Performance effects on servers using containers in comparison to hypervisor based virtualization

University of Oulu
Degree Programme in Information Processing Science
Bachelor’s thesis
Perttu Kärnä
21.05.2018
Abstract

The current direction of web-based software is evolving has brought many challenges regarding locations, scalability, maintainability and performance. In most of the cases tiny differences in performance are not really important since the user base of the software may not be remarkably high. However, when the user base of the system expand and the stress of the system has high peaks, smaller things start to have meaning in the software and infrastructure. This paper addresses the performance differences between today’s usual web software deployment solutions, containers and VMs.

This study is based on literature review, which has been conducted by studying previous studies about the topic. The main focus in the study is Linux Container based containerization solutions such as Docker, and traditional hypervisor based virtual machines such as KVMs.

The main categories of performance this paper is addressing to are memory, CPU, network, disk I/O, database applications, DevOps and overall system performance. The categorizing is based on the studies reviewed in this paper which obviously results in slight overlapping on some categories. In the end of this paper the results are summed up and some implications for them are presented.

Keywords
docker, container, virtual machine, performance
## Contents

Abstract .................................................................................................................. 2  
Contents .................................................................................................................. 3  
1. Introduction .................................................................................................... 4  
2. Terminology .................................................................................................... 6  
   2.1 DevOps ....................................................................................................... 6  
   2.2 Docker container ....................................................................................... 6  
   2.3 Kernel-based virtual machine ..................................................................... 9  
3. Research methods ............................................................................................ 11  
   3.1 Research method ....................................................................................... 11  
   3.2 Literature definition ................................................................................... 12  
4. Literature review ............................................................................................... 13  
   4.1 Previous research ..................................................................................... 13  
      4.1.1 Docker in research ............................................................................ 13  
      4.1.2 Containers and virtual machines ...................................................... 13  
   4.2 Performance review .................................................................................... 14  
5. Discussion and implications ............................................................................ 18  
6. Conclusions ..................................................................................................... 20  

References ........................................................................................................... 22  
Appendix A. .......................................................................................................... 25
1. Introduction

While container-based technology might not be new, container technologies have gained a huge popularity amongst web developers and web service providers in the past few years. Probably the most commonly known container platform, Docker was introduced in 2013 and since then has affected remarkably to the modern web development and microservices architecture. One of the main problems that Docker, and some other application isolation tools such as Vagrant, are solving in web development is the consistency of environments between developers and different deployment environments. Plenty of time can be spent on installing development dependencies and tools even before the real development can be started, not to mention different deployment environment configurations. In most cases big part of this fuzz can be avoided with Docker.

There is a huge amount of options available these days for deployment strategies and production environments. While web server providers offer pure metal hardware in addition to virtual private servers, web developers get interested in performance differences between virtualization levels. The performance in this context means not only calculating performance in computers, but also scalability and infrastructure provisioning when needed on cloud services. The abilities such as scaling services fast, updating nodes without downtime, recovering from failures efficiently and reverting back to previous version when needed are important things in web software development process.

In addition to web community, Docker has reached also the research community. Docker containers can be used to provide reproducible research environment in computer science (Chamberlain & Schommer, 2014). For example, packaging and shipping research environments in containers within an international research group is easy and fast. Versioning images and using Dockerfiles makes the management of containers clear for everyone. The availability of public and private image repositories makes the research even more transparent and reproducible for the whole science community.

Docker containers provide also an easy way to manage and distribute services (microservices), which makes web software easier to scale and manage (Kratzke, 2015; Amaral et al., 2015). Docker can be harnessed to boost project’s continuous integration and deployment by using some of the DevOps platforms there is available these days (e.g. Shippable, CodeShip, Travis CI). When configured properly, these “pipelines” offer a great way to develop and run the web application on multiple environments at the same time with almost fully automated management. Even though Docker’s most significant impact on users is not computing performance, that aspect can’t be ignored either. With right configuration, Docker containers can outperform traditional kernel-based virtual machine quite easily (Felter et al., 2015). Thus, this technology is without consideration a huge influencer in web developer world and probably will have a significant impact in research also.

This research focuses on performance differences between pure metal servers, KVMs, and containers, and scalability between containers and KVMs. In particular this study will try to answer to the two following questions:

- How do containers and hypervisor based virtual machines differ in performance?
- How does it affect to the usage of these technologies?

This research is based on related research, analysing and comparing studies and providing conclusions based on them.
This paper is structured in following way. Terminology is explained in the section 2. Research methods and literature definition are defined in the section 3. Section 4 covers previous research and literature review. Discussion and implications are covered in section 5 and conclusion at section 6.
2. Terminology

This section explains briefly the main terms discussed in this paper.

2.1 DevOps

DevOps (Development Operations) is a set of methods to set up and maintain development process in cloud-based software development (Kang, Le, & Tao, 2016). So called continuous integration- and continuous deployment pipelines are used for testing and delivering just developed code into different deployment environments. One of the key point in DevOps is to make the process scale up as autonomously as possible whenever necessary. Even though DevOps require very little or none coding, it is one of the most important things in making cloud-based software development and deployment successful (Kang et al., 2016).

