An Experimental Evaluation of the Kubernetes Cluster Autoscaler in the Cloud

Mulugeta Ayalew Tamiru, Johan Tordsson, Erik Elmroth, Guillaume Pierre

An Experimental Evaluation of the Kubernetes Cluster Autoscaler in the Cloud

Mulugeta Ayalew Tamiru
Elastisys AB
Umeå, Sweden
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
mulugeta.tamiru@elastisys.com

Johan Tordsson, Erik Elmroth
Elastisys AB
Umeå, Sweden
{tordsson,elmroth}@elastisys.com

Guillaume Pierre
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
guillaume.pierre@irisa.fr

***Abstract*—Despite the abundant research in cloud autoscaling, autoscaling in Kubernetes, arguably the most popular cloud platform today, is largely unexplored. Kubernetes' Cluster Autoscaler can be configured to select nodes either from a single node pool (CA) or from multiple node pools (CA-NAP). We evaluate and compare these configurations using two representative applications and workloads on Google Kubernetes Engine (GKE). We report our results using monetary cost and standard autoscaling performance metrics (under- and over-provisioning accuracy, under- and over-provisioning timeshare, instability of elasticity and deviation from the theoretical optimal autoscaler) endorsed by the SPEC Cloud Group. We show that, overall, CA-NAP outperforms CA and that autoscaling performance depends mainly on the composition of the workload. We compare our results with those of the related work and point out further configuration tuning opportunities to improve performance and cost-saving.**

***Index Terms*—Cloud computing, autoscaling, Kubernetes.**

I. Introduction

One of the main innovations made possible by dynamic cloud resource provisioning is elasticity, where the set of compute, storage and networking resources allocated to an application can vary over time to accommodate fluctuations in the workload created by end users. The choice of the amount of resources allocated to an application is typically made by an autoscaler which dynamically adjusts the amount of resources according to user demands. Numerous autoscalers have been developed over the years to react to variations of either measured workloads (e.g., [1]) or short-term predictions of future workloads (e.g., [2]). Other autoscalers combine both reactive and proactive components (e.g., [3]–[6]).

Classical cloud platforms encourage the use of horizontal elasticity where capacity is adjusted by adding or removing identically-configured virtual machines. In the same essence, Kubernetes – the leading open-source container orchestration platform – proposes the Cluster Autoscaler (CA) that dynamically adjusts the number and size of VMs on which containers run. Like the other Kubernetes components, CA is highly configurable. In its default configuration, CA adds or removes identical nodes. However, Kubernetes recently introduced a node auto-provisioning (NAP) capability that adds nodes automatically from multiple node pools. Unlike most autoscalers from the state-of-the-art, CA-NAP allows dynamic provisioning of differently-sized nodes. This is especially useful when some pods have significantly lower or greater resource request than the rest of the pods in the workload. CA-NAP can then provision nodes that specifically match the request of these pods. Moreover, it has the potential for significant cost saving in public clouds by selecting the right size VMs to match the workload.

Although CA exposes some configurable parameters including NAP, choosing the best configuration is far from being trivial. The main objective of this paper is to address the following questions. (1) How much cost saving does CA-NAP offer as compared to CA?; (2) How do the two configurations compare with regards to autoscaling performance? (3) How do CA and CA-NAP compare with other autoscalers in the related works?

To address these questions, we conduct extensive experiments run on Google Kubernetes Engine using two representative applications with respective real-world and synthetic workloads. We provide detailed analysis of the performance of the Kubernetes Cluster Autoscaler in the two configurations using standard autoscaling performance metrics (i.e., under- and over-provisioning accuracy, under- and over-provisioning timeshare, instability of elasticity and deviation from the theoretical autoscaler) endorsed by Cloud Group of the Standard Performance Evaluation Corporation (SPEC) [7].

Our results show that even though CA-NAP outperforms CA in terms of autoscaling performance, it does not offer significant cost saving. Moreover, the autoscaling performance of CA and CA-NAP is influenced by the composition of the applications deployed on the cluster. We show the potential of further performance improvement and cost reduction by tuning additional configuration parameters of the Kubernetes cluster autoscaler.

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765452. The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

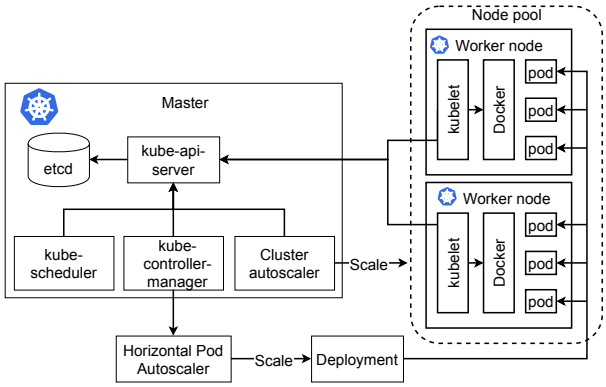


Fig. 1. Kubernetes architecture.

II. BACKGROUND

Kubernetes is an open-source container orchestration platform inspired by the Borg cluster management system from Google and later donated to the Cloud Native Foundation [8], [9]. In the last few years, Kubernetes has been adopted by enterprises for deploying applications in private data centers, public cloud, and hybrid cloud environments. Kubernetes is based on a master-worker architecture and has several components. Figure 1 shows the Kubernetes architecture.

A pod – which consists of one or more containers sharing networking and storage namespaces – is the smallest unit of execution in Kubernetes. Kubernetes also provides higher-level controllers such as Deployment, StatefulSet, and Job for managing a group of pods that belong to the same service. When a user requests *kube-api-server* to create pods on Kubernetes, *kube-scheduler* selects the most suitable node in the cluster to place the pod(s). *kube-scheduler*'s default policy is to place pods on nodes that have the most free resources while spreading out pods from the same deployment across different nodes. By doing so, *kube-scheduler* tries to balance out resource utilization of the nodes in the cluster.

