Leveraging Kubernetes for adaptive and cost-efficient resource management

Stef Verreydt, Emad Heydari Beni, Eddy Truyen, Bert Lagaisse, Wouter Joosen
imec-DistriNet, KU Leuven
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 summary, providing suitable resource configurations increases cost-efficiency without any major downsides. The effects of using the default Kubernetes horizontal autoscaler, however, depend on the type of application and the user workload at hand.

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

1 Introduction

1.1 Context

Cloud computing has become increasingly popular since its introduction and is expected to keep growing in the years to come [19]. 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, in order to reduce the cost. One way to reduce the amount of resources needed is by locating multiple applications on the same node. This entails new challenges, such as 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 [6]. 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 [7]. Containers belonging to the same application are grouped together in pods. The resources used by a container can be limited, for example, 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 [9]. When there is resource contention in a node, the pods with the lowest requests will be throttled or evicted first. 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 requires to reschedule Pods when dynamically adjusting requests or limits[1].

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 (i.e. [22] [4]). As a result, identifying the optimal way to manage resources in Kubernetes can be a difficult task. Multiple questions may arise when configuring a Kubernetes cluster, such as 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.

This paper evaluates the effects of simulating vertical scaling through the use of requests and limits and co-locating high-priority and low-priority pods, horizontal scaling using the default Kubernetes HPA, and a combination of both, in different environments. A test application deployed on a Kubernetes cluster is subjected to multiple experiments to illustrate the impact of different configurations.

- **Requests and limits** The impact of different request and limit configurations is examined by deploying another, low priority pod on the same node as the main application's pod. This impact is described by providing answers to two questions: first, how are resources divided among these pods; second, what is the impact of the co-location on the performance of the higher priority pod?
- **The Kubernetes HPA** If the user load rises to a point where SLAs get violated even if lower priority pods get throttled or evicted, applications may need to be scaled to provide the desired services. The performance impact of adding the default Kubernetes horizontal autoscaler to a cluster is illustrated. Furthermore, the effects of combining the autoscaler with the request and limit mechanisms, are described.

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

[1] although there is a plan to resolve this issue

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. Scaling memory bound systems is less common as performing a static optimization of needed memory often suffices. Furthermore, Kubernetes does not offer support for configuring network and disk resources.

The remainder of the paper is structured as follows. Section 2 discusses relevant related work on resource management in cloud environments. Section 3 provides an overview of the used test environment, focusing on the cluster setup and the used test applications. In Section 4, the experiments are described and their results are show. Finally, Section 5 presents a conclusion and possible starting points for future work.

2 Related work

This section describes recent work relevant to this paper.

Caravel [4] is a scheduler developed by Deshpande et al. 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 and 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].

Deshpande et al. developed an eviction algorithm that addresses these concerns by letting stateful containers evict stateless containers as these can be restarted in only a few seconds on another node. However, each stateful application can only evict a limited number of containers, limiting the effects of large bursts. The total amount of resources acquired simultaneously by all stateful applications is also limited to prevent them from overwhelming the cluster. Finally, future evictions are spread fairly across applications. This approach thus tolerates bursty behavior of stateful applications and reduces their evictions by preferring to evict stateless containers. It also imposes a mechanism to control excessive evictions of stateless applications. This paper explores 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. [22] describes the (dis)advantages of vertical and horizontal scaling and proposes a hybrid solution. The solution is 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 noted by Wong is that the default Kubernetes scaling solutions only consider one resource when making scaling decisions. Second, Wong explains how the Kubernetes autoscalers often lead to sub-optimal configurations. The HyScale 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 [22].

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" [22]. When comparing the HyScale 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.

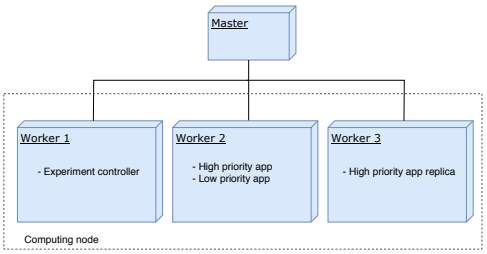


Figure 1. Cluster setup

3 Test Environment

This section describes the specific tools used in the experiments. It starts with presenting the deployed Kubernetes cluster's layout. Then, the tools needed to run the actual experiments are introduced. Finally, this section concludes with a brief overview of the three test applications used

3.1 Kubernetes cluster

All the experiments are run on a Kubernetes cluster deployed on the DistriNet private cloud, which is based on the OpenStask platform [8]. 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.

