Flexible Migration in Blue-Green Deployments within a Fixed Cost

Eddy Truyen, Bert Lagaisse, Wouter Joosen
imec-DistriNet
KU Leuven
Belgium
eddy.truyen@cs.kuleuven.be

Arnout Hoebreckx
Dept. of Computer Science
KU Leuven
Belgium
arnout.hoebreckx@gmail.com

Cédric De Dycker
Dept. of Computer Science
KU Leuven
Belgium
cedric.dedycker@icloud.com

ABSTRACT

This paper presents the concept of PolyPod that consists of multiple Pods that run different versions of the same container image on the same node in order to share common libraries in memory. Its novelty is that it proposes a blue-print for blue-green deployments in order to balance maximum flexibility in the number of migration steps with maximum workload consolidation within a fixed total resource cost. This balance between flexibility and improved resource utilization is important for various application areas where users are served by the same application instance and have different time preferences for being upgraded to a new application version. The PolyPod concept is also relevant for a planned feature of Kubernetes so that Pods can be vertically scaled without re-starting them, but where scaling actions are aborted if the capacity of the node is to be exceeded. We explain how the PolyPod concept supports balancing flexible migration and resource utilization, with and without Pod restarts, by simulating various migration scenarios based on a quantitative cost model.

CCS CONCEPTS

Computer systems organization ~ Architectures~ Distributed architectures~ Cloud computing

KEYWORDS

Continuous deployment, Resource management, Container orchestration

ACM Reference format:

Eddy Truyen, Arnout Hoebreckx, Cédric De Dycker, Bert Lagaisse and Wouter Joosen. 2020. In *Proceedings of Containers Workshop on Container Technologies and Container Clouds (WOC'20). ACM, New York, NY, USA, 6 pages.* https://doi.org/10.1145/3429885.3429963

WOC'20, December, 2020, Delft, The Netherlands
 978-1-4503-8209-0/20/12...$15.00

1 Introduction

Vertical scaling of Pods without restarting the Pods will likely be a new feature of Kubernetes [1]. The design of this feature is complex because it breaks the architectural invariant that the resources allocated to a Pod should be immutable. Therefore, applications must explicitly opt-in for no-restart so they can adapt their configuration when resource allocation changes. Moreover, pre-emptive scheduling is not supported in this new feature: when scaling up a Pod on a node will cause the sum of requests to exceed the capacity of that node, no eviction of a lower-priority Pod will be attempted, but the scaling action will be aborted [1]. This imposes research towards application-specific architectures and abstractions where by design vertical scaling will never exceed node resources without unnecessary under-provisioning of nodes and opting out for no-restarts is taken into account as well.

This paper explores such possible abstraction in the context of continuous deployment. In particular blue-green deployments run multiple versions of the same application and allow to migrate workload to newer versions at different times in different migration sizes. Vertical scaling of Pods (scaling up/down) without restarts rather than horizontal scaling (scaling out/ in) is better suited for facilitating the above migration scenarios because it allows to maintain t+1 fault-tolerance requirements for all versions [2]. Moreover it is more cost-efficient for scaling down and roll backs. However, to support zero-down upgrades, there is a surge cost when the new version has been scaled up and the old version has not yet been scaled down. When applications opt-out for no-restart, this surge cost is even much higher.

We define the PolyPod concept, discuss its implementation feasibility and explore the trade-off between maximum flexibility and workload consolidation within a fixed resource cost. Understanding this trade-off allows for flexible planning of the number of migration steps and their respective workload migration sizes without reaching a step

where workload never can be migrated due to a too high surge cost on an already consolidated node.

The remainder of this paper is structured as follows. Section 2 motivates the work and provides the necessary background. Section 3 presents the PolyPod concept and discusses implementation feasibility. Section 4 analyzes the abovementioned trade-off by simulating all possible migration scenarios based on a quantitative cost model of the surge cost. Section 5 presents related work and Section 6 concludes.

2 Motivation and Background

In the paradigm of agile software development, continuous deployment is often used as a strategy for making applications available to end users. This strategy differs from traditional deployment strategies in terms of the greater frequency with which applications are upgraded to a newer version [3]. It is important that the service reduction is minimal during upgrades in order not to violate performance and availability SLAs.