Kubernetes supports autoscaling at two different levels. At the application (container) level, the Horizontal Pod Autoscaler (HPA) adjusts the number of pod instances based on CPU and memory utilization or other metrics (e.g., response time), whereas the Vertical Pod Autoscaler (VPA) adjusts the CPU and memory request of pods based on past and present resource utilization. At the infrastructure level, Kubernetes offers the Cluster Autoscaler (CA) for adding/removing nodes to/from the cluster. Its flowchart is shown in Figure 2.

CA watches *kube-api-server* periodically (by default every 10 seconds) for pods that are not scheduled due to resource shortage or other reasons. It assesses the specifications of the pods and simulates *kube-scheduler* to check whether adding more nodes to the cluster would enable placing the unscheduled pods. If so, CA adds new node(s) to the cluster.

CA also periodically checks the resource utilization of the cluster nodes. A node becomes a candidate for removal if the total sum of CPU and memory requests of its pods are less than 50% of the node's allocatable resources. The allocatable resource is defined as the amount of compute

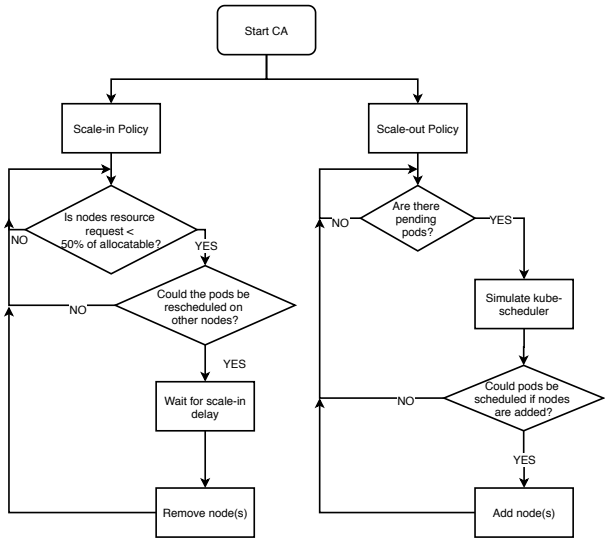


Fig. 2. Kubernetes Cluster Autoscaler (CA) algorithm flowchart.

resources available for pods, excluding resources needed for the OS and system daemons. If the pods running on the node can be rescheduled on other nodes, the node gets removed from the cluster after the scale-in time (by default 10 minutes).

One of the many configurable parameters for CA is how it manages node pools. A node pool is a set of nodes of identical size. CA, by default, adds all nodes from a single node pool, resulting in a cluster where all nodes have the same size. On the other hand, CA can be configured with node auto-provisioning (CA-NAP) which manages multiple node pools. For example, at the time of writing, CA-NAP can create node pools in Google Cloud with machines from N1 machine types with up to 64 vCPUs. CA-NAP dynamically selects the right size of the node to be added based on the resource request of the unscheduled pods. CA uses the concept of *expanders* which provides different strategies for selecting the node pool from which new nodes will be added. As a result, the cluster may have differently-sized nodes.

As traditional CPU-usage-based autoscalers offered by cloud providers do not care about pods when scaling up and down, they may add a node that does not have any pods or remove nodes that have system-critical pods on them. CA makes sure that all pods in the cluster have a place to run irrespective of CPU load. Moreover, it tries to ensure that there are no unneeded nodes in the cluster. However, for the correct functioning of CA, developers need to explicitly specify the right amount of resources for their workload. It is also important to design workloads to tolerate the transient disruptions that may result when pods are moved from one node to another during scale down.

III. RELATED WORK

Our work complements a large body of work in autoscaling. Autoscalers proposed by the industry and the research community can be categorized according to the application architecture they support, session stickiness, adaptivity to change, the scaling indicators used, the resource estimation techniques,

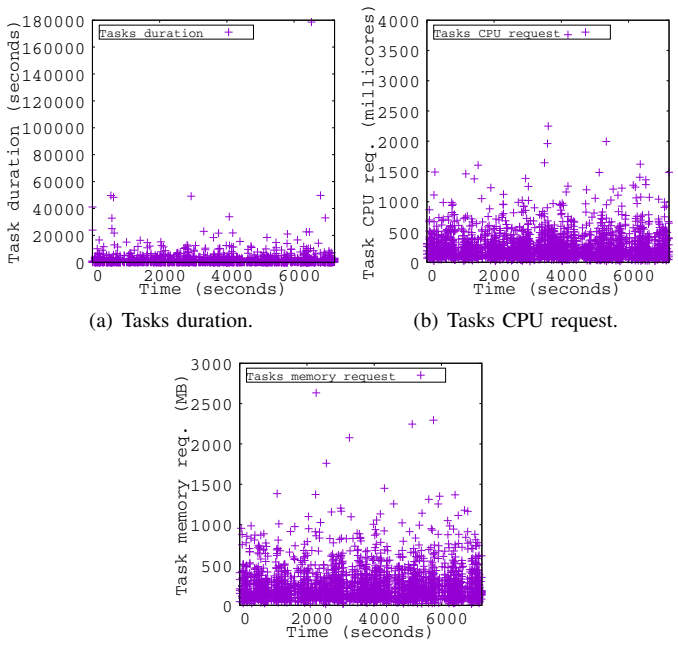
oscillation mitigation approach, scaling timing and the scaling methods they use. Autoscalers can use estimation techniques such as rule-based, fuzzy inference, application profiling, analytical modeling, machine learning and hybrid [10]. They can be classified further into horizontal and vertical according to the scaling methods they use. Autoscalers can be reactive [1], proactive [2], [11], [12] or hybrid [3]–[6] based on the timing of scaling. Finally, they differ based on the infrastructure they rely on – VMs or containers [13].