3.2 Experimentation tools

Running the experiments requires various tools to send a customizable load to the test applications, as well as to monitor the application's performance and the resource usage of all the nodes and pods.

Listing 1. Load profile example

```
## 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=500,1000,1500
user_warmup_fraction=1
user_warmup_duration=0
user_ramp_up_duration=0
user_peak_duration=400
user_ramp_down_duration=0
user_cooldown_duration=0
user_wait_inbetween_runs=0
```

3.2.1 Resource monitoring

To monitor the resources in the cluster, a combination of Heapster [14], InfluxDB [11] and Grafana [18] is employed. Heapster collects each node's metrics through Kubelet, which is the primary Kubernetes node agent running on each worker node [17] [20]. The collected data is written to InfluxDB. InfluxDB is a time-series database, a database optimized for time-stamped or time series data [12]. Grafana is a metric dashboard and graph editor which supports, among others, InfluxDB [10].

3.2.2 Experiment controller

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]. K8-Scalar can be customized to test a specific system by implementing Java user and request classes fit for that system. Multiple users may be implemented, for example one user which performs CPU-intensive requests and one which performs memory-intensive requests.

When preparing an experiment, the distribution of the types of users can be specified, for example 95% of the users executing memory intensive requests and 5% executing CPU intensive requests. Furthermore, the exact workload profile can be described using a template, part of which is shown in listing 1. An experiment in K8-Scalar consists of multiple runs, and the peak load for each of these runs must be provided. A run in turn consists of a ramp-up phase, a peak load phase and a ramp-down phase. The duration of all of these phases can be configured. In between runs, there can also be constant low load phases. Ramping up and down is done linearly. Besides sending the requests, K8-Scalar monitors the time needed by the tested applications to process these requests [3].

3.3 Deployed applications

3.3.1 LMaaS: Log Management as a Service

In accordance with the K8-Scalar paper [3], a Log Management-as-a-Service (LMaaS) application based on Cassandra is selected as the first high priority application with which to evaluate the Kubernetes mechanics in a CPU intensive environment. The main service offered by the application is real-time aggregation and storage of log files. 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 well-suited as for the experiments as its design is optimized for write-heavy workloads. [3].

3.3.2 Artificial SaaS application

A second high priority application is also experimented with to to verify whether the experimental results are valid for more than just one application. An artificial SaaS application developed at KU Leuven [13] is selected for this purpose. 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.

3.3.3 Low priority application

The application shown in Figure 2 is deployed as the low priority application when testing the effects of co-locating pods. It uses up CPU cycles by constantly executing a multiplication. If sufficient resources are free, this application continually uses one full CPU since it is single threaded. The main benefit of this application is that its exact CPU usage is known and roughly constant.

```
if __name__ == '__main__':
    while True:
        3*3
```

Figure 2. Low priority application Python code

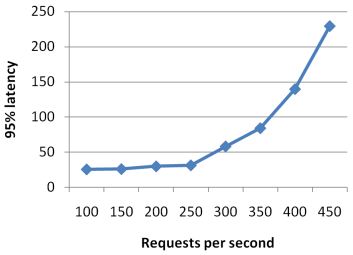


Figure 4. 95th percentile latencies during Experiment 1. At around 400 requests per second, the proposed SLO is violated.

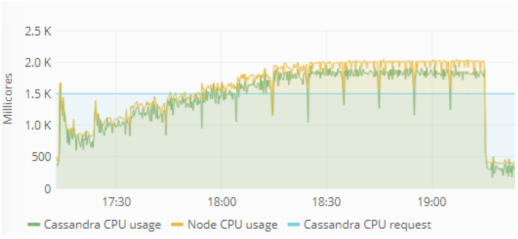


Figure 5. Worker node CPU usage during Experiment 1. The CPU usage of Cassandra rises above its CPU request, illustrating that the application is able to use all of the overprovisioned resources.

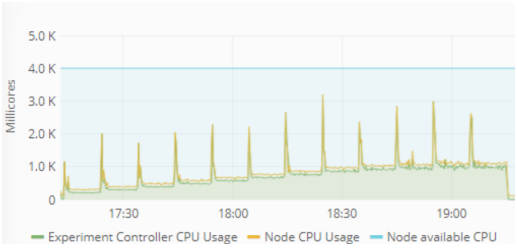


Figure 6. Experiment controller CPU usage during Experiment 1. The node's CPU usage never reaches its capacity.