2.1 Blue-green deployments

There are different techniques to perform a continuous deployment, but in this paper two techniques are important: rolling upgrades and blue-green deployments [4]. In a rolling upgrade, an application is upgraded to a new version by means of gradual replacement of application instances.

In blue-green deployments, older and newer versions of the application instances co-exist in parallel for a determined time, and all versions can receive user requests in parallel. Over time, users are migrated from older to newer versions in a gradual fashion in multiple upgrade phases. This is especially relevant for multi-tenant SaaS applications where multiple tenants are served by the same application instance. Service degradation or outages that may appear as part of a failed upgrade or an upgrade of a stateful application triggers the issue that not all tenants prefer an upgrade to happen during the same time slot [5].

In the container-based cluster orchestrator Kubernetes, Pods are the unit of deployment and therefore correspond to a separate application instance. Different Pod versions typically differ in their container image and resource allocation. Blue-Green deployments can be set up by creating different sets of replicated Pods, one for each application version.

2.2 Memory sharing

To meet fault-tolerance and availability requirements, replicated Pods of one version must be deployed on different nodes. However, it pays off to co-locate Pods of different versions on the same node to save memory.

After all, an advantage of containers is the possibility to share dynamically linked libraries in memory. In the article by Ferreira et al. [6], an experiment was conducted proving that containers with corresponding images share memory between shared libraries. The research was conducted on an older container technology LXC. However, it is also an interesting find for blue-green deployments in Kubernetes because this motivates placing Pods with different container image versions on the same node.

The original experiment was performed on LXC containers and an AUFS file layer. In this paper, we reproduced this experiment on Kubernetes where Docker is used as a container technology and overlay2 as a more modern implementation of AUFS [16].

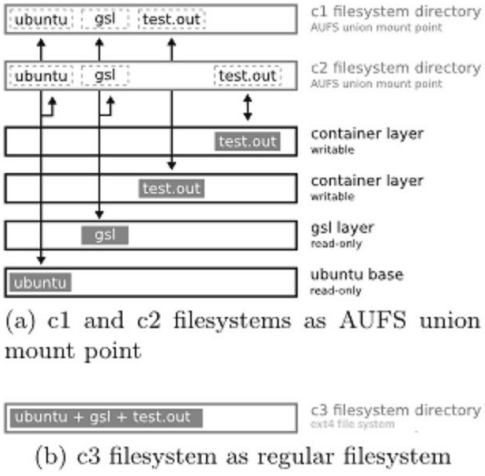


Figure 1: Experiment by Ferreira et al. [6]

The original experiment has been positioned as follows. There are three containers C1, C2 and C3 as shown in Figure 1. For C1 and C2 their file layers are an AUFS union mount point, consisting of several layers including a shared library. C3 consists of the same components but these are placed on an ext4 filesystem, which is the default filesystem for multiple Linux distributions. The results showed that libraries were shared between C1 and C2 but not with C3.

Table 1: Memory usage of Pods on one node (in kB)

	1 Pod		**2 Pods**	
Process	Shared	Private	Shared	Private
/a.out	0	12	4	8
/lib/x86_64-linux-gnu/libm-2.19.so	0	524	356	136
/lib/x86_64-linux-gnu/libc-2.19.so	956	136	1036	88
/gnu/gsl/lib/libgslcblas.so.0.0.0	0	72	64	8
/gnu/gsl/lib/libgsl.so.19.3.0	0	684	524	136
/lib/x86_64-linux-gnu/ld-2.19.so	140	8	140	8
Total	**1096**	**1436**	**2124**	**384**

The reproduction of this experiment uses an image containing the same layers as in the original experiment being the GNU Scientific Library (GSL) and an Ubuntu base. The memory measurements are done via / proc / <pid> / smaps, this shows the private and shared memory of the process. In the first scenario, one Pod is started up and this situation forms a reference against which the next scenario is compared. In the second scenario, a second identical Pod is started. Table 1 shows the measurements, whereby the shared and private memory per process is shown in kilobytes (kB).