Regarding the topic of this paper, DevOps has two goals to achieve. The first one is to get the development environment and automated processes right. This means not only that the version control is set up and hooks for automated tests are configured right, but also that it’s easy for new developers to join the team and set up the development environment regardless of the development machine. The second goal is to make the software deployment process and management as easy as possible. The deployment to production should be as automated as possible (perhaps manual trigger) and as fast as possible, without any downtime. When a node crashes, it should be automatically restarted as fast as possible. The policies of deployment process and production environment management may vary in projects, so these things should be easily configurable.

2.2 Docker container

Docker is quite new and very popular open source tool that provides portable deployment of containers across platforms, component reuse, sharing, archiving and versioning of container images (Boettiger, 2015). Docker containers are not virtual machines (Docker, 2016). Still, they are often referred as VMs since they have many things in common. They both provide an isolated runtime environment for applications to run, and they both are presented as binary artefact which can be transported between hosts (Docker, 2016).

Docker containers share the underlying infrastructure on the host. They exploit LXC (Linux Containers), which consist of Linux Kernel features such as cgroups and and namespaces for performant task isolation and resource control (Joy, 2015). They can be linked to each other, they can depend on each other, and they’re configurable overall. Container’s ports can be bind to hosts port to pass network traffic into the container. That is the only way to interact with the container from outside the host.

Docker lightens the developing by reducing dependencies the developer has to manually install, setup and manage. In addition to that, one can setup and use multiple containers at the same time through a tool called docker-compose. It provides a chance to make a docker-compose.yml –file, which usually describes services (containers), networks and volumes the developer wants to control. One can define a Dockerfile or base image for a service, a list of depending containers, links between containers, runtime environment variables for the containers, networks and so on. Taking web developing as an example, one can successfully boot up a database container with a volume container attached for permanently saving data into the host, application service container(s), authentication
service container, front-end service container and a proxy service container with load balancer, all at the same time, with one command. Sure, this requires also a lot of configuration, but the best part is that the configuration can be really close the same in development environment and production environment.

An example of Dockerfile describing a Nodejs service image:

```dockerfile
FROM node:9-alpine
LABEL maintainer="perttu.karna@gmail.com"

# App source directory
WORKDIR /usr/src

# Copy project definition files
COPY package*.json ./

# Install dependencies
RUN npm install --quiet --only=production && npm cache clean --force

# Copy application files
COPY . .

# Change the ownership of files to the user "node"
RUN chown -R node:node /usr/src/*

# Change current user to "node"
USER node

# Expose port
EXPOSE 3000

# Default launch command
CMD ["node", "bin/www.js"]
```

This example represents a usual way to deploy Nodejs application into production. Main points to notice is that no volumes are described, dependencies are built with production flag, and that the user running the launch command is not root but node.
An example of docker-compose.yml file which would use the Dockerfile above to build web service container and a database container from PostgreSQL base image:

```yaml
version: "3"
services:
  web:
    build:
      context: .
      dockerfile: Dockerfile
      depends_on:
        - db
    ports:
      - "8080:3000"
    restart: unless-stopped
    environment:
      POSTGRES_USER: $PG_USER
      POSTGRES_PASSWORD: $PG_PWD
      POSTGRES_DATABASE: $PG_DB
  db:
    image: postgres:9
    volumes:
      - postgres-data:/var/lib/postgresql/data
    restart: unless-stopped
    environment:
      POSTGRES_DB: $PG_DB
      POSTGRES_USER: $PG_USER
      POSTGRES_PASSWORD: $PG_PWD
    volumes:
      postgres-data:
        external: true
```

In this file, we describe that the web service:

- Will be built from Dockerfile in the current context
- Will depend on the db service
- Should bind the port 3000 to the host port 8080
- Should be always restarted unless stopped via user command

And the db service:

- Uses the official base PostgreSQL image
- Uses external volume container called `postgres-data`
- Should be always restarted unless stopped via user command
- Is not exposed to any port on host machine

For both containers, there are PG_USER, PG_PWD and PG_DB environment variables for PostgreSQL connection setup. These values will be read from `.env` file in the compose file directory. Networks for the containers could be defined in similar way, but for the sake of simplicity, they’re left out of the example above. Default network is bridge.
2.3 Kernel-based virtual machine

Kernel-based Virtual Machines (KVMs) use deeper virtualization than Docker. KVMs require hypervisor on top of the host OS, on which the guest will virtualize the OS and applications. The hypervisor manages and shares all system resources for the KVMs (Adufu, Choi, & Kim, 2015). While Docker containers are based on resource sharing, virtual private server providers use KVM technology to provide more resources to the users that there is physically available on the bare metal server (Adufu et al., 2015).

The typical way of usage of KVMs are using a provider like VirtualBox or Xen. However, the mentioned providers differ quite much in their use cases. VirtualBox is mainly used in desktop environments such as OS X, Windows and Linux (VirtualBox, 2018). The virtual machine can then be installed and launched via VirtualBox Manager, and the user can access the virtual machine desktop if there is one. Xen on the other hand uses a system called *domain 0 operating system*. Xen hypervisor doesn’t run on top of another system, but the *domain 0* is needed for providing the drivers and management capabilities to the installation (Xen Project, 2018). Thus, Xen has seen some usage in cloud services such as AWS EC2 used it initially (Xen Project, 2018).

