UNIVERSITY OF HERTFORDSHIRE

Faculty of Engineering and Information Sciences

MCOM0177 Computer Science MSc Project (Online)

Final Report

September 2019

Development of deployment management system to solve microservices deployment complexity and runtime scalability challenges

Walid Gad

Table of Contents

Table of Figures

Acknowledgements

I would like to thank Dr Xianhui Cherry for his assistance and continuous support during the past few months, and my supervisor Dr John Sapsford for his invaluable research support and academic advice throughout this research.

Finally, I would like to thank my wife and my family for inspiration and understanding during the course of this project, without their assistance this work would not have been possible.

Abstract

Microservice architecture is a novel technology that was introduced as a solution for the incapability of existing software architecture to deal with the continuously evolving technology and fluctuating demands of users. However, technology poses a challenge in the form of the increased complexity of the deployment and scaling. The aim of this study is to develop a system through which the complexity found in microservice architecture can be reduced and furthermore, resources and performance of the application can be monitored and scaled automatically depending on the demand and requirement. For this purpose, the study explored two approaches, i.e. virtualization and containerization. In virtualization, Microsoft Hyper-V was used for the provisioning of virtual machines and in containerization, Docker containers engine was used for the provision of containers. The results of the experiment shown that the average time taken for the provisioning of containers was lesser than that of VMs. However, in deployment, the database image of container consumed more time than VM whereas application image of the container took lesser time than VM. Moreover, the average time taken for scaling container was higher than the average time taken for scaling VM.

1 Introduction

1.1 Background

The main aim of this report is to find a solution for the complexity issues found in the microservice architecture. The microservice architecture consists of a complex network as it is based on the structure of several small independent services interacting with each other. This complexity can make the debugging, auditing, monitoring, and deploying of the whole application increasingly difficult. The complexity of microservice architecture can cause a delay in the deployment of the application, which will consequently discontinue the lifecycle of software delivery. Another hindrance in continuous delivery of software was found to be because of static resource provisioning as fewer resources can cause application response delays and interruptions; whereas, excessive resources can result in unused resources which would increase the operational cost. Hence, in order to tackle this issue, automatic scaling is adapted by the architecture.

Most of the studies focused on exploring the implementation of deployment and auto-scaling for cloud-based hosting; however, in this project, the research is focused on the implementation of deployment and auto-scaling on on-premise servers. This will allow the developers to maintain continuous software delivery without needing to have host applications over the public cloud. In particular, the project is attempting to find an optimal solution which will provide continuous software delivery, deployment of all the services of microservice-based applications in different environments and continuous monitoring of hosted applications for autoscaling.

1.2 Research Aim

The aim of this research is to develop a system through which the complexity of microservices-based applications deployment can be simplified, resources and performance of applications can be monitored, and service scaling can be automated according to demand requirement.

1.3 Objectives

The core objectives of the research include:

- To develop a system through which the complexity issue of deploying and updating microservice-based applications in distinct environments can be solved.

- To simplify the transition deployments between different application environments such as testing, staging, and production environments.
- To minimise the time between the development and deployment of services by providing auto-provisioning of service hosting, which would allow continuous software delivery.

However, the advanced objectives set for this project include:

- Automation of application scaling upon demand.

1.4 Research Question

The research question used for this study is:

"How to solve the complexity of microservice-based application deployment and how to automate service scaling according to the requirement?"

1.5 Methodology

The methodology used in this project is research and development. This research methodology employs applied research where the accepted theories and concepts are applied over existing technology in order to find effective solutions for the issue, under consideration (Rajasekar, Philominathan and Chinnathambi, 2013). In the first step, the study identifies the deployment challenges of microservice-based applications, areas of complication, and deficiency of automatic scalability present in the host environments. The required data for the research is collected from academic studies, researches, and existing practices in the industry. Moreover, according to this study, the development of deployment management system is making an effort to find a solution to the complexity issue of the deployment process along with automatic resource scaling. Furthermore, the experiment is also conducted in which data is collected and analysed to examine the system usability and performance while checking if the project is successfully meeting the research objectives.

In order to solve the deployment complexity of the system, two approaches are explored in the research to host application services.

- In the first approach, a virtualization engine (Microsoft Hyper-V) is used, through which virtual machines can be provisioned. On these virtual machines, the platform to host application services is installed according to the microservice framework. As the purpose of the system is to simplify the development process, the provisioning of virtual machines

is automated by the system, along with the installation of the operating system and application platform.

- In the second approach, the container engine (Docker) is used for the purpose of automation of application deployment. In Docker, an abstract layer to host the applications and platforms over the host operating system is used. Containers offer isolation to the processes while minimising the utilisation of resources in comparison to virtual machines, where an ideal environment is provided for the microservice-based applications.

Similarly, in order to attempt automation of scalability of hosted applications, one approach is analysed. Scaling mechanism is developed to scale up the computing resources of the application service according to the load demand. For example, when an application service receives peak load from user requests, the scaling mechanism increases the CPU count or memory capacity for the service to accommodate the demand.

The two deployment approaches are then compared with each other, which provides the approach that is best suited to solve the complexity issue.

Lastly, the management methodology used in the project is agile project management with Kanban framework. Kanban is a project management framework, originated in Toyota. Kanban provides various benefits in project management, such as visualization for workflows and minimization of work in progress (Brechner, 2015). This method limits the opened tasks and focuses the attention towards the tasks at hand.

1.6 Deliverables

The deliverables submitted at the end of the project are:

a. The deployment management system interface.
b. System APIs.
c. System Engines.

1.7 Technologies and frameworks used on this project

- Windows Server 2019
- Docker
- Microsoft SQL server
- MYSQL
- Hyper-V
- Rabbit MQ 3.7.15

- Angular 7
- .Net Core 2.1
- .Net Framework 4.6
- PowerShell scripting

2 Chapter 1: Background

2.1 Microservice Architecture

In modern software development techniques, no single process is compatible with all kinds of software development projects as they are dealing with constantly evolving conditions and dynamic environment (O'Connor and Clarke, 2015). Consequently, the development process requires some level of tailoring in accordance with the situation. Cerny, Donahoo and Trnka (2018) stated in their study that due to the requirements of change and evolution in management, architectures such as Java RMI, Enterprise Service Bus and Common Object Request Broker Architecture (CORBA) were introduced in the market and Service-Oriented Architecture (SOA) was popularly used by various large-enterprises in order to meet the industrial demands. However, microservice architecture seems ready to take over from SOA as the dominant architecture, used in the industry.

The microservice architecture was first introduced by Dr. Peter Rodgers in 2005 in the presentation of Microsoft Service Edge Conference, where he explained that microservice architecture works by dividing the development of a single application into multiple small services (Castiglioni, Proietti & De Gaetano, 2016). Each of the small services runs its own processes and establishes communication with each other through lightweight mechanisms such as HTTP APIs. As SOA also emphasized the importance of self-management and self-adaptation of services along with their lightweight nature, it can be stated that microservice architecture is an extension of SOA (Zimmermann, 2017). However, unlike SOA, the microservice architecture supports the division of services without the implementation of global governance over the system. It leads to higher autonomy between services since they are not depending on the agreement with other services, regardless of the fact that this makes the services responsible for their own management and the establishment of effective communication with other services.

2.1.1 Microservice Architecture Advantage

Microservice architecture introduces multiple advantages in software development processes, especially in agile software development (Taibi et al, 2017). Some of the advantages of Microservice Architecture are: *fault isolation* – as the services are divided into multiple independent components, each service can be deployed and redeployed without compromising other services or the whole application; *quicker deliveries* – different autonomous teams can be working on different services simultaneously which would allow them to deliver the application

faster; *flexibility in technology* – as the microservices are deployed autonomously, different technologies can be applied to them as well. This would allow services of the different technological stack to operate in a combined environment effectively while eliminating the limitation of using specific technologies and ease of understanding. With all the components of the application divided, it becomes easier for the developers to understand and maintain the application (Villamizar et al, 2015). Furthermore, microservice architecture also offers more agility and improves the continuous delivery approach of the software. Still, microservice architecture possesses some challenges, which include increased complexity, as microservices contain more components they require careful planning and automation in the processes of inter-service communication, monitoring, testing, and deployment. Another challenge is associated with the requirement of cultural changes, as the adaption of microservice will require the cultural environment to shift to a mature agile and DevOps culture (Taibi, Lenarduzzi, and Pahl, 2018). Therefore, in order to effectively implement microservice architecture over the system, its pros and cons need to be evaluated thoroughly.

2.2 Virtualization vs Containerization

This section discusses and compares the concepts of virtualization and containerization. In virtualization, various guest operating systems are running over a single physical server and they are called Virtual Machines (VM). Hyper-V is an example of VM runtime engine (hypervisor), that enables a server to run multiple operating systems as Virtual Machines over Windows host (Microsoft, 2018). On the other hand, containers consist of lightweight operating systems, which use fewer resources and time for which they can be described as flexible tools. These tools are used for the purpose of packaging and deploying software applications and services (Peinl, Holzschuher, &Pfitzer, 2016). While Virtual Machines (VM) can be deployed over a single system, they can run their own operating system and be completely isolated from other VMs that are deployed over the same machine. However, containers can share the environment of a singular OS and utilise the libraries, binaries, and drivers associated with it (Mavridis&Karatza, 2019). This would allow the containers to run applications with far less overhead as compared to VMs because the hardware is not utilised extensively in this scenario.

Virtualization technologies can be divided into server virtualization and network virtualization. In server virtualization, entirely new multiple virtual systems are established over an existing system. For example, establishing a virtual machine running Ubuntu Linux with

hypervisor over a machine that is already running Windows Microsoft (Jin, Wen, & Chen, 2012). Server virtualization offers partitioning – ability to run several OS over one machine, isolation – secure and isolated environments, hardware independence – ability to migrate VM to any physical server, and security – all the environments are running with their own security protocols in place (Barrett & Kipper, 2010). However, some disadvantages of server virtualization reside in its expensiveness – high product cost as compared to physical servers; reduced performance – as the resources are shared, it would increase the overhead; learning curve – new technology integration would require new training (Wang et al., 2012; Jin, Wen, and Chen, 2012).

Containerization and virtualization both make up the foundations of cloud computing technology and present their own set of advantages to the system in the form of less performance overhead and robust isolation respectively (Felter et al, 2015). Hence, various studies have suggested a combined implementation of containers and VMs where containers are run on top of VMs, as this does not /only increase the security of the system, but also overcome the issue of incompatibility between hardware and software in a physical server and also provide better management and easier upgrade of the system (Celesti et al, 2016; Mavridis&Karatza, 2017; Mavridis&Karatza, 2019). Container-Based virtualisation is a lightweight approach towards computing, which can maintain and deploy software application and services far more efficiently as compared to simple VMs.

2.3 Manual Application Deployment

Manual application deployment requires the developer to be involved at every stage of the software delivery process where everything from file configurations, altering database to synchronisation of folders and restarting service is done manually. The stages of manual deployment can vary depending on the project as all the software products are unique (Soni, 2015). While manual deployment has been practised in various organisations for a long time, they are all progressing towards automated deployment because of various reasons. One problem is of human error; since everything in the process is done by human beings, the probability of errors is high because humans are prone to forget or overlook things. Similarly, another issue faced in manual deployment is lack of transparency and accountability, as there is no centralised space where the developers can see what version of the application was deployed or who have deployed it. Still, the biggest hindrance comes in the form of delayed deployment because around 27% of the respondents of the survey stated that the manual deployment process takes 15-30 mins. However,

40% claimed that it took 1-3 hours a day (Deploy, 2013). This scenario led to around 60% of the surveyed firms towards deploying applications only once or twice a month. All these issues have led the giant enterprises to adopt automated deployment processes for the production of their software applications.

