A Novel Approach to Containerize Existing Applications

Vineet Haresh Palan
A Novel Approach to Containerize Existing Applications

by

Vineet Haresh Palan

A thesis submitted to
Florida Institute of Technology
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
June 2018
We, the undersigned committee, hereby approve the attached thesis.

A Novel Approach to Containerize Existing Applications

by

Vineet Haresh Palan

_____________________________________________________________________________________

Marco Carvalho, Ph.D.
Major Advisor
Dean, College of Engineering and Computing
Executive Director, Harris Institute for Assured Information

_____________________________________________________________________________________

William Allen, Ph.D.
Committee Member
Associate Professor
Harris Institute for Assured Information

_____________________________________________________________________________________

Muzaffar Shaikh, Ph.D.
Committee Member
Department Head
Engineering Systems
Abstract

Title: A Novel Approach to Containerize Existing Applications
Author: Vineet Haresh Palan
Committee Chair: Marco Carvalho, Ph.D.

Cloud computing has dominated the way different services are offered, utilized and managed over the internet. Typically, applications run inside the virtual machines in an isolated environment. However, there is always a hardware virtualization overhead for running these virtual machines. Recently, Linux containers have gathered attention because of their significantly lower overhead when compared with VM through OS-level virtualization. They mainly provide high availability, isolation, portability, interoperability and scalability. Thus, everyone is trying to move their existing applications onto Linux containers with the help of available tools and framework. However, these tools fail to gather all the relevant information required to containerize these applications because of their specific and rigid approach. To make this process accurate yet straightforward, we propose a generic approach that handles this problem without being dependent on any vendor, cloud provider or ap-
application. Moreover, we show the feasibility and effectiveness of our approach by
(1) Creating a prototype of an automated framework that uses our approach along
with human interaction. (2) Demonstrating two case studies of the applications
that can be migrated using our approach and framework.
## Contents

1 Introduction 1

2 Background 5

2.1 Virtualization Technology .......................... 5

2.1.1 Hardware Level Virtualization .................... 6

2.1.2 Operating System Level Virtualization ........... 7

2.2 Containers Overview .................................. 8

2.2.1 Kernel Features .................................. 10

2.2.2 Advantages of Containers ........................ 11

2.3 Open Container Initiative ............................ 14

2.3.1 Image Specification .............................. 14

2.3.2 Runtime Specification ............................ 14

2.3.3 Distribution Specification ....................... 15
2.4 Container Technologies ........................................... 15
  2.4.1 Application Container ................................. 16
  2.4.2 System Container ........................................ 17

3 Literature Review .................................................. 19
  3.1 Manual Approach ................................................. 19
    3.1.1 Lift and Shift ........................................ 20
  3.2 Commercial Tools .............................................. 21
    3.2.1 Docker’s Modernize Traditional Applications(MTA) .... 22
    3.2.2 Virtuozzo ................................................ 23
    3.2.3 App2Cloud ............................................... 24
  3.3 Open Source Tools .............................................. 24
    3.3.1 VM2Docker ............................................. 24
    3.3.2 Image2Docker .......................................... 26
  3.4 Analysis of Docker and Dockerfile .................... 29
    3.4.1 Docker’s Popularity ................................. 30
    3.4.2 Docker Images ...................................... 30
    3.4.3 Understanding Dockerfile .......................... 32
  3.5 Shortcomings of existing system .................... 33
4 Proposed Approach

4.1 Steps

4.1.1 Information gathering: ................................. 36

4.1.2 Create temporary machine ............................ 36

4.1.3 Comparing both the machines ......................... 37

4.1.4 Resolve Package Dependencies ....................... 39

4.1.5 Process-Package Relationship ....................... 40

4.1.6 Application files and configuration ................... 40

4.1.7 Create Container Image ............................... 42

4.2 Automation Framework ................................. 42

4.2.1 Driver ............................................. 43

4.2.2 Aggregator ......................................... 44

4.2.3 Provisioner ........................................ 44

4.2.4 Image Builder ....................................... 45

5 Prototype

5.1 Convert2Container ...................................... 46

5.1.1 User Credentials .................................... 47

5.1.2 Target machine details ............................. 47
6 Case Study

6.1 Scenario I

6.1.1 Setup

6.1.2 Experiment Execution

6.1.3 Evaluation

6.2 Scenario II

6.2.1 Setup

6.2.2 Experiment Execution

6.2.3 Evaluation

6.3 Scenario Observations:
List of Figures

2.1 Type 1: Native or Bare-metal Hypervisor ........................ 7
2.2 Type 2: Hosted Hypervisor ........................................... 7
2.3 Operating System-Level Virtualization ............................... 8
2.4 The architecture of Linux Containers in Red Hat Enterprise Linux 7 9
2.5 Density of virtual machines and containers .......................... 11
2.6 Resource Utilization of virtual machines and containers .......... 12
2.7 Scaling virtual machines and containers .............................. 13
2.8 Types of Container ..................................................... 16
3.1 Docker adoption by Datadog customers from 2013 to 2017 1 20
3.2 Decompose an application to small parts to move them into appli-
cation containers. ....................................................... 21
3.3 Move an entire application into system containers. .................. 21
3.4 Steps to transform an existing application to modern microservices based application. 3.2.1 22

3.5 Overview of VM2Docker framework. 25

3.6 Image2Docker approach. 27

3.7 Docker growth stats from 2013-2016 presented in Dockercon 2017 30

3.8 Docker end to end flow. 31

3.9 A sample Dockerfile used to create nginx server image. Each line in the file is a instruction which translated to a layer in the Docker image. 32

3.10 Example of a Dockerfile with Dockerfile linter suggestions 33

4.1 The target machine has applications running on them. Clone machine has same OS without any customization. 37

4.2 The set difference of installed packages on target machine with the clone machine. 37

4.3 Example of dependency graph for OpenJDK-8 and Nodejs packages. 39

4.4 Example of process-package relation between Java Application & openjdk-8-jdk package and Node.js Application & nodejs package. 40

4.5 A high-level view of the framework’s components 43

6.1 A high-level view of the VINE’s architecture. 54
6.2 Accepts the IP Address and user credentials of the target machine. 55

6.3 Shows the gathered information about the target machine. . . . . . 56

6.4 Provides an option to use already created clone machine or create
clone machine using Vagrant and VirtualBox. . . . . . . . . . . . . . . 57

6.5 Shows the running processes that are not present on clone machine
along with its associated ports. . . . . . . . . . . . . . . . . . . . . 58

6.6 Shows the process-package relation. . . . . . . . . . . . . . . . . 59

6.7 Shows the process-file relation. . . . . . . . . . . . . . . . . . . . 60

6.8 Shows the options to create container image. . . . . . . . . . . . . 61

6.9 A high-level view of the MEAN stack application. . . . . . . . . . 63
List of Tables

6.1 Shows the size of Docker Images created for VINE application. . . . 62
6.2 Shows the size of Docker Images created by both the methods. . . 65
Listings

A.1 Dockerfile generated for Java Process. ........................................... 73
A.2 Dockerfile generated for Tomcat Process. ................................. 73
A.3 Dockerfile generated for MySQL Process. ................................. 74
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VM</strong></td>
<td>Virtual Machine</td>
</tr>
<tr>
<td><strong>LXC</strong></td>
<td>Linux Container Technology</td>
</tr>
<tr>
<td><strong>KVM</strong></td>
<td>Kernel Virtual Machine</td>
</tr>
<tr>
<td><strong>OS</strong></td>
<td>Operating System</td>
</tr>
<tr>
<td><strong>AWS</strong></td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td><strong>GCP</strong></td>
<td>Google Cloud Platform</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td><strong>PID</strong></td>
<td>Process Identifier</td>
</tr>
<tr>
<td><strong>IPC</strong></td>
<td>Inter Process Communication</td>
</tr>
<tr>
<td><strong>TCP</strong></td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td><strong>SSH</strong></td>
<td>Secure Shell</td>
</tr>
</tbody>
</table>
Acknowledgements

This thesis was possible with the help and guidance of many essential people.

