A Framework for Satisfying the Performance Requirements of Containerized Software Systems Through Multi-Versioning

Sara Gholami, Alireza Goli, Cor-Paul Bezemer
Department of Electrical and Computer Engineering
University of Alberta, Edmonton, Canada
{sgholami,goli,bezemer}@ualberta.ca

Hamzeh Khazaei
Department of Electrical Engineering & Computer Science
York University, Toronto, Canada
hkh@yorku.ca

ABSTRACT

With the increasing popularity and complexity of containerized software systems, satisfying the performance requirements of these systems becomes more challenging as well. While a common remedy to this problem is to increase the allocated amount of resources by scaling up or out, this remedy is not necessarily cost-effective and, therefore, often problematic for smaller companies.

In this paper, we study an alternative, more cost-effective approach for satisfying the performance requirements of containerized software systems. In particular, we investigate how we can satisfy such requirements by applying software multi-versioning to the system's resource-heavy containers. We present `DockerMV`, an open-source extension of the `Docker` framework, to support the multi-versioning of containerized software systems. We demonstrate the efficacy of multi-versioning for satisfying the performance requirements of containerized software systems through experiments on the `TeaStore`, a microservice reference test application, and `Znn`, a containerized news portal application. Our `DockerMV` extension can be used by software developers to introduce multi-versioning in their own containerized software systems, thereby better allowing them to meet the performance requirements of their systems.

CCS CONCEPTS

• **Software and its engineering → Software performance**.

KEYWORDS

Software Multi-versioning, Containerized Software Systems, Performance Requirements

ACM Reference Format:
Sara Gholami, Alireza Goli, Cor-Paul Bezemer and Hamzeh Khazaei. 2020. A Framework for Satisfying the Performance Requirements of Containerized Software Systems Through Multi-Versioning. In *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering (ICPE '20), April 20–24, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3358960.3379125

ICPE '20, April 20–24, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-6991-6/20/04…$15.00
https://doi.org/10.1145/3358960.3379125

1 INTRODUCTION

As the popularity and complexity of software systems increase, it becomes more challenging to satisfy the performance requirements of such systems. For example, one common problem that may happen for a web-based software system is the Slashdot effect. The Slashdot effect is a resource allocation problem that happens when a high-traffic website posts a link to a low-traffic website [1, 2]. If the low-traffic website is not capable of handling the sudden increase in traffic, it may experience prolonged response times or unavailability, thereby violating the website's performance requirements. One common remedy to this problem is to allocate more server resources to make sure that the performance of the website satisfies the requirements. However, this approach can become very expensive and could add high over-provisioning costs, which not every project can afford. An alternative solution could be to have different versions of the services provided by the website. For instance, if the website had lightweight versions of some of its essential, resource-heavy components, it could use them during the high load to reduce its resource usage while maintaining reasonable response times. A similar example of this *software multi-versioning* concept has been used by Google's Gmail, which has a lightweight HTML-based version that is used when the user's browser does not support the feature-rich but resource-heavy JavaScript-based version [14]. By falling back on the lightweight version, the user would still be able to use Gmail, albeit at a reduced quality of service.

Software multi-versioning is traditionally applied to mission-critical systems, such as flight or nuclear power plant control systems, to improve their dependability, reliability, or fault tolerance [9, 10, 21, 29]. As these systems are often monolithic, software multi-versioning requires maintaining several full versions of the system, making it a costly process. As a result, software multi-versioning has never been widely used for non-critical systems, as the cost of maintaining several versions usually does not outweigh the benefits for non-critical systems.

However, the advent of systems with containerized architectures, such as microservice-based ones, opens many new opportunities for applying software multi-versioning. As these systems are divided into smaller components that each run inside their own container, we can apply software multi-versioning to a component rather than the whole system. Figure 1 shows an example of an architecture of a microservice-based application (the `TeaStore` application [41]) in which the `Recommender` microservice uses multi-versioning. For every request, the system can select at runtime whether the Light-Weight or HeavyWeight version of the `Recommender` microservice will be used to fulfill the request.

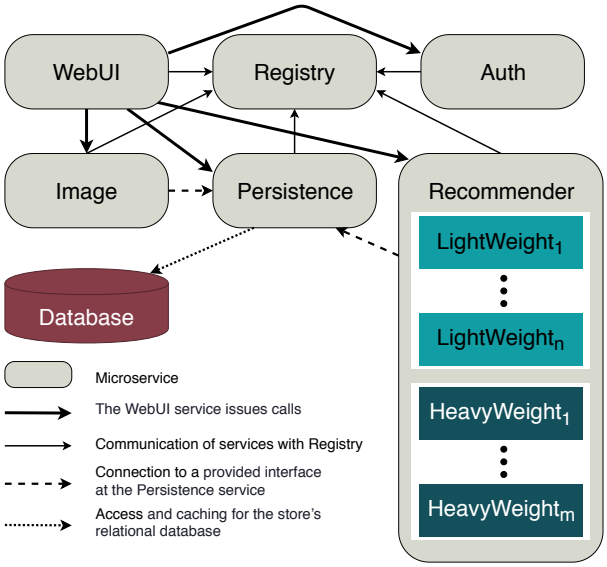


Figure 1: High-level architecture of the `TeaStore` application with multi-versioning of the `Recommender` microservice.

In this paper, we examine how software multi-versioning can help satisfy the performance requirements of containerized software systems. We conduct two experiments on the performance of two containerized systems under varying loads. In the first experiment, we study the `TeaStore` application [41], which is a reference application for benchmarking and testing microservices. We applied multi-versioning to its `Recommender` service to simulate an accurate but resource-heavy recommendation algorithm, and a less accurate but more lightweight version. In our second experiment, we study a containerized three-tier online news application (the `Znn` application [15]) where our adapted version of the `Znn` application reduces the level of service during the high load by using different versions of the content-providing component.

To implement our experiments, we present an extended version of the `Docker` container platform (`DockerMV`) that allows the creation of multi-version services by deploying several containers for each version of the service. To allow service developers to control the load balancing between the multiple versions of their service in a transparent manner, `DockerMV` provides a rule-based load balancer that can be configured at the service-level rather than at the system level. Hence, by using `DockerMV`, developers can extend their own containerized systems with multi-versioned services in a manner that is transparent to the rest of the system.

