Leveraging Kubernetes for adaptive and cost-efficient resource management

Stef Verreydt, Emad Heydari Beni, Eddy Truyen, Bert Lagaisse, Wouter Joosen
imec-DistriNet, KU Leuven, Belgium
firstname.lastname@cs.kuleuven.be

Abstract

Software providers face the challenge of minimizing the amount of resources used while still meeting their customer's requirements. Several frameworks to manage resources and applications in a distributed environment are available, but their development is still ongoing and the state of the art is rapidly evolving, making it a challenge to use such frameworks and their features effectively in practice. The goal of this paper is to research how applications can be enhanced with adaptive performance management by relying on the capabilities of Kubernetes, a popular framework for container orchestration. In particular, horizontal as well as vertical scaling concepts of Kubernetes may prove useful to support adaptive resource allocation. Moreover, concepts for oversubscription as a way to simulate vertical scaling without having to reschedule applications, will be evaluated. Through a series of experiments involving multiple applications and workloads, the effects of different configurations and combinations of horizontal and vertical scaling in Kubernetes are explored. Both the resource utilization of the nodes and the applications' performance are taken into account. In brief, the resource management concepts of Kubernetes allow to simulate vertical scaling without a negative effect on performance. The effectiveness of the default horizontal autoscaler, however, depends on the type of application and the user workload at hand.

Keywords Auto-scaling, Adaptive resource management, Container orchestration frameworks

1 Introduction

Companies tend to move their resources to the cloud rather than keeping them on their own infrastructure as it offers numerous advantages (i.e. [1]). SaaS providers aim to provide their services as cost-efficiently as possible. In this regard, they face the continuous challenge of utilizing only a minimal amount of resources while still meeting their customers' requirements. One way to reduce the amount of resources needed is by locating multiple applications on the same node. This entails new challenges, e.g., determining how to divide over-provisioned resources among the deployed applications, and how to handle resource contention in general.

Kubernetes is a popular open source framework for managing containerized applications in a distributed environment, providing basic mechanisms for deployment, maintenance and scaling of applications [8]. It also offers several useful features for resource management. A pod in Kubernetes is the smallest deployable unit of computing which can be created and managed [9]. Containers belonging to the same application are grouped together in pods. The resources used by a pod can be limited, e.g., by setting its so-called requests and limits. The request is the amount of resources which it is guaranteed to get; the limit is the maximal amount of resources it can obtain [11]. When there is resource contention in a node, the resources are divided among the pods according to the relative weight of their requests. This mechanism is reffered to as *cpu-shares*. Another way of adjusting the resources available to an application is to scale it. Kubernetes offers default horizontal and vertical autoscalers, called the Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA) respectively. However, a well-known disadvantage of the VPA is that it currently requires to reschedule Pods when dynamically adjusting requests or limits [6].

Kubernetes is evolving rapidly. Meanwhile, both documentation and research are struggling to keep up with its development, especially since most research is aimed towards custom scaling techniques (e.g., [19] [4]). Little academic work is available on documenting the Kubernetes mechanisms themselves. Multiple questions may arise when configuring a Kubernetes cluster, e.g., how to choose suitable requests and limits, on which nodes to locate certain pods and how to configure an autoscaler. The goal of this paper is therefore to research how different Kubernetes features can be leveraged to cost-effectively manage resources in the framework while also meeting SLOs in the presence of dynamically evolving workloads. More concretely, we aim to provide answers to the following questions:

- What is the impact of different request and limit configurations?
- Does co-locating a high priority pod with a low priority pod affect the performance of the high priority pod?
- What is the performance impact of scaling an application using the HPA?

Different configurations may result in different performance gains (or losses) depending on the environment. For example, the user load may be very bursty, in which case using the oversubscription concepts or the HPA may not always lead to the expected results. Hence, the aforementioned impacts of the different mechanisms will be tested for both linearly increasing and bursty workloads. In this paper, however, only CPU intensive workloads will be examined.