First, I would like to thank my advisor, Dr. Marco Carvalho for always believing in me. His constant motivation and invaluable advice during the crucial times helped me to finish my thesis in time.

I would like to thank Evan Stoner for mentoring me during my research assistantship and providing me with an opportunity to work on Linux containers from where I got the motivation for this thesis.

I would like to thank my parents, Manisha and Haresh Palan and my sister Dr. Tanvee Somaiya for their patience and encouragement throughout my thesis.

I would also like to thank Daljeet Kaur Kaushal, Neha Sagar Khan, Daanish Millwala and Tapas Joshi for listening to my ideas, reviewing my thesis and providing timely feedback which helped me to put together this document.
This thesis is dedicated to my parents Manisha and Haresh Palan
for constantly encouraging
and believing in me.
Chapter 1

Introduction

As cloud computing has evolved over the years, the use of different virtualization techniques has drastically increased. These changes have resulted in the massive demand for highly efficient and performant solutions for virtualization, outperforming the existing methods. Virtualization helps in optimizing the utilization of the available physical resources by treating them as logical or virtual resources which can be allocated dynamically on the fly.

Recently, operating system level virtualization technology, i.e., Linux containers have gained traction as the preferred mode of deployment. By utilizing the operating system’s virtualization principles, containers provide great interoperability. Although virtual machines and Linux containers use virtualization techniques, virtual machines have much higher overhead due to hardware emulation when compared to Linux container [1]. Containers reduce the emulation cost by shifting the virtualization layer from hardware to operating system level.
Lately, Linux containers’ popularity has brought a paradigm shift [9]. Everyone is planning to containerize their existing applications. Containerization is the process of providing an application specific environment with all the required libraries, files, and configuration using Linux containers [11]. However, there are few challenges when performing containerization:

1. What information is required to containerize an application.

2. How to remodel the existing architecture to support container-based architecture.

3. What best practices need to be followed while creating a Linux container.

In this thesis, we target these challenges by proposing an approach that will address these issues. Our proposed approach expects the user to follow a series of steps to complete the migration process. However, each step when performed manually could be time-consuming and error-prone. Thus, we also propose an automated framework that follows our approach to containerize an application easily. Our framework will be independent of any vendor or platform; instead, it will be a generic solution that can be used by anyone trying to containerize the application. Based on the limitations of existing tools, we define the following requirements for our framework:

1. The framework must be platform independent. It should support any Linux Operating System such as Ubuntu\(^1\), Fedora\(^2\), CentOS\(^3\), etc.

---

\(^1\)https://www.ubuntu.com/
\(^2\)https://getfedora.org/
\(^3\)https://www.centos.org/
2. The framework must be vendor independent. It should support any Linux Container vendors such as Docker\(^4\), Rkt\(^5\), LXC\(^6\), etc.

3. The framework must be cloud-provider independent. It should support any provider such as AWS\(^7\), GCP\(^8\), Azure\(^9\), etc.

4. The framework must be hypervisor independent. It should support any hypervisors such as KVM\(^10\), VirtualBox\(^11\), etc.

5. The framework must allow a user to interact and decide what happens at each step.

To design a framework that meets all these requirements, we study the working of existing tools, their functionality, and their limitations. We use these findings to develop a new framework along with a prototype implementation of it.

The main contributions of this thesis are as follows:

- We review the features and limitations of the existing approaches.

- We propose a guided, step by step procedure to containerize applications running on Linux based OS.

- We propose a framework along with a prototype implementation of it.

---

\(^4\)https://www.docker.com/
\(^5\)https://coreos.com/rkt/
\(^6\)https://linuxcontainers.org/
\(^7\)https://aws.amazon.com/
\(^8\)https://cloud.google.com/
\(^9\)https://azure.microsoft.com/
\(^10\)https://www.linux-kvm.org/
\(^11\)https://https://www.virtualbox.org/
• We perform two case studies and evaluate the effectiveness of our approach along with its advantages.

The rest of this thesis is organized as follows: Chapter 2 contains the background of virtualization along with different virtualization techniques. We will also discuss a brief history of Linux containers. Chapter 3 contains the literature review of all the work done in the field of application containerization and tools used to do so. Chapter 4 describes the proposed approach along with automated framework. Chapter 5 explains the prototype implementation of the framework. Chapter 6 evaluates the effectiveness of our proposed approach and framework. Chapter 7 summarizes the benefits and future scope of our solution.
Chapter 2

Background

In this chapter, we look into virtualization technology and its types. We will explain how Linux container came into existence, underlying technology, different types, and advantages. We will also discuss why standards are created around the Linux container.

2.1 Virtualization Technology

Virtualization technology refers to the abstraction of computing resources such as CPU, storage, network, memory, application stack, and database from applications & end users consuming the service [14]. Virtualization technology relies on the software components to simulate the hardware functionality by creating virtual resources. In today’s computing world, virtualization is mainly used at hardware and operating system level.
2.1.1 Hardware Level Virtualization

Hardware Level Virtualization has widely been used over the last decade for implementing virtualization. It uses a hypervisor to create an abstraction layer between the software and the underlying hardware. A hypervisor is a piece of software that creates virtual resources which run directly on the hardware isolated from the underlying host operating system [10]. A hypervisor is classified into two types:

1. **Type 1 - Native or Bare-metal Hypervisor:**
   Type 1 hypervisors run directly over the host hardware without the need of host OS as shown in figure 2.1. It handles the access to the hardware resources made by the guest operating systems thereby acting as a mediator [2].

2. **Type 2 - Hosted Hypervisor:**
   The Type 2 hypervisor works inside the host OS, thereby ensuring access to the hardware for guest OS as shown in figure 2.2. The type 2 hypervisor creates complete isolation and abstraction from the host machine’s hardware. This type of virtualization is also known as full virtualization [2].
2.1.2 Operating System Level Virtualization

Operating System Level Virtualization has gained traction in last few years because of its low overhead [12]. Hardware level virtualization is considered as heavyweighted because it relies on hardware emulation. Alternatively, in OS-level virtualization, there is a container engine that uses kernel features like cgroups, namespaces, chroot, etc. for creating isolated user-space instances known as containers, on top of the host machine as shown in figure 2.3. The containers share the host machine’s kernel with the help of container engine instead of running a full operating system as it does in hypervisor-based virtualization, thereby reducing the overall overhead.
Figure 2.3: Operating System-Level Virtualization

Containerization was implemented in UNIX operating system back in 1979 using chroot. Later, as containerization technology evolved, more essential features were implemented for isolating file system, users, and networking. The first full implementation of Linux container manager is LXC\(^1\). Later, Docker\(^2\) came up with the full ecosystem to manage containers. Rkt\(^3\) was created by coreOS to overcome the drawbacks of Docker [2].