When comparing to containers, VMs are quite heavy to manage. However, there is a tool bit similar to Docker called Vagrant. Vagrant has been created to tackle the problem of development workflow and environment differences (Vagrant, 2018). Vagrant is a tool for managing virtual machines that web developers use during development. As does Docker, also Vagrant has available a large number of public, community-supported base images, called boxes, for different types of VMs (Vagrant, 2018). It also enables building the virtual machine from a file called *Vagrantfile*, which is much similar to earlier mentioned *Dockerfile*. However, the workflow for managing the software during the lifecycle is a bit different than the one with Docker. A usual way of setting up pipelines and environments include using Vagrant on top of virtual machine providers such as VMware or AWS, and standard provisioning tools such as Chef of Puppet (Vagrant, 2018).

An example of a *Vagrantfile*:

```ruby
Vagrant.configure(2) do |config|
  config.vm.box = "centos/7"
  config.vm.network "forwarded_port", guest: 3000, host: 3000
  config.vm.synced_folder ".", "~/vagrant"
  config.vm.provision 'ansible' do |ansible|
    ansible.groups = {
      'appservers' => ['default'],
      'dbservers' => ['default']
    }
    ansible.extra_vars = {
      db_name: 'vagrant-db',
      db_user: 'vagrant-user',
      db_password: 'vagrant-pwd',
    }
    ansible.playbook = 'ansible/playbook.yml'
  end
end
```
In the example above, we define following things:

- Use CentOS 7 as box
- Forward network traffic from the host port 3000 to the guest machine port 3000
- Mount the current directory as a volume directory to the guest VM directory `/vagrant`
- Define groups and variables for the generated Ansible inventory file
- Execute the Ansible playbook from local system `ansible/playbook.yml`

Unlike with Docker tools, we don’t refer directly to any actual web technology within `Vagrantfile`. The database service and most likely at least one web service would be described and deployed in the Ansible playbook. Just like Docker, Vagrant can be configured in many ways and all the configuration options and provisioners remain outside the scope of this paper.
3. Research methods

This chapter describes how this study has been structured by means of literature search and research methods.

3.1 Research method

This research is pure literature research. It means that this study is conducted by searching scientific articles on related topics, categorizing them and then studying and analysing them. Since the technology this topic focuses on is relatively young, there is no massive amount of studies conducted on this topic. Hence, this study is rather qualitative than quantitative.

One method used for analysing the previous studies is the grid used in appendix B. It is done by selecting articles about empirical studies slightly related to each other and compiling information about them in a grid. One thing to note here is the fact that the grid should not contain specific information about studies, results and so on. Only the main things are presented on high level.

Using the table in appendix B more relevant table for the research was formed. The table (Table 1) contains all the studies that had direct contribution to the section 4.2, excluding the DevOps aspect of performance. The table describes which aspects of performance each study has measured. Due to the variation of both container- and virtual machine technologies in the studies, the winning technology and in case the study had exceptional technologies in comparison (i.e. Docker and Flockport), also the context of comparison have been marked in the table.

3.2 Literature definition

There were two different approaches for finding literature and material for this research. The method was either searching articles from scientific database archives such as Scopus and Google Scholar, or by looking for references in scientific articles, web articles or somewhere else. However, the search terms and viewpoints varied in several ways.

The first thing to do was finding out how much there is articles available on this topic in general manner. Databases were used with simple search terms such as “docker”, “container”, “cloud” and “technology”. Scopus scored 345 results with a search term “docker containers”. It seemed that there is somewhat enough material available.

First “search line” was about finding empirical research articles about performance differences between container technologies and traditional VMs. This was surprisingly easy. With search term “docker AND container AND performance”, 149 documents were found from Scopus. There were several useful articles amongst them when ordered the results by citations. On the other hand, with search term “container AND performance AND comparison” there was 984 articles which was obviously a sign of too broad scope. Finally, term “docker AND performance AND comparison OR evaluation” found 70 results, and with “virtual AND machine” added in between the results narrowed down to 20, which gave enough material to continue to another view on this topic. However, since
this technology is so young, the articles were relatively new and it was hard to find articles which has plenty of citations.

Second search line was about finding research articles about containers and Development Operations. This ended up being slightly harder since this kind of topic is harder to narrow down and in general the empirical research requires a lot more than in performance analysis. It would require organized interviews with users, probably some test cases about usability and productivity of container technology, and so on. This topic is covered pretty well by amateur bloggers and DevOps platform service companies’ documentations but only a few research articles were found on this topic. The process was started with Scopus by searching for “docker AND DevOps”, which gave 17 results. A few of the articles were suitable for this study, and one article was already found on this topic earlier, so it was time to move to the next view of this topic.

While looking for articles related to performance and DevOps, term “reproducible research” kept bumping out from the mass of articles. It turned out that container technology is a win for research community also. The volume of these articles was not that large, but there were a few good researches on this topic. Again, the first article had already been found, so by looking for more articles with search term “container AND reproducible AND research”, 53 results were found on Scopus. The articles picked from this result set felt satisfying enough for moving on towards the next phase of the research.
4. Literature review

This section covers previous research and main findings based on selected studies in terms of performance analysis.

4.1 Related work

As there were a few references on related work in chapter 3.1, this chapter focuses more deeply into it. This chapter provides references into articles about using containers in computer science research and performance comparison between containers and virtual machines.

