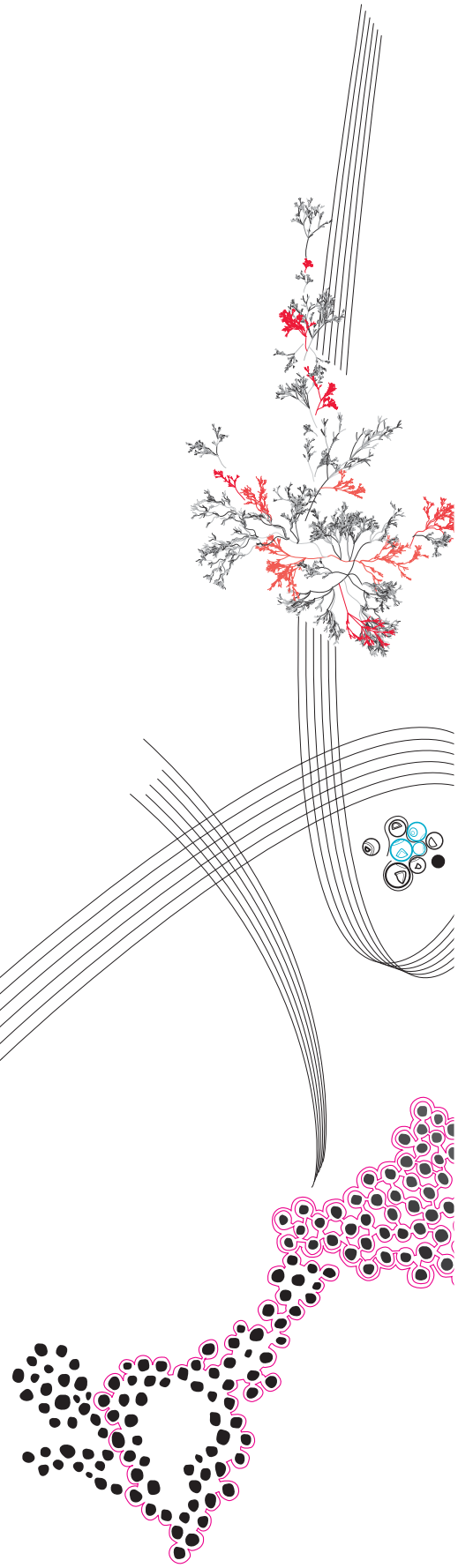

MSc Computer Science
Final Project

Analysis of automated Virtual Machine generation and automation around system testing at TKH Airport Solutions.

Tom Grooters

Supervisors: Petra van den Bos (UT), Johan Foederer (TKH-AS)

December, 2023

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

1 Introduction **1**
- 1.1 Overview . . . 1
- 1.2 Motivation . . . 2
- 1.3 Research questions . . . 3

2 Background **4**
- 2.1 TKH Airport Solutions . . . 4
  - 2.1.1 Confidential information . . . 4
  - 2.1.2 AGL & CEDD® . . . 4
  - 2.1.3 Components . . . 5
- 2.2 Virtualization terms . . . 6
  - 2.2.1 Virtualization . . . 6
  - 2.2.2 Virtualization Technology . . . 6
  - 2.2.3 Hypervisor . . . 6
  - 2.2.4 Virtual Machine . . . 6
  - 2.2.5 Container . . . 6
  - 2.2.6 Virtual Image . . . 7
  - 2.2.7 Image Generation . . . 7
  - 2.2.8 Host Machine . . . 7
- 2.3 Virtualization Technologies . . . 7
  - 2.3.1 VirtualBox . . . 8
  - 2.3.2 KVM . . . 8
  - 2.3.3 Docker . . . 8
- 2.4 Testing infrastructure . . . 9
  - 2.4.1 Test system setup . . . 9
  - 2.4.2 System Test suites . . . 10
  - 2.4.3 Jenkins pipeline . . . 10
- 2.5 Initial process and setup . . . 10
  - 2.5.1 Image generation . . . 11
  - 2.5.2 System testing . . . 12
- 2.6 Restart Issues VirtualBox . . . 12
  - 2.6.1 Severity . . . 13

3 Related work **15**
- 3.1 Image generation . . . 15
- 3.2 Performance . . . 16
  - 3.2.1 Runtime performance . . . 16
  - 3.2.2 Build performance . . . 16
  - 3.2.3 Startup performance . . . 17

3.3 System testing . . . 17
3.4 Pipeline . . . 17
3.4.1 CI/CD . . . 17

4 Methodology **19**
4.1 Overview . . . 19
4.2 Technology selection . . . 20
4.3 Image Generation Implementations . . . 21
4.4 Pipeline improvements . . . 21
4.5 Results and conclusion . . . 21
4.6 Metrics . . . 22
4.6.1 Data collection and scripting . . . 22
4.6.2 Median data . . . 23
4.6.3 Stability . . . 23
4.6.4 Automation . . . 24
4.6.5 Performance . . . 25
4.6.6 Company impact . . . 26

5 Technology selection **27**
5.1 Criteria and requirements . . . 27
5.1.1 Boot reliability . . . 27
5.1.2 Multiple network interfaces . . . 27
5.1.3 Full automation/scripting . . . 28
5.1.4 Platform support . . . 28
5.1.5 Partitions . . . 28
5.1.6 Cost . . . 28
5.1.7 Summary . . . 28
5.2 Virtualization Technologies . . . 29
5.3 Decision Matrix . . . 29
5.3.1 Criteria scoring . . . 29
5.3.2 Decision matrix and argumentation . . . 30

6 Image Generation Implementations **32**
6.1 Base image . . . 32
6.2 Packer . . . 32
6.3 Docker . . . 33
6.4 Cloud-Init . . . 34
6.5 Virt-Customize . . . 35
6.6 Ansible . . . 35

7 Pipeline redesign **36**
7.1 Overview . . . 36
7.1.1 Setup host machine . . . 36
7.1.2 Deploy Basic Virtual . . . 37
7.1.3 Run system test . . . 37
7.2 Changes . . . 37
7.2.1 No stash and unstash Virtual Images . . . 37
7.2.2 Delete old Virtual Images . . . 38
7.2.3 Start Virtual Machines . . . 38
7.2.4 Headless starting . . . 38

7.2.5 Deploy Master AM configuration . . . 38
7.2.6 Network interfaces management . . . 39
7.2.7 Minimizing higher privilege . . . 39
7.3 Python version validation . . . 39
7.4 Full automation . . . 39

8 Results **40**
8.1 Stability . . . 40
8.1.1 Image generation . . . 40
8.1.2 Pipeline . . . 41
8.1.3 Summary . . . 42
8.2 Automation . . . 43
8.2.1 Image generation . . . 43
8.2.2 Pipeline . . . 43
8.2.3 Summary . . . 44
8.3 Performance . . . 44
8.3.1 Duration . . . 45
8.3.2 Resource usage . . . 46
8.3.3 Pipeline . . . 48
8.3.4 Summary . . . 49
8.4 Company impact . . . 50
8.5 Overview . . . 51

9 Conclusion **53**
9.1 How can we improve the process of generating new system images? . . . 53
9.2 How can we improve the testing pipeline? . . . 54
9.3 How can we improve the technical process around the system testing performed at TKH Airport Solutions? . . . 54
9.4 Future work . . . 55

A Graphs **58**
A.1 VirtualBox reliability . . . 58
A.2 Resource usage different Image Generation processes . . . 59

Abstract

System testing is an important step in validating the correct workings of larger systems. One way to reduce cost and shorten testing times is by using a virtual system instead of the physical system.

In this case study in collaboration with TKH Airport Solutions we will be looking at what is the best Virtualization Technology to run these systems in. An important aspect of this decision is the techniques we can use to rapidly build new versions of these systems. For this, we need a reliable and automated system such that the generation process of the Virtual Machines can be automated.

Next to the generation process, we will also be looking at the deployment and how to run the system tests using a Jenkins pipeline, discovering what changes and extensions are needed to transform into a fully automated build and test cycle using virtualization.

Keywords: Virtualization Technology, System Image Generation, System testing, Test automation, Pipeline automation, VirtualBox, KVM, Docker, Containers

Chapter 1

Introduction

During this research, we will aim to discover an effective way to generate and test a virtualized version of a system consisting of multiple machines. We will be exploring different techniques to generate the Virtualized machines, as well as methods on the setup for and how to perform system testing on these systems.

In this research, we will be collaborating with TKH Airport Solutions. They are "an innovator in airfield ground lighting (AGL), providing a complete range of LED AGL products." Their ground lighting is controlled through *Contactless Energy & Data Distribution* CEDD®[1]. In order to manage the ground lighting and the CEDD network there are several systems in place spanning the CEDD AGL System. As with any software, especially in the case of airports, this requires thorough testing to ensure correct working under many conditions. In the case of TKH Airport Solutions, this is, among other testing, achieved through system testing.

This system testing is performed on an emulated version of the CEDD AGL system which has the different components running in separate Virtual Machines. Together these Virtual Machines form the whole of a virtual setup dubbed the Virtual Basic setup. The pipeline that has been set up is there to take a new release version of these Virtual Machines and prepare/set up a physical machine of the developer to be able to run the system tests.

In this research, we aim to look at alternative technologies to run these Virtual Machines, ways to generate the Virtual Machines themselves and look at ways we can improve the pipeline to be fully automated and reliable.

1.1 Overview

As is common this thesis is split up into multiple chapters each about a certain aspect of the overall research project. We will start with the Introduction in which we will introduce the topic of this thesis, the motivation and the research questions. After this, we will continue to the Background in which we will introduce the company (TKH Airport Solutions) with which this research was a collaboration. We will also introduce some of the terminology and technologies we will be using throughout this thesis and we will give more context about the CEDD AGL systems at TKH Airport Solutions. After this we will be looking at the Related work, in this chapter we will be looking at other works covering Virtualization Technologies, zooming in on the relative performance. We will also take a look at system testing and pipelines.

Once we have this baseline of knowledge set we will look at the Methodology, which will

[1] https://www.tkh-airportsolutions.com/cedd-airfield-ground-lighting

explain how we tackle this research. Our plans for implementing the different techniques and how to measure this to get to a satisfying and concrete result.

The selection of which technologies to analyse will be made in Technology selection, then in Image Generation Implementations we will implement the various technologies and give our experiences. In chapter Pipeline redesign we will look at the changes made to the pipeline to improve the experience.

After having a good overview of all our efforts and changes we will be presenting the numbers in Results. In this chapter, the values for our measured metrics will be presented. Finally in Conclusion we will be concluding our research, presenting our recommendation and looking at possible future work in areas we saw more potential or could not do enough research ourselves.

1.2 Motivation

This research aims to take a closer look at different ways and methods to generate Virtual Machine images. More specifically we wish to look at ways to set up the CEDD AGL system. This includes a mix of standard software packages, provided by the Operating System, along with custom software produced by the company. There are several papers looking into the performance differences between different Virtualization Technology but we have not been able to find much related to the setup and installation of the Virtual Image used within these virtualization techniques.

The main topic of this case study will be research towards the generation of these Virtual Images. We aim to find a good balance between flexibility, reliability and performance while trying to play into the knowledge and expertise the company already has onboard to minimize the time needed to learn new tooling stacks. During this research, we also look at how this combines with the process of system testing which is done after the Virtual Images are generated. This is done by working on the existing testing pipeline and streamlining this process. We will look at automating several tasks which are now performed manually and increasing the overall success rate of the pipeline process.

Currently, the system testing is done, for a large part, by using Virtual Machines which are being run inside VirtualBox, which is a tool to manage and run these machines. The CEDD AGL systems running on the Virtual Machines are then in turn being tested through a Jenkins pipeline using Robot Framework

The Virtual Machines are then in turn being prepared and the CEDD AGL system is tested using a Jenkins pipeline, which allows us to run a bunch of actions/scripts to complete a larger process. These actions prepare the Host Machine by installing the required software and libraries and setting up the Virtual Machines accordingly. After this the system tests are being run with Robot Framework[2].

Although this process is functional we notice that a lot of the pipeline runs fail, often due to issues with the stability. A lot of these issues seem to be correlated to VirtualBox, this is the main reason why we want to research the alternatives to VirtualBox. In the process, we also wish to find if the alternatives gave a better suitability for the company compared to VirtualBox. And despite the stability, if there is more reason to switch to a different technology or not. Additionally, we will be improving the pipeline process itself, adding more automation and improving error reporting if an issue may arise.

[2] https://robotframework.org/

1.3 Research questions

We will be doing this by answering the research questions we propose in this section. As the main question we will be focused on improving the overall process, the generation of the Virtual Image and the running of the Jenkins pipeline. To do this we propose the following Main Research Question

MQ How can we improve the technical process around the system testing performed at TKH Airport Solutions?

To answer the different aspects of this quite generic question we subdivide into two sub-questions;

RQ1 How can we improve the process of generating new system images?

With this question we will take a look at the process of creating the Virtual Images used by several Virtualization Techniques. We will compare different processes and implementations and measure various metrics on their reliability, automation potential, performance and the impact they may have on the company.

RQ2 How can we improve the testing pipeline?

With this subquestion we will focus towards the later stages of the testing, after generating the images they need to be correctly set up and tested. This subquestion will look at what changes we may make to the existing pipeline to increase, especially its reliability and attempt to make it a fully automated hands-off process.

Chapter 2

Background

To give some light on the terminology used throughout this research, and also give a wider context, we will first explain something about the company we are working with (TKH Airport Solutions). This will be done to give more context to where this research fits in with their practices. After this, we will explain a variety of terminology and dive deeper into the context of this research giving more context about the company processes we will be interfacing with as well as more background on the problem that gave rise to this research.

2.1 TKH Airport Solutions

TKH Airport Solutions is the collaboration partner for this research project, they describe themselves as:

"TKH Airport Solutions is an innovator in airfield ground lighting (AGL), providing a complete range of LED AGL products. We build upon the know-how from a long and successful tradition of pioneering developments in the AGL and connectivity industry." from TKH Airport Solutions Company Profile[1]

2.1.1 Confidential information

As with many businesses, also at TKH Airport Solutions, some of their information is confidential and cannot be shared. This includes but is not limited to, the exact internal workings of their systems and the source code and builds of many of their software. To still produce a relevant and exciting thesis, we will abstract away from these details. Since our results mostly focus on the process and outside tooling and the results do not rely on the exact programs being installed, we do not believe it is needed to give a comparable workload.

2.1.2 AGL & CEDD®

Airfield ground lighting (AGL) is, among other things, the lighting equipment on and around an airfield intended for pilots and ground staff to navigate an airplane. This includes lighting on and around runways and taxiways.

Contactless Energy & Data Distribution (CEDD®) is "a smart, safe and sustainable airfield ground lighting technology. Energy and data transport are combined in a single

[1] https://www.tkh-airportsolutions.com/company/company-profile

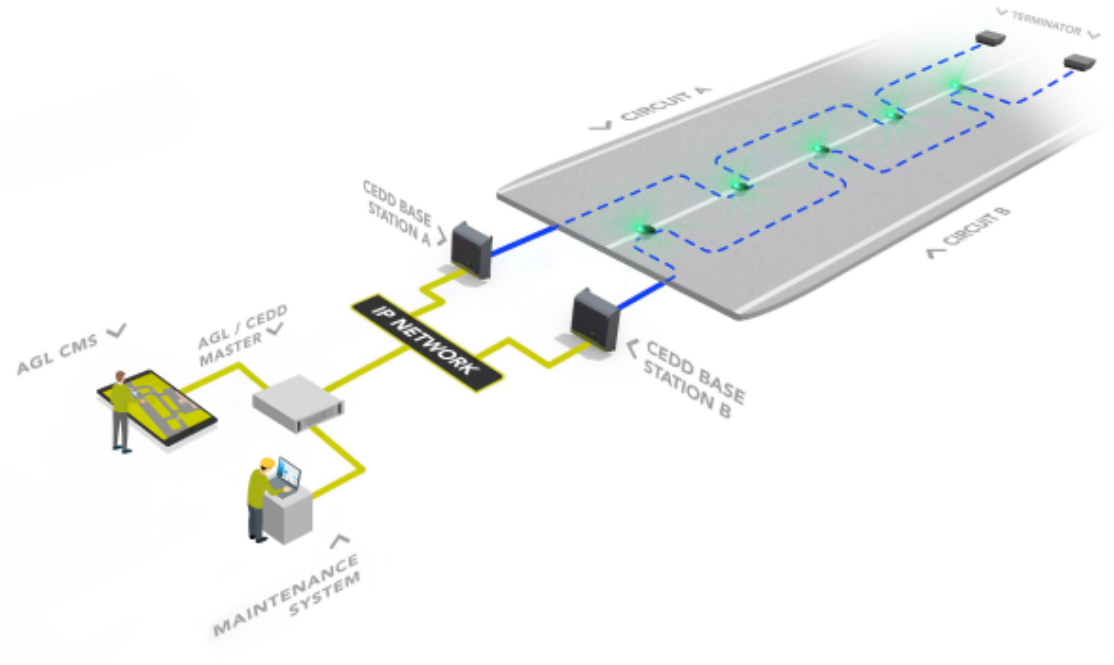

FIGURE 2.1: Simplified example of a setup as in production, here the Master CMS and AM are pictured together as *AGL / CEDD Master*.

cable. Airfield lights receive their power and data by means of induction, and can therefore easily and safely be installed or replaced without making electrical contact."[2]

2.1.3 Components

The control system behind the CEDD® AGL technology consists of three systems, one Master AM (Asset Management) and two Master CMSs (Control and Monitoring System) which we will refer to as *the Masters*. These three masters talk to the Base stations which in turn control a set of lights [3]. together these compose the *components* of the *CEDD AGL System* A simple representation of a setup can be found in figure 2.1. In this figure the different Masters are combined into one *AGL / CEDD Master*.

In our context, we are working with a fully virtualized version of this CEDD AGL System where the different masters but also the Basestations are virtualized into VirtualBox Virtual Machines. These Virtual Basestations have in turn a set of lamps in their memory without a physical counterpart.