4 Results

In this section, the Kubernetes mechanisms for resource management are evaluated through a series of experiments. The first experiment aims to demonstrate the effects requests and limits. In the second experiment, a pod is added to the cluster and the effects of different request and limit configurations are evaluated. The impact of bursty workloads is described in the third experiment. The remainder of the experiments describe the effects of adding an HPA to the cluster.

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. The goal of this experiment is to validate this hypopaper using Cassandra.

4.1.1 Setup

Before performing any experiments, the expected performance of the Cassandra application needs to be defined. This can be 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.*

Figure 3 shows the high-level setup for the first experiment. Kubernetes-specific pods are not shown for readability reasons. The Cassandra application is deployed with a CPU and memory request of respectively *1500m* and *2GiB*. No limits are set. In this experiment, the experiment controller 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 run. Each run lasts for 600 seconds. Heapster monitors the resource usage and the experiment controller records the latency of each request.

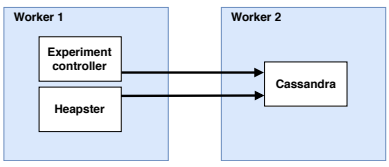


Figure 3. Experiment 1 setup. The experiment controller sends requests to the Cassandra application and monitors their latencies. Heapster monitors the resource usage of the nodes and pods in the cluster.

4.1.2 Results

Figure 4 shows the 95th percentile latencies of the requests. At around 400 requests per second, they exceed 150ms, violating the SLO. Figure 5 shows the CPU usage of the Cassandra application and the node on which it is deployed. 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 shows that the bottleneck is indeed the CPU. Furthermore, Figure 5 shows that the Cassandra application is able to use almost all of the overprovisioned resources. This in in line with expectations, as the resources on the node are only contended by the Cassandra application.

It should be verified that the experiment controller has sufficient resources to run the experiments. Preliminary tests show that the experiment-controller is CPU bound. Figure 6 shows the experiment controller's CPU usage during the experiment described above. A peak in CPU usage can be examined at the start of each run, but these peaks stay well below the available amount of CPU in the node. The experiment controller should thus not be the bottleneck during the experiments.

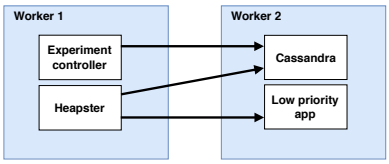


Figure 7. Experiment 2 setup. A low priority application is added to the cluster.

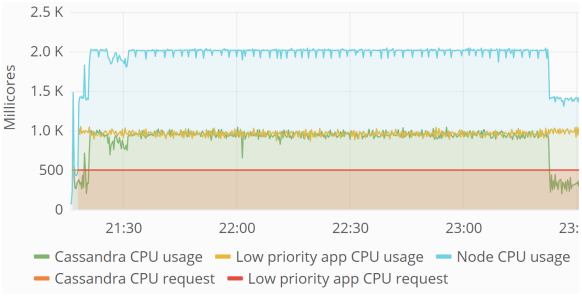


Figure 8. Worker node CPU usage during the first test of Experiment 2. The deployed applications have an equal CPU request, resulting in them receiving an equal share of the available CPU.

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 section **??**. Through two tests, we illustrate the effects of different request configurations when a high and low priority pod are co-located on a node.

4.2.1 Setup

Figure 7 shows the high-level setup. Compared to the first experiment, the application described in section 3.3.3 is added to the cluster as a low priority pod. The experiment consists of two separate tests. During the first one, the Cassandra application 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 as a result of the aforementioned *cpu-shares* mechanism.

4.2.2 Results

Figure 8 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 9 illustrates that the SLO posed earlier gets violated at about half the amount of requests per second when compared to Experiment 1.

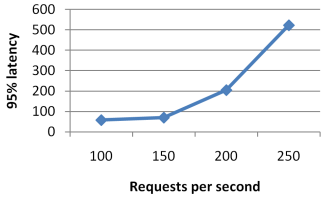


Figure 9. 95th percentile latencies during the first test of Experiment 2. The proposed SLO is violated at around 200 requests per second. Compared to Experiment 1, the Cassandra application is able to process only half the amount of requests per second as a result of it only having access to half of the resources.