4.1.1 Docker in research

There are multiple studies that deal with Docker containers in computer science research. Adufu et al. (2015) states that Docker containers beat traditional virtual machines in calculation time and memory management, which results to the Docker being a better environment for high performance scientific applications than KVMs. When comparing containers and virtual machines, Chamberlain and Schommer (2014) find several advantages using containers. First, they state that “they (VMs) are cumbersome to build reproducibly without a high level of systems administration knowledge”. They also mention the size of virtual machines, which is significantly larger than containers, which easily fills the disk. Finally, they are a lot harder to version and archive than containers (Chamberlain & Schommer, 2014).

Boettiger (2015) lists the best practices for using Docker in reproducible research. He states that using Docker during development is important since the environment is as close as possible to the production environment, and that Docker containers should be built from Dockerfile rather than interactively since it’s easier to track what’s going on during the build. He also suggests using and sharing proper base images, as well as sharing Dockerfiles and images overall. These images provide the reproducibility in this manner. Images can be shared through registries such as DockerHub, or as packages.

There are plenty of services that provide developers and researchers a possibility to share and manage the source code of their project (GitHub, GitLab, BitBucket) with as many people as they like. However, usage of these services requires usually a high level of proficiency in computer programming and system administration and adoption of specific languages, data formats and workflows (Chamberlain & Schommer, 2014).

4.1.2 Containers and virtual machines

There is plenty of studies about containers’ performance in comparison to virtual machines’ performance. (Joy, 2015; Kozhirbayev & Sinnott, 2017; Kratzke, 2015; Morabito, 2016; Morabito, Kjllman, & Komu, 2015; Raho, Spyridakis, Paolino, & Raho, 2015; Salah, Zemerly, Yeun, Al-Qutayri, & Al-Hammadi, 2017; Xavier et al., 2013) studied network performance in containers and virtual machines. These studies show in general, that both technologies show small overhead in network performance, though containers do it slightly less. Running containers on top of a virtual machine adds non-negligible overhead in network performance in comparison to bare virtual machine. (Amaral et al., 2015; Felter, Ferreira, Rajamony, & Rubio, 2015; Kozhirbayev & Sinnott, 2017; Morabito et al., 2015; Raho et al., 2015; Salah et al., 2017; Seo, Hwang, Moon,
Kwon, & Kim, 2014; Xavier et al., 2013) studied calculating performance (CPU) in previously mentioned environments. In conclusion from these studies, containers add less overhead in CPU performance than virtual machines. In addition to these, there is also studies revealing stats on disk I/O performance (Felter et al., 2015; Kozhirbayev & Sinnott, 2017; Morabito et al., 2015; Raho et al., 2015; Xavier et al., 2013), memory usage performance (Felter et al., 2015; Kozhirbayev & Sinnott, 2017; Morabito et al., 2015; Xavier et al., 2013) and overall MySQL database read-write performance (Felter et al., 2015; Morabito, 2016). Raho et al. (2015) and Adufu et al. (2015) studied also overall system performance with tools that did not define any specific measured resource area. Each of these studies compare containers (mostly Docker/LXC), virtual machines, and in some cases host machines. The section 4.2 will review these areas more deeply.

4.2 Performance review

Determining the performance differences between different containers and virtual machines is not a simple task. There are plenty of things that effect to the measurement. Not only effects the environment to the tests, but also configurations of virtual machines and containers. These configurations include for example network drivers, port mappings, file system mounting and allocated memory to the guest. It should be noted also that these technologies evolve really fast. However, some kind conclusion from the knowledge based on previous studies can be made to sum up the performance differences. For the reference, in further text referring to a container includes all LXC–based containers such as LXC itself and Docker container. Other types of containers are explicitly mentioned in the context. The categorizing of the resource areas has been done by considering the reviewed studies and categorizing in them.

**Network** performance is one of the most significant part of the container performance since it is used mostly for web applications. When investigating the network performance, the results seem to vary. By using different types of tools, the results indicate that containerized solution outperforms a virtual machine on average. Xavier and his associates (2013) conclude that Linux-VServer has almost native bandwidth and latency closely followed by LXC, leaving Xen behind. Raho et al. (2015) on the other hand state that Xen dom0 shows near native network performance, leaving Docker containers slightly behind. Xen domU on the other hand has the worst performance. Joy (2015) and Morabito et al. (2015) concluded that containers outperform traditional KVM. Salah et al. (2017) concluded that running containers in a virtual machine is not performance efficient and should rather be run on bare metal server.

**CPU** performance seems to be important in scientific applications and heavily used cloud applications. Amaral et al. (2015) observed the slowdown of benchmark application on native, container and virtual machine when running them in parallel and increasing the instances on at a time. The results show that there is not significant impact on performance with CPU-intensive work when comparing containers and virtual machine to bare metal. Morabito and Komu (2015) used plenty of benchmarking tools to find out the CPU performance differences, and the results show that containers outperform virtual machine. The same goes on in the study by Xavier and his associates (2013) where LXC outperforms Xen and OpenVZ container. Raho et al. (2015) found out that even though Docker container shows almost no CPU performance overhead compared to bare metal, in some UnixBench benchmark applications Xen domU and KVM has better results than Docker container. For example, on pipe throughput and pipe-based context switch this is
the case. “On context switch, the virtual machine score better than the host, this is mostly due to caching mechanisms”, they conclude.