The communication between the CEDD AGL components happens over a bonded network interface[4] in broadcast mode, in the image represented by *IP network*. This results in all data packets being sent over the network getting sent on each of the interfaces, this makes an easy way to add redundancy to the networking such that if one of the two networks gets disconnected, or there is a different issue, data is still able to reach the other components.

[2] https://www.tkh-airportsolutions.com/cedd-airfield-ground-lighting

[3] https://www.tkh-airportsolutions.com/airfield-products/cedd-components/cedd-agl-network-layout.html

[4] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_networking/configuring-network-bonding_configuring-and-managing-networking

2.2 Virtualization terms

Virtualization and the accompanying terms may get confusing as they are used very differently in different contexts. We will be describing these to give a clearer overview of how we will use them in this thesis and to give the less experienced reader a better understanding of these terms. For more information and context on these terms in a more general application the footnotes contain links to webpages with more details.

2.2.1 Virtualization

Virtualization[5] is a technology that allows for the creation of an abstraction layer over physical hardware, enabling multiple operating systems (known as "guests") to run concurrently on a single physical machine (the "host"). This is achieved through the use of a software layer known as a Hypervisor.

2.2.2 Virtualization Technology

Virtualization Technologies are the different technologies that implement or manage the Virtualization, these are the different types of hypervisors available. Some examples of these technologies are VirtualBox, Docker and KVM.

2.2.3 Hypervisor

The Hypervisor[6] is a software application that runs on the Host Machine. It manages the distribution of resources such as CPU time, memory, and I/O between the guest operating systems, ensuring that each guest has access to the resources it needs without interfering with the operation of other guests. The hypervisor allows for efficient utilization of hardware resources, as multiple workloads can be run on a single machine without the need for additional physical hardware.

2.2.4 Virtual Machine

Virtual Machines[7] (VMs) is one such type of guest and are a key component of this technology. A VM is essentially a software emulation of a physical computer, running an operating system and applications just like a physical computer. VMs are isolated from each other, providing security and fault isolation.

2.2.5 Container

Containers[8], like virtual machines, are a form of virtualization technology, but they operate at the application level rather than the operating system level. A container packages an application along with its runtime environment, including the libraries and other dependencies it requires to run. This ensures that the application will run consistently across different computing environments.

A simplified comparison would be that a Virtual Machine is an entire computer running inside another computer. Whereas with containers multiple computers are running alongside each other on the same hardware.

[5] https://www.ibm.com/topics/virtualization
[6] https://www.ibm.com/topics/virtualization#Hypervisors
[7] https://www.ibm.com/topics/virtual-machines
[8] https://www.redhat.com/en/topics/containers/whats-a-linux-container

2.2.6 Virtual Image

A Virtual Image[9] (also known as a System Image or Container Image) is a single static unit that contains a snapshot of the operating system, configuration, software, and setup files necessary to run an instance of an (operating) system or a container. It serves as a template from which individual virtual machines or containers can be created. When a VM or container is launched or imported, it's based on such an image, which defines its operating system and initial state.

2.2.7 Image Generation

Image Generation is our own definition and refers to the process of creating these virtual images. The Image Generation involves using scripts or other automation tools to install the necessary operating system and software components onto an image file. This file can then be used to create new VMs or containers. Automation ensures that every image is created with the same configuration, reducing errors and inconsistencies that can occur with manual setup. Tools like Packer[10] or DockerFile[11] are often used for automated image generation.

2.2.8 Host Machine

The Host Machine or Host System is the system that is used to run the virtualization tasks. This is often a physical machine, as opposed to the virtualized environments focused on earlier. In our specific case, this was a machine with the following specifications:

- **Machine:** Dell Optiplex 9020 MT
- **CPU:** Intel I7-4790
- **GPU:** (Integrated) Intel HD graphics 4600
- **Memory:** 4x 4GB DDR3 1600MHz (16GB total)
- **Storage:** Samsung SSD 860 EVO 500GB
- **Operating System:** Linux
  - **Distro:** Ubuntu Mate
  - **Release version:** 22.04.3 LTS
  - **Architecture:** x86_64
  - **Kernel**: 5.15.0-86

2.3 Virtualization Technologies

In this thesis, we will talk about several Virtualization Technologies. To better understand what each of these technologies entails, this section will explain the three main ones we will focus on, VirtualBox, KVM and Docker. Although there are several other Technologies, we have opted to put our focus on these three.

[9] https://www.sciencedirect.com/topics/computer-science/virtual-machine-image
[10] https://www.packer.io/
[11] https://docs.docker.com/engine/reference/builder/

2.3.1 VirtualBox

VirtualBox[12], or *Oracle VM VirtualBox* in full, is a type-2 hypervisor for x86 virtualization. It was created in 2007 by InnoTek Systemberatung GmbH, a German company. In 2008, Sun Microsystems acquired InnoTek, and in 2010, Oracle Corporation acquired Sun Microsystems.

VirtualBox has support for multiple operating systems including Windows, macOS, Linux, Solaris, and OpenSolaris. It allows for the creation and management of guest virtual machines running different operating systems like Windows, Linux, macOS and many more.

VirtualBox provides a user interface called VirtualBox Manager for creating, configuring, and managing virtual machines. It also supports features such as snapshots for saving the state of a VM at any point in time and reverting to that state when needed. VirtualBox is widely used in enterprise settings for running different operating systems on a single machine for testing or development purposes.

2.3.2 KVM

Kernel-based Virtual Machine (KVM)[13] is an open-source Virtualization Technology built into the Linux kernel. KVM allows Linux to operate as a hypervisor that enables the host machine to run multiple, isolated environments, called guests or VMs, similar to VirtualBox.

Every VM is implemented as a regular Linux process, scheduled by the standard Linux scheduler, with dedicated virtual hardware like a network card, graphics adapter, CPUs, memory, and disks. KVM was first announced in 2006 and merged into the mainline Linux kernel version a year later. Because KVM is part of existing Linux code, it immediately benefits from every new Linux feature, fix, and advancement without additional engineering.

KVM requires a processor with hardware virtualization extensions. It has been ported to other operating systems such as FreeBSD and Illumos in the form of loadable kernel modules. KVM supports hardware-assisted virtualization for a wide variety of guest operating systems including other Linux systems, BSD, Solaris, Windows, macOS, and many more.

It supports running on Windows through Windows Subsystem for Linux (WSL for short). This is made possible through the nested virtualization of WSL2, which allows KVM to run on a Linux kernel hosted on Windows. This integration allows for the execution of Linux-based virtual machines directly on a Windows system, expanding the capabilities of both KVM and WSL.

2.3.3 Docker

Docker[14] is an open platform for developing, shipping, and running applications. It employs OS-level virtualization to deliver software in packages known as containers. These containers are lightweight and encapsulate everything needed to run the application, thereby eliminating dependencies on the host system.

Docker's primary advantage is its ability to separate applications from infrastructure, enabling rapid software delivery. This separation allows developers to manage infrastructure in the same way they manage applications. By leveraging Docker's methodologies

[12] https://www.virtualbox.org/
[13] https://linux-kvm.org/page/Main_Page
[14] https://www.docker.com/

for shipping, testing, and deploying code, developers can significantly reduce the delay between writing code and running it in production.

The Docker platform provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow many containers to run simultaneously on a given host. Docker provides tooling and a platform to manage the lifecycle of these containers.

Docker streamlines the development lifecycle by allowing developers to work in standardized environments using local containers which provide your applications and services. Containers are great for continuous integration and continuous delivery (CI/CD) workflows.

Docker's container-based platform allows for highly portable workloads. Docker containers can run on a developer's local laptop, on physical or virtual machines in a data centre, on cloud providers, or in a mixture of environments. Docker's portability and lightweight nature also make it easy to dynamically manage workloads, scaling up or tearing down applications and services as business needs dictate, in near real-time.

In conclusion, Docker is a powerful tool for creating isolated, reproducible environments for software development and deployment. Its use of container technology simplifies many of the challenges associated with software development, including dependency management, environment consistency, and deployment speed. As such, Docker has become an integral part of modern software development workflows.

2.4 Testing infrastructure

At TKH Airport Solutions there are several test setups and sets of tests. These setups offer varying degrees of complexity and dedicated hardware accordingly to test the real communication between them. Similarly, there are also several sets of tests, also called test suites, to test the deployment and conformity of the CEDD AGL systems that are under test. In this section, we first introduce the different system setups and then introduce the test suites.

2.4.1 Test system setup

There are three different (CEDD AGL) system setups that are relevant in our situation, besides there are also demo setups for conferences and a setup at Twente Airport[15]. In increasing order of complexity these are the following:

Virtual Basic setup

The most basic setup is a fully virtualized setup and nicknamed accordingly as *Virtual Basic*. This setup consists of several Virtual Machines set up using VirtualBox and (usually) runs entirely on the developer's system or a dedicated test PC. This setup will also be the focus of our research as this is the setup we wish to improve the testing and generation of said Virtual Machines for.

Small physical setup

Next, we have the *Small physical setup* this setup consists of one physical Basestation and a VirtualBox setup with the Masters. The Basestation is connected to several lamps to test functionality between the lamps and the Basestation.

[15] https://www.tkh-airportsolutions.com/career/locations

Large Hybrid setup

Finally, we have the *Large Hybrid setup*, this is a full-scale setup with 2 physical Basestations and connected lamps, complemented with many virtual Basestation and virtual lamps to reach full scale. The two CMS Masters are running virtualized on their own physical machines, with the Master AM being on the CMS 1 machine, housed in a server rack.

2.4.2 System Test suites

Similar to how there are multiple setups there are also several standardized test suites that are being used. These test suites are designed through Robot Framework[16] which is an automation framework with a wide support base and many integrations.

Smoke test

The smoke test is a very basic test which quickly checks some basic functioning of the system by switching some lights. By doing this the configuration and communication between all components can be tested in a very efficient manner before more time is spent on larger tests which often also require a more lengthy setup to ensure the pre-conditions are met.

Release test

The release test runs all available and applicable tests for the given test system. This will test a large part of the functionality of the systems and their interoperability. Due to the extensive nature of this test, a single run also takes quite a lot of time. On the Basic Virtual setup, it can take over 1 hour to complete.

Endurance test

As the name implies this test performs an endurance test operation. Using this setup a subset of the available tests can be repeatedly run for an extended time. This is to ensure correct operation over a larger amount of time.

2.4.3 Jenkins pipeline

TKH Airport Solutions makes use of a Jenkins pipeline[17] to set up the Virtual Basic setup. Through this method, a lot of the manual work for changing versions and setup is automated. Additionally through the pipeline, a quick Smoke test is run to verify the correct setup of the CEDD AGL system.

2.5 Initial process and setup

There are two areas we want to do research for at TKH Airport Solutions. First, we have the generation of the system images. These will be a virtual form of the masters and the Virtual Basestation as they are mentioned in section 2.1.3. And secondly, we have the running of the system tests using the Jenkins pipeline. Figure 2.2 shows a simplified overview of how the process ties together.

[16] https://robotframework.org/
[17] https://www.jenkins.io/doc/book/pipeline/#overview

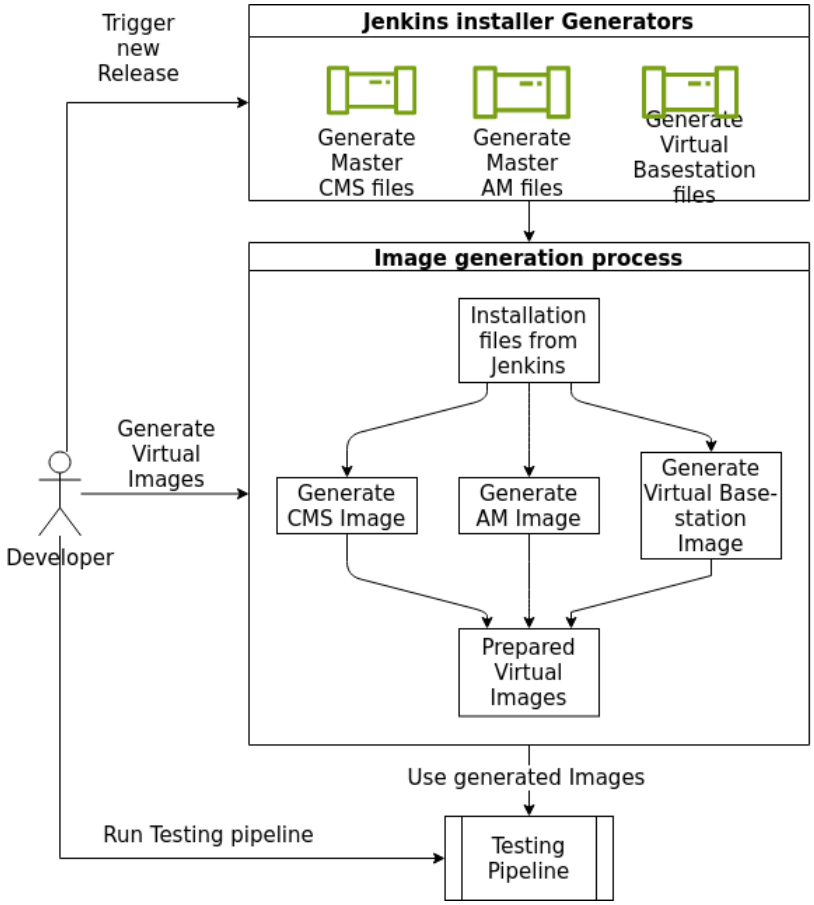

FIGURE 2.2: Overview of the whole process including Image Generation and the system testing pipeline.

2.5.1 Image generation

Packer is currently being used in the Image Generation process to create the VirtualBox images. To generate the images, the software that is to be installed on the Virtual Images is first compiled and bundled through Jenkins pipelines. After they are bundled they are all stored on the release area (a shared network point) which the developers can access.

Once these bundles are made it can then be used by a developer to generate a system image of the desired component (one of the CEDD AGL Components), this is done through Packer. First Packer makes a base image with Ubuntu 22.04 installed along some standard packages. Once this base image is made a snapshot is made to revert to the base. From this snapshot one by one the CEDD AGL Components are being installed, and exported and finally the image is reverted to the old snapshot for the next system to be installed.

To install the component Packer fetches and uploads the bundled software along with several configuration files which are stored alongside the Packer configuration. Then Packer puts these files in the correct locations with the right permissions and the installer scripts are run to install the software components on the images.

When a component is completely set up it gets exported through VirtualBox, when all the components are done some last configuration changes are done on the files. After this process, they can all be bundled together as a new (internal) release ready for testing.

2.5.2 System testing

After the system images are generated they can be tested. When starting the research this was often done manually or through a simple pipeline with much room for improvement. Many steps had to be manually prepared before the pipeline could be run. And, often due to VirtualBox stability, there were multiple points where the pipeline could fail or become unrecoverable.

This process involved preparing the machine by removing old images and settings. After this, the images are imported into VirtualBox and VirtualBox is set up for networking. Once this is done a clone is made of the Master CMS image which is converted to a Master CMS 2. This is done because cloning and changing the configuration for a Master CMS is more efficient than generating two almost identical images.

Once all images are ready the Virtual Machines are started and the configuration is uploaded. First, the Virtual Basestation system is configured with a small number of Basestations. After this configuration is done the overall network configuration is set up through the Master AM. This is done through a robot framework[18] suite. Once all the configuration is done a Smoke test is run to validate the basic working of the CEDD AGL System. This quickly ensures that the most basic of operations is possible such that a Release test can be run to get a final test report which a test engineer can use in their validation of the new (internal) release.

2.6 Restart Issues VirtualBox

One of the main reasons to look for an alternative strategy for building the system images automatically is the instability of VirtualBox. Throughout a complete configuration and test run using the basic pipeline, there are a large number of issues that can occur causing failures of this pipeline. Partially due to this reason some of the developers do not make use of this existing pipeline. We have however identified that a large number of the issues with this pipeline, and other issues with testing the CEDD AGL System, are due to the instability of VirtualBox.

When a Virtual Machine is started or rebooted, which happens multiple times during a full test run, there is the potential to end up in an erroneous state. The most noticeable effect of the issue was that the boot process would halt partway through with the timing information, which normally displays as the time since start in seconds, at a very large number (11 digits) as well as the line *RAS: Correctable Errors collector initialized.*. VirtualBox would however report the VM as running normally so there was no reliable way of detecting this issue. We could monitor services coming online, like SSH, to verify that the VM was booting. But we had no way of knowing if the boot took a little longer due to different factors or if the VM would never end up successfully booting.

Since this has a lot of unpredictable effects on our testing suite this is a major concern. Due to the CEDD AGL Systems not starting a large number of tests could fail. The fact that during the tests the CEDD AGL Systems are rebooted at certain times made it possible for this issue to start at almost any point during our test runs giving varying degrees of failed tests that could waste a lot of time. This makes it difficult to discern between a bug happening or if a VM did not (re)start correctly.

[18] https://robotframework.org/

	CMS1	CMS2	AM	VBase	Overall
Total failures	138	137	165	150	590
Failure Percentage	1.84%	1.82%	2.20%	1.99%	11.83%
Failure period	54.4	54.8	45.5	50.1	8.5

TABLE 2.1: Failure rates among VMs in VirtualBox (version 6) out of 7486 cycles.

	CMS1	CMS2	AM	VBase	Overall
Total failures	70	75	79	82	306
Failure Percentage	1.40%	1.50%	1.58%	1.64%	6.14%
Failure period	71.2	66.5	63.1	60.8	16.3

TABLE 2.2: Failure rates among VMs in VirtualBox (version 7) out of 4987 cycles.

2.6.1 Severity

To get an idea of how severe these issues were we performed a large number of stability tests to predict the potential impact. However, we quickly found out that the rate these issues occurred seemed to depend on external factors. Since the failure rates[19] can vary wildly between 1 in 20 under good conditions, but we have also seen it occur as often as 1 in 4.