The rest of the paper is organized as follows. Section 2 provides background information about containerized software systems, microservices, and managing containers. Section 3 presents a motivational example for our approach. In Section 4, we present the concept of our approach. In Section 5, we explain our experimental setup. Section 6 discusses the results of our experiments. Section 7 gives an overview of the related work, and Section 8 explains the threats to the validity of our work. Finally, Section 9 concludes the paper.

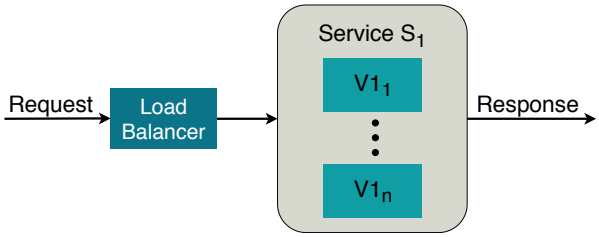


Figure 2: High-level architecture of a regular service in `Docker` where requests are load balanced in a Round Robin manner.

2 BACKGROUND

In this section, we provide background information about containerized software systems and managing containers.

2.1 Containerized Software Systems

One of the essential techniques that enable cloud computing is virtualization [35], which is used to create virtual environments in which processes or services are isolated from each other [11], thereby allowing multi-tenancy of hardware resources [12, 28]. Traditionally, virtualization is achieved using a hypervisor. A hypervisor is a process to create and run virtual machines (VMs) on a host system, making it appear that each VM is using its independent hardware resources. Some well-known examples of hypervisors are VMware ESX, KVM, Xen, and Hyper-V [3, 32, 40]. When using hypervisors for virtualization, each virtual machine runs its operating system (OS) on the host system, which makes the virtual machines resource-heavy and severely limits the number of virtual machines that can run in parallel on a single host.

A recent advancement in virtualization techniques is the advent of lightweight software containers, which share the OS, binaries, and libraries of the host system. As a result, containers are smaller and more lightweight than virtual machines that are started by a hypervisor. Hence, it is possible to run hundreds of containers on a single host machine. Also, as these containers use the host's OS, they can be started much faster [11].

2.1.1 Microservices. Microservices are a popular architectural approach for creating containerized software systems that are inspired by service-oriented computing [20]. In the microservices architecture, the system is developed from a set of small independent services [33]. While microservices can be deployed in virtual machines, the best way to leverage their full potential is to run them inside containers [11, 39]. The independence of microservices allows developers to work on them separately and use the most suitable technology to develop each of them [19]. Also, microservices can be modified independently as the requirements of the system change. Microservices communicate through RESTful APIs or a message-based protocol, which allows scaling an application quickly by replicating the microservice that is under heavy load [30].

2.2 Managing Containers

A popular framework for deploying software containers is the open-source Docker container platform.[1] Docker combines several kernel-level technologies such as LXC and cgroups to enable the deployment and reuse of highly portable, lightweight containers [32]. A Docker container is a runtime instance of a Docker image. A Docker image specifies everything that is necessary to run an application as a container, for example, which libraries should be enabled in the container and how they should be configured [6]. When Docker executes in swarm mode [8], (replications of) containers are started as services that are part of a larger, service-based system (e.g., a microservice-based one) [7]. Figure 2 shows the high-level architecture of a service S_1 that consists of n exact replicas of containers that run version V_1 of the service. The load balancer balances the traffic to the service in a Round Robin-manner between its n containers. One of the main benefits of a Docker service is that the service appears as a single unit to other parts of the system, regardless of the number of replicated containers it consists of. Hence, other parts of the system need not be aware of the load balancing.

One downside of Docker services is that all the containers of a service are exact replicas. In the next section, we present a motivating example in which it would be beneficial to have a service that consists of containers that run different versions of the service.

3 A MOTIVATING EXAMPLE

Erica is a developer who works for an e-commerce start-up company that sells products online. The start-up company has migrated all of its software systems to containerized ones. As the start-up has limited financial resources, it is important to run their systems in a cost-effective manner. Erica is responsible for designing the algorithm that recommends new products to the customers based on the customer's shopping cart, the customer's order history, or the popularity of the items. Erica suggested several algorithms for the recommendation system, each with their strengths and weaknesses. While the algorithm needs to be fast, it should provide high-quality recommendations as well.

Unfortunately, Erica noticed that the performance requirements of one of the software systems could not be satisfied when the recommendation algorithm was enabled. Erica's first solution was to increase the allocated server resources. However, the start-up company cannot afford these extra costs. Instead, Erica decided to implement two versions of the algorithm; one resource-heavy version that provides high-quality recommendations, and one lightweight version that provides lower quality but still acceptable recommendations. Hence, by switching between the algorithms as the availability of resources allows, the system can make the trade-off between resource usage and recommendation quality. For example, when there is a sale event happening on the website, the lightweight algorithm can be used, to ensure the recommendation algorithm does not consume too many resources.

4 OUR APPROACH

Software multi-versioning is the concept of developing and running several different versions of a software system or component to improve one or more of the system's quality attributes. Our approach is to apply software multi-versioning to the containers of a service in a containerized software system. To deploy multi-version services, we need to deploy multiple containers, each of which are instantiated from different container images. Our goal is to implement multi-versioning in a transparent manner, i.e., users of the services and/or containers are not aware of the multi-versioning. Hence, our multi-version containers should form a unified service that can be treated like a regular single-version service.

[1] https://www.docker.com/

Listing 1: The original `docker service create` command. We omitted the arguments that are not relevant to our work for clarity.

```
1 $ docker service create [$OPTIONS] $IMAGE [$REPLICATIONS]
```

4.1 Implementation

To implement our approach, we extended the Docker framework into the DockerMV framework. To create a service with the original Docker framework, the `docker service create` command in Listing 1 is used. The original command takes the following parameters:

- `$OPTIONS`: Optional parameters that can be used to configure container-specific parameters, such as the environment variables and the memory limit.
- `$IMAGE`: The image from which the container should be created.
- `$REPLICATIONS`: The number of replications of the container that should be created.

The command in Listing 1 will create a service that consists of `$REPLICATIONS` exact copies of the container that is created from the `$IMAGE` image with the configuration options specified in `$OPTIONS`. The `docker service create` command in Listing 1 does not support multi-versioning. Therefore, we extended the command's implementation to accept multiple images with different replication and configuration parameter values.

