FastContainer: A Homeostatic System Architecture High-speed Adapting Execution Environment Changes

Ryosuke Matsumoto
SAKURA Internet Inc.
Email: r-matsumoto@sakura.ad.jp

Uchio Kondo
GMO Pepabo, Inc.
Email: udzura@pepabo.com

Kentaro Kuribayashi
GMO Pepabo, Inc.
Email: antipop@pepabo.com

Abstract—**Price reduction and performance improvement of cloud computing and web hosting services by making use of container virtualization technology engender greater demands on the efficiency of highly used multi-tenant user environments and for maintenance, security, and appropriate resource management. However, maintaining the availability and load balancing on access congestion remain dependent on the configurations of the respective systems. As described herein, we propose a homeostatic system architecture that rapidly adapts to execution environment changes. It reactively decides invocation, running periods, simultaneous running numbers, and assigned resources of the containers according to the incoming HTTP requests. Our proposed architecture enables automatic and rapid load balancing by generating and discarding the containers following the access frequency in cases of access congestion. The method improves resource utilization efficiency by automatically discarding the containers within a fixed period, which contributes to increasing opportunities for reflecting the library updates.**

1. Introduction

Opportunities for users to express themselves on the internet are increasing along with the diversification of companies and individuals working on the internet. Especially for individuals, by spreading contents created using Twitter or Facebook, it has become possible to increase the number of visits to contents efficiently. Consequently, high quality contents are spread further. It is becoming possible to brand-identify individuals according to their linked contents. Generally, web hosting services and cloud services are used for individuals to distribute web contents [13].

Along with price reduction and performance improvement of web hosting services and cloud services, OS virtualization technology [3] is used to provide stable and secure execution of multiple execution environments of web applications on a single web server. Among OS virtualization technologies, using a container-type virtualization technology [17] that can manage resources by isolating a user area on a per-process basis, one can more efficiently accommodate multiple execution environments than virtual machines can.

For cloud services, users must implement a mechanism of autoscale [16] that can withstand access concentration. It is necessary to use the autoscale functions provided by the respective service providers. With an autoscale function provided by a service provider such as AWS, based on monitoring items provided by the service provider itself, it is necessary to start up a self-built virtual machine internally or to use an external service. Because it is necessary to start the virtual machine automatically, it takes time to do scale processing against burst access concentration. In many cases, the load distribution is not in time [7]. Furthermore, it is difficult for users with insufficient technical knowledge to construct a load distribution mechanism quickly.

As described in this paper, on the premise that people with knowledge sufficient to use only the standard web hosting service distribute web contents without technical expert knowledge, we propose a system architecture by which service users need not construct a load distribution system and operate/manage libraries. This architecture is homeostatic. It can quickly adapt to changing the execution environment by reactively determining startup of a container in an HTTP request. The ”reactive determination” described here represents detection of the load state and response performance based on external factors on an HTTP request basis, and ascertaining of the configuration of the container quickly according to the situation. We designate this architecture as FastContainer.

As a concrete method to realize the load balancing process quickly after clearly separating the user data and the application process in the execution of the web application, the architecture determines the state of a container, such as activation or inactivation, continuation of the activation time, resource allocation, and the number of activation containers each HTTP requests. Furthermore, to activate the web application executed on the container at high speed, the web application process at the time of completion of the startup process is imaged by CRIU [4] and is restored from the image for changing the states. In addition, the container stops in a certain period because the changing state time of the process is short. This feature increases the resource efficiency of the multi-tenant system [11]. Simultaneously, when the library is updated, the containers are updated to a new state quickly.

The structure of this paper is the following. We explain autoscale and its tasks in web hosting services and cloud services in section 2. In section 3, we describe the architecture, called FastContainer. Implementation of the proposed method to solve the difficulty is presented in section 2. In section 4, we evaluate the response time for scaling and for changing the statistics using the process image. We summarize it in section 5.