2.4 Automate Deployment using Scripts

Software applications designed with Microservice Architecture are made of highly flexible, scalable, and loosely coupled services (Dragoni et al., 2017). This feature enables the software development process to ensure continuous deployment and autoscaling over the application as microservices allow the applications to be split into separate units, which can be deployed independently (Lewis & Fowler, 2014). However, the focus of these practices is currently over single microservice and they fail to utilise the benefits offered by the interdependencies within the architecture. These benefits include optimisation of global scaling (like decreasing the performance overhead caused because of redundancy in the process of scaling each microservice one-by-one through the detection of the inbound traffic) or avoiding domino effects caused by the unstructured scaling (like outages or increased downtime) (Woods, 2015). Hence, it is imperative that the microservice architecture is used on its full potential in order to achieve optimisation in software development processes. As the popularity of cloud computing increases, automation of application deployment has also been studied thoroughly and various system management tools have been designed as a solution (Bravetti et al., 2019). One approach of automated deployment is continuous deployment in which the delivery of software products is done with high frequency. This process is especially beneficial on large scale projects where organisations, developing software products are able to roll out new functionalities at a fast pace (O'Connor, Elger, and Clarke, 2017). Thus, the automated deployment also offers a competitive advantage to the organisations with fast and frequent deliveries of new products and services. While automated deployment using scripts offers various benefits, it is not devoid of disadvantages. These disadvantages factors such as *heterogeneity* – because of the use of varying technologies in the services, the scripts consist of varying structure and templates, which can introduce inconsistencies and complexities in the adaptation of scripts. The second disadvantage is associated with the *SLA requirements*, as strict SLA requirements (i.e. availability, performance related to time and cost, etc.) are applied on most of the applications that are built over microservices architecture; hence, developers are required to update the scripts in order to

minimise the downtime and cost associated with the application (Boyer, 2018). In order to effectively employ automated deployment over the system, all the positive and negative aspects of the process should be evaluated thoroughly.

2.5 Scaling Applications

In this era, there is continuously fluctuating demand of resources for software applications as the technology is advancing. In short, this aspect has introduced the need for application scaling. With the scaling of applications, the computing resources can be adjusted in a real-time manner in order to maintain the performance of the applications and meet the requirements of Quality of Service (QoS) (Bacigalupo et al., 2011). At the same time, it also helps in reducing the operational cost of the infrastructure, while allowing the businesses to remain competitive in the market.

The most common method used for scaling of the resources (Scaling Up) is based on managing the capacity of service computing resources where new resources are added to the service according to demand (Han et al., 2012). However, this method can result in increased overhead and cost as the permanent allocation of new resources can result in wasting the computer resources, specifically the ones that are not used by the applications (Bilal, Vajda & Tarik, 2016). It has been established that the up- and down-scaling of application resources does not necessarily require alterations to be made on the computing infrastructure, as it can be done by modifying virtual service resources. This may include the capacity of the existing resources that produce similar results in minimal time and cost (Bacigalupo et al., 2011). In order to achieve more cost-efficient scaling of resources, a lightweight scaling approach was proposed by Han et al. (2012). This approach was particularly beneficial to already deployed applications because of the improvements introduced in the resource utilisation, according to the varying demands of the application. The main concepts of the approach included fine-grained scaling approach, which is associated with the scaling of the applications at the level of underlying resources such as CPU and memory. Another concept includes improved resource utilisation that deals with using the idle resources to release loads from overworked resources. This approach was evaluated in the research and it was found that lightweight scaling can be applied to reduce the operational costs and increase resource management.

2.6 Automatic Application Scaling

Automated scaling is commonly applied with the help of a set of rules, defined by the service owners that explain how the up- and down-scaling of the resources will occur in order to

effectively manage the varying workload (Taherizadeh and Stankovski, 2018). The degree of automation can be varying based on the requirements of customisation and abstraction for users. In automated scaling of applications, the users will no longer have to make decisions about the time and resources, required for the deployed application. Instead, the application service hosting will be updated with the required computing resource depending on the rules and load demands, (Xiao, Chen, and Luo, 2014). For the purpose of maintaining the QoS of the application, under the conditions where the workload is varying dynamically, it is vital to employ automated scaling over the applications so that they can maintain their relevance to the business (Taherizadeh et al., 2018). It allows the organisations to ensure the quality of their software products; hence, maintaining their competitiveness in the market. Taherizadeh and Stankovski (2018) in their study proposed a lightweight container-based virtualisation concept, adaptation of which can improve the application performance and resource utilisation for dynamically varying workload. When the autoscaling of resources is inadequate, the changes in the workload intensity are not addressed efficiently, which results in under-provisioning of the resource; thereby, resulting in decreasing the performance of the application or over-provisioning of the resources – which means some resources are sitting idle (Chen &Bahsoon, 2015). Hence, the lightweight container-based virtualisation improves the autoscaling process in software applications hosting. It is due to the fact that it allows dynamic specifications for setting rules of autoscaling and for the purpose of getting a more fine-grained reaction to the fluctuations in the workload; hence, improving the performance and resource utilisation of the application.

2.7 Literature Review

2.7.1 Provisioning Evaluation

Abar et al. (2014) have successfully implemented a mechanism to automate the provisioning of virtual machines over on-premises servers in their study, which is similar to the provisioning part of the implemented system in this research. However, the study was using opensource cloud computing platform (Openstack) and OpenStack component called (Nova) that can integrate with different hypervisors for virtual machine provisioning. In another study, conducted by Khazaei et al (2016), the researchers proposed a model for performance analytics. The model was validated by conducting experiment where the provisioning performance of microservice was studied. In this study, a microservice platform was designed and developed over Amazon EC2 cloud with the use of Docker technology. However, in the current study the

experiment was conducted in two parts. In the first one, an engine was built that provisioned virtual machine on Hyper-V hypervisor, similar to the hypervisors used by Abar et al (2014). In addition to this, the second engine was built to provision container on Docker engine. However, in this research the VM and container engines are provisioning the services host with the required platforms, installed on top of it.

2.7.2 Deployment Evaluation

Bravetti et al. (2019) investigated the problem of automated microservices deployment in their research study and proposed an implemented solution. In order to deduce the applicability of their research, the researchers implemented an automatic deployment tool and computed a deployment plan for a genuine microservice architecture. This deployment plan was modelled in Abstract Behavioral Specification (ABS) language. ABS language is used for modelling of distributed, feature-rich, and object-oriented systems that are abstract but are also very precise. With ABS language, architectural concepts can be embedded as components and features of the systems can be connected to each other by their implementation in the form of product families (Hähnle, 2012). The solution is based on public cloud providers and containers orchestration system like Kubernetes. However, the deployment mechanisms used in this project are divided into two sections, i.e. web application deployment service and database deployment service.

The research of Wana et al (2018) has examined the application deployment problem and has proposed a framework to solve the microservices deployment resources allocation issue. Furthermore, the paper compares the use of VM and Docker containers as a microservices hosting approach. Moreover, the study also discusses different microservices placement approaches over the physical machine, which is more vital to both private and public cloud hosting providers. In the current research work, the microservices provisioning and deployment on both virtual machine and containers are examined and implemented to provide the system user with all the available options for services hosting. It is also expected to be a crucial feature in the case, the microservice platform is not available as a docker container image.

2.7.3 Auto-Scaling Evaluation

The research of Vaquero, Rodero-Merino, and Buyya (2011) discusses different scaling mechanisms (horizontal and vertical scaling) for both virtualization and containerization hosting approaches. This paper provides an overview of the researches conducted in the relevant subject by referring to various technologies and trends, followed in the market. Furthermore, the paper

also discusses the challenges associated with these technologies and approaches that may provide a better understanding of future researches and presenting a cloud system with ideal scalability. The study of Vaquero et al (2011) discussed scalability in implemented cloud providers like Amazon. However, the auto-scaling engine uses concepts like vertical scaling. Such vertical scaling is used in the current experimental study that represents scaling up and user-defined scaling rules to trigger the auto-scaling mechanism.

Similarly, Taherizadeh and Stankovski (2018) developed a dynamic auto-scaling model on the basis of infrastructure metrics. The development was carried out by monitoring CPU and memory utilization along with application-level monitoring. In this proposed method, seven (7) different scaling methods were compared in artificial and real-world workload scenarios. This action proved to be significant as the result of the implemented scaling method produced the best result, as compared to seven other existing auto-scaling methods. As the results of the study were found to be satisfactory, the researchers implemented it over the software engineering system called SWITCH for cloud applications that were sensitive to time. In the experiment, conducted for the current research, project infrastructure monitoring is only adapted to trigger the auto-scaling process as the scaling engine scales the service hosting resources vertically.

2.7.4 Literature Review Summary

In previous researches, there is a gap in implementing the microservices deployment for virtual machines, while providing a deployment platform and auto-scaling framework for on-premises hosting setup. To bridge this gap, the current study has attempted to present a deployment management system that suits the on-premise hosting setup. Previous researches have been extended by investigating and implementing deployment and auto-scaling management system for the microservices application.

3 Chapter 2: Experiment

3.1 System Implementation

The chapter characterizes the experiment that was done over the deployment management system. In this account, two different system settings have been discussed, i.e., the virtualization approach and the containerization approach across provisioning, deployment, and scaling.

3.2 Hardware Configuration

All system implementation and tests are performed on Dell PowerEdge R730 server. The hardware configuration of the server is as follows:

Host Server

- CPU
    - Model: Intel Xeon E5-2620 v4
    - Number Of sockets: 2
    - Speed: 2.10-3.0 GHz
    - Cores: 16
    - Threads: 32
- Memory Capacity: 256 GB DDR4
- Storage: 4 x 600 GB 10K RPM HDD

3.3 System Configuration

The deployment management system has a single-host server setup, the windows version used in the configuration is 2019 standard edition. Microsoft Hyper-V is configured on the host server to accommodate the system virtualization requirements.

Two virtual machines have been created for system provisioning services, including

1. DNS server to enable the creation, update and deletion of DNS records that map service IP to the service domain name.
2. Docker windows server to enable provisioning of Windows containers on top of windows virtual machine. The virtual machine processor supports nested virtualization as a docker for windows need to set up Hyper-V on top of the virtual machine.

3.4 System Supportive Virtual Machines

3.4.1 DNS Server

DNS server is used to manage application domain names, addresses, and support application scalability onward as application service can be scaled to a number of nodes.

3.4.2 DNS Zone

For the purpose of demonstration, DNS Zone was created under the name of DMS.com to host DNS records.

3.4.3 DNS A host Record

“A host record” is recorded in the DNS zone that maps the domain name to specific static IP, and DNS engine creates A host records under DMS.com DNS zone.

3.4.4 Docker Server

Docker version 19.03.0 has been installed using Microsoft windows PS Gallery, experimental features have been enabled and Docker API has been exposed using demon on port 2375 by adding hosts line to docker demon JSON file.

Docker (l2bridge) virtual network has been created to enable containers to connect directly to the network and to be accessible through the DMS environment network. The network enables containers to get IP from network DHCP dynamically or assign static IP to containers.