To convert this into useful numbers, we compare two deployment options: (i) two Pods run on separate nodes and (ii) 2 Pods run on the same node.

- Scenario 1: Pods on 2 nodes
  - Total cost = 2 * (Shared + Private)
  - 2 * (1096 + 1436) = 5064
- Scenario 2: Pods on 1 node
  - Total cost = Shared + 2 * Private
  - 2124 + 2 * 384 = 2892

This shows that it is enormously cheaper to run the containers on the same node if they have the same libraries.

2.3 Horizontal vs. Vertical Scaling

The most used approach in Kubernetes for increasing and decreasing resources in blue-green deployments is by scaling out and in, i.e. horizontal scaling of Pods. After all, as workload increases drastically, the increasing demand of primarily stressed resources can only be satisfied by creating new Pods on other nodes.

However, horizontal scaling is not always an optimal option. This is especially the case when running the last set of users in an almost removed version. After all, to meet availability requirements, at least $t + 1$ Pods of that older version must always be running [2]. As such, decreasing below t+1 Pods is not desired. Instead, when the last batches of users are being upgraded to a new version, vertical scaling is preferred. Vertical scaling will keep the resource allocation of the Pods of the older version in an optimal state, i.e. the right amount of resources are allocated so there is no unnecessary overprovisioning of resources [7].

Another disadvantage of horizontal scaling is that scaling-in requires to wait for the to-be-deleted Pods to reach a quiescent state which may take a long time and therefore takes a lot of resources during each upgrade phase.

Finally, a roll-back to an older version becomes very expensive when the ongoing upgrade to the newer version has almost completed.

The latter two disadvantages also disappear with vertical scaling provided that vertical scaling can be done without container restarts. Technically speaking, Docker and other container runtimes already allow vertical scaling of containers without restarts [8]. However the current design of Kubernetes requires that Pods always must be rescheduled when adjusting their resource allocation. This is because the `request` and `limit` fields that respectively specify resource reservation and resource quota are immutable fields.

However, recently, a lot of work by the Kubernetes community has been spent on allowing in-place updates of Pod resources without restarting Pods [1]. The design of this new feature does not depend on the pre-emptive scheduling feature of Kubernetes scheduler: when the vertical scaling action of a Pod will cause the Pod to not fit on its current node, no eviction of a lower-priority Pod will be tried by the scheduler. Instead, the scaling action is aborted and re-tried later by the local Kubelet agent of that node.

3 The PolyPod Concept

The PolyPod concept is not meant as an extension to the concept of Pod in Kubernetes. Instead, it intends to be a blue-print for application architectures while remaining compatible with the existing state-of-practice of blue-green deployments as outlined in Section 3.1. By using Pod affinity constraints, different versions of the same application Pod can be placed together on the same node.

We propose the PolyPod concept based on the following three tenets. Firstly, a PolyPod consists of different Pods that run different versions of the same container image; if these versions depend on common libraries, memory can be saved provided the Pods are always co-located on the same node so that the libraries can be shared among Pods. To define the expected resource capacity of a node, a PolyPod defines the total resource request for all Pods as immutable values.

Secondly, for each Pod in a PolyPod a reservation is set for the amount of resources needed for serving an atomic unit of work. We term this concept user. Users can then be migrated among Pods in different sizes (i.e. number of users) by means of vertical scaling. A PolyPod can opt-in or opt-out for no-restart of its Pods during vertical scaling.

Thirdly, if the container orchestrator allows to manage shared memory among Pods, a PolyPod allows to set an estimated reservation for the shared memory and separate reservations for the private memory of its constituent Pods.

A PolyPod P is thus defined as ($\{P_1,...,P_m\}$, R, $M_{P\setminus s}$, $\{R_1,..., R_m\}$, $\{T_1,...,T_m\}$, s), where P_i is a Pod running application version i and all $P_{i:1..m}$ must be placed on the same node, R is the immutable request for resources by P and defined as (cpu_P, mem_P), $M_{P\setminus s}$ is the request of memory to be shared

among the Pods of P, $R_i=(cpu_i,mem_{i\backslash p})$ is the request for resources by Pi so that a single user can be served by Pi where $mem_{i\backslash p}$ is the request for the private part of memory of Pi, T_i is the current number of users served by P_i. Finally, if s=1, all Pi opt-in for no-restart, if s=2, all Pi opt-out for no-restart.

