TU WIEN Informatics

From Infrastructure to Code

Generating Dockerfiles from Server Snapshots

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Moritz Ilg, BSc
Matrikelnummer 11776869

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Wien, 29. Jänner 2024

Moritz Ilg   Jürgen Cito

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Informatics

From Infrastruktur to Code

Generating Dockerfiles from Server Snapshots

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Moritz Ilg, BSc
Registration Number 11776869

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc

Vienna, 29th January, 2024

Moritz Ilg   Jürgen Cito

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Moritz Ilg, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 29. Jänner 2024

Moritz Ilg

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Kurzfassung

Diese Arbeit untersucht die Transformation von Legacy-Systemen in Cloud-Infrastrukturen, mit einem besonderen Fokus auf die Automatisierung des Migrationsprozesses unter Verwendung eines neuartigen Tools. Dieses Tool ist darauf ausgelegt, Dockerfiles aus Snapshots von virtuellen Maschinen (VM) zu generieren und spricht damit den wachsenden Bedarf von Unternehmen an, ihre komplexen, über Jahre gewachsenen Legacy-Systeme effizient in Cloud-Dienste zu überführen. Diese Systeme sind oft mit komplizierten Abhängigkeiten und Konfigurationen belastet, was erhebliche Herausforderungen bei der Migration darstellt.

Die Forschung führt eine Methodik ein, um VMs zu analysieren und die Migration von On-Premise-Servern in die Cloud zu vereinfachen. Dies wird durch den Einsatz von Containerisierungstechnologien wie Docker und LXC erreicht, die bei der Bewältigung der oft in Legacy-Systemen anzutreffenden "Dependency Hellëine wichtige Rolle spielen. Insbesondere Docker ist entscheidend für die Isolierung von Abhängigkeiten und somit für die Vermeidung potenzieller versions Konflikte. Darüber hinaus befasst sich die Studie mit verschiedenen Werkzeugen und Praktiken zur Erstellung zuverlässiger Konfigurationsmanagement-Skripte, einem Schlüsselaspekt für einen reibungslosen Übergang in Cloud-Umgebungen.

Ein bedeutender Beitrag dieser Dissertation ist die Entwicklung des Tools, das die Extraktion von Bereitstellungsskripten von On-Premise-Servern automatisiert. Dieses Tool vereinfacht nicht nur die Neuimplementierung dieser Systeme auf Cloud-Plattformen, sondern konzentriert sich auch auf die Erstellung von Dockerfiles, die speziell für Migrationen zur Infrastructure as a Service (IaaS) und Platform as a Service (PaaS) angepasst sind. Dadurch wird eine der grundlegenden Herausforderungen bei der Cloud-Migration angesprochen: die Umwandlung komplexer, alter Infrastrukturen in cloud-kompatible Formate.

Diese Arbeit skizziert auch potenzielle Bereiche für zukünftige Forschungen, einschließlich der Untersuchung komplexerer Systeme, die mehrere miteinander verbundene Dienste wie Datenbanken, Backends und Frontends umfassen, sowie die Entwicklung umfassender Docker Compose-Dateien zur nahtlosen Integration dieser Dienste.

Insgesamt präsentiert diese Arbeit einen strukturierten Ansatz zur Migration von Legacy-Systemen und bietet Einblicke und Werkzeuge, die den Übergang von Unternehmen zu cloud-basierten Lösungen erheblich erleichtern und beschleunigen können.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Abstract

This thesis investigates the transformation of legacy systems into cloud infrastructure, with a particular focus on automating the migration process using a novel tool. This tool is designed to generate Dockerfiles from virtual machine (VM) snapshots, addressing the increasing need for businesses to efficiently move their complex, legacy systems to cloud services. These systems are often laden with intricate dependencies and configurations accumulated over years, posing significant migration challenges.

The research introduces a methodology to analyze VMs and streamline the migration of on-premise servers to the cloud. This is achieved using containerization technologies like Docker and LXC, which are instrumental in managing the "Dependency hell" often encountered in legacy systems. Docker, in particular, plays a crucial role in isolating dependencies, thus mitigating potential conflicts. Additionally, the study delves into various tools and practices for creating reliable configuration management scripts, a key aspect of ensuring smooth transitions to cloud environments.

A significant contribution of this thesis is the development of a tool that automates the extraction of deployment scripts from on-premise servers. This tool not only simplifies the redeployment of these systems into cloud platforms but also focuses on generating Dockerfiles that are tailored for Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) migrations. By doing so, it addresses one of the fundamental challenges in cloud migration - transforming complex, legacy infrastructures into cloud-compatible formats.

The thesis also outlines potential areas for future research, including the exploration of more complex systems involving multiple interconnected services such as databases, backends, and frontends, and the development of comprehensive Docker Compose files to integrate these services seamlessly.

Overall, this thesis presents a structured approach to legacy system migration, offering insights and tools that can significantly ease and expedite the transition of businesses to cloud-based solutions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung vii

Abstract ix

Contents xi

1 Motivation 1

2 Related Work 3

3 Background 5
- 3.1 Operating Systems 5
- 3.2 Package manager 7
- 3.3 Dependency hell 8
- 3.4 Strategies for Reproducible System Builds 8

4 Approach 21
- 4.1 Theoretical Approach 21
- 4.2 Implementation 31
- 4.3 Limitations 34

5 Evaluation 41
- 5.1 Methodology 41

6 Conclusion 53
- 6.1 Future Work 54

List of Figures 55

List of Tables 57

List of Listings 59

Bibliography 61

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Motivation

The market for cloud infrastructure services is witnessing a rapid expansion. An increasing trend towards cloud migration is being observed among companies, moving away from traditional on-premise infrastructures. This shift has been further accelerated by the COVID-19 crisis, with a significant surge in cloud adoption [13, 12]. In the first quarter of 2022 alone, business expanses on cloud services reached nearly $53 billion, marking a 34% increase over the previous year [14].

Despite this growth, the transition of long-standing on-premise servers to the cloud presents considerable challenges. These servers, often operational for decades, represent a complex patchwork of configurations and customizations. Over the years, they have been altered by various individuals through the installation and removal of programs, updates to configuration files, and additions of scripts. A common issue faced by many companies is the lack of comprehensive understanding of the precise functions and dependencies of these servers. This uncertainty often leads to reluctance in modifying or migrating these systems, for fear of disrupting existing infrastructure.

The unique complexity of each server makes it difficult to establish common procedures and effective migration tools, leading migration experts to rely on their own expertise and simple tools for successful cloud migration, as noted in recent comparative studies [26]. Consequently, migration experts are forced to rely heavily on their personal expertise and rudimentary tools in orchestrating successful cloud transitions.

This thesis aims to address these challenges by introducing an approach that assists migration experts in redeploying on-premise servers to the cloud. The tool extracts simple deployment scripts from existing machines, facilitating a more efficient and less error-prone migration process.