3.5 System Architecture

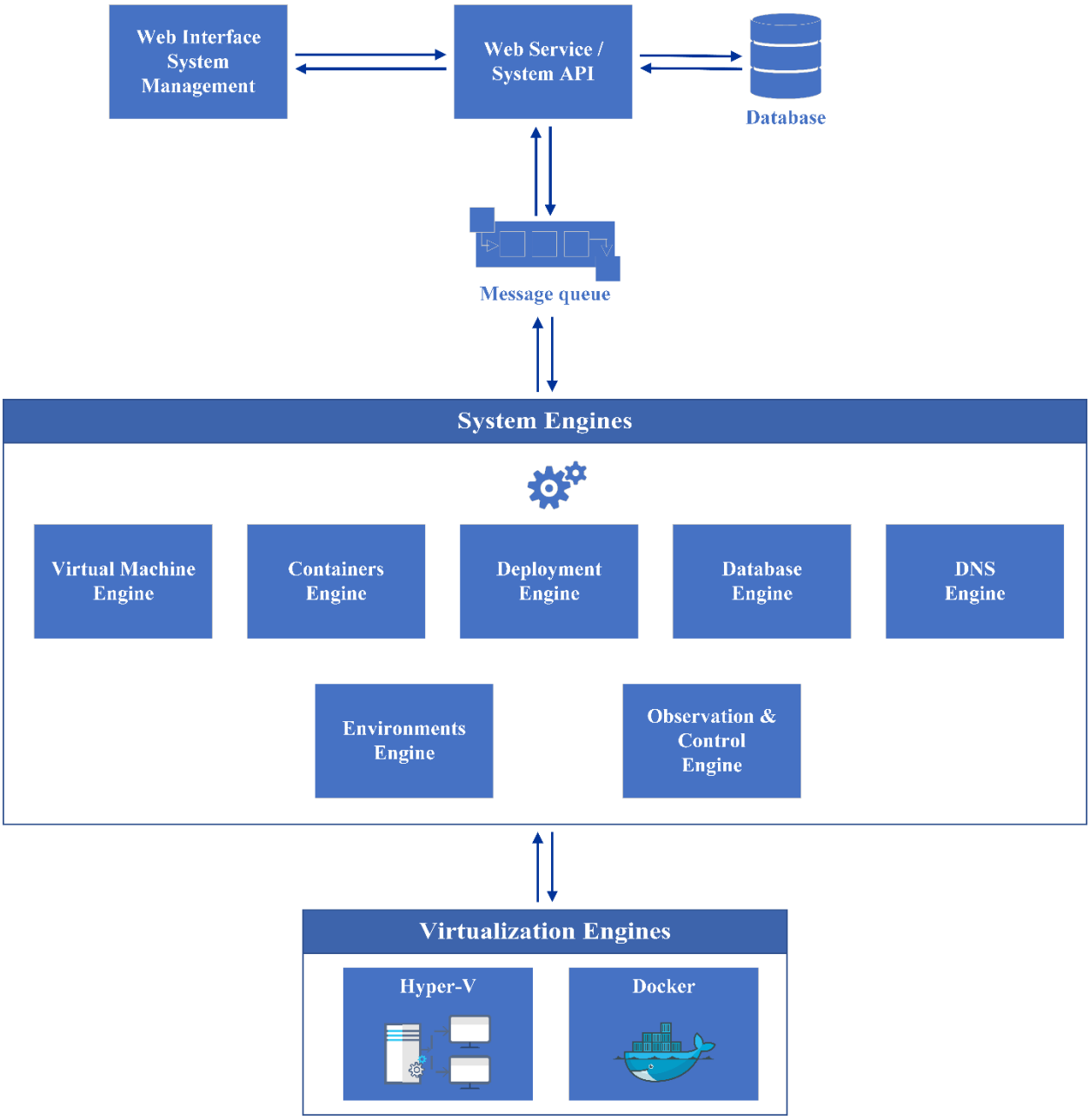

Figure 1: System Architecture

The system architecture is shown in Figure 1 consists of six main sections

1. Web interface / System Management
2. Web service – System API
3. Database
4. Message queue

5. System engines
6. Virtualization engines

All of these sections are discussed in the chapter.

3.6 System Web Interface

System web interface is used as the management interface of the deployment management system. The web interface provides easy-to-use, and user-friendly interface to help achieve the main project objective of optimizing the microservices management complexity.

3.6.1 Users Management

The User management page is used to create, update, and delete user access to the system.

3.6.2 Platforms

The Platform page is used to create, update, and delete platforms. These platforms provide flexibility in platform stack used in the system, so the system user can create platform dynamically and then link it to platform image.

3.6.3 Platform Images

The platform images page is used to create, update, and delete platform images for both virtual machines and containers. System users can select the platform, hosting type, virtualization image name, specify the version of the image, minimum VCPU, memory, and storage needed to run the hosting service base on that image.

Platform image is the image used in the provision of virtual machine or container based on a specific platform, like MS SQL image for a virtual machine, which is an MS SQL server based on windows server image ready to be provisioned as a microservice hosting.

3.6.4 Applications

The applications page is used to create, update and delete applications, the application definition provides a base for application environments to be created on top of it and application services to be deployed.

3.6.5 Application Services

Application services component provides a definition of platform images on the application. For example, a microservice application can have 3 web applications services and 2 database services. Application service provides a mapping between the application services hosting requirements and the platform image that is available for applications provisioning.

3.6.6 Application Environments

Application environment component provides management of different application environments, for example, testing, staging, production, etc. The page component offers creation of a new environment, cloning of application environment, and removal of the existing environment.

3.7 System Web Services (Web API)

System web service is the system middleware that provides linkage between system web interface, database, and messaging queue which communicates the execution messages to the system engines.

System API uses framework .Net Core 2.1. The API is created using the REST protocol and supports all HTTP methods (GET, POST, PUT, DELETE) for most of the system entities. API accepts JSON payload for any POST or PUT requests and return JSON messages for all HTTP methods.

3.7.1 Web Services Solution Design

Web service uses a repository design pattern and consists of four class library projects.

1. Entities and repository context class library, which holds entities model definition and entity framework database context.
2. Repository class library, which holds the repository base definition and repository wrapper.
3. Contracts class library, which holds the repository interfaces.
4. Logger class library, which holds logging methods definition.

The Web service is .Net core 2.1. Web application project that consists of 14 API controllers and application services queue management helper class.

3.7.2 Database Connection

Web services use entity framework core version 2.1.4 with 3rd party library for MYSQL Database connection (“Pomelo.EntityFrameworkCore.MySql”) version 2.2.0.

Lazy loading is disabled on entity framework database context, and relationships between tables can be accessed using ‘Include’ cluster to include related entities on the return result. Various database return functions are created on the repository base to be used on web services.

3.8 System Database

System database was created using MYSQL DBMS, to provide reliable storage for data to the deployment management system. The database contains ten tables: two system authentication tables, seven for handling applications deployments and scaling settings, and one for system performance test results.

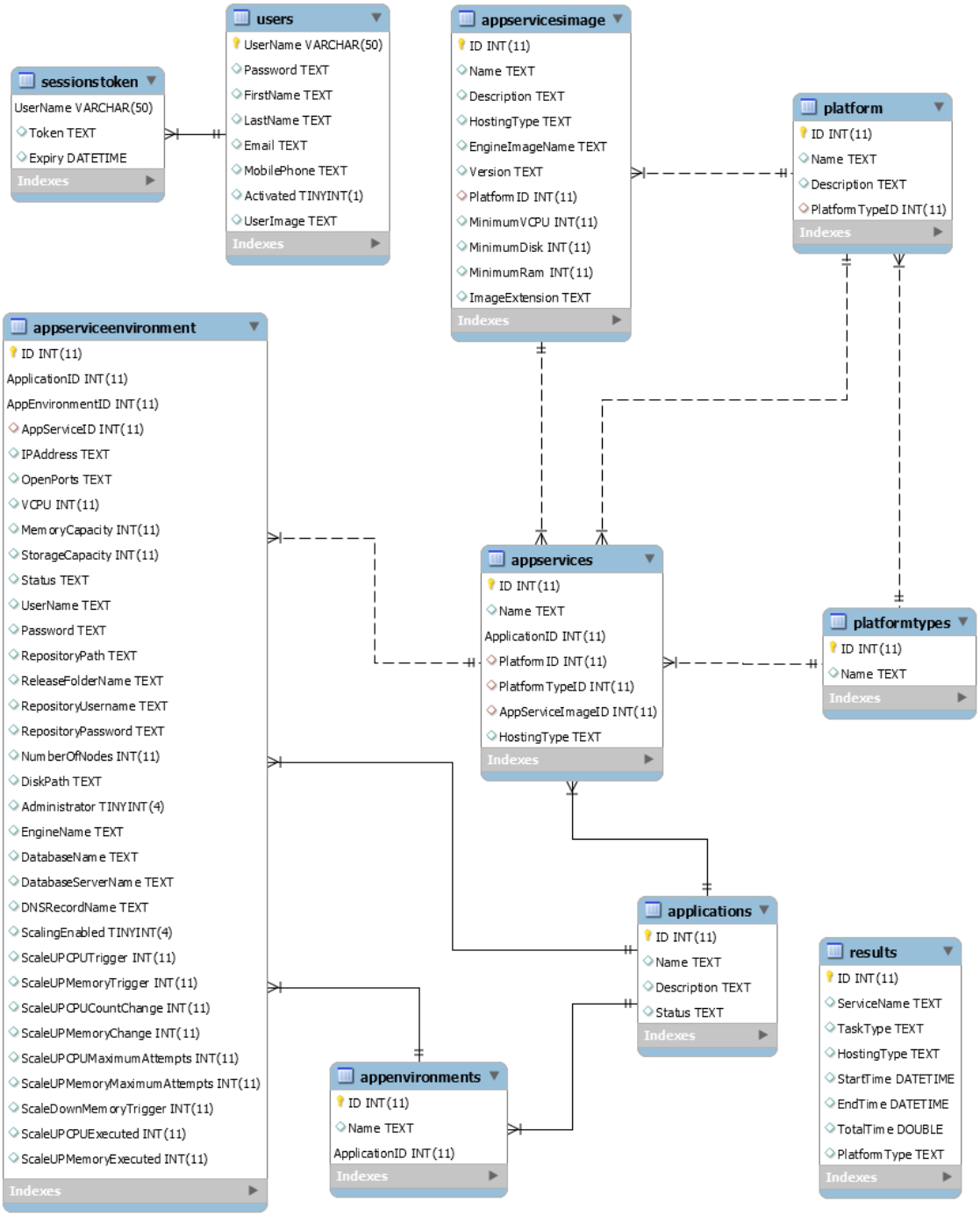

Figure 2: Database Scheme

3.9 Messaging Queue

Messaging queue is used as a system layer to enable asynchronous communication between the system API and system engines, so the web API can add more requests to the engine queue without waiting for the engine to finish its work in progress. It also removes the dependency between system components and reduces the errors that occur when one of the system engines go down or offline. Rabbit MQ is used as the messaging queue solution for this project.

Task model has been created to unify task execution objects that are passed between all system engines. Task model consists of two main attributes: task type and task body. A task body attribute is a dynamic object, holding all application service details, and each engine will parse the task body by the right model structure that fits the engine needs.

3.10 Service Deployment Workflow

Once users initiate a deployment request from the application interface, web service will send provisioning request to provisioning engines based on hosting type. For example, in the case of virtual machine web service will add the request to the virtual machine engine queue.

Once the provisioning engine receives the request, the engine creates the service hosting environment based on the service platform, then provisioning engine sends DNS creation request to DNS engine queue and sends another message to service deployment engines queue. In case, the service type was a database, the request will be sent to the database engine. However, in case the service is an application service the request will be sent to application deployment engine.

The deployment engines are responsible for deploying the service to the service host that is created by provisioning engines.

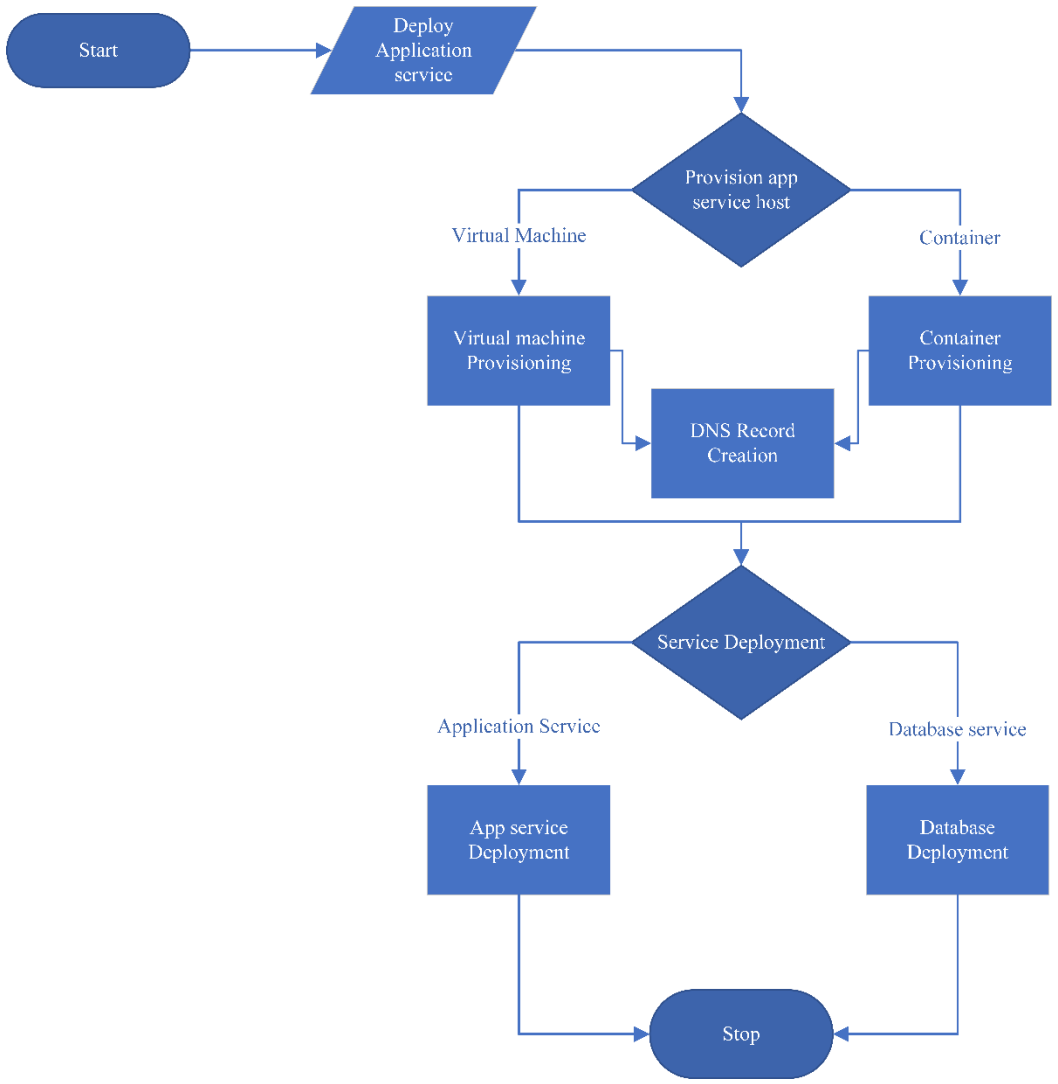

Figure 3: Application service deployment flowchart

3.11 Virtual Machine Engine

The engine is responsible for all virtual machines operations on the system, such as provisioning new virtual machines, updating virtual machine resources and removing virtual machines from the hypervisor.