2.2 Containers Overview

Containers are used to provide isolation and low overhead in Linux environment. The management interface or the container engine interacts with the kernel components and provides tools for creating and managing the containers as shown in

\(^1\)https://linuxcontainers.org/
\(^2\)https://www.docker.com/
\(^3\)https://coreos.com/rkt/
Since containers are lightweight, we can achieve higher density and better performance as compared to virtual machines [11].

![Figure 2.4: The architecture of Linux Containers in Red Hat Enterprise Linux 7](image)

Running containers on non-Linux based OS is technically not possible as it requires access to Linux kernel. A small, lightweight virtual machine is created on non-Linux based OS like Mac or Windows. The hardware virtualization feature should be enabled on the host machine to support spinning up of a virtual machine. Once a small Linux OS is running, containers are created using the virtual machine’s kernel.
2.2.1 Kernel Features

cgroups

cgroups or control groups is a feature in the Linux kernel that provides metering and limiting of resources like memory, CPU, the group of processes and disk throughput. For resource management, most of the container technology uses cgroups as an underlying mechanism [4]. cgroups can be monitored, configured, denied access to specific resources, and even reconfigured dynamically on a running system.

namespaces

The Linux namespace provides each process with its own view of the system, i.e., limits what a process can see. The Linux namespaces create an abstraction of the resources and make it appear as a separate instance inside the container. Thus, different containers can use the same resources simultaneously.

SELinux

Security-Enhanced Linux or SELinux is an implementation of multi-level security (MLS), mandatory access control (MAC) mechanism and multi-category security (MCS) in the Linux kernel. By applying SELinux policies, it provides secure separation of containers.
chroot

chroot or change root is used to change the root directory of the process and its children to a new directory which is only visible to a given process. The idea to use chroot by containers is to isolate and share the filesystem across multiple containers [4].

### 2.2.2 Advantages of Containers

#### Density

The density is the maximum number of containers or virtual machine that a host machine can support without any of them being starved for resources. Containers are lightweight than virtual machines, i.e., they have less memory utilization per instance. Thus more containers can run on the host machine as compared to virtual machines as shown in figure 2.5.

![Figure 2.5: Density of virtual machines and containers](image)
**Resource utilization**

Containers provide efficient resource utilization when compared with a Virtual machine as they share resources between them. In case of the virtual machines, a full operating system is booted even if a single application is going to run inside it. As shown in figure 2.6, the required resources from the host machine are reserved for each virtual machine.

![Figure 2.6: Resource Utilization of virtual machines and containers](image)

**Scaling**

Resizing the limit of the resource (such as CPU or RAM) in the container can be done on the fly, without rebooting using cgroup kernel feature. However, in case of virtual machines, we have to stop them first, reconfigure and then restart. Subsequently, as the figure 2.7 depicts, there is always a downtime while scaling virtual machines.
**Figure 2.7:** Scaling virtual machines and containers

### Faster Provisioning

Containers share the host machine’s kernel and thus require significantly less time to start. They act as a process running in an isolated space. In case of virtual machines, a full operating system is loaded and thus requires more time to spin up when compared to containers.

### Portability

Portability is moving an application from one environment to another. Containers provide an environment to run an application along with the required files and configuration. Thus, running them on any operating system or hardware does not affect the application inside it.
2.3 Open Container Initiative

Over a period of years, many container technologies have emerged. Every vendor leads its way with container technology thereby having its own specification and method of running. Different leaders in container industry such as CoreOS, Docker came together to create an open governance project called Open Container Initiative (OCI)\(^4\) under the Linux Foundation. OCI helps in providing open industry standards around container formats and runtime, i.e., it defines specifications to create OCI Image (image-spec) and OCI Runtime (runtime-spec) to run the OCI Image.

2.3.1 Image Specification

Image Specification (image-spec) defines a way to create an OCI based container image. The OCI Image is the collection of image manifest (metadata about contents), image configurations (information related to application and environment) and layered filesystem containing application files and settings.

2.3.2 Runtime Specification

Runtime Specification (runtime-spec) defines a way to run the OCI Image by unpacking the OCI image into an OCI Runtime filesystem bundle. OCI Image contains all the information required to launch the application (such as command, arguments, arguments, \footnote{https://www.opencontainers.org/}
users, environment variables). Any container engine like Docker, rkt can run the OCI image without any additional arguments.

- docker run my_app:v1.0.0
- rkt run my_app,version=v1.0.0

2.3.3 Distribution Specification

Currently, OCI is also working on creating Distribution Specification to standardize the way image distribution is done. This new specification will support interoperability within the container ecosystem.

2.4 Container Technologies

Usually, containers are application-centric, i.e., they contain one application per container and are known as application containers. Alternatively, there are system containers, which work differently. They behave like a full operating system but share the host machine’s kernel. All the components of the application run inside the single system container [13]. Depending upon the requirement, any of these types of containers can be selected. Both container technologies have their pros and cons which are discussed further.
2.4.1 Application Container

An application container as shown in figure 2.8 runs a single process inside a container and thus creates a separate container for each application component. It contains everything required by each component of an application for its execution. Therefore, it runs similarly on any environment regardless of its operating system or underlying hardware. It is more appropriate for creating projects that follow microservice architecture. Thus, by using application containers, it is quite easy to scale one or more components of the application.

Docker

Docker is an open-source container technology that provides a platform for building, running and shipping applications along with their dependencies inside a virtual environment using containers. A French company called DotCloud created docker.
DotCloud provided PaaS (Platform as a Service) hosting for web applications. Later it was renamed to Docker Inc. Initially, Docker used LXC under the hood. However, with version 0.9, Docker created its own driver called libcontainer\textsuperscript{5} to access the Linux kernel’s features. Later, with the advent of OCI, they upgraded to runC.

**Rkt**

CoreOS is an open-source, lightweight operating system designed specifically to support containerization. CoreOS and Docker had an agreement to create a simple container engine. However, Docker started to add new features like clustering, networking, etc. making it a more complex system. On the other hand, CoreOS’s container engine, rkt, preserves simplicity and provides the most rigorous security model. Rkt also follows the OCI standards to create OCI based container images known as App Container Image (ACI) and runs them using OCI based runtime environment.

### 2.4.2 System Container

A system container as shown in figure 2.8 acts more like a full Operating System. It can run full-featured init systems like SysVinit, OpenRC, and Systemd that allows multiple processes to run inside a single Linux container. This type of container is preferable for many legacy or monolithic architecture-based applications, as it enables reusing the application’s architecture and configurations. System containers

\textsuperscript{5}\url{https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/}
might have a slower start-up time as they are heavier when compared to Application Containers due to the additional services required for running multiple processes[6].