This work introduces a new way to analyze VMs and create Dockerfiles from their snapshots. It simplifies, turning complex, old systems into formats that work well in the cloud. The research covers examining dependencies, setting up file systems, managing

environmental variables and starting the necessary applications. This leads to making Dockerfiles that fit well with IaaS and PaaS migrations. By making these processes automatic, the tool cuts down the manual work needed for cloud migration, making it easier for businesses to move their systems to the cloud.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 2

Related Work

As cloud computing increasingly becomes a cornerstone of modern IT infrastructure, the shift towards cloud adoption is gaining momentum. Various strategies for hosting data and systems in the cloud have emerged, each with its own set of methodologies and challenges.

Pahl and Xiong conducted a comprehensive study [11] that compared migration strategies on-premise to cloud. Their research focuses primarily on three cloud hosting solutions: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). The study aims to dissect the cloud migration processes from traditional on-premise architectures and identify common activities within these processes. Their findings provide crucial insights for this thesis, particularly in understanding the core mechanisms of cloud migration. Pahl and Xiong's work helps to demystify migration processes by breaking them down into manageable, operational-level activities, improving transparency, and adapting them to various business needs.

A key aspect of their research is the development of a system to ease the transition into Cloud Computing environments, with a special focus on assisting Small and Medium Enterprises (SMEs). Despite the benefits of their standardized migration process, they acknowledge the persistence of numerous individual tasks, particularly in architecture migration - the primary focus of this thesis.

Studies by Hazi et al. [16, 19, 8] dive deeper into cloud migration, outlining detailed steps in the migration procedure. Other works have extensively shown why it is difficult to do cloud migration and what the issues are [11, 8, 29, 30]. Most issues fall back on the uncertainty of what each part of a legacy system does and how and why they interact with each other. Building on these frameworks, Balobaid and Debnath [4] provide an overview and comparison of various cloud migration tools. They discuss the importance of cloud migration, its benefits, and associated risks.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Most cloud providers offer proprietary tools tailored for migrating data to their respective clouds, such as Azure's and AWS's data migration tools. However, these tools primarily focus on data transfer, with limited support for migrating applications and processes. Additionally, they often result in vendor lock-in.

Major cloud providers like Amazon [32], Oracle [23, 24], and Microsoft [21] offer guidelines and solutions for cloud migration in their whitepapers. These documents serve as valuable resources for understanding the best practices and methodologies for moving infrastructure to the cloud.

This thesis aims to develop a more generalized, vendor-independent solution for cloud migration. The focus will be on leveraging IaaS and PaaS to facilitate the migration of on-premise machines into containers. This approach seeks to reproduce on-premise environments in the form of maintainable Dockerfiles that can be built into images and started as containers, enabling straightforward deployment to any IaaS or PaaS provider.

In the realm of containerization, technologies like LXC [1] and Docker [2] are pivotal. Their ease of use, coupled with reliability and isolation capabilities [9, 33], make them integral to this thesis's strategy for cloud migration. While container orchestration tools like Kubernetes add another layer of complexity by facilitating the dissection of an on-premise machine and redistributing its components across multiple containers, this aspect will not be considered in the current scope of work. The exploration of container orchestration and the utilization of Kubernetes represent promising avenues for future research and development. Delving into these technologies could further enhance the process of replicating and managing complex systems within cloud environments.

One of the significant challenges in transferring legacy systems to Docker is managing dependencies. These dependencies often conflict with one another, may be outdated, or even unavailable. Horton and Parnin address this issue in their paper on "DockerizeMe" [17], which focuses on Python scripts and their dependencies. Their work is pivotal in creating Dockerfiles that include all necessary dependencies to run a specific Python script. When encountering Python files on a machine, their methodology can be effectively applied to containerize the script, ensuring it is ready for execution.

In addition to dependency management, the generation of configuration management scripts is another crucial aspect, as explored by Hummer and colleagues in their papers [18, 15]. They demonstrate, using Puppet as an example, how to reliably generate these scripts to test for idempotence. While Puppet itself does not create Dockerfiles or images, the theoretical underpinnings of this thesis draw conceptual inspiration from their work. Moreover, the contributions of Fu et al. [10] and Shambaugh et al. [28] in developing tools for Puppet are noteworthy. Their research offers substantial theoretical frameworks for asserting the correctness and validity of Puppet scripts, which are crucial for ensuring the reliability of the configuration management process.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 3

Background

3.1 Operating Systems

An operating system (OS) [6] is an essential intermediary between the user and the computer hardware. Defined by John Daintith and Edmund Wright as "The set of software products that jointly controls the system resources and the processes using these resources on a computer system" [6], an OS simplifies user interaction with complex hardware operations. It manages tasks such as file handling, memory allocation, process control, and operating input/output devices such as keyboards, mice, and printers. The kernel, a central component of the OS, enables these functionalities by directly interacting with the hardware.

3.1.1 Process

A process in operating systems is an execution instance of a program. Unlike a static program, a process introduces dynamic elements such as CPU and memory usage.

The structure of a process includes:

- **Text Section:** Contains the executable code.
- **Data Section:** Space for static and global variables initialized pre-execution.
- **Heap Section:** Used for dynamic memory allocation, e.g., via `malloc`, `free`.
- **Stack Section:** Holds local variables during runtime.

Processes undergo several states:

- **New:** In the phase of creation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **Ready:** Waiting for execution.
- **Running:** Currently executing instructions.
- **Waiting:** Paused for an I/O operation.
- **Terminated:** Execution completed.

Diving deeper into the intricacies of process management, it becomes evident that the transition of a process through various states – from creation to termination – is a dynamic and complex journey. To adeptly manage this journey, the operating system employs an important mechanism known as the Process Control Block (PCB). The PCB is not merely a collection of process-related information; it is the cornerstone of effective process management.

The role of the PCB is indispensable in an environment where multiple processes exist simultaneously. Each process, while moving through states like Ready, Running, or Waiting, must have its particular environment and requirements preserved.

It acts as a comprehensive dossier for each process, holding critical information such as the process state, program counter, CPU registers and memory pointers. This memory section is essential for the operating system to maintain continuity and coherence in process execution, especially during context switches.

In essence, the PCB is the backbone of process management in an operating system. Its detailed record-keeping ensures that despite the intermittent execution and the complexity of multitasking, each process progresses smoothly towards its completion. To achieve this task the PCB contains the following informations in the following order:

- Process State/Pointer: Reflects the current state of the process..
- Process ID and Parent Process ID: Unique identifiers for the process and its parent
- Program Counter: Address of the next instruction to execute.
- CPU Scheduling Information: Includes priority and process niceness.
- Memory Management Information: For instance, page tables.
- Accounting Information: Usage time and limits.
- I/O Information: Details about open files and used devices.

Each process that is being run and has one of those PCB also has an environment, which normally has specific environment variables that are needed for its execution. Those might be some user credentials or just information about how the process should behave in its execution. When thinking back to the process stack, the environment variables find their place above the stack with other command-line arguments. Those environment

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

variables are copied upon the fork of the parent process and can be looked at with the following way.