The remainder of the paper is structured as follows. §2 discusses related work on resource management in cloud environments. §3 provides an overview of the test environment. In §4, the experiments and their results are presented. §5 concludes this paper.

2 Related work

Caravel [4] is a scheduler which co-schedules stateful and stateless applications in containerized environments. Stateful applications require more care to schedule when compared to stateless applications, as each replica of a stateful application is unique and the order of scheduling matters. When scaling out, stateless applications can scale instantly by spawning an identical copy. Scaling stateful applications requires more planning and is thus also slower in most cases. This makes vertical scaling the preferred method of scaling for stateful applications during a load peak, as no new replicas need to be scheduled. Vertically scaling stateful applications

entails its own risks, however. In container orchestration frameworks, applications using more than their requested amount of resources risk being evicted. In turn, a second problem called burst propagation arises: eviction of one replica results in the load being redirected to another replica. Now this replica's resource usage increases. It will in turn be prone to eviction, and so on. Another option for container orchestration frameworks is to throttle the resources of a container instead of evicting it, but this will cause a drop in performance during load peaks [4].

Caravel addresses these concerns by letting stateful containers evict stateless containers as these can be restarted in only a few seconds on another node. By doing so, it tolerates bursty behavior of stateful applications and reduces their evictions. Furthermore, Caravel imposes several mechanisms to control excessive evictions of stateless applications. In this paper, we explore how similar behavior could be achieved by only using Kubernetes mechanisms. However, the distinction is not made between stateful and stateless applications but rather between high and low priority applications.

Wong et al. [19] describes the (dis)advantages of vertical and horizontal scaling and proposes a hybrid solution, HyScale, developed on top of Docker Engine without any container orchestration framework. Wong compares their scaling solution to the standard autoscalers in Kubernetes. A first disadvantage of Kubernetes noted by Wong is that its default scaling solutions only consider one resource when making scaling decisions. Second, Wong explains how the Kubernetes autoscalers often lead to sub-optimal configurations. Wong's algorithm differs from the default Kubernetes HPA in two ways: the use of vertical scaling and the consideration of multiple metrics to make scale decisions. A complete discussion of the algorithm does not fall within the scope of this paper. Wong's experimental results show that HyScale outperforms Kubernetes for bursty workloads. For non-bursty workloads, their performance is similar [19].

When comparing horizontal and vertical scaling for CPU intensive workloads, Wong describes that Docker's *cpu-shares* mechanism can be used to "induce a form of vertical scaling, as increasing or decreasing shares directly correlate with an increase or decrease in CPU resource allocation to a container" [19]. When comparing their solution to Kubernetes, however, they only consider the Kubernetes HPA, without mentioning these *cpu-shares*. This paper evaluates the effects of combining the HPA with the *cpu-shares* mechanism in Kubernetes.

3 Test Environment

This section describes the specific tools used in the experiments and a brief overview of the three test applications.

Kubernetes cluster. All the experiments are run on a Kubernetes cluster deployed on the DistriNet private cloud, which is based on the OpenStask platform [10]. Four virtual machines are deployed: one master and three worker nodes. The cluster itself is created using *kubeadm* [5]. All of the machines are running the Ubuntu 16.04 operating system. Three of the nodes, including the master node, are allocated 2 CPUs and 4 GB of memory. The last node is assigned 4 CPUs and 8 GB of memory. The three worker nodes are placed on the same OpenStack computing node to minimize latencies between applications running on the nodes. On the first worker node, which is the node with 4 CPUs and 8GB of memory, an experiment controller is deployed. The second and third worker nodes are reserved for the applications to be deployed on. An overview of the setup is shown in Figure 1.

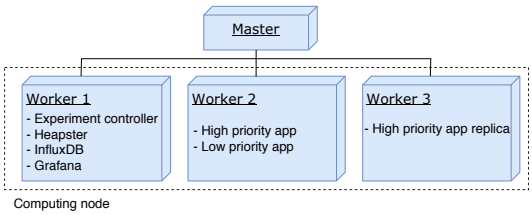


Figure 1. Test cluster setup

Experimentation tools. To monitor the resources in the cluster, a combination of Heapster [14], InfluxDB [12] and Grafana [17] is employed. To run and monitor the experiments, K8-Scalar [3] is used. K8-Scalar is an extensible workbench exemplar for implementing and evaluating different self-adaptive approaches to autoscaling container-orchestrated services [2].