LXC

Linux Container or LXC is the first complete implementation of Linux Container Manager. LXC is something in between a chroot and a virtual machine. The main aim of LXC is to create an environment which is as close as possible to a standard Linux installation without any need of hardware emulation and a separate kernel. LXC uses AppArmor Security profile for security. It also uses cgroups to limit access to the containers. Thus, the container only sees its own file system and process space. Currently, LXC uses the following features of the kernel:

- Kernel namespaces (such as user, ipc, mount, uts, net, and pid)
- SELinux profiles and AppArmor
- Chroots
- CGroups or Control Groups
- Seccomp policies
- Kernel capabilities

The main advantage of LXC is its lightweight implementation, helping it to perform at native speeds and providing better filesystem and network isolation [4].
Chapter 3

Literature Review

In this chapter, we review the available containerization approaches using commercial and open source tools. We further evaluate how each approach works and summarize their limitations. These limitations then become the baseline for this thesis.

3.1 Manual Approach

Today, many companies are using containers. Figure 3.1 shows the Datadog’s\(^1\) research based on 10,000 companies, and 185 million containers running in real world. This research found that 40% customers had adopted Docker containers in a year.

\(^1\)https://www.datadoghq.com/docker-adoption/
Figure 3.1: Docker adoption by Datadog customers from 2013 to 2017

3.1.1 Lift and Shift

Lift and shift is an approach by which we move a running application from one environment to another. This approach was initially used to move the applications from bare-metal to virtual machines on the cloud. Now, a similar strategy is used to containerize applications. An application running on a machine can be moved into System containers or Application containers based on their architecture. If an application can be decomposed into smaller parts, they are migrated to application containers using Docker, Rkt, etc. Figure 3.2 shows how each part of an application can be moved into a different application containers.
Figure 3.2: Decompose an application to small parts to move them into application containers.

If the application is tightly coupled, it is preferable to move it into System container using OpenVZ, LXC. Figure 3.3 shows how an entire application is migrated into system container.

Figure 3.3: Move an entire application into system containers.

3.2 Commercial Tools

Many companies are leading the way in virtualization technologies. These companies also provide solutions to migrate physical machine to virtual machine or
containers. These solutions usually are offered as premium services along with additional cost. We will go through the migration solutions provided by different companies.

### 3.2.1 Docker’s Modernize Traditional Applications (MTA)

Docker’s Modernize Traditional Applications (MTA)\(^2\) is a digital learning resource that provides information about transforming the legacy application into microservice driven architecture. Docker supports this service to all its users who have opted for Docker’s Enterprise Edition. Figure 3.4 shows step by step process to containerize application using Docker’s MTA process.

![Figure 3.4: Steps to transform an existing application to modern microservices based application.](image)

The existing application is converted into Dockerfile using open source tool Image2Docker. Once Dockerfile is ready, the decision to refactor, or introduce CI/CD automation or deploying Docker Image to new infrastructure based on the individ-

\(^2\)https://goto.docker.com/MTAkit.html
ual project requirement is taken. Docker’s cloud and infrastructure architects help their clients to make this transition.

### 3.2.2 Virtuozzo

Virtuozzo\(^3\) is a software company that provides different virtualization solutions including container virtualization based on open-source OpenVZ project. It uses patented OS virtualization solution. Virtuozzo supports migration from physical machines to virtual machines or containers and vice versa using "prlctl migrate" command. OpenVZ’s prlctl is a command line utility used to manage virtual machines and containers.

Virtuozzo supports two types of migration:

**Offline migration:** Support any suspended or stopped virtual machines/containers.

**Online (live) migration:** Support any running or paused virtual machines/containers.

OpenVZ has not described anything about the internal workings of this tool in its documentation other than its usage. Majorly, this tool is used to move running VM or containers from one machine to another machine.

---

\(^3\)https://virtuozzo.com
3.2.3 App2Cloud

App2Cloud is a California based start-up company that migrates legacy applications into Docker containers. They use their propriety automation tools and a team of highly skilled developers to transform traditional applications to fit inside Docker containers. The customers can use their on-demand web platform to upload the project. However, it is unrealistic for a company or an individual to share their source code for migration.

3.3 Open Source Tools

With the increase in popularity of Docker, the development of the containerization tools revolve around it. However, the existing work in containerization domain is quite limited. We will go through the working of different open source tools available along with pros and cons. Since these projects are community-based, most of the tools have inadequate support, incomplete documentation and lots of open issues. The usage of these open source tools might involve a huge learning curve. Also, these tools do not allow users to intervene in the ongoing process.

3.3.1 VM2Docker

Eric Lubin has proposed the VM2Docker framework as part of Masters thesis in 2015. This thesis aims at providing an automated way of converting Linux-based
Virtual Machines to a set of Docker containers as shown in figure 3.5. This framework requires each VM to run the agent (piece of software). Once the agent is running on VM, it passes information about the VM such as Operating System, running processes, installed packages, open ports to the chief (main program) over a socket. The agent is written in C, and it sets up TCP socket through which chief makes remote procedure calls. The chief initiates the conversion process and creates a Dockerfile using all the information gathered about the VM. The chief uses rsync Linux utility to make sure VM file system is in sync with Docker file system. All the processes running inside the VM are started inside the Docker container. Thus Docker container acts like system container and runs multiple processes inside it[7].

![Figure 3.5: Overview of VM2Docker framework.](image)

**Pros:**

- All the required information regarding the target VM is gathered programmatically.
• It takes advantage of Docker's layered file system by using the same base image for each container.

• The conversion results in smaller sized container when compared to VM.

• It supports major Linux distributions.

Limitations:

• It does not support Multi-Container Orchestration.

• There is not support for networking between containers.

• It has security issues as a port is kept open on target VMs.

• It does not gather information like users, ssh config, and password.

• This approach is tied to Docker.

• It does not provide any documentation for installing and testing the prototype.

3.3.2 Image2Docker

Image2Docker⁵ ⁶ is open source community based tool started in December 2016 under the umbrella of Docker Inc. Image2Docker allows a user to convert Windows and Linux based Virtual Machine Disk file into Dockerfile. To support Windows and Linux based applications, there two different versions of Image2Docker. Figure 3.6 shows the generic steps involved in the Image2Docker process.

⁵https://github.com/docker/communitytools-image2docker-win

⁶https://github.com/docker/communitytools-image2docker-linux
Image2Docker (Windows)

Image2Docker (Windows) is a PowerShell module which ports the existing applications to Docker. It mainly supports IIS (Internet Information Services) and ASP.NET based applications. It discovers various running applications like ASP.NET web applications, Microsoft SQL Server instances, Apache Web Server, etc. Every application is considered as an artifact, and a separate code is written to discover it. There are two main components of Image2Docker:

- **Discover function**: Discovers the desired artifacts from Virtual Hard Disk.

- **Generate function**: Generates the contents of Dockerfile based on discovered artifacts.

Image2Docker (Linux)

Image2Docker (Linux) is a Golang application, used to convert Linux based Virtual Hard Disk into Docker image. The application is divided into four parts:
• CLI and Workflow Orchestrator

It is the crux of the program. It orchestrates all the plugins and controls the flow of data. Once all the provisioners have completed the task, and relevant data is collected, orchestrator creates the final Dockerfile.

• Packager Plugins

The packager plugin runs before any other plugin and makes sure that the content of the virtual disk is available to other plugins via volume. The contents of the virtual disk are available only to packer plugin and in read-only format. It extracts all the material required by other plugin and sends this information to other plugins.

• Detective Plugins

The detective plugin inspects the source file system and gathers any information that provisioner will require to create a Docker Image. A simple example of a detective plugin is identifying the operating system.

