Containerized Workflow Scheduling

Research Project 1
Klop, Isaac
University of Amsterdam
isaac.klop@os3.nl

July 11, 2018

Abstract

Workflow systems often employ scheduling algorithms that order the execution of tasks in a workflow to improve performance and efficiency. Container orchestrators are often used to manage and schedule containers on a large scale. We design a containerized workflow, a workflow where all tasks are containers, with a critical path. We run the workflow on two container schedulers, Kubernetes and Docker Swarm. The goal of this research is to find a method to order the execution of a containerized workflow on Kubernetes and Swarm. We order the execution by setting a priority for certain tasks when submitting the tasks to Kubernetes. For Swarm, we phase the submission of the tasks to influence the order in which the tasks are executed. The results for Kubernetes show that the priority provides a slight improvement in execution time. The results for Swarm are inconclusive.

1 Introduction

Operating system (OS) virtualization, such as containers, is becoming increasingly popular in the DevOps and cloud communities. Users often run large volumes of interconnected containers on their cloud infrastructure, requiring container orchestration tools such as Kubernetes [1] and Docker Swarm [2] to manage them. These orchestrators provide a layer of abstraction on top of the underlying infrastructure and provide the user with an API. The user provides the orchestrator with computing resources, usually in the form of Virtual Machines (VM). The user can then submit containers to the API and the orchestration tool will take care of the scheduling and the monitoring of the containers.

Workflows are a widely used abstraction for describing large scientific applications. "A workflow comprises three components: a list of tasks or operations, the set of dependencies between the interconnected tasks (the flow), and the set of data resources used to generate or terminate the flow" [3]. The execution of these workflows can be automated by a workflow management system (WMS) such as Pegasus [4] or Taverna [5]. The WMS organizes the execution of the

workflows, taking into account the flow of the workflow and the resources needed for the execution.

The benefits of containers can also be reaped in scientific workflow applications [6]. Some WMSs already provide operators that use containers. However, there is little work done in combining the scheduling algorithms in WMSs and the scheduling algorithms in container orchestrators. Currently there is no standard or specification for interfacing WMSs and container orchestrators.

Research Question

The main task of a container scheduler is to schedule containers on a set of computing nodes. The logic of the scheduler is aimed at finding the right node for each container. Container schedulers have different strategies and different algorithms that they use, but the logic is always aimed "downward" towards the infrastructure. The container scheduler does not take the context of the containerized task into account. In the case of workflows this means that the container scheduler is not aware that other tasks may be dependent on the task inside the container or that the task itself may be dependent on other tasks.

Not only dependencies, but also ordering is an inherent part of workflows. Workflow systems often employ scheduling algorithms that order the execution of tasks to improve the performance of a workflow by improving its efficiency [7]. If we are to employ the same scheduling algorithms when executing workflows on container schedulers we need a way to influence the order in which the containerized tasks are scheduled. The research question of this project is:

How can we order the execution of a containerized workflow on a container scheduler?

2 Related Work

There have been successful attempts to incorporate container scheduling in workflows. Zheng et al. [8] connected Makeflow [9] (a WMS) and Mesos [10] (a container scheduler) to run a large bioinformatics workflow. Zheng et al. explore the possibility of running scientific workflows on a container-based scheduling platform. They implement a batch job system for Mesos and connected it to Makeflow and Work Queue [11].

Applatix recently introduced Argo, an open source container-native workflow engine for Kubernetes [12]. Argo expands on Kubernetes as a custom resource definition [13]. It is designed around running workflows of containerized tasks on Kubernetes. Argo is, however, limited to Kubernetes as its container scheduler.

Apache Airflow is a workflow system sponsored by the Apache Incubator [14]. "Airflow is a platform to programmatically author, schedule and monitor workflows." Airflow is currently developing an integration with Kubernetes as shown in this [15] open Jira ticket.

3 Method

For the experiments we pick two container schedulers: Docker Swarm and Kubernetes. For each scheduler we set up a small cluster of 5 nodes with 1 vCPU and 1GB of RAM. For Swarm, we use Docker version 18.03.1-ce and for Kubernetes we use version 1.10.5.