There are several ways to retrieve environment variables. All of them involve getting the process id of the target process. On Linux this can be via the following command `pidof <name>`. To check for the environment variables of this process, the following can be run: `cat /proc/6110/environ`. This then outputs one line of all environment variables contained in this process. This might not be readable so replacing 0 with new lines will help. `cat /proc/6110/environ | tr '\0' '\n'`

3.2 Package manager

A package manager, in its core functionality, is a software application that helps the user to find and manage other software applications on their operating system. Such functionalities can be to search for an application, download, install, remove or update it. By using a package manager, finding packages and installing them can be done in an easy and fast way since normally Linux distros contain a list of sources where to download software from. For the user, this not only brings the advantage to easily manage to software that is on the OS but also security since there is not being used some arbitrary site to get the software from and the package manager keeps the software up to date.

Packages are normally a binary executable, other dependencies and metadata. This gets bundled into a "package" and has an extension to its name, so the OS gets a hint on how to run it (for example: .rpm, .deb, ...). However dependencies that are also contained in the package manager will not be in the package itself, but rather a requirements list that needs to be available on the system to run the software. The package manager then goes ahead and downloads all the software that is required to install the actual requested software from the user.

An interaction with a package manger, in this example aptitude might begin with "apt update". By doing so the package manager creates a local cache of the metadata (package, version number and description, etc.) on the local system and updates said local cache with the metadata from the repository. When running "apt install" the manager checks in the local cache to retrieve the package from the given address. This package now might have dependencies which have to be met to successfully run the package. If there are any unmet dependencies this is the moment where the package manager download those dependencies and installs them before going forward with the installation of the desired package.

Almost the same happens when a package is removed. Either the package manager deletes the dependencies on its own or informs the user that there are unused packages on the system that can be deleted.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.3 Dependency hell

"Dependency hell" is a common challenge encountered in the development of large, modern software applications, as well as systems installing different software on the same machine. These applications often integrate various pre-existing components and rely on multiple external services and applications. Each component typically has its own set of dependencies, and sometimes these can conflict with the dependencies required by other components.

For example, consider a program that integrates two applications: Application A and Application B. Suppose that application A requires 'libexample' version 1.1, while application B needs 'libexample' version 1.2. If the system cannot support different versions of 'libexample' simultaneously, a conflict arises. As a result, it becomes impossible to run applications A and B together within the same environment. This leads to a scenario where the entire program is unable to function properly due to these conflicting dependencies.

Docker offers a robust solution to the dependency hell problem on a system level. It allows each application to run in its isolated environment, known as a container, with its specific set of dependencies. This isolation ensures that applications with conflicting dependencies can coexist on the same system without interfering with each other. Thus, Docker effectively resolves the conflicts inherent in dependency hell for system packages.

3.4 Strategies for Reproducible System Builds

The primary aim of this thesis is to replicate a system accurately and define the necessary modifications to attain the desired system state. This process is at the heart of Reproducible System Builds, the art of methodically defining system changes to reach a specific state.

Achieving a reproducible system requires meticulous attention to several key elements: version control, dependency management, build tooling, and configuration. This section dives into the various available strategies for achieving repeatable build systems, exploring how these strategies function, their appropriate use cases, and the advantages they offer.

3.4.1 Version Control and Tracking

Important technologies to ensure reproducibility are version control systems like Git or Mercurial. These systems are pivotal in tracking modifications made to source code, configuration files, and build scripts, allowing developers to pinpoint and recreate precise build states. Essential practices in version control for reproducible builds include:

Committing Build-Related Files: It is crucial to maintain version control over all build scripts, configuration files, and other pertinent files. This practice ensures that any changes to these files are systematically tracked, linking them to specific build versions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Tagging Releases: Using tags in version control systems helps to mark specific points in the development timeline that correspond to stable and reproducible builds. Tagging facilitates easy identification and retrieval of these specific build states for future reference or deployment.

3.4.2 Isolation and Virtualization

Isolation and virtualization techniques play a crucial role in creating self-contained and independent environments for reproducible system builds. These techniques ensure the isolation of software dependencies, configurations, and runtime environments, guaranteeing reproducibility across different systems. In this section, the significance of isolation and virtualization techniques, specifically focusing on containerization and virtual machines (VMs) will be discussed.

Containerization, a lightweight virtualization technique, enables the creation of isolated environments known as containers. Containers encapsulate applications and their dependencies, providing a complete and portable runtime environment. Unlike traditional virtualization methods, which involve running multiple operating systems on a single physical machine, containerization allows for running multiple containers on a shared host operating system. Popular containerization technologies include Docker, Podman, and LXC/LXD.

Containerization offers several advantages compared to conventional virtualization methods. Firstly, containers ensure efficient resource utilization since they are lightweight and share the host operating system's kernel. This results in minimal overhead and faster startup times, making containers well-suited for rapid deployment and scaling of applications. Secondly, containers provide isolation at the application level, ensuring that each container has its own isolated environment. This isolation facilitates reproducible builds by avoiding conflicts between software components. Additionally, containers are highly portable as they can be easily transported between different environments.

Container images capture the entire runtime environment, including dependencies and configurations, simplifying the replication of the same environment on various systems. Furthermore, container orchestration platforms like Kubernetes enable the management and scaling of containerized applications across clusters, offering features such as automatic scaling and load balancing. Containerized environments can be hosted in various settings, including local development environments where containers run on developers' local machines, on-premises setups where organizations establish their own container orchestration platforms in data centers, and cloud platforms such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP), which provide managed container services like Amazon Elastic Container Service (ECS), Azure Kubernetes Service (AKS), and Google Kubernetes Engine (GKE). Containers also offer the flexibility to deploy applications across hybrid and multi-cloud environments, taking advantage of different cloud providers or on-premise infrastructure.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In contrast, virtual machines (VMs) provide complete operating system environments that are separate from the host system. VMs run on a hypervisor and emulate the entire hardware infrastructure, including the operating system. This allows for the simultaneous execution of multiple VMs on a single physical computer. Notable virtualization platforms include VMware, KVM, and Hyper-V.

VMs offer distinct characteristics and use cases compared to containers. Firstly, VMs provide complete isolation, enabling the running of different operating systems and configurations on the same physical hardware. This level of isolation is particularly advantageous for working with legacy systems or applications that have strict operating system dependencies. Not only isolation is great for those systems but also with VMs they are easier to maintain and support. VMs also allow for operating system flexibility, supporting the simultaneous use of multiple operating systems on the same host. They are commonly employed in environments that require diverse operating systems or specific configurations.

VMs can be hosted on on-premise infrastructure using software like VMware ESXi or Hyper-V, on cloud platforms such as Amazon EC2, Microsoft Azure Virtual Machines, and Google Compute Engine, and in virtual desktop infrastructure (VDI) setups. VDI allows for central control of user environments and the provision of virtual desktops for remote access, contributing to reproducibility in user desktop setups.