• Provisioner Plugins

Once the detective plugin finds out the required information, the corresponding provisioner is executed. A provisioner contributes to the final Dockerfile with relevant information. For example, if detective plugin identifies Ubuntu OS version 16.10, the corresponding provisioner will contribute to Dockerfile as "FROM ubuntu:16.10".
Pros:

- The tool has a modular architecture. Thus the addition of new plugin is effortless.
- The tool runs directly on Virtual Disk. It does not require VM to be in running state.
- It supports Windows as well as major Linux distributions.

Limitations:

- Since no generic plugin works for each type of application, the users have to code a plugin according to their needs.
- This tool selects all the packages that are installed inside a VM and adds them to the Docker Image.
- Since it is a community-based project, there are a lot of open issues and no real support.

3.4 Analysis of Docker and Dockerfile

Docker is the most popular container technology today [9]. Docker natively supports creation and execution of containers on Linux-based operating system. However, a small lightweight Linux-based virtual machine is used to spin up containers on Mac and Windows-based operating system.
3.4.1 Docker’s Popularity

In the last few years, many major companies have backed Docker.\(^7\). Figure 3.7 shows that the numbers rose significantly within three years\(^8\).

![Docker growth stats from 2013-2016 presented in Dockercon 2017](https://hub.docker.com)

*Figure 3.7:* Docker growth stats from 2013-2016 presented in Dockercon 2017 \(^8\)

It is believed that Docker entered the market at just the right time when companies were looking for different application deployment solutions. They wanted a solution that provided easier, faster and portable way of deploying applications without affecting performance. Moreover, Docker had all these features along with support from open source community. Thus, Docker had already saturated the market by 2016 when other players were still trying to establish themselves.

3.4.2 Docker Images

Docker follows OCI standards to create container image known as Docker image. Docker Images can be created manually or can be downloaded from Docker registry. Docker Hub\(^9\) is a docker registry which is used to store and distribute docker

\(^7\)https://www.docker.com/customers
\(^8\)https://2017.dockercon.com
\(^9\)https://hub.docker.com
images [8]. Docker images on Docker Hub are built and contributed by individuals, companies and community members. Docker promotes the use of official Docker Images as they are optimized for the image size, provide better documentation and follow best practices.

Figure 3.8: Docker end to end flow.

Figure 3.8 shows that docker images are needed for running Docker containers. These Docker images are read-only templates which may include a base image of the operating system or a prebuilt application along with all the requirements. In figure 3.8, we have shown an example of a prebuilt image of an application.

When building images with Docker, every command executed forms a new layer on top of the previous one. We can run these commands manually or use Dockerfile to run commands automatically. A Dockerfile is a simple text document that contains the declarative instructions that are used to define how the contents of the Docker Image will be put together. Docker file is interpreted by Docker engine and converted to Docker image.

31
3.4.3 Understanding Dockerfile

Last year, Jürgen Cito did an empirical analysis of over 70000 unique Dockerfiles to characterize the Docker ecosystem, find basic quality issues and understand the evolution of Dockerfiles. The instructions in the Dockerfiles helped to answer different questions such as the most common quality issues, most common base image used, type of programming language used, etc. We provide explanations of some instructions used in Figure 3.10 which will be used in the prototype implementation.

1. FROM command that specifies the base image to be used.

2. RUN command executes the command specified. Here, it updates the apt cache and installs ngnix server.

3. ENTRYPOINT command tells the docker engine to start this image with the specified command/file. Here ngnix server is started.

4. EXPOSE command specifies on what port the container will be listening.
Once Dockerfile is created, it needs to be validated for best practices and quality issues\textsuperscript{10}. There are tools like Dockerfile Linter\textsuperscript{11} to validate these issues. Figure 3.10 shows the Dockerfile Linter’s suggestions based on the analysis it did by checking each line against the set of rules. These rules are formed by the contribution of the community members based on their expertise, issues incurred and the result achieved by following best practices [3].

\section*{3.5 Shortcomings of existing system}

By understanding the functionality and limitations of each tool, we will be addressing the following issues:

\textsuperscript{10}https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

\textsuperscript{11}https://github.com/hadolint/hadolint
• The existing systems do not gather all the information to containerize applications. In our approach, we will aggregate all the information that existing system fails to do.

• The existing systems do not follow best practices while creating container image. In our approach, we will follow the best practices as described in section 3.4.3.

• The existing systems are tied to specific containerization vendor (Docker). The user will have an option to create vendor independent container image using our approach.

• The existing systems do not allow a user to participate in the migration process. The user will be able to interact and decide what happens at each step using our approach.

• The existing system either supports system or application container. The user will have an option to create system or application container or a hybrid of both using our approach.

Therefore, the proposed framework will have the power of automation and human understanding together.
Chapter 4

Proposed Approach

In this chapter, we describe our proposed approach and discuss how this approach will help in application containerization using an automated framework. This framework will allow the users to move the applications running inside a bare metal or virtual machine into the Linux containers.

4.1 Steps

The proposed approach follows the same lift and shift philosophy explained in section 3.1.1 with some modifications. Our approach provides an intuitive way to identify the running application along with its dependencies required for containerization. As a requirement of our proposed approach, we perform the following steps.
4.1.1 Information gathering:

First, we must collect all the information about the target machine running the application. We gather information such as operating system, its distribution & version, installed packages, running processes, SSH configurations, users, and environment variables. We should also gather information about cloud platform if the target machine is hosted on it. Similarly, if the target machine is running on the local cloud platform, we gather information about the hypervisor and the tools used to create it. We then extract the meaningful data from the collected information and pass it to the next step.

4.1.2 Create temporary machine

Using the information gathered from the previous step, we create a temporary clone of the target machine with the same operating system. Thus, the cloned machine will have the vanilla version of the operating system. The word "vanilla" is used to describe the original version of the software without any customization. We use this cloned machine as a baseline for comparison with the target machine to find out all the customizations done on the target machine. The customization may include installing packages, creating new users, running processes, open TCP connections, etc. Figure 4.1, shows the additional applications running on the target machine as compared to the clone machine. Thus, any customization that is not present on the clone machine will be added to the container image. We delete the clone machine once all the required information is gathered.
4.1.3 Comparing both the machines

Once we have gathered all the details from the target (T) and the clone (C) machines, we take the set difference (T-C) on them. The figure 4.2 shows the set difference of the installed packages on the target and the clone machines. The highlighted part in the figure shows the extra packages that are not present in the clone machine and thus should be added to the container image.

Figure 4.1: The target machine has applications running on them. Clone machine has same OS without any customization.

Figure 4.2: The set difference of installed packages on target machine with the clone machine.
Similarly, we derive the other information by finding the set difference on each type of information gathered from both the machines. All this information becomes a critical part of the container image. The other type of derived information are as follows:

1. **Running Processes:** Each process will become the startup process for each container.

2. **Open TCP Ports:** To allow connection to and fro with the outside world, we bind the container’s port to the host machine’s port.

3. **Users and Groups:** Many applications are started using specific user or group. Thus, these users and group should be present in the container image to maintain consistency.

4. **Environment Variables:** Many applications depend on environment variables for their executing. Thus, the derived environment variables are added to the container image.

