Reproducible Distributed Clusters with Mutable Containers: To Minimize Cost and Provisioning Time

Hooman Peiro Sajjad
Royal Institute of Technology
Stockholm, Sweden
shps@kth.se

Kamal Hakimzadeh
Royal Institute of Technology
Stockholm, Sweden
mahh@kth.se

Shelan Perera
Royal Institute of Technology
Stockholm, Sweden
shelanp@kth.se

ABSTRACT

Reproducible and repeatable provisioning of large-scale distributed systems is laborious. The cost of virtual infrastructure and the provisioning complexity are two of the main concerns. The trade-offs between virtual machines (VMs) and containers, the most popular virtualization technologies, further complicate the problem. Although containers incur little overhead compared to VMs, VMs are required for their certain guarantees such as hardware isolation.

In this paper, we present a mutable container provisioning solution, enabling users to switch infrastructure between VMs and containers seamlessly. Our solution allows for significant infrastructure-cost optimizations. We discuss that immutable containers come short for certain provisioning scenarios. However, mutable containers can incur a large time overhead. To reduce the time overhead, we propose multiple provisioning-time optimizations. We implement our solution in Karamel, an open-sourced reproducible provisioning system. Based on our evaluation results, we discuss the cost-optimization opportunities and the time-optimization challenges of our new model.

CCS CONCEPTS

• **Networks → Cloud computing**; • **Computer systems organization → Cloud computing**;

KEYWORDS

Containers, Reproducible Clusters, Mutable, Provisioning, Cloud

ACM Reference format:
Hooman Peiro Sajjad, Kamal Hakimzadeh, and Shelan Perera. 2017. Reproducible Distributed Clusters with Mutable Containers: To Minimize Cost and Provisioning Time. In *Proceedings of HotConNet '17, Los Angeles, CA, USA, August 25, 2017,* 6 pages.
https://doi.org/10.1145/3094405.3094409

1 INTRODUCTION

Reproducible and *repeatable* provisioning of large-scale computing clusters (*cluster*) from Cloud resources has received increasing attention for recurring applications such as reproducible experiments of distributed systems [14], and repeatable per-job cluster allocation [12, 26, 29]. Repeatability refers to re-provisioning a cluster for a variable number of times without changing the cluster's *configuration parameters* (e.g., repeating an experiment to reduce errors). However, reproducibility deals with re-provisioning a cluster as the configuration parameters are modified [17] (e.g., scalability assessment of a system against different workloads). Due to the frequent repetitions in such applications, it is highly desirable to optimize the infrastructure cost along with the provisioning time of a cluster.

Cloud users have to make some trade-offs for choosing between virtual machines (VM) and containers (the de-facto standards for conveniently re-producing an infrastructure in Cloud [21]). VMs guarantee a higher level of isolation compared to containers, reducing the interference among applications by placing them on different VMs [28]. However, in contrast to VMs, containers do not run a guest operating system, introducing little to no overhead on host resources. *Neighboring containers* within the same host can be used to mitigate wasteful expenses of VMs that inflates the infrastructure cost. Specially, for applications with orthogonal resource-type requirements neighboring containers are more plausible. For example, in the phase of incremental experiment development and sanity check the neighboring containers are more suitable while as the experiment becomes more stable, VMs are preferred due to the hard resource isolation. As there are trade-offs among VMs and containers, the provisioning system has to offer an infrastructure-agnostic model such that it can seamlessly switch between VMs and containers.

Docker [22], a widespread container system, offers a light-weight and fast (sub-second) launch of container images [16, 28]. However, the fast deployment of containers requires pre-built docker images (also called *immutable* approach), which is not compatible with the requirements for a reproducible provisioning of distributed systems. A big shortcoming of the immutable approach is due to the tight dependency between the life-cycle of containers and their hosted services. The simultaneous launching of containers and their hosted services leaves no room for intermediate configurations steps that are necessary in some scenarios. For instance, in a cluster with a controller container that has to have SSH access into other containers, the following steps should be taken in order: (i) launch the controller container, (ii) generate an SSH key-pair in the controller, (iii) launch the other containers, (iv) copy the SSH public-key of the controller into the other containers (the authorized-host file in linux), (v) launch SSH service on the other containers, and (vi) run the controller service. As can be observed in this example, there is a loop in these configuration steps from the controller container into the other containers. Such orchestration scenarios have no clear solution in the immutable approach. Moreover, one has to

HotConNet '17, August 25, 2017, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5058-7/17/08…$15.00
https://doi.org/10.1145/3094405.3094409

construct numerous container images in order to cover all the required cluster setups according to different values for configuration parameters (e.g., in experiments). Even though Docker containers are fast in start/destroy, the image construction and versioning are not so fast because of the layered file system with copy-on-write. Thereby, due to the overhead of image construction and the orchestration problem, there is a need of a mutable approach for containers provisioning.