2. Load distribution and operation technology

In situations in which access is most concentrated, a high probability exists that contents diffuse widely. The server becomes a massive load state and access becomes difficult. Often the opportunity of valuable content diffusion is missed. In this section, we organize the autoscale and related operation technologies for load balancing in web hosting service and cloud service.

The scale methods for load distribution have a scale-up type that increases the hardware resources allocated to a single instance like Virtual Machine (VM) and a scale-out type that increases the activation number of instance.

2.1. Web hosting services

In the web hosting service [13], the web content of the service user is accommodated in a specific web server. The web server and the web content are linked. Autoscale corresponding to the load is difficult in terms of data consistency. Therefore, it is difficult to control the resources used on a per-host basis properly or to investigate the cause quickly. It is not possible to calculate the scale at the time of scale or add as much resources as necessary when the user requires the autoscale. Additionally, it is difficult for the hosting service to respond quickly to the load with a scale-up type.

From the viewpoint of operational technology, when updating the library, the effect of restarting the server process becomes large because of the characteristic that many hosts run on a single server process. Furthermore, because it is difficult to limit the resources appropriately when the server is under heavy load, it is costly to investigate and control the cause of high load. The effect on service quality is also considerable.

Regarding service users, middleware and functions, including available web server software, are common to all hosts. The degrees of freedom in system construction are few.

2.2. Cloud services

Cloud service is a service that provides cloud computing [12] as various services. For a cloud service, it is necessary for the service users themselves build not only the web contents, but also the web server software and the database. Therefore, although it has a high degree of freedom in terms of being able to design the system individually for load balancing, it requires expert knowledge. As for autoscale, cloud providers provide the function of increasing or decreasing instances according to the load [1]. However, the monitoring interval of the load and the startup time of the instance are rate limiting. The time to detect against sudden access lengthens.

Even if a virtual machine starts up under a high load situation, the process itself for autoscale can not catch up at times of sudden high loads such as the influence of television broadcasting, often leading to service stoppage. In addition, a method of using containers exists to solve the difficulty of starting time of virtual machines [7] or a service that can define detailed conditions for scaling by external service cooperation [9]. However, the startup processing of web application server processes such as Ruby on Rails [14] on containers is still slow. Immediacy is low against a sudden load.

Generally, to respond quickly to a high-load state, a method of activating a virtual machine of an expected amount to some degree is taken in advance. However, it is difficult to form an appropriate estimate from the balance of limited costs such as hardware, operation, and service level.

The AWS of the cloud service provider determines the computer resource automatically by installing an application by the notation specified by the provider and automatically scaling the computer resource on the provider side under high load such as AWS Lambda [2]. However, these services are intended for engineers who have technical knowledge. When using such a service, it is challenging to autoscale after publishing web contents without technical knowledge for users targeted by the web hosting service.

3. Proposed method

Considering the characteristics of various current services, controlling the instant instantly under the load within a limited resource range is necessary. On the premise of a web server function with reactiveness, an architecture must manage instances flexibly. Moreover, it must ensure performance that does not entail difficulties in practical use. The requirements are summarized below.

- The architecture can scale-up and scale-up instances quickly with a granularity of HTTP request units.
- The architecture monitors the instance at a granularity of HTTP request units and issue scale processing instruction of the instance.
- To improve the resource efficiency of servers, the architecture stops unnecessary instances. It can activate the instances with an HTTP request trigger when necessary.

Furthermore, this architecture has the following requirement as a web hosting service.

- The service provider supports server operation, such as by update of OS and the library.
- The service provider supports a widely used general web application such as WordPress.
- The service provider supports autoscale when the load concentrates, even if there is no specialized technical knowledge related to the load distribution.

- The service provider supports pay-per-use at the granularity of about the web application execution time.
- The service reduces hardware costs by increasing the host accommodation efficiency.
- The service updates OS and libraries to ensure security at high frequencies.