Listing 2 shows the extended command, which allows the creation of multi-version Docker services. In particular, the extended command allows the creation of a Docker service that consists of $\$REPLICATIONS_1$ + ... + $\$REPLICATIONS_n$ containers, that were created from n images. In addition, the extended `docker service create` command supports the following parameters:

- `Network`: The name of the overlay network to connect the containers to each other (fixed for all containers in the Docker service) (Required).
- `Name`: The Docker service name (fixed for all containers in the Docker service) (Required).
- `Environment variables`: The environment variables (fixed for all containers in the Docker service) (Optional).
- `Memory`: The memory limit for a container (Optional).
- `Swap memory`: The swap memory limit for a container (Optional).
- `CPU`: The number of CPUs for a container (Optional).

Listing 2: The extended `docker service create` command

```
$ docker service create [$OPTIONS]
    $IMAGE_1 $REPLICATIONS_1
    ...
    $IMAGE_n $REPLICATIONS_n
```

Listing 3: An example invocation of the extended `docker service create` command that deploys two versions of the `teastore-recommender` service.

```
$ docker service create
    e REGISTRY_HOST=host_IP e REGISTRY_PORT=10000
    e HOST_NAME=host_IP e SERVICE_PORT=3333
    10.2.5.26 Network recommender 8080 rules.txt
    sgholami/teastore-recommender:HeavyWeight 1
    sgholami/teastore-recommender:LightWeight 1
```

- `Container port`: The port that the containers of the `Docker` service will listen on (fixed for all containers in the `Docker` service) (Required).
- `Rule-set`: The location of the user-defined rule-set.

In our extended command, the `network, name, environment variables`, and `container port` parameters have the same value across all containers of the service. However, the `memory, swap memory`, and `CPU` can be configured differently for each container in the service. Listing 3 shows an example invocation of the extended command (which is part of our `DockerMV` extension). In particular, two versions of the `teastore-recommender` service are started (one replication of each), that are connected to the `recommender` network. Each of the containers initializes four environment variables (`REGISTRY_HOST`, `REGISTRY_PORT`, `HOST_NAME` and `SERVICE_PORT`).

4.2 Load Balancing

Figure 2 shows an example architecture of a `Docker` service. As shown in Figure 2, a `Docker` service has a load balancer that distributes the incoming requests between the service's containers. As these containers are created from the same image, the load balancer usually distributes the incoming traffic in a round-robin manner (i.e., an equal amount of traffic to each container) [8]. Figure 3 shows the architecture of a service that consists of containers made from different images. As these containers are created from different images, they may perform a similar task at the different quality of service levels, e.g., comparable to our motivating example in Section 3. Hence, it may no longer be desirable to distribute the traffic in a round-robin manner. Instead, we would like to balance the load based on performance metrics of the service, such as median response times or CPU utilization. Therefore, we implemented a rule-based load balancer in our services. We used a customized version of NGINX[2] as the load balancer amongst the different replications of a service's containers. Our customized load balancer has a user-defined rule-set which defines how to balance the incoming traffic to satisfy a system's performance requirements. Listing 4 shows the format of the rules for the load balancer.

Listing 4: Format of the rules for the load balancer

```
$METRIC $OPERATOR $THRESHOLD ,
    (version $VERSION_NAME perc=$PERCENTAGE;)+
```

Listing 5: Example rule for the load balancer

```
RT > 0.4 ,
    version recommender:HeavyWeight perc=40;
    version recommender:LightWeight perc=60;
```

The parameters in the rule in Listing 4 are as follows:

- `$METRIC`: The metric that is used to check whether a rule should fire. Currently only `RT` (median response time) is supported.
- `$OPERATOR`: The relational operator (<, <=, >, >= or ==) that is used in the condition to check whether a rule should fire.
- `$THRESHOLD`: The threshold for the metric that is used in the condition to check whether a rule should fire.
- `$VERSION_NAME`: The name of one of the versions of the service.
- `$PERCENTAGE`: The percentage of requests to be directed to the container (between 1 and 100).[3]

Listing 5 shows an example rule, in which 40% of the requests are directed to the first container (i.e., the HeavyWeight version of the service), and the second (LightWeight) container handles the other 60% of the requests. We recalculate the median response time every five seconds from NGINX's log file. NGINX uses this median value to decide which rule should be used when balancing the incoming traffic. NGINX saves the `$time_local`, and `$request_time` for each of the incoming requests. The `$time_local` returns the local time of the machine, and we use that time to identify the requests which were received in the last n seconds. The `$request_time` is the elapsed time since the first bytes were read from the client.

4.3 On the Necessity of Our Approach

One could argue that software multi-versioning could easily be achieved using `if`-statements inside a service's source code, or by simply starting multiple services (i.e., one for each version). However, source code-based solutions have the disadvantage that they clutter the source code, making maintenance and understanding of the code more challenging. In addition, starting multiple services causes software multi-versioning to no longer be transparent, which has obvious (negative) consequences for the other parts of the system. For example, the system now needs to be aware of more complex load balancing requirements. Hence, our approach is necessary to provide multi-versioning in containerized systems in a transparent, non-cluttered manner.

[2] https://www.nginx.com

[3] NGINX does not accept 0 as the percentage of requests.

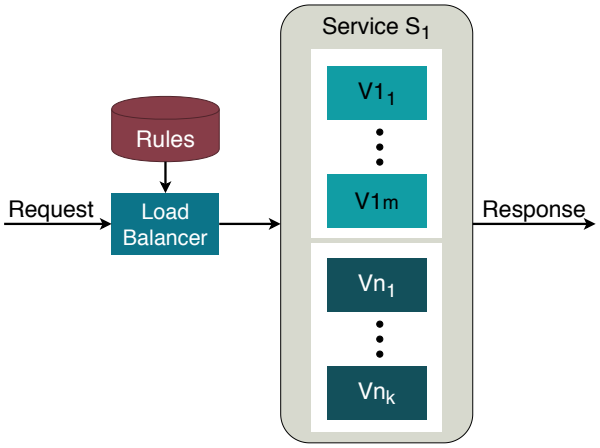


Figure 3: High-level architecture of a service with multi-versioning where requests are balanced based on a rule-set.

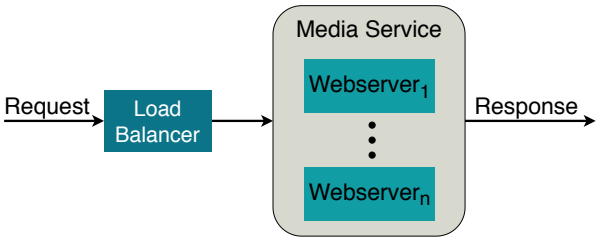


Figure 4: High-level architecture of the Znn application

5 EXPERIMENTAL SETUP

In this section, we elaborate on our experimental setup. The goal of our experiments is to study the benefits of software multi-versioning for satisfying the performance requirements of containerized software systems.