To enable comparison of novel autoscaling methods not only to static provisioning as done in the past, but also to other autoscaling algorithms, the SPEC Cloud Group developed a set of standard autoscaling performance metrics [7] that are now being used by a number of works for comparing multiple autoscalers. Ilyushkin et al. [14] use these metrics to compare seven autoscaling policies from the state of the art, whereas Versluis et al. [15] present a simulation-based experimental evaluation of autoscaling workloads of workflows in data centers. These works provide a better understanding of the performance of autoscaling policies proposed in the past decade. Our work extends these works by using similar metrics to quantify the performance of the autoscaling policy of the Kubernetes platform which is popular in the cloud-native application development paradigm. However, [14] focuses on scientific workflows while [15] focuses on scientific, industrial and engineering workflows. In contrast, we focus on two representative containerized applications for Kubernetes.

Other works employ the SPEC Cloud Group metrics to report on the performance of newly-proposed autoscalers [4], [16]–[18]. Similar to [16] and [17], we use a microservices application as one of our test applications. However, our work is focused strictly on the Kubernetes autoscaler and not on general-purpose autoscalers. Ramirez et al. [18], Podolskiy et al. [19] and Bauer et al. [16] are also concerned about autoscaling at the container level. The work in [19] is particularly interesting as it presents a comparison of various cloud providers' autoscalers with that of the Kubernetes cluster autoscaler, and the later is the most suitable for Kubernetes workloads. Similar to these works, we show that the performance of the autoscalers depends on the composition of the workloads. Moreover, we show that both CA and CA-NAP under-perform in some aspects but out-perform in others compared to the autoscalers in [4], [14]–[16].

IV. EXPERIMENTAL EVALUATION

We present our experimental setup and results from the comparison of the two strategies that Kubernetes uses for resizing the cluster, i.e., the default (CA) where nodes are provisioned only from one node pool and node auto-provisioning (CA-NAP) where nodes are provisioned from multiple node pools. In our experiments, we deploy two types of applications on Google Kubernetes Engine (GKE) and we use elasticity metrics from SPEC Cloud Group and cost for comparison.



(a) Tasks duration. (b) Tasks CPU request.

(c) Tasks memory request.

Fig. 3. Characteristics of workload used in E1, which is based on Google cluster traces.

A. Applications and Workloads

We designed two sets of experiments *E1* and *E2* which differ in the deployed applications and the corresponding workloads.

In *E1*, the application is composed of Deployments and Jobs based on a subset of the tasks data set in the widely studied Google cluster traces[1] [20]–[22]. Google used Linux containers to execute the tasks on a collection of physical machines. The characteristics of our workload can be seen in Figure 3 and a summary of the statistics is presented in Table I. Since the original traces contain 29-days-long data, we analyzed and fitted appropriate statistical distributions on the task duration, CPU request and memory request datasets separately, and generated a 2-hour-long workload. The workload contains a total of 2,467 tasks of different durations. We create long-running tasks as *nginx* web server Deployments and short running tasks as Jobs and we set the corresponding CPU and memory requests as per the *CPU* and *memory* request of the tasks in the workload traces.

In *E2*, we use a microservices-based test and reference application – TeaStore [17] – composed of six services. We chose this application as a representative because Kubernetes is preferred to run microservices as it makes it easy to run the loosely coupled, self-contained components using containers and also provides abstractions for service discovery. We use a synthetic workload created on Apache JMeter with increasing and decreasing load intensity with up to 1000 concurrent user threads emulating users browsing the application using a profile which comes with TeaStore[2].

[1]Google Cluster Traces – https://github.com/google/cluster-data
[2]TeaStore Testing and Benchmarking – https://bit.ly/38OuRHD

TABLE I
WORKLOAD CHARACTERISTICS FOR E1.

	min	max	mean	std. err.
Duration (seconds)	0.97	178,600.77	1,451.52	99.77
CPU req. (millicores)	15.00	3,760.00	258.41	4.73
Memory req. (MB)	13.00	2,633.00	256.83	4.46

B. Experiment Setup

The experiment setup, whose details are shown in Table II, consists of the application, Kubernetes cluster with cluster autoscaling enabled, and a workload generator. Under each major class of experiment, we perform 6 experiments, each repeated 3 times, by varying the autoscaling configuration (*CA* or *CA-NAP*) and the size of the worker nodes (*small (4 vCPUs, 15 GB RAM)*, *medium (8 vCPUs, 30 GB RAM)* or *large (16 vCPUs, 60 GB RAM)*). All experiments run on Kubernetes version 1.14.7-gke.14 in Google Kubernetes Engine (GKE) in europe-west4-a region. CA is enabled for all clusters (default configuration for CA and with node auto-provisioning parameter enabled for CA-NAP). In all experiments, we inject the workload from VMs in Google Cloud in the same region but separate from the Kubernetes clusters.

In *E1*, all experiments start with only one worker node and the cluster autoscaler adds/removes nodes to/from the cluster in response to workload changes. For each experiment in *E1*, we inject the workload for two hours and wait for an additional 30 minutes to observe scale-in.

For each experiment in *E2*, we start with different number of worker nodes as follows, to have just enough resources to place all six Deployments in the application, – 4 in *Scenario 1*, 2 in *Scenario 2* and 1 in *Scenario 3*. We deploy TeaStore services on Kubernetes using the Deployment manifest provided by the developers[3]. To automatically scale the Deployments in response to workload changes, we enable Horizontal Pod Autoscaling (HPA) for all six Deployments. The details of the configuration of the Deployments and HPA can be seen in Table III. We access the application using the IP address exposed by the LoadBalancer Service of the WebUI Deployment. For each of the six experiments we run the workload for 1 hour and wait an additional 30 minutes to observe scale-in.