**Memory** performance is one of the most important aspects in web services. Needless to say, it is significant part of container technologies also. Xavier and his associates (2013) used a benchmarking tool called STREAM to find out memory performance. They found out that LXC, OpenVZ and Linux-VServer show near native performance. However, Xen seems to be left behind with an average of 31% of performance overhead when comparing to the native throughput. On the other hand, Morabito et al. (2015) used the same benchmarking tool for testing memory performance, and resulted in almost same overhead for LXC, Docker container and KVM. For the reference, they used Qemu emulator, version 2.0.0 as KVM platform. They included in their tests also an operating system called OSv, which is designed for running single applications in the cloud similar to containers are, but they run on top of a hypervisor (Morabito et al., 2015). The OSv shew roughly half of the performance the other candidates were able to perform.

**Disk I/O** split the results between containers and KVM. Raho et al. (2015) used a disk test tool called **IOzone** to measure the disk I/O operations. The results they got for both 4KB and 1MB file sizes indicate the same thing. Docker container outperforms the KVM and Xen DomU only on standard write operation, while on read, re-read, random read and re-fread the results show too high deviation to determine who is actually leading with the file size of 4KB. They suppose the tests with small file size are highly influenced by caching mechanism. With 1MB file sizes the deviation is significantly lower and it’s fair to say that Docker contain can’t match the KVM and Xen DomU in any other category than standard write. The host system and Xen Dom0 perform similar way and they score better than Docker container in every test that do not show too high deviation to evaluate the results. However, the KVM and Xen DomU outperforms the host system and Xen Dom0 on average in read operations with small file sizes. Morabito et al. (2015) on the other hand end up in slightly different results. In their tests, Docker container and LXC perform significantly better than the KVM. For sequential read and sequential write tests, containers show near native performance as the “KVM write throughput is roughly a third and read throughput almost a fifth of the native one”, they state. On random write LXC lost roughly by 10%, Docker about 15%, and KVM 50% to the host machine, and on random seeks for the same list goes 11%, 42% and 93% respectively. All of these tests were performed with **Bonnie++**, and the last test they did for disk I/O was done with a tool called **dd** which they used to “test hypervisors and containers capacity to read and write from special device files such as /dev/zero/”. The results show that against the host, Docker lost by 7%, LXC by 25% and KV by 59%. The results by Xavier et al. (2013) show the same kind of results. Their tests were performed with **IOzone**, just as with Raho et al. (2015) but in this case the record size was 4KB and the file size was 10GB. LXC and Linux-VServer both perform at near native performance, as the Xen show a bit over half of the native performance on write, and a half of the native performance on read.

Felter and his associates (2015) chose to evaluate a **database**, MySQL to be exact, since it’s “widely used in the cloud, and it stresses memory, IPC, network and filesystem subsystems.” Thus, this test was performing MySQL throughput by transactions per second against concurrency. They used **SysBench oltp** benchmark to run tests against one instance of MySQL 5.5.37. They ran their tests for various configurations for Docker, but the results were quite similar to the native with all of them. The configurations were host network with a volume (net=host volume), normal network and a volume (NAT volume), and normal network with storing the database in the container (NAT AUFS). Only the last option got clearly left behind the others. On average the KVM scored remarkably under half of the native performance.
Overall system performance was measured by Raho et al. (2015) and Adufu et al. (2015). They were labelled under overall system performance since both of these test cases didn’t define a certain resource they were measuring. Raho and his associates (2015) used Hackbench to perform their tests. In each of their tests, “1000 messages of 100 bytes are sent between a varying number of receiver/sender pairs”. The number of tasks increase as the number of sender/receiver pairs increase. The results show that the Docker container, the host system and Xen Dom0 perform slightly better than KVM and Xen DomU, which show about 5% overhead for 640 tasks. Adufu et al. (2015) compared Docker container to KVM by using a molecular modelling simulation software called autodock3. The results show that it took 176 seconds to boot up a container and execute the software in it, while the same process took 191 seconds with the VM. In other test, they launched multiple instances of containers and VMs, each with a memory allocation of 3GB. They ran the autodock3 in them in parallel. With total memory allocation of 12GB and 36GB, the Docker containers had lower execution time. However, with 24GB, the VM was slightly faster. With total memory allocation of 48GB the autodock3 failed to run in the VMs “due to Openstacks available memory overcommit ratio of 1.5 RAM.”