5. **SSH Config:** Many applications establishes SSH connection with different machine during their execution. To allow the same behavior even when application is containerized, we copy the user’s SSH keys into the container image.
4.1.4 Resolve Package Dependencies

Once we have a list of unique packages that will be installed in the container, we try to resolve the dependencies between them. Many packages are never installed directly. They are installed as a part of the dependencies for some other package. So, we create a dependency graph for each package as shown in figure 4.3

In figure 4.3, package **openjdk-8-jdk** has three dependencies. Similarly, **nodejs** package has six dependencies. Once we have created the graph for each package, we choose only the root node (package) for installation. Internally, the operating system will find the dependent packages for the root node and install them. Since there will be fewer packages after resolving dependencies, it will be easier to associate each package with the running process.
4.1.5 Process-Package Relationship

Once the number of installed packages are reduced, we then create a process-package relation, i.e., a mapping between the running process of an application and the installed packages.

![Diagram showing the process-package relation between Java Application & openjdk-8-jdk package and Node.js Application & nodejs package.]

**Figure 4.4:** Example of process-package relation between Java Application & openjdk-8-jdk package and Node.js Application & nodejs package.

Figure 4.4 shows that the openjdk-8-jdk package must be installed to run a Java application. Similarly, to run a Node.js application, nodejs package needs to be installed. If we want to move these applications into containers, we have to install the associated packages in their respective containers.

4.1.6 Application files and configuration

Once we build the process-package relation, we know the associated packages for running the application. However, some applications might also require files and
configuration for their execution. Thus, there should be a generic solution to find the associated configuration files and folders for each application process.

**Solution:**

We monitor every file and folder that is accessed by each process during its execution. This information helps us to determine the required files and configuration for each application process. However, on the target machine, since the application is already running, we might not be able to find the files that were accessed during its startup.

To identify all the file and configurations accessed by the process during its execution, we use the clone machine to run the same processes and monitor them. To achieve this, we try to make the clone machine as similar as possible to the target machine by performing the following steps:

1. We all the extra packages.

2. We create same users and groups.

3. We set environment variables.

4. We copy all the extra files and folders from the target machine.

Once we have the clone machine similar to the target machine, we start the processes and monitor the files they access during the execution. Thus we have the list of associated files and configurations for each process.
4.1.7 Create Container Image

Since we have all the information required to create container image, we can either create a system container or an application container. If we can decompose the application without making any changes to the source code, creating an application container would be an ideal option. However, if application is tightly coupled and decomposition requires significant refactoring, creating system container would be a wise decision for this scenario. This decision is taken by the user performing the containerization.

4.2 Automation Framework

The proposed approach will help the user to containerize the applications running inside a bare metal or a virtual machine. However, this approach might become time-consuming and may require significant effort as the size of the application increases. Thus, we introduce an automated framework that uses our proposed approach to perform application containerization. The framework will run each step automatically and display the results back to the user. The user decides what information needs to be transferred to the next step. Figure 4.5 shows the high-level view of the proposed solution. The number represents the steps our framework follows throughout the containerization process.
Figure 4.5: A high-level view of the framework’s components

The proposed framework is divided into four components. Each component must follow the requirements described in chapter 1. Each component is explained in detail in the following sections.

4.2.1 Driver

The driver component initiates the containerization process by getting the user credentials required to establish a connection with the target machine. The user
credentials (such as username, password or SSH Key) are shared with the next component for further processing.

### 4.2.2 Aggregator

The aggregator is an integral component of our framework as it connects to the target machine, provisioner and image builder. It first connects to the target machine to gather all the required information and extract meaningful data from it. The extracted data is shared with other components as they do not have access to the target machine.

The aggregator forwards the OS related information to the provisioner to create the clone of the target machine. Once the clone is up and running, the aggregator gathers the same information as it did for the target machine.

The aggregator then takes the set difference on each type of information as explained in section 4.1.3. This computed information is passed to the Image builder to create the container image.

### 4.2.3 Provisioner

The provisioner creates the clone of the target machine using the information provided by the aggregator. To generate the clone as similar as possible to the target machine, provisioner must comply with the following set of rules.
• For the applications running on a bare metal system: The provisioner creates a clone of the target machine by using any available hypervisor.

• For the applications running on a virtual machine: The provisioner identifies whether the target machine is hosted on a cloud or on-premises platform. The provisioner uses the same platform to create a clone of the target machine so that the clone as similar as possible to the target machine.

4.2.4 Image Builder

The Image Builder uses the information received from the aggregator to create a container image. It allows the user to create system or application container as discussed in section 2.4. It also supports the creation of vendor-independent image since all the information required to create a container image is available. Image builder also follows the best practices as explained in section 3.4.3. Eventually, the image created using Image Builder is used in running the Linux container.
Chapter 5

Prototype

In this chapter, we discuss the prototype implementation of our proposed framework. Although the prototype is based on a simplified interpretation of our framework, it only offers a subset of the features we described in chapter 4. However, it presents a reasonable basis for discussion about the feasibility and advantages of our framework.

5.1 Convert2Container

The Convert2Container (C2C) is a prototype implementation of our framework. C2C is a web-based application that allows users to migrate their existing applications into Docker Containers. Since its just a prototype of our framework, it supports the creation of Docker-based container image only. The C2C is step by step guided system that provides the user power to interact and decide what infor-
Information is relevant and should be transferred to the next step. On every step, we show the information that is not present on clone machine and allow the user to decide whether that information should be used to create the container image.

5.1.1 User Credentials

In the first step, C2C asks the user to provide credentials required to establish a connection with the target machine. Here user will provide a username, password or private SSH Key along with the IP Address of the target machine.

**Under the hood:** Using these credentials, it connects to the target machine using Secure Shell (SSH) and gathers the required information.

5.1.2 Target machine details

On the next step, C2C will display the gathered information like the type of machine (such as virtual or bare metal machine), operating system details like distribution, and version.

**Under the hood:** To find information about target machine, we perform following operations.

- We check the release files in etc directory for OS related information.
- We use `dmidecode` utility to find hardware information.
5.1.3 Clone machine creation

Meantime, in the background, a clone of the target machine is created by the provisioner as explained in section 4.2.3. The support for different on-premises and cloud-based platform can be easily extended because of its component-driven architecture.

Under the hood:

C2C provisioner supports creating virtual machine using Virtualbox\(^1\) and KVM\(^2\) based hypervisor and AWS\(^3\) cloud platform.

5.1.4 Running Processes

In this step, we show the extra processes running on the target machine when compared to the clone machine. The user can choose which all processes to containerize and send that information to the next step. We also show the open ports associated with each process.

Under the hood:

- We use \texttt{ps} utility to find the running processes.
- We use \texttt{netstat} utility to find open ports and associated process Id.

Thus, we create the mapping between the processes and open ports.

5.1.5 Resolved software packages

In this step, we show only the resolved packages. This information is derived as explained in the Section 4.1.4

**Under the hood:**

We use following commands to find dependencies for each package:

- `apt-cache depends` package_name - Debian-based OS
- `yum deplist` package_name - Red Hat-based OS .

5.1.6 Process-Package Relation

Next, we display the process-package relation, i.e., what package is required by each process to execute.