In this project implementation, Microsoft Hyper-V is used as a virtualization engine, and the engine integrates with Microsoft Hyper-V hypervisor using PowerShell commands.

3.11.1 Virtual Machine Provisioning

For the provisioning of virtual machines, the engine executes the following actions:

- Deserialize the incoming queue message task body to application service object and select the required VM image from the app service object.
- Copy VM image from images stores location on the Hyper-V server to Hyper-V VM store directory.
- Create a VM using the PowerShell command.
- Get VM dynamic IP after machine startup.
- Change VM IP to static IP capture from user input in the Web interface.
- Create a local user on the VM.
- Rename VM to the unique service ID.
- Send service deployment queue message to deployment engine queue.

3.12 Containers Engine

The engine is responsible for provisioning and removing containers on the system. Docker is used as containers engine for this project, containers engine integrates with docker using docker API version 1.39. In some cases, the engine uses PowerShell commands to communicate with docker engine.

3.12.1 Containers Provisioning

For the provisioning of new containers, the engine executes the following actions:

- Deserialize the incoming queue message task body to the application service object, and select the right container image from the app service object.
- Send create docker API request to docker API.
- Send network to connect API request to connect the container to l2bridge network.
- Send start container API request.
- Send service deployment queue message to deployment engine queue.

3.13 Deployment Engine

3.13.1 Application Service Deployment

As application service can be application release that needs to be deployed to a web server or it can be a database needs to be deployed to SQL server instance. The approach is to identify the service hosting (Virtual machine, Container), and then to identify the application service platform type (Application, Database). The Deployment API controller will then build a queue

message to be sent to the deployment engine or database engine based on the type of application service engine and platform.

3.13.2 Application Service

In the case of web application service, there are two types of deployment that are based on the type of hosting of the application service.

3.13.3 Application Deployment on Virtual Machine

In the case of virtual machine hosting, the engine starts by cloning the application release to the local storage of the deployment engine server. The repository cloning operation depends on the information captured from users in application service deployment screen, such as repository path, repository credentials, and release folder name.

In this project implementation, only GIT repositories are supported. First of all, GIT client is used towards retrieving application files from the repository using the captured information.

Secondly, the engine will start copying the application release files to the virtual machine IIS directory “Inetpub”. Application ID, application service ID, and application service environment ID is concatenated to form a new directory name.

After that IIS application pool will be created using server administrator credentials as a pool user identity and finally the new web application is created under the default web site with the application service ID with public access to any user to browse the application and finally the engine will set the application service status to up for successful deployment.

3.13.4 Application Deployment on Container

On the other hand, in case of a web application to containers deployment, the engine starts by cloning the application files to the local disk of the deployment engine server. In the same way, it is used on virtual machine cloning operation which uses the information captured from the users to retrieve application files and saves it to local storage.

Next, the deployment engine stops the container to be able to copy application files to the container storage volume. Afterwards, the application files are copied to the container IIS directory “Inetpub” and after that, the engine starts up the container again and the engine will set the application service status to up for successful deployment.

3.14 Database Engine

One important part of any microservices application is the database services where the application stores its persistent data. In a microservice architecture, the application may have one or more database services to achieve the decoupling of services.

Database engine provides database deployment. The engine supports two main deployment functions: database deployment on virtual machine and database deployment on containers.

The sample implementation used on the database engine is to deploy MS SQL database on MS SQL developer edition.

3.14.1 Database Deployment on Virtual Machine

In case of database deployment to virtual machine, the engine will start by setting the application status to ‘deploying’; secondly the engine will copy the database from local upload location where the web service saves the database backup file (.bak) to database virtual machine local location using network impersonation; thirdly the engine will execute (Restore-SQLDatabase) PowerShell command with the information captured from user interface like Database name, which the database will restore on the new server and finally the engine will set the application service status to up for successful deployment.

3.14.2 Database Deployment on Container

In the case of database deployment to the container, the engine will start by updating the application status to ‘deploying’. Secondly, the engine will stop the provisioned container to copy database backup file from local upload location where the web service saves the database backup file (.bak) to the provisioned container local location using docker ‘CP’ command. Thirdly the engine will execute (Restore-SQLDatabase) PowerShell command with the information captured from user interface like Database name, which the database will restore on the new server and finally the service will set the application service status to up for successful deployment.

3.15 DNS Engine

DNS engine is responsible for creating, updating, and removing DNS records on the system DNS server. DNS provides application service domain name which offers flexibility in case the application service hosting requires alterations. These include changing the virtual machine or container of the application service or even the IP address of the virtual machine.

Engines integrate with Microsoft DNS server using “System.Management” library to create, update and delete DNS A records from DMS.com configured DNS zone.

3.16 Environment Engine

Environments engines provides management operations for application environments like creating new application environment, clone environment to new environment, and delete environment.

The cloning feature provides cloning of application environments through one-click to deploy to a new environment. For example, the application has only testing environment and the user needs to create a new staging environment with the same deployed services in the testing environment. The user will just specify the new environment's name, new IP range and click deploy, and environment's engine will create the environment's replica without any user interaction.

3.16.1 Environment Cloning

The engine iterates through the application environment services on the cloned environment, and on each iteration, the engine will check the service hosting type and based on the hosting type the engine will create a new clone of the running application service. Application service hosting will have the new environment ID on it. After provisioning of each service, the engine will create a database record for each cloned service.

3.17 Observation and Control Engine

Engine, responsible for monitoring all deployed application services, checks for any performance degradation caused by the lake of computing resources.

The engine uses one type of scaling mechanisms, i.e. scaling up by adding more computing resources (CPU, Memory) to the service host.

3.17.1 Virtualization Scaling (Hyper-V)

Virtualization engines (hypervisors) usually support hot resources scaling, which means that we can upgrade virtual machine resources while the machine is running. This method provides zero downtime scalability option. On the other hand, Hyper-V only supports upgrading memory and network while the virtual machine is running but does not support modifying CPU count while the virtual machine is running. This limitation means that if we need to scale the virtual machine processor, we will shut down the virtual machine to be able to scale the virtual processor count then power it up again.

The engine supports 3 types of VM scaling:

- Scaling up VM memory: which preformed through 2 main steps getting the current VM memory size to add it on the scaling up memory capacity (current memory capacity Count + Scale-up memory capacity) and then set the running VM memory with the new required capacity.
- Scaling up VM CPU: the engine will get the current VM CPU count then shut down the VM, updating the VM CPU count with (current CPU Count + Scale-up CPU count), and finally starting up the VM again.
- Scaling down VM memory: once the VM memory utilization reaches low percentage specified by system user then the engine will scale down the VM memory though 2 main steps getting the current VM memory capacity then subtract the scale-up capacity from it and update the VM memory capacity with the subtraction result.

3.17.2 Container Scaling (Docker)

Docker for Windows does not support scaling up resources, even if the container is shutdown.

The scaling-up approach for docker is to create a new container replica with the new resources required for the scaling up.

There will be no downtime for the scaling-up process as the container will be replicated then the IP will be changed on the newly created container to the main service IP, once the container is up and ready to serve incoming requests.

3.17.3 Database Services Scaling

As most of the microservice applications are data-driven, they usually require robust data operation, which makes database scaling as a challenging topic.

Scalability mechanisms for database services can be categorized into three types. It includes the utilization of a NoSQL database, distributed memory caching, and database clustering (Vaquero, Rodero-Merino, & Buyya, 2011). However, only relational database scaling is covered in the research, which does not offer support for scale-out operations that require creating new instances (Das, Li, Narasayya, & König, 2016). It is due to the fact that the data operations are committed across all the nodes of database hosting in relational database management systems (DBMS).

Lastly, the autoscaling on database node resources in the research will only be done by increasing the computing resources of the hosting environment in case of demand on the database service.

3.18 Implementation Issues

1. Docker for Windows does not support resources update which means the system cannot update container CPU count or memory capacity for the deployed containers.
2. Windows containers do not support file copying while the container is running. This will cause small downtime for services deployment.
3. Hyper-V does not support Hot scaling of CPU Count.

3.18.1 Overcoming Implementation Issues

1. To overcome Docker for windows scaling limitation, a new replica of the container is created from the container that is needed to be scaled up with the new required computing resources and then change the new container IP to the original service IP and shut down the old container.
2. On deploying new application release to the container, the deployment engine will shut down the container for 10 seconds to copy the new code release and starts up the container again.
3. Scaling VM CPU up will require to shutdown VM for 5 seconds to change the CPU count and start up the VM again.

4 Chapter 3: Results

This chapter contains the characterization of the experiment done over the deployment management system. For this purpose, two different types of microservices hosting approaches are discussed, i.e., the virtualization approach and the containerization approach.

4.1 Research methodology

Quantitative research methodology has been utilized for this study and all the data is collected from the system to measure the different aspects of system implementation successfulness, system performance and comparison between the two virtualization approaches. As the research requires the adoption of formal instruments and structured procedures for the purpose of data collection, hence quantitative research methodology is the most suitable methodology for this study (Queirós, Faria& Almeida, 2017). The data is collected accurately and methodically, which allows a thorough analysis of the collected information to produce results.

In order to achieve the research objective i.e. to solve the complexity of deploying microservices applications, it has been measured how successfully the application has dealt with the complexity part of the microservices deployment.

The deployment management system has provided two ways of deploying microservices-based applications:

- Deploying microservice on a virtual machine.
- Deploying microservice on a container.

The results of both methods of hosting with respect to provisioning time and deployment time are presented in the report, and then the utilization of the resources of system components are shown along with whether system components have achieved the required objectives, for all system testing results (See Appendix A).

4.2 Virtual Machine Provisioning

This section contains the calculated time taken by the virtual machine engine to preform provisioning of a new service host.

Two virtual machine images have been configured and both images have the same OS (Windows Server 2019 standard edition).

Virtual machine images are configured as follows:

- Internet Information Services 10 is used in Application Image.
- MS SQL Server 2017 Developer edition is used in Database Image.

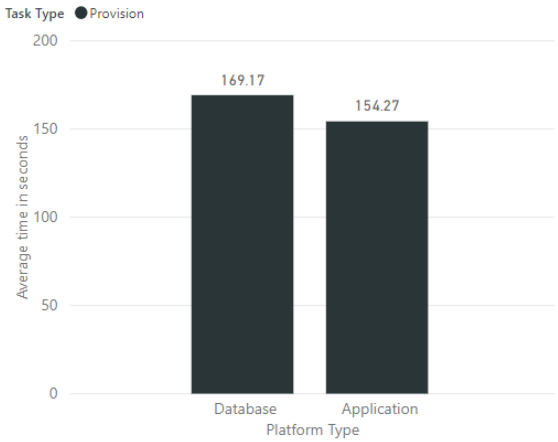

Figure 4: Average time to provision virtual machine by seconds

Figure 4 shows the average time taken by the virtual machine engine to provision two types of virtual machine platforms. In case of application server VM provisioning, the average time taken was 154.27 seconds. On the other hand, the average time taken for database server VM provisioning was 169.17 seconds.

As can be seen from the results MS SQL server VM takes more time for provisioning as compared to application VM, hence it is concluded that the time difference between provisioning application and database returns because of the size of database image, which is nearly 1.5 the size of the application server image.