3.1 Deployed applications

Cassandra based application. A Cassandra based application is selected as the first high priority test application. Only write operations are examined in this paper, as they are CPU intensive. The main QoS requirement for the application is thus the latency of write requests. Cassandra is well-suited for the experiments as its design is optimized for write-heavy workloads [3].

Artificial SaaS application. An artificial SaaS application developed at KU Leuven [13] is selected as a second high priority test application. The main benefit of this SaaS application is that the stressed resource is easily configurable through the application's REST interface. For example, if a CPU intensive workload needs to be tested, the memory intensity of the application can be set to zero by executing a simple REST command at runtime.

Low priority application. The low priority application used for testing the effects of co-locating pods executes a multiplication in an infinite loop. If sufficient resources are free, it continually uses a full CPU since it is single threaded. The exact CPU usage is known and roughly constant.

4 Results

In this section, the Kubernetes mechanisms for resource management are evaluated through a series of experiments.

4.1 Experiment 1: Determining the effects of request and limit configurations

If there is just one pod scheduled on a node, and if that pod has no limits set, then it should be able to use all of the nodes resources. This experiment tests this hypothesis using a Cassandra application.

Setup. First, the expected performance of the Cassandra application is described in an SLO as follows:

SLO. *95% of the requests sent to the Cassandra application must be handled within 150ms, as measured by the experiment controller.*

The Cassandra application is deployed on the second worker node with a CPU and memory request of respectively *1500m* and *2GiB*. No limits are set. In this experiment, the experiment controller

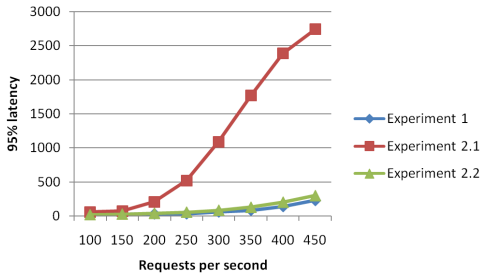


Figure 2. 95th percentile latencies during Experiment 1 and 2.

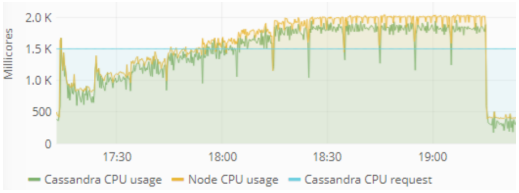


Figure 3. Second worker node CPU usage during Experiment 1. The CPU usage of Cassandra rises above its CPU request, illustrating that the application can use all of the overprovisioned resources.

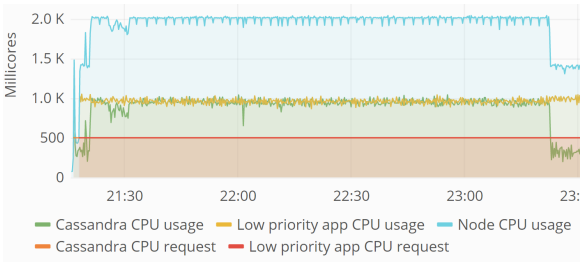


Figure 4. Second worker node CPU usage during the first test of Experiment 2. The deployed applications receive an equal share of the available CPU.

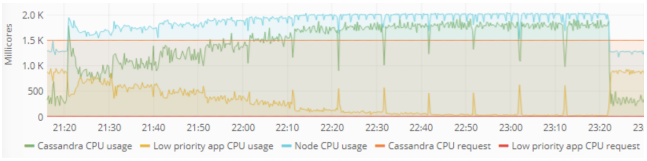


Figure 5. Second worker node CPU usage during the second test of Experiment 2. As the Cassandra workload rises, the amount of CPU cycles granted to the low priority application decreases.

sends an increasing amount of requests to the Cassandra application, starting at 100 requests per second up to 600 requests per second, increasing with 50 requests per second every 600 seconds.