To our knowledge, this is the first work that considers mutable containers for an infrastructure-agnostic provisioning model. The mutable approach enables us to switch the infrastructure type between VMs and containers without re-configuring any application setting and with minimal cluster configurations. To have a compatible VM-container model, we enable containers with multi-host networking, which is not trivial. In our mutable approach, the layered file system with copy-on-write in Docker puts overhead on the installation. We explain the bottlenecks that cause excessive provisioning time and our optimizations for mitigating them. We integrate[1] our solution in Karamel [20], a recent provisioning system. Finally, based on our evaluation results, we discuss cost-optimization opportunities and time-optimization challenges of our new model for efficient reproducible and repeatable provisioning of containerized clusters.

2 BACKGROUND

In this section, we give some insights about Docker, the container platform, and Karamel, the provisioning engine, that we used in our implementation.

2.1 Docker

Docker [22] is a container platform that is known for its image versioning system and fast deployments. *Docker-engine* is the management component of Docker that should be installed on the hosting machines containers. Docker uses a copy-on-write file system to store the files inside containers. Copy-on-write semantic enables docker with a rich versioning of immutable images.

As Docker was, primarily, aimed for single host deployment of containers, its default networking option was virtual bridged networking [7]. Overlay networking is the suggested way of networking for connecting multiple hosts [8]. Overlay networking requires an underlying distributed key-value store service (e.g., consul, etcd, and Zookeeper). A user should configure a key-value store service and pass its configuration information to Docker daemon. Furthermore, to create a successful overlay network, the hosting machines should be able to communicate with each other.

Docker uses union file system [11], which operates by creating layers. It is possible to bypass this intermediate storage layer and write directly to the host's file system, for example for permanent data after terminating a container. Further, if data needs to be shared among containers, a shared host directory can be used. Docker data volumes assist in managing shared and persistent data independent of the container's life cycle.

[1]The source code is available at https://github.com/karamelchef/karamel/tree/docker

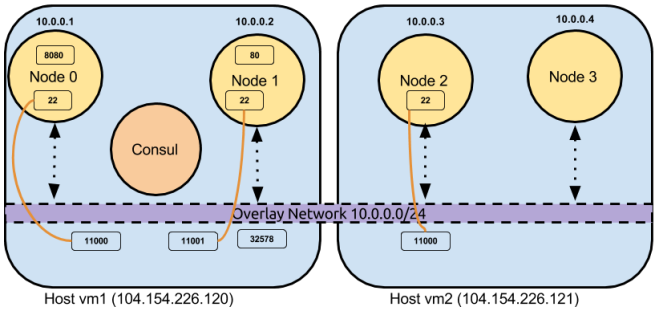


Figure 1: Overlay networking

2.2 Karamel

Karamel [19, 20], is a recent provisioning system for end-to-end provisioning of large-scale clusters in public clouds (e.g., Amzon, Google, OpenStack) as well as private clouds (e.g. bare-metal, OpenStack). Over the past two years, several systems use Karamel for provisioning [13, 15, 18, 23]. In those systems, Karamel has been repeatedly used to provision clusters from a single baremetal machine up to more than 100 servers on Amazon EC2, many of which were experimental clusters [15, 25].

The provisioning model in Karamel is based on *distributed idempotent functions* (DIF). As the DIFs are distributed in a cluster, Karamel supplies arguments of DIFs with a set of dataflow operations (e.g, pipe, set, split). With the functional model, Karamel supports heterogeneous provisioning tasks from infrastructure (e.g., hosts, network, and disk) to software (e.g. download, installation, configuration, and launch services). The dataflow mechanism improves the automation by embedding the dependencies of DIFs inside them, as the dependencies are extracted at run-time for a fast construction of execution plans. The scheduler in Karamel, which is a combination of a token based traversal of the execution plan and a FIFO based ordering at hosts offers a maximum parallelism by running independent DIFs in parallel.