C. Evaluation Metrics

To assess the performance of the Kubernetes autoscaler we use some of the autoscaling performance metrics proposed by SPEC Cloud Group [7]. These metrics quantify the autoscaling capabilities of the two Kubernetes autoscaling strategies and help the developer community to select the appropriate strategy for their workload. The provisioning accuracy metrics θ_U and θ_O describe the relative amount of under-provisioned or over-provisioned resources, respectively, during the measurement interval. The wrong-provisioning timeshare metrics τ_U and τ_O measure the time in which the autoscaler under-provisions or over-provisions, respectively, during the time of the experiment. The instability of elasticity metric υ measures the fraction of time in which the demand and the supply change in different directions. The autoscaling deviation σ measures the deviation of a given autoscaler compared to the theoretically optimal autoscaler, that does not exist but is assumed to supply exactly the resources demanded by the workload. We calculate these metrics for the total CPU and memory demanded by our workload and supplied by the CA or CA-NAP in the different scenarios of our experiments. For each of these metrics, the smaller the value is the better the autoscaler performs for that metric.

The metrics and the equations used to calculate them are summarized in Table IV. Here we define: (i) T as the experiment duration and the current time as $t \in [0, T]$, (ii) s_t as the total amount of CPU cores or memory supplied by the cluster at time t, (iii) d_t as the total amount of CPU cores or memory demanded by the pods of the application at time t, and (iv) sgn is the signum function. Finally, Δt denotes the time interval between the last and the current change either in demand d or supply s.

In addition to the autoscaling performance metrics, we also calculate the hourly cost of running the clusters use it to compare the autoscaling policies.

D. Experimental Results

The following are the main findings of our extensive experiments on the autoscaling performance of CA and CA-NAP.

1) **Overall, CA-NAP outperforms CA**, as it provisions differently-sized nodes to match the demand of the workload better.
2) Contrary to our expectations, **CA-NAP does not offer significant cost saving compared to CA.**
3) **The performance of CA-NAP is influenced mainly by the composition of the workload**, performing better for workloads made up of several short- and long-running pods with diverse resource requests.
4) **CA and CA-NAP show worse over-provisioning but better under-provisioning accuracy and under-provisioning timeshare** than the autoscalers studied in the **related works** [4], [14]–[16].
5) **CA and CA-NAP could offer even better performance** if the other configuration parameters such as autoscaling interval, scale-in time and expander are **tuned properly.**

The detailed discussion of our results follows.

1) Autoscaling Dynamics: In figures 4 and 5, we present the total CPU and memory demand of the application pods and the total CPU and memory supplied by the autoscaling strategies in each experiment scenario. For the sake of brevity, we present the plots from only one of the three runs of each experiment. Unlike some of the related works [4], [14], [16], we cannot use the number of VMs for comparing CA and CA-NAP as the latter supplies nodes of different sizes. Instead, we report the CPU and memory demand and supply.

The slight CPU over-provisioning by CA-NAP in Figures 4(c) and 4(e) can be explained by the fact that CA-NAP supplies more smaller nodes during scale-out than CA as shown in Table V, which means more resource overhead

[3]Run TeaStore on Kubernetes – https://bit.ly/36w4K6M

TABLE II
DETAILS OF EXPERIMENTS SETUP.

Exp.	Scen-ario	AS Config	App	Workload	Node type	Starting no. of nodes	Autoscaler configuration			
							min no. of nodes	max no. of nodes	Max memory GB)	Max CPU
E1	1	CA	nginx Deployments and Jobs based on Google cluster traces	Based on Google cluster traces	small	1	1	100	-	-
		CA-NAP							1500	400
	2	CA			medium	1	1	100	-	-
		CA-NAP							1500	400
	3	CA			large	1	1	100	-	-
		CA-NAP							1500	400
E2	1	CA	Teastore	Synthetic with increasing and decreasing load intensity	small	4	4	100	-	-
		CA-NAP							1500	400
	2	CA			medium	2	2	100	-	-
		CA-NAP							1500	400
	3	CA			large	1	1	100	-	-
		CA-NAP							1500	400

TABLE III
PODS AND HPA CONFIGURATION IN EXPERIMENTS E2.

pod configuration				HPA configuration			
Request		Limit		Scaling metric	Threshold	min no. of pods	max no. of pods
CPU (cores)	Memory (MB)	CPU (cores)	Memory (GB)				
0.5	1024	0.5	1024	CPU	50%	1	100