A full implementation of the PolyPod concept does not exist yet. The implementation of the PolyPod concept is based on the expectation that resource requests and limit will be mutable fields in a future version of Kubernetes [1] (cfr. Section 2.2). We have experimented with an implementation where Pods do not declare resource requests and resource allocation is managed by means of a front-end request scheduler. The concept of a shared memory request $mem_{P\backslash s}$ and a set of private memory requests $T_i.mem_{i\backslash p}$ could be implemented as part of the Linux kernel cgroups hierarchy [9]. The total amount of memory requested (i.e. $mem_{P\backslash s}+T_1.mem_{1\backslash p}+..+T_m.mem_{m\backslash p}$) is passed to the parent of the pause container of every Pod P_i [10]. Kubernetes already relies on the cgroup hierarchy for supporting Pod level resource management constructs [11].

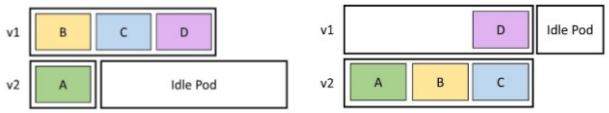
4 Understanding the Trade-off

This section analyzes the trade-off between maximum migration flexibility, hosting the highest amount of users and never end up in a situation where after a number of migration steps, users cannot be migrated anymore from an older to a newer version due to a too high surge cost.

4.1 Modeling the surge cost

The surge cost is needed to support zero-down upgrades and appears when scaling up and down. This surge cost is the largest when Pods opt-out for no restart and illustrated in Figure 2. When migrating from an application version 1 to version 2, the resources of version 2 must first be increased. When Pods opt-out for no-restart, this must be performed by means of a rolling upgrade, where Pods are replaced one-by-one, in order to prevent service disruption for application version 2. After users are migrated to version 2, the Pods of version 1 are down-scaled which again involves a rolling upgrade. Of course when Pods do opt-in for no-restart, the cost will be lower, but there is still a surge cost. We model this surge cost for a deployment on one node as follows.

A migration step consists thus of a scaling-up and scaling down phase. A migration step for PolyPod P is defined as G=(P,i,j,n), where $n <= T_j$ is the number of users that are upgraded from Pod P_j to Pod P_{j+i}.



$U_2(2)=R_1.3+R_2.1+R_2.(1+2)$ $D_1(2)=R_1.3+R_1.1+R_2(1+2)$

Figure 2: Scaling up and down with Pod restarts

Then the total request cost for scaling up Pod P_{j+i} is defined as follows:

$$U_{j+i}(n) = (R_j.T_j) + s.(R_{j+i}.T_{j+i}) + (R_{j+i}.n) \quad (1)$$

The total request cost for scaling down Pod P_j is defined as:

$$D_j(n) = s.(R_j.T_j) + (R_{j+i}.T_{j+i}) + (R_{j+i}.n) - (R_j.n) \quad (2)$$

The goal is to host as many users as possible and never run into a situation where users cannot be migrated due to a too high surge cost[1]:

$$\max(U_{j+i}(n)_{cpu}, D_j(n)_{cpu}) < cpu_P - \sum_{\substack{1\le k\le m \\ k\neq j,k\neq j+i}} (cpu_k.T_k) \quad (3)$$

$$\max(U_{j+i}(n)_{mem}, D_j(n)_{mem}) < mem_P - M_{P\backslash s} - \sum_{\substack{1\le k\le m \\ k\neq j,k\neq j+i}} (mem_{k\backslash p}.T_k) \quad (4)$$

$$maximize \sum_{1\le k\le m} T_k \quad (5)$$

4.2 Simulating different migration scenarios

We make a simulation of the possible migration steps based on the above formulas using Desmos[12] for CPU resources only. The simulation applies to an example situation for a PolyPod with 2 Pods for CPU resources only. Assuming our cluster has nodes with 2 CPU cores, the PolyPod's request R is set to 1750 millicores.

