles for elasticity and (iii) adding an abstraction layer to Scalar that allows separating reusable workload types from database-specific client software. K8-Scalar is available at https://github.com/k8-scalar/k8-scalar.

ACKNOWLEDGMENTS

We thank the reviewers for their helpful comments. We also thank Thomas Heyman and Davy Preuveneers for the development of Scalar. This research is partially funded by the Research Fund KU Leuven and the IWT-SBO DeCoMAdS project.

REFERENCES

[1] Apache. 2018. Apache Mesos. URL: http://mesos.apache.org/, accessed 2018-01-23. (2018).

[2] Khun Ban, Robert Scott, Huijun Yan, and Kingsum Chow. 2011. Delivering Quality in Software Performance and Scalability Testing. In *Pacific Northwest Software Quality Conference.*

[3] Cornel Barna, Hamzeh Khazaei, Marios Fokaefs, and Marin Litoiu. 2017. Delivering Elastic Containerized Cloud Applications to Enable DevOps. *Proceedings - 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2017* (2017), 65–75. https://doi.org/10.1109/SEAMS.2017.12

[4] Carl Boettiger. 2015. An Introduction to Docker for Reproducible Research. *SIGOPS Oper. Syst. Rev.* 49, 1 (Jan. 2015), 71–79. https://doi.org/10.1145/2723872.2723882

[5] E. Brewer. 2012. CAP twelve years later: How the "rules" have changed. *Computer* 45, 2 (2012), 23–29. https://doi.org/10.1109/MC.2012.37

[6] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (2016), 50–57. https://doi.org/10.1145/2890784

[7] Jeff Carpenter and Eben Hewitt. 2017. *Cassandra: The Definitive Guide: Distributed Data at Web Scale.* O'Reilly. Chapter 2, page 18.

[8] Apache Cassandra. 2017. MemTables and SSTables. URL: https://wiki.apache.org/cassandra/MemtableSSTable, accessed 2017-08-04. (2017).

[9] Apache Cassandra. 2018. Understanding the architecture. URL: http://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archTOC.html, accessed 2018-01-29. (2018).

[10] Yong Chen and Xian-He Sun. 2006. Stas: A scalability testing and analysis system. In *Cluster Computing, 2006 IEEE International Conference on.* IEEE, 1–10.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10* (2010), 143–154. https://doi.org/10.1145/1807128.1807152

[12] Ioannis Konstantinou Dimitrios Tsoumakos and Christina Boumpouka. 2013. *Automated, Elastic Resource Provisioning for NoSQL Clusters Using TIRAMOLA.* IEEE. Conference paper at Delft, Netherlands.

[13] Amr El Abbadi Divyakant Agrawal and Sudipto Das. 2011. *Database scalability, elasticity, and autonomy in the cloud.* Springer-Verlag Berlin. DASFAA'11, Volume Part I, pages 2-15.

[14] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. 2006. A Survey of Autonomic Communications. *ACM Trans. Auton. Adapt. Syst.* 1, 2 (Dec. 2006), 223–259. https://doi.org/10.1145/1186778.1186782

[15] Docker. 2018. Docker Engine. https://docs.docker.com/engine/, accessed 2018-01-24. (2018).

[16] Docker. 2018. Swarm mode overview. URL: https://docs.docker.com/engine/swarm/, accessed 2017-08-04. (2018).

[17] Betty H. C. Cheng et al. 2009. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems (Lecture Notes in Computer Science)*, Betty H C Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee (Eds.), Vol. 5525. Springer, 1–26.

[18] Ludwig Heiko et al. 2003. *Web service level agreement (WSLA) language specification.* IBM Corporation. pages 815-824.

[19] Google. [n. d.]. Compare Mesos, Kubernetes and Docker Swarm. URL: https://trends.google.com/trends/explore?q=mesos,Kubernetes,DockerSwarm, accessed 2018-01-23. ([n. d.]).

[20] Grafana. 2017. The leading open source software for time series analytics. URL: https://grafana.com/, accessed 2017-08-04. (2017).

[21] Neil J. Gunther. 2007. *Guerrilla Capacity Planning.* Springer-Verlag, Heidelberg, Germany.

[22] Neil J. Gunther. 2008. A general theory of computational scalability based on rational functions. *arXiv preprint arXiv:0808.1431* (2008).

[23] Thomas Heyman, Davy Preuveneers, and Wouter Joosen. 2014. Scalar: Systematic scalability analysis with the Universal Scalability Law. In *2014 International Conference on Future Internet of Things and Cloud.* 497–504. https://lirias.kuleuven.be/handle/123456789/460752

[24] KU Leuven imec DistriNet. 2018. K8-Scalar. URL: https://github.com/k8-scalar/k8-scalar, accessed 2018-01-29. (2018).