Karamel has a collection of pre-defined infrastructure DIFs implemented in Java, and it uses Chef [4], a well-known configuration management framework, for the low-level coding of the software DIFs. Karamel introduces a minimum effort extension on Chef so as to leverage a great body of already developed Chef code easily.

The modularity and re-usability of the DIFs enables clusters to be defined in a brief *Cluster Definition Language*. In cluster definitions, users specify host-groups, number of machines, DIF-host assignments, and configuration parameters.

Last but not least, for Karamel, hosts in a cluster are network addressable entities with a SSH connection as all the configurations and installations happen through SSH commands. These abstract models (DIFs and hosts) are very convenient grounds to base our heterogeneous (container, VM, and bare-metal) cluster solution on.

3 PROVISIONING MODEL

In this section, we explain our key design decisions to enable an infrastructure-agnostic provisioning model including mutable containers, multi-host networking, and some container-dependent optimizations to reduce the provisioning time.

Algorithm 1 Recurring container-based provisioning

```
1:  procedure PROVISION(cluster-definition cd)
2:      hosts ← prepareHostingMachines(cd)
3:      nodes ← hosts
4:      if container-based then                          ▷ only container clusters
5:          engines = installDockerEngines(hosts)
6:          configureOverlayNetworking(engines)
7:          enableCaching(hosts)                         ▷ download cache
8:          downloadBasicContainerImage(cd, hosts)
9:      end if
10:     while tries < cd.tries do                        ▷ repititions
11:         if container-based then
12:             reservePortMapping(cd, hosts, containers)
13:             containers ← launchContainers(cd)
14:             nodes ← containers
15:         end if
16:         softwareProvisioing(cd, nodes)               ▷ use cache
17:         ...                                          ▷ run experiment/job
18:         terminate(cd, nodes)
19:     end while
20: end procedure
```

3.1 Mutable Containers

Our solution is based on mutable containers in which after launching a base Docker image, during runtime, we do software installations, necessary configurations such as network configuration (Section 3.2), and services orchestration. Treating containers as mutable infrastructure enables us to separate cluster definitions, and software orchestration logic from the Docker technology and consequently, provides a portable provisioning model between VMs and containers. In addition, this provides opportunities for provisioning-time optimizations (Section 3.4). Optimizations such as automatic image management to provide semi-mutable images can be done by the provisioning system instead of users, therefore, relieving users from analysing dependencies among software components. We consider such automated optimizations as future work.

3.2 Multi-host Networking

Provisioning of distributed systems across multiple hosting machines creates two primary network requirements for containers including the connectivity: (i) among containers within a cluster (e.g., for master/slave or p2p services), and (ii) from outside of the cluster (e.g., to access web consoles from the Internet).

To satisfy the first network requirement, we employ a networking overlay in Docker backed by a distributed key-value store, Consul [5], for network resolutions and service discoveries. We run Consul as a container.

To enable connectivity from outside the cluster, we use port-mapping from container ports to the ports of the hosting machines. Our port-mapping has two phases (i) port-reservation for the ports that are used in the provisioning, and (ii) random port assignment for the rest of the public ports. For example, in Figure 1, we reserve the ports 11000 and 11001 for the mappings of SSH ports (22 by default) in the containers (node0 and node1). These reserved ports are known and used by Karamel's provisioning engine (Section 3.3).

3.3 Integration with Karamel

The provisioning engine in Karamel has a portable model for infrastructure. Having network communications enabled for containers, Karamel dataflow engine can treat containers like VMs and bare-metals for software installation, configurations, and service start-ups. Algorithm 1 briefly shows all the steps for the provisioning of a cluster. The infrastructure-agnostic (VM, container, and bare-metal) provisioning is supported by the flipping 'container-based' flag (lines 4 and 11). These two if-statement blocks show the extra steps that we add to plug Docker containers into Karamel.