**Under the hood:**

We use following commands to find the package that is associated with the process:

- `apt-file search` process_name - Debian-based OS
- `yum whatprovides` process_name - Red Hat-based OS

5.1.7 Users & Groups and Environment variables

In this step, we show the extra users and groups created on the target machine when compared with clone machine. We also show the additional environment
variables that are present on the target machine. The user can decide what details should be associated with each process.

**Under the hood:**

We use `/etc/passwd`, `/etc/shadow`, `/etc/group`, `/etc/gshadow` files to gather users and groups related information.

We use `printenv` utility to gather environment variables.

### 5.1.8 Files and Configuration

As discussed in section 4.1.6, we show the files and configuration associated with each process and allow the user to alter the mapping suggested by our framework.

**Under the hood:**

We use `rsync` utility to get all the files which are not present on the clone machine.

**Command:**

```bash
rsync -avcn --compare-dest="/" --exclude-from='exclude_folders.txt'
user@target_machine:/ /different_files
```

- `compare-dest:` - Directory of the local machine to compare.
- `exclude-from:` - Exclude files and folders which should not be compared.

### 5.1.9 Create and Show Dockerfile

In this step, the user has an option to select the type of container (system or application) that should be created. Depending on the user’s selection, Dockerfile
is created as described in section 3.4. The generated Dockerfile is available to the user to perform any modification. Once User is satisfied with the generated file, it is converted to Docker Image.

**Under the hood:**

Based on the discovered application, if Docker Image is present on the Docker Hub, we use the existing image to create our Dockerfile. If not, we create Dockerfile from scratch. The Dockerfile is then converted to Docker Image using Docker’s build command.

We show how Dockerfile for application and system container is created.

**Application Container:**

To create Dockerfile for Application Container, we add the information gathered into different containers based on its association. We use the following two methods to create Dockerfile.

1. **Create Dockerfile from scratch:**

   Creating Dockerfile from scratch is using the base image of OS like CentOS, Ubuntu, Debian and then manually installing all the related packages along with its configuration:

   We used following steps to create Dockerfile:

   (a) Used OS information like distribution, version to get the base image.

   (b) Installed packages associated with the process.

   (c) Set the environment variables.
(d) Created user and groups.
(e) Copied process related files into the container.
(f) Copied the SSH configuration.
(g) Exposed the port(s) associated with the process.
(h) Set the working directory.
(i) Set the startup process for the container.

2. **Create Dockerfile using images available on Docker Hub:**

We used the process, OS and installed packages’ information to fetch already existing Docker Image from the Docker Hub.

We use the fetched image as a base image for our Dockerfile and follow the same steps from (d) - (i) as described above.

The official images available on Docker Hub are minimized and optimized specially for Docker.

**System Container:**

To create Dockerfile for System Container, we add all the information gathered into the single container instead of distributing it into different containers. System container will be just like VM with all the components of application inside it but without Operating System related files. We follow the same steps as described in section 5.1.9 for every process but inside the same container.

---

4[https://docs.docker.com/docker-hub/official_repos/#should-i-use-official-repositories](https://docs.docker.com/docker-hub/official_repos/#should-i-use-official-repositories)
Chapter 6

Case Study

In this chapter, we will evaluate the prototype of our automated framework. We will go through two different scenarios.

6.1 Scenario I

In this scenario, we will be containerizing our internal tool Virtual Infrastructure for Network Emulator (VINE). VINE is a cyber experimentation environment that helps to create, deploy, execute, and monitor different network scenarios [5].
6.1.1 Setup

To start containerization, we first installed VINE on Virtual Machine created using Vagrant. VINE was installed using the Ansible\(^1\) roles. Since VINE was built on the same machine, some packages were required only for compiling the application. Thus, for running the application, we may not need those packages. The decision to skip or add it to the container will be taken during the execution by the user. VINE was running on IP address 192.168.0.19.

6.1.2 Experiment Execution

Since C2C is step by step process, every step is explained as follows.

\(^1\)https://www.ansible.com/
**Step 1:** We provide IP address (192.168.0.19) and user credentials (username and SSH Key) of the target machine to establish a connection. The user has to provide either password or SSH Key for authentication. Once the user credentials are verified, the aggregator component as described in section 4.2.2 starts the information gathering in the background and the next screen is loaded with the OS and hardware details.

![Figure 6.2: Accepts the IP Address and user credentials of the target machine.](image-url)
**Step 2**: This step shows the gathered information about the target machine. Later, this information about the OS will be used in the creation of the container image.

**Figure 6.3**: Shows the gathered information about the target machine.
**Step 3**: We create the clone VM using the same cloud provider or provisioning software so that the clone is as similar as possible to the target machine. In this scenario, as shown in figure 6.3, Vagrant is used to provision the target machine using VirtualBox hypervisor. In this step, we can use the same provisioner to create the clone manually or automatically using this framework. The framework manages the automatically created VM, but for the manually created VM, user credentials must be provided. The manual handling provides more options to the user like the creating VM on another machine, ssh into the machine to check logs, etc.

![Figure 6.4](image.png)

**Figure 6.4**: Provides an option to use already created clone machine or create clone machine using Vagrant and VirtualBox.

---

2https://www.vagrantup.com/
Step 4: As soon as the clone is created, we gather all the required information from the clone machine and start deriving the information by calculating set difference over the information. The user can select the list of running processes for containerization. This functionality gives a user the ability to decide if the process needs to be containerized or not. All the selected processes will be containerized.

Figure 6.5: Shows the running processes that are not present on clone machine along with its associated ports.
Step 5: We show the process-package relation. For every process to execute, it requires the associated packages. Figure 6.6 show the process and its associated packages. The user can add and remove packages if required. We also have extra packages for which there was no direct relation to the running process because the running process might be using them internally. The extra packages are shown in figure 6.6 will be installed in all the containers if the user does not associate them with the available processes.

Figure 6.6: Shows the process-package relation.
**Step 6**: We show the files associated with each process along with the files and directories that our framework was unable to associate. In this case, the user can associate them manually. If the user is also not able to associate the files with the process, then the remaining files will be added to all the containers.

---

**Figure 6.7**: Shows the process-file relation.
**Step 7**: Now we have all the information to create the Dockerfile. The user has two options:

- Use the Docker Image from Docker Hub if available
- Create Dockerfile from scratch and then create Docker Image.

Based on the use cases and advantages & disadvantages discussed in the section 2.4, the user can also decide the type of container (such as application container or system container) to be created.

![Docker Container Image Options](image.png)

**Figure 6.8**: Shows the options to create container image.
6.1.3 Evaluation

We have created the application and system containers as described in the section 5.1.9. To accomplish this, we generated Dockerfile from scratch as well as used the available Docker Hub images. Listing A.1 shows Dockerfile generated from scratch for java process. Similarly listing A.2 shows Dockerfile created for tomcat process and listing A.3 shows Dockerfile created for MySQL process.

The following table shows the size of Docker Images created from Dockerfile for each process. We have used official Docker Images such as Java (java:openjdk-8-jre), Tomcat ("tomcat:7-jre8-slim") and MySQL (mysql:5.7) to create Dockerfile.