Results. Figure 2 shows the 95th percentile latencies of the requests, as reported by the experiment controller. At around 400 requests per second, the SLO is violated. Figure 3 shows the CPU usage of Cassandra and the second worker node. It illustrates that at around the same amount of requests per second, the node uses all of its available CPU, as 2.0K millicores equals 2 CPUs. This validates that the bottleneck is indeed the CPU. Furthermore, Figure 3 shows that the Cassandra application is able to use almost all of the overprovisioned resources. This in line with expectations, as the resources on the node are only contended by the Cassandra application.

4.2 Experiment 2: Determining the effects of co-locating a high and low priority application

The goal of this experiment is to answer whether it is possible to increase cost-efficiency by using the Kubernetes oversubscription mechanisms described in §1. Through two tests, we illustrate the effects of different request configurations when a high and low priority pod are co-located on a node.

Setup. The low priority application described in §3.1 is added to the second worker node. The experiment consists of two separate tests. During the first one, Cassandra and the low priority pod each have a CPU request of *500m*. For the second test, Cassandra has a CPU request of *1500m* while the low priority pod has a CPU request of only *10m*. The workload applied is the same as the one applied during Experiment 1. During the first test, both pods should receive an equal amount of CPU cycles. The results of the the test should indicate significantly higher 95th percentile latencies when compared to the results from Experiment 1, because the Cassandra pod cannot use all of the resources on the node. During the second test, Cassandra's performance should only be affected slightly due to the aforementioned *cpu-shares* mechanism.

Results. Figure 4 shows the CPU usage of both the Cassandra pod and the low priority pod during the first test, when their requests are equal. It shows that the available CPU is split equally between both pods. As expected, Figure 2 illustrates that the proposed SLO gets violated at about half the amount of requests per second when compared to Experiment 1.

Figure 5 shows the worker node's CPU usage during the second test of this experiment. As the amount of CPU needed by the Cassandra pod increases, the amount of CPU available to the low priority pod decreases, which is according to expectations. The 95th percentile latencies shown in Figure 2 illustrate a slight increase of latencies when compared to the Experiment 1.

The slight decrease in performance comes with the benefit of a higher resource utilization. Comparing the worker node's total CPU usage during Experiment 1 to the node's total CPU usage during this test, the latter shows a significantly higher resource utilization when the workload of the Cassandra pod is low.

4.3 Experiment 3: Determining the effects of bursty workloads

In this experiment, a bursty workload is applied to examine its effects on Cassandra's performance. In the previous experiment, Cassandra could process 350 requests per second without violating the proposed SLO. This experiment should clarify whether Kubernetes can divide resources in time so that Cassandra can process the bursts of 350 requests per second without violating the SLO.

Setup. The setup for this experiment is equal to the one used in Experiment 2. In this experiment, 5 minutes of a manageable workload (200 requests per second) is applied, followed by a one minute burst of 350 requests per second. This pattern is repeated 20 times.

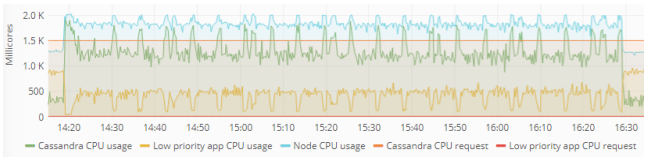


Figure 6. Second worker node CPU usage during Experiment 3.

Results. Figure 6 shows the resource utilization during this experiment. It depicts the expected behavior: during a burst, CPU cycles are taken away from the low priority application and granted to the high priority one. The low priority application is allowed to use more resources in between bursts, increasing cost-efficiency. The experiment controller reported an average 95th percentile latency of *109.3 ms* during the bursts, which is comparable to the latencies reported during the previous experiment at 350 requests per second. The type of workload, bursty or more seasonal, thus seems to have no effect on the operation of the *cpu-shares* mechanism.