As described in this paper, to make it possible to deploy the user area quickly to multiple servers when distributing web contents, we propose an architecture that the service user need not construct the load distribution system or manage the libraries. This architecture reactively determines changes of states such as web application container startup, startup duration, the number of containers, and scale processing judgment for each HTTP request. This architecture can adapt quickly to changes in the execution environment. Moreover, it has homeostasis. We designate this architecture as FastContainer.

The FastContainer architecture uses Linux containers as instances, not virtual machines. The container on Linux [5] is a virtualization technology to isolate the OS environment virtually at the process level while sharing the kernel.

3.1. Reactive and mortal architecture

The FastContainer architecture combines the benefit that the container can start up faster than the virtual machine and that the architecture secures performance while improving resource efficiency by high-speed adapting execution environment changes. The FastContainer architecture separates the data and application processing in the execution processing of the web application started on the container. Furthermore, under the load state and the response performance for each HTTP request, the activation processing of the web application container, the activation duration time, the container activation number, and the resource allocation are all determined reactively.

In the proposed method, an immediate response is transmitted if one or more containers is activated. When the container stops, the processes are performed for a certain period after starting the container with the request as a trigger. As a result, even if all the containers stop, the availability becomes high because the container startup on a per-request basis. Additionally, by activating the container for a certain period, once the container is activated, the response can be transmitted without affecting the activation time. FastContainer is a reactive and mortal architecture.

At the time of access concentration, the already-started container monitors the throttled value of cgroup [15], indicating failure of the allocation of the CPU own time of the container itself, scaling out if 80% or more failed, 5 min. If the average failure value is less than 10%, the container that scaled out is stopped based on the container startup duration.

When scaling out, it automatically registers the container information of the new containers to the configuration management database (CMDB) via the management API. Subsequently the web proxy located in front of the container transfers the request to the new container based on the configuration management database.

In this transfer operation, the web dispatcher on the server containing the containers transfers the request to the specified container if the container is active. If it is not activated, then the container configuration information is acquired from the CMDB, the request is transferred after activating the container first. However, if the container is running, then the request is transferred only to the already activated container so that the architecture can distribute to a new container after the container completes the startup processing.

FastContainer saves data related to web applications on shared storage. If a server group containing containers is mounted in the same area, irrespective of which server the container is activated, then it can operate correctly on the CMDB based on the container configuration information.

To realize the FastContainer architecture as a system, complicated control of containers is necessary. Kondo et al. developed Haconiwa [6], and we use Haconiwa for FastContainer implementation. In addition to configuring container resource allocation and process isolation configuration information, Haconiwa can describe Ruby Domain-specific language (DSL) in various phases at container startup, shutdown, and container setting. It is a container runtime that can define container pluggable behaviors.

3.2. High-speed state transition for containers

As described in section 1, a method of using containers exists to solve the difficulty of the starting time of virtual machines. However, the startup processing of web application server processes such as Ruby on Rails on containers is still slow. Even for containers, immediacy is low against a sudden load.

To perform state transition at high speed, the web application process running on the container uses CRIU [4] to image the process state immediately after finishing the startup of the process. At the time of container startup, CRIU restore the process from that image. Even if using software with a long startup time such as Ruby on Rails, it can be started at high speed.

Features of the proposed method are that CRIU accelerates the container activation, that containers start-up reactively triggered by request, and that a managed process monitors resources for autoscaling each HTTP request. With these features, it is possible to autoscale rapidly against sudden loads.

As for scale-up, container resource management is controlled by cgroup on a per-process basis. Using the cgroup feature, an assignment such as CPU usage time can be changed immediately even if the process is running.

A container is discarded in a certain period, thereby reducing the number of unnecessary processes to be activated and increasing the container accommodation efficiency. Furthermore, when the library is updated, it is guaranteed to be updated to the new state at all times.