5.1 Subject Systems

In our experiments, we study the `TeaStore` [41] and the `Znn` applications. The `TeaStore` application is a reference microservice application that can be used for performance testing and benchmarking. The `TeaStore` application simulates an online store that is composed of six microservices (see Figure 1). Every microservice runs inside its container. In addition, the database runs inside its container.

The `Znn` application [15] is a three-tier web-based news portal that can be used for testing and benchmarking of self-adaptive applications. The `Znn` application contains a pool of web servers, a MySQL database with news-related text and multimedia contents, and a load balancer that receives requests from clients and distributes them among the web servers in a Round Robin manner. The high-level architecture of the `Znn` application is shown in Figure 4. The source code of the `TeaStore`[4] and the `Znn`[5] applications are both publicly available.

[4]https://github.com/DescartesResearch/TeaStore
[5]https://github.com/cmu-able/znn

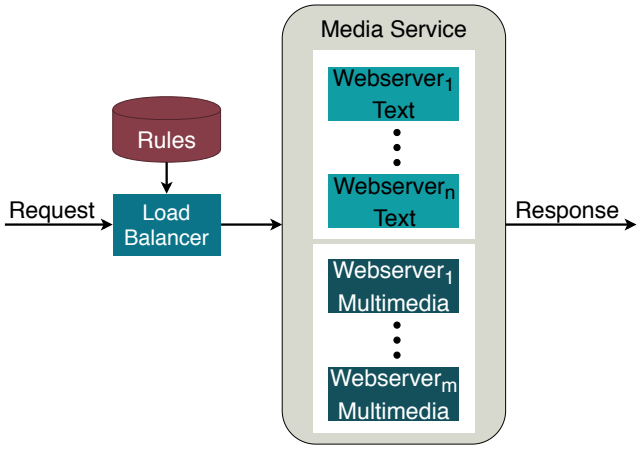


Figure 5: Containerized deployment of `Znn` in which we have two different versions of the `Media` service.

5.2 Introducing Multi-Versioning in the Subject Systems

To introduce multi-versioning in the `TeaStore` application, we adapted the `Recommender` service, which is designed to return recommendations based on the user's history and items in their shopping cart (similar to our motivating example in Section 3). The `TeaStore` application provides several algorithms and trains them once the service is first launched. To conduct our experiment, we selected one of the algorithms, which is the SlopeOne algorithm, and forced retraining every two minutes. The multiple retraining is applied to simulate a higher load and pressure on the system in which the `Recommender` service is replicated (and hence retrained) on several containers. The retraining causes slower response times of the `Recommender`. As a result, we use two versions of the `Recommender` in our experiment, one with regular retraining (`HeavyWeight`) and another with a single training (`LightWeight`). Figure 1 shows the architecture of the `TeaStore` application with the multi-versioned `Recommender` service.

The `Znn` application returns news articles that contain multimedia contents (such as a video that is sent by the web server). Therefore, when the load of the system increases, the system's median response time rises (as the network bandwidth becomes a bottleneck). Hence, we created two different versions of the web servers. The first version provides the original news article along with its multimedia contents, while the other version of the service returns only the text contents of the news. Figure 5 shows the high-level architecture of the containerized version of the `Znn` application with multi-versioning.

5.3 Experiments

We conducted three experiments for each of the subject systems:

- **Ideal Case Experiment**: In this experiment, we tested the "ideal case" for each of the systems, i.e., the case in which all requests are served by the heavyweight versions of the services. Hence, for the `TeaStore` application, all requests are served by the `Recommender` that is constantly retrained,

and for the Znn application, all requests receive a multimedia response.

- **Worst Case Experiment**: In the second experiment, we tested the worst-case setup (in terms of quality of service, i.e., we only used the lightweight versions of the services) for each of the subject systems. For the TeaStore application, the worst case is to use only the Recommender service with a single training. In the Znn application, the worst case is to return only the text responses.
- **Adaptive Experiment**: Finally, we studied how multi-versioning, together with an adaptive balancing of the workload, can help to satisfy the performance requirements. For each of the subject systems, we deployed both of the versions of the services and balanced the load based on a customized rule-set. In this setup, we used our extended version of Docker (DockerMV) along with our customized NGINX load balancer.

These experiments are summarized in Table 1. To demonstrate our approach, we defined the performance requirements as follows:

- For the TeaStore application, we set 450 milliseconds as the upper limit for the median response time.
- For the Znn application, we set 1 second as the upper limit for the median response time.

Both of these performance requirements were defined empirically based on the ideal and worst case experiments. Please note that the exact choice of performance requirements does not matter much—our sole purpose in this paper is to demonstrate the efficacy of software multi-versioning to satisfy performance requirements.

5.4 Workload

We used Apache JMeter,[6] a tool for load testing web applications to generate workloads for our experiments.

For the TeaStore application, we generated the workload using the JMeter script that is provided by the TeaStore developers and modified it to add more items to the shopping cart to put more pressure on the Recommender service.[7] We generated a workload of 100 users who concurrently send HTTP requests to the TeaStore application for different purposes such as opening the home page, logging in, or adding items to the cart. This workload continued for 1,000 seconds. Each user sends an HTTP request to the server and receives an HTML page, and as soon as the user receives a response, they send the next request.

For the Znn application, we generated a workload that sends HTTP requests and simulates multiple users sending requests to the Znn application concurrently. Figure 6 shows the shape of the workload and the number of active users during each of the two-hour experiments. For instance, at the highest peak where the number of active users is 200, it means that 200 threads concurrently send requests to the servers, and when they get a response, they send another request. We use the same workload across the experiments for each of the subject systems.

[6] https://jmeter.apache.org

[7] The workload's JMeter script is available on the project's GitHub repository [23].