We distinguish between the cases where $R_1=R_2$ and $R_1\neq R_2$. In the case where values for $R_1=R_2$ = 90 millicores and no-restart is opted-in, we see that maximum 18 users can be hosted by a single PolyPod provided the migration size n is always kept to 1 (see Figure 3a). So 18 different scaling actions are needed to migrate all users. If all users need to be migrated in one step, only 9 users can be hosted. If users can be migrated in two steps (9 users, 1 user), 10 users can be hosted (see Figure 3b). If no-restart is opted out, a maximum of 9 users can be hosted, but the migration size n does not matter (see Figure 3c-d). Note that when $R_j=R_{j+i}$ the cost for scaling down $D_j(n)$ becomes independent of the migration size n.

[1] Formulas (3) and (4) do not express the temporal logic that this constraint must "always" be satisfied, but they allow us to instantiate multiple migration steps over time using the graph calculator desmos[12]

In the case $R_1>R_2$, Pod P_2 will be over-provisioned (see Figure 4). Let $R_1=90$ and $R_2=65$. If no-restart is opted in, then still 18 users can be hosted if n=1 (see Figure 4a). If all users need to be migrated in one step, 11 users can be hosted at most (see Figure 4b). If no-restart is opted out, also 11 users can be hosted, but migration size matters: all 11 users must be migrated in at most three steps and the first step must be higher than 8 users (see Figure 4e), i.e., the following steps are possible (11),(10,1),(9,2),(9,1,1). However, if 9 users are hosted, migration size does not matter anymore (see Figure 4c-d).

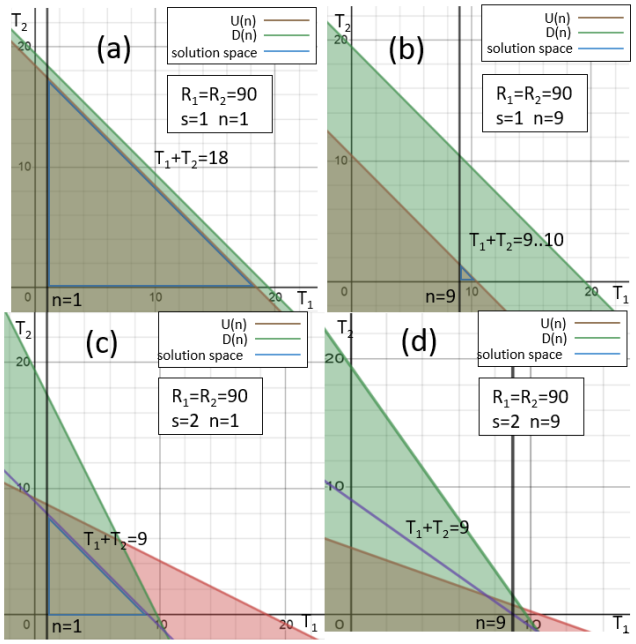


Figure 3: $R_1=R_2$. For all possible combinations of T_1xT_2 as scoped by the blue triangle, it is always possible to migrate n users within a fixed cost of 1750 millicores. The optimal capacity is given by the hypotenuse (the longest side of the triangle).

In the case $R_1<R_2$, i.e. $R_1=90$ and $R_2=115$, it is only possible to migrate all users if P_1 is under-provisioned from the beginning with 14 users in case s=1 and n=1 (see Figure 5a). If s=1 and n=8, only 8 users can be hosted by Pod P_1 (see Figure 5b). If s=2 and n=8, at most 8 users can hosted, but again migration size matters: all 8 users must be migrated in one step, or in two equal steps of 4 users (see Figure 5d-e) . If 7 users are hosted, migration size does not matter (see Figure 5c).

4.3 Discussion

Based on the above simulations we can draw the following insights on the trade-off between maximum flexibility and maximum workload consolidation $\sum_{1\leq k\leq m} T_k$ within a total fixed cost R. First, the resources needed for running a single user should be as equally as possible or at least $\forall\, i{:}\, 2..m{:}\, R_i \leq R_1$. Otherwise, not all workload can be migrated from old to new versions by means of vertical scaling. A consequence from this is that the request R_i could be relaxed towards an atomic unit of migration size instead of an atomic unit of workload and to meet user demands the horizontal Pod auto-scaler of Kubernetes should be used to manage service-level objectives.