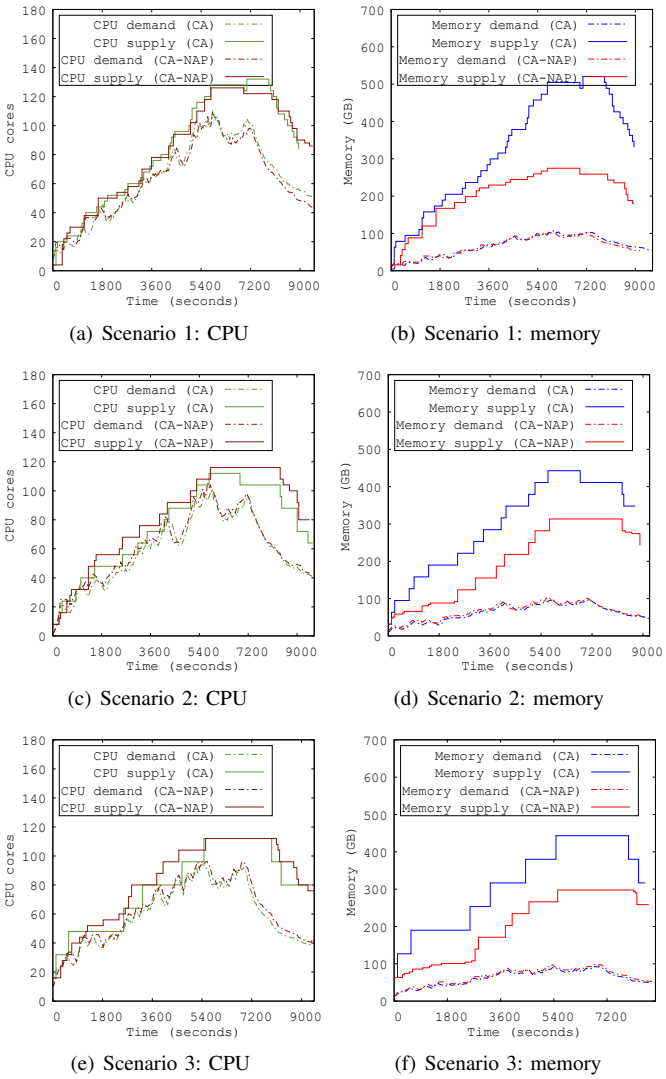
as the resources reserved for the operating system and system daemons are multiplied by the number of nodes. We see more overhead in Scenario 3 than Scenario 2 as fewer nodes of larger size are supplied by CA in Scenario 3 and we conclude that the overhead becomes larger as the node size increases.

Another interesting observation is the memory over-provisioning by CA as compared to CA-NAP seen in Figures 4(b) – 4(f), reaching up to 100.73%, 30.8%, and 48.30% for the three scenarios, respectively. This is because CA-NAP supplies nodes with higher CPU-to-memory ratio than CA, as can be seen in Table V reflecting the nature of the workload.

We, therefore, conclude that for workloads made up of several short- and long-running tasks with high diversity in resource demand, CA-NAP matches the demand better than CA. Moreover, CA scales-in faster than CA-NAP in all 3 scenarios, but more so in Scenarios 2 and 3 as larger nodes are removed at a time than CA-NAP. Furthermore, in all plots for *E1* we see the impact of the composition of the workload and the fast autoscaling interval (10s) on CA-NAP in that CA-NAP performs more autoscaling actions than CA.

In the plots for Experiments *E2*, we do not see significant differences in the way CA and CA-NAP supply resources – except for the case of Scenario 3 (Figs. 4(e) and 4(f)), in which case CA provisions more CPU and memory than CA-NAP. This is because in *E2* at the application level the HPA scales-out the pods as the traffic intensity increases, creating multiple pods having the same resource request (0.5 CPU cores and 1024 MB of memory each). Unlike *E1* the pods in *E2* do not have diverse resource requests, and thus as shown in Table V the nodes supplied by CA-NAP do not have diversity in their CPU-to-memory ratio compared to those in *E1*.

Unlike most of the general-purpose autoscalers studied in [14] or the Chameleon [4] and Chamulteon [16] autoscalers, we see in all the plots for Experiments *E1* and *E1* that both CA and CA-NAP scale-in slowly. This is because of the scale-in cool-down period of 10 minutes configured in the autoscalers, which, unfortunately, is not currently modifiable in GKE.



(a) Scenario 1: CPU (b) Scenario 1: memory

(c) Scenario 2: CPU (d) Scenario 2: memory

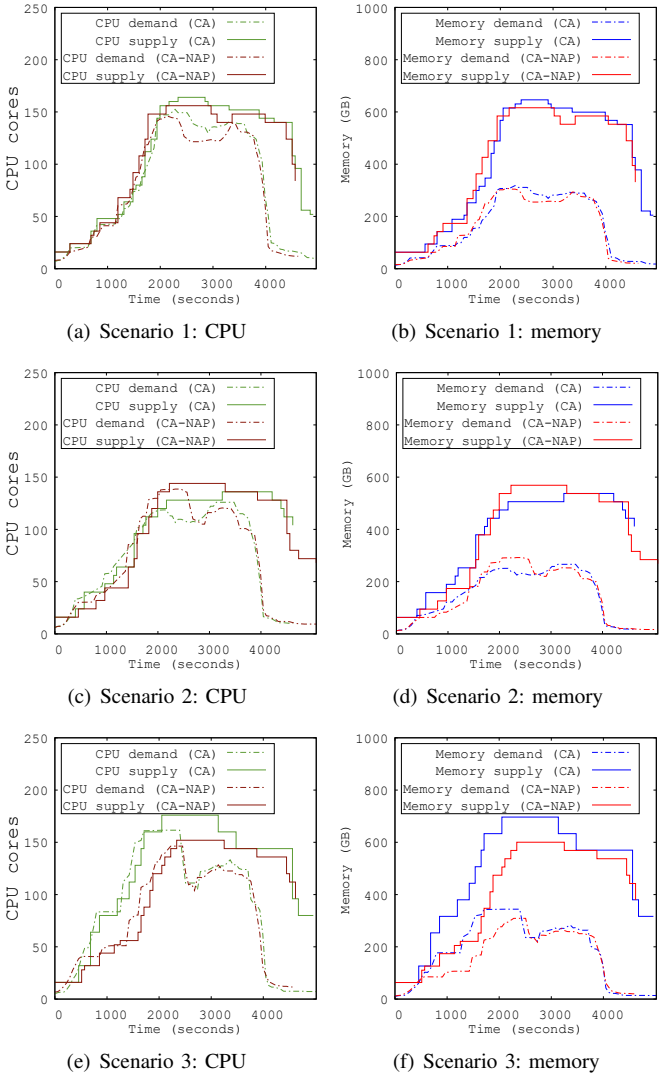
(e) Scenario 3: CPU (f) Scenario 3: memory

Fig. 4. Resource demand vs. supply for experiments in *E1*.