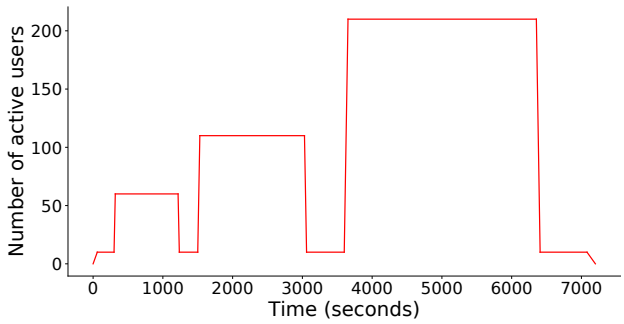


Figure 6: Shape of the Znn application workload

Listing 6: NGINX rule set for the TeaStore application

```
RT < 0.1 ,
    version recommender:HeavyWeight perc=99;
    version recommender:LightWeight perc=1;
RT < 0.25 ,
    version recommender:HeavyWeight perc=90;
    version recommender:LightWeight perc=10;
RT < 0.4 ,
    version recommender:HeavyWeight perc=80;
    version recommender:LightWeight perc=20;
RT >= 0.4 ,
    version recommender:HeavyWeight perc=70;
    version recommender:LightWeight perc=30;
```

5.5 Deployment of the Subject Systems

Table 2 shows the description of the containers that we used for the experiments. We limited the containers' memory, swap memory, and CPU to stop them from growing too much and allocating all of the available resources. These limits were defined based on our experience with the subject systems.

We provisioned one virtual machine in the Compute Canada cloud[8] and one virtual machine in the Cybera Rapid Access Cloud[9] to run our containers for both of the experiments. In particular, we ran the JMeter script on the Compute Canada cloud and the subject systems on the Cybera cloud. Table 3 summarizes the configurations of our virtual machines.

The source code of the DockerMV and more details about our experiments can be found on the project's GitHub repository [23].

5.6 Load Balancing

For the TeaStore application, we set the rules presented in Listing 6 for NGINX to balance the load between the versions of the Recommender service. Listing 7 shows the set of rules for the Znn application. Both rule sets were defined empirically based on observations during preliminary runs of the experiments.

[8] https://www.computecanada.ca/research-portal/national-services/compute-canada-cloud

[9] https://www.cybera.ca/services/rapid-access-cloud/

Table 1: A description of the experiments that we conducted for the TeaStore and Znn applications

	TeaStore Description	Znn Description
Ideal case experiment	Recommender with multiple training	Multimedia responses only
Worst case experiment	Recommender with single training	Text responses only
Adaptive experiment	Adaptive load distribution	Adaptive load distribution

Table 2: Description of the containers in the experiments

Name	Docker Image	Memory	Swap Memory	CPU
HeavyWeightRecommender	sgholami/teastore-recommender:HeavyWeight	1G	1G	0.4
LightWeightRecommender	sgholami/teastore-recommender:LightWeight	1G	1G	0.4
Multimedia	alirezagoli/znn-multimedia:v1	1G	1G	0.4
Text	alirezagoli/znn-text:v1	1G	1G	0.4
NGINX	sgholami/nginx-monitoring	unlimited	unlimited	unlimited
NGINX_official	NGINX	unlimited	unlimited	unlimited
MySQL	alirezagoli/znn-mysql:v1	unlimited	unlimited	unlimited

Table 3: Description of the virtual machines

Cloud	Instance	VCPUs	Memory	OS
Cybera	Experiment	4	8GB	Ubuntu-18.04
Compute Canada	JMeter	4	15GB	Ubuntu-18.04

Listing 7: NGINX rule set for the Znn application

```
RT < 0.1 ,
    version znn-multimedia:v1 perc=99;
    version znn-text:v1 perc=1;
RT < 0.2 ,
    version znn-multimedia:v1 perc=80;
    version znn-text:v1 perc=20;
RT < 0.3 ,
    version znn-multimedia:v1 perc=70;
    version znn-text:v1 perc=30;
RT < 0.6 ,
    version znn-multimedia:v1 perc=40;
    version znn-text:v1 perc=60;
RT < 0.8 ,
    version znn-multimedia:v1 perc=30;
    version znn-text:v1 perc=70;
RT >= 0.8 ,
    version znn-multimedia:v1 perc=20;
    version znn-text:v1 perc=80;
```

6 EXPERIMENTAL EVALUATION

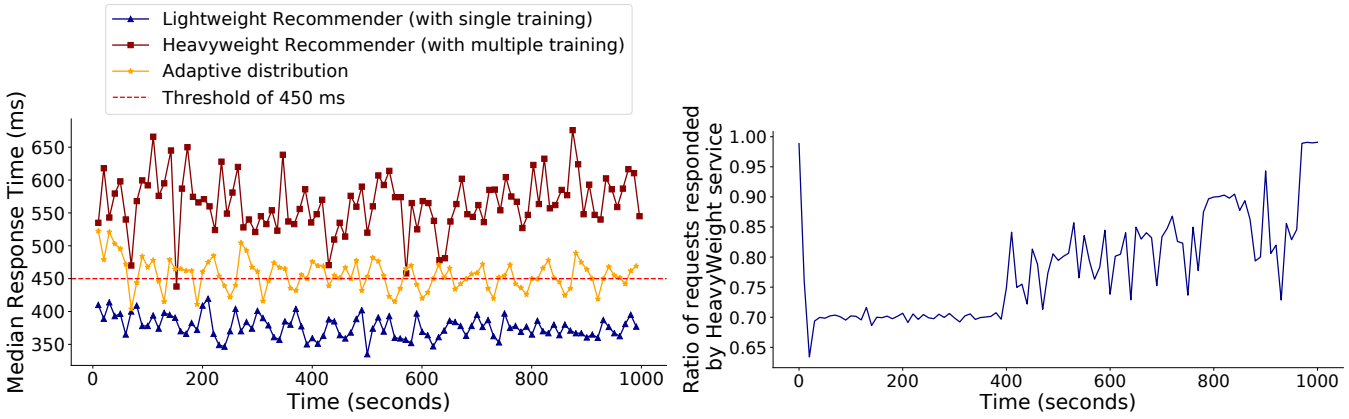
In this section, we discuss the results of our experiments for each subject system.

6.1 Experiments with the TeaStore Application

Figure 7a shows the median response times of the TeaStore application in our experiments. We illustrated the result of all tests in one plot as the range of their values is close, and it is possible to observe the changes in all of them together. Figure 7b shows the ratio of requests that were responded to by the HeavyWeight version of the Recommender service. We observe that the median response time fluctuates around our performance requirement threshold as the load balancer distributes the load between the HeavyWeight and LightWeight versions of the service.