4.3 Virtual Machine Deployment

In this section, the time taken by the virtual machine engine to deploy application release or database to the service host has been calculated.

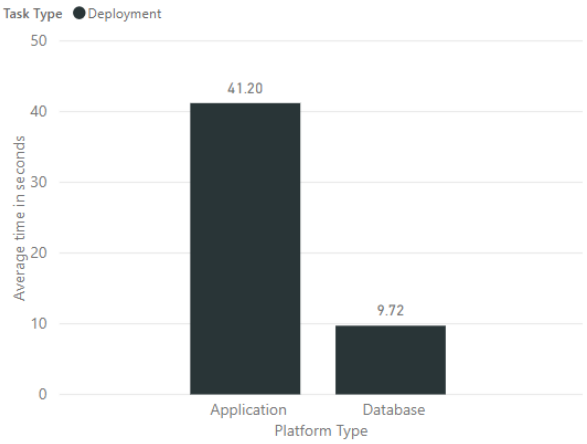

Figure 5: Average time to deploy application service to a virtual machine by seconds

Figure 5 shows the average time taken by deployment and database engines to deploy application service to the application service host. In the case of web application service deployment, the average time taken was 41.20 seconds. Where in contrast, the average time taken to deploy database service was 9.72 seconds.

Application deployment engine takes more time as the engine first clones the GIT repository of the application then copies the release folder content to the VM IIS folder. Whereas, the database engine only copies the database backup file to the database VM and initiates database restore request.

4.4 Containers provisioning

Here, the time taken by container engines for the provisioning of a new service host has been calculated.

Two docker container images have been used on the system, that was pulled from Docker Hub repository. The container images are as follows:

- Docker Image used by Application Image “mcr.microsoft.com/dotnet/framework/aspnet”.
- Docker Image used by Database Image “Microsoft/MySQL-server-windows-developer”.

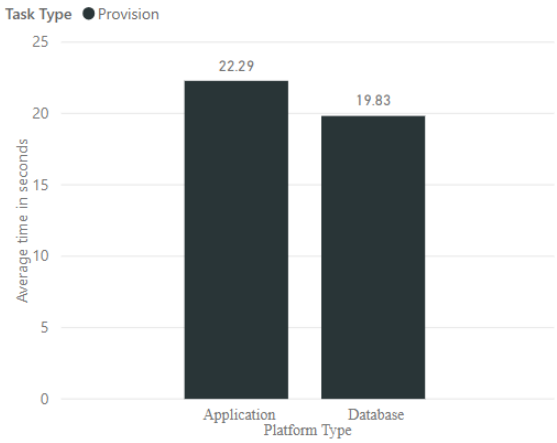

Figure 6: Average time to provision application service container by seconds

In figure 6, the average time taken by the container engine for the provisioning of two types of containers platforms images can be seen. For application container, the average time taken for provisioning was 22.29 seconds, whereas, the average time taken for database container provisioning was 19.83 seconds.

4.5 Containers deployment

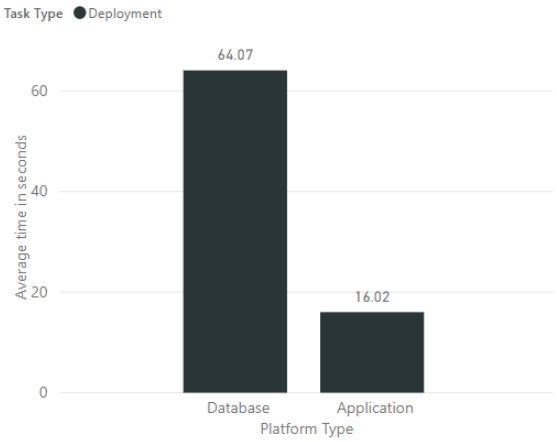

Figure 7: Average time to deploy application service to a container by seconds

Figure 7 shows that the average time taken by the deployment engine to deploy application service to the container was 16.02 seconds. In comparison, the average time taken by the database engine to deploy a database to the container was 66.07 seconds.

Furthermore, in this figure, a significant difference between the deployment time taken to deploy the database and to deploy the web application can be seen. After successfully executing the database deployment process, the database engine has to provision the container and then shut it down. Subsequently, the database engine has to copy the database backup file, restart the container and after the container is running smoothly and the status of the database container is confirmed to be healthy, the restore database command is initiated. While on the other hand the application deployment engine simply shuts down the container to copy the release files to the container and then restarts the container after which the deployment of the application service is ready.

4.6 Comparison between two virtualization approaches

4.6.1 Provisioning Scenario

During the test of provision service, every service provision time is captured by time metrics. This section contains the comparison of the performance of the provisioning of two virtualization engines.

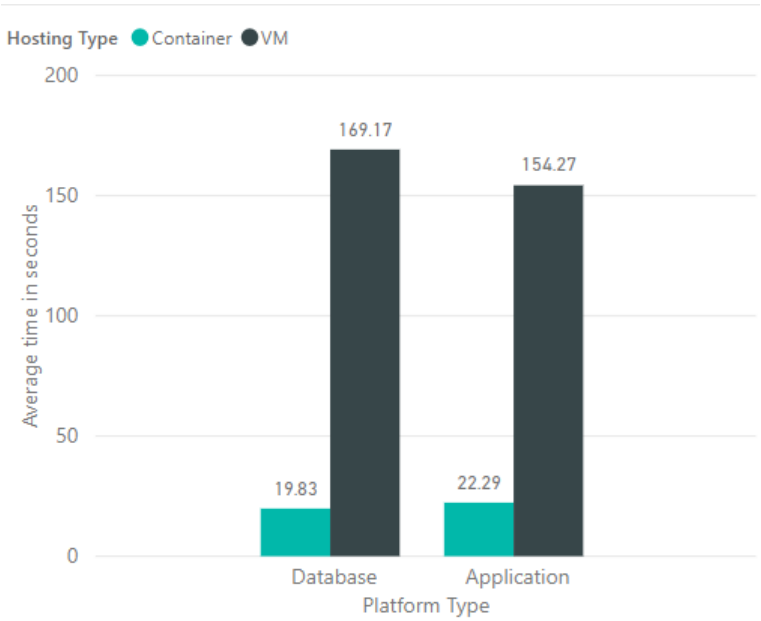

Figure 8: Average time to provision application service host by seconds

Figure 8 shows the average time taken by virtual machine engine to provision database service instance, which is 169.17 seconds on average whereas, the time taken for the provisioning

of an application service instance is 154.27 seconds. In comparison, the time taken to provision database service instance on container engine is 19.83 seconds and 22.29 seconds is the time taken to provide application service instance on the container.

However, the result is expected as the containers share the host OS, and hence the container is just isolating part of the host resources and importing the container image platform. While in contrast, provisioning virtual machine means configuring virtual machine OS, drivers, network and platform which takes more time to provision.

4.6.2 Deployment Scenario

Here, the study has compared the deployment of application services and database on the two virtualization engines to measure the deployment performance.

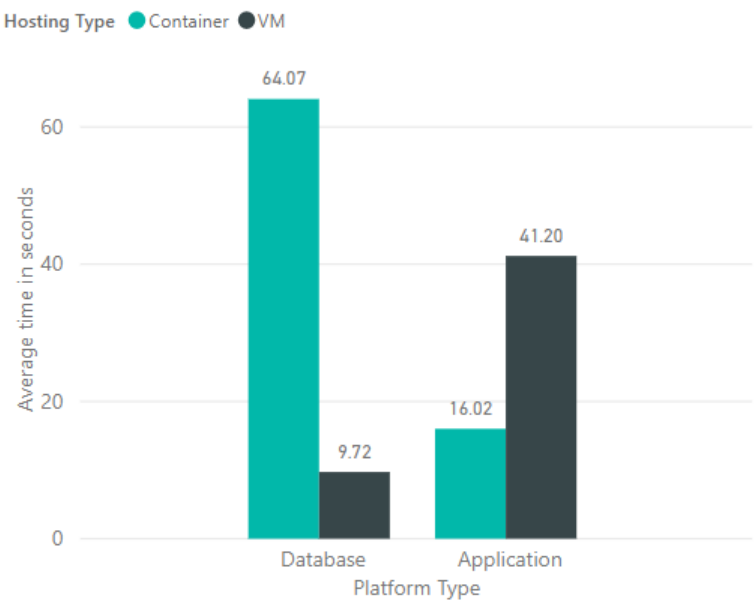

Figure 9: Average time to deploy application service by seconds

In figure 9, it can be seen that the average time taken by the database engine to deploy the database to the container is 64.07 seconds, while the time taken to deploy database service to VM is 9.72 seconds. Whereas in comparison, it took 16.02 seconds to deploy application service to container and 41.20 seconds to deploy application service to VM.

4.7 Environment Cloning

This section contains the calculation of the time taken by environment engine to perform cloning of existing environment. This process involves cloning virtual machines, and containers with the deployed code and in this test, the cloned environment consists of 4 services, 2 virtual machines, and 2 containers. The test was executed 6 times through which the average time to clone environment of 4 - 6 services was found to be 267.14 seconds.

4.8 Scaling Operations Performance

This section demonstrates the results of services instance auto-scaling test. The test has been conducted over both virtual machines and containers using stress testing tool to initiate the auto-scaling service on the deployment management system.

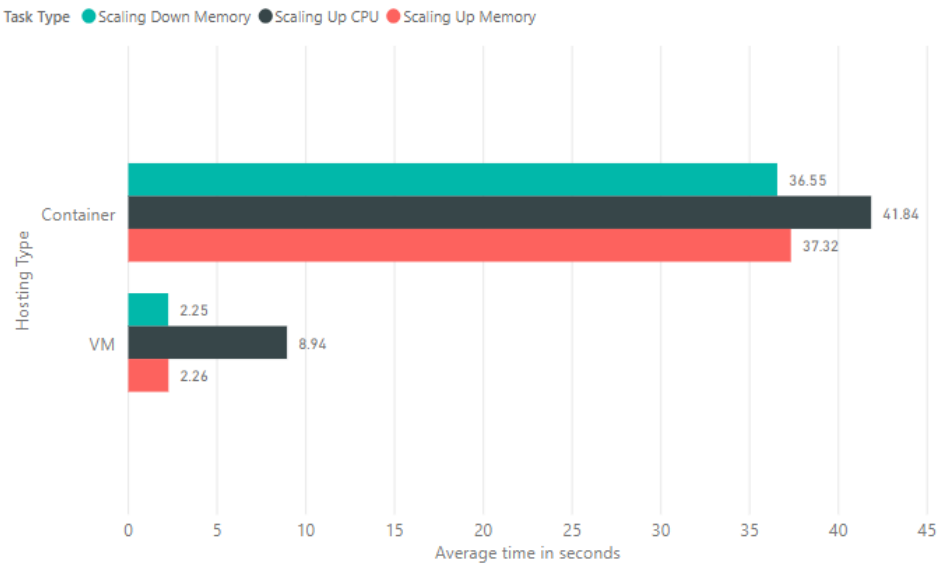

Figure 10: Average time to scale service instance for both containers and virtual machines

Figure 10 shows the average time taken to auto-scale service instance by seconds. Virtual machine memory scaling takes 2.26 seconds to scale the memory up and 2.25 seconds to scale the VM memory down if memory utilization is less than the percentage specified by the system user. As scaling up the VM CPU count requires VM to shutdown then modify the CPU count and start up the machine again, the average time taken to scale the VM CPU is 8.94 seconds.

However, containers scaling the average time is much higher compared to VM scaling numbers. This is because of the limitation of Docker for windows as it does not support the modification of container resources. Hence the implemented solution was to clone the running container that required scaling and then shut down the old container and start the new container with the new scaled resources using the same IP.

The scaling up operation for container memory takes the average time of 37.32 seconds while scaling down memory takes 36.55 seconds and scaling CPU takes 41.84 seconds.