When considering the difference between containerization and VMs, several factors come into play. Containers excel in resource utilization as they are lightweight and share the host operating system's kernel, resulting in lower overhead and faster startup times. They provide sufficient isolation at the application level. VMs offer stronger isolation by running on separate virtual hardware, enabling the use of diverse operating systems and configurations. VMs are suitable for legacy applications and scenarios that require complete isolation.

Both containerization and VMs have their advantages and can be employed in reproducible system builds depending on specific requirements. Containers are well-suited for lightweight and portable deployments, offering efficient resource utilization, application-level isolation, and easy portability. VMs provide stronger isolation, support for diverse operating systems, and are ideal for legacy applications. The choice between containerization and VMs should consider factors such as resource efficiency, isolation needs, compatibility requirements, and performance considerations.

In summary, isolation and virtualization techniques, including containerization and VMs, are crucial for achieving reproducible system builds. These techniques ensure the isolation and consistency of software dependencies, configurations, and runtime environments across different systems. Containerization provides lightweight and portable deployment options, while VMs offer stronger isolation and support for diverse operating systems. The choice depends on specific requirements, balancing factors such as resource efficiency, isolation needs, and compatibility requirements.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

3.4.3 Containerization Technologies

Containerization technologies, such as Docker, have emerged as powerful tools in the field of reproducible system builds. Docker is a widely used and popular containerization platform that provides developers with a robust and user-friendly environment for creating, deploying, and managing containers. In this section, the workings of Docker are explored as a detailed example of containerization technology, and its key features and functionalities are examined.

Docker operates based on the concept of container images, which are lightweight, standalone, and portable units that encapsulate an application, its dependencies, and the necessary runtime environment. A Couple of different technologies are used to make the interaction with those containers as easy as possible. LXC (Linux Containers), a process for virtualization on OS level and create isolate environments for each of those Linux systems on one single host, build the backbone of Docker. LXC does this by using kernel-level name spaces to create a border between the virtual OS's to the host machine. This makes sure that a root user on a guest system has not root rights on the host system. The process name space makes sure that processes are only listed on their system and the network name space creates an isolated network stack. Other name spaces that are being used are the ipc namespace (Inter Process Communication), the mnt name space (Mount) and the uts namespace (Unix Timesharing System). LXC provides Control Groups (cgroups) which make sure that a container gets a limited and specified amount of resources compared to the name spaces which isolate the container. Images are built using Dockerfiles, which are text files containing instructions for assembling the image layer by layer. The Dockerfile specifies the base image, the required software packages, configuration files, and any customizations needed for the application.

Once the Dockerfile is defined, the Docker Engine builds the container image by executing the instructions step-by-step. Each instruction in the Dockerfile corresponds to a layer in the image, allowing for incremental and efficient image builds. This layered approach also facilitates caching and reusing of previously built layers, speeding up subsequent builds and reducing bandwidth usage.

Once the container image is built, it can be instantiated as a container, which is a running instance of the image. Containers are isolated environments that operate within the host operating system's user space, leveraging the host kernel for system calls and resource management. This lightweight virtualization approach ensures that containers have minimal overhead and startup time compared to traditional virtual machines.

Docker empowers containerization with a comprehensive range of features that enhance usability and flexibility. These include image distribution, container lifecycle management, networking and service discovery, volume management, and orchestration with Docker Swarm or Kubernetes.

To facilitate image sharing and distribution, Docker offers Docker Hub, a central repository where developers can publish their custom-built images. This fosters collaboration, enables the sharing of best practices, and facilitates the deployment of pre-built images.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Container lifecycle management is simplified through Docker's commands for creating, starting, stopping, pausing, and deleting containers. These commands ensure efficient management throughout the container's lifecycle, ensuring consistent and reproducible behavior.

Docker's networking capabilities enable seamless communication between containers and external systems. Networking features facilitate easy network setup, routing, and port mapping, promoting connectivity. Service discovery mechanisms simplify the dynamic detection and connection of containers within distributed application architectures.

For data persistence and sharing between containers and the host system, Docker employs volume mounts. Volumes serve as a mechanism for storing and sharing data, ensuring that essential data is retained even when containers are stopped or removed.

Docker Swarm, the native orchestration solution, enables the management and scaling of containerized applications across multiple hosts. It incorporates features such as service discovery, load balancing, and automatic container rescheduling. Additionally, Docker seamlessly integrates with Kubernetes, a powerful container orchestration platform, to leverage advanced capabilities for managing large-scale container deployments.

Docker's extensive ecosystem of tools and integrations further enhances its capabilities. For instance, Docker Compose simplifies the orchestration of multi-container applications, empowering developers to declaratively define and manage complex application stacks.

In conclusion, Docker exemplifies the capabilities of containerization technologies. It leverages container images, Dockerfiles, and a robust set of features to enable developers to create, deploy, and manage self-contained and reproducible environments. The flexibility, efficiency, and portability offered by Docker contribute to the reproducibility of system builds and facilitate the development and deployment of applications across diverse environments.

Dockerfile and Container Image Construction

The Dockerfile is a vital component of Docker's containerization technology, offering a declarative and reproducible method to define the construction of container images. Let's explore the Dockerfile in more detail to understand how it facilitates image building.

Composed as a plain text file, the Dockerfile comprises a series of instructions that guide the image construction process. These instructions are executed sequentially, resulting in the creation of distinct image layers. Each layer represents a specific modification or addition made to the underlying base image.

Every Dockerfile begins with the `FROM` instruction, specifying the base image from which the container image will be built. The base image provides the starting point for the construction process and typically contains a minimal operating system like alpine or a pre-configured environment like a posgres database image. Multiple Dockerfiles can share the same base image, allowing for the reuse and sharing of common configurations.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

After defining the base image, the Dockerfile includes a set of instructions for installing dependencies and configuring the runtime environment of the application. Common instructions include **RUN**, **COPY**, **ADD**, **ENV**, and **WORKDIR**:

1. **RUN** executes commands during the image build process, such as installing software packages, running scripts, or configuring settings. These commands are provided as they would be written into a terminal.

2. **COPY** and **ADD** copy files and directories from the build context (local directory) into the image.

3. **ENV** sets environment variables within the image, allowing for customization and configuration.

4. **WORKDIR** sets the working directory within the image, specifying the location where subsequent commands will be executed.

The Dockerfile also supports other instructions for exposing ports (**EXPOSE**), defining the entry point (**ENTRYPOINT**), specifying the default command (**CMD**), and configuring user permissions (**USER**), among others. These instructions enable fine-grained control over the behavior and configuration of the resulting container image.

The **EXPOSE** command does not have any effect on the resulting image. It's only perpose is to hint the reader that there is a port that can be reached from outside.