Secondly, when no-restart is opted-in, more workload can be hosted by a single node if the migration size is as small as possible, but this requires a large number of migration steps.

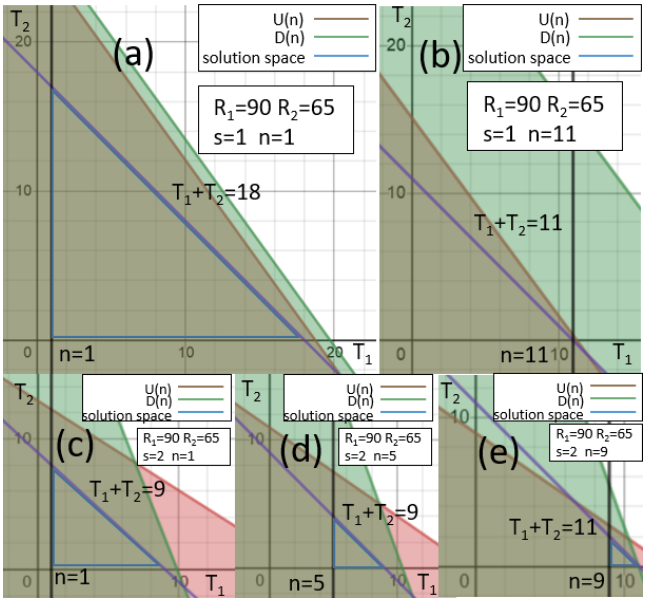


Figure 4: $R_1 > R_2$ (cfr. Fig.3 for explanation)

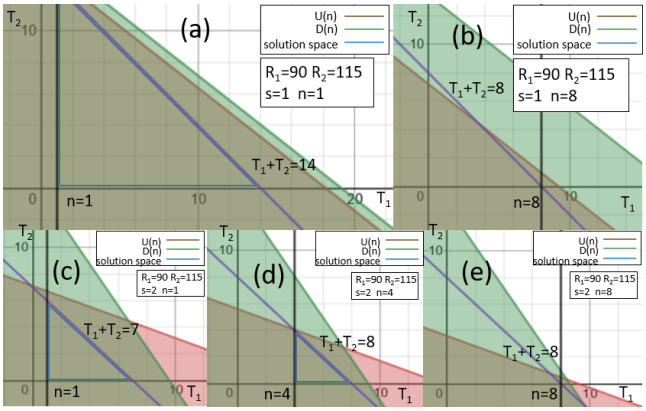


Figure 5: $R_1 > R_2$ (cfr. Fig. 3 for explanation)

If all workload must be migrated in one step, only a lower amount of workload can be hosted.

Thirdly, if no-restart is opted-out, the migration size does not matter when the sum of the requests of the Pods is always a constant. Moreover if $\forall\, i{:}\, 2..m{:}\, R_i \leq R_1$ the migration size neither matters provided an amount of over-provisioning of P_1 is tolerated.

The above insights enable application architects to select an appropriate trade-off between flexible migration and workload consolidation within a total resource cost.

5 Related Work

Shekhar et al [13] combines vertical scaling and machine learning to support pro-active automated scaling by predicting the workload on the system and its performance. This work has been evaluated on top of Docker that allows adjusting resource allocation of containers without restarting them.

Vertical Pod Autoscaler (VPA) [14] is an add-on functionality in Kubernetes that also supports vertical scaling in an automated way to migitate over-provisioning.

RUBAS et al. [15] proposes an alternative of VPA where it is possible to dynamically change the resources of Pods without causing a service downtime by relying on the Checkpoint Restore in Userspace (CRUI) functionality. This makes it possible to pause a container, save the complete state of the container and restart the container at a later time with the correct state. This technique is very effective when it comes to stateful applications. The results show that vertical scaling via a migration with RUBAS can be performed faster than when the resources are restarted with VPA. While the time of a migration is short, it is still significant enough to cause a service interruption. As long as the pod is migrating, it cannot process requests and must be compensated elsewhere.