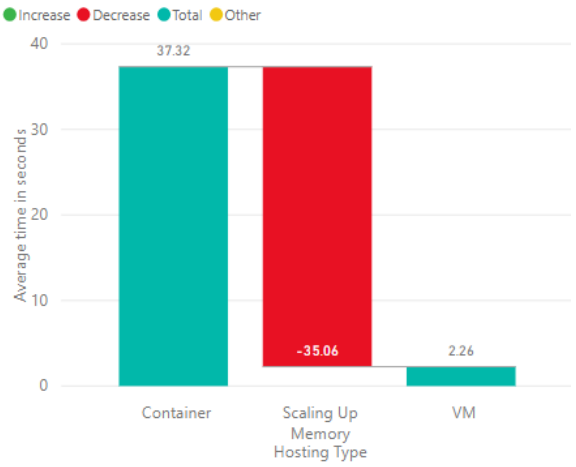

Figure 11: Difference between scaling up memory time for containers against VM

Figure 11 shows the difference between scaling up memory for VM and Container in seconds. The time difference between the memory scaling of container and VM was found to be 35.06 seconds. Observation and control engine can scale VM memory in 2.26 seconds which is more beneficial for microservice operations that require fast auto-scaling to VM resources.

4.9 Deployment management system operations

In figure 12, system operations performance has been summarized. Regardless of hosting types of services, cloning operation of the environment takes an average of 267.14 seconds, while provisioning operations take an average time of 88.92 seconds. Whereas the average time taken in the deployment of services is 36.07 seconds, scaling of services CPU takes 12.81 seconds, scaling up memory takes 5.77 seconds on average, and scaling memory down takes 5.37 seconds on average.

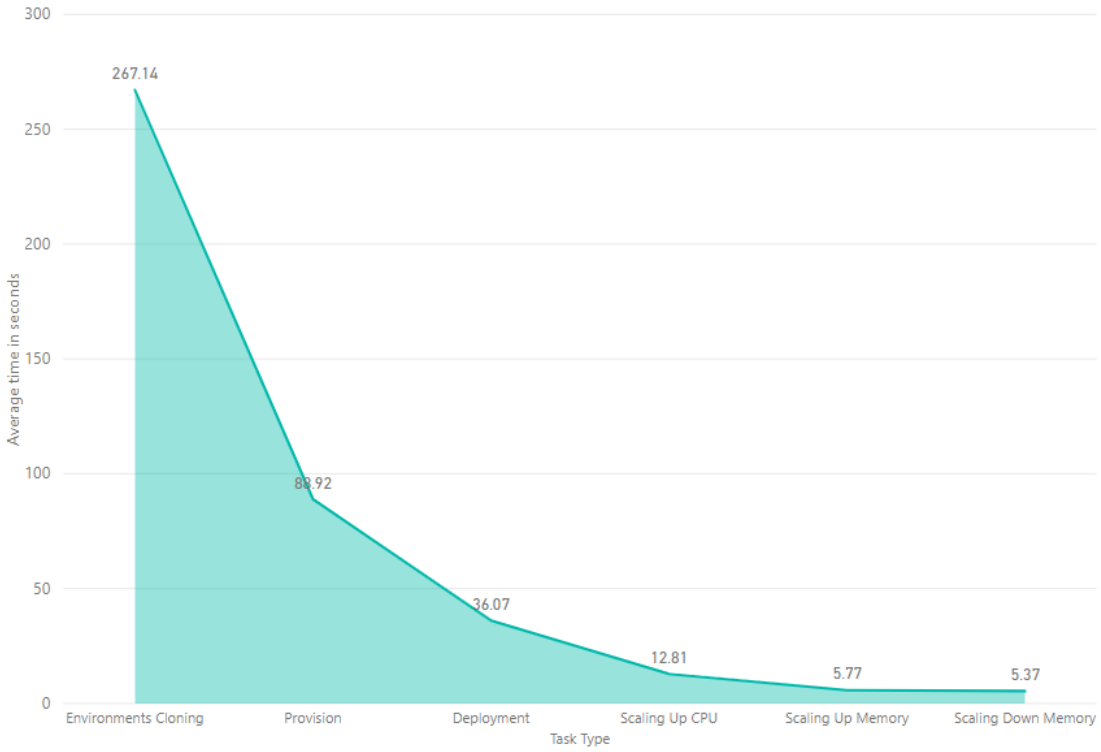

Figure 12: Deployment Management System operations average time

4.10 System Components

Resources utilization test has been conducted on a client machine to measure the resources utilization of each system component.

The machine on which the resources performance test has been carried out on consists of the following specifications:

- CPU: Intel Core i7-8650U
  - Cores: 4 Cores
  - Base speed: 2.11 - 3.93 GHz
- Memory: 16 GB of RAM DDR4

The following system components have been tested for 1 minute to measure the utilization of their resources.

4.10.1 Web Service

Figure 13 shows that the average CPU utilization was 25% with 100% maximum utilization of all machine processors, while a maximum of 338 MB of memory was utilized and an average of 198 MB of memory was utilized.

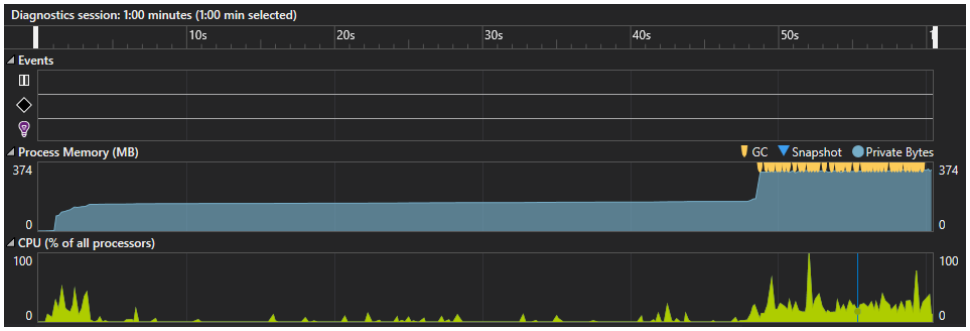

Figure 13: Web services resources utilization

4.10.2 Virtual Machine Engine

Figure 14 shows that the average CPU utilization was 2% with 14% maximum utilization of all machine processors. Maximum of 27 MB of memory was utilized and an average of 19 MB of memory was utilized.

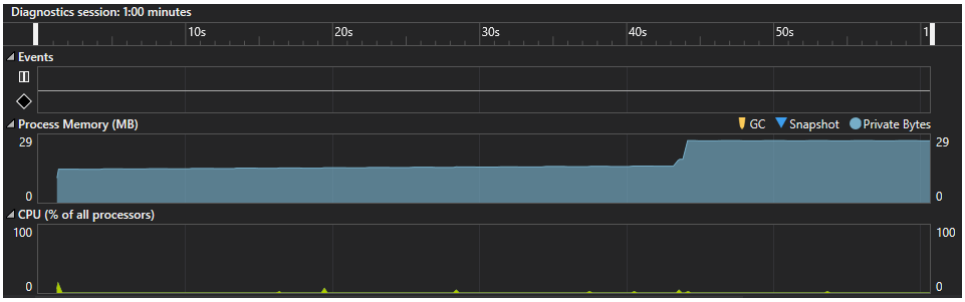

Figure 14: Virtual machine engine resources utilization

4.10.3 Containers Engine

Figure 15 shows that the average CPU utilization was 1% with 13% maximum utilization of all machine processors. Maximum of 39 MB of memory was utilized and an average of 27 MB of memory was utilized.

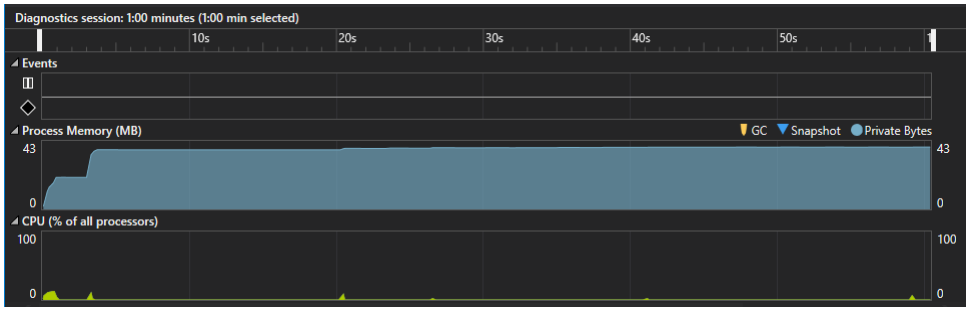

Figure 15: Container engines resources utilization

4.10.4 Deployment Engine

Figure 16 shows that the average CPU utilization was 4% with 13% maximum utilization of all machine processors. Maximum of 41.5 MB of memory was utilized and an average of 30 MB of memory was utilized.

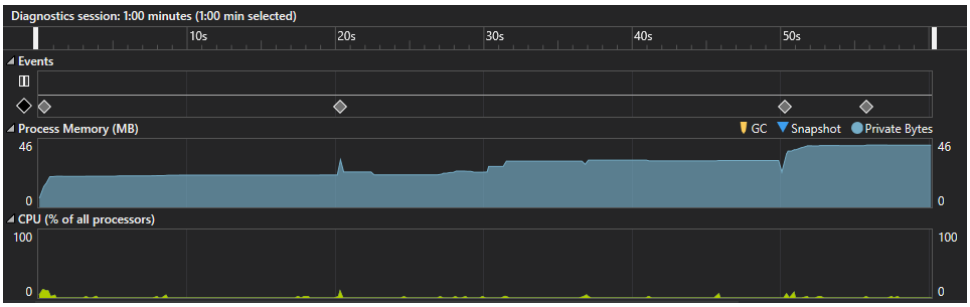

Figure 16: Deployment engine resources utilization

4.10.5 Database Engine

Figure 17 shows that the average CPU utilization was 2% with 13% maximum utilization of all machine processors. Maximum of 39 MB of memory was utilized and an average of 26 MB of memory was utilized.

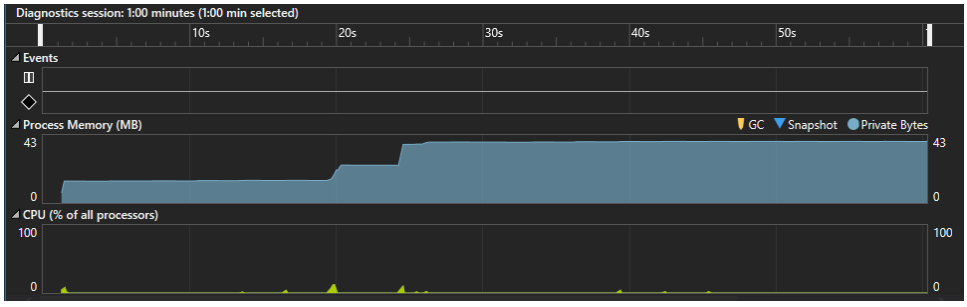

Figure 17: Database engine resources utilization

4.10.6 Environments Engine

Figure 18 shows that the average CPU utilization was 1% with 12% maximum utilization of all machine processors. Maximum of 36 MB of memory was utilized and an average of 30 MB of memory was utilized.

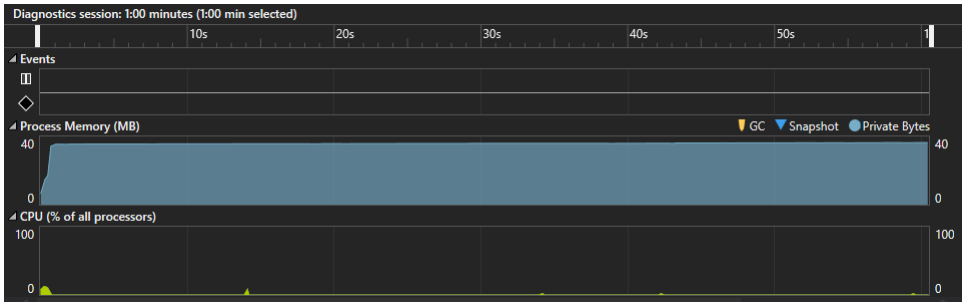

Figure 18: Environments engine resources utilization

4.10.7 Observation and Control Engine

Figure 19 shows that the average CPU utilization was 2% with 13% maximum utilization of all machine processors. Maximum of 67 MB of memory was utilized and an average of 49 MB of memory was utilized.