To get more perspective for practicalities when developing web services, let’s take a look on the DevOps side. When speaking of performance, Kang et al. (2016) measured time for provisioning Docker containers and virtual machines. They used OpenStack, a common open source infrastructure cloud managing software as a testbed. According to them, OpenStack architecture usually contains three types of nodes. The first is control plane, which runs management services. Compute nodes and network nodes do all the VM provisioning tasks. On their implementation, they used controller service (containing Glance, Keystone, Nova, Neutron servers, etc.), MySQL service and RabbitMQ service each in their own container on each of the three nodes. The compute and network nodes contain also one container with Nova and Neutron agent services in it. What they tested was different types of DevOps tasks. On deployment task, the containerized approach took 358 seconds to deploy while the VM approach took 535 seconds. Upgrading the controller instances took 207 and 326 seconds, respectively. For container, it took 32 seconds to recover from MySQL failure while the same thing took 74 seconds for VM. When looking at the deployment process, it can be split into two phases: initialization and scaling out. The process starts by initializing one instance of each service on one node in parallel, which takes for containers about 20-30 seconds for messaging and database services, and about 240 seconds for controller service. For the VMs, it takes about 60 seconds for messaging and database services and about 280 seconds for controller service to boot up. When comparing these two, the difference is remarkably large, only about 40 seconds. The more significant difference lies in the scaling process, where each service is added into other nodes. For containers, this takes roughly the same time for database and messaging services, but only about 125 seconds for controller service. In contrast, for VMs it takes about 140 seconds to scale out messaging and database services and about 250 seconds for controller service. Also, when looking at the sizes of these images, the database service containing MySQL is 530MB for VM and 107MB for container. The difference is significant.
<table>
<thead>
<tr>
<th>Reference</th>
<th>Network</th>
<th>CPU</th>
<th>Memory</th>
<th>Disk I/O</th>
<th>Database</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Xavier et al., 2013)</td>
<td>Cont.</td>
<td>Cont.</td>
<td>Cont.</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(Felter et al., 2015)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Cont.</td>
<td>-</td>
</tr>
<tr>
<td>(Kozhirbayev &amp; Sinnott, 2017)</td>
<td>Depends on tool</td>
<td>Flockport container (over Docker)</td>
<td>Docker (over Flockport)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(Salah et al., 2017)</td>
<td>VM (EC2 over ECS)</td>
<td>VM (EC2 over ECS)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(Morabito et al., 2015)</td>
<td>Cont.</td>
<td>Cont.</td>
<td>Cont.</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(Amaral et al., 2015)</td>
<td>-</td>
<td>Cont.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(Joy, 2015)</td>
<td>Cont.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(Adufu, Choi, &amp; Kim, 2015)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Cont.</td>
</tr>
</tbody>
</table>
5. Discussion and implications

When looking at the literature review results, it’s quite easy to determine which virtualization solution is more performant at the moment. There is plenty of evidence that container-based virtualization is more performant than hypervisor-based virtualization.

The containerized solution was clearly faster than KVM on network performance. However, it’s worth mentioning that the results vary between different types of virtualizations. For example, Linux V-Server and Xen’s dom0 layer seem to be almost as performant as native computer. Also, when testing the database on container and VM, the containerized solution was obviously faster. This test case represents a good case of using containers in web technology stack, since it uses system level resources and networking. The overall system results on hackbench test by Raho et al. (2015) show that container is more performant than KVM. Adufu et al. (2015) tested also overall system performance, which produced the same results. This is most likely due to the fact that containers boot up time is so small. This results in good performance in DevOps sense. The results by Kang et al. (2016) show that containers fit better on DevOps tasks than VMs. For example, containers recover from database failure significantly faster than VMs. Especially scaling process was faster on containers than on VMs.

When it comes to memory, it’s hard to determine the clear winner. It seems that on average, there is no significant performance difference between virtualization technologies and containers. It’s the same case with CPU performance. Even though the container outperforms the VM on average (Morabito et al., 2015; Xavier et al., 2013), on some test cases the VM can outperform even the host. As I referred earlier to Raho et al. (2015), this is most likely due to the caching mechanism. The caching mechanism affects also on disk I/O operations. The results on disk I/O performance vary too much to be able to say which technology is faster on average. The only thing that was clear was standard write operation, which is slightly faster on container than on KVM.

As Salah et al. (2017) concluded that the containers should not be run on VMs but rather on native systems. Ironically, this is the opposite case when usually implementing web services. Usually the web service providers make it so much more profitable to use virtual private servers than bare metal servers. Also, the cloud-based services rely on infrastructure provisioning and increasing resources on demand, which makes the virtualization technologies kind of mandatory.

Even though the literature review did not address directly to the hands-on usage of VMs and containers, it should be noted also. Setting up a container with Docker is remarkably easier with do than to set up a virtual machine on top of a host machine. The possibility to build and launch containers from files and the speed of build and launch process makes containers superior in most usage scenarios when comparing to VMs. The only rival in development environment is Vagrant, which offers the developers almost the same features. The downside of Vagrant is that it still uses heavier images (boxes), which do not build and boot up as fast as containers. However, Vagrant approaches the same problem as Docker with a bit different solution which means that they aren’t exactly comparative and both have their own use cases.

The results on this paper can be used when designing the architecture for a web-based service. There are several things that should be noted before making any decisions in either direction. Firstly, will the software be publicly available and widely used, or will it be a system used in an internal network? The volume of users and the need of availability...
are critical things to notice. If the software is expected to get a lot of users over time and it will be publicly available, then it’s a sign of cloud-based software. Whether to use containers on top of VMs or not is up to the probability of gaining high peaks on usage of the software and the nature of the development process. Continuous delivery to production is probably easier to set up with containers than without. VM provisioning can be automated and the deployment on new nodes can be automated with right DevOps tools. Pushing images from deployment pipelines to an image repository gives a change to version deployments and revert back deployment in case some release doesn’t work as supposed to. On the other hand, some systems may gain massive amounts of users even in a corporation’s internal usage. If that’s the case, it would be best to consider a self-hosted containerized solution. This makes it possible to run containers efficiently on bare metal servers, but they’re still easy to manage.