To give a better idea of the problems and their severity we have measured the amount of problematic runs using both VirtualBox 6 and VirtualBox 7. We do not suspect the VirtualBox version having any effect and account for the difference in the varying degrees of system load and background processes.

To test the values we made a script that repeatedly reboots all of the VMs, after this, we check which of the VMs comes back online within a set amount of time, which is well sufficient for a normally operating VM. After this period it is logged when a VM does not come back online, in this case, the VM also gets a hard reset such that it operates again on the next loop. Each of these loops we call a cycle since we cycle through each of the VMs being booted.

This process is then repeated a large number of times, often overnight, to get a large amount of data for comparable results. During these runs, we see that there is no bias towards a specific component failing more often than the others, all fail at roughly the same rate and throughout the process it differs which one fails.

The specific numbers can be found in table 2.1 and table 2.2 for VirtualBox 6 and 7 respectively. The rows denote the number of failures as a whole, the percentage of the total (failure proportion) and thirdly the failure period, every once in X times the VM fails to boot on average. The columns are each counting a specific VM with the last column counting the statistics covering all 4 combined. Since our tests require all components to be operational this is the most relevant number. The others are included to give a comparison between the different VMs.

Additionally, the graphs showing the interval between failures can be found in Appendix A.1. These graphs show at which iteration a Virtual Machine failed. This is to show that the failure rate does not change a lot over time, showing its consistency.

[19] We define failure rates in this context as on average a failed attempt happening once in every X times/iterations.

Stability workaround

Despite extensive testing with many different setups, including different machines, Virtual Images, Linux kernel versions and different versions of VirtualBox, we have not been able to identify the root cause of the problem. However, quite late into the research process, we found a workaround towards these issues.

This workaround was to set the paravirtualization-config to minimal as opposed to the default setting. To verify the stability we reran our reboot tests for almost 3000 cycles only detecting 2 failures that seem unrelated to our previous issues. With a failure rate of only 1 in 1500, we are confident that this workaround is viable. We have also run the release tests and compared the results with a setup using the old configuration to this setting and have found no negative impacts on our test results.

With this resolution, we are confident that we have put VirtualBox back on the list of contending Virtualization Technologies where, due to its lack of stability, we at first had major doubts about its success.

Although this has made the need to change much lower, and makes the alternatives seem much less appealing. The reader must keep in mind that this was not clear for a large part of this research, and the belief up to that point was that an alternative was required if found viable through other options.

Chapter 3

Related work

Before starting our research it is important to keep in mind other works that already have been published, in this regard we can look at several topics others have already researched. We will first look at other's findings in the Image Generation field. After that, we will take a look at various performance metrics among different Virtualization Technology. Finally, we will be zooming in on system testing and pipelines.

3.1 Image generation

As we wish to automatically generate Virtual (Machine) Images we need to find relevant ways to know which technologies can support us with this. One popular technology that may be employed is Docker[1]. Wu et al.[14] find that the build failure rate for a majority of open-source Docker projects is relatively low. Of the 3.828 analyzed projects 2.623 (68,5%) have a failure frequency of just 20% or lower with a median rate of 10.5%.

Additionally, other tooling, that is focused on system provisioning, may be employed, Święcicki[9] compared several system configuration tools that work with a central master where the different systems are the agents being configured. They also propose their tool designed to work as a standalone program. Torberntsson and Rydin[12] did a more comprehensive analysis of these configuration management systems. Although these are all focused at larger cloud scales we may be able to employ them for our use with Virtual Machines. For example, Salt[2] is also able to run without a master to configure the same machine itself[3].

With the aim of automating system testing van der Burg[13] looked at using NixOS to build Virtual Machines. NixOS is an operating system using the Nix package manager which allows declarative configuration for setting up an entire system. Although using NixOS, being a different operating system altogether, may be moving too far away from our source image it offers many insights in how we may be able to employ other tools.

This tells us there are multiple ways to configure and set up the systems that need to be on the Images that we wish to generate. Other works were however lacking in the regard of employing these techniques on Virtual Machines to generate them from scratch. Therefore, we believe this research to be novel in this regard.

[1] https://www.docker.com/
[2] https://docs.saltproject.io/en/latest/topics/about_salt_project.html
[3] https://docs.saltproject.io/en/3005/topics/tutorials/quickstart.html

3.2 Performance

Since we don't want our runs to take a longer amount of time but rather see a decrease we also need to take the performance of systems/technologies into account. The performance can be divided into three relevant areas for our case study; runtime performance, build performance and startup performance.

With runtime performance, we focus on the resource usage and duration for a certain workload to complete. This will be during the testing process.

Build performance is the time it takes to go from an empty operating ISO to a completed Virtual Image which we can use and run our tests on, this should be set up with the correct software packages and versions.

Startup performance relates to the time it takes for a machine to boot and load all processes such that it is ready to start using it in tests.

3.2.1 Runtime performance

When looking for performance testing/benchmarking between Virtual Machines (VMs) and containers a few papers can be found. Potdar et al.[7] compared the performance of KVM (Kernel-based Virtual Machine)[4] with Docker[5], where they found vast performance differences in favour of Docker. Chae et al.[1] did a similar study focusing more on resource usage.

Additionally, Siroky developed VTmark in their master thesis[2] which was used by Giallorenzo et al.[3] to compare a multitude of virtualization benchmark researches against their findings.

Overall all these benchmarks produce synthetic loads which may not be relevant in our more real-world-like situation. This will likely require the need to find other (types of) benchmarks/load testing to get a better idea.

One overall trend we can see when comparing older papers with newer ones is that the differences in technologies are decreasing. This can partially be explained by better implementations and continued work on their efficiency. As well as Hardware-assisted Virtualization[6] being introduced to CPUs and slowly getting implemented into these technologies. However, it seems that with more recent improvements the differences are decreasing, and the existing differences seem to favour Docker/container workloads.

Using these previous works we believe that for our research, we can abstract away from what is happening inside the Virtual Machines and put our focus towards the larger processes.

3.2.2 Build performance

One thing we were unable to find among other research which we are interested in is the build performance between the different technologies. These are ways to automatically generate Virtual Machine images with the desired software installed, or Docker images respectively. We can however note that there are different techniques, Docker has their build in Docker Images[7] we can utilize. On the Virtual Machine side there are multiple ways, one that is already in use by the company is Packer[8]. An alternative we might want

[4] https://www.redhat.com/en/topics/virtualization/what-is-KVM
[5] https://www.docker.com/why-docker/
[6] https://en.wikipedia.org/wiki/Hardware-assisted_virtualization
[7] https://docs.docker.com/glossary/#image
[8] https://www.packer.io/

to explore is utilizing Ansible to get the image in the desired state. This might even allow us to efficiently update VM images without replacing them entirely.

3.2.3 Startup performance

Another key aspect is the startup performance, also called deployment or provisioning in the hosting industry. How long do the systems take to start up their processes, and with hosting provide a space available for the system to run. This is another area with little relevant research, however, Lingayat et al.[5] looked at the time it takes for Docker images to boot on bare metal[9] and within a VM and found a large difference in performance in favour of bare metal. Mao et al.[6] and Hao et al. [4] looked at the time duration of VM provisioning among different cloud providers. These papers found that, among other things, in more recent years deployment times have drastically reduced. But for both of the papers their focus lies on the difference between data centres and providers. Direct research on boot times between different situations is not something we have been able to find.

3.3 System testing

For our system testing currently Robot Framework[10] is being used. Since this is already tightly integrated with a vast number of existing tests it is unlikely this will change, this will also fall quickly outside our scope and as such will not be extensively considered. However, it can still be valuable to compare it against other methods. The advantage the Robot Framework has is that it is fully automated. However, as Taipale et al.[10] found there are trade-offs and require the right type of system/rigidity for automated testing to give optimal results compared to manual testing.

3.4 Pipeline

The current pipeline being used at TKH Airport Solutions is managed through Jenkins[11], although not a lot of research is available about (testing) pipelines, it can still offer a way to see the alternatives and their practicality. Tanzil et al.[11] have done a study on DevOps challenges by looking at Stack Overflow questions. Here they found that under the category "Cloud & CI/CD Tool" the various Jenkins projects — pipeline, distributed architecture, and build projects, among others — score very well under their popularity metrics. However, they do score higher on the difficulty metrics, where Azure scores less popular but also less difficult. Another alternative is Gitlab CI, however, this is tightly integrated with Gitlab as a code repository which, although possible, does not suit the current setup using subversion.

3.4.1 CI/CD

Although our use case does not strictly follow CI/CD, a number of the papers covering this use case seem to be relevant towards the pipeline and its implementation. Additionally, this may be a direction the company is heading if the whole process can be stable and fast enough.

[9] physical hardware without a virtualization layer
[10] https://robotframework.org/
[11] https://www.jenkins.io/

We see that Singh et al.[8] compared Jenkins CI with Gitlab CI on an Amazon server. Finding that the overall performance between the two is similar. Jenkins is easier to use with support for many plugins but can grow complex with larger setups. Whereas Gitlab has a single configuration but offers less flexibility. Due to the complexity and the company using subversion, which GitLab does not support, Jenkins seems the better fit. Although looking at an alternative solution will be kept in mind, looking at an alternative will not be a priority during this research.

Chapter 4

Methodology

Before we can start measuring our data and metrics we first introduce our methodology explaining what metrics we will focus on and how and why we measure these.

We will first give a general outline of the process we will be following, after that we will zoom in further on each of the processes, what they entail and what we want to research within this topic. Lastly, we will give an overview of the metrics we wish to measure for each of our categories.

4.1 Overview

To start we will take several baseline measurements, the results from these will be presented along with the other measurements and results in chapter 8. The metrics we will be analysing during this research can be found in section 4.6.

Afterwards, we will be making a selection of Virtualization Technologies we can use and research. We will first make a selection of requirements and other criteria the Virtualization Technologies must follow before we can consider them. Then we will verify these Technologies against these criteria and make a ranking among them.

Using this ranking, we will implement each of the Technologies and accompanying Image Generation methods. We will give a short evaluation of each of these after which we will move our focus towards the pipeline. For the pipeline we will attempt to improve the overall usability, since many issues arise from the stability of VirtualBox and human error we will attempt to increase the stability significantly and reach as close as possible to full automation.

Once we have all our implementations we will be presenting our (more detailed) findings and measurements in the Results. Here we will give a comparison between each of the Image Generation methods and the old and the new pipeline based on each of the categories. We will conclude each category with a short conclusion ranking each of the Image Generation methods.

With our Virtual Technologies in hand, we will be looking at the image generation process to answer Research Question 1. Then to answer Research Question 2, if necessary, we will be implementing this technology in the pipeline. Our goal will be a fully automated pipeline from Image generation, using the previously built files to install. Then the setup of the images on the host machine, and finally executing a full test run.

With all this in hand, we can answer our Main Research Question about how we improved the System Testing that is being done at TKH Airport Solutions.

To give more context to each of these steps we will explain in more detail in the following sections.

- Technology selection
  - Criteria and requirements
  - Virtualization technologies
  - Decision matrix
- Image Generation Implementations
- Pipeline redesign
- Results
- Conclusion

4.2 Technology selection

Overall this chapter will focus on finding the shortcomings of the potential Virtualization Technologies for the Image Generation and the Virtualization Technologies. We will determine what a new technology must support to be considered as a replacement and finally make a selection of which new Technologies to explore further.

Criteria and requirements First, we will be looking at what technologies are available and their potential shortcomings. We will also analyze what is required for these technologies to work within the company and current processes. To get these requirements we will be in close collaboration with the employees at TKH Airport Solutions. We will be discussing what each of the components does within the CEDD AGL System and how they communicate. We will also be discussing the operational requirements and communication between the components and we will be discussing potential issues or pitfalls they have run into in the past.

Once we have a clear view of the requirements we will be making a list of the most important criteria each of the new Image Generation and Virtualization Technologies must abide by to be considered in our future comparisons. We will be doing this by making a decision matrix comparing, which is introduced later, and scoring each of the Virtualization Technologies to the criteria and scoring them. The scores for the Virtualization Technologies will dictate in which order we will be implementing our technologies. Such that in cases of time delays or other obstacles the candidates with the highest potential are implemented.

Validation through prototypes Based on these criteria we will need to do some validations to confirm whether or not a certain technology suffices by our standards set. This will prevent us from spending time on a candidate which may never meet our minimum requirements. This will be done separately by implementing simple prototypes pertaining just to the aspect that is desired to be tested.

The discussion and results from these prototypes will be discussed along the decision matrix in the relevant criteria.

Decision matrix Once we have set up all our criteria and have collected the information we can fill in our decision matrix. Once we have filled our decision matrix with the data and scored each Virtualization Technology accordingly we will look at the accompanying Image Generation Techniques that can be combined with these Virtualization Technologies.

4.3 Image Generation Implementations

Once we have selected the Image Generation techniques we will first make a simple implementation manually in the Virtualization Technologies. This is to confirm this platform can be supported and used. As well as finding any shortcomings and tweaks we may need to account for with the Image Generation techniques. Once we have confirmed that the Virtualization Technology matches our expectations, and can run a successful release test to confirm it is working, we will be starting the implementation of the Image Generation techniques.

Once we have built the CEDD AGL Components, and measured various metrics from the build process, it is key to know the final state of the build process. Since this can be hard to detect we use the Smoke test to quickly test the final state of the build process and whether it builds the images correctly.

Using this data we will already be making a selection of which Image Generation technique is best fit with the company. This will be done by giving a presentation providing all our findings at this point to the colleagues in the team we are working with. After this presentation, we will have a discussion where we take all final considerations into account and make a selection among the techniques with which we will be going forward. This Image Generation technique will then be integrated into the pipeline where further adjustments will be made.

In this chapter, we will explain our experiences and findings using these techniques. The overall metrics and results will be shared later in the chapter Results.

4.4 Pipeline improvements

Once we have our Image Generation technique we can look at how we need to adapt the pipeline to accommodate this technology if necessary, and how we can implement other improvements.

These improvements will be mainly focused on streamlining and stabilizing the process. We aim to achieve this through improving the stability through catching problems. But also by automating more steps such that there is less room for human error. We will also attempt to increase the ease of use and the sense of security such that more of the developers are comfortable and eager to use the new pipeline.

4.5 Results and conclusion

To answer our research questions we will be gathering the Metrics we have collected during the baseline measurement and while implementing and measuring the Image Generation process and the pipeline.

This data will first be presented in the Results chapter where we will analyse each metric separately. We will first introduce our data for the different topics of this research, the image generation process and the system testing process, and discuss them shortly giving our take on the results we found. At the end of each metric, we will give a summary concluding our findings and giving a ranking among the Image Generation Technologies.

In the Conclusion we will take the resulting information from each of the metrics and look at the overall larger picture. We will be combining the results of each metric and the advantages found in the Technology selection to select a winner. Within the company, this winner will already be chosen earlier, with close collaboration with the developers, testers

and team lead, which allows us to implement the new technology in the pipeline when necessary.

After discussing the overall improvements we will focus on each of the research questions separately. For the first question, we will be discussing the Image Generation Techniques and in what situations some of the Techniques may be applied in the future. After this, we will zoom in on our most impactful changes to the pipeline and system testing process in general.

After this, we will take both the Image Generation process and the pipeline into account to answer the Main Research Question on how we improved the system testing at TKH Airport Solutions. We will also be discussing with employees to find what our impact has been on their processes and their findings for the new pipeline and the Image Generation process.

Lastly, we will give a few suggestions for potential future works which we had to leave out of our scope either due to complexity or time constraints.

4.6 Metrics

During the research we want to collect multiple metrics, in this section, we explain these metrics; what we want to measure, how will we measure and what is their purpose. First, we will explain how we collect a large part of our data, after that, we will expand upon the different metrics we will put our focus on. These metrics are based on the following topics; Stability, Automation, Performance and Company impact. What each of these topics entails will be explained in their respective sections after the data collection and scripting.

4.6.1 Data collection and scripting

To collect a large part of our data in a (partially) automated manner we will be using some wrapper scripts to automate the process together with several existing process measuring tools to collect the data. For most cases, we will often make use of three scripts. We will be calling these scripts the *cycle*, *benchmark* and the *process* scripts after their functional tasks.

The purpose of these scripts is to allow for ease of use in automatically/repeatedly running the tasks at hand as well as collecting valuable information. Most of this information gathering is focused on the resource usage of various processes as well as on how stable something is running by logging this information.

The cycle script is a simple script which does some basic setup like creating the folder where all the logs will reside and then will run the benchmark script a defined number of times (often enough to run throughout the night).

The benchmark script which gets repeatedly started by our cycle script is the core of our benchmarking process. This script restores the host machine to a starting state by, for example, deleting residual files from the previous run, as well as removing old images. Once everything is ready for a new run this script starts the benchmarking tools like procpath and psrecord, and then will run the process script. After the process script is done the benchmark script often checks the state of the output, if not already directly done by the process, for example, by running a Smoke test or checking certain file states.

The process script as mentioned is what is actually running the process we want to measure. This can, for example, run the entire Image Generation for a certain tool, or it can cover a different set of actions we wish to measure. All the system resources used

during this process are measured by the tools from the benchmark scripts. This is done by, besides measuring the process script itself also, logging all child processes spawned by the commands being run in the process script. This way we can easily measure multiple tasks being done by different commands without repeatedly setting up the monitoring for each separate command.

Additionally, these scripts also contain a large number of timestamp logging, this information can then be used to extrapolate a timeline for each install to get information on how long a certain process took, for example, the cloning/exporting of an image or how long it took to run the installation script.

Deliverables

Due to the confidential nature of the software being installed and tested, combined with the close interfacing these scripts at times do with this software, we can not publish the used scripts. These scripts, along with the configurations for the image generation, will be made available within the company for potential future reference. Additionally, we plan to implement our findings in a final pipeline setup for both the image generation process and the system testing process.