We design a workflow consisting of containerized tasks. For the tasks we create a simple Python application that uses Stress [16] to emulate CPU and memory usage for a given amount of time. We then package this application in a Docker container. This gives us a containerized task for which we can set the resource use and duration. We can set different parameters for each individual task in the workflow. The container starts CPU cycles and allocates memory based on the parameters with which the container is run. The source code of the Python application can be viewed on GitHub [17].

We design the workflow based on two requirements: It should have an identifiable critical path [18] and its resource requirements should exceed the capacity of the cluster. We design the workflow and its critical path in such a way that the order in which the tasks are executed should have an effect on the total execution time of the workflow. By exceeding the resources of the cluster, we make sure that the scheduler cannot execute all tasks in parallel. This forces the scheduler to make decisions on the order of the tasks.

We then execute the workflow based on two scheduling approaches: Batch and Critical Path. In batch, we identify which tasks are ready to be scheduled (have all dependencies met) and immediately schedule them on the container schedulers. In batch, we do not influence the order in which the tasks are scheduled. In the case of critical path we do influence the order in which the ready tasks are executed, specifically to prioritize the tasks of the critical path.

We measure the execution time of our workflow from start to finish, to assess the effectiveness of the different scheduling approaches and whether the ordering of tasks has a noticeable effect on the execution time.

4 Experiment Setup

4.1 The Workflow

The workflow used in the experiments consists of 34 tasks. Figure 1 shows the structure of the workflow with each node representing a task and each edge representing a dependency. The flow starts with a single preprocessing task. This task is followed by four subflows, three of the subflows have a single task followed by three parallel tasks. The three parallel tasks can only start if the first task of the subflow is completed. The last flow, shown on the bottom of the graph, consists of twenty parallel tasks. The entire workflow ends in a single postprocessing task, this task is dependent on all other tasks in the workflow.

For the experiments we only use the duration and memory parameters, the task will not instruct Stress to do CPU cycles. We set the duration for most

of the tasks at 20 seconds and the memory at 500MB as indicated in figure 1 (green tasks). In order to create a critical path we increase the duration of the tasks of one of the subflows. These 3 blue tasks in figure 1 have a duration of 90 seconds instead of 20. The red path in figure 1 shows the critical path.

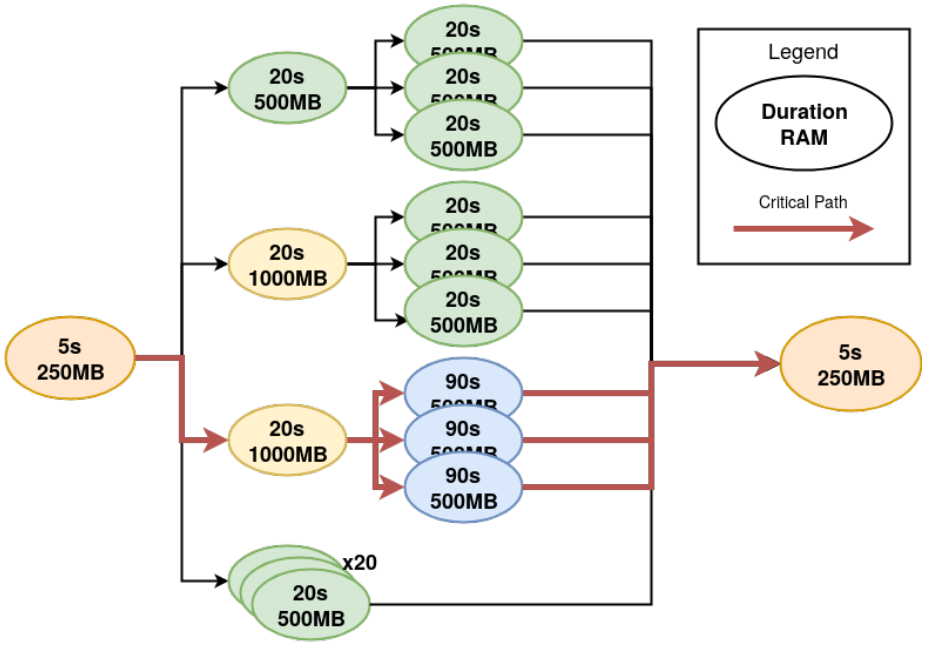

Figure 1: Graph of the workflow used in the experiments including resource and duration parameters. The red path is the critical path.