4.4 Experiment 4: Determining the effects of the Kubernetes HPA on Cassandra performance

The goal of this experiment is to validate the correct performance of the HPA in combination with the application at hand, Cassandra.

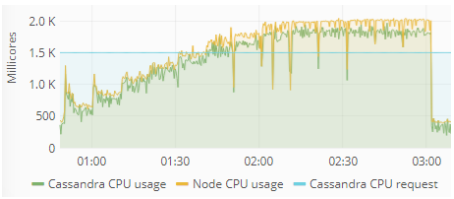
Setup. In this experiment, the HPA is added to the cluster, and the third worker node is made available to deploy a replica of the Cassandra application on. The low priority pod is removed from the cluster. The workload described in Experiment 1 will be applied to Cassandra. The HPA is configured to scale Cassandra when its CPU usage rises above 110% of its CPU request, so at around *1650 millicores*. 110% is selected as the point to scale as spinning up a new replica takes some time. Scaling when the node's resources are fully used may be too late and may thus result in SLO violations.

Results. Figure 7 shows the CPU usage of both the primary Cassandra pod and the replica added by the HPA. The graphs illustrate that a new Cassandra replica is added when scaling threshold is breached. Despite this, the load on the original Cassandra replica does not decrease when the new replica is activated. Since the experiment controller sends the workload to the Cassandra service, and this service should load balance over all available replicas, this is not in line with expectations. The 95th percentile latencies of the requests, shown in Figure 8, reflect this unexpected behavior. Instead of the latencies going down when a new replica is added, they go up significantly. The full explanation is beyond the scope of this paper. Truyen et al. [18] found that Kubernetes introduces a performance overhead when running Cassandra.

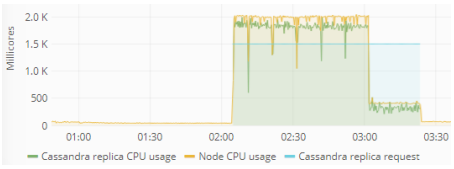
4.5 Experiment 5: Determining the effects of the Kubernetes HPA on the SaaS application performance

The previous experiment is redone with the artificial SaaS application (introduced in §3.1) replacing Cassandra. Through two tests, this experiment verifies whether the performance of the SaaS application increases after a replica is added, or if it encounters the same scalability issues in Kubernetes as Cassandra.

Setup. The SLO posed for the SaaS application is equal to the one posed for the Cassandra application in Experiment 1. Two separate tests are run. The first one subjects the SaaS application



(a) Second worker node's CPU usage



(b) Third worker node CPU usage

Figure 7. CPU usage during Experiment 4. At 110% of the CPU request, the given CPU threshold, a new Cassandra replica is added by the HPA. The CPU usage of the original replica does not decrease when the new replica is added, indicating scalability issues of Cassandra in Kubernetes.

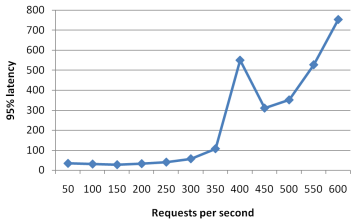


Figure 8. 95th percentile latencies during Experiment 4. The latencies increase rather than decrease when a replica is added.

to a linearly increasing workload to see how much requests one replica can handle. The SaaS application's CPU request is set to 1.5 CPU and it is subjected to a linearly increasing workload similar to the one described earlier. For the second test, the HPA is added to the cluster and linearly increasing workload is again applied to the SaaS application. Again, 110% of the request is selected as the point of scaling for the HPA.

Results. The blue and red graphs in Figure 9 show the latencies recorded during this experiment. The SaaS application is able to process 150 requests per second without the HPA. Figure 10 illustrates that at around 150 requests per second, the CPU in the node is fully used up, confirming that CPU is the bottleneck. With the HPA, the application is able to process 250 requests per second without violating the SLO. This is slightly less than double the 150 requests per second which the SaaS application can process without the HPA. Hence, there is still some overhead associated with scaling the SaaS application, but it is relatively small compared to the overhead detected when scaling Cassandra in Kubernetes. Figure 11 confirms that the workload is distributed over the replicas.