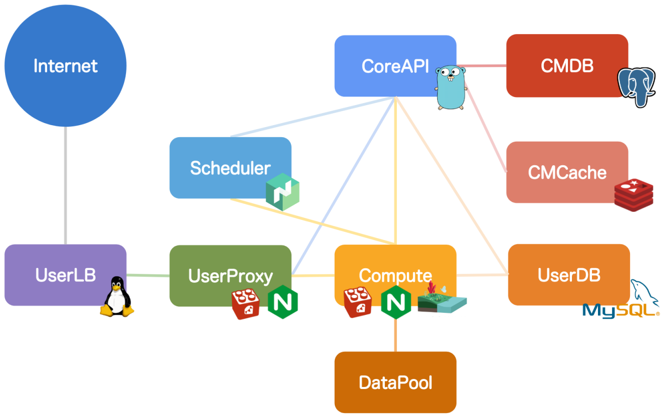

Figure 1. Example of FastContainer System using Haconiwa.

3.3. Differences from Serverless architecture

The Serverless architecture [10] does not operate the application on the infrastructure side, whereas the FastContainer architecture allows general CMS such as WordPress to be deployed or auto-scalable. Part of the infrastructure side corresponds to the application operation. In addition, because Serverless architecture similar to AWS Lambda requires predetermined coding by the user, it is intended mainly for engineers and researchers. By contrast, the FastContainer architecture can use general-purpose web applications and can be scaled on the infrastructure side when concentrating access. Users without engineering expertise receive the value of scale by this feature.

Heroku [8] reported a free plan to restrict the uptime of the container and the sleep time that is unable to handle the request after reactivating the container on a per-request basis to reduce the usage of server resources. Because Heroku's pay plan has no sleep time, it is apparently an architecture that processes requests stably using an approach that continues to activate containers while periodically restarting.

The FastContainer architecture speeds change of a state such as a stop, startup, and scaling process. Regarding resource efficiency, FastContainer circulates containers by reactively changing states. It produces change-resistant homeostasis. Those features are the primary objective of FastContainer.

4. Experiment

4.1. Evaluation of scale processing

To confirm the effectiveness of the FastContainer architecture, we constructed a prototype environment using FastContainer, as shown in Figure 1, and evaluated the scale processing of the container. Table 1 presents the experiment environment and the various roles. NIC and OS of each role of experiment environment were all 1 Gbps for NIC and Ubuntu 14.04 Kernel 4.4.0 for OS. For the experiment, we built only the role of table 1 and benchmark the container started with Compute from UserProxy. The experiment environment saves container data into DataPool, with NFS mounted from Compute to DataPool.

TABLE 1. EXPERIMENT ENVIRONMENT

	Items	Specifications
Compute	Server on which the container runs	
	CPU	Intel Xeon E5-2650 2.20 GHz 12 core
	Memory	39 GBytes
UserProxy	Transfer request to container based on CMDB	
	CPU	Intel Xeon E5620 2.40 GHz 4 core
	Memory	4 GBytes
CoreAPI	Control container configuration management information	
	CPU	Intel Xeon E5620 2.40 GHz 8 core
	Memory	8 GBytes
CMDB	Save container configuration management information	
	CPU	Intel Xeon E5620 2.40 GHz 4 core
	Memory	16 GBytes
DataPool	Contain container contents	
	CPU	Intel Xeon E5620 2.40 GHz 2 core
	Memory	4 GBytes

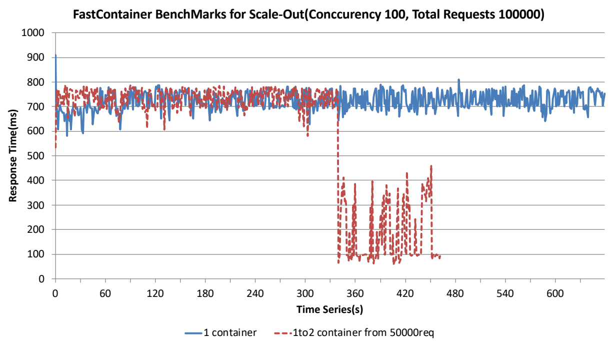

Figure 2. FastContainer Auto-Scale-Out.