The memory requirements and the number of parallel tasks puts the cluster into a resource constrained state. This forces the container scheduler to make decisions as to the order in which it schedules the tasks.

We set the memory requirements for the first task of the critical path and one other task at 1000MB (yellow tasks in figure 1). Each node in the respective cluster can handle two 500MB tasks in parallel or one 1000MB task. The 1000MB task cannot be scheduled on a node that already contains another task. This makes it more difficult to schedule these tasks which increases the odds of these tasks being scheduled later in the ordering. We expect that this slightly exaggerates the difference between the batch and critical path scheduling approaches.

The pre- and postprocessing tasks have a duration of 5 seconds and a memory requirement of 250MB (orange tasks in figure 1). The dependencies of the workflow state that these two tasks are always executed alone and therefore the parameters are not important. We are interested in the ordering and scheduling of the tasks in between.

Execution Time

Kubernetes does not allow containers to be scheduled on a master node by default. Docker Swarm does allow this. We adhere to these defaults, which leaves 4 worker nodes for Kubernetes and 5 for Swarm. Assuming no overhead and assuming a best case scenario where the critical path is scheduled first, the execution time of the workflow is 130 seconds on Kubernetes and 120 seconds on Swarm as displayed in table 4.1. In a worst case scenario where the scheduler schedules the critical path at the last possible moment, the execution time of the workflow is 180 seconds for Kubernetes and 160 for Swarm.

Scheduler	Lowest	Highest
Swarm	120s	160s
Kubernetes	130s	180s

Table 1: Lowest/Highest possible total execution times assuming no overhead.

4.2 Executing the Workflow

Using the Python clients for Swarm and Kubernetes [19, 20], we create a Python application to execute the workflow on the two container schedulers.

The application submits the containerized tasks to the container scheduler and passes the duration and memory parameters to the container, along with a unique ID. The container starts up Stress and allocates a certain amount of memory for the duration of the task. After Stress returns, the container sets a key (based on the ID) in a Consul [21] key value store. After submitting the container, the Python application starts a thread that watches this key in Consul. The thread returns when the key is set, which signals the end of the task.

The Python application manages the dependencies of the workflow. It submits a task as soon as that task's dependencies are met.

Kubernetes has a resource definition called `Job` [22]. The containers scheduled using the Job definition are expected to terminate. Most other definitions schedule containers that are not expected to terminate, Kubernetes will continuously try to recover these containers by restarting (rescheduling) them. The Python application uses the Job resource when scheduling tasks on Kubernetes.

Docker Swarm does not have such a resource definition. There is a discussion on this topic in this [23] issue on GitHub. Ellis, in a blog, provides two methods of working around this limitation [24]. One is a Golang CLI that he himself maintains, the other is using the `Service` resource but with a `Restart Policy` of `None`. This prevents Swarm from restarting or rescheduling the container after it finishes. This has the downside that Swarm will not reschedule the container when it stops for any other reason. The Python application uses the latter method as it is easier to implement.

We configure the containers in both Kubernetes and Swarm to reserve the amount of memory the task requires using the respective resource reservation

flags. The container scheduler uses this knowledge when scheduling the containers, it will make sure that the containers do not directly compete over resources. Both Kubernetes and Swarm do not have a flag for the duration of a container. Even when, as in our experiments, the duration of the task is known beforehand there is no way to indicate this to the container scheduler.

5 Analysis

5.1 Ordering

The simplest way to order the execution of the containerized tasks is to submit them in order to the container scheduler. Both Kubernetes and Swarm, however, do no not respect this ordering. We observe that the order in which the containers are scheduled differs from the order in which the containers are submitted. We see that the first containers that are submitted are scheduled somewhat in order, but also here we see unpredictable ordering. It becomes even more unpredictable when the first containers occupy all available nodes. Both Kubernetes and Swarm pick a seemingly random container from the queue as soon as a node becomes available. This makes the container scheduler's queue an unpredictable and unreliable method of ordering the tasks.