4.6 Experiment 6: Determining the effects of bursty workloads on the performance of the Kubernetes HPA

The goal of this experiment is to test how the Kubernetes HPA performs when the workload is bursty rather than linearly increasing. Even with the HPA added to the cluster, the SaaS application

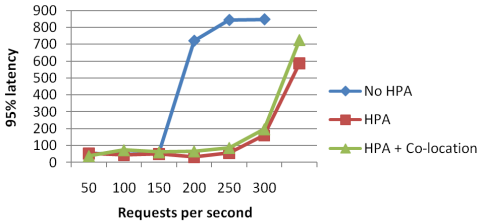


Figure 9. 95th percentile latencies during Experiments 5 and 7.

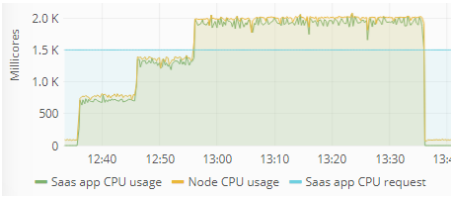
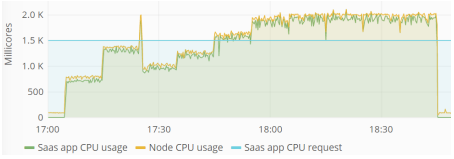
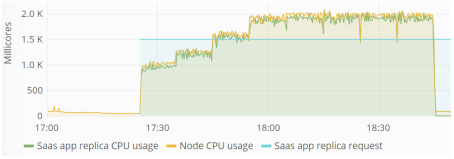


Figure 10. Second worker node CPU usage during the first test of Experiment 5.



(a) Second worker node CPU usage.



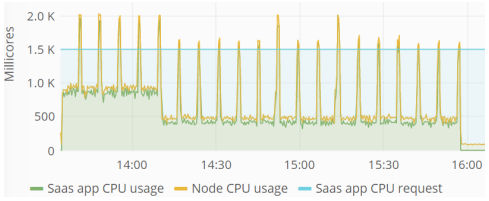
(b) Third worker node CPU usage.

Figure 11. CPU usage during the second test of Experiment 5. The CPU usage of the original replica decreases when the HPA schedules a new replica of the SaaS application.

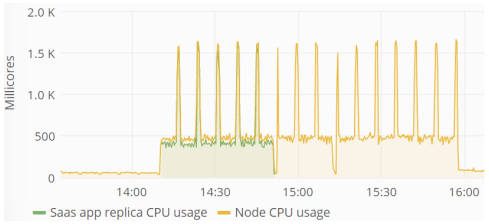
might not be able to process bursts of 250 requests per second, since starting a new replica can be slow.

Setup. The setup for this experiment is the same as the one used during the second test of the previous experiment. The bursty workload applied consists of five minutes of 60 requests per second followed by a one minute peak of 250 requests per second. This pattern is repeated 20 times.

Results. Figure 12 shows the worker node's CPU usages. The graphs illustrate that during the first couple of bursts, no scaling happens. This is unexpected since the scaling threshold is clearly breached. The latencies also report SLO violations during these bursts. One possible explanation is that the HPA does not poll the resource usage during the burst and thus does not notice the burst. This is, however, not the case, since the HPA queries the resource utilization every 15 seconds by default, and a burst lasts for 60 seconds. Another observation made from this experiment's results is that the HPA does not scale the application down after each burst.

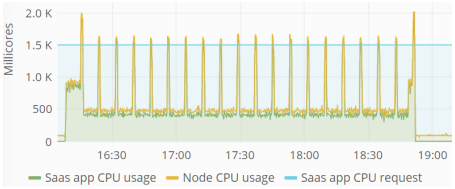


(a) Second worker node CPU usage.