4.6.2 Median data

For our data metrics, we will mostly focus on the median[1] values. Opposed to the average value the median is less susceptible to outliers in the data. Which, given that our image generation should mostly focus on standard, near ideal, circumstances the outlier data is not interesting for our comparisons. Additionally as can be seen in the graph of figure 4.1 with the other situations available in Appendix A.2 the resource usage is quite stable. Couple this with the fact that the data we collect is dependent on a lot of outside factors[2].

Additionally, outliers may often be caused by hard-to-explain external factors stalling a process or taking up resources that the process wants to use. One example having a major impact was the disk usage, while exporting or cloning the Virtual Machine this process could often take up all the read/write power on the SSD in the host machine. This could cause a larger variation in the data when other tasks, like using the browser or going through test reports, were performed. These are not representative of happening on a dedicated build machine or a system with more I/O performance.

Taking these considerations into account we feel that the median will give us a sufficiently reflective reference number for our data without the need to dive into deeper and more complex mathematics to take out outliers.

4.6.3 Stability

With stability we will focus on how stable the process is during running, this will be done through a focus on the success rate. What portion of running certain actions does succeed without further interaction, this can be running tests, configuration or image generation.

An important aspect of this is the boot reliability. Early on we found that the reliability for VirtualBox booting was simply too low, impacting both the image generation and the system testing. As such we will also emphasize the stability in regards to successful reboots.

[1] The median is the value in the middle of a data set, this means that half of the number of values are below this value, and the other half above.

[2] During our tests multiple other background processes may or may not have affected various performance metrics measured

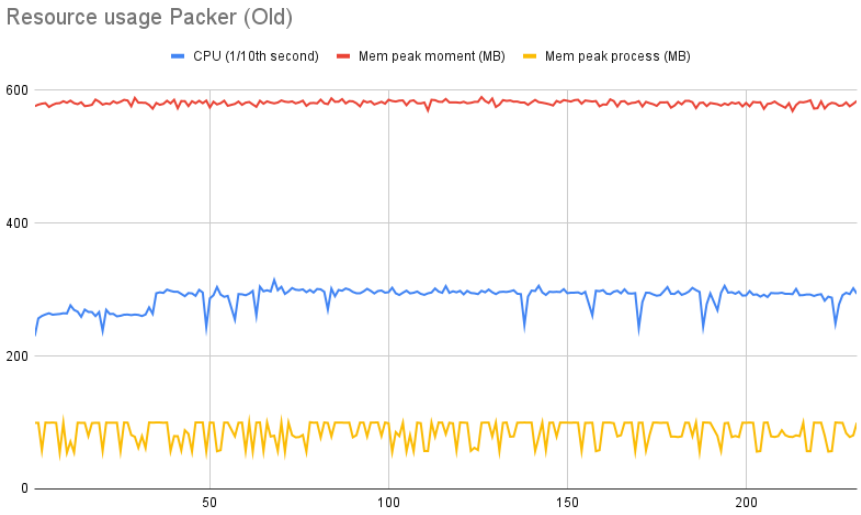

FIGURE 4.1: Example of the overall stability of resource usage over multiple runs. This data is from the old Packer implementations

For the image generation this will focus on when a build is successful, a build is considered successful when the build process exits with an intact build. We consider a build intact if there is no corruption with the files and no obvious wrong signs. In situations where this can be done easily, we will perform a smoketest to validate the success of the build.

For system testing/pipeline we will focus on successful runs of the pipeline. Since the pipeline is designed to stop when a step fails we have opted to not consider the results of the large release test. This is due to the fact that not all tests are working correctly at the time of research. The release tests cover several known issues that have a low priority to solve for the developers at the moment. This often covers edge cases or scenarios that are nearly impossible to find in the real world.

4.6.4 Automation

The automation measurement will be based on how well the stage can execute on its own. We want to minimise the manual steps/interventions of the user. As such this metric should measure the amount of effort a user needs to perform to run the desired stage. Although automation is closely related to stability, often requiring manual restarting of the pipeline after some instability occured, we will count these incidents separately to focus more objectively on the specific task at hand.

This task will not be able to be automated since it inherently relies on the amount of human interaction that is needed, at what points they are needed and the amount of effort required to perform the task. We will manually be logging these instances where interactions are required, we will do this by running the pipeline several times throughout our research such that we do not become too experienced with every step and simulate the experiences a developer/tester has who occasionally uses the pipeline.

In measuring the automation we will focus on two aspects, how often user input is required and how much effort this input requires. For the first, we will measure the frequency and moments the user needs to perform certain actions and categorise them.

An initial setup or tear down[3] is less bothersome than having to manually perform a task partway through the process, possibly at a random moment.

Next to this, we will classify the effort of the action. For this, we will look at how long it took to perform the action and the complexity. Was it clicking a single button or was it a way more involved task deleting select files in certain locations. Although complexity and duration are often linked it may be the case that a few button presses with some waiting moments takes more time than a hugely complex series of operations.

In the context of manual steps for the second research question, we will also look at categorising the kind of issue the manual action triggers. This can be forgetting to execute a previous step or performing an, often error-prone, task (partially) incorrectly.

4.6.5 Performance

Although less important than the stability and automation metrics it is important to keep the build and testing stage performing well. We do not want the duration of testing to take exponentially longer or many more resources to be required to perform all tasks. As such we have the performance metric which will measure how long the build stage takes and how long the tests need to run. We will also look at the resource usage of both stages to see that this does not explode and stays manageable.

Measuring the resource usage of the Virtual Systems themselves is very tricky, due to them being managed by a hypervisor and the resources are often loosely coupled/hard to get the process information for. Also, different Virtualization Technologies use different ways to measure their resource usage making this data unreliable. As a result, we decided to leave the resource usage of the Virtualized system out of scope and we will focus solely on the resource usage of the tooling/programs used.

For the build stage, we want to measure the memory and CPU usage as well as how much disk space gets used by potential caches and the final system images. Storage space is important when we want to keep older versions, especially the usage of completed builds. These older versions may be used for comparison and to see where a regression may have introduced itself.

The testing stage will be measured by manually making scripts that perform the same steps as the pipelines and measuring the resource usage of these steps. Since we plan to make large overhauls to the pipeline we will make very granular measurements not taking separate steps into account but looking at a larger set of actions to perform greater tasks like configuring the host machine. Our focus will lie on the memory and CPU usage, since a developer/tester usually only has one version they use and all PCs are equipped with SSD storage the storage usage and intensity are less of a concern.

We make heavy use of procpath[4] version 1.6.1 to take measurements and log the memory usage and the usage of CPU time. We also take support from psrecord[5] version 1.2 to keep track of overall CPU % and Memory usage and we use pidstat[6] version 12.5.2[7] to monitor disk usage (read and write speeds) per process.

To ensure that we focus on the metrics we want and not, for example, the resource monitoring tools themselves, we have been able to filter most of the noise from the measurement of our tooling, additionally, we have manually checked at various points through

[3] A teardown is an action at the end of a process to finalise this task.

[4] https://procpath.readthedocs.io

[5] https://github.com/astrofrog/psrecord

[6] https://man7.org/linux/man-pages/man1/pidstat.1.html

[7] The version of pidstat is not directly available, the version used is packaged in sysstat 12.5.2 on Ubuntu

the measured data to check the impact of the leftover 'noise'. From this, we can conclude that the impact of our measuring toolset should be minimal and negligible between different test runs as well as the different implementations.

4.6.6 Company impact

Our goal is to keep the pipeline simple, as such we wish to introduce as little new software as possible, or keep them as simple as we can, such that the employees have a minimal learning curve towards working with the new pipeline.

With this in mind, our preference goes out to keep using tooling that is already used, like Packer and Docker, or have other tooling be similar in setup/usage. With this metric covering both the build and testing stages, we count the amount of newly introduced tooling and classify the complexity of new tooling/features compared to the existing setup. Although this type of classification is quite abstract we will be looking at the potential learning curve, the documentation clarity and the length and availability of tutorials, plugins and other helpful resources that may aid in the usage of the tooling.

Chapter 5

Technology selection

An important decision we need to make is to decide which Virtualization Technologies we will be looking at during this research. This decision will also influence which Image Generation processes we can look at, which will be explored in the next chapter (Image Generation Implementations).

During this chapter, we will first be setting up several criteria each of the Virtualization Technologies has to support. We will be basing a ranking on these criteria on how well they score. The better they score on the criteria, where there is room for variation, the more interesting this Virtualization Technology will likely be for our research. Using these criteria we will be scoring the Virtualization Technologies on a matrix which will decide the order in which we will explore the technologies.

5.1 Criteria and requirements

In an attempt to potentially rule out some of the technologies that may not fit our needs and requirements we can set up some criteria these technologies need to meet before we will start implementing a generation set up with them.

5.1.1 Boot reliability

Since a large number of our issues can be related to the problems with booting the Virtual Machines we will need a more reliable technology. For this, we will verify that the desired Technologies can repeatedly reboot a Virtual System without running into an error during the boot.

5.1.2 Multiple network interfaces

The current setup makes tight use of network bonding to add resilience towards cables being cut or other temporary hardware failures in the network stack. Although not critical in the virtual test setup it is still very much desired to test this functionality. To accurately test this we ideally employ a Virtual setup where network bonding is supported. Although we acknowledge that one of the interesting contenders, Docker, has no support for this due to how it is integrated with the Linux Kernel. To still accurately test the other aspects of the network we require at least the support for multiple (virtual) networks (bridges) to be set up.

5.1.3 Full automation/scripting

Since our ultimate goal is to automatically test our software stack after code commits we need a way to fully automate the generation of images without the need for user interaction. As such we will need a system that can reliably generate these images, ideally, from the command line.

5.1.4 Platform support

While we were in the process of implementing and testing the various Virtualization Technologies and Image Generation tools an extra requirement became apparent. This was the need that other departments also had to use the Virtual Basic setup, with the very strong desire to only rely upon one generation technique/implementation for the Virtual Images this technology had to be supported on both Windows and macOS.

5.1.5 Partitions

The CEDD AGL Components currently make use of different partitions to store different types of data like configurations and logging. This is to ensure that (failures causing) excessive logging would not prevent other parts of this machine from operating by being full on disk space.

Ideally, we should employ a technology that supports multiple partitioning or limits on storage as this emulates the production best. However since our test suite does not rely on this functionality, and this has never come up within nominal testing operations we do not consider this a hard requirement.

5.1.6 Cost

Although it is not a problem for software to have a cost, in enterprise situations these costs can quickly grow to large numbers. With this in mind, we will take a look at the licensing models and pricing and predict what kind of effect this has on our needs. Since each employee has their own machine (often also a laptop that may be used), and several other machines may be used by the developers and testers the potential cost can quickly rise if each developer machine needs a unique license key.

5.1.7 Summary

In summary, we have the following criteria to consider ordered from most important to least important;

- Boot reliability
- Platform support
- Automation/scripting
- Network interfaces
- Cost
- Partitions

5.2 Virtualization Technologies

The Virtualization Technologies we found that abide by our minimum requirements, and thus will be scored on our decision matrix are the following:

- VirtualBox
- KVM
- Docker
- VMWare

There are some other Virtualization Technologies that we found, they do however not abide by our requirements for various reasons. The most prominent is that they do not have support for Linux which is the Operating System that most developers use.

Some of the dropped options and their reason for being left out are:

- Hyper-V (Windows only)
- Parallels (macOS only[1])
- LXC (Containerization which is more efficiently covered by Docker)

5.3 Decision Matrix

Using the previous criteria and findings we produce our decision matrix. First, we introduce how we will be scoring each of the criteria and afterwards, we will show our matrix and will explain some of our argumentation.

5.3.1 Criteria scoring

First, we will compare each of our technologies (both virtualization and image generation) against the set of criteria we found. We will be doing this on a four-tier scoring system. Each tier will have a name and a character representing the tier, the characters will be used in the decision matrix to show information more quickly and compactly. The scores are as follows:

- **Inadequate** (x): This is the lowest tier. It signifies that the performance or quality is not up to our standard or expectations. This tier signifies that the scoring in this regard is insufficient for us to use this tool covering these criteria. Further changes along the way may make room to loosen this requirement.
- **Workable** (-): This is the second tier, above "Inadequate". It signifies that while the performance or quality is not fully up to standard, it is functional or acceptable for the time being but requires improvement or adjustment to reach the standard.
- **Adequate** (=): This is the third tier. It signifies that the performance or quality meets the basic standards or expectations but still has room for improvement. This will be able to perform the necessary tasks but does not necessarily offer great support for further depending on this tooling set.

[1] Parallels dropped support for other operating systems since 2013

- **Excellent** (+): This is the highest tier. It signifies that the performance or quality not only meets but exceeds the standards or expectations. With this score, the criteria are well met along with any extra nice-to-have/use features it will likely accommodate.

With these scores and our decision matrix, we will be ranking the Virtualization Technologies. An inadequate score in a requirement which can not be overcome or worked around should invalidate the Technology from being used further. Since this would invalidate our efforts and block progression towards a more suited tool.

5.3.2 Decision matrix and argumentation

	Boot Reliability	Network Interfaces	Automation /Scripting	Platform support	Partitions	Pricing	Overall Score
Virtual-Box old	x	+	=	+	+	+	x
Virtual-Box new	+	+	=	+	+	+	+
Docker	+	-	+	=	-	=	=
KVM	+	=	=	-	+	+	=
VMWare	+	+	=	+	=	-	=

TABLE 5.1: Comparison of the various Virtualization Technologies.

VirtualBox Old Our main concern with the old VirtualBox situation is the boot reliability. As we have previously determined this was insufficient causing us a lot of trouble. As a result, we can not give this a passing mark.

VirtualBox New The new situation of VirtualBox which has found a workaround for the boot reliability solves our concerns and as a result, can receive a passing mark. Considering it scores well in many of the other categories we can even give it an excellent score.

Docker Docker has a lot of nice features outside our direct needs. One concern we have with Docker is the lack of support for network bonding. We have done extensive testing to try and get network bonding working but we found out later that due to the design and how it is connected with the Linux Kernel, it is not possible. Technically there are some ways to work around this issue but these would defeat the purpose of bonding in the first place. Next to this, there is no real/proper support for partitions. Docker data resides on the container itself and there are ways to limit the size of the storage but these all work differently and not as well as partitions would in this situation.

Considering these limitations we can not give Docker an excellent score, but since it performs well in many of our criteria we give it an adequate score.

KVM KVM is a tool which also offers a lot of features outside our direct need. It has very good performance and also a good amount of tooling available for automation. It lacks however in the platform support. Since KVM is directly tied to the Linux kernel this limits in which ways it can be used. Using online resources we found that KVM can be

used through Windows Subsystem for Linux[2]. Direct support on macOS is also lacking but QEMU is supported[3], which is closely related to KVM and should be able to run many of the same images. Additionally, through nested Virtualization these different platforms could be supported and since the situations where other platforms need to be used should be minimal and workable it should be possible to continue with KVM.

Since KVM offers us a lot of features, but may not be able to cover full native platform support, we can not give it an excellent rating, but since many of our requirements are still met and we can work around the downsides we can still give it a score of adequate.

VMWare VMWare has a lot to offer and seems to be similar in supported features to VirtualBox. One huge downside to VMWare is its licensing cost making it very expensive for a company to run. Additionally, this cost comes back with our testing implementation. There is only a 30-day free trial offered which caused us to select this technology last such that we would have the most experience to find a proper implementation quickly. That also meant that if time ran short this option would be the first to be dropped, which turned out to be the case.

[2] https://serverfault.com/a/1115773
[3] https://stackoverflow.com/a/53783839

Chapter 6

Image Generation Implementations

In this chapter, we will be looking at and giving our impressions of alternative options for the Image Generation process. In the last chapter we looked at the different Virtual Technologies we will be looking at for this research. Based on those technologies we have made a selection of Image Generation Technologies. Some of these are only available for one Virtual Technology whereas others have support for multiple, in each case, the supported Virtual Technologies are indicated between brackets.

- Packer new (VirtualBox)
- Docker (Docker)
- Cloud-init (KVM)
- Virt-Customize (VirtualBox/KVM)
- Ansible (VirtualBox/KVM)

6.1 Base image

One quick time-saving option we identified at the start was to make use of the base image. Even in the old Packer implementation, a base image was created on which the other components were installed. This base image was a simple installation of the operating system along with a few packages that were shared among all the separate components of the CEDD AGL System.

The advantage of using a base image is that instead of building each machine from scratch, including the operating system install, this process only has to be done once. By keeping this base image around and using it for future installations this time was saved. Except for the Cloud-init method which on purpose installs the whole image from scratch.

In each of our other implementations, we made use of this base image. The downside is that a copy of this base needs to be made. VirtualBox offers an efficient clone operation but requires to be exported afterwards. With KVM we can make a copy of the disks this base image is installed on and use those copies to install the components.

6.2 Packer

Packer[1] is the tooling already used by TKH Airport Solutions, and as such was already configured to be used. Implementing new ideas was therefore very straightforward since

[1] https://developer.hashicorp.com/packer/integrations/hashicorp/virtualbox

we could easily adapt the current configuration. One downside of Packer is that it is only compatible with VirtualBox and not with KVM due to their KVM implementation not supporting multiple network interfaces.

As mentioned in the background section 2.6.1 we have solved the main issue with stability under VirtualBox. The solution was the para-virtualization mode for the VM. How this changes things exactly and what the underlying issue was is as of now still unclear. This does mean that Packer is among the potential candidates again

The first point of improvement compared to the old implementation was that instead of regenerating the base image we now keep a spare copy of the base image. This base image was reused (one by one) for each of the systems to be installed. Using VirtualBox to make copies of this base image for each of the Components to be implemented also allowed us to generate each of the Components in parallel instead of consecutively. Although this would bottleneck the disk I/O more this would allow for more, less resource-intensive, tasks to run at the same time. This seemed to speed up the generation process quite a bit.

Overall not many changes were needed since all the hard parts were already figured out and implemented.