Another implication for the results on this paper is research. For example, when designing implementation for computer science research in which the researchers will be located at multiple places, it’s easier to share and version images and collaborate by using containers. Also, the actual implementation of test cases is faster and easier, especially when implementing software that will be run on multiple containers, or which will be constructed by microservices.
6. Conclusions

This paper covers literature review about studies performed mostly during the past five years. These studies are used to compare the performance difference between LXC based container solutions and virtual machines (KVMs mostly). The defined research questions were:

- How do containers and hypervisor based virtual machines differ in performance?
- How does it affect to the usage of these technologies?

The research method is described in section 3 and the literature review can be found from section 4.

The results of this study reveal that LXC based solutions are generally more performant than KVMs. Containers beat KVMs especially at network level operations and at overall system level operations. For example, the boot up time of the container is significantly lower than for virtual machine in general. This results in better scalability.

On the other hand, the winner for some resource areas were harder to determine. The memory and CPU performance results show variation between studies which makes it context-sensitive to find the winner. The containers outperform the VMs on average (Morabito et al., 2015; Xavier et al., 2013). However, sometimes the VMs can outperform even the host machine at some operations. It’s very likely that the caching mechanism has something to do with this (Raho et al., 2015). The caching mechanism affects also to the disk I/O operations, which resulted in large variation of results. Only the standard write operation showed low enough variation to determine that the containers were slightly faster than KVMs.

These results can be used when designing web-based software architecture. There is plenty of things to take into consideration when designing such software, and the key characteristics in deciding is usually the amount of end users, the probability of the user base to grow up, the availability of the software (public or organizational) and the nature of the software (will there be high usage peaks, how much does an average user stress the system). In addition to web-based software stack, the containers can be used in other kinds of software projects. Especially in research the containerized solutions are profitable solution since they provide the necessary isolated and reproducible environment but significantly easier and faster than with VMs.

However, it should be noted that the usual way of deploying software, whether using containers or not, is to deploy it to the cloud. Even though there are bare metal servers, the cost of virtual servers are usually so competitive that it does make sense to deploy on top of virtual machines.

Since these technologies are improving constantly, this kind of performance studies keep the users updated. However, in the future it would be good to include more containerized solutions into the comparison. Such technologies could be for example Flockport, OpenVZ and CoreOS Rocket. In addition to that, the aspect of security often rises when comparing containers and VMs. The VMs are usually considered more secure than Docker containers in system level, which would be good place to study. Improving the security of Docker containers through design science method would be one interesting topic.
Another path to take in further studying is the differences of LXC technologies and pure virtual machines in DevOps. This is a vast topic and obviously should be properly narrowed down such as picking up just two technologies and exploring their usage in real world development teams. One example of such technologies would be Docker and Vagrant as they try, at least partially, to solve the same problem of environment consistency across the software development lifecycle.
References