Kubernetes has the option of configuring a custom scheduler [25]. Containers can be configured to use this scheduler. The custom scheduler, however, is invoked when a container has already been taken from the queue. The custom scheduler replaces the logic that decides on which node a container should be scheduled. The custom scheduler is still bound by the same queue limitations of the default scheduler.

Docker Swarm has a configuration option that changes the strategy of the scheduler [26]. As with Kubernetes, this strategy only affects the way in which nodes are selected for a container. It does not change the behavior of the queue.

Containers can have a priority in Kubernetes [27]. The priority is an integer and can take many different values, this allows for very fine-grained control over the prioritization of containers. Containers with a higher priority are scheduled before containers with a lower priority. In Kubernetes version 1.10 (the version used for the experiments) Kubernetes also preempts running containers to free up resources for the prioritized container. The preemption can be damaging to the total execution time when it kills running tasks and even more so when it kills tasks that have no graceful way of handling the shutdown. This is fixed in the newer version of Kubernetes (1.11), containers can now have a priority while preemption is disabled. However, the preemption is still enabled by default.

Docker Swarm does not have the ability to set a priority for a service or container. We can influence the order of execution by submitting the tasks of the critical path together with only a part of the other tasks. The remaining part of those tasks can be submitted after the critical path has finished its first task. By waiting for the critical tasks to complete before scheduling the non-critical tasks, we can force Swarm to execute the critical tasks first. The critical

tasks alone, however, do not take up all available resources. To overcome this inefficiency, we submit the critical tasks together with a part of the other tasks.

5.2 Results

We run the workflow one hundred times per experiment on the two container schedulers and we measure the average execution time. For Kubernetes we do two experiments: one where we set a higher priority for the fist task of the critical path and one where all tasks have the same priority.

Figure 2 shows the average execution time for both experiments on Kubernetes. We observe that the execution time, when setting a higher priority for the critical path, is slightly lower at 244 seconds versus 260 seconds without the priority.

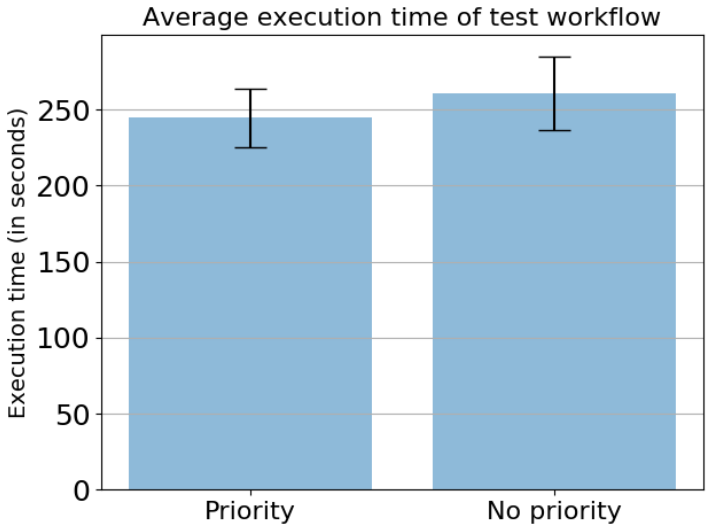

Figure 2: Barplot showing the mean execution time of the workflow on Kubernetes including an indication of 1 standard deviation.

In both experiments we observe a high standard deviation. This is to be expected as there are many different orders in which the workflow can be executed. Even in the experiments with the priority set there are still different orders of execution.

In order to prioritize the critical path in Swarm we hold back some of the 20 parallel tasks shown on the bottom of figure 1. We run three experiments: One where we submit all ready containers at once, one where we hold back 5 of the parallel tasks until the first task of the critical path has finished and one where we hold back 10 of the parallel tasks until the first task of the critical path has finished.