Table 1: Standard Machine Types in GCE

Machine Type	No of Virtual CPUs	Memory (GB)	Price ($)
n1-standard-1	1	3.75	0.0475
n1-standard-2	2	7.5	0.0950
n1-standard-4	4	15	0.1900
n1-standard-8	8	30	0.3800

3.4 Optimizations

Despite the fact that Karamel performs all operations with maximal parallelism (by converting them into a DAG, e.g, Docker engine installation at line 5), the extra steps for the container-based infrastructure increase the provisioning time compared to the mere VM-based infrastructure. Our goal is to minimize the extra provisioning time.

We skip the redundant downloads of binaries, by caching binaries inside hosts (line 7) and sharing them between the neighboring containers inside each host. This reduces network traffic and download times. The cache folder is mounted as a Docker volume in containers.

Downloading container images is network intensive and the download time depends on the size of the image. In the provisioning of Docker containers, even though each cluster requires its own software components to be installed, there are certain components that are common among all experiments running in Karamel. Therefore, we can pre-build a basic Docker image to speed-up provisioning time (semi-immutable approach, line 8). For example, all containers need to be SSH-enabled with a passwordless sudo user. Some installations are Karamel-specific (e.g., chef and berkshelf) and some are cluster-specific (e.g., JDK). We add the common installations into our basic pre-constructed container image. We leave further optimizations based on common installations for future work.

As the provisioning is repeated, some of the steps can be reused in the subsequent runs after the first one (we skip lines 4-9 after the first run). For instance, as we run an experiment multiple times (line 17), the provisioning time after the first run includes only the time to terminate and re-launch the containers. This consumes considerably less time than the first run of the experiment.

4 EVALUATION

In this section, we present some preliminary results based on our experiments that open up new avenues for a cost-optimized reproducible and repeatable provisioning system. Further, we analyze the challenges posed by the integration of containers in order to reach a time-optimized provisioning of clusters. In sections 4.2 and 4.3, we

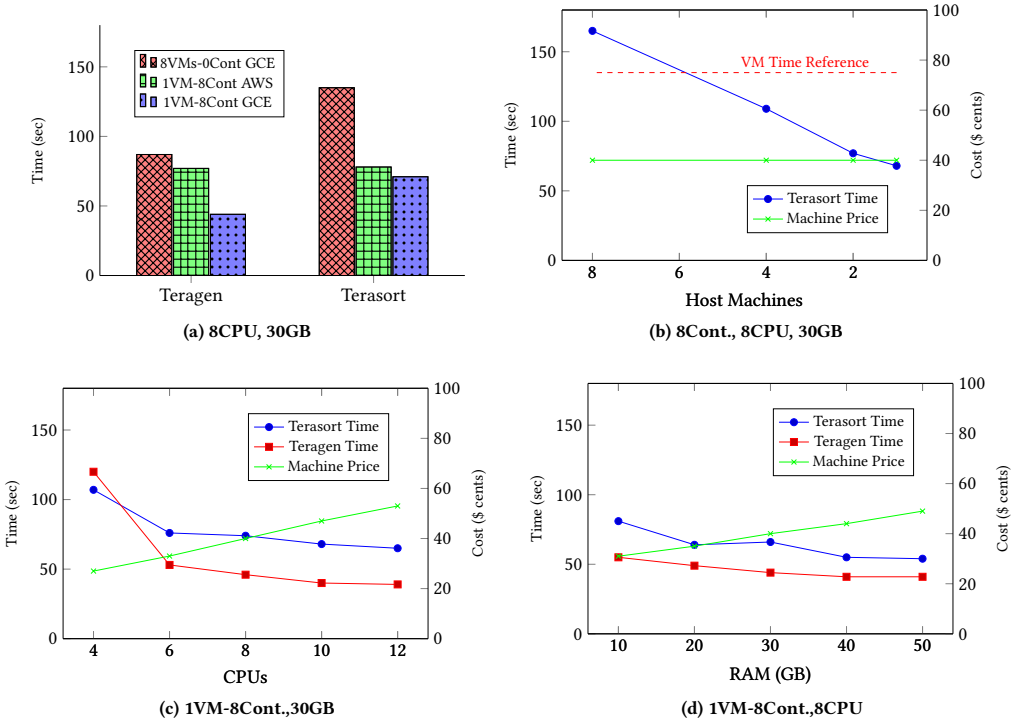


Figure 2: 10GB Teragen and Terasort with 8 nodes

Table 2: Approximate Resource Wastage

Machine Type	CPU idle%	Memory used%
n1-standard-8 (1 instance)	2.30	3.3
n1-standard-1 (8 instances)	18.40	26.4

evaluate our container-based provisioning solution implemented in Karamel against the Karamel's original VM-based solution.