On the container, activate Apache 2.4.10 in one process, install PHP 5.6.30, and execute the content that only executes the phpinfo( ) function to acquire PHP environment information. In addition, the maximum CPU usage of the container is limited to 30% of 1 core by the cgroup function. We adopted the benchmark-setting value that can use up 30% of CPU from preliminary experiments, 100 simultaneous connections, and 100,000 total requests.

We used the ab command for the benchmark. In this experiment, we make requests to CoreAPI for adding containers manually. We do not use CRIU in this experiment for evaluation by another experiment.

In the benchmark, the average value of response time per second was created as time series data and graphed. Subsequently, we run the benchmark. When the number of processing requests exceeds 50,000, execute the load correspondence of the scale-out type and the scale-up type. Then, we confirm that the scale processing runs immediately and that the response time becomes short. Scale-out type is shown in Figure 2 Scale-up type results are shown in Figure 3.

The denoted by ”1 container” of Figure 2 shows results

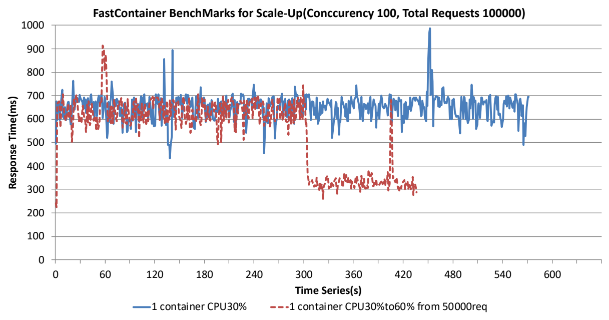

Figure 3. FastContainer Auto-Scale-Up.

for the benchmark for one container. During the benchmark, the limited maximum CPU usage rate of 30 % is always in use. In this state, the response time is about 700 ms, as shown generally. Additionally, increasing the number of concurrent connections does not further increase the number of failed requests processing. Therefore, this value is the maximum response time in which one container can process without failing to process the request.

Next, the graph indicated by ”1 to 2 container” descending from around the horizontal axis of 340 s is the result when the experiment system performs autoscale-out at 335 s on the horizontal axis that reached 50,000 requests. Furthermore, from the graph, which shows the high speed of the container rising in a few seconds and the auto scale-out process reactively activated for the request, quickly scaled out, the response time is inferred to be about half. The experiment system can perform auto scale-out at high speed, as shown in the graph. As a concrete numerical value, auto scale-out occurs at the time point of 335 s. Because the scaling process is being conducted until 339 s, the response time remains 700 ms. The experiment system completes the scale-out at 340 s. The response time for the second is between 100 ms and 400 ms.

In the case of ”1 container”, processing 100,000 requests took about 720 s. As the response time becomes short, the processing of 100,000 requests was completed in about 470 s, as shown in the graph.

”1 container” of Figure 3 shows a graph in the case of not doing any scale processing, similarly to ”1 container” of Figure 2. The graph of ”1 container CPU 30 %–60 %” descending from 300 s on the horizontal axis shows the response time transition when scaling up the CPU maximum usage rate to 60% at the horizontal axis 301 s.

In addition, from this graph, the experiment system immediately performs scale-up from the point of 301 s on the horizontal axis where the request processing reached 50,000 requests, without delay in long response time or failure in response processing.

As a concrete numerical value, auto scale-up occurred at 301 s. The response time started falling from 700 ms to 574 ms from 302 s. It became 358 ms at the time of 304 s and stabilized at around 350 ms after that.

The auto scale-up affects the immediate response time rather than the auto scale-out because it is easier to increase the resources of the direct operating container than scale-out. The performance cost of adding a new container for activation such as scale-out is high.