2) Analysis of autoscaling performance metrics: We present the results of the autoscaling performance metrics in Tables VI – VIII. In these tables, the rows show the experiment sets and scenarios being compared whereas each column shows a metric. The metric values are reported as the mean from three runs of each experiment. To complement the results in the tables, we present the spider charts shown in Figure 6. Each spider chart contains six points on the

TABLE IV
OVERVIEW OF AUTOSCALING PERFORMANCE METRICS.

No.	Metric name	Equation
1	Under-provisioning accuracy	$\theta_U[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} \frac{max(d_t - s_t, 0)}{d_t} \Delta t$
2	Over-provisioning accuracy	$\theta_O[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} \frac{max(s_t - d_t, 0)}{d_t} \Delta t$
3	Over-provisioning timeshare	$\tau_U[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} max(sgn(d_t - s_t), 0) \Delta t$
4	Under-provisioning timeshare	$\tau_O[\%] := \frac{100}{T} . max(sgn(s_t - d_t), 0) \Delta t$
5	Elasticity instability	$\upsilon[\%] := \frac{100}{T - t_1} \cdot \sum_{t=2}^{T} min(\lvert sgn(\Delta s_t) - sgn(\Delta d_t) \rvert, 1) \Delta t$
6	Overall provisioning accuracy	$\theta := \frac{1}{2}(\theta_U + \theta_O)$
7	Overall wrong provisioning timeshare	$\tau := \frac{1}{2}(\tau_U + \tau_O)$
8	Autoscaling deviation	$\sigma[\%] := (\theta^3 + \tau^3 + \upsilon^3)^{\frac{1}{3}}$



(a) Scenario 1: CPU (b) Scenario 1: memory

(c) Scenario 2: CPU (d) Scenario 2: memory

(e) Scenario 3: CPU (f) Scenario 3: memory

Fig. 5. Resource demand vs. supply for experiments in *E2*.

circumference of a circle, one for each autoscaling metric. The closer to zero and the thinner the web, the better the autoscaling configuration performs.

The autoscaling performance metrics are the average of the respective metrics for CPU and memory provisioning calculated using the equations given in Table IV. *It is important to notice that, the smaller a value is, the better.* The best values for each metric in the respective experiment set are presented as bold. The cost metric is calculated by aggregating the CPU and memory provisioned by the clusters for the duration of the experiment and multiplying by Google Cloud's per-hour pricing for CPU and memory. Here, we report the cost of running the clusters for one hour.

TABLE V
TOTAL NUMBER OF NODES AND CPU AND MEMORY SUPPLY AT THE PEAK OF EXP. *E1* AND *E2*.

Exp.	Scenario	Node type	Memory (GB)	No. of CPU cores	Qty
E1	CA Scenario 1	n1-standard-4	15	4	33
	CA-NAP Scenario 1	n1-standard-4	15	4	13
		n1-highcpu-2	1.80	2	5
		n1-highcpu-8	7.20	8	8
	CA Scenario 2	n1-standard-8	30	8	14
	CA-NAP Scenario 2	n1-standard-8	30	8	8
		n1-highcpu-4	3.60	4	12
		n1-standard-2	7.5	2	2
	CA Scenario 3	n1-standard-16	60	16	7
	CA-NAP Scenario 3	n1-standard-16	60	16	1
		n1-highcpu-4	3.6	4	12
		n1-standard-8	30	8	6
E2	CA Scenario 1	n1-standard-4	15	4	41
	CA-NAP Scenario 1	n1-standard-4	15	4	11
		n1-standard-8	30	8	14
	CA Scenario 2	n1-standard-8	30	8	20
	CA-NAP Scenario 2	n1-standard-8	30	8	16
		n1-standard-4	15	4	9
	CA Scenario 3	n1-standard-16	60	16	11
	CA-NAP Scenario 3	n1-standard-16	60	16	1
		n1-standard-4	15	4	10
		n1-standard-8	30	8	12

As shown in Table VI, almost all cases exhibit large values of over-provisioning accuracy θ_O. This can be explained by the over-provisioning of memory in all cases and scale-in delay of 10 minutes. This also explains the large values of the wrong over-provisioning time τ_O as the clusters are over-provisioned for the largest part of the experiment duration. This is because both CA and CA-NAP do not scale-out base on CPU utilization threshold but rather do so whenever there are pods that could not be scheduled due to a shortage of resources. The large values for θ_O are similar to those of some of the general-purpose autoscalers studied in [14] but different from those reported by [15], Chameleon [4] and Chamulteon [16], whereas almost all the above works except [15] report large values fo τ_O similar to ours.

Again, unlike the findings in [4], [14]–[16], we show that CA and CA-NAP in all cases have better performance in terms of under-provisioning accuracy θ_U and under-provisioning timeshare τ_U. This is because the default autoscaling interval of 10s is smaller than the one in the other works and the fast VM booting times in Google Cloud.

3) CA vs. CA-NAP Overall Comparison: First, to help compare CA and CA-NAP per experiment set, we present

TABLE VI
AUTOSCALING PERFORMANCE METRICS FOR ALL SCENARIOS IN EXPERIMENTS *E1* AND *E2*.