6.2 Experiments with the Znn Application

Figure 8a shows the median response time of the ideal case for the Znn application when we are using only the multimedia version of the service. During this experiment, the median response time of the application goes up to around 25 seconds, which indicates that the resources for the application are severely under-provisioned. Figure 8b shows the median response time of the worst-case experiment, which shows that the available resources can easily handle this type of traffic. However, the quality of service is considerably reduced since all requests are handled by the text version of the service. Figure 9a shows the median response time when using software multi-versioning to balance between the multimedia and text version of the service. In addition, Figure 9b shows the ratio of the requests which were responded to by the multimedia server in the adaptive experiment. Figures 9a and 9b show that the system deals with the increases in workload by balancing the majority of the requests (first approximately 50-70% and then approximately 80%) to the text version of the service.



(a) Median response time of the TeaStore application

(b) The ratio of requests responded by the HeavyWeight version of the service

Figure 7: The TeaStore application experiments and the distribution ratio of traffic using software multi-versioning and adaptive load balancing

Summary: *Multi-versioning allows us to satisfy the performance requirements of subject systems while maintaining a level of Quality of Service that is as high as possible using the given resources.*

7 RELATED WORK

In this section, we discuss prior work that is related to ours. In particular, we discuss related work on software multi-versioning, software multi-versioning for containerized systems, and performance engineering of containerized systems.

7.1 Software Multi-Versioning

Until now, software multi-versioning has been used for several purposes, such as improving a system's security [13, 22, 31, 36], safety [25], reliability [16], and availability [24]. In these cases, software multi-versioning is often used as a means to achieve software redundancy, i.e., to have several different versions of the software that are functionally equivalent, yet different in terms of, e.g., implementation or used implementation language.

Larsen et al. [31] studied the effect of automated software redundancy on a system's security. Franz et al. [22] used software multi-versioning as a defense mechanism for a system. The idea of their approach is that as the system has several versions, it is harder to figure out for the attackers which version they are attacking. Therefore, they have less chance to succeed in their attack, which increases the system's security. Persaud et al. [36] used software redundancy by using Genetic Algorithms to enhance the security of the system. Cigsar et al. [16] considered software multi-versioning as an approach to improve the reliability of repairable systems. Gracie et al. [25] stated that there had been designs for using redundancy for safety purposes. Gorbenko et al. [24] used software multi-versioning for a web service to extend its functionality and improve its attributes such as availability and reliability.

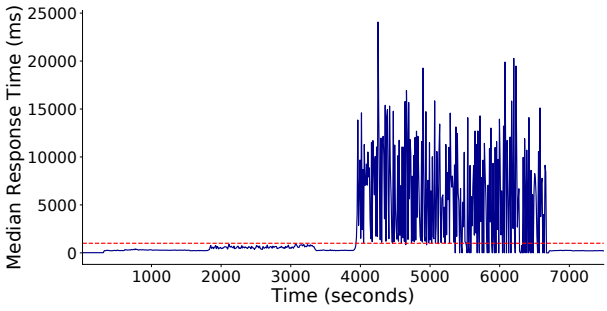
7.2 Software Multi-Versioning for Containerized Systems

In containerized systems, software multi-versioning has been used mostly for purposes such as enhancing the fault tolerance. [42, 45], security, and reliability [44] of the systems. For example, Wang et al. [42] suggested the idea of applying multi-versioning to critical components of cloud-based software to improve the fault tolerance of the system. Wang et al. proposed an approach to find the critical components of the system and apply software multi-versioning only to those critical components to reduce the cost and complexity of software multi-versioning while improving the system's fault tolerance. Also, Zheng et al. showed that software multi-versioning could be used to improve the reliability [44] and fault tolerance [45] of service-oriented systems. However, the choice of using multi-versioning can affect the quality of service of the system. Therefore, Zheng et al. formulated the reliability requirements as an optimization problem and proposed a heuristic algorithm to maintain the quality of the system by solving this optimization problem.

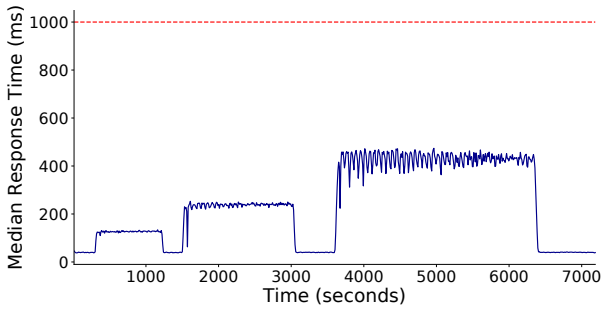
All prior studies on applying software multi-versioning to containerized software systems focused on improving the reliability of a system. We are the first to study the benefits of software multi-versioning for satisfying the performance requirements of a system.

7.3 Performance Engineering of Containerized Systems

Recently, performance engineering researchers have started to study performance engineering for microservices. For example, Heinrich et al. gave an overview of the challenges of performance engineering microservices [26]. They identified performance testing, monitoring, and modeling of microservices as the primary performance engineering challenges. Performance testing microservices is challenging, as the services are developed and maintained independently. Therefore, Camargo et al. [18] presented an approach to automate the performance testing for microservices. In

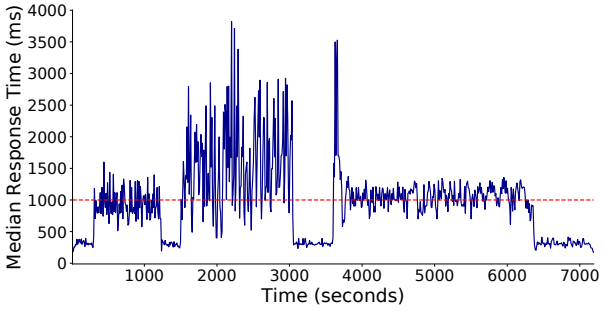


(a) The median response time when running only the multimedia-version of the service

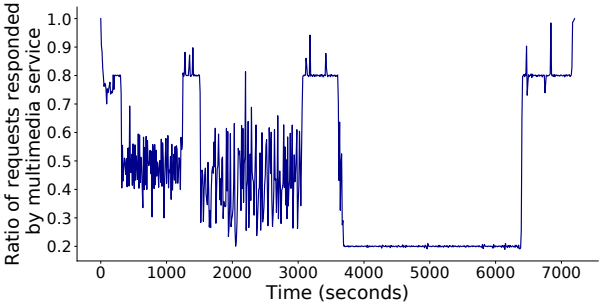


(b) The median response time when running only the text version of the service

Figure 8: The Znn application experiments using only the multimedia vs only the text version of the service. Note that the scales on the y-axes are different.