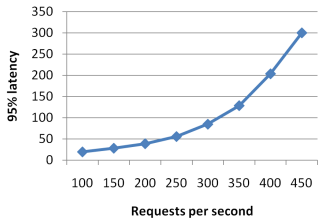


Figure 10. 95th percentile latencies during the second test of Experiment 2. The latencies are comparable to the ones examined during Experiment 1, showing that performance is only slightly impacted.

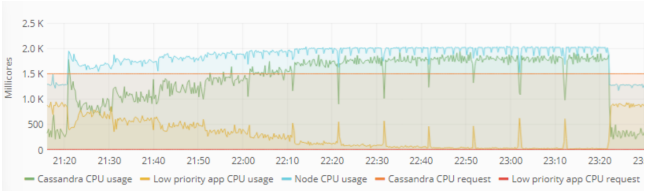


Figure 11. Worker node CPU usage during the second test of Experiment 2. As the Cassandra workload rises, the amount of CPU cycles granted to the low priority application decreases and vice versa. This increases cost-efficiency without heavily impacting the amount of CPU available to the Cassandra application.

During the second test, the Cassandra pod should receive the vast majority of the available CPU when it needs to process a high workload. Figure 11 shows the worker node's CPU usage during this test. 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 are shown in Figure 10. The proposed SLO is breached between 350 and 400 requests per second. This is slightly lower when compared to the Experiment 1, where SLO violations started occurring at about 400 requests per second.

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, in Figure 11, the latter shows a significantly higher resource utilization when the workload of the Cassandra pod is low.

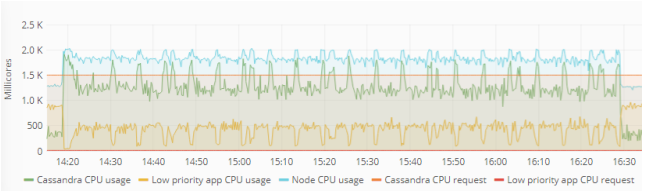


Figure 12. Worker node CPU usage during Experiment 3. During a burst, CPU cycles are taken away from the low priority application and granted to the high priority one. The low priority application is allowed to use more resources in between bursts, increasing cost-efficiency.

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 was able to process 350 requests per second without violating the proposed SLO. This experiment should clarify whether Kubernetes is able to divide resources in time so that Cassandra is able process the bursts of 350 requests per second without violating the SLO.

4.3.1 Setup

The setup for this experiment is equal to the one used in Experiment 2. In this experiment, K8-Scalar applies 5 minutes of a manageable workload (200 requests per second) followed by a one minute burst of 350 requests per second. This pattern is repeated 20 times.

4.3.2 Results

Figure 12 shows the node's resource utilization during this experiment. It depicts the expected behavior: during bursts in the workload, Cassandra's CPU usage increases rapidly, while the low priority pod's CPU usage changes inversely. 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. Futhermore, adding a low priority pod increases cost-efficiency in between bursts.

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

When the resources available to an applications are insufficient for it to meet its QoS requirements, replicas may need to be added. An application should be able to handle significantly more workload if it is able to scale out. The goal of this experiment is to validate the correct performance of the HPA in combination with the application at hand, Cassandra.

4.4.1 Setup

In this experiment, the HPA is added to the cluster, as well as another worker node on which the HPA can deploy a replica of the Cassandra application. The low priority pod is removed from the cluster. The workload described in Experiment 1 will be applied to Cassandra: an increasing amount of requests per second starting at 100 requests per second up to 600 requests per second, increasing with 50 requests per run. Each run again lasts for 600 seconds. 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 SLA violations. The experiment controller divides the workload among all active replicas by sending the requests to the Cassandra application service, which takes care of the load balancing. Heapster automatically monitors all new replicas.

4.4.2 Results

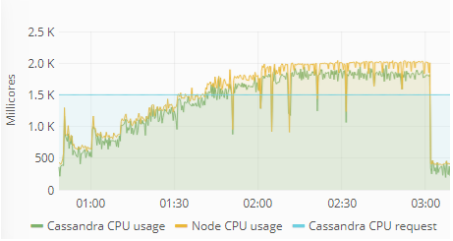
Figure 13 shows the CPU usage of the primary Cassandra pod and the replica added by the HPA. The graphs indeed illustrate that a new Cassandra replica is added when the original one uses approximately *1650 millicores*. Despite this, the load on the original Cassandra replica does not decrease when the 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 CPU usage of the original replica should decrease when a new replica is available. The 95th percentile latencies of the requests, depicted in Figure 14, reflect this unexpected behavior. Instead of the latencies going down when a new replica is added, they go up significantly. This can partially be explained by the fact that there is a certain warm-up associated with new Cassandra replicas: the first requests sent to a new Cassandra replica have higher latencies. This short-lived warm-up period explains the peak in latencies around 400 requests per second (the point of scaling), but does not justify the whole picture. The full explanation is beyond the scope of this paper but in short, Truyen et al. [21] found that Kubernetes introduces a performance overhead when running Cassandra. Additional experiments performed during this paper illustrated that this performance overhead increases significantly when more replicas are added. Scaling Cassandra in a native deployment, without using Kubernetes, did not cause major performance losses. In conclusion, the overhead was shown not to be caused by the HPA algorithm, but by Kubernetes itself.