Scenario	θ_U[%]	θ_O[%]	τ_U[%]	τ_O[%]	υ[%]	σ[%]	Cost($)
E1 CA 1	1.52	183.88	12.76	**87.66**	**83.33**	114.62	4.18
E1 CA 2	0.39	204.32	5.08	94.92	91.71	125.35	4.10
E1 CA 3	**0.32**	224.48	**3.16**	96.84	95.96	134.43	4.16
E1 CA-NAP 1	1.09	126.10	5.80	94.20	87.46	**101.72**	**3.91**
E1 CA-NAP 2	1.38	169.34	7.62	92.38	92.11	117.00	4.24
E1 CA-NAP 3	0.46	**124.14**	3.80	96.20	92.44	104.99	4.00
E2 CA 1	3.66	**85.42**	19.59	80.41	60.27	75.83	5.70
E2 CA 2	**2.67**	115.36	18.24	81.76	62.79	83.46	**5.47**
E2 CA 3	4.12	197.89	18.74	81.26	68.32	114.82	5.86
E2 CA-NAP 1	2.73	95.06	**15.37**	83.73	58.03	**75.81**	5.76
E2 CA-NAP 2	5.88	105.35	23.65	**76.35**	**55.94**	78.13	5.94
E2 CA-NAP 3	3.23	184.57	17.04	82.96	61.58	107.22	5.65

TABLE VII
AUTOSCALING PERFORMANCE AGGREGATED PER EXPERIMENT AND AUTOSCALING POLICY.

Exp.	AS Policy	θ_U[%]	θ_O[%]	τ_U[%]	τ_O[%]	υ[%]	σ[%]	Cost($)
E1	CA	**0.75**	204.23	7.00	**93.14**	**90.33**	124.80	4.14
	CA-NAP	0.98	**139.86**	**5.74**	94.26	90.67	**107.90**	**4.05**
E2	CA	**3.48**	132.89	18.86	81.14	63.79	91.37	**5.68**
	CA-NAP	3.95	**128.33**	**18.69**	**81.01**	**58.52**	**87.05**	5.78

TABLE VIII
OVERALL AUTOSCALING PERFORMANCE METRICS FOR *CA* AND *CA-NAP*.

AS Policy	θ_U[%]	θ_O[%]	τ_U[%]	τ_O[%]	υ[%]	σ[%]
CA	**2.11**	168.56	12.93	**87.14**	77.06	108.09
CA-NAP	2.46	**134.09**	**12.22**	87.63	**74.59**	**97.48**

the metrics aggregated by auto-scaling configuration (i.e., CA and CA-NAP) for Experiments E1 and E2 in Table VII as well as in Figures 6(a) and 6(b). The presented metrics in the table are the mean of nine measurements per autoscaling configuration. For the comparison in E1, CA-NAP shows the best θ_O (139.86%), τ_U (5.74%) and σ (107.90%) whereas CA performs better in the remaining three metrics. However, CA-NAP shows the best σ (107.90%) because its θ_O (139.86%) is significantly lower than that of CA (204.23%). In the case of E2, CA-NAP outperforms CA on all metrics except θ_U.

Next, we look at the comparison of CA and CA-NAP across all scenarios in the two experiment sets as presented in Table VIII. The presented metrics are the mean from 18 measurements per each autoscaling configuration. The same results are plotted using a spider chart in Figure 6(c). In this comparison, CA-NAP outperforms CA in four our of six metrics: θ_O (134.09%), τ_U(12.22%), υ(74.59%) and σ(97.48%).

4) Cost comparison: In Tables VI and VII we also present the hourly cost of running the clusters in each scenario. Although the cluster under CA-NAP is cheaper than CA for the case of E1, and CA is cheaper in E2, we observe no significant cost savings by one over another. In the case of *E1*, although CA provisions far more memory than CA-NAP, it is not significantly more expensive than CA-NAP because CA-NAP slightly over-provisions CPU and the unit cost of memory is much less than that of CPU. For the case of *E2*, we do not see a significant difference in cost since both CA and CA-NAP are close to each other in resource supply.

5) Influence of Workloads: Taking the deviation from the theoretically optimal autoscaler σ as the single most important metric, since it captures all the other metrics, we conclude that CA-NAP outperforms CA in autoscaling performance, more so in *E1* than *E2*. The composition of the workload influences the performance of CA-NAP in that it performs better for workloads like the one in *E1* that are composed of several short- and long-running tasks with diverse resource requests, thus allow it to provision differently-sized nodes to match the demand. The nature of the workload influences the cost of the cluster as well as discussed in Section IV-D4. In *E1*, CA-NAP would have been significantly cheaper than CA if the workload resource request had high memory-to-CPU ratio as opposed to the high CPU-to-memory of our workload, in which case CA would have supplied far more CPU and would have been more expensive since the unit cost of CPU cores is far more than that of memory ($0.0347721 / vCPU hour vs. $0.0046607 / GB hour).

6) Influence of Configuration Parameters: The Kubernetes Cluster Autoscaler has several configuration parameters that influence its performance in addition to the strategy for node provisioning (CA and CA-NAP) we have studied in this paper. The three most important parameters that would influence the performance of CA and CA-NAP are autoscaling interval, scale-in cool down time, and extender. We have used the default values for these parameters in this work.

The autoscaling interval has a default value of 10 seconds and impacts the speed of scale-out, the number of scaling actions, the over- and under-provisioning timeshare metrics, and in the case of CA-NAP the size of nodes to be provisioned. The smaller the autoscaling interval is we observe faster scale-out, more autoscaling actions, better under-provisioning timeshare, worse over-provisioning timeshare, and in the case of CA-NAP smaller nodes are provisioned, and vice-versa.

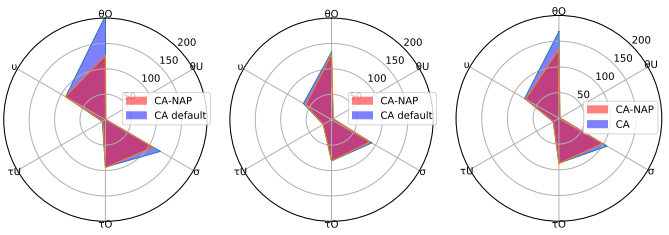
The scale-in time has a default value of 10 minutes and influences the over-provisioning accuracy metric, speed of scale-in, and cost of the cluster. The smaller this parameter is the better over-provisioning accuracy, the faster scale-in and the lower the cost of the cluster, and vice-versa.