(a) The median response time when running the services with multi-versioning and using the rules defined in Listing 7



(b) The ratio of requests responded by the multimedia version of the service

Figure 9: The Znn application experiment using software multi-versioning and adaptive load balancing

this approach, each microservice provides a test specification that was used for performing the tests. Jindal et al. [27] addressed the performance modeling of microservices by capacity planning. They identified a microservice's capacity to find the appropriate resource needed for the microservices. As a result, the system would not violate the performance requirements. Amaral et al. [5] studied two models for microservices architecture using containers. They compared the performance of CPU and network for the master-slave and nested-container models to provide a guide for system designers.

A large body of the existing performance engineering work for containerized systems is about the performance of cloud systems. As performance engineering for cloud systems is a very broad topic, a thorough discussion of this body of work is outside the scope of this paper, and we refer the reader to one of the excellent surveys on this topic, e.g., the ones by Xu et al. [43] or Nuaimi et al. [4]. Also, Ruan et al. [38] studied the performance of cloud systems by using containers from different perspectives.

8 THREATS TO VALIDITY

In this paper, we studied how software multi-versioning can help to satisfy the performance requirements of containerized software systems. In this section, we discuss the threats to the validity of this study.

8.1 External Validity

The Choice of Subject Systems. We studied two open source applications, one of which is a microservices application (the `TeaStore` application), and the other is a more traditional three-tier application (the Znn application). The Znn application is not originally a containerized application, although based on our experience, many three-tier applications are containerized in a similar manner as we did in this study. While we aimed to select systems that are representative of larger groups of systems, future studies should investigate how well software multi-versioning works for a wider range of systems, such as industrial systems.

8.2 Internal Validity

The Choice of Performance Requirements. The performance requirements that we used in our experiments were defined empirically based on the ideal and worst case experiments. As our purpose in this paper is to demonstrate the efficacy of software multi-versioning to satisfy performance requirements, the exact values of the requirements do not matter much. There could exist characteristics that make some requirements more difficult to satisfy than others. For example, if a lightweight version of a service already has difficulties to satisfy a performance requirement given the available resources, software multi-versioning will not help much (since the load balancer will simply divert all traffic to the lightweight version). Hence, future studies should further investigate how the choice of performance requirements impacts the efficacy of software multi-versioning to satisfy those requirements.

The Choice of Load Balancing Rules. As the focus of our work is to demonstrate the efficacy of software multi-versioning for satisfying performance requirements and not to present a novel load balancing technique, in our experiments, the load balancing is done by a simple static approach. Users of `DockerMV` can easily adapt the load balancing rules to implement more advanced load balancing techniques for their own systems, such as those proposed by Niu et al. [34], Radojevic et al. [37] or Dasguptaat al. [17]. While our experiments show that the used simple rule sets can already yield satisfactory results, future studies should investigate how to optimize the rules on a per-system and per-workload basis.

8.3 Construct Validity

The Choice of Performance Metric. We chose median response time as our performance metric as it the primary metric that is used for measuring the user-perceived performance. Future studies should consider how software multi-versioning can benefit other performance metrics, such as CPU utilization or memory usage.

The Overhead of Software Multi-Versioning. We did not measure the overhead that is added by introducing software multi-versioning to containerized systems. However, given that the additional load balancing is fairly simple and straightforward, there should not be a significant amount of additional overhead introduced.

9 CONCLUSION

Traditionally, software multi-versioning has been applied only to mission-critical systems due to the high cost of maintaining multiple versions of the software. Recently the increase in popularity of containerized software systems has opened many new opportunities for the application of software multi-versioning, as the technique can be applied to smaller parts of these systems.

In this paper, we study how software multi-versioning can help to satisfy the performance requirements of containerized software systems. In summary, our paper makes the following contributions:

- **A demonstration that software multi-versioning can effectively be applied to satisfy the performance requirements of containerized software systems.** We show through experiments on two open-source applications that software multi-versioning can effectively be applied to containerized systems to satisfy performance requirements while maintaining a quality of service-level that is still acceptable given the available resources.
- **A framework to deploy services with software multi-versioning.** We extended the `Docker` container platform to allow the creation of multi-version services. Our `DockerMV` platform supports custom rule-based load balancing between the versions of a service that can be controlled by the service developer and hence is transparent to the other parts of the system. Our `DockerMV` implementation is publicly available [23].

We are one of the first ones to study the benefits of software multi-versioning for containerized systems. In particular, we are the first to demonstrate how software multi-versioning can help to satisfy the performance requirements of such systems. Our expectation is that our `DockerMV` platform can help to satisfy other nonfunctional requirements of a containerized software system, such as dependability, reliability, availability, and security requirements. Hence, future studies can leverage our platform to investigate how software multi-versioning can be applied to further help satisfy the nonfunctional requirements of containerized software systems.

As future work to our study, we plan to investigate the overhead of developing and maintaining multiple `Docker` containers. Also, future studies are needed on the selection of parameters for the load balancer rules.

REFERENCES

[1] Stephen Adler. 1999. Addendum to the Slashdot Effect Internet Paper. (1999).
[2] Stephen Adler. 1999. The Slashdot effect: an analysis of three Internet publications. *Linux Gazette* 38, 2 (1999).
[3] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. 2017. Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on Services Computing* 11, 2 (2017), 430–447.
[4] Klaithem Al Nuaimi, Nader Mohamed, Mariam Al Nuaimi, and Jameela Al-Jaroodi. 2012. A survey of load balancing in cloud computing: Challenges and algorithms. In *Proceedings of the 2nd Symposium on Network Cloud Computing and Applications*. IEEE, 137–142.
[5] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. 2015. Performance evaluation of microservices architectures using containers. In *Proceedings of the 14th IEEE International Symposium on Network Computing and Applications*. IEEE, 27–34.
[6] Docker Documentation Author. [n. d.]. Docker Container. https://docs.docker.com/glossary/?term=container. ([n. d.]). Accessed: 2019-10-09.
[7] Docker Documentation Author. [n. d.]. How services work. https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/. ([n. d.]). Accessed: 2019-10-10.
[8] Docker Documentation Author. [n. d.]. Swarm mode key concepts. https://docs.docker.com/engine/swarm/key-concepts/#load-balancing. ([n. d.]). Accessed: 2019-01-10.
[9] Algirdas Avizienis and John PJ Kelly. 1984. Fault tolerance by design diversity: Concepts and experiments. *Computer* 8 (1984), 67–80.
[10] Algirdas Avizienis and J-C Laprie. 1986. Dependable computing: From concepts to design diversity. *Proc. IEEE* 74, 5 (1986), 629–638.
[11] David Bernstein. 2014. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing* 1, 3 (2014), 81–84.
[12] Cor-Paul Bezemer and Andy Zaidman. 2010. Multi-tenant SaaS applications: maintenance dream or nightmare?. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*. ACM, 88–92.
[13] Hayley Borck, Mark Boddy, Ian J De Silva, Steven Harp, Ken Hoyme, Steven Johnston, August Schwerdfeger, and Mary Southern. 2016. Frankencode: Creating diverse programs using code clones. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 604–608.
[14] Gmail Help Center. [n. d.]. Gmail Help. https://support.google.com/mail/answer/15049?hl=en. ([n. d.]). Accessed: 2019-09-27.
[15] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. 2009. Evaluating the effectiveness of the rainbow self-adaptive system. In *Proceedings of the 9th Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 132–141.