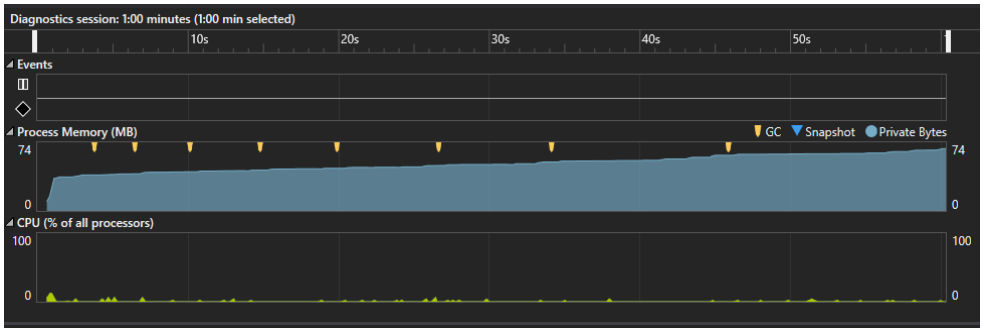

Figure 19: Observation and control engine resources utilization

5 Chapter 4: Discussion

In this experimental study, a system is developed through which the researcher has tried to simplify the complexity of microservice-based applications while monitoring the resources and performance of applications and automating the scaling of services according – as per the user requirement. Though various studies have been conducted to reduce the complexity of the microservices-based application, they have mostly been for the cloud platform (Bravetti et al., 2019; Abar et al., 2014; Vaquero, Rodero-Merino and Buyya, 2011). In particular, barely any research has been conducted to reduce the complexity of on-premises deployment of microservices applications. For this purpose, the present research has conducted an experiment with two different approaches, i.e. virtualization approach and containerization approach. This chapter discusses the results of the experimental study and relates the findings with the relevant literature to draw coherent conclusions.

5.1 Using Virtualization as Microservice applications hosting platform

In the virtualization approach that was followed in the experiment, two virtual images are configured containing the same OS i.e., Windows Server 2019 standard edition. One as an application image and another as a database image. The results show that the database image takes more time to provision than the application image; hence, it is deduced that the size of the database image is larger than the size of the application image. Previous studies conducted on the similar subject also support the results by concluding that the type and size of VM image affect the response time or provision latency of the system (Abar et al., 2014). Moreover, experimental results of virtual machine deployment show that the application engine, in deployment, takes more time than a database engine.

System performance and cloud performance can be analyzed by comparing the current experimental study with the study conducted by Mao and Humphery (2012) on the VM provisioning performance over the public cloud environment.

Cloud	OS	Average VM startup time
Amazon EC2	Windows	810.2 seconds
Microsoft Azure	Windows	356.6 seconds
Rackspace	Windows	429.2 seconds

By comparing the public cloud provisioning time with the system provisioning time of the current study, it was found that virtual machine provisioning time taken by Amazon was 810.2 seconds, while Azure takes 356.6 seconds, and Rackspace takes 429.2 seconds. However, time taken by the system in the current study was 160.79 seconds on average to provision windows virtual machines, which is less than half of the least public cloud provider Microsoft azure.

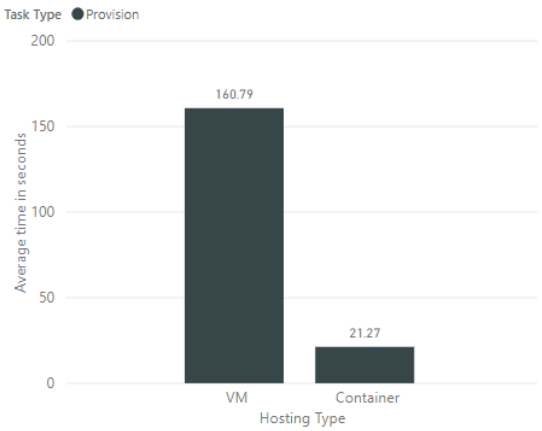

Figure 20: Average time to provision VM and container

5.2 Using Containerization as Microservice applications hosting platform

In the containerization approach, kernel utilities of an operating system are collected and configured to effectively manage the hardware resources for a specific application section. In a cloud environment, containerization allows the service providers to proficiently manage the hardware resources by offering a more flexible way for initiating, reassigning, and optimizing them according to the application requirement (Xavier et al., 2013). As the containers occupy the same operating system, they incur lower overhead as compared to VM approach and hence provide improved deployment time for the applications (Fazio et al., 2016). While this has been proven for cloud-based environments, this has experimented the case in a physical server-based environment for microservices.

In the containerization approach of the experiment, two container docker images have been configured that were downloaded from the Docker Hub repository, one as an application image and another as database image. The results of the experiment show that the provisioning time was higher in the application container as compared to the database container. Whereas in deployment time of application container image was quite lower than the database container image.

5.2.1 Comparison between Containerization and Virtualization Approaches

Containerization and virtualization have been compared in various studies in order to deduce which approach is more suitable for microservices application hosting and provisioning (Fazio et al., 2016; Amaral et al., 2015; Khazaei et al., 2016; O'Connor, Elger and Clarke, 2017; Mavridis and Karatza, 2017; Hasselbring and Steinacker, 2017). Most of these studies have been conducted in cloud-based environments and found that the best approach for proficient development and deployment of microservice applications is container-based virtualization. Container-based virtualization is the combination of containerization and virtualization where the weaknesses found in both approaches are eliminated by the strengths of each other. This way, a formidable system can be designed with which microservice applications can be deployed efficiently and with reduced complexities (Fazio et al., 2016; Mavridis and Karatza, 2017). In this study, both approaches have been tested over the physical server and their results are compared in order to form an analysis.

When the provisioning time of the two virtual machines are compared in the results, provisioning time of application image and database image of the container were quite lower than that of the application image and database image of VM. The results are supported by Seo et al. (2014) who states that as containers do not include implementation of Guest OS, it can be implemented in a short time and use the resources economically. However, in the scenario of deployment, the results show that the time taken for the deployment of the container database image was higher than the deployment of the VM database image. Whereas, the deployment time of the container application image was lower than the deployment time of the VM application image.

The current study is conducted to compare the performance of container-based and virtualization-based services and it was found that VM-based services perform better than the container-based services, specifically in response-time. In particular, VM-based services show more than 125% efficiency in response time, as compared to container-based services (Salah et al., 2017). However, in another study conducted by Amaral et al (2015), it was found that container-based services are also gaining popularity because of their lightweight virtualization capabilities and their suitability when dealing with microservices-based applications where they have superior performance and improved provisioning and deployment time. Therefore, it can be affirmed that

containerization is the better option for the provisioning and deployment of microservices-based applications as it shows better provisioning and deployment time, whereas VM is more suitable to microservices applications that require frequent auto-scaling.

5.3 Automation of Application Scaling on Demand

In automating the scaling part of the experiment, the test was conducted over virtual machines and containers with the help of stress testing tools with which the auto-scaling service was initiated over the deployment management system. In stress testing, the system sent resource requests till it runs out of resources and has to auto-scale in order to meet the user requirements or face requests overload and a failure of the whole system (Joy, 2015). According to the current experimental study, the time taken for the scaling of containers was found to be much higher than the time taken for the scaling of virtual machines. Joy (2015) contradicts these results by concluding that while the virtual machine took 3 minutes to scale in their study, containers only took 8 seconds to scale up and handle the request. The main reason for this contradiction can be attributed to the fact that Joy (2015) used Linux based Docker containers running on top of Ubuntu 14.04 server whereas, in the current study, Windows-based containers are used. Another study was conducted to compare the performance of Docker containers on Windows and Linux and it was found that Windows provides a limitation for Docker engine as a container host because it does not allow modifications in the resources of the containers (Pfaller and Hartley, 2018). Hence, it can be deduced that the time taken by containers for scaling is dependent on the environment it is being run on and can be more or less than the time taken for the scaling of virtual machines.

By using high-level metrics and altering the thresholds for autoscaling dynamically, a system can be designed that provides a more efficient reaction to the fluctuations found in the workload and respond swiftly and effectively (Taherizadeh and Stankovski, 2019). This way the accuracy of the auto-scaling of the whole system can be increased. Automatic scaling of resources is considered to be an important aspect that needs to be implemented in the system to ensure its effectiveness (Abar et al., 2014). With the help of autoscaling, the microservices applications hosted on the system will receive automatic scaling of the computing resources as the system is aware of its existing conditions and will be able to gauge its possible requirements and adjusting the backup, accordingly. It is imperative to develop a system that provides a quick response for fluctuating workloads in this fast-paced era.

5.4 Meeting Developer's needs by the Deployment Management System

In a survey conducted by Zhang, Vasilescu, Wang and Filkovm (2018), when developers were asked about their motivation behind the need for continuous deployment solution like the deployment management system 42.9% of the developers, answered that it helps them in deploying systems automatically instead of doing it manually. However, 19.3% of the developers answered that continuous deployment solution enables easier and smoother deployment processes. In another question, related to the developer's needs that were not addressed, 13.3% selected better multi-platform build support, which has been addressed in this project on dynamic platform support. System users can create and map platform image on the system and then it can be used at any time.

When developers were asked about unmet needs in the continuous deployment solution used by them, 16.9% of the survey participants selected the options of 'easier to learn and configure' as an unmet need. This need has been met with the implementation of a user-friendly interface and flat application structure on the deployment management system.

Another question was about what their continuous deployment solution consists of and in response to it, 25% of the participants answered that they use custom scripts to automate their application deployment. This was one of the project considerations to solve the complexity of deployment using custom scripts.

On another survey conducted by Knoche and Hasselbring (2019), it was deduced that deployment complexity was one of the main reasons for the lack of adoption of microservices in different industries. In the results of another survey, resource provisioning was the fourth most important implementation challenge faced by the participants, working in different industries. The deployment management system has successfully tackled the resources provisioning issue and deployment complexity that different industries faced to adopt microservices applications.

5.5 Summary

In this chapter, the results of the experiment have been discussed and compared with other studies. The main objectives of the research were to reduce the complexity of microservice-based applications and to introduce automation of scaling of applications, according to the demand present for the resources of the applications. In order to reduce the complexity of the microservice applications, two approaches were used in this experimental study, i.e. containerization and

virtualization. In virtualization, the results showed that the database image took a long time in provisioning as compared to the application image. However, the application image took longer time than database image when it came to deployment. In containerization, the time taken by application container was longer than database container in provisioning, whereas the database image of container took longer time in deployment than the application image. When the results of both approaches were compared with each other, it is found that in provisioning, the overall time taken in containerization is much improved as compared to the time taken in virtualization approach. However, in deployment, the time taken by database image of VM is much less than the time taken by the database image of container. Contrary to this, the time taken by application image of VM is longer than the time taken by application image of container. Furthermore, the results of autoscaling test, conducted with the help of stress testing, show that in virtualization the average time taken to scale VM CPU is higher than the average time to scale up and down the VM memory. However, the time taken in container scaling is higher to VM scaling time because containers in this experiment are using Docker for Windows and they do not support resources modification of containers.

6 Conclusion

In this paper, concepts of microservice architecture have been discussed. Basically, the aim of the experimental study was to develop a system with which the complexity of microservices architecture can be reduced and the scaling of the resources can be automated. For this purpose, the study experimented with the implementation of virtualization and containerization for microservice-based applications. The methodology used for this project is based on research and development where challenges and complications associated with microservice-based applications are identified and the limitations of automatic scaling are recognized. After discovering the complexity of deployment of microservice architecture, the study explored two hosting provisioning approaches to solve this issue. In one approach, a virtual machine manager Microsoft Hyper-V is used for the provisioning of virtual machines. These virtual machines host application services via the required service framework and for the purpose of decreasing the complexity of microservice architecture, the provisioning of virtual machines has been automated. However, in the other approach, Docker containers engine has been used in order to automate the deployment of application services. With the help of Docker, the applications and platforms are hosted over an abstract layer on top of the host operating system. With containers, the processes are isolated strongly while the resource utilization is also minimum that resulted in reducing the overhead as compared to virtual machines. This way, an ideal environment is established for microservice-based applications. Similarly, in order to execute the effective automation of the scalability, for the resources of the hosted application services, the auto-scaling approach implemented in this research was scaling up of application services where the computing resources of application service is scaled according to the demand by adjusting the capacity of CPU unit or RAM accordingly.

Chapter 1 discusses the background of all the technologies used in the project. The microservice architecture was first introduced in 2005 and the architecture works by dividing a single application into multiple small services. Each service runs its own processes independent of other services and communication is maintained between the services with the help of lightweight mechanisms such as HTTP APIs. This architecture provides higher autonomy for the services as they are not relying on other services to execute. While microservice architecture involves advantages such as fault isolation, quick deliveries of services, and flexible technology. It is found that some challenges are also associated with their structure. One of these challenges is the

increased complexity of the system as it requires intensive planning and accuracy in the process of communication between the services, along with their monitoring, testing, and deployment. In order to reduce the complexity of the architecture, two approaches of virtualization and containerization have been explored by various researchers. In virtualization, multiple virtual machines are run over a single machine for the services and they consist of their own operating systems. On the other hand, in containerization, containers run over a single machine, which provides an abstract layer over the host operating system and the services over them are run in complete isolation. The chapter further discusses manual application deployment and automated deployment with the help of scripts. Moreover, the chapter entails the scaling processes of the applications and the way the automatic scaling of application services can be beneficial in the microservice architecture.