6.3 Docker

Docker[2] is a very popular tool in the world of automated testing and deployment. Although Docker is designed to be used for running applications and not for running larger parts like an operating system, and it is often recommended against, it is still possible to do so. Since Docker has the potential to offer a large set of features, like build caching and a large set of tooling, it is still a candidate worth considering for investigation.

A very useful tool that recently got integrated into Docker is Docker-Compose[3]. Docker-compose makes managing multiple containers, and the networking and other tasks much easier. Using Docker-Compose we could much more easily manage the multiple containers we were working with to emulate the whole CEDD AGL System as a single sort of unit. This made the higher level management of building, setting up, turning on or of the Virtual Basic setup as a whole possible with single commands.

What we found out when implementing Docker was that support for nested Docker-in-Docker was less optimal than hoped for. We encountered some trouble getting a reliable way set up, first looking at the SysBox[4] environment. Although very promising in their ability we ran into some compatibility issues within our nested Docker containers which needed more privileges. As a result, we decided to run with privileged[5] containers to extend this support.

An additional issue we ran into was that we could not import/build the needed Docker images inside the containers without starting them up. This removed one of the large potential advantages Docker had with its cache. To circumvent this we could start up the containers, perform the required actions which require the containers to be running and then export the complete state using *Docker commit*[6]. Alternatively, it is possible to leave the tasks for which Docker needs to be started to the developer during the import. This would, however, greatly extend the setup time for a new version and could much more efficiently be done once when generating the images.

[2] https://www.docker.com/
[3] https://docs.docker.com/compose/
[4] https://github.com/nestybox/sysbox
[5] https://docs.docker.com/engine/reference/commandline/run/#privileged
[6] https://docs.docker.com/engine/reference/commandline/commit/

Due to the design/integration of Docker in line with the Linux Kernel, it is also not possible to make use of network bonding. Due to the fact that both are managed at a kernel layer and Docker receives network interfaces at a lower level than where bonded traffic is sent to the OS, it is impossible to implement. Luckily Docker does offer the ability to make use of multiple network interfaces, such that the bonded network can be emulated and communication is still possible to be tested.

Overall once set up the configuration and management for Docker is quite simple, there are a small number of pitfalls that one can run into when adapting the DockerFile or when changing some of the generation steps. One concern is also the lack of support for network bonding, this is a fairly high priority internal to be working and the company would very much like this being tested using the early stage Virtual Basic setup.

6.4 Cloud-Init

Cloud-init[7] is a tool for configuring cloud instances, it is also used as the automatic network installer of Ubuntu, which works with both VirtualBox and KVM. The Cloud-init method is mostly aimed at installing complete pre-defined hardware devices, including all drivers and other potential device-specific aspects. This tool does however also fit our needs in regards to the automated installation. One advantage, but also a drawback, is its simplicity. This simplicity allows one configuration file with which the installed packages, partitions, network interfaces and more can be configured. However, this is also a large downside since post-installation steps are a lot more difficult, which we need for our configuration. Additionally, for file transfers we need to look at an alternative strategy, like setting up a webserver to share the files.

The largest problem with Cloud-init was the stability. The tooling would often work the first time, but consecutive runs often had a weird error where the configuration, which can be hosted over the network in a central location, was never even requested. One trick around this was to host the desired configuration inside an ISO file, which likely also could be applied to the files that needed to be transferred but we elected to keep this method out of scope.

Finally what also hurt the stability of Cloud-init was failures during the OS installation. This would silently fail with no real way of detecting this from the outside besides some timeout method. The info reporting why this would fail also consisted of crash logging, including stack traces, in which we could, with our experience, not find any useful information.

Overall the perceived stability of the Cloud-init method made it far less favorable compared to the other options. A lot of the issues we encountered while setting this method up were unclear as to what their cause was. Additionally, we found some conflicting information in the documentation which was also very sparse to find. However, we do see potential in this tool when working in an environment where the OS needs to be installed from scratch on unknown hardware.

[7] https://cloud-init.io/

6.5 Virt-Customize

Virt-customize[8] is a tool part of lib-guestfs which *"is a set of tools for accessing and modifying virtual machine (VM) disk images."*[9] Only a select few types of disks are supported, however. This limits Virt-customize in the Virtualization Technologies that it can be together with, in our case only KVM.

This allows us to edit the System Images, install packages, insert files and change permissions as well as set up a boot script with which we can further finalize the installation. Using this script we were able to easily perform the latest configuration steps that required a running system, like importing/setting up the internal Docker systems. This allowed us to simply prepare the System Images, which we could copy from a base image, and run them a single time for a full setup. Setting a shutdown command as the last step in the boot script made it easy for the generation script to get a signal when the process was done.

Overall the process using Virt-customize was very enjoyable and easy to manage. It took little effort to get an initial version working, which may partially be because of our previous experience, but the tooling itself also was easy to work with.

6.6 Ansible

Ansible comes from a set of tooling that is aimed at platform configuration, where pre-installed cloud machines can be set up with the desired software and configuration. This type of tooling knows multiple alternatives, like salt[10], puppet[11] and Chef[12]. From these tooling types, we decided to look at one option to not get overwhelmed and look at the viability of applying this type of software within this context.

We have chosen Ansible, compared to the other alternatives, since it is much easier to set up the machines to be configured. Where Ansible needs to be able to SSH to the nodes, other tooling often requires specific software to be run on the nodes and additional configuration. This allowed for easier integration and less hassle in the setup part, before the image generation process.

When using Ansible the broad selection of options proved to be very useful to get done exactly what was needed in a straightforward manner. The downside of this is that there are a lot of options and finding the correct one can, at times, be challenging. For example, 19 different actions can be formed from the *ansible.builtin.file* module[13] alone.

Some troubles we had with Ansible was with it using the right Python version internally. It tried using the outdated python2.7 internally for which it tried to automatically install the python-apt library which was not available anymore.

Overall when the right options to use are figured out Ansible is very flexible in its operation. We did, however, have some trouble getting Ansible set up in regards to dependencies and Python versions.

[8] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration_guide/sect-guest_virtual_machine_disk_access_with_offline_tools-using_virt_customize

[9] https://www.libguestfs.org/

[10] https://docs.saltproject.io/salt/install-guide/en/latest/topics/quickstart.html

[11] https://www.puppet.com/

[12] https://www.chef.io/products/chef-infrastructure-management

[13] https://docs.ansible.com/ansible/latest/collections/ansible/builtin/file_module.html

Chapter 7

Pipeline redesign

During this research, we also looked at a large number of ways to improve the pipeline and improve the stability and avoid issues, which could often be caused by user error. We planned to achieve a lot of this through automating more processes implementing fallbacks and better logging about wrong states when something unrecoverable does come up.

For this research, we have made a copy of the pipeline as it was at the start of this research and will solely work on this copy. This will ensure that the old baseline state stays intact as a point to compare our new pipeline with.

7.1 Overview

The overall process that was to be achieved can be viewed as three different goals, of which the old pipeline only covered the first two. In the new pipeline, we split each of the stages into a separate pipeline since these sets of actions are intended to be used together but at different frequencies.

1. Setup of the host machine
2. Deployment of the Virtual Basic setup
3. Running of the Release test

Each of these pipelines/goals consists of several stages each achieving a smaller task, for example installing the correct version of VirtualBox, or deploying the configuration to the Master AM component. For most of these tasks, a selection can be made for whether or not they should be executed. This allows for extended flexibility if only part of the pipeline process is desired.

To give a better overview of what each of these pipelines is meant to achieve and give further background information about them, the following subsections will each describe their respective pipeline.

7.1.1 Setup host machine

This pipeline is there to set up the host machine. The focus is solely on the physical machine a developer uses and so far does nothing with the (virtualized) CEDD AGL System.

This section will set up all required software and configurations. This will for example install the required (python) packages and set up a *Python Virtual Environment*[1] with

[1] https://docs.python.org/3/tutorial/venv.html

Robot Framework[2]. Or install/update VirtualBox and set some system configurations to use all required VirtualBox features.

This pipeline should only be needed very infrequently when a new setup is required or the developer wishes to ensure a good working environment to be set up from scratch again, after manually cleaning up the old state.

7.1.2 Deploy Basic Virtual

This pipeline is where the deployment and setup of the Virtual Basic setup takes place. Old versions of the CEDD AGL System are optionally removed after which the latest version will be installed. Additionally, all setup and configuration of the CEDD AGL System such that it is ready for testing will be performed. Once the CEDD AGL System is set up a quick Smoke test is performed to confirm basic functionality and proper setup. After this, the developer can run manual tests, change files/configurations they want to test or run the next pipeline section such that a proper release test is done.

7.1.3 Run system test

Once the desired system setup is done the developer can make use of the last pipeline to run the Release test. This pipeline also ensures the Virtual Machines are started and running such that this simple step can not be forgotten. One of the improvements aimed at reducing user error and increasing the reliability. However, this is not often necessary due to it often running after the second pipeline, which on purpose leaves the VMs running by default.

7.2 Changes

As mentioned before a major overhaul we did was splitting the original pipeline up into three distinct sections each functioning as their own pipeline. Besides this, we worked also on a lot of automation and implementing fallbacks or error reporting.

Since going into every detail would not be interesting, and would likely get too close to confidential company information, we will write some of the more interesting changes in this section. These changes might also be of more interest to the reader since these could be considered more general recommendations towards the design of a pipeline.

Each of these subsections will receive a short description of the process as the title

7.2.1 No stash and unstash Virtual Images

Based on time and CPU savings the most impactful change was the removal of the transfer of the Virtual Machine Images through the Jenkins internal tooling using stash/unstash[3]. Although this was a very simple solution to transfer the files from the *buildserver* hosting the Images to the developer's machine it did take a long time for the files to get compressed and decompressed.

By serving the files over a network mount we could circumvent the need for this slow process. The network mounts were already present to have access to other files but were not yet used by the pipeline.

[2] https://robotframework.org/

[3] https://www.jenkins.io/doc/pipeline/steps/workflow-basic-steps/#stash-stash-some-files-to-be-used-later-in-the-build

One downside we noticed with this change was that the step to import the images into VirtualBox, which now directly accesses the images over the network mount, could get severely slowed down when multiple users were running the pipeline at the same time. This comes down to the maximum upload speed the *buildserver* hosting the files could achieve. Although this is something we did not test on the old pipeline we expect that the old pipeline would have run into similar bottlenecks, albeit in a different step of the pipeline.

7.2.2 Delete old Virtual Images

Effort-wise one of the most impactful implementations was the (optional) automatic removal of older images still present in VirtualBox. Beforehand a developer had to manually remove the old Virtual Images from VirtualBox. This process could easily be done incorrectly.

Since this process requires machines to be removed in the correct order. Due to a technicality in VirtualBox when a machine is cloned, as is the case for CMS2, first the clone needs to be removed before removing the original machine. If this is not done correctly some files are left behind which may cause issues when importing the same version again.

By automating this process, and ensuring the correct order, this issue is prevented from happening. Since this task is quite cumbersome this can save a lot of time and intricate work.

7.2.3 Start Virtual Machines

Although this step was already automated the implementation was quite naive. This stage would detect which machines to start, send the start command to VirtualBox and then wait 30 seconds for the machines to be started.

Our new implementation interfaces with the machines over SSH to detect when they are started up. By doing this instead of naively waiting 30 seconds we can speed up this process. This also allows us to set a timeout for if a machine did not boot correctly within a set amount of time giving a proper alert to the user running the pipeline as opposed to a future stage failing because one of the machines was not online.

7.2.4 Headless starting

Along with the previous point about more efficient starting of the Virtual Machines, we also made them start headless. This would have the advantage that the VirtualBox Manager did not have to be opened beforehand. The exact reason for this has not been found out but it seems to be a quirk with VirtualBox that non-headless Virtual Machines require the manager to be opened.

Using the headless Virtual Machines will also allow us to run the Virtual Basic setup on a machine without a display (required to open the VirtualBox Manager) like a server.

7.2.5 Deploy Master AM configuration

This step was improved by one of the other employees at TKH Airport Solutions, however is visible in the timing data when comparing the old and the new pipeline. The old process was found to be too naive and could not correctly handle the situation where the Master AM was just started and not all internal components were loaded. This led to issues on multiple occasions where we decided to implement more situational checks and waits to ensure the Master AM was in the correct state before uploading the configuration.

The downside to this change is that the process takes almost double as long, but it is much more stable and this stage has yet to fail due to being in this state.

7.2.6 Network interfaces management

In the older version of the pipeline, the user had to manually delete the network interfaces created in VirtualBox. This was done such that the pipeline could always create three interfaces to ensure that these were present. This was required for the Virtual Basic setup to run, but also because the pipeline needed these to be present for other configuration steps.

The new pipeline instead detects the presence of these interfaces through the *vboxmanage list hostonlyifs* command and creates new interfaces only if required.

7.2.7 Minimizing higher privilege

One discomfort some developers indicated was that to run, a part of, the old pipeline escalated privileges with sudo were required. To achieve these permissions with the pipeline it is required to give the user access to the sudo command without a password prompt. Since the need to run commands with sudo is not unavoidable, due to the required higher privileges the apt package manager requires, we have minimized the use of them to only be present in the first pipeline. We designed the other pipelines in such a way that these escalated privileges were not required. This has put the concerns of these developers at ease with the result of them also making (more) use of the pipeline.

7.3 Python version validation

A functionality that is newly implemented in the second and third pipelines is that Python versions will be reevaluated with the configured versions. For this, the requirements.txt[4] file will be reevaluated with pip[5] to match the specified versions. This is to ensure that at all times the versions as intended by the developers are being used. Even when an older version is being retested this will properly downgrade the required packages to match the state as desired.

7.4 Full automation

For the pipeline, our goal was to remove as many manual steps as possible. We believe we achieved support for full automation with the new pipeline from setup to rolling out of the Virtual Basic setup to running the release test and collecting the results. We have run tests where we have executed the second and third pipelines multiple times consecutively. These runs all went successfully without the need for any manual actions besides starting the pipeline.

There are, however, still some situations where a bad state could prevent full automation. Due to time constraints and other technicalities, we have not been able to solve all these issues. As a result, a bad actor could manage to bring the pipeline to a state where the automation can not fully recover. We believe however that this is not a huge concern since deliberate abuse would be required to achieve these states and with some manual cleanup, the normal operations should be able to be resumed.

[4] https://pip.pypa.io/en/stable/reference/requirements-file-format/
[5] https://pypi.org/project/pip/

Chapter 8

Results

After executing our plans from the Methodology we can present the results from these measurements. We will split this chapter according to the four different types of metrics we have gathered:

- Stability
- Automation
- Performance
- Company impact

For each metric we will first be presenting and shortly discussing our results, we will finish each metric by giving a summary and a ranking of how we believe each of the virtualization Technologies scored for the metric. We will also be comparing the old and the new pipelines for each metric to compare the improvements we have made.

Lastly, we will also be giving an overview summarising how each of the technologies scored in the different metrics. This will be supported by a table giving an overview of each of the rankings of the different Image Generation Technologies.

8.1 Stability

The focus of the stability is for a large part getting the image generation and the pipeline to run consistently. The major hurdle in this regard was the lack of stability from VirtualBox as explained in Section 2.6. However, with the workaround we found for VirtualBox we worked on a new implementation for Packer.

8.1.1 Image generation

To test the Image Generation Techniques we repeatedly ran the installation scripts we developed and checked the final state. We tried doing this by running a Smoke test after setup, however, at times this did not get set up correctly. As a result, we have a set of data containing just timing and performances and another set which also includes the smoketest as extra validation for proper setup. During this section, we will focus on the data with the smoke test where sufficient.

When looking at the installation stability data, which is represented in table 8.1, we can see that overall the stability is quite high. We see however that the Cloud-init method can be considered as the outlier having quite a high number of failures, granted the total amount of runs is also quite low. Despite this, our experiences when setting up Cloud-init were also that it could fail quite often, expecting a similar trend if more runs were made.

If we look at the data with which the smoke test was executed we can conclude that except for the Cloud-init, Docker and old packer methods the other options are well within the margins we aim to find.

	Ansible	Cloud-init	Virt-customize	Docker	Packer old	Packer new
Total runs	52	19	57	32	120	140
Success	52	16	57	29	113	139
Failure	0	3	0	3	7	1
Success rate	100%	84%	100%	90%	93.4%	99%

TABLE 8.1: Image Generation success rates among various tools as verified by the smoketest

8.1.2 Pipeline

Stability in the pipeline was also a major concern. When trying out the pipeline the first couple of times there were a lot of small things that failed. This would result in a failed stage in the pipeline which would bring the whole process to a halt to avoid further issues. Although these failures would often be relatively easy to solve they did still add up to the amount of time a developer would need to spend on getting a complete run through the pipeline. Next to the overhead of having to wait multiple times to reach the same state of the pipeline.

In figure 8.1 a Sankey Diagram can be seen depicting the different flows of the pipeline runs. In this diagram we can see that the successful pipeline runs often contain only a few actions, this is because often after a failed run the developer would only run part of the pipeline again from the point that failed onwards. Out of the 100 total runs, only 13 would perform (almost) the whole pipeline.

Additionally, we can see some clustering in the failed pipeline steps in the smoketest failing, this could often happen because although a previous setup step did complete it did not have sufficient post-condition checking to confirm the process was entirely successful. We also observed several times that the Virtual Machines were not properly booted causing some of these issues.

In the new pipeline for the host machine configuration, we have only seen one failure in the last 10 times the pipeline has been run. This issue was caused by a misconfiguration in the host machine which caused the *Apt update* command to fail. A situation which should be far out of scope for such a pipeline to deal with.

For the second pipeline (Deploy Basic Virtual) we also see an increase in the success rate but smaller. Out of the last 55 runs that either Succeeded or failed (not counting runs to timeout due to other issues), we see 23 failures against 32 successes for a success rate of 58%. Of note is the fact that one issue with a chromedriver was recently solved which accounts for 7 of the failures. Not counting these gets us to a passing rate of 16 out of 48 or 66%.