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

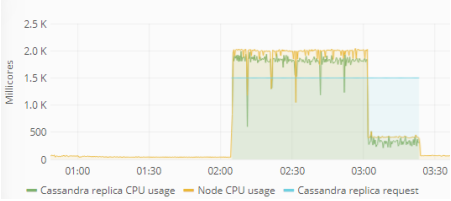
The previous experiment is redone with the artificial SaaS application (introduced in section 3.3.2) replacing Cassandra. Through two tests, this experiment verifies whether the performance of the SaaS application increases after a replica is added, or if it encounters the same scalability issues in Kubernetes as Cassandra.

4.5.1 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 to a linearly increasing workload to see how much requests one replica can handle. The SaaS application is configured so that it processes the requests in a CPU intensive manner. Its CPU request is set to 1.5 CPU. The setup is depicted in Figure 15. The linearly increasing workload applied to the SaaS application starts at 50 requests per second and increases with 50 requests per second each 600 seconds, up to 300 requests per second. For the second test, the HPA is added to the cluster and linearly increasing workload is again applied to



(a) First worker node's CPU usage



(b) Second worker node CPU usage

Figure 13. 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.

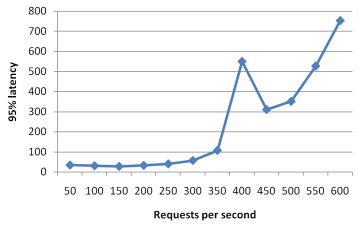


Figure 14. 95th percentile latencies during Experiment 4. The latencies increase rather than decrease when a replica is added, confirming the scalability issues of Cassandra in Kubernetes.

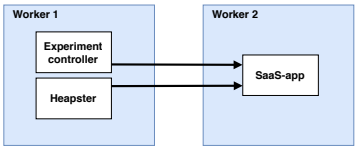


Figure 15. Experiment 5.1 setup. The SaaS application replaces Cassandra.

the SaaS application. The workload now rises up to 600 requests per second instead of up to 300. Again, 110% is selected as the point of scaling for the HPA. A new worker node is made available for the HPA to schedule a new replica of the SaaS application on, as shown in Figure 16.

4.5.2 Results

Figure 17 shows the 95th percentile of latencies during the first test. The SaaS application is able to process 150 requests per second

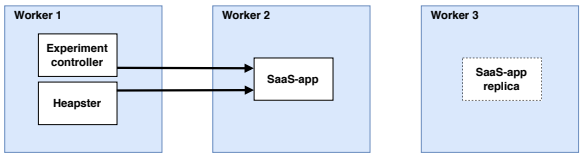


Figure 16. Experiment 5.2 setup. An extra worker node is made available for the HPA to scale a replica of the SaaS application on.

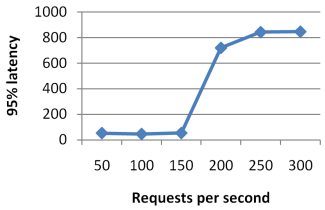


Figure 17. 95th percentile latencies during the first test of Experiment 5. The application is able to process 150 requests per second without violating the SLO, but not 200 requests per second.

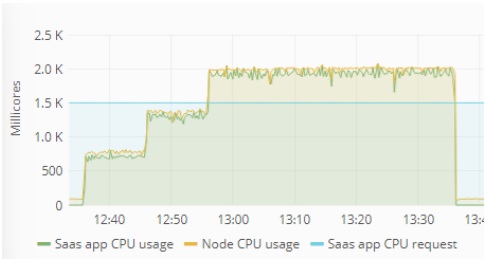


Figure 18. Worker node CPU usage during the first test of Experiment 5. The resources of the node are fully used up at 150 requests per second. Since the SaaS application cannot acquire the resources needed to process 200 requests per second, the SLO gets violated.

without violating the SLO. Figure 18 illustrates that at around 150 requests per second, the CPU in the node is fully used up, confirming that CPU is the bottleneck.