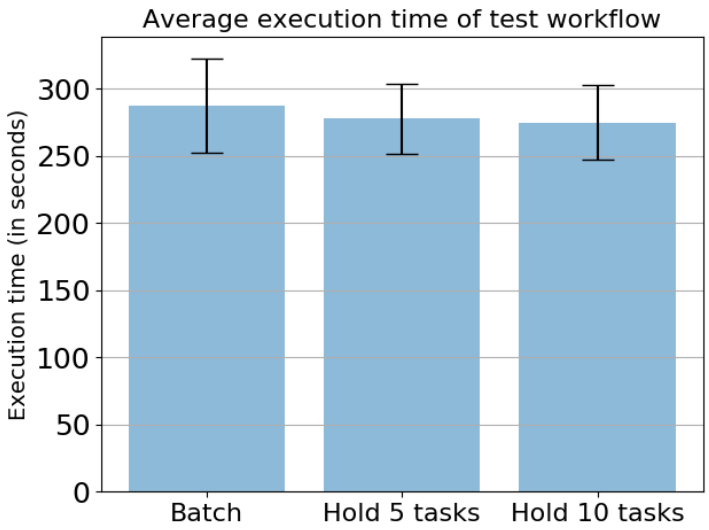

Figure 3: Barplot showing the mean execution time of the workflow on Swarm including an indication of 1 standard deviation.

Figure 3 shows the average execution times for the three Swarm experiments. We observe that the average execution time of the batch approach is slightly higher at 287 seconds versus 277 when we hold back 10 of the parallel tasks and 275 seconds when we hold back 5 of those tasks.

We see that the standard deviation is higher than the standard deviation in the Kubernetes experiments. Part of this difference can be explained by Swarm having a higher average execution time. We also see that the execution times are closer together. The difference in execution time is at 12 seconds less than half of the standard deviation of the experiments.

6 Conclusion

The queue of the scheduler in both Swarm and Kubernetes does not reliably behave as a first in, first out queue. We cannot use this queue when ordering the execution of a containerized workflow on Kubernetes or Swarm.

In the case of Kubernetes we can configure a priority per container. The scheduler will always try to schedule the container with the highest priority first. This feature allows one to order the containers in Kubernetes' queue and it that way order the execution of a containerized workflow. The results of the experiments show a slight improvement of 16 seconds when prioritizing the critical path of the workflow used in the experiments.

Docker Swarm lacks the features to order the containers in the queue itself. One has to assume that Swarm will pick a random container from the queue as soon as resources become available. One has to submit the containers to Swarm

in a carefully timed manner. By submitting containers after the first task of the critical path has finished we gain a slight improvement in the total execution time. The differences between the execution times are small, however, and the standard deviation over 100 experiments is relatively high. The results indicate a slight improvement but do not definitively prove that this method of ordering the tasks is effective.

7 Discussion

Kubernetes and Docker Swarm are not designed around workflows. This brings certain limitations. Containers can be configured with a resource reservation but the duration or estimated duration of a task cannot be indicated to the scheduler. Even in different scheduling strategies or custom schedulers, the container scheduler has no way of taking into account the context of the container. There are ways of indicating dependencies between containers but a dependency in that context means that one container has to be up and running before the other container can start. In the context of workflows a dependency means that a task has to finish before another task can start.

Workflow Management Systems in turn are often not designed around containers or container schedulers. Some workflow systems can be extended through plugins or operators. In this research we ordered the execution of a containerized workflow in a static setting. We first determined the order and then execute the workflow in that order. This can only be achieved when parameters such as resource use and duration are known beforehand. When running a workflow it may be useful to actively monitor the progress of a workflow and dynamically change the ordering of the tasks in the case of a task failure. This will require the WMS to interface with the container scheduler.

The experiments performed in this research can be expanded. In the case of Kubernetes we can set a unique priority for every task in the workflow and see how that approach compares, especially when preemption is disabled. In the case of Swarm we now wait until the first task of the critical path has finished, but we can also query the Docker API to see whether the task has started. We can also use a form of dynamic scheduling where we interface with the scheduler's API to determine when to submit which task.

The experiments performed have a high standard deviation. This is in part expected due to the nature of the experiments where there are many ways of executing the workflow used in the experiments. In future research one can do the same experiments with a different workflow or by leaving fewer different ordering options to the container scheduler to see if this high standard deviation persists. This may provide a more definitive answer as to the effectiveness of the ordering approaches.