Most of these last 16 issues can be accounted for by odd states the host machine/pipeline should not be able to get to when running the pipeline alone. This can happen when developers want to test partial systems where not all preconditions are met, which will result in a failing pipeline but often with a clear message indicating what went wrong in the process.

In the third pipeline for running the Run system test) we saw three failures among the

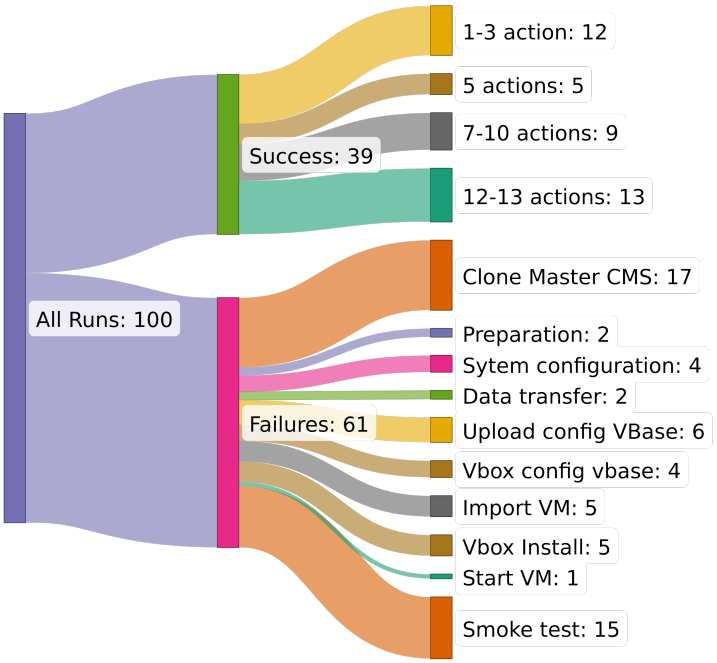

FIGURE 8.1: Sankey Diagram covering the latest 100 test runs on the old pipeline.

last 26 runs of the full pipeline. These three failures seem to be caused by a race condition occurring with the Virtual Machines starting.

8.1.3 Summary

Overall we see that most of the Image Generation Processes score very high, with three of the five new processes succeeding in 99% or more of the cases. Although some of the other tools have lower success rates this is also based on lower numbers of runs so these may be more skewed by a few failures.

Nevertheless, we see a clear difference between Ansible, Virt-customize and the new Packer implementation each having even fewer failures among a much higher amount of tests. In this regard, there is a clear two-split possible with these three tools (Ansible, Virt-customize and the new Packer implementations) against the other three options (Cloud-init, Docker and the old Packer implementations).

This gives rise to the following order:

1. Ansible / Virt-customize
2. Packer New
3. Docker
4. Packer Old
5. Cloud-init

Regarding the pipeline, we also see a huge increase in the amount of successes. We also see a benefit in the new split structure where the use of the pipeline can be better targeted towards the desired effect/actions the user wishes to achieve.

8.2 Automation

In the automation regard, we can focus mostly on the pipelines since the Image Generation process was designed from the bottom up to be fully scripted/automated. As a result, there is not much to compare in this regard.

8.2.1 Image generation

From the aspect of the Image Generation, there was not a lot of difference between the different Image Generation Techniques. All steps could be automated in some form or another. Some tools, like virt-customize, made the transfer of files simpler than others. However, everything we needed to do was possible through automation. As a result, there was no real difference in this aspect between the system Image Generation methods.

As a result, we do not make a ranking among the tools based on automation. They all, eventually, complied with our requirements and expectations and the differences between them feel too minor to make a differentiation between them.

8.2.2 Pipeline

In regards to the pipeline, as mentioned in section 7.4 we believe we have achieved the point where full automation is possible with the new pipeline. This was also our goal. To verify us reaching this stage we have run the second and third pipelines after each other multiple times. During this process, we have not found any issues and the pipeline worked as expected. As mentioned a bad actor could bring the pipeline to a wrong state but we do not believe this to be a concern within our context.

To compare the new and old pipelines we can compare them on two aspects, the preparation actions which are always needed to run a pipeline and the actions which may be incidentally needed based on some issue appearing due to an action being forgotten/executed wrongly or the process failing in some form.

Preparation steps

The preparations needed to perform a new run of the new pipelines have been brought to essentially zero. The user only needs to select the pipeline steps that are desired to be executed, of which the recommended set that is most often used is pre-selected, select the machine they desire the pipeline to execute on, mostly their own machine, and click on the start button.

For the old pipeline, this process was much more involved and included removing the old Virtual Machines, in the correct order, removing the network interfaces and several other steps. As confirmed with employees all these actions could take up to 5 to 10 minutes for each run. Since this pipeline is now used much more as well this amounts to a large time saving in the overall development process.

Incidental repair steps

Besides the regularly required steps at times, some extra steps were required because the pipeline failed or some of the preparation was performed incorrectly. This would stop the pipeline, costing time in the need to restart the process, next to the time dedicated to solving the issue at hand.

Below are some of the issues we encountered and an estimation of the amount of time we estimate solving this issue would take. Additionally, some more context around these issues is given.

- Wrong order of removal of Master CMS from VirtualBox (2 minutes)
  More context is given in section 7.2.2.
- Deleted disks remain in VirtualBox (5 minutes)
  The cause of this issue is unknown, but at times after deleting a Virtual Machine and the accompanying files from the host machine traces would be left inside the VirtualBox configuration. This needs one-by-one deletion from the VirtualBox UI. Often discovering and realising this issue costs a lot more time than solving it.
- Not all processes being cleaned up (1 minute)
  Sometimes some processes would not be cleaned up and be dangling in the background.
- Chromedriver version mismatch (15 minutes)
  The Chromedriver used by Selenium could run out of sync by Google Chrome updating itself in the background. This would require manually installing a newer version, since the official repository by Google did not contain the newest version yet, or downgrading the browser. Although not required often this could be a relatively lengthy process.

8.2.3 Summary

The differences in regards to automation in the Image Generation process are too small to give a meaningful ranking between the different Techniques.

However, for the pipeline, we see much more difference, mainly in the time that is being saved through the additional automation, especially in human labour. Talking with employees at TKH Airport Solutions also reveals that they have started using the pipeline much more during development already since it is a lot easier to use and requires much less effort.

The approach of setting it and forgetting it with full automation is also a feature that is highly praised. Since the trust in the pipeline is much larger they can focus on doing other work while the pipeline is running in the background without the fear of the run having failed after a few minutes.

8.3 Performance

We can split the performance aspect into two fields, timing and resource usage. Since one of our desires is to set up a quicker process such that entire system tests can be run more often a focus is the reduction in time. Additionally, it is a desire that the resource usage doesn't increase too much, such that current hardware is sufficient and to keep costs for resources in mind.

In regards to resource usage, our focus was mostly on the Image Generation process. The pipeline has had too many changes to make a reasonable comparison of the resource usage. One major example of this is the move from the pack/unpack feature of Jenkins to accessing the System Images via the network. This reduces a lot of the CPU usage to perform the compression and decompression for the pack/unpack process. However, we will still compare the base state with the final state to give an overview of the degree of change this brought up.

In regards to the duration of processes, we have more things to compare.

8.3.1 Duration

Looking at the duration of the process we can again look at the image generation and the pipeline separately. For the Pipeline, since it changed so much, we will try to compare the three different stages we developed against a similar set of actions being performed in the old pipeline, since we have seen that timewise the pipeline is fairly stable, we will focus on a small set of timings due to the difficulty of getting old timing data per stage.

Image generation

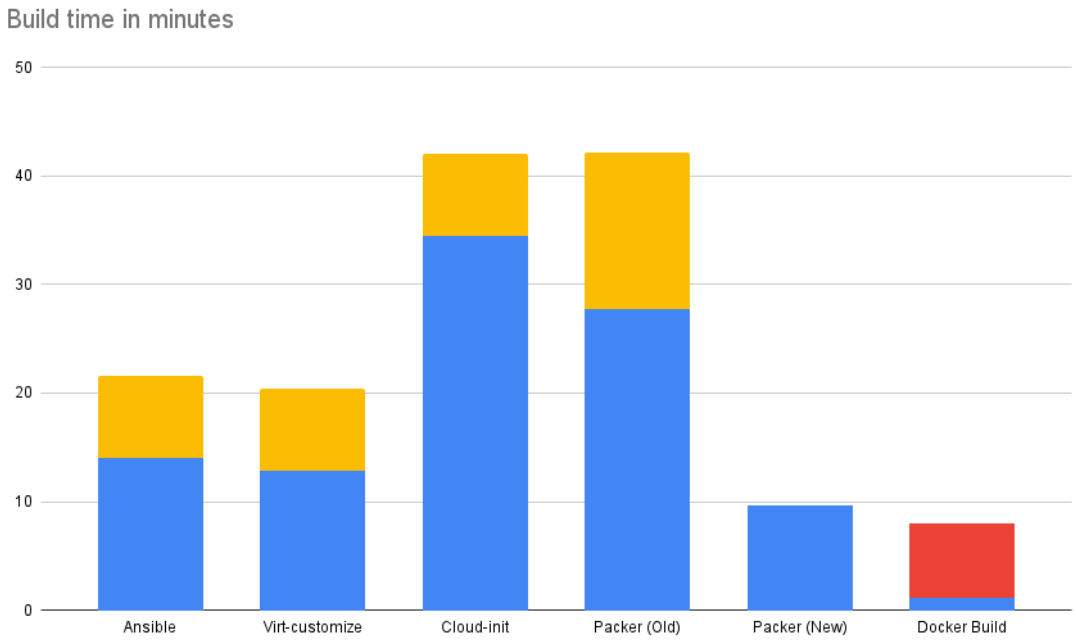

FIGURE 8.2: Building times of Image Generation Techniques in minutes

For the timing of the Image Generation process, we can take a look at the median numbers for each of the processes. This can be found in figure 8.2. The overall time for the process to finish is given in blue, the yellow markings are the time it takes to create a copy of the images for KVM. This is to copy the Base image and start a new installation. In the Packer Old case, this is the time it took to export the images to new files. For Packer New, this could not be measured because multiple processes happened at the same time due to it being parallelized. In this case, the time is included.

For Docker the blue time is how long it takes the Docker Builder to create the new images and containers, in red the time indicates how long the installation of the components takes after starting the containers.

Additionally, the data from all different runs is combined in figure 8.3. This graph shows the progress of build times of the first 60 runs. This image also shows the overall stability of the build times, a small upward trend can be seen with some implementations. A suspected reason for this is the host machine becoming slower due to more background processes having been started and more caches and other system resources filling up.

Overall we can see that the new Packer method is much quicker than the old method, almost 3 times as fast as the building alone. Additionally, the building and exporting of

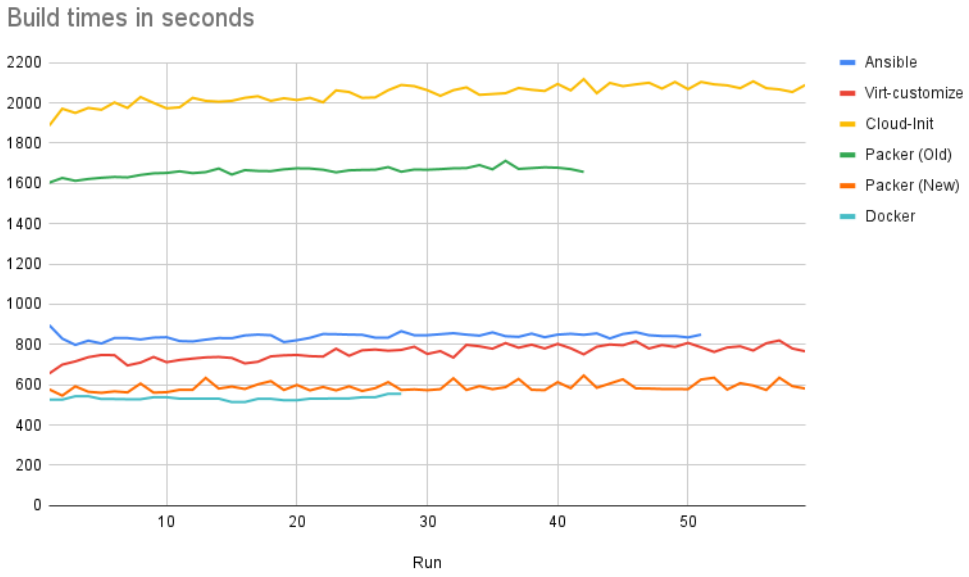

FIGURE 8.3: Build times over multiple runs for different generation techniques

the images alone is already faster than the exporting of the previous images. This can be explained by the exporting happening at the same time as opposed to one after the other. By using this we make much more efficient use of spare CPU and disk IO resources reducing the overall time.

Pipeline

As mentioned in chapter 7 Pipeline redesign we remodelled the pipeline into three separate pipelines each accomplishing a specific goal. In table 8.2 we compare the duration of these separate pipelines to the old state. A note must be taken of the fact that the release test was not yet part of the old pipeline and was triggered manually. Additionally, some tasks were performed in a different order. We attempt to map each previous stage to a new one as closely as possible.

One major roadblock with comparing the data is the first pipeline for the Host machine setup. The duration of these tasks relies very heavily on the previous state of the host machine. For example, some libraries that need to be compiled on the host which adds a great amount of time, or other parts are already being installed. Since the data coming from these statistics is so unreliable we have opted to leave these statistics out.

Most of the comparable data is therefore in the second pipeline. From this pipeline, we will therefore compare the relevant points.

8.3.2 Resource usage

Measuring the performance of each stage in regards to the resource usage we will take a look at the CPU time, peak memory for a process and peak memory for all processes combined.

	Old state	New state	Remarks
Stash/Unstash	15 minutes	Removed	This step got removed in the new pipeline in favour of accessing the files directly over a network mount.
Update python libraries	New	4 seconds	This describes the overhead caused by this action since it has not yet happened while we have had the pipeline in production.
VirtualBox removes old VMs	Manual Action	5 seconds	Some additional time may be required if there were still VMs running, sending the stop command and 5 seconds of sleep are added in such a case.
Import VMs	2 Minutes 20 seconds	2 Minutes 30 seconds	This action happens over a network mount. This may take longer if multiple developers are importing images at the same time. However, the network overhead is minimal compared to importing from the same machine with the old pipeline. Compared to the reduced pack/unpack time which was previously needed it is very unlikely for this combined process overall to take longer.
Network configuration	1 second	2 seconds	
Clone CMS	33 seconds	33 seconds	
Start Virtual Machines	37 seconds	1-25 seconds	Adaptive timing in new pipeline. Can directly continue if VMs are already running and when starting looks when ready instead of naive 30 second sleep.
Upload Virtual Basestation configuration	3 seconds	4 seconds	Handles existing configuration without crashing.
Upload Master AM configuration	38 seconds	1 minute 10 seconds	Many more failsafe features implementing costing extra time to perform the checks.
Run smoketest	22 seconds	22 seconds	

TABLE 8.2: Durations of various pipeline tasks in the old and the new pipeline situation.

Image generation

During the Image Generation process, we kept track of the resource usage of each of the tools. we must note that this does not include the Virtual Environments themselves, this is for a large part due to their trickiness in attaching monitoring tools to them. As well as that we noticed that the resources they took seemed to differ minimally between the tools, as such we left it out of scope for this research. As a result, the following numbers focus solely on the tooling being used for generating the images.

As mentioned in the methodology we measured the CPU in time it was busy performing the operations as a total number in seconds, in the graph displayed in 10th of seconds for better scaling. We also measured the peak memory consumption of a single process within the stack and the overall peak usage by all tooling combined. The median numbers for each tool can be found in figure 8.4. it can be seen that the memory usage of the tooling is staying below 1GB of memory. This is in line with a Running Virtual Machine.

More detailed information is available in appendix A.2 with a graph displaying the resource usage over consecutive runs for each of the different Image Generation techniques. These graphs also show the relative consistency between runs in most of the tooling.

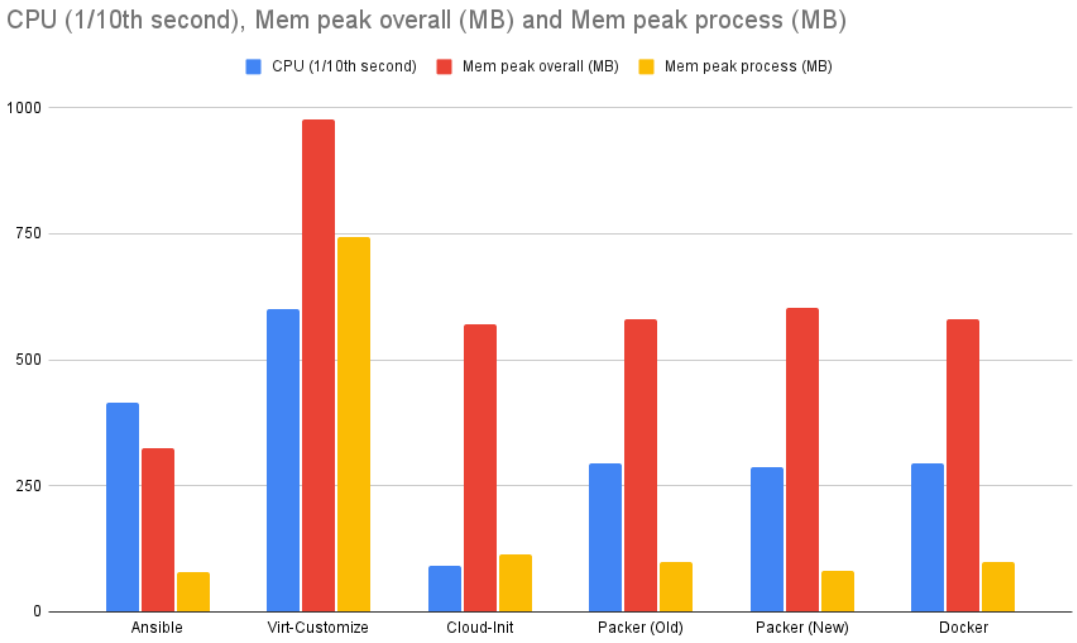

FIGURE 8.4: Differences in resource usage between Image Generation techniques

8.3.3 Pipeline

Testing the pipeline for system performance is not very comparable with the complete overhaul we did. Many tasks are sped up or removed/reworked, for example, the pack and unpack operations on the Virtual Images. Other tasks got extra checks to improve error reporting or recover from certain states the previous pipeline could not recover from, for example, the Master AM configuration.