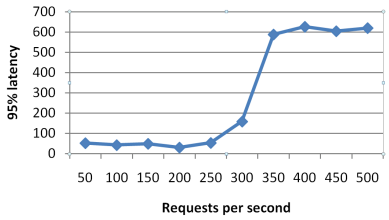
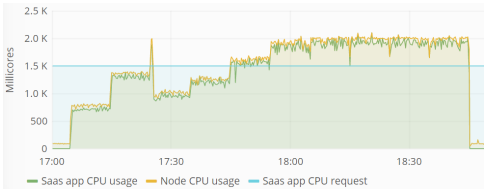
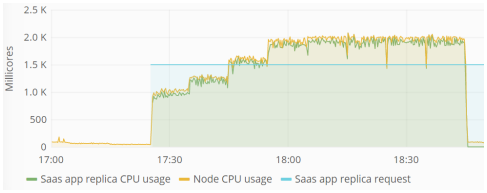


Figure 19. 95th percentile latencies during the second test of Experiment 5. The SaaS application can process a significantly higher amount of requests per second if an extra replica is added.



(a) First worker node's CPU usage during the second test of Experiment 5.



(b) Second worker node's CPU usage

Figure 20. 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, confirming that the load is distributed among all replicas.

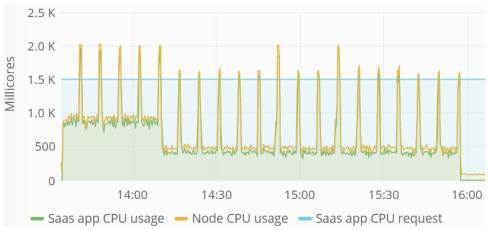
Figure 19 shows the latencies during the second test of this experiment. These illustrate that the SaaS application is able to process 250 requests per second without violating the SLO. This slightly less than double the 150 requests per second which the SaaS application can process without the HPA. Hence, there is still some overhead associated with having multiple replicas of the SaaS application, but it is relatively small compared to the overhead detected when scaling Cassandra in Kubernetes. Figure 20 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

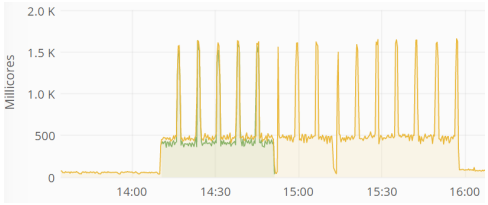
The goal of this experiment is to test how the Kubernetes HPA performs when the workload is bursty rather than linearly increasing. Even with the HPA added to the cluster, the SaaS application might not be able to process bursts of 250 requests per second, since starting a new replica can be slow. The burst could be over by the time that the replica is fully started up.

4.6.1 Setup

The setup for this experiment is the same as the one used during the second test of the previous experiment, shown in Figure 16. A bursty workload is applied to the SaaS application. This workload 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.



(a) First worker node's CPU usage.



(b) First worker node's CPU usage.

Figure 21. CPU usage during the first test of Experiment 6. The HPA does not schedule a new replica of the SaaS application until the 5th burst. After that, the replica is removed and rescheduled multiple times. When a new replica is scheduled or removed, is seemingly random.

4.6.2 Results

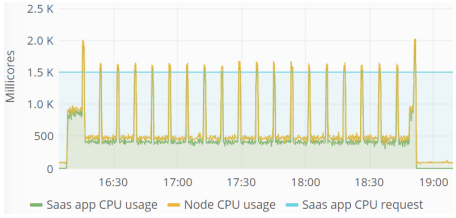
Figure 21 show the worker node's CPU usages during this experiment. The graphs illustrate that during the first couple of bursts, no scaling happens. This is unexpected. The 95th percentile latencies also report SLA violations during these bursts. One possible explanation is that the HPA does not poll 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. The HPA's algorithm details [15] proved insufficient to find the cause of this behavior.