TABLE 2. Experiment Environment

	Items	Specifications
Client	Client server running benchmark	
	CPU	Intel Xeon E5-2650 2.20 GHz 1 core
	Memory	2 GBytes
Compute	Container accommodating server	
	CPU	Intel Xeon E5-2650 2.20 GHz 8 core
	Memory	51 GBytes
UserProxy	Forwarding request to container based on CMDB	
	CPU	Intel Xeon E5-2650 2.20 GHz 1 core
	Memory	2 GBytes
CoreAPI	Controlling container configuration management information	
	CPU	Intel Xeon E5-2650 2.20 GHz 1 core
	Memory	2 GBytes
CMDB	Save configuration management information of container	
	CPU	Intel Xeon E5-2650 2.20 GHz 1 core
	Memory	16 GBytes
DataPool	Contain container contents	
	Storage	NetApp FAS8200A
	FlashPool	8.73 TB

4.2. Evaluating startup speed from images

For this experiment, we used the CRIU to dump the image of processes in the startup completion state beforehand. We measured the response time when starting from the image according to the response.

Table 2 shows the experiment environment and the various roles. As in the experiment in the 4.1 section, we built a prototype environment using FastContainer as shown in Figure 1. NIC and OS of each role of experiment environment were all 1 Gbps for NIC, Ubuntu 16.04 for OS, and 4.4.0-59-generic for the kernel. Container data were saved on DataPool and were mounted from Compute to DataPool with NFSv3.

As a web application server, we used the default page of Ruby on Rails, which takes a long time to start up, for experiments. The version of Ruby on Rails is 5.1.3. The version of Ruby is 2.5.1. The experiment system allocates one CPU core and 1 GBytes memory of the Compute server to the container.

For comparison, we prepared containers with and without CRIU. For the experiment, we benchmarked Ruby on Rails running on the container using the ab command with the number of simultaneous connections is 1. When the benchmark has passed 40 s, the environment system stops the container. The FastContainer architecture starts reactively when the next request is received even if the container is down. At that time, we compared whether the response time differs between the case of starting from the image of processes and the case of not activating from the image.

Figure 4 shows the response time of the Rails container when not using CRIU. Figure 5 presents the response time when using CRIU. The default page of Ruby on Rails loads 67 gems, which are extension libraries of Ruby, and uses about 75 MBytes of RSS (the real memory resident set size

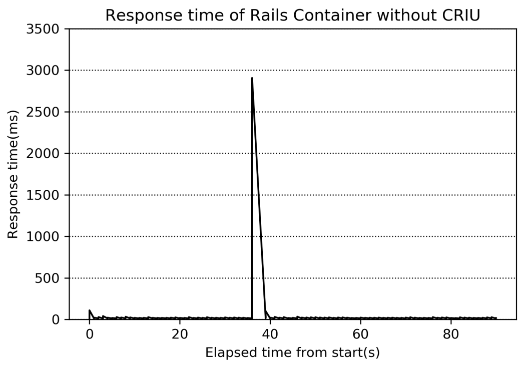

Figure 4. Response Time of Rails Container without CRIU.

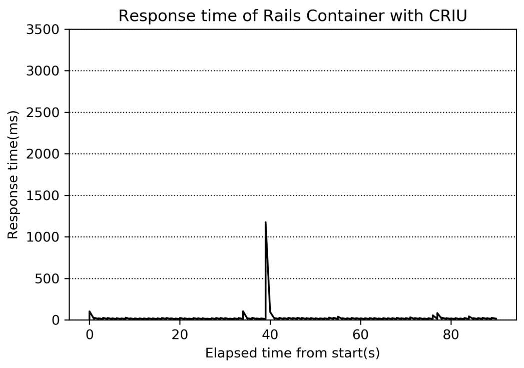

Figure 5. Response Time of Rails Container with CRIU.

of the process). In the case of such containers and web applications such as the Figure 4, if rails do not use CRIU, it will take about 2900 ms to start up. By contrast, when using CRIU, it is possible to start up in about 1200 ms and return a response such as that in Figure 5.