CMD as mentioned defines a default command which is executed upon starting the container if nothing else is provided. This default command is not written like an ordinary terminal command. Else it's a list of string token that correspond to the terminal command split up by whitespace. `tail -f /dev/null` will be `CMD ["tail", "-f", "/dev/null/"]`.

It is important to note that Docker employs a layered file system for image construction. Each instruction in the Dockerfile creates a new layer on top of the previous layers, forming a stack of read-only layers. This layered approach allows for efficient caching and incremental rebuilding. If an instruction in the Dockerfile remains the same across builds, Docker can reuse the previously built layer, significantly speeding up subsequent builds. More detailed information are in Section 3.4.3

Once the Dockerfile is defined, the Docker CLI (Command Line Interface) is used to build the container image. The 'docker build' command reads the Dockerfile and executes the instructions, resulting in the creation of a new image. The image is then tagged with a specific version or repository name to facilitate identification and sharing.

The resulting container image is a self-contained unit that encapsulates the application, its dependencies, and the specified runtime environment. It can be shared, distributed, and instantiated as containers on various systems and environments. By defining the container image construction process using the Dockerfile, developers ensure the reproducibility of the image and eliminate inconsistencies that may arise from manual configuration.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In conclusion, the Dockerfile plays a crucial role in the construction of container images. It provides a declarative and reproducible way to define the steps required to build the image, including base image selection, dependency installation, and environment configuration. The layered file system approach, caching mechanisms, and incremental builds contribute to efficient image construction.

Docker Build Process and Image Layers

The Creation of container images is done in the build process. It is responsible for executing the instructions defined in the Dockerfile and this creating the corresponding container image. In this section, the details of the Docker build process are delved into, and the concept of image layers is explored.

The Docker build process starts with the build context, which can be a directory path, a URL or STDIN. The context contains all information that are needed for the `docker build` command to successfully build the image. It always contains the Dockerfile and if needed any additional files or directories that should to be copied into the image during the build process. This is important if the context for example is a URL to a .tar.gz archive. If the context is a local directory path files that should be copied do not necessarily need to be in the context path. If the Dockerfile references it and during the build process the file can be located on disk it does not matter where it is stored.

Once the build context is established, Docker begins parsing the Dockerfile. It reads the instructions sequentially and processes each instruction to perform the necessary actions.

Docker employs a layered file system approach to optimize image construction and minimize duplication. Each instruction in the Dockerfile creates a new layer on top of the previous layers, forming a stack of read-only layers. These layers are lightweight and share common elements, such as the base image or previously built layers.

During the build process, Docker creates intermediate containers for each instruction in the Dockerfile. Each container represents the state of the file system after executing a specific instruction. These intermediate containers are not persisted and are discarded once the instruction is completed. They serve as a means to capture the changes made by each instruction and generate the subsequent layer.

The layered file system utilizes a copy-on-write strategy to ensure efficiency and avoid duplication. When a new layer is created, it only contains the modified or added files and directories compared to the previous layer. Any unchanged files are not duplicated but are referenced from the lower layers. This copy-on-write mechanism optimizes both storage space and build speed.

Once all the instructions in the Dockerfile are executed, Docker generates the final image. The final image consists of a stack of layers, each representing a specific modification or addition to the file system. These layers are stacked in a way that allows Docker to reconstruct the complete file system of the container.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

After the image is built, it can be tagged with a specific version or repository name to facilitate identification and sharing. Tags provide a way to label and reference different versions or variants of the same image.

The resulting container image is a composite of the base image, the modified or added files and directories from the Dockerfile instructions, and any dependencies or configurations defined during the build process. It is important to note that container images are immutable, meaning they cannot be modified once created. Any changes to the image require rebuilding a new image. By leveraging the layered file system, caching, and copy-on-write mechanisms, Docker ensures efficient and reproducible image builds. The layered structure allows for incremental updates, faster rebuilds, and optimized storage utilization. The Docker build process, combined with the Dockerfile, provides developers with a powerful and declarative approach to construct container images with consistency, portability, and reproducibility.

Layered File System

The layered file system is a fundamental concept in containerization, including Docker. It is a technique used to optimize image construction and improve the efficiency of container operations. In a layered file system, each modification or addition to the file system is stored in a separate layer, forming a stack of read-only layers. In this section, the benefits and drawbacks of the layered file system are discussed, along with its application in containerization and the considerations to be mindful of.

Since each layer in the file system represents a specific modification or addition, Docker can leverage layer caching during the build process. When a layer has been built previously and its context remains unchanged, Docker can reuse the existing layer rather than rebuilding it. This caching mechanism significantly speeds up subsequent builds by eliminating redundant steps and reducing the overall build time.

Without this layered file system incremental updates to container images would not be possible. When changes are made to an image, only the affected layer needs to be modified or added, while the unchanged layers remain intact. This allows for faster updates and reduces the amount of data that needs to be transferred or pulled when deploying or distributing images.

Each is layer uniquely identified by its content and position within the stack. This enables versioning and reusability of layers across different images. Layers that are common across multiple images, such as base operating system layers or shared libraries, can be reused, eliminating the need for duplication and improving storage efficiency.

This type of file system provides granular control over image layer management. Each layer can be inspected, analyzed, and manipulated independently, allowing for efficient troubleshooting, debugging, and image customization. Layers can be added, removed, or modified without affecting the integrity of other layers, providing flexibility and agility in image composition.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

By leveraging the copy-on-write mechanisms layered file system can improve container performance. When a container modifies a file, the file is copied to a new layer, ensuring that the original layer remains unchanged. This copy-on-write strategy reduces disk I/O operations and optimizes container runtime performance.

Shared files and directories between layers are stored only once and referenced by multiple layers and thus minimizes duplication of data across different images and containers. This reduces the overall disk space required for storing images and allows for efficient use of storage resources. But this is also a huge drawback. Each layer in a Docker image is stored separately, which can result in increased disk space usage. Over time, as more layers are added during the image build process or when images are updated, the disk space usage can grow significantly.

Layered file systems add complexity to the management and maintenance of Docker images. As the number of layers increases, it becomes more challenging to understand and manage the dependencies and interactions between layers. This complexity can make troubleshooting and debugging more difficult.

Another downside is that performance overhead that comes along with it, especially when accessing or modifying files that are spread across multiple layers. Each layer needs to be traversed to locate or modify a file, which can slow down operations. Additionally, if multiple containers are running simultaneously and sharing layers, there can be contention for resources, leading to performance degradation.

Once layers are added to a Docker image, it can be challenging to remove or modify intermediate layers. This can impact image maintenance and make it harder to achieve efficient image size management.

Security risks is another aspect that has to be considered. If a layer contains vulnerabilities or malicious content, it can impact the overall security of the image. Additionally, as layers are shared between images, any security vulnerabilities in shared layers can affect multiple images, potentially leading to a broader security impact.