4.1 Cost-Optimization Opportunities

Containers enable us to pack multiple software components in a VM, employing fewer numbers of more powerful VMs rather than employing plenty of weak VMs. Beside reducing the network overhead for network intensive applications, the packing enables several cost-optimization opportunities.

In Table 1, the specifications of some of the machine types in Google Compute Engine (GCE) [9] are given. There is no difference among the machine types based on their pricing per Virtual CPU (vCPU) and memory unit (GB). For example, for the same price, we can build an infrastructure including 8 vCPUs and 30 GB RAM by either using 8 instances of the n1-standard-1 machine or using a single instance of the n1-standard-8 machine. However, due to the resource wastage, 8 instances of n1-standard-1 gives us less effective resources than a single instance of the VM type n1-standard-8. This is mainly because of the overhead incurred by the operating system and the dependent software in each VM. We approximate the resource wastage in VM instances by obtaining the CPU idle percentage and the memory usage from GCE machines. In Table 2 can be seen that by using 8 instances of n1-standard-1 we waste around 7 GB of memory and 1.3 vCPUs, which costs more than having an extra n1-standard-1 instance.

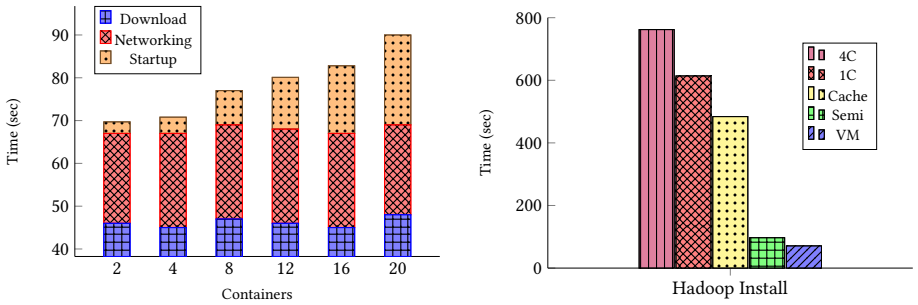
In addition, the flexibility in packing multiple containers in a VM enables us to use more customized machine types, considerably reducing the expenses, e.g., up to 40% in GCE [6].

Another opportunity for the cost reduction is by using *transient servers* (spot instances in EC2 [1] or preemptibel instances in GCE [10]). Transient servers can be bidden and can be used unless the bid amount is surpassed by the market pricing. Transient instances can be cheaper than the on-demand instances up to 10 times. Usually there are lower demands for powerful transient instances compared to more-popular weak machines with lower to medium specifications. Therefore, there is a great opportunity for a significant cost saving by using fewer powerful transient servers in applications such as experiments and tests.

4.2 Containers vs VMs

We compare time and cost efficiency of containers versus VMs through a series of experiments with a cluster setup of 8 nodes (see Figure 2). We run Terasort benchmark [24] on a Hadoop cluster [3] to sort 10 GB of data generated by Teragen.

First, we compare a VM-based cluster (8 n1-standard-1 VMs) with two container-based clusters (8 neighboring containers): (i) one m3.8xlarge on EC2, and (ii) one n1-standard-8 on GCE (see



(a) Container Setup **(b) Optimizations**

Figure 3: Optimizations on Container Provisioning

Figure 2a). Despite having an equal amount of total resources (8 vCPUs and 30 GB RAM) in all these clusters and an equal cost for GCE clusters, the neighboring-container clusters outperform the VM based cluster. Two potential reasons, the networking overhead and the resource wastage, can explain this difference. The difference between GCE and EC2 cluster is not clear for us but we have decided to do rest of the experiments on GCE.

Second, we further study the causes of networking and resource wastage by distributing containers across variable number of VMs (see Figure 2b). As can be seen, the total time decreases almost linearly as we pack more containers into less number of host VMs. Likewise, the total amount of resources and the cost of cluster are equal in all runs.