Using CRIU, even a server for which a startup time of the web application is slow can be activated quickly by starting from the image of processes completed in advance for the startup processing. As a result, using CRIU as the state transition of FastContainer, it can respond to the request reactively, without depending on the start time of the web application. For that reason, it can duplicate containers efficiently, even in autoscaling.

5. Conclusion

This study examined our proposed FastContainer of a container management architecture, which allows the user environment composed of containers to autoscale at HTTP request timing, without requiring specialized knowledge for service users in web hosting service. Even for a process that takes time to start up for a web application such as Ruby on Rails, by application of the process image restoration by CRIU, the time of the auto scale-out with reactivity is considerably short. Additionally, resource efficiency of FastContainer is higher by discarding the container during a certain period. When the library is updated, the frequency of updating to the new state also increases.

As we develop new software such as FastContainer architecture by ourselves and with other individuals, it is necessary to clarify the standing position of container-related software and the comparison target and to make the cooperation of each layer more generalized. For that purpose, we plan to continue research and development while considering collaboration with other tools and technical background firmly, in addition to discussion with tool developers.

References

[1] Amazon Web Services: Auto Scaling, https://aws.amazon.com/autoscaling/.

[2] AWS Lambda, https://aws.amazon.com/lambda/.

[3] Che J, Shi C, Yu Y, Lin W, A Synthetical Performance Evaluation of Openvz, Xen and KVM, IEEE Asia Pacific Services Computing Conference (APSCC), pp. 587-594, December 2010.

[4] CRIU. checkpoint/restore in userspace. http://criu.org/.

[5] Felter W, Ferreira A, Rajamony R, Rubio J, An Updated Performance Comparison of Virtual Machines and Linux Containers, IEEE International Symposium Performance Analysis of Systems and Software (ISPASS), pp. 171-172, March 2015.

[6] Haconiwa: MRuby on Container / A Linux container runtime using mruby DSL for configuration, control, and hooks, https://github.com/haconiwa/haconiwa.

[7] He S, Guo L, Guo Y, Wu C, Ghanem M, Han R, Elastic application container: A lightweight approach for cloud resource provisioning, Advanced Information Networking, And Applications (AINA 2012) IEEE 26th International Conference, pp. 15-22, March 2012.

[8] Middleton, Neil, and Schneeman, Richa, Heroku: Up and Running: Effortless Application Deployment and Scaling, O'Reilly Media, Inc., 2013.

[9] RightScale Cloud Management, https://www.rightscale.com/.

[10] M Roberts, Serverless Architectures, https://martinfowler.com/articles/serverless.html.

[11] Mietzner R, Metzger A, Leymann F, Pohl K, Variability Modeling to Support Customization and Deployment of Multitenant-aware Software as a Service Applications, the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, pp. 18-25, May 2009.

[12] P Mell, T Grance, The NIST Definition of Cloud Computing", US Nat'l. Inst. of Science and Technology, 2011, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[13] Prodan R, Ostermann S, A Survey and Taxonomy of Infrastructure as a Service and Web Hosting Cloud Providers, 10th IEEE/ACM International Conference on Grid Computing, pp. 17-25, October 2009.

[14] Ruby on Rails — A web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern, https://rubyonrails.org/.

[15] Rosen R, Resource Management: Linux Kernel Namespaces and cgroups, Haifux, May 2013.

[16] Ferdman M, Adileh A, Kocberber O, Volos S, Alisafaee M, Jevdjic D, Falsafi B, Clearing the clouds: a study of emerging scale-out workloads on modern hardware, ACM SIGPLAN Notices, Vol. 47, No. 4, pp. 37-48, March 2012.

[17] Soltesz S, Pötzl H, Fiuczynski M E, Bavier A, Peterson L, Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors, ACM SIGOPS Operating Systems Review, Vol. 41, No. 3, pp. 275-287, March 2007.