# Appendix A

<table>
<thead>
<tr>
<th>Reference</th>
<th>Research Goal</th>
<th>Research methods</th>
<th>Resources measured</th>
<th>Main findings</th>
<th>Implications</th>
<th>Further studying</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miguel G. Xavier, Marcelo V. Neves, Fabio D. Rossi, Tiago C. Ferreto, Timoteo Lange, Cesar A. F. De Rose (2013)</td>
<td>To evaluate the performance differences in native environment and container environment for HPC applications in a perspective of using containers as isolated environments</td>
<td>Empirical research; use benchmarks to test performance between native environment and container environment</td>
<td>Network I/O, CPU, Disk I/O, Memory</td>
<td>The container based technology is not mature enough yet in a perspective of complete isolation. It can isolate successfully the CPU, but not memory, disk and network. However, this technology can still be very attractive in HPC environment due to its low performance overhead.</td>
<td>HPC (High Performance Computing) clusters.</td>
<td>The performance and isolation of container-based systems for other kinds of working loads, including I/O bound applications.</td>
</tr>
<tr>
<td>Wes Felter, Alexandre Ferreira, Ram Rajamony, Juan Rubio (2015)</td>
<td>To isolate and understand the overhead introduced by virtual machines (specifically KVM) and containers (specifically Docker) relative to non-virtualized Linux.</td>
<td>Empirical research; use benchmarks to test the performance of containers, virtual machines and native environments</td>
<td>Database performance, CPU, Memory, Disk I/O</td>
<td>Both KVM and container technologies are mature technologies. KVM performance has improved significantly since its creation. In every test case Docker equals or exceeds KVM in performance.</td>
<td></td>
<td>Software deployment on servers, how cloud infrastructure should be built.</td>
</tr>
<tr>
<td>Authors</td>
<td>Title</td>
<td>Methodology</td>
<td>Results</td>
<td>Implications</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>-------------</td>
<td>---------</td>
<td>--------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zhanibek Kozhirbayev, Richard O. Sinnott (2017)</td>
<td>To compare and contrast a range of existing container-based technologies (Flockport and Docker) for the Cloud and evaluate their pros and cons and overall performances.</td>
<td>Literature research: critically review performance experiments of related works and identify performance-oriented case studies; Empirical research: implement several case studies to evaluate the performances of microhosting technologies.</td>
<td>Network I/O, CPU, Memory, Disk I/O. There were nearly no overheads on memory utilization or CPU by container technologies used in case studies, while I/O and operating system interactions incurred some overheads.</td>
<td>How the architecture of Cloud environments should be designed. The implications and performance impact on shared resources.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tasneem Salah, M. Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri, Yousof Al-Hammadi (2017)</td>
<td>To compare Container-based cloud services and VM-based cloud services in performance.</td>
<td>Experimental research; provision different test services and use testing tools to get performance results.</td>
<td>Network I/O, CPU. VM-based web services outperform container-based web services (AWS EC2 and ECS in this case), due to the reason that Amazon runs containers on VMs.</td>
<td>Similar measurement s on other popular IaaS cloud platforms (Google Compute Engine, Microsoft Azure), or other services (database, analytics, streaming).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Roberto Morabito, Jimmy Kjällman, Miika Komu (2015)</td>
<td>To quantify the level of overhead introduced by these platforms and the existing gap compared to a non-virtualized environment.</td>
<td>Empirical research: use benchmarking applications to evaluate performance differences.</td>
<td>Network I/O, CPU, Memory, Disk I/O. Containers are more performant than VMs with the cost of security.</td>
<td>Repeat the study with other similar technologies e.g. with Linux Container Daemon.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Author(s)</td>
<td>Study Title</td>
<td>Research Methodology</td>
<td>Results</td>
<td>Future Work</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>----------------------</td>
<td>---------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Moritz Raho, Alexander Spyridakis, Michele Paolino, Daniel Raho (2015)</td>
<td>To present a comparative performance study of two open source hypervisors and an open source container, form ARM platforms</td>
<td>Empirical research: test different environment with a bunch of benchmarking tools</td>
<td>Benchmark results show very low performance overhead for all three solutions, with some variation depending on the performed test</td>
<td>Extend the study by running the container in VMs, and consider multi core performance results and scalability tests</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Theodora Adufu, Jieun Choi, Yoonhee Kim (2015)</td>
<td>To demonstrate that the lightweight feature of container-based virtualization compared to hypervisor-based virtualization reduces the overall execution times of HPC scientific applications</td>
<td>Empirical research: test virtualization performance with heavy scientific applications</td>
<td>Overall execution times for container-based virtualization systems are less than in hypervisor-based virtualization systems for scientific applications</td>
<td>Investigate a framework for running scientific applications with different characteristics, in cluster of container-based virtualization systems and evaluate the optimal environment for efficient execution</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nane Kratzke (2015)</td>
<td>To study the performance impact of Linux containers on top of hypervisor based virtual machines logically connected by an (encrypted) overlay network</td>
<td>Empirical research: test network performance with different setups of multiple servers provided by AWS</td>
<td>Containers on top of the hypervisor-based virtual machine have non-negligible impact on network performance.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Marcelo Amaral, Jordà Polo, David Carrera, Iqbal Mohamed,</td>
<td>To compare the performance of CPU and network</td>
<td>Empirical research: use a bunch of benchmarks to find the</td>
<td>Nested-containers approach is a suitable model;</td>
<td>To provide a benchmark analysis guidance for</td>
<td>Implement an extension on Docker to fully support OVS and</td>
<td></td>
</tr>
<tr>
<td>Merve Unuvar, Malgorzata Steinder (2015)</td>
<td>running benchmarks in the two different microservices architecture model: master-slave and nested-container</td>
<td>performance differences</td>
<td>doesn’t have a significant impact on CPU performance</td>
<td>system designers</td>
<td>study the performance of a real application implementation based on microservices architecture using nested-containers</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>---------------------------------------------------------------</td>
<td>------------------------</td>
<td>-----------------------------------------------</td>
<td>-----------------</td>
<td>------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Bukhary Ikhwan Ismail, Ehsan Mostajeran Goortani, Mohd Bazli Ab Karim, Wong Ming Tat, Sharipah Setapa, Jing Yuan Luke, Ong Hong Hoe (2015)</td>
<td>To evaluate Docker as a platform for Edge Computing</td>
<td>Empirical research: complete tests and analyze results</td>
<td>Deployment and termination, resource and service management, fault tolerance, caching</td>
<td>Docker provides fast deployment, small footprint and good performance which makes it potentially a viable Edge Computing platform</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thouraya Louati, Heithem Abbes, Christophe Cérin, Mohamed Jemni (2017 / 2018)</td>
<td>To propose LXCloud-CR; decentralized Checkpoint-Restart model based on a distributed checkpoints repository using key-value store on a Distributed Hash Table DHT.</td>
<td>Design science</td>
<td>Image snapshotting and failure detectioning</td>
<td>To provide users the Checkpoint-Restart model as a SaaS by changing the LXC Web Panel tool, to work with tightly coupled applications and to manage dynamically the frequency of checkpoints to reduce the storage overhead</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ann Mary Joy (2015)</td>
<td>To provide a comparison between Linux</td>
<td>Empirical research: use benchmarks for testing</td>
<td>Network, scalability</td>
<td>Containers have outperformed virtual</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| containers and virtual machines in terms of performance and scalability | different resources | machines in terms of performance and scalability | }

29