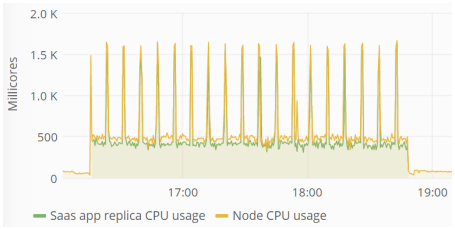


(b) Third worker node CPU usage.

Figure 12. CPU usage during the first test of Experiment 6. Replicas are added and removed inconsistently.



(a) Second worker node's CPU usage.



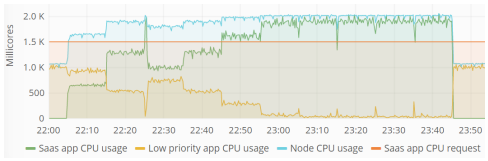
(b) Third worker node CPU usage.

Figure 13. CPU usage during the second test of Experiment 6. Increasing the time between bursts to 7 minutes, replicas are not removed inbetween bursts, which is not as expected.

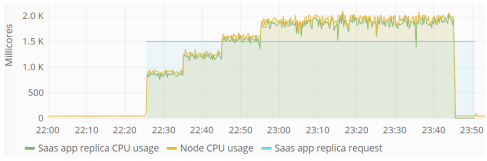
The HPA's algorithm details [15] proved insufficient to find the cause of these inconsistent scaling decisions.

It is possible that the HPA does not scale down the SaaS application since the default *downscale stabilization* parameter of the HPA algorithm being five minutes [16]. This parameter specifies a period of time during which the HPA considers all recommendations before scaling down. In other words, the HPA only scales down if its decision to scale down has not changed for five minutes. To verify this, another test is run where the downtime between each burst is set to 7 minutes. The other experiment parameters remain unchanged. Figure 13 shows the CPU usages during this test. The HPA scaled up the SaaS application during the first burst, but did not scale it down until after the test was completed.

Neither these experimental results nor the official documentation about the HPA [15] clarify how the HPA decides when to scale an application. Furthermore, the Kubernetes design proposals [7], do not list any issues relating to scaling down. Discovering the cause



(a) Second worker node's CPU usage.



(b) Third worker node's CPU usage.

Figure 14. CPU usage during Experiment 7. A new replica is added when the scaling threshold is breached, and the freed up resources are made available to the low priority application.

of this unexpected behavior is left for future work. Kubernetes is, however, evolving rapidly, so this may be fixed in the future.

4.7 Experiment 7: Determining the effects of combining the Kubernetes HPA with the presence of a low priority pod

Experiment 2 illustrated that co-locating a high and low priority pod has a minor impact on the high priority application's performance, while in increases the overall cost-efficiency. The goal of this experiment is to test whether this is still the case when the HPA is added to the cluster.

Setup. The scaling point for the SaaS application is again set to 110%. The application is co-located with the low priority application and subjected to the linearly increasing workload described earlier.

Results. The green graph in Figure 9 shows the latencies recorded during this experiment. They are only slightly higher compared to the latencies recorded during Experiment 5. This slight decrease in performance again comes with the benefit of a higher resource utilization, as illustrated by Figure 14a. The low priority pod is able to use the excess of resources on the node during low workloads. As the workload rises, the low priority pod is given access to less CPU cycles. When a new replica of the SaaS application is added to the cluster, resources are freed up for the low priority pod to use.

4.8 Threats to validity

Some of the conclusions drawn from the experiments results may only be valid for the specific setup used in this paper. The experiments considered an environment with two user applications: one high priority application and one low priority application. Setting suitable requests and limits can, however, become a complex task if multiple pods with each different priorities are scheduled on the same node.

The applications used can also impact the experiment results. The artificial SaaS application has a very short start-up time, making it well suited to be horizontally scaled. This may not be the case for other applications. Furthermore, this paper assumed that empty nodes were available for new replicas to be scheduled on. In practice, nodes may need to be acquired from IaaS providers and configured to the specific environment before they are ready to host applications. This could further increase the start-up time of new replicas.