To mitigate these drawbacks, it is important to follow best practices when building and managing Docker images. This includes minimizing the number of layers, regularly cleaning up unused layers, using efficient image build techniques, and regularly updating base images to ensure security patches are applied. Additionally, using tools and strategies for image optimization, such as multi-stage builds, can help reduce the impact of layered file systems on disk space usage and performance. Despite these drawbacks, the benefits of layered file systems in Docker, such as efficient disk space utilization, faster image builds, caching, and version control, make them a valuable component of Docker's architecture. By following best practices and optimizing image builds, the impact of the drawbacks can be minimized while leveraging the advantages of layered file systems.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Benefits of Containerization for Reproducible Builds

In summary of the last chapters containerization hase a couple of advateges that shine in reproducable builds.

1. Dependency Management: Containers encapsulate applications along with their dependencies, eliminating dependency conflicts and ensuring consistent build environments across different systems.
2. Isolation: Containers provide isolation from the underlying host system, reducing variations caused by differences in operating systems, libraries, and configurations.
3. Portability: Containers are portable units that can be deployed on different systems, ensuring consistent builds regardless of the underlying infrastructure.
4. Version Control: Container images can be versioned, tagged, and stored in registries, enabling precise control over the versions used in builds and ensuring reproducibility.
5. DevOps Integration: Containers seamlessly integrate with DevOps practices, enabling continuous integration, continuous deployment, and automated testing, which further enhance reproducibility.

3.4.4 Configuration Management Tools

Configuration management tools play a crucial role in achieving reproducibility in system builds. They enable developers to define and manage the configuration settings and parameters of the software components used in the build process. These tools automate the process of configuration management, reducing errors, improving efficiency, and ensuring consistency. The following chapters will explore different popular configuration management tools and it's benefits as well as drawbacks.

Ansible

Ansible is an open-source configuration management tool that focuses on simplicity and ease of use. It follows an agentless architecture, meaning that it does not require installing any additional software on the target systems. Key features of Ansible that contribute to reproducible system builds include: Infrastructure as Code (IaC), Idempotence, Version Control Integration

Ansible allows developers to define the desired state of the infrastructure and software components using code. This code, referred to as Ansible Playbooks, provides a clear and reproducible description of the system's configuration requirements. Playbooks are written in YAML, which is human-readable and easily understandable by both developers and operations teams.
Furthermore Ansible ensures idempotence, meaning that running the same configuration repeatedly produces the same result. This guarantees that the system is consistently

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

configured regardless of the number of times the configuration is applied. Idempotence reduces the risk of unintended changes during system builds and promotes reproducibility. Ansible Playbooks can be stored in version control systems such as Git. This enables versioning, collaboration, and traceability, ensuring that specific versions of configurations are used for reproducible builds. Developers can track changes, revert to previous versions, and review the evolution of configurations over time.

Chef

Chef is an open source configuration management tool that allows DevOps to configure, setup and manage automations. These automations can include anything as long as it can be described in the form of code. Chef Infra allows DevOps to specify how, where and when infrastructure should be deployed. What is the setup of the infrastructure. Where does it operate on. Is it on-premise or is it deployed into the cloud. How are infrastructures connected. How is the network structured. Everything can be configured and is written into so-called recipe. These contain every information needed to setup the infrastructure. Recipes are written in the programming language Ruby and are collected in a so-called cookbook. Cookbooks provide the structure for a variety of different services, resources, and infrastructure recipes and organizes them.

Chef Workstation is a suite of Chef tools for DevOps to administer and configure infrastructure. It includes ChefSpec, Chef InSpec, Cookstyle and Test Kitchen. These tools help properly writing recipes for the infrastructure to make sure it does what it is intended to do.

These recipes are then uploaded to the Chef Infra Server. This server manages all the cookbooks with its recipes. Not only manages them, it also is responsible for applying policies, rolling out configurations and communication with nodes.

Infrastructure created by recipes is deployed to nodes. These nodes are computing resources such as virtual machines, containers or servers. All of these nodes are managed by the Chef Infra server and need to have Chef Infra Client running on them. The Client checks in and compares its infrastructure with the infrastructure described in the latest cookbooks. If they are not equal, then the client updates its infrastructure to reflect the state defined in the cookbook.

Considerations for Configuration Management Tools

Configuration management tools offer numerous advantages that contribute to the achievement of reproducible system builds.

One key benefit is the aspect of consistency. Configuration management tools ensure that configurations remain consistent across different systems and environments, reducing variations and improving reproducibility. By defining configurations as code and applying them consistently, researchers and scientists can avoid discrepancies that may affect the integrity of their experiments or analyses, thereby maintaining a reliable build process.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Automation is another crucial advantage provided by configuration management tools. These tools automate the process of configuration management, reducing the occurrence of manual errors and ensuring the consistent application of configurations. By automating repetitive tasks involved in system configuration, scientists can save time, effort, and resources. Furthermore, the risk of human error is minimized, leading to more accurate and reproducible system builds.

Version control integration is a vital feature offered by configuration management tools. By leveraging version control systems, configurations can be effectively versioned, tracked, and rolled back if necessary. This ensures reproducibility and traceability, allowing researchers to precisely identify the configuration state used in a particular experiment or analysis. Moreover, version control facilitates collaboration among team members, enabling them to review changes and maintain a comprehensive history of configuration states for auditing and troubleshooting purposes.

The scalability aspect of configuration management tools is particularly advantageous for scientific endeavors. These tools provide efficient mechanisms for managing configurations across a large number of systems. By defining configurations as code and leveraging automation, scientists can easily scale their system builds to accommodate the growing infrastructure required for their research. This scalability feature allows for consistent and reproducible configurations across a diverse set of environments.

However, it is important to consider certain factors when utilizing configuration management tools for reproducible system builds. One such factor is the learning curve associated with these tools. New users may need to invest time in learning the features, best practices, and conventions of the configuration management tools they choose to use. Adequate training and documentation are crucial to maximize the benefits and ensure successful adoption, ultimately enhancing the reproducibility of system builds.

Security considerations should also be taken into account when employing configuration management tools. These tools should adhere to security best practices to protect sensitive configuration data and prevent unauthorized access. Encryption of sensitive data and secure access control mechanisms for configuration repositories are important measures for maintaining the integrity and confidentiality of configurations within scientific workflows.

To further ensure reproducibility, rigorous testing and validation processes should be implemented. Automated testing frameworks can be utilized to verify the desired state of the system, validate configuration changes, and identify any discrepancies or errors. Thorough testing and validation contribute to the accuracy, completeness, and reproducibility of configurations within scientific system builds.

In summary, configuration management tools such as Ansible and Chef offer powerful capabilities for achieving reproducible system builds in scientific contexts. These tools provide automation, version control integration, consistency, and scalability, enabling scientists and researchers to define and manage configurations in a reliable and reproducible manner. Proper consideration of learning curves, security measures, and testing

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

practices is essential for the successful utilization of configuration management tools in the pursuit of reproducible system builds within scientific endeavors.