Additionally, one major change we made was including the Release test in the pipeline. This causes the overall runtime of a pipeline to be significantly longer. Due to the release test simply taking quite a while.

What we have elected to do is compare two select sets of actions that are performed in both pipelines and compare these. These actions start the Virtual Machines, configure the CEDD AGL system and then run a smoketest. We only selected the runs that had a passing smoke test.

The results of the resource usage during this process can be found in figure 8.5. From these results, we can see that there is a minimal increase in resource usage between either pipeline. Most of this can be accounted for by the Chromedriver performing the extra checks that are performed in configuring the Master AM. Analysing the raw details we also noted that there were a lot of different Chromedriver processes with each having a low memory footprint but which quickly add up together. This is logical by following the design of chrome[1]

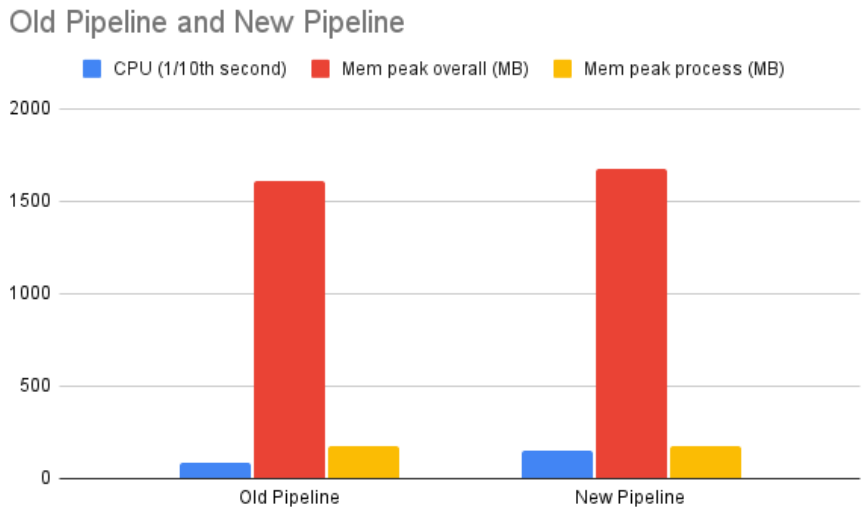

FIGURE 8.5: Resource usage between a select set of actions in the old and the new pipelines

8.3.4 Summary

Overall we see a fairly similar footprint between each of the Image Generation techniques, where there can be a relatively large difference in resource usage note must be taken that this does not include the running of the CEDD AGL Systems. This was simply too complex for us to keep in our scope. It must also be taken into account that although there are some larger differences the scale is relatively small for an enterprise situation. Currently, most computer systems on the market should have plenty of resources to manage the current resource usage and the difference of half a GB should be negligible. To give a reference this can be achieved by closing a few browser tabs since they each can easily take around 100MB of ram[2].

Based on the timing we see two clear leaders in the speed department, the Docker and the New Packet methods. These have the clear advantage of being twice as fast as the next close pair, being Ansible and Virt-customize. The slowest two options could be explained

[1] Although this answer talks about the situation in Windows. the design is similar under other operating systems https://stackoverflow.com/a/58521183

[2] https://cloudzy.com/blog/which-browsers-use-the-least-memory/

by Cloud-init having a lot of overhead for installing the Base operating system and by being a sequential process where all others were (mostly) parallelized.

Based on the fact that the resource usage should be negligible on our scale and most of our concern lies with a fast build process we can rank our methods according to the Build times.

This gives rise to the following order:

1. Docker
2. Packer New
3. Virt-customize
4. Ansible
5. Cloud-init
6. Packer Old

8.4 Company impact

This metric is quite abstract but we will estimate how much effort it would be to employ each of the new Technologies in the company. Additionally, we attempt to look at more quantifiable aspects as mentioned in the Methodology. We will present this data in a table and also give a short description of our expectations of how such a tool may be handled within the company. Within the table, we will score each of the aspects in three tiers from low (-), medium (=) to high (+), the optimal score being high. Most of our focus will lie on the learning curve and maintainability scores since these will have the most direct impact.

Looking at the (perceived) complexity of each of the Virtualization Technologies and the Image Generation tools we can order them by what impact they would have on the usability within the company. For this research, we will take the current skillset and experience of the employees into account which very likely is not representative for every company. We urge the reader to take their situation into account.

As we already mentioned Packer and Docker are both tools that are already used within the company. More specifically Packer is already being used for this process. As such this would play mostly into the already existing experience and knowledge, especially in regards to fixing potential future issues.

Next to this, the Cloud-init is already partially used for generating the base image. This tool should require minimal to no guidance in being set up. It, however, lacks some of the handlebars around it, that other tools like Packer and Docker offer, making automation more difficult.

The Virt-customize process is unused and very new, it however makes things fairly easy using scripting, be it in a new manner, we believe this tool should be able to be picked up with a minimal learning curve.

Ansible is the tool most outstanding in this grouping, it is a completely new type of Technology for configuring the host machines, to our knowledge no tools like it are being used within the company. This also makes learning more difficult, it does however have a lot of online presence with many tutorials and a lot of documentation easing the difficulty of the learning curve. Ansible still takes a long time to learn thoroughly, and although not always required, it gives this the bottom pick among the tooling.

Summary Overall we see very good scores from Docker and Packer, as mentioned these tools are already used but besides that, these tools also offer a large amount of documentation, tutorials and guidance. They are more rigid and less flexible than Ansible but this

	Learning Curve	Maintain-ability	Documentation	Tutorials	Plugins/ flexibility	Overall
Packer	+	+	=	=	=	+
Docker	+	+	+	+	=	+
Virt-customize	=	=	=	=	-	=
Ansible	-	=	+	+	+	=
Cloud-init	=	-	=	=	-	-

TABLE 8.3: Evaluation of the company impact using various Image Generation techniques

tool suffers a lot from its high learning curve and complexity.

Virt-customize was a very pleasant tool to work with, the documentation and online resources are good but not as abundant as Docker and Packer. Cloud-init scores the worst. This is for a large part due to it being an Ubuntu-only system which does not have the most complete and compelling online presence among documentation, tutorials and guidance.

This gives rise to the following order:

1. Packer New/Packer Old
2. Docker
3. Virt-customize
4. Ansible
5. Cloud-init

8.5 Overview

Overall the picture for the new pipeline is clear. It offers many improvements in automation and stability, where we even achieved the point that it can run fully autonomously. However, this does come at a cost of slightly more resource usage and longer duration in similar tasks. This effect is for a large part mitigated by removing the very costly old process for transferring the System Images to the developer's machine. By using a network mount this process is hugely more efficient saving up to 15 minutes per run.

The situation for the Image Generation process is more nuanced, here we see distinct advantages for different aspects of each Technology. To give a better overview we present table 8.4 which contains the ordering of ranking of the Image Generation Technologies among the different metrics.

	Stability	Automation	Performance	Company Impact	Overall score by position[3]
Packer New	3	1	2	1	7
Docker	4	1	1	3	9
Virt-customize	1	1	3	4	10
Ansible	1	1	4	5	11
Packer Old	5	1	6	1	13
Cloud-init	6	1	5	6	16

TABLE 8.4: Ranking of Image Generation tools along the different metrics. Ordered from best overall scoring to worst scoring

[3]Lower score is better, this number is the sum of all positions.

Chapter 9

Conclusion

For a quick and fast conclusion we can look at the comparison between the advantages of the Virtualization Technologies and the Results of the Image Generation process, especially the ranking in the Overview. We can see a clear preference for VirtualBox with the new Packer implementations. After finding a workaround for the stability issues with VirtualBox it became a very enticing option due to it outscoring the other options in most of our metrics.

Combining this knowledge with the fact that this Technology is already well implemented in the processes of TKH Airport Solutions we see no clear reason to switch to a different Virtualization Technology or a different Image Generation process.

Since our advice is to stay with the current Virtualization Technology, we could also put more of our efforts into improving the pipeline as opposed to adapting/implementing a new Technology in the processes. The clear result of this is the fact that we can run the pipeline fully autonomously without the need for human interaction whereas before multiple manual steps were required as preparation before the pipeline could be run.

In the following sections, we will dive deeper into a more formal answer to each of our research questions and also give some recommendations about potential future changes. At the end, we will give several suggestions for potential future works.

9.1 How can we improve the process of generating new system images?

When looking at the accumulated scores of the Virtualization Technologies when we made our selection in Section 5.3.2 and the results of the Image Generation processes in Section 8.5, we can see that VirtualBox with the new Packer implementation scores as the optimal combination. Thanks to our find of the workaround fo the VirtualBox stability issuer it has also become a viable option again. With the added consideration that VirtualBox is already a well-known and widely used tool within the company, there is in our opinion no doubt as to why VirtualBox is the obvious best choice.

Besides taking VirtualBox with Packer as the optimal solution there were however still some changes that we could implement to improve the process compared to the old state. This included building each of the CEDD AGL System components in parallel, where they previously were generated consecutively. These improvements combined has been shown to reduce the time it takes to a quarter of the original.

As with any consideration and comparison, each factor may play a different role in the future or for other entities. Interestingly we can note that our final ranking would not differ if we take the company impact score out. Without consideration of the present knowledge and processes at a company, we believe that this advice could be universal. However,

having gained a lot of experience with each of the tools we also see a lot of potential for the other Image Generation tools in other business cases.

Virt-Customize The Virt-customize tool was a very pleasant tool to work with, it often worked very well and we had little trouble making various implementations. Although the resource usage was marginally higher than the other tools it had a great reliability. Where it fell short was that it has no support for VirtualBox Images and only supports KVM. This makes it less optimal since KVM did score lower on our usability score than VirtualBox. However, for a case where only KVM needs to be supported, we are sure that this tool could be a great fit.

Cloud-Init The situation where Cloud-init really could shine is installations on bare metal from scratch. When for example, a new installation is made on a hardware server, attention has to be paid to the compatibility of the libraries that are installed. Using an installer integrated into the Operating system that handles such dependencies could be a great solution in favor of custom-created tooling.

9.2 How can we improve the testing pipeline?

Our main advancement in the system testing process was full automation from setting up the system on a host machine to getting a test report. This has been achieved by looking at which tasks were done manually and finding solutions for automating these. Additionally, we have looked at ways the pipeline can trip up/end up in erroneous states and found fail-safes or detection for these states. If possible we have implemented workarounds for these situations such that the pipeline could keep working. Where this would be too complex to achieve we have implemented clear warnings for the user such that they can easily identify these issues. This way they can work on a solution manually without the need to do a lot of complex debugging.

As always it is not possible to account for every possible situation, as such when an error is detected the pipeline stops running and the latest output is available to the user to debug this. We have also kept the option to select which parts of the pipeline are to be executed. This will allow the user to quickly get back to the point that failed without the need to wait on other lengthy processes that are not required in the current state.

Overall we saw a great speedup by working around the stash/unstash step with a network mount and overall increased the stability and usability of the pipeline. After talking with some of the developers and testers they also indicated using the pipeline a lot more compared to the old situation, since the new pipeline saves them a lot of effort. Where previously due to the stability issues they often would resort to manually running each of the steps. Additionally, a great effect this has had is the fact that they have become much more comfortable in trying new things without worrying about breaking the system, since rolling back to a fresh installation is much easier.

9.3 How can we improve the technical process around the system testing performed at TKH Airport Solutions?

Overall we have introduced several improvements to the system testing process at TKH Airport Solutions. These include speeding up and stabilizing the process of creating System

Images as well as improving the pipeline itself that is used to optimize the running of the system tests.

Our most impactful change must be finding, and implementing, the workaround to the stability issues with VirtualBox, After finding out that, besides the stability, VirtualBox was one of the most interesting Virtualization Technologies, and along this also had a very impressive Image Generation process with Packer. It was a very important find to allow us to continue using VirtualBox which has shown to be the best option. Where at first the aim was set on finding an alternative technology this find allowed us to select the optimal solution instead.

Next to the Image Generation process we have also automated or circumvented all the preparation tasks that were required to run the pipeline, allowing full automation that could be triggered from the pipeline based on a new version being built.

What this research has taught us however is that in the business context of TKH Airport Solutions the most optimal Image Generation process and the best Virtualization Technology is the combination of VirtualBox with Packer. Additionally, we have found and implemented a large number of improvements to the pipeline granting an overall better experience which has resulted in more employees making good use of the new pipeline.

9.4 Future work

There are several topics we had wished to look further into but did not have the time or resources for. In this chapter, we will introduce what we think are some interesting open-ended questions that could be picked up in the future.

Deeper look into the VirtualBox stability issue to find the root cause and solve this. Although the issue with VirtualBox's stability has been solved to a certain extent with our workaround the root cause has still not been identified. It could be important to identify this and solve it in the future if for example it is found that a different para-virtualization setting is required. Additionally, this might be a bug inside VirtualBox or elsewhere that could impact other users without their direct knowledge.

Analyze resource usage of Virtual Machines/Containers During this research, we decided to leave the tracking of the resource usage of the Virtual Environments out of our scope due to the complexity of tracking this. Each different technology showed a different way of spawning the Virtual Environment and had different, sometimes very rudimentary, tooling to track different parts of the resource usage.

Future work could look into making a framework, wrapper or other kind of tool to more easily track these resources. With this kind of tool at hand, another look could be taken at the differences in the installation between the tools. This time attention could be paid to the resource usage of the internal processes besides the tooling alone.

Analyse the Image Generation process with VMWare Due to time constraints, we had to leave VMWare out of the scope for this research, we believe however that in certain business cases, VMWare could be a very powerful alternative to VirtualBox. A future work could therefore consider this tool and see if it would fit a situation where licensing costs are less of a concern.

Bibliography

[1] MinSu Chae, HwaMin Lee, and Kiyeol Lee. "A Performance Comparison of Linux Containers and Virtual Machines Using Docker and KVM". In: *Cluster Comput* 22.1 (Jan. 1, 2019), pp. 1765–1775. ISSN: 1573-7543. DOI: 10.1007/s10586-017-1511-2. URL: https://doi.org/10.1007/s10586-017-1511-2 (visited on 05/02/2023).

[2] Filip Široký. "Thesis Report Filip Siroky: Performance and Virtualization Trade-offs". Master. Odense, Denmark, Aug. 1, 2020. 79 pp. URL: https://gitlab.com/phillwide/vtmark/-/raw/master/Thesis%20Report%20Filip%20Siroky.pdf (visited on 05/02/2023).

[3] Saverio Giallorenzo et al. "Virtualization Costs: Benchmarking Containers and Virtual Machines Against Bare-Metal". In: *SN COMPUT. SCI.* 2.5 (Aug. 7, 2021), p. 404. ISSN: 2661-8907. DOI: 10.1007/s42979-021-00781-8. URL: https://doi.org/10.1007/s42979-021-00781-8 (visited on 05/02/2023).

[4] Jianwei Hao et al. "An Empirical Analysis of VM Startup Times in Public IaaS Clouds". In: *2021 IEEE 14th Int. Conf. Cloud Comput. CLOUD*. 2021 IEEE 14th International Conference on Cloud Computing (CLOUD). Sept. 2021, pp. 398–403. DOI: 10.1109/CLOUD53861.2021.00053.

[5] Ashish Lingayat, Ranjana R. Badre, and Anil Kumar Gupta. "Performance Evaluation for Deploying Docker Containers On Baremetal and Virtual Machine". In: *2018 3rd Int. Conf. Commun. Electron. Syst. ICCES*. 2018 3rd International Conference on Communication and Electronics Systems (ICCES). Oct. 2018, pp. 1019–1023. DOI: 10.1109/CESYS.2018.8723998.

[6] Ming Mao and Marty Humphrey. "A Performance Study on the VM Startup Time in the Cloud". In: *2012 IEEE Fifth Int. Conf. Cloud Comput.* 2012 IEEE Fifth International Conference on Cloud Computing. June 2012, pp. 423–430. DOI: 10.1109/CLOUD.2012.103.

[7] Amit M Potdar et al. "Performance Evaluation of Docker Container and Virtual Machine". In: *Procedia Computer Science*. Third International Conference on Computing and Network Communications (CoCoNet'19) 171 (Jan. 1, 2020), pp. 1419–1428. ISSN: 1877-0509. DOI: 10.1016/j.procs.2020.04.152. URL: https://www.sciencedirect.com/science/article/pii/S1877050920311315 (visited on 04/25/2023).

[8] Charanjot Singh et al. "Comparison of Different CI/CD Tools Integrated with Cloud Platform". In: *2019 9th Int. Conf. Cloud Comput. Data Sci. Eng. Conflu.* 2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence). Jan. 2019, pp. 7–12. DOI: 10.1109/CONFLUENCE.2019.8776985.

[9] Błażej Święcicki. “A Novel Approach to Automating Operating System Configuration Management”. In: *Inf. Syst. Archit. Technol. Proc. 36th Int. Conf. Inf. Syst. Archit. Technol. – ISAT 2015 – Part II*. Ed. by Adam Grzech et al. Advances in Intelligent Systems and Computing. Cited By :1. Cham: Springer International Publishing, 2016, pp. 131–142. ISBN: 978-3-319-28561-0. DOI: 10.1007/978-3-319-28561-0_10.

[10] Ossi Taipale et al. “Trade-off between Automated and Manual Software Testing”. In: *Int J Syst Assur Eng Manag* 2.2 (June 1, 2011), pp. 114–125. ISSN: 0976-4348. DOI: 10.1007/s13198-011-0065-6. URL: https://doi.org/10.1007/s13198-011-0065-6 (visited on 06/07/2023).