[16] Candemir Cigsar and Yongho Lim. 2017. Modeling and analysis of cluster of failures in redundant systems. In *Proceedings of the 2nd International Conference on System Reliability and Safety (ICSRS)*. IEEE, 119–124.

[17] Kousik Dasgupta, Brototi Mandal, Paramartha Dutta, Jyotsna Kumar Mandal, and Santanu Dam. 2013. A genetic algorithm (ga) based load balancing strategy for cloud computing. *Procedia Technology* 10 (2013), 340–347.

[18] Andre de Camargo, Ivan Salvadori, Ronaldo dos Santos Mello, and Frank Siqueira. 2016. An architecture to automate performance tests on microservices. In *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services*. ACM, 422–429.

[19] Namiot Dmitry and Sneps-Sneppe Manfred. 2014. On micro-services architecture. *International Journal of Open Information Technologies* 2, 9 (2014).

[20] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. In *Present and ulterior software engineering*. Springer, 195–216.

[21] Dave E Eckhardt, Alper K. Caglayan, John C. Knight, Larry D. Lee, David F. McAllister, Mladen A. Vouk, and John P. J. Kelly. 1991. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transactions on Software Engineering* 17, 7 (1991), 692–702.

[22] Michael Franz. 2010. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the New Security Paradigms Workshop*. ACM, 7–16.

[23] Sara Gholami, Alireza Goli, Cor-Paul Bezemer, and Hamzeh Khazaei. [n. d.]. DockerMV. https://github.com/pacslab/DockerMV. ([n. d.]). Accessed: 2019-10-19.

[24] Anatoliy Gorbenko, Vyacheslav Kharchenko, and Alexander Romanovsky. 2009. Using inherent service redundancy and diversity to ensure web services dependability. In *Methods, Models and Tools for Fault Tolerance*. Springer, 324–341.

[25] Emil Gracie, Ali Hayek, and Josef Borcsok. 2017. Evaluation of FPGA design tools for safety systems with on-chip redundancy referring to the standard IEC 61508. In *Proceedings of the 2nd International Conference on System Reliability and Safety (ICSRS)*. IEEE, 386–390.

[26] Robert Heinrich, Andre van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. 2017. Performance engineering for microservices: research challenges and directions. (2017).

[27] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. 2019. Performance modeling for cloud microservice applications. In *Proceedings of the 10th ACM/SPEC on International Conference on Performance Engineering*. ACM, 25–32.

[28] Jaap Kabbedijk, Cor-Paul Bezemer, Slinger Jansen, and Andy Zaidman. 2015. Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective. *Journal of Systems and Software* 100 (2015), 139–148.

[29] John P. J. Kelly, Thomas I. McVittie, and Wayne I. Yamamoto. 1991. Implementing design diversity to achieve fault tolerance. *IEEE Software* 8, 4 (1991), 61–71.

[30] Hamzeh Khazaei, Rajsimman Ravichandiran, Byungchul Park, Hadi Bannazadeh, Ali Tizghadam, and Alberto Leon-Garcia. 2017. Elascale: autoscaling and monitoring as a service. In *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., 234–240.

[31] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated software diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*. IEEE, 276–291.

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

[33] Dmitry Namiot and Manfred Sneps-Sneppe. 2014. On micro-services architecture. *International Journal of Open Information Technologies* 2, 9 (2014), 24–27.

[34] Yipei Niu, Fangming Liu, and Zongpeng Li. 2018. Load balancing across microservices. In *Proceedings of the IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 198–206.

[35] Claus Pahl. 2015. Containerization and the paas cloud. *IEEE Cloud Computing* 2, 3 (2015), 24–31.

[36] Bheesham Persaud, Borke Obada-Obieh, Nilofar Mansourzadeh, Ashley Moni, and Anil Somayaji. 2016. Frankenssl: Recombining cryptographic libraries for software diversity. In *Proceedings of the 11th Annual Symposium On Information Assurance. NYS Cyber Security Conference*. 19–25.

[37] Branko Radojevic and Mario Žagar. 2011. Analysis of issues with load balancing algorithms in hosted (cloud) environments. In *2011 Proceedings of the 34th International Convention MIPRO*. IEEE, 416–420.

[38] Bowen Ruan, Hang Huang, Song Wu, and Hai Jin. 2016. A performance study of containers in cloud environment. In *Asia-Pacific Services Computing Conference*. Springer, 343–356.

[39] James Turnbull. 2014. *The Docker Book: Containerization is the new virtualization*. James Turnbull.

[40] VMWare. [n. d.]. Hypervisor. https://www.vmware.com/topics/glossary/content/hypervisor. ([n. d.]). Accessed: 2019-10-09.

[41] Joakim von Kistowski, Simon Eismann, Norbert Schmitt, Andre Bauer, Johannes Grohmann, and Samuel Kounev. 2018. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '18)*. IEEE.

[42] Lei Wang and Kishor S Trivedi. 2019. Architecture-based Reliability-sensitive Criticality Measure for Fault-Tolerance Cloud Applications. *IEEE Transactions on Parallel and Distributed Systems* (2019).

[43] Fei Xu, Fangming Liu, Hai Jin, and Athanasios V Vasilakos. 2013. Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions. *Proc. IEEE* 102, 1 (2013), 11–31.

[44] Zibin Zheng and Michael R Lyu. 2013. Selecting an optimal fault tolerance strategy for reliable service-oriented systems with local and global constraints. *IEEE Trans. Comput.* 64, 1 (2013), 219–232.

[45] ZiBin Zheng, Michael Rung Tsong Lyu, and HuaiMin Wang. 2015. Service fault tolerance for highly reliable service-oriented systems: an overview. *Science China Information Sciences* 58, 5 (2015), 1–12.