3.4.5 Containerization Technologies vs Configuration Management Tools

Containerization technologies and configuration management tools have emerged as indispensable solutions for achieving reproducible builds in both scientific and industrial software environments. Containerization technologies, such as Docker, provide isolated and portable environments encapsulated within containers, enabling consistent and reproducible execution of software applications. Configuration management tools, such as Ansible and Puppet, automate the management of system configurations, ensuring consistent and traceable deployments. Both approaches offer distinct advantages and considerations for reproducible builds. Containerization technologies excel in providing lightweight and encapsulated environments, facilitating the replication of complex software dependencies. Configuration management tools excel in automating the provisioning and management of system configurations, allowing for versioning and maintaining consistency across multiple environments. Because of the overhead, system setup and steep learning curve that configuration management tools have with longer times between iterations containerization technologies are a better fit for prototyping and thus chosen in this thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 4

Approach

4.1 Theoretical Approach

This section introduces a model of a system state and action that can be executed on the system. The model and state transition definitions are later on used for implementing a working prototype.

4.1.1 System Model

Table 4.1 describes each element of our model and the used symbols. Note that $\mathcal{P}$ denotes the *powerset* of a given set. The notation $x[j]$, x_j is used to refer to the jth item of a tuple x.

A task T consist of multiple state transition with dependencies D_T between them, assuming the total ordering $\forall st_i, st_j \in T : (st_i \neq st_j) \iff ((st_i, st_j) \in D_T) \oplus ((st_j, st_i) \in D_T)$.
The same applies to a process P. It consists of multiple tasks with dependencies D_P between them, with a total ordering defined as $\forall t_i, t_j \in T : (t_i \neq t_j) \iff ((t_i, t_j) \in D_T) \oplus ((t_j, t_i) \in D_P)$. A process is usually only executed once, and so is each task in it. In contrast, state transitions within a task are usually executed multiple times.

4.1.2 State Actions

Operations applied on a system state can be categorized into two primary types. The first type are system analysis operations, which are dedicated to verifying the existence of specific properties or values within the state, such as files and their content. The second type involves state transition operations, which actively apply modifications to the current system state, thereby leading to the formation of a new state as a consequence of these operations. This section explains the operations that are needed.

Symbol	Definition
K, V	Set of possible **state property keys** (K) and **values** (V).
$d : K \to \mathcal{P}(V)$	**Domain** of possible values for a given state property key.
$A := K \times V$	Possible **property assignments**.
$S = \{\sigma_1, \sigma_2, \ldots, \sigma_i\}$	Set of possible system states.
$\sigma_i \subseteq [K \to V]$	The state is defined by (a subset of) the state properties and their values.
σ_t	Target system state.
$T = \{st_1, st_2, \ldots, st_n\}$	A **Task** consists of a set of state transitions
$st : (S \times T) \to S'$	**State transition** of each task. Pre-state S maps to post-state S' after applying all state transitions.
$p : T \to I$	Set of **input parameters** (denoted by set I) for a task.
$P = \{t_1, t_2, \ldots, t_m\}$	A **Process** consists of a set of Tasks
$D_T \subseteq \mathcal{P}(T \times T)$	**State transition dependency** relationship: state transition st_1 must be executed before state transition st_2 if $(st_1, st_2) \in D_T$
$D_P \subseteq \mathcal{P}(P \times P)$	**Task dependency** relationship: task t_1 must be executed before task t_2 if $(t_1, t_2) \in D_P$
$miss, add : (S \times K) \to V$	Compares the state property values of a key k between system state S with target state σ_t and returns a set of missing values in S and non-existing values in σ_t.

Table 4.1: System Model

State Analysis

The following actions are predicates used to obtain information about the current system state.

- $\langle$is_installed$(pm, pn), \sigma\rangle$
  Returns true if the package named pn is installed using package manager pm.

- $\langle$file_exist$(p), \sigma\rangle$
  Returns true if a file at path p exists.

- $\langle$file_is_same$(p, d), \sigma\rangle$
  Returns true if a file at path p contains content identical to d.

- $\langle$environment_variable_has_value$(n, v), \sigma\rangle$
  Returns true if the environment variable n is set to the value v.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- $\langle \text{process_runs}(n, p), \sigma \rangle$
  Returns true if a process with the name n is running and was started with parameters p.

State Transitions

The following state transitions are essential for the reconstruction of a target system state.

- $\langle \text{install_package}(pm, pn), \sigma \rangle$
  Installs a package with the name pn using the package manager pm, where $pm \in \{\text{apt}, \text{apk}, \text{npm}, \text{pip}\}$.

- $\langle \text{file_change}(p, d), \sigma \rangle$
  Changes, modifies, or deletes the file at the given file path p. If the path does not exist, all parent directories are created. d represents the content of the file as a byte array. If d is null, the file is deleted.

- $\langle \text{change_working_directory}(p), \sigma \rangle$
  Changes the current working directory to the path provided by parameter p.

- $\langle \text{set_environment_variable}(n, v), \sigma \rangle$
  Sets the environment variable n to the value v.

- $\langle \text{start_process}(n, p), \sigma \rangle$
  Starts the process named n and passes parameters p to the program.

Operational Semantics

The operational semantics for the state transitions are defined as follows:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

$$\text{(INSTALL-PACKAGE)} \quad \frac{\neg\langle \text{is_installed}(pm, pn), \sigma\rangle}{\langle \text{install_package}(pm, pn), \sigma\rangle \rightarrow \sigma_{pac} \cup \{pn\}}$$

$$\text{(CREATE-FILE)} \quad \frac{\neg\langle \text{file_exists}(p), \sigma\rangle}{\langle \text{file_change}(p, d), \sigma\rangle \rightarrow \sigma_{fs}[p \mapsto d]}$$

$$\text{(CHANGE-FILE)} \quad \frac{\langle \text{file_exists}(p), \sigma\rangle \wedge \neg \text{file_is_same}(p, d, \sigma)}{\langle \text{file_change}(p, d), \sigma\rangle \rightarrow \sigma_{fs}[p \mapsto d]}$$

$$\text{(DELETE-FILE)} \quad \frac{\langle \text{file_exists}(p), \sigma\rangle \wedge d = \text{null}}{\langle \text{file_change}(p, d), \sigma\rangle \rightarrow \sigma_{fs} \setminus \{p\}}$$

$$\text{(CHANGE-DIR)} \quad \frac{\sigma_{cwd} \neq p}{\langle \text{change_working_directory}(p), \sigma\rangle \rightarrow \sigma_{cwd \mapsto p}}$$

$$\text{(SET-ENV)} \quad \frac{\neg \text{environment_variable_has_value}(n, \sigma)}{\langle \text{set_environment_variable}(n, v), \sigma\rangle \rightarrow \sigma_{env}[n \mapsto v]}$$