Wang et al [16] explores how to enable resource sharing between containers and how to reduce container image sizes. The goal is to achieve efficiency here in a different way. A new container management system is proposed where containers run on top of a shared layer that contains the essential software and libraries for all containers. The paper compares this architecture to Docker in terms of image sizes and startup time. The proposed architecture scores better in both areas, with smaller image sizes and a lower container start-up time. Whether the proposed architecture is also more memory efficient or guarantees a better performance is not presented.

To our knowledge, none of the above works aim to balance migration flexibility and cost in blue-green deployments.

6 Conclusion

This paper has presented the PolyPod concept for migrating workload in blue-green deployments by means of vertical scaling with and without Pod restarts. It supports balancing the trade-off between maximum flexibility in the number of migration steps and maximum workload consolidation within a fixed total resource cost. This keeps blue-green deployments pinned on the same set of nodes in order to benefit from sharing of common libraries in memory.

REFERENCES

[1] V. Kulkarni, "enhancements/20181106-in-place-update-of-pod-resources.md at master · kubernetes/enhancements." [Online]. Available: https://github.com/kubernetes/enhancements/blob/master/keps/sig-node/20181106-in-place-update-of-pod-resources.md. [Accessed: 17-Sep-2020].

[2] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Jan. 1990.

[3] G. G. Claps, R. Berntsson Svensson, and A. Aurum, "On the journey to continuous deployment: Technical and social challenges along the way," in *Information and Software Technology*, 2015, vol. 57, no. 1, pp. 21–31.

[4] T. A. Limoncelli, S. R. Chalup, and C. J. Hogan, "The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems, Volume 2," 2014. [Online]. Available: http://the-cloud-book.com/. [Accessed: 14-Jan-2016].

[5] F. Gey, D. Van Landuyt, and W. Joosen, "Evolving multi-tenant SaaS applications through self-adaptive upgrade enactment and tenant mediation," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2016, no. 11, pp. 151–157.

[6] J. Bravo Ferreira, M. Cello, and J. O. Iglesias, "More sharing, more benefits? a study of library sharing in container-based infrastructures," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2017, vol. 10417 LNCS, pp. 358–371.

[7] K. Rzadca *et al.*, "Autopilot: workload autoscaling at Google," in *EuroSys 2020*, 2020.

[8] S. Shekhar, H. Abdel-Aziz, A. Bhattacharjee, A. Gokhale, and X. Koutsoukos, "Performance Interference-Aware Vertical Elasticity for Cloud-Hosted Latency-Sensitive Applications," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018.

[9] T. Heo, "linux/cgroup-v2.rst at master · torvalds/linux," 2015. [Online]. Available: https://github.com/torvalds/linux/blob/master/Documentation/admin-guide/cgroup-v2.rst. [Accessed: 21-Sep-2020].

[10] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile Cold Starts for Scalable Serverless," in *HotCloud 2019*, 2019.

[11] "community/pod-resource-management.md at master · kubernetes/community." [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/node/pod-resource-management.md. [Accessed: 22-Sep-2020].

[12] Desmos Inc., "Desmos API v1.0 documentation," *Desmos, Beautiful, Free Math*. [Online]. Available: www.desmos.com. [Accessed: 21-Sep-2020].

[13] S. Shekhar, H. Abdel-Aziz, A. Bhattacharjee, A. Gokhale, and X. Koutsoukos, "Performance Interference-Aware Vertical Elasticity for Cloud-Hosted Latency-Sensitive Applications," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018.

[14] Cloud Native Computing Foundation, "autoscaler/vertical-pod-autoscaler at master · kubernetes/autoscaler." [Online]. Available: https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler. [Accessed: 12-Nov-2018].

[15] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes," *IEEE Int. Conf. Cloud Comput. CLOUD*, vol. 2019-July, pp. 33–40, 2019.

[16] W. Wang, L. Zhang, D. Guo, S. Wu, H. Cui, and F. Bi, "Reg: An Ultra-lightweight Container that Maximizes Memory Sharing and Minimizes the Runtime Environment," in *2019 IEEE International Conference on Web Services (ICWS)*, 2019.