Then, we extend the single VM setup (8 neighboring containers) by varying the amount of resources allocated into the VM (custom VMs in GCE). As can be seen in Figures 2c and 2d, Terasort is still more timely and economical with the minimum allocated resources, 107s with 4 vCPUs and 81s with 10 GB RAM, compared to VM-based which takes 135s. We believe this improvement is because of two reasons: (i) the lack of networking overhead in the neighboring containers, and (ii) the soft resource isolation between neighboring containers that allows idle resources to be used by other containers.

4.3 Provisioning-Time Optimization

In our observations, the setup phase of the containers and installations are the two major time-consuming components in our container-based provisioning model (see Figure 3). In this section, we measure the the effect of different factors on the provisioning time. These factors are the scale (the number of containers), the download time, and the service-host placement. In the end, we show the effect of our optimizations on reducing the time wastage compared to an equivalent VM-based setup.

Figure 3a shows that the setup phase adds an overhead of 60-90 seconds (for 2-20 containers inside one hosting machine). As we can observe, the time of launching containers increases linearly as we increase the number of containers. As we do not perform downloading container's image and setting up networking repeatedly (see lines 6 and 8 in Algorithm 1), these steps are not very significant.

In the next experiment (see Figure 3b), we compare Hadoop's installation phase in a cluster of 4 Hadoop components - NameNode (NN), Resource Manager (RM), DataNode (DN) and NodeManager (NM). We use 4 VMs (2 vCPUs and 2GB RAM each) for the VM-based setup such that each VM hosts a separate component. To test the effect of component placement, first we place each Hadoop component in a separate container (4 containers on 2 VMs, we call it *4C*) then we co-locate all components inside a similar container (*1C*). On top of the co-located components, first, we enable cache repository for downloads (*cache*), then we use the docker image with a pre-installed JDK (*semi*).

Figure 3b demonstrates how much a naive installation (without any optimizations) can suffer from time inefficiencies (762s in 4C compared to 71s in VM). Note that this is a recurring overhead that happens in all iterations of a repeating job, e.g., an experiment. The co-location of components helps to mitigate the installation time from 762s to 614s. This is because all Hadoop components need to install Hadoop, in 1C it happens only once while in 4C it happens 4 times but in parallel. This is an important aspect in provisioning of distributed systems. It is more time efficient in provisioning to place several components from a similar system in a container with more resources than individual containers. But then we have to trade-off the co-location of the components with the lost of isolation between the components.

Enabling the download caching (Figure 3b) in the 1C setup further reduces the installation time (from 614s to 484s) but by using the pre-constructed docker image (semi) we observe an abrupt drop in the installation time (from 484s to 97s) which is very close to the provisioning time of the VM-based setup (71s).

These results suggest that the time overhead of the container-based provisioning is mitigatable with the illustrated optimization techniques. This is an enabler for a flexible cluster provisioning between container and VM with time and cost benefits.

5 RELATED WORK

Docker [22] enables portability of containers across different platforms with minimal overhead [16]. Docker provides a limited orchestration service, *Docker-Compose*, to control the order of services startup. However, Docker-Compose does not provide the semantics of wait until a service is 'ready', but rather it can only wait until a container is running. Orchestration in Kubernetes [27] happens through complex orchestration rule definitions, and data-dependencies through environment variables or the *service etcd* (a

distributed key-value store to discover services). Neither Docker nor Kubernetes provides a switchable model between VMs and containers. Ansible [2] is an open-sourced automation engine that supports both VM- and container-based provisioning. Our model, optimizations, and results are complimentary to Ansible and can be integrated in such an automation engine. Karamel [20] is another automation engine that does not support Docker containers in which we integrate our solution.

6 CONCLUSIONS

In this paper, we explained that containers can effectively reduce the cost in reproducible and repeatable provisioning of distributed systems. However, containers do not replace VMs. Therefore, we propose a model based on mutable containers that enables reproducible provisioning of distributed systems and provides an infrastructure-agnostic model that can inter-operate between VMs and containers. Our model provides a ground for several automated optimizations to move toward semi-mutable containers that can benefit from fast image launch.