Another observation made from this experiment's results is that the HPA does not scale the application down after each burst. The HPA calculates the desired amount of replicas as follows [15]:

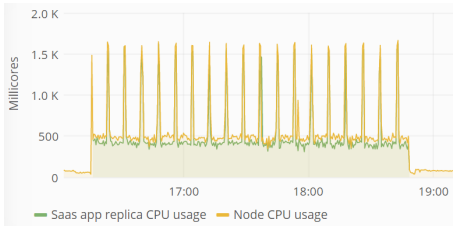
$$desiredReplicas = ceil[currentReplicas * (currentValue / desiredValue)]$$

The *desiredValue* was set to *110%* of the request, so approximately *1650m*. The *currentValue* is calculated by "taking the average of the given metric across all pods in the HorizontalPodAutoscaler's scale target" [15]. In between bursts, figures 21a and 21b show that this value is around *500m*. Given these values, the number of desired replicas is one.

It is possible that the HPA does not scale down the SaaS application due to 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



(a) First worker node's CPU usage.



(b) First worker node's CPU usage.

Figure 22. CPU usage during the second test of Experiment 6. Increasing the time between bursts results in the HPA scaling a new replica during the first burst. This replica is not removed until the test is over.

before scaling down. In other words, the HPA only scales down if its decision to scale down has not changed for five minutes. The downtime during each burst in this experiment was set to exactly five minutes. Since the HPA should not scale down during a burst, this could lead to the HPA not scaling down at all. Combined with the knowledge that the metrics get queried only every 15 seconds, this could also explain why the HPA did scale down during the 11th and 14th burst.

To verify these assumptions, another test is run in which the downtime between each burst is set to 7 minutes. The other experiment parameters remain unchanged. Figure 22 show the CPU usage of the two worker nodes. 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 algorithm. Discovering the cause of this unexpected behavior is left for future work. The results show, however, that the HPA is unfit to process bursty workloads effectively.

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.

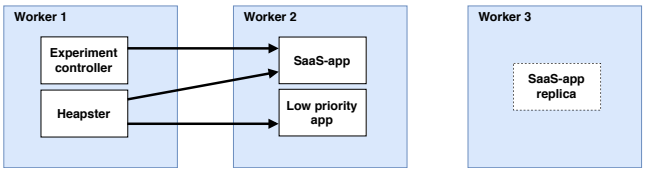


Figure 23. Experiment 6.1 setup

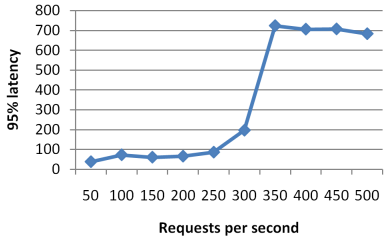


Figure 24. 95th percentile of latencies during experiment 7. They are only slightly higher than the ones recorded during Experiment 5. This indicates that adding a co-locating a high and low priority pod increases cost-efficiency, even when the high priority pod is able to scale.

4.7.1 Setup

The setup is depicted in Figure 23. The scaling point for the SaaS application HPA is again set to 110%. The application is subjected to the workload described in Experiment 5: starting at 50 requests per second and increasing with 50 requests per second each 600 seconds, up to 600 requests per second.

4.7.2 Results

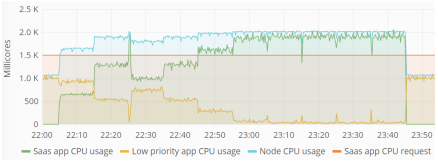
Figure 24 shows the 95th percentile latencies recorded during this experiment. They are only slightly higher compared to the latencies recorded during Experiment 5. This slight decrease in performance again comes with the benefit of a higher resource utilization during low workloads, as illustrated by Figure 25a. 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, the amount of requests which the original replica needs to process lowers, again freeing up resources for the lower priority pod to use.

5 Conclusion

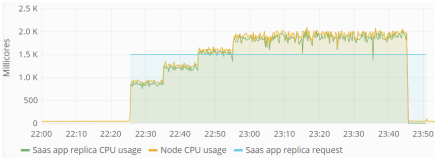
5.1 Summary of findings

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. The findings are the following.

Request and limit configurations Experiments demonstrated that request and limit configurations greatly impact applications' performances on nodes with multiple pods. Allowing pods to use overprovisioned resources when needed effectively enables them to scale vertically without having to be rescheduled. A relatively high request gives pods a better claim to overprovisioned resources, and vice versa. Applications with a low request are allowed to use



(a) First worker node's CPU usage.



(b) First worker node's CPU usage.