The extender parameter has 5 possible values (*random*(default), *most-pods, least-waste, price*, and *priority*) and specifies the strategy used by CA-NAP to decide from which of the multiple node pools to provision a node during scale-out. Because of the default *random* value used in our experiments, we observe that CA-NAP does not provision the same types of nodes at every run of the experiments. Therefore, it is important to study the behaviour of CA-NAP with the other possible node pool selection strategies.

V. CONCLUSION

In the last few years, Kubernetes has emerged as the de-facto container orchestration platform in the cloud. Even though autoscaling is a widely-studied research topic in the community, the autoscaling mechanisms offered by Kubernetes remain largely unexplored.

In this paper, we report results from our extensive experimental evaluation of the Kubernetes cluster autoscaler under two configurations. In its default configuration (CA) the autoscaler provisions nodes at the time of scale-out from only one node pool, whereas when configured with node auto-provisioning (CA-NAP) it provisions nodes from multiple node pools. We compare these two configurations using



(a) CA vs. CA-NAP in E1 (b) CA vs. CA-NAP in E2 (c) CA vs. CA-NAP overall

Fig. 6. Scaling behavior overview using spider charts.

SPEC's autoscaling performance metrics and monetary cost of running the clusters. We show that CA-NAP outperforms CA overall because it provisions nodes of different sizes to match the workload demand better. CA-NAP shows better performance for applications that are composed of several short-running tasks and long-running services with diverse resource requests. The composition of the applications also influences the cost of running the clusters, as it determines the size and number of nodes to be provisioned. Moreover, CA and CA-NAP perform better in terms of under-provisioning timeshare and worse in terms of over-provisioning accuracy compared to other autoscalers from the related work [4], [14]–[16].

In this work, we showed the impact of different configurations and applications on the autoscaling performance and cost of running of a Kubernetes cluster. As the Kubernetes cluster autoscaler is highly configurable, it would be interesting to study the impact of tuning additional parameters on autoscaling performance and cost saving. Furthermore, the complex interaction of the container-level autoscaling mechanisms (horizontal and vertical pod autoscalers) and VM-level autoscaling (cluster autoscaler) remains unexplored.

References

[1] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal, "Dynamic scaling of web applications in a virtualized cloud computing environment," in *Proc. IEEE ICEBE*, 2009.

[2] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *Proc. IEEE IC2E*, 2014.

[3] A. Ali-Eldin, O. Seleznjev, S. Sjöstedt-de Luna, J. Tordsson, and E. Elmroth, "Measuring cloud workload burstiness," in *Proc. IEEE UCC*, 2014.

[4] A. Bauer, N. Herbst, S. Spinner, A. Ali-Eldin, and S. Kounev, "Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field," *IEEE Trans. Parallel and Distributed Systems*, vol. 30, no. 4, 2018.

[5] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janeceka, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Elsevier Future Generation Computer Systems*, vol. 27, no. 6, 2011.

[6] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," in *Proc. IEEE ICAC*, 2005.

[7] N. Herbst, R. Krebs, G. Oikonomou, G. Kousiouris, A. Evangelinou, A. Iosup, and S. Kounev, "Ready for rain? A view from SPEC research on the future of cloud metrics," SPEC RG Cloud Working Group, Tech. Rep. SPEC-RG-2016-01, 2016.

[8] Google Cloud Platform, "An update on container support on google cloud platform," https://cloudplatform.googleblog.com/2014/06/an-update-on-container-support-on-google-cloud-platform.html, 2014.

[9] Kubernetes, "Borg: The predecessor to Kubernetes," https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/, 2015.

[10] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Computing Surveys*, vol. 51, no. 4, 2018.

[11] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Proc. IEEE CloudCom*, 2011.

[12] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications," in *Proc. ICAC*, 2014.

[13] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: State of the art and research challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, 2017.

[14] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. Bauer, A. V. Papadopoulos, D. Epema, and A. Iosup, "An experimental performance evaluation of autoscalers for complex workflows," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 3, no. 2, 2018.

[15] L. Versluis, M. Neacsu, and A. Iosup, "A trace-based performance study of autoscaling workloads of workflows in datacenters," in *Proc. IEEE/ACM CCGRID*, 2018.

[16] A. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, and S. Kounev, "Chamulteon: Coordinated auto-scaling of micro-services," in *Proc. IEEE ICDCS*, 2019.

[17] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A micro-service reference application for benchmarking, modeling and resource management research," in *Proc. IEEE MASCOTS*, 2018.

[18] Y. M. Ramirez, V. Podolskiy, and M. Gerndt, "Capacity-driven scaling schedules derivation for coordinated elasticity of containers and virtual machines," in *Proc. IEEE ICAC*, 2019.

[19] V. Podolskiy, A. Jindal, and M. Gerndt, "IaaS reactive autoscaling performance challenges," in *Proc. IEEE CLOUD*, 2018.

[20] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Towards understanding heterogeneous clouds at scale: Google trace analysis," Intel Science

and Technology Center for Cloud Computing, Tech. Rep., 2012.

[21] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz, "Analysis and lessons from a publicly available Google cluster trace," University of California, Berkeley, Tech. Rep. UCB/EECS-2010-95, 2010.

[22] Z. Liu and S. Cho, "Characterizing machines and workloads on a Google cluster," in *Proc. IEEE ICPP Workshops*, 2012.