8 Future Work

More research is needed into the behavior of the schedulers and their queues. Both Swarm and Kubernetes are open source. It should not be difficult to find the reason for the erratic behavior of the queue in the source code. We expect that the scheduler loops over the containers in the queue and schedules every container that it can schedule. When resources become available, the scheduler can be anywhere in its loop and thus anywhere in the queue. It might be worth to research whether this scheduling behavior can be modified or whether it is possible to provide an alternative scheduler, one that might be slower but is predictable.

This research focuses on Kubernetes and Docker Swarm. The same research can be done for other container schedulers such as HashiCorp's Nomad and Apache Mesos' Marathon.

References

[1] "Kubernetes," https://kubernetes.io/, Accessed 01-07-2018.

[2] "Docker Swarm," https://docs.docker.com/engine/swarm/, Accessed 01-07-2018.

[3] C. S. Liew, M. P. Atkinson, M. Galea, T. F. Ang, P. Martin, and J. I. V. Hemert, "Scientific workflows: moving across paradigms," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, p. 66, 2017.

[4] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny *et al.*, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[5] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher *et al.*, "The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud," *Nucleic acids research*, vol. 41, no. W1, pp. W557–W561, 2013.

[6] T. Adufu, J. Choi, and Y. Kim, "Is container-based technology a winner for high performance scientific applications?" in *Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific*. IEEE, 2015, pp. 507–510.

[7] A. Bala and I. Chana, "A survey of various workflow scheduling algorithms in cloud environment," in *2nd National Conference on Information and Communication Technology (NCICT)*. sn, 2011, pp. 26–30.

[8] C. Zheng, B. Tovar, and D. Thain, "Deploying high throughput scientific workflows on container schedulers with makeflow and mesos," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press, 2017, pp. 130–139.

[9] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids," in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. ACM, 2012, p. 1.

[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[11] P. Bui, D. Rajan, B. Abdul-Wahid, J. Izaguirre, and D. Thain, "Work queue+ python: A framework for scalable scientific ensemble applications," in *Workshop on python for high performance and scientific computing at sc11*, 2011.

[12] "Argo - GitHub," https://github.com/argoproj/argo, Accessed 01-07-2018.

[13] Kubernetes Documentation, "Custom Resources," https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/, Accessed 01-07-2018.

[14] "Apache Airflow (incubating) website," https://airflow.apache.org/, Accessed 01-07-2018.

[15] "Airflow kubernetes integration," https://issues.apache.org/jira/browse/AIRFLOW-1314, Accessed 01-07-2018.

[16] "Stress - Manpage," https://linux.die.net/man/1/stress, Accessed 02-07-2018.

[17] Isaac Klop, "task-emulator - GitHub," https://github.com/IsaacKlop/task-emulator, Accessed 02-07-2018.

[18] D.-H. Chang, J. H. Son, and M. H. Kim, "Critical path identification in the context of a workflow," *Information and software Technology*, vol. 44, no. 7, pp. 405–417, 2002.

[19] "Docker SDK for Python - GitHub," https://github.com/docker/docker-py, Accessed 08-07-2018.

[20] "Kubernetes Python Client - GitHub," https://github.com/kubernetes-client/python, Accessed 08-07-2018.

[21] Hashicorp, "Consul," https://www.consul.io/, Accessed 08-07-2018.

[22] Kubernetes Documentation, "Jobs - Run to Completion," https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/, Accessed 08-07-2018.

[23] "[feature request] Swarm mode should support batch/cron jobs in addition to persistent services," https://github.com/kubernetes-client/python, Accessed 08-07-2018.

[24] Alex Ellis, "One-shot containers on Docker Swarm," https://blog.alexellis.io/containers-on-swarm/, March 11, 2017.

[25] Kubernetes Documentation, "Configure Multiple Schedulers," https://kubernetes.io/docs/tasks/administer-cluster/configure-multiple-schedulers/, Accessed 09-07-2018.

[26] Docker, "Scheduler Strategy," https://github.com/docker/docker.github.io/blob/master/swarm/scheduler/strategy.md, Accessed 09-07-2018.

[27] Kubernetes Documentation, "Pod Priority and Preemption," https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/, Accessed 09-07-2018.