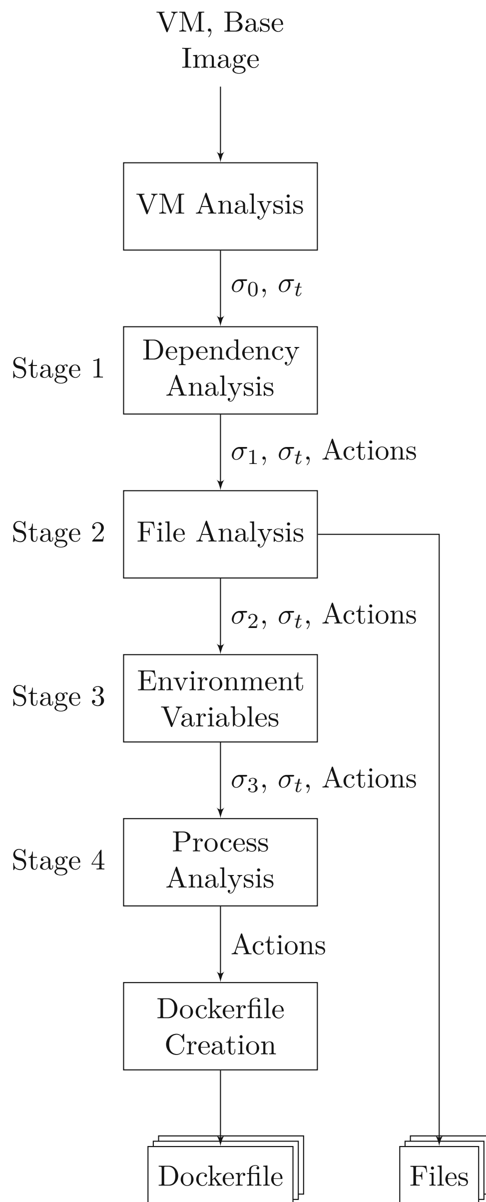
$$\text{(START-PROCESS)} \quad \frac{\neg \text{process_runs}(n, p, \sigma)}{\langle \text{start_process}(n, p), \sigma\rangle \rightarrow \sigma_{proc} \cup (n, p)}$$

Where:

σ	Current system state.
σ_{pac}	Set of all installed packages.
$\sigma_{pac} \cup \{pn\}$	Current state with package pn installed.
σ_{fs}	Files system of current state.
$\sigma_{fs}[p \mapsto d]$	Updated state file system writing data d to path p.
$\sigma_{fs} \setminus \{p\}$	State with removed file at path p.
σ_{cwd}	Current working directory in the state.
$\sigma_{cwd \mapsto p}$	Current working directory set to path p.
σ_{env}	Mapping of environment variables of current state.
$\sigma_{env}[n \mapsto v]$	Sets environment variable n to value v.
σ_{proc}	Set of running processes of current state.
$\sigma_{proc} \cup (n, p)$	State with process n and parameters p running.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.1.3 Explanation



The analysis initiates by converting the VM snapshot and base image into two key states: the target state σ_t and an initial empty state σ_0.

Stage 1 (Dependency Analysis) uses σ_t and σ_0 to resolve dependency differences, resulting in needed actions and state σ_1, setting the stage for further analysis.

Stage 2 (File System Analysis) aims to align file system variances. Actions from this stage, along with Stage 1 outcomes, are combined and passed on. Altered files will be written to disk separately for Dockerfile creation.

Stage 3 (Environment Variable Augmentation) uses the evolved state σ_2 to add missing environment variables, refining the system to state σ_3. Recording of all actions and states continues through this stage.

Stage 4 (Process Management) ensures alignment of processes, aiming for congruence between the target state and the current state σ_3. After this stage, σ_3 and σ_t should be the same and only logged actions are passed on.

The last step takes the recorded actions and saves them as a Dockerfile.

Figure 4.1: Flow graph depicting stage transitions from inputs *Virtual Machine (VM)* and a provided *Base Image* all the way to the resulting *Dockerfile*

For a better understanding, the following example illustrates the system model applied to a concrete case. The goal (in this example) is to recreate the target state, which is a representation of the image built by the Dockerfile as listed in 1. It builds an Ubuntu system which has python installed, a new directory with a new file created and one

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
FROM ubuntu
RUN apt install python
RUN mkdir /app
WORKDIR /app
COPY main.py .
ENV DEMO_PORT=999
CMD python main.py
```

Listing 1: System Model Dockerfile example

additional process running. As mentioned in 4.1.2 State Actions, these transitions are needed to recreate the system from an empty Ubuntu installation. Each system state $\sigma \in S$ consists of a number of system properties, that have been defined as key-value pairs. The system possesses several key properties, namely: $K = \{files, dependencies, environment_variables, running_processes\}$. Each of these properties plays a crucial role in describing and understanding the system's state. The *files* property provides an essential snapshot of the system's current state. It captures all file names and their corresponding content that have been altered since the initial state. By analysing this property, we can understand the specific modifications made within the system and how they have shaped its current configuration.

Another crucial aspect to consider is the *dependencies* property. It contains comprehensive information about the packages that have been added to the system. Changes in the file system are not the only important information we need to consider, but also what the current system configuration for running a new process would look like. The *environment_variables* property stores all the additional environment variables that have been introduced into the system. Lastly, the *running_processes* property sheds light on the dynamic nature of the system. It encapsulates details about the processes that have been initiated since σ_0. The values of the system properties are at the initial system, state σ_0 all empty since nothing has yet changed. All these keys are needed to rebuild an empty initial system state to be the same as an already predefined target system state σ_t.

Figure 4.1 illustrates the necessary steps to transform a base image and document the operations required to attain the target system state. Initially, a snapshot of the VM is captured and analysed, leading to its conversion into the target state σ_t. The base image is designated to serve as σ_0.

All the mentioned keys are needed to achieve the desired system state σ_t, which is derived from the initial empty system state σ_0. This process involves three defined tasks: resolving dependencies, recreating the file system, and starting the missing processes. Table 4.2 provides an overview of the resulting process, including its associated tasks and state transitions.

In the following, we illustrate this process based on a concrete example. The initial state

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

σ_0 has the following properties, with their descriptions provided in YAML format in Listing 2.:

```
initial state σ0:
    files: []
    dependencies: []
    environment_variables: []
    running_processes: []
```

Listing 2: Properties of initial state σ_0

The target state for this example can be represented as shown in Listing 3.

```
target state σt:
    files:
    - name: /app/main.py
      content: <content>
    dependencies:
    - package_manager: apt
      package: python
    environment_variable:
    - name: DEMO_PORT
      value: 999
    running_processes:
    - command: python main /app/main.py
      environment:
      - name: DEMO_PORT
        value: 999
```

Listing 3: Properties of target state σ_t

The defined process begins with the task of installing all missing dependencies. To accomplish this, the differences between the current state (initial state σ_0) and the target state σ_t must be evaluated. Specifically, a function has been implemented to calculate the difference between two states based on a given property key. The values represented by *miss* indicate elements that need to be added to the current system