Figure 25. CPU usage during Experiment 7. The resource utilization of the node increases by adding a low priority pod without heavily impacting the amount of resources to which the high priority pod has access.

overprovisioned resources only if the other applications' workload is low. Co-locating low and high priority pods therefore increases cost-efficiency while only slightly impacting the performance of high priority applications, making the mechanism a useful tool for server consolidation purposes.

These findings were confirmed for two applications: one Cassandra based log management service and one artificial SaaS-application. Moreover, the findings are valid for both seasonal and bursty workloads. The available resources are redivided almost instantly when there is a burst of workload. This makes the mechanism better suited to deal with bursty workloads than most horizontal scaling techniques, as starting new replicas is usually slow.

The Kubernetes HPA Kubernetes introduces an overhead when running Cassandra. Experiments revealed that this overhead increases significantly as more replicas are added. This causes the performance of Cassandra to decrease rather than increase when scaling it in Kubernetes. The overhead is shown not to be caused by the HPA, but by Kubernetes itself.

For a seasonal workload, when scaling the artificial SaaS application using the HPA, only a minor overhead is observed. Scaling from one to two replicas causes the application's capacity to almost double. Furthermore, adding low priority pods to the node does not affect the operation of the HPA. The cost-efficiency benefit gained from setting adequate request configurations can thus also be leveraged when scaling applications using the HPA.

With bursty workloads, experiments illustrated that using the default HPA leads to unexpected results. The HPA's scaling decisions are inconsistent: sometimes it decides to scale during a burst and sometimes it does not. At the time of writing, Kubernetes' official documentation concerning the HPA does not provide sufficient explanations for this behavior.

5.2 Limitations of findings

Some of the conclusions drawn from the experiments results may only be valid for the specific setup used in this paper. The performed 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 type of test 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.

5.3 Lessons Learned

Experiments illustrated that, in environments with a small amount of applications and where low priority applications do not need any guarantees, choosing proper request configurations increases cost-efficiency without major drawbacks. 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. For bursty workloads or more complex environments, sophisticated 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.

5.4 Future Work

A first possible extension to this paper is to further evaluate the HPA for bursty workloads. The cause of the HPA performing inconsistently could possibly be discovered by examining the exact scaling algorithm rather than the documentation.

A second extension could investigate the performance impact of scaling applications with a longer start-up time. In this case, the HPA may need to be configured differently to prevent SLA violations. Utilizing merely the HPA may not be sufficient to prevent

SLA violations, however. Allowing high priority applications to use overprovisioned resources during scaling may prevent SLA violations when starting up a new replica is slow.

A third possible extension could describe the effects of using the Kubernetes mechanisms when the workload is memory intensive. In this case, low priority pods get evicted instead of throttled. The performance impacts of request and limit configurations as well as scaling could be different.

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, Dimitri Van Landuyt, and Wouter Joosen. 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)*, pages 33–39. 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. Kubernetes Github page. https://github.com/kubernetes/kubernetes/. Accessed: 2018-11-19.

[7] Cloud Native Computing Foundation. What is a Pod? https://kubernetes.io/docs/concepts/workloads/pods/pod/. Accessed: 2019-03-08.

[8] OpenStack Foundation. OpenStack. https://www.openstack.org/. Accessed: 2019-08-13.

[9] 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.

[10] InfluxData. Grafana github page. https://github.com/grafana/grafana. Accessed: 2019-08-02.

[11] InfluxData. Influxdb. https://www.influxdata.com/. Accessed: 2019-08-02.

[12] InfluxData. What is a time series database? https://www.influxdata.com/time-series-database/. 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] Kubernetes. Kubelet. https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/. Accessed: 2019-08-02.

[18] Grafana Labs. Grafana. https://grafana.com/. Accessed: 2019-08-02.

[19] Statista.com. Size of the public cloud computing services market from 2009 to 2021. https://www.statista.com/statistics/273818/global-revenue-generated-with-cloud-computing-since-2009/. Accessed: 2018-11-19.

[20] Kublr Team. How to utilize the "heapster + influxdb + grafana" stack in kubernetes for monitoring pods. https://kublr.com/blog/how-to-utilize-the-heapster-influxdb-grafana-stack-in-kubernetes-for-monitoring-pods/, 2017. Accessed: 2019-08-02.

[21] 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, pages 156–159. ACM, 2019.

[22] Jonathon Paul Wong, Anthony Kwan, and Hans-Arno Jacobsen. Hyscale: Hybrid scaling of dockerized microservices architectures. Master's thesis, University of Toronto, Toronto, ON, 2018.