5 Conclusion

This paper described the effects of different resource management mechanisms offered by Kubernetes, namely resource allocation via request and limit configurations and the Horizontal Pod Autoscaler. In environments with a small amount of applications, experiments show that choosing proper request configurations increases cost-efficiency without major drawbacks. This was verified for a Cassandra based application and for an artificial SaaS application, as well as for both seasonal and bursty workloads. Due to an overhead introduced by running Cassandra on Kubernetes, scaling Cassandra in Kubernetes decreases performance instead of increasing it, regardless of the scaling algorithm used. The HPA performs well for an artificial SaaS application if the workload is seasonal, even if pods are co-located. For bursty workloads, other approaches may be preferred. In conclusion, despite some limitations, the scaling capabilities of Kubernetes show great potential to prevent SLA violations and increase resource cost-efficiency in container-centric environments.

References

[1] M. Ahmadi and N. Aslani. Capabilities and advantages of cloud computing in the implementation of electronic health record. *Acta Informatica Medica*, 26(1):24–28, 2018. Accessed: 2018-11-19.
[2] Wito Delnat and Eddy Truyen. K8-scalar. https://github.com/k8-scalar/k8-scalar/blob/master/docs/overview.md. Accessed: 2019-08-02.
[3] Wito Delnat, Eddy Truyen, Ansar Rafique, and Joosen Wouter Van Landuyt, Dimitri. K8-scalar: A workbench to compare autoscalers for container-orchestrated database clusters. In *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2018.
[4] Umesh Deshpande. Caravel: Burst tolerant scheduling for containerized stateful applications. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, November 2019.
[5] Cloud Native Computing Foundation. Creating a single master cluster with kubeadm. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/. Accessed: 2019-06-13.
[6] Cloud Native Computing Foundation. In-place Update of Pod Resources. https://github.com/kubernetes/enhancements/blob/29a22b61241b35bb280de83edc0aee40d1bd87bf/keps/sig-autoscaling/20181106-in-place-update-of-pod-resources.md. Accessed: 2019-09-09.
[7] Cloud Native Computing Foundation. Kubernetes design documents and proposals. https://github.com/kubernetes/community/tree/master/contributors/design-proposals/. Accessed: 2019-10-09.
[8] Cloud Native Computing Foundation. Kubernetes Github page. https://github.com/kubernetes/kubernetes/. Accessed: 2018-11-19.
[9] Cloud Native Computing Foundation. What is a Pod? https://kubernetes.io/docs/concepts/workloads/pods/pod/. Accessed: 2019-03-08.
[10] OpenStack Foundation. https://www.openstack.org/. Accessed: 2019-08-13.
[11] Google.com. Kubernetes best practices: Resource requests and limits. https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-resource-requests-and-limits. Accessed: 2019-03-08.
[12] InfluxData. Influxdb. https://www.influxdata.com/. Accessed: 2019-08-02.
[13] Andrë Jacobs. Haalbaarheidsstudie van container orchestratie voor performantie-isolatie in multi-tenant saas-applicaties, 2017.
[14] Kubernetes. Heapster. https://github.com/kubernetes-retired/heapster/. Accessed: 2019-08-02.
[15] Kubernetes. Horizontal pod autoscaler: Algorithm details. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#algorithm-details. Accessed: 2019-08-12.
[16] Kubernetes. Horizontal pod autoscaler: Support for cooldown/delay. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#support-for-cooldown-delay. Accessed: 2019-08-12.
[17] Grafana Labs. Grafana. https://grafana.com/. Accessed: 2019-08-02.
[18] Eddy Truyen, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. Performance overhead of container orchestration frameworks for management of multi-tenant database deployments. volume Part F147772. ACM, 2019.
[19] Jonathon Paul Wong, Anthony Kwan, and Hans-Arno Jacobsen. Hyscale: Hybrid scaling of dockerized microservices architectures. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019.