[11] Minaoar Hossain Tanzil et al. “A Mixed Method Study of DevOps Challenges”. In: *Information and Software Technology* 161 (Sept. 1, 2023), p. 107244. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2023.107244. URL: https://www.sciencedirect.com/science/article/pii/S0950584923000988 (visited on 06/06/2023).

[12] Kim Torberntsson and Ylva Rydin. *A Study of Configuration Management Systems : Solutions for Deployment and Configurationof Software in a Cloud Environment*. 2014. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-228139 (visited on 06/13/2023).

[13] Sander van der Burg and Eelco Dolstra. “Automating System Tests Using Declarative Virtual Machines”. In: *2010 IEEE 21st Int. Symp. Softw. Reliab. Eng.* 2010 IEEE 21st International Symposium on Software Reliability Engineering. Nov. 2010, pp. 181–190. DOI: 10.1109/ISSRE.2010.34.

[14] Yiwen Wu et al. “An Empirical Study of Build Failures in the Docker Context”. In: *Proc. 17th Int. Conf. Min. Softw. Repos.* MSR ’20. New York, NY, USA: Association for Computing Machinery, Sept. 18, 2020, pp. 76–80. ISBN: 978-1-4503-7517-7. DOI: 10.1145/3379597.3387483. URL: https://dl.acm.org/doi/10.1145/3379597.3387483 (visited on 06/13/2023).

Appendix A

Graphs

A.1 VirtualBox reliability

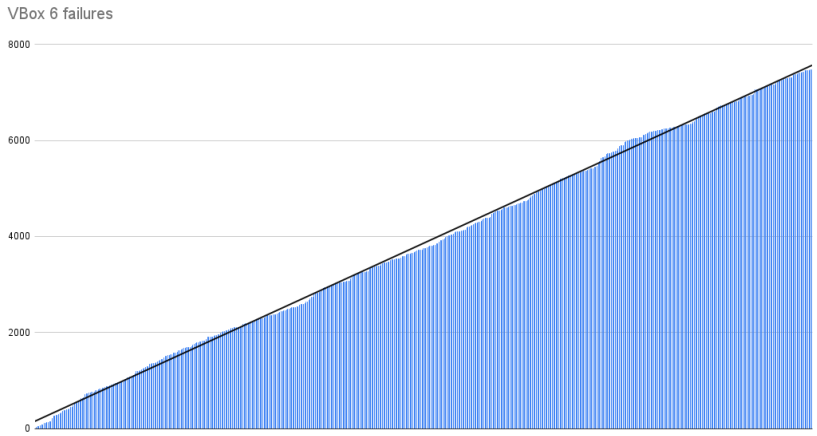

FIGURE A.1: Failures over time in VirtualBox Version 6

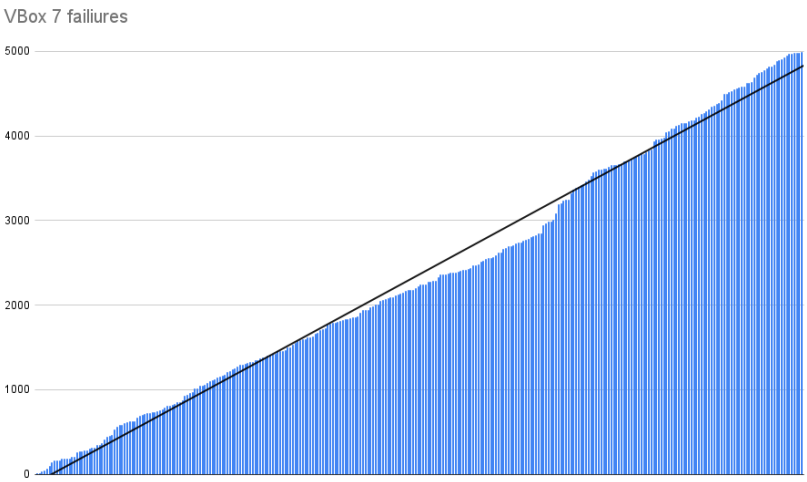

FIGURE A.2: Failures over time in VirtualBox Version 7

A.2 Resource usage different Image Generation processes

This section lists the various resource usages over multiple iterations of the build process. Some graphs have been minimally modified to remove the outliers with failed build results.

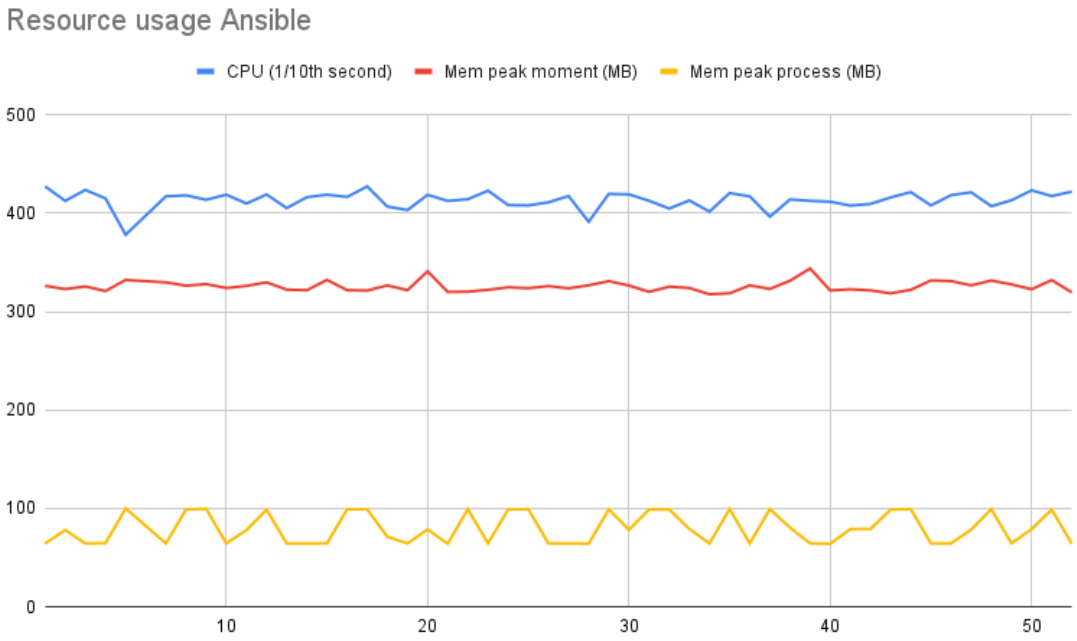

FIGURE A.3: Resource usage over multiple runs for Ansible image generation method.

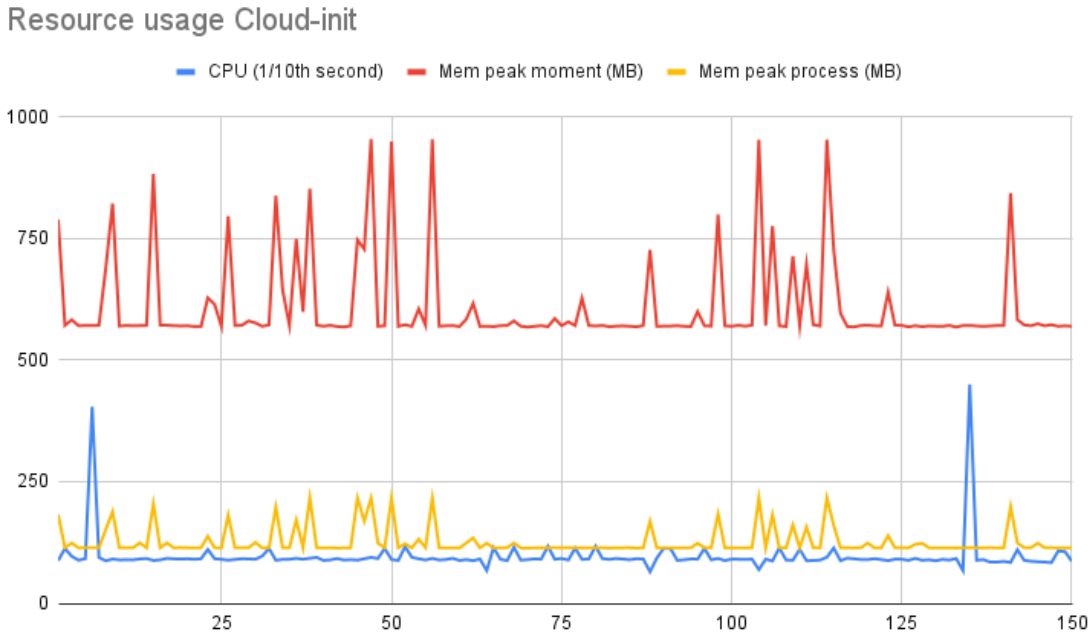

FIGURE A.4: Resource usage over multiple runs for Cloud-init image generation method.

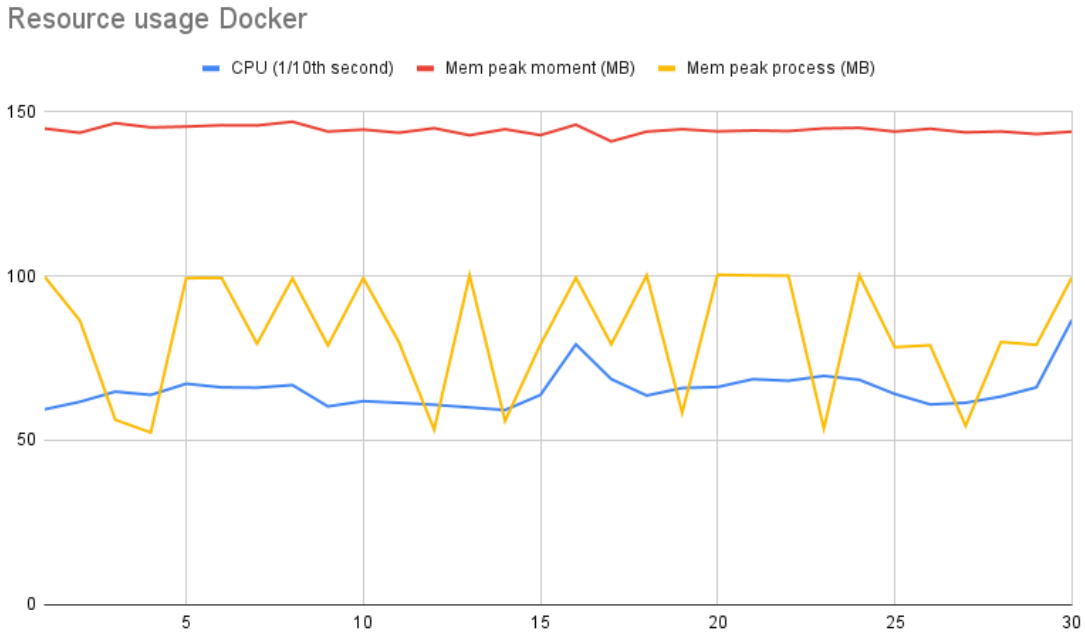

FIGURE A.5: Resource usage over multiple runs for Docker image generation method.

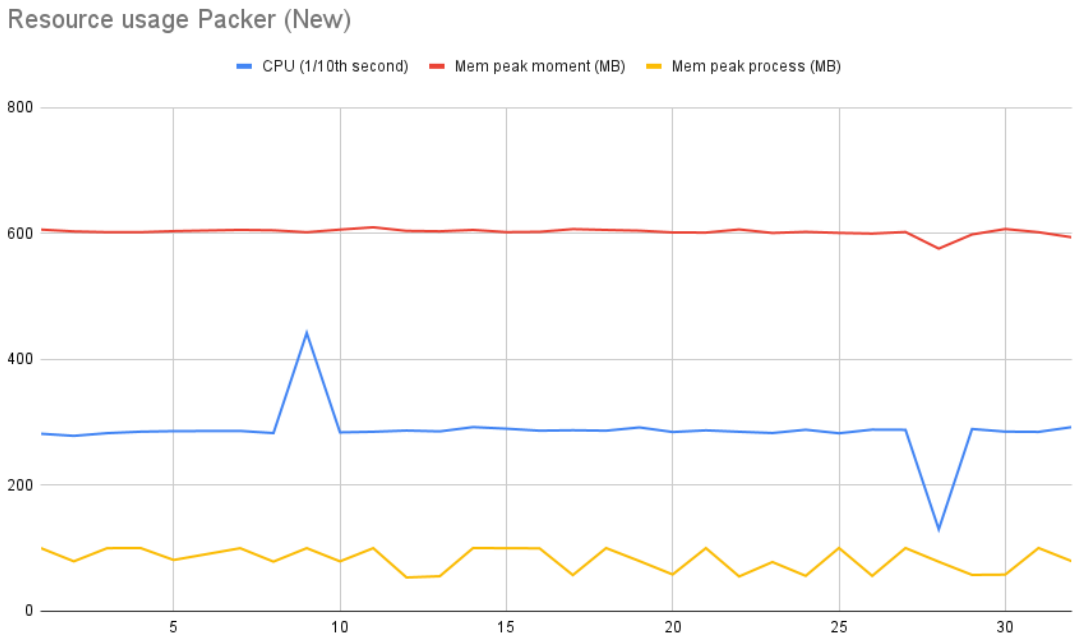

FIGURE A.6: Resource usage over multiple runs for Packer (New) image generation method.

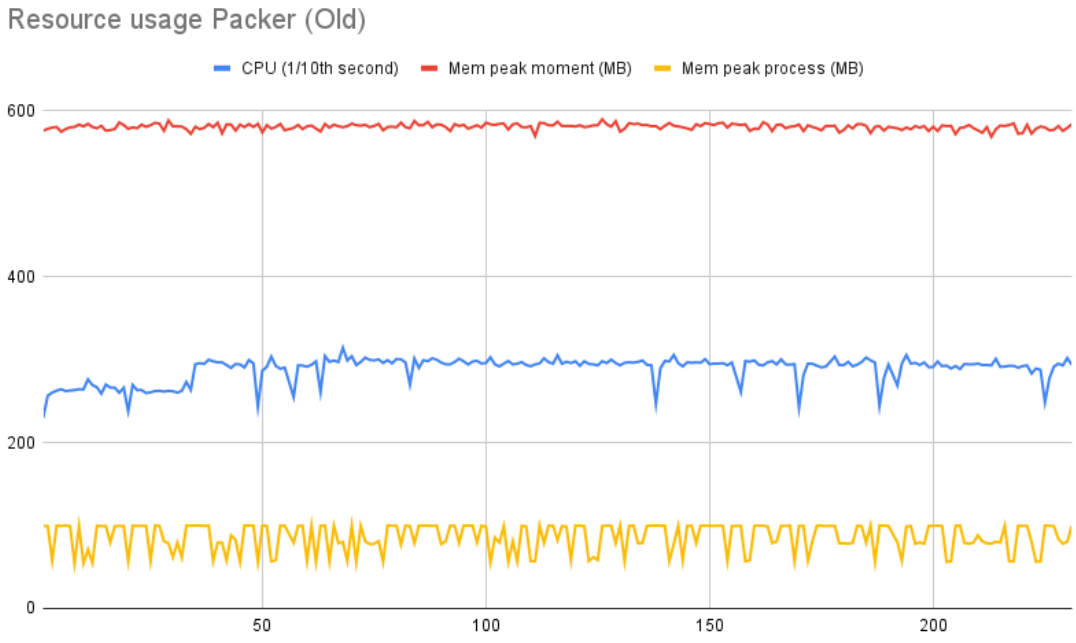

FIGURE A.7: Resource usage over multiple runs for Packer (Old) generation method.

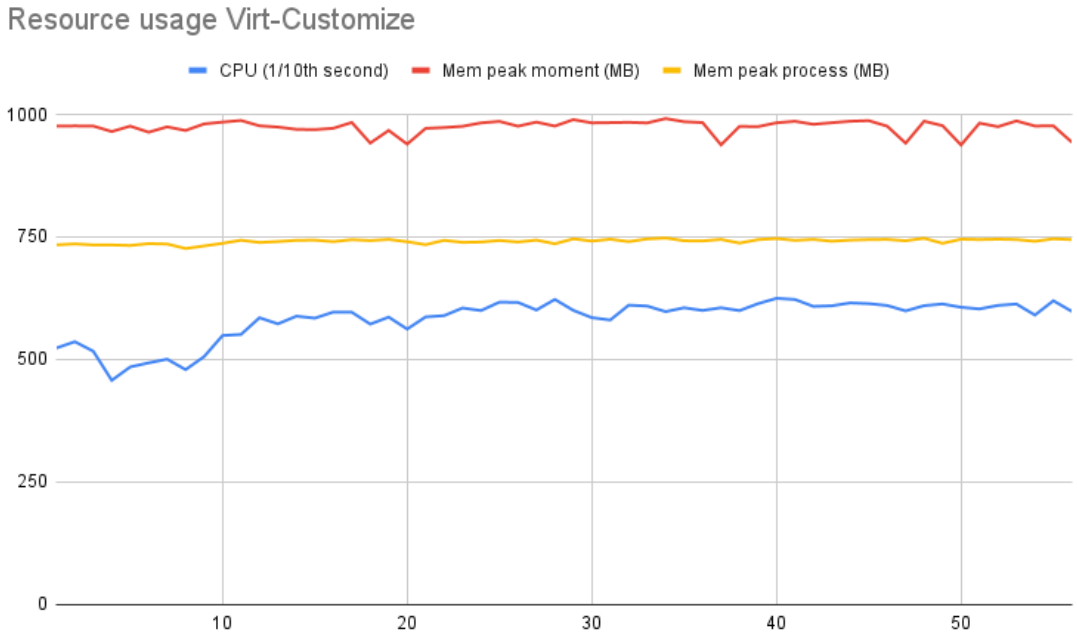

FIGURE A.8: Resource usage over multiple runs for Virt-customize image generation method.