[25] Prasad Jogalekar and Murray Woodside. 2000. Evaluating the scalability of distributed systems. *IEEE Transactions on parallel and distributed systems* 11, 6 (2000), 589–603.

[26] Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50. https://doi.org/10.1109/MC.2003.1160055

[27] Kubernetes. 2017. Heapster. URL: https://github.com/kubernetes/heapster/blob/master/docs/storage-schema.md, accessed 2018-01-26. (2017).

[28] Kubernetes. 2017. Heapster sink configuration. URL: https://github.com/kubernetes/heapster/blob/master/docs/sink-configuration.md, accessed 2018-01-26. (2017).

[29] Kubernetes. 2017. Kubernetes components. URL: https://kubernetes.io/docs/concepts/overview/components/, accessed 2017-08-04. (2017).

[30] Kubernetes. 2018. Kubernetes setup solutions. https://kubernetes.io/docs/setup/pick-right-solution/. (2018). Accessed: January 23 2018.

[31] Kubernetes. 2018. Production-Grade Container Orchestration. URL: https://kubernetes.io/, accessed 2018-01-23. (2018).

[32] Kubernetes. 2018. StatefulSets. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/. (2018). Accessed: January 23 2018.

[33] Kubernetes. 2018. Using kubeadm to Create a Cluster. URL: https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/, accessed 2018-01-29. (2018).

[34] Michlmayr A Leitner P. 2011. *Monitoring, prediction and prevention of SLA violations in composite services.* IEEE International Conference on Web Services (ICWS). Florida, USA, Miami, pp 369âĂŞ376.

[35] John DC Little and Stephen C Graves. 2008. Little's law. In *Building intuition.* Springer, 81–100.

[36] Ron Miller. 2017. 36 companies agree to a Kubernetes certification standard. https://techcrunch.com/2017/11/13/the-cncf-just-got-36-companies-to-agree-to-a-kubernetes-certification-standard/. (2017).

[37] OpenStack. 2017. OpenStack User Survey Report November 2017. https://www.openstack.org/assets/survey/OpenStack-User-Survey-Nov17.pdf. (2017). Accessed: January 23 2018.

[38] Ansar Rafique, Dimitri Van Landuyt, and Wouter Joosen. 2018. PERSIST: Policy-Based Data Management Middleware for Multi-Tenant SaaS Leveraging Federated Cloud Storage. *Journal of Grid Computing* (14 Mar 2018). https://doi.org/10.1007/s10723-018-9434-6

[39] Riemann. 2017. Riemann monitors distributed systems. URL: http://riemann.io/, accessed 2017-08-08. (2017).

[40] Riemann. 2017. Riemann Streams - moving event window. URL: http://riemann.io/api/riemann.streams.html, accessed 2017-08-04. (2017).

[41] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y C Tay. 2016. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th International Middleware Conference (Middleware '16).* ACM, New York, NY, USA, 1:1—-1:13. https://doi.org/10.1145/2988336.2988337

[42] Stepan Shevtsov, Danny Weyns, and Martina Maggio. 2018. Self-adaptation of software using automatically generated control-theoretical solutions. In *Engineering Adaptive Software Systems (EASSy).* Chapter 1, 1–17. to appear, preprint available at https://people.cs.kuleuven.be/ danny.weyns/papers/2018EASSy.pdf.

[43] Niclas Snellman, Adnan Ashraf, and Ivan Porres. 2011. Towards automatic performance and scalability testing of rich internet applications in the cloud. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on.* IEEE, 161–169.

[44] Eddy Truyen, Dimitri Van Landuyt, Vincent Reniers, Ansar Rafique, Bert Lagaisse, and Wouter Joosen. 2016. Towards a container-based architecture for multi-tenant SaaS applications. In *ARM 2016 Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware.* ACM. https://doi.org/10.1145/1235

[45] Martti Vasar, Satish Narayana Srirama, and Marlon Dumas. 2012. Framework for monitoring and testing web application scalability on the cloud. In *Proceedings of the WICSA/ECSA 2012 Companion Volume.* ACM, 53–60.

[46] Macro Netto et al. Vincent Emeakaroha. 2012. *Towards autonomic detection of SLA violations in Cloud infrastructures.* Future Generation Computer Systems. CVolume 28, Issue 7, July 2012, Pages 1017-1029.

[47] Miguel Gomes Xavier, Marcelo Veiga Neves, and Cesar Augusto Fonticielha De Rose. 2014. A Performance Comparison of Container-Based Virtualization Systems for MapReduce Clusters. *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2014), 299–306. https://doi.org/10.1109/PDP.2014.78

[48] L. Zhao, S. Sakr, and A. Liu. 2015. A Framework for Consumer-Centric SLA Management of Cloud-Hosted Databases. *IEEE Transactions on Services Computing* 8, 4 (July 2015), 534–549. https://doi.org/10.1109/TSC.2013.5