<table>
<thead>
<tr>
<th>Processes</th>
<th>Scratch (MB)</th>
<th>Docker Hub (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>java</td>
<td>414</td>
<td>323</td>
</tr>
<tr>
<td>tomcat</td>
<td>286</td>
<td>242</td>
</tr>
<tr>
<td>mysql</td>
<td>484</td>
<td>373</td>
</tr>
<tr>
<td>total</td>
<td>1184</td>
<td>938</td>
</tr>
</tbody>
</table>

**Table 6.1:** Shows the size of Docker Images created for VINE application.

1. **Application Containers:**

   Size of all the three images created from scratch = 1184 MB

   Size of all the three images created using Docker Hub Images = 938 MB

2. **System Container:**

   Size of Docker Image = 656 MB

3. **Target Machine:**
Size of Virtual Machine = 1933 MB

The overall size of application containers is around 45% less than the target machine’s size. Another significant advantage of application container is that it allows component-based scaling. Thus we can scale any required container of our application.

The overall size of system container is 66% less than the target machine’s size.

For this scenario, the manual process took around 43 minutes to create container image. However, it took approximately 8 mins using the automated framework. The automated framework was 5 times faster than the manual process.

6.2 Scenario II

In this scenario, we will be containerizing an application created using MEAN stack. The MEAN stands for MongoDB (NoSQL Database), Express.js (Server-side Framework), AngularJS (Client-side Framework), and Node.js (Execution environment for Client-side and Server-side framework).

![Diagram of MEAN stack](image)

**Figure 6.9:** A high-level view of the MEAN stack application.
6.2.1 Setup

We have installed an open-source MEAN stack application from Github repository. The machine was created on AWS\(^3\).

6.2.2 Experiment Execution

We followed the same steps as described in Scenario I (Section 6.1). Our framework detected that there were three running processes (such as Node.js to run AngularJS, Node.js to run ExpressJS, and MongoDB). Each process was listening on its respective TCP port. As per the application container’s design, three different containers should be created. However, Node.js is the underlying environment used to run both AngularJS and ExpressJS, we select both the process as part of a single container and associated packages and file and configuration manually from the screen using drag and drop interface. Thus, user’s intervention helped us to create more accurate container image.

Our framework tried to create separate containers for each process as per the design, but we had to override it through GUI to create more refined container image.

\(^3\)https://aws.amazon.com/
6.2.3 Evaluation

We have created the application (hybrid) and system containers. Following table shows the size of Docker Image created from Dockerfile for each process. We have used official Docker Images such as NodeJS (nodejs:carbon) and MongoDB (mongo:latest) to create Dockerfile.

<table>
<thead>
<tr>
<th>Processes</th>
<th>Scratch (MB)</th>
<th>Docker Hub (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodejs</td>
<td>329</td>
<td>673</td>
</tr>
<tr>
<td>mongo</td>
<td>314</td>
<td>368</td>
</tr>
<tr>
<td>total</td>
<td>643</td>
<td>1041</td>
</tr>
</tbody>
</table>

Table 6.2: Shows the size of Docker Images created by both the methods.

1. **Application Containers:**

   Total Size of both the images created from scratch = 643 MB

   Total Size of both the images created using existing Docker Hub Images = 1041 MB

2. **System Container:**

   Total Size of Docker Image = 573 MB

3. **Target Machine:**

   The total size of Virtual Machine = 1467 MB

The overall size of hybrid application containers (created from scratch) is 56% less than the target machine’s size.
The overall size of system container is 61% less than the target machine’s size.

For this scenario, the manual process took around 49 minutes to create container image. However, it took approximately 15 mins using the automated framework. The time required to create container image increased as compared to Scenario I because we had to decide the association of packages, file, and configurations with each process manually.

6.3 Scenario Observations:

As of today, both the open-source tools Image2Docker and VM2Docker, are having open issues. So we are not to run our scenarios using these tools to compare our results. However, based on their framework and limitations discussed in the literature review, we can say that our framework has covered them along with following options:

- Provides GUI based step by step approach to containerize application. It provides an option to alter the derived information and pass it to the next step.

- Uses Secure Shell (SSH) Port for communication with target and clone machine.

- Gathers information parallely using thread.

- Support Multi-Container Orchestration, i.e., Application Containers as well as System Container.
• Supports networking between container by bridging them to the same host network.

• Installs only the associated packages.

• Information about users, groups, environment variables and SSH configuration is used to create Linux container Image along with the process and package information.

• The derived information is used to create vendor independent container image.

Along with these advantages, there are few issues with our framework.

• Our framework is suitable only for applications running on the Linux based operating system.

• If our framework is not able to associate the information gathered, user’s intervention may be required. However, this helps to create an accurate container image rather than adding everything in every container.

Thus, our prototype provides easy and user-friendly approach along with promising results. We can easily extend our prototype to support more OS, vendor and cloud providers.
Chapter 7

Conclusions

In this thesis, we reviewed the existing tools available for application containerization and identified their shortcomings. Based on these limitations, we proposed our approach to containerize Linux-based applications along with a component-driven automation framework. This framework uses the power of automation and human intelligence to gather and consolidate all the information required to create the container image. Unlike other approaches, our approach is platform, vendor, cloud-provider or hypervisor independent. We have also created a prototype implementation of this framework to show the feasibility and effectiveness of our approach.

Our framework eliminates the error-prone manual process of information gathering, thus providing all the information required to create Linux container image. From the scenarios illustrated in the case studies, we speculate that our framework is intuitive, user-friendly and performant. In future, we hope to integrate artificial
intelligence and machine learning with our the framework, so that it can learn from the user selections and automatically provide suggestions to the user.
Bibliography


Appendix A

Generated Docker files

```bash
FROM ubuntu:16.04
RUN apt-get update && apt-get install -y default-jre && apt-get clean && rm -rf /var/lib/apt/lists/*
RUN useradd -ms /bin/bash vine
USER vine
COPY vine /usr/local/vine
WORKDIR /usr/local/vine
EXPOSE 9400
ENTRYPOINT ["/usr/bin/java.jar/vine.jar.conf/vine.properties"]
```

**Listing A.1:** Dockerfile generated for Java Process.

```bash
FROM ubuntu:16.04
RUN apt-get update && apt-get install -y tomcat7 && apt-get clean && rm -rf /var/lib/apt/lists/*
RUN useradd -ms /bin/bash vine
```

```
USER vine
COPY vine /var/lib/tomcat7/webapps/vine
COPY vine.war /var/lib/tomcat7/webapps/vine.war
ENV CATALINA_BASE: /usr/share/tomcat7/
ENV CATALINA_HOME: /usr/share/tomcat7/
ENV CATALINA_TMPDIR: /usr/local/tomcat/temp
ENV CLASSPATH: /usr/local/tomcat/bin/bootstrap.jar:/usr/share/tomcat7/bin/tomcat-juli.jar
ENV PATH="/usr/share/tomcat7/bin/:${PATH}"
WORKDIR /usr/share/tomcat7/bin/
EXPOSE 8080
CMD ["catalina.sh", "run"]
```

### Listing A.3: Dockerfile generated for MySQL Process.

```
FROM ubuntu:16.04
RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install -y mysql-server && apt-get clean && rm -rf /var/lib/apt/lists/*
COPY mysql /var/lib/mysql
COPY log_bin_trust_function_creators.cnf /usr/local/etc/log_bin_trust_function_creators.cnf
EXPOSE 3306
ENTRYPOINT ["/usr/sbin/mysqld"]
```