In chapter 2, the experiment conducted for the project has been discussed. The deployment management system for the experiment consisted of a Windows host setup with windows 2019 standard edition version installed. Furthermore, two virtual machines are set up over Microsoft Hyper-V that is configured over the host server to meet all the virtualization needs of the experiment. One of these virtual machines acted as a DNS server for the experiment, while the other hosting Docker containers engine which ensured the provision of the containers over this VM. Additionally, a system web interface is used as a management interface for the deployment management system. The web interface provides, a user-friendly interface to the system which assisted in achieving the main objective of the project, i.e. minimizing the complexity of microservice architecture. System web service framework used in the experiment is .Net Core 2.1, the web service which provides a link between the system web interface, database, and messaging queue with which execution messages are communicated to the system engines. The chapter also elaborates the implementation issues faced during the experiment, such as Windows do not support resource modification for Docker due to which the CPU count and memory capacity cannot be updated over the containers, Docker for Windows do not support file copying process while the containers are in running condition, which can introduce a small downtime for the services deployment and Hot scaling of CPU count is not supported by Hyper-V.

Chapter 3 of the report discusses the results gained from the experiment that is conducted in the study. To achieve the objective of reducing the complexity of microservices architecture,

microservice-based applications were deployed on the deployment management system in two different ways, i.e. on a virtual machine and on a container. When the two approaches were compared with each other, it was found that while in provisioning, the container had utilized overall lesser average time than the VM, in deployment the database image of containers took more time than database image of VM. However, the application image of the container took less time than the application image of VM. For the experiment regarding the automation of scaling, the results showed that the average time taken for scaling the containers is higher than the average time of VM. The results are further discussed thoroughly in chapter 4 and compared with pertinent literature in order to conduct an analysis.

6.1 Future Work

For future prospects, the system can be implemented on other platforms especially Linux, and integrated with other hypervisors like VMware, while discovering another scaling approach like scaling out the application service by creating a new instance of the same microservice hosting and load balance the incoming traffic between all the created instances. This will also provide an opportunity to discover different aspects of load balancing, the application traffic, and develop a more efficient algorithm for scaling the application services. Also, one of the interesting topics to look at in future work is using machine learning in auto-scaling mechanisms. It can also lead to the implementation of an artificial intelligence driven auto-scaling engine.

References

Abar, S., Lemarinier, P., Theodoropoulos, G.K. and OHare, G.M., 2014, May. Automated dynamic resource provisioning and monitoring in virtualized large-scale datacenter. In *2014 IEEE 28th International Conference on Advanced Information Networking and Applications* (pp. 961-970). IEEE.

Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M. and Steinder, M., 2015, September. Performance evaluation of microservices architectures using containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications* (pp. 27-34). IEEE.

Bacigalupo, D.A., Van Hemert, J., Chen, X., Usmani, A., Chester, A.P., He, L., Dillenberger, D.N., Wills, G.B., Gilbert, L. and Jarvis, S.A., 2011. Managing dynamic enterprise and urgent workloads on clouds using layered queuing and historical performance models. *Simulation Modelling Practice and Theory*, *19*(6), pp.1479-1495.

Barrett, D. and Kipper, G., 2010. *Virtualization and forensics: A digital forensic investigator's guide to virtual environments*. Syngress.

Bilal, A., Vajda, A. and Tarik, T., 2016, September. Impact of network function virtualization: A study based on real-life mobile network data. In *2016 International Wireless Communications and Mobile Computing Conference (IWCMC)*(pp. 541-546). IEEE.

Boyer, F., Etchevers, X., de Palma, N. and Tao, X., 2018, May. Poster: A declarative approach for updating distributed microservices. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)* (pp. 392-393). IEEE.

Bravetti, M., Giallorenzo, S., Mauro, J., Talevi, I. and Zavattaro, G., 2019. Optimal and automated deployment for microservices. In *International Conference on Fundamental Approaches to Software Engineering* (pp. 351-368). Springer, Cham.

Brechner, E., 2015. *Agile Project Management with Kanban*. Microsoft Press.

Castiglioni, F., Proietti, P., and De Gaetano, R., 2016. Cloud solutions vs. traditional web apps. IBM Developer. Retrieved from: https://www.ibm.com/developerworks/cloud/library/cl-get-the-most-out-of-cloud-1-trs/index.html

Cerny, T., Donahoo, M.J. and Trnka, M., 2018. Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review*, *17*(4), pp.29-45.

Chen, T. and Bahsoon, R., 2015. Toward a smarter cloud: Self-aware autoscaling of cloud configurations and resources. *Computer*, *48*(9), pp.93-96.

Das, S., Li, F., Narasayya, V. R., & König, A. C. 2016. *Automated Demand-driven Resource Scaling in Relational. (pp. 1923-1934*). San Francisco: ACM New York.

Deploy. 2013. *White Paper: The Benefits of Deployment Automation*, Octopus Deploy, Retrieved from, http://download.octopusdeploy.com/files/whitepaper-automated-deployment-octopus-deploy.pdf

Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L., 2017. Microservices: yesterday, today, and tomorrow. In *Present and ulterior software engineering* (pp. 195-216). Springer, Cham.

Fazio, M., Celesti, A., Ranjan, R., Liu, C., Chen, L. and Villari, M., 2016. Open issues in scheduling microservices in the cloud. *IEEE Cloud Computing*, *3*(5), pp.81-88.

Felter, W., Ferreira, A., Rajamony, R. and Rubio, J., 2015, March. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)* (pp. 171-172). IEEE.

Hähnle, R., 2012, September. The abstract behavioral specification language: a tutorial introduction. In *International Symposium on Formal Methods for Components and Objects* (pp. 1-37). Springer, Berlin, Heidelberg.

Han, R., Guo, L., Ghanem, M.M. and Guo, Y., 2012, May. Lightweight resource scaling for cloud applications. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)* (pp. 644-651). IEEE Computer Society.

Hasselbring, W. and Steinacker, G., 2017, April. Microservice architectures for scalability, agility and reliability in e-commerce. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)* (pp. 243-246). IEEE.

Jin, Y., Wen, Y. and Chen, Q., 2012, March. Energy efficiency and server virtualization in data centers: An empirical investigation. In *2012 Proceedings IEEE INFOCOM Workshops* (pp. 133-138). IEEE.

Joy, A.M., 2015, March. Performance comparison between linux containers and virtual machines. In *2015 International Conference on Advances in Computer Engineering and Applications* (pp. 342-346). IEEE.

Khazaei, H., Barna, C., Beigi-Mohammadi, N. and Litoiu, M., 2016, December. Efficiency analysis of provisioning microservices. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (pp. 261-268). IEEE.

Knoche, H. and Hasselbring, W., 2019. Drivers and barriers for microservice adoption–a survey among professionals in Germany. Enterprise Modelling and Information Systems Architectures (EMISAJ)–International Journal of Conceptual Modeling: Vol. 14, Nr. 1.

Lewis, J. and Fowler, M., 2014. *A Definition of this New Architectural Term*. Retrieved from: https://martinfowler.com/articles/microservices.html

Mavridis, I. and Karatza, H., 2017, July. Performance and overhead study of containers running on top of virtual machines. In *2017 IEEE 19th Conference on Business Informatics (CBI)* (Vol. 2, pp. 32-38). IEEE.

Mavridis, I. and Karatza, H., 2019. Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing. *Future Generation Computer Systems*, *94*, pp.674-696.

Microsoft, 2018. Introduction to Hyper-V on Windows 10. Microsoft Docs. Retrieved from: https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/

Microsoft, 2019a. *Overview of Azure Service Fabric*. Microsoft Azure Documentation. Retrieved from: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-overview

Microsoft, 2019b. *Why use a microservices approach to building applications?* Microsoft Azure Documentation. Retrieved from: https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-overview-microservices

O'Connor, R.V. and Clarke, P., 2015, August. Software process reflexivity and business performance: initial results from an empirical study. In *Proceedings of the 2015 International Conference on Software and System Process*(pp. 142-146). ACM.

O'Connor, R.V., Elger, P. and Clarke, P.M., 2017. Continuous software engineering—A microservices architecture perspective. *Journal of Software: Evolution and Process*, *29*(11), p.e1866.

Peinl, R., Holzschuher, F. and Pfitzer, F., 2016. Docker cluster management for the cloud-survey results and own solution. *Journal of Grid Computing*, *14*(2), pp.265-282.

Pfaller, S. and Hartley, T., 2018, *Comparison of Windows and Linux as Docker Hosts.* The 9th annual conference of Computing and Information Technology Research and Education New Zealand

Queirós, A., Faria, D., & Almeida, F. (2017). Strengths and limitations of qualitative and quantitative research methods. *European Journal of Education Studies, 3*(9), 369-387.

Rajasekar, S., Philominathan, P. and Chinnathambi, V., 2013. Research Methodology. *arXiv preprint physics/0601009v3.* pp. 1-53

Salah, T., Zemerly, M.J., Yeun, C.Y., Al-Qutayri, M. and Al-Hammadi, Y., 2017, March. Performance comparison between container-based and VM-based services. In *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)* (pp. 185-190). IEEE.

Seo, K.T., Hwang, H.S., Moon, I.Y., Kwon, O.Y. and Kim, B.J., 2014. Performance comparison analysis of linux container and virtual machine for building cloud. *Advanced Science and Technology Letters*, *66*(105-111), p.2.

Soni, M., 2015, November. End to end automation on cloud with build pipeline: the case for DevOps in insurance industry, continuous integration, continuous testing, and continuous delivery. In *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)* (pp. 85-89). IEEE.

Taherizadeh, S. and Stankovski, V., 2018. Dynamic multi-level auto-scaling rules for containerized applications. *The Computer Journal*, *62*(2), pp.174-197.

Taherizadeh, S., Jones, A.C., Taylor, I., Zhao, Z. and Stankovski, V., 2018. Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review. *Journal of Systems and Software*, *136*, pp.19-38.

Taibi, D., Lenarduzzi, V. and Pahl, C., 2018, March. Architectural Patterns for Microservices: A Systematic Mapping Study. In *CLOSER* (pp. 221-232).

Taibi, D., Lenarduzzi, V., Pahl, C. and Janes, A., 2017, May. Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. In *Proceedings of the XP2017 Scientific Workshops* (p. 23). ACM.

Vaquero, L.M., Rodero-Merino, L. and Buyya, R., 2011. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, *41*(1), pp.45-52.

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R. and Gil, S., 2015, September. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)* (pp. 583-590). IEEE.

Wan, X., Guan, X., Wang, T., Bai, G. and Choi, B.Y., 2018. Application deployment using Microservice and Docker containers: Framework and optimization. *Journal of Network and Computer Applications*, *119*, pp.97-109.

Wang, A., Iyer, M., Dutta, R., Rouskas, G.N. and Baldine, I., 2012. Network virtualization: Technologies, perspectives, and frontiers. *Journal of Lightwave Technology*, *31*(4), pp.523-537.

Woods, D., 2015. *On Infrastructure at Scale: A Cascading Failure of Distributed Systems*, Medium. Retrieved from: https://medium.com/@daniel.p.woods/on-infrastructure-at-scale-a-cascading-failure-of-distributed-systems-7cff2a3cd2df

Xavier, M.G., Neves, M.V., Rossi, F.D., Ferreto, T.C., Lange, T. and De Rose, C.A., 2013, February. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (pp. 233-240). IEEE.

Xiao, Z., Chen, Q. and Luo, H., 2014. Automatic scaling of internet applications for cloud computing services. *IEEE Transactions on Computers*, *63*(5), pp.1111-1123.

Zhang, Y., Vasilescu, B., Wang, H. and Filkov, V., 2018. One size does not fit all: An Empirical Study of Containerized Continuous Deployment Workflows. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symnposium on the Foundation of Software Engineering.* (pp. 295-306). ACM.

Zimmermann, O., 2017. Microservices tenets. *Computer Science-Research and Development*, *32*(3-4), pp.301-310.