REFERENCES

[1] *Amazon EC2 Spot Instances.* Retrieved 2017-03-24 from https://aws.amazon.com/ec2/spot/

[2] *Ansible automation engine.* Retrieved 2017-03-24 from https://www.ansible.com/

[3] *Apache Hadoop Home Page.* Retrieved 2017-03-20 from https://hadoop.apache.org

[4] *Automated Configuration Management.* Retrieved 2017-03-20 from https://www.chef.io

[5] *Consul by HashiCorp.* Retrieved 2017-03-20 from https://www.consul.io/

[6] *Custom machine types.* Retrieved 2017-03-23 from https://cloud.google.com/compute/pricing#custommachinetypepricing

[7] *Customize the docker0 Bridge.* Retrieved 2017-03-20 from https://docs.docker.com/engine/userguide/networking/default_network/custom-docker0/

[8] *Docker Libnetwork Project.* Retrieved 2017-03-20 from https://github.com/docker/libnetwork

[9] *Google Compute Engine.* Retrieved 2017-03-20 from https://cloud.google.com/compute/

[10] *Preemptible VM Instances.* Retrieved 2017-03-24 from https://cloud.google.com/compute/docs/instances/preemptible

[11] *Union File System.* Retrieved 2017-03-20 from https://docs.docker.com/engine/reference/glossary/#/union-file-system

[12] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. USENIX Association.

[13] Alysson Bessani, Jörgen Brandt, Marc Bux, Vinicius Cogo, Lora Dimitrova, Jim Dowling, Ali Gholami, Kamal Hakimzadeh, Micheal Hummel, Mahmoud Ismail, et al. 2015. BiobankCloud: a platform for the secure storage, sharing, and processing of large biomedical data sets. In *the First International Workshop on Data Management and Analytics for Medicine and Healthcare (DMAH 2015)*.

[14] Tomasz Buchert, Cristian Ruiz, Lucas Nussbaum, and Olivier Richard. 2015. A survey of general-purpose experiment management tools for distributed systems. *Future Generation Computer Systems* 45 (2015), 1–12.

[15] Marc Bux, Jörgen Brandt, Carsten Lipka, Kamal Hakimzadeh, Jim Dowling, and Ulf Leser. 2015. SAASFEE: scalable scientific workflow execution engine. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1892–1895.

[16] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2014. An Updated Performance Comparison of Virtual Machines and Linux Containers. *Technology* 25482 (2014), 171–172. https://doi.org/10.1109/ISPASS.2015.7095802

[17] Juliana Freire, Philippe Bonnet, and Dennis Shasha. 2012. Computational reproducibility: state-of-the-art, challenges, and database research opportunities. In *Proc. of SIGMOD'12*. ACM, 593–596.

[18] Kamal Hakimzadeh, Hooman Peiro Sajjad, and Jim Dowling. 2014. Scaling HDFS with a strongly consistent relational model for metadata. In *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 38–51.

[19] Karamel. *One Click Installation for Clusters.* Retrieved 2017-03-20 from http://www.karamel.io/

[20] Karamel. *Reproducing Distributed Systems on Cloud.* Retrieved 2017-03-20 from https://github.com/karamelchef/karamel

[21] Chris Nyberg Ordinal Technology Corp. Naga Govindaraju Microsoft (Azure) Mehul A. Shah, Amiato. 2014. *CloudSort: A TCO Sort Benchmark.* Retrieved 2017-03-16 from http://sortbenchmark.org/2014_06_CloudSort_v_0_4.pdf

[22] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[23] Salman Niazi, Mahmoud Ismail, Steffen Grohsschmiedt, Mikael Ronström, Seif Haridi, and Jim Dowling. 2017. HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases. In *FAST*. 89–103.

[24] Owen OâĂŹMalley. 2008. Terabyte sort on apache hadoop. *Yahoo, available online at: http://sortbenchmark. org/Yahoo-Hadoop. pdf,(May)* (2008), 1–3.

[25] Shelan Perera, Ashansa Perera, and Kamal Hakimzadeh. 2016. Reproducible Experiments for Comparing Apache Flink and Apache Spark on Public Clouds. *arXiv preprint arXiv:1693949* (2016).

[26] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. 2016. PerfOrator: eloquent performance models for Resource Optimization. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 415–427.

[27] David K. Rensin. 2015. *Kubernetes - Scheduling the Future at Cloud Scale.* 1005 Gravenstein Highway North Sebastopol, CA 95472. All pages. http://www.oreilly.com/webops-perf/free/kubernetes.csp

[28] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. 2016. Containers and Virtual Machines at Scale: A Comparative Study. In *Proceedings of the 17th International Middleware Conference*. ACM, 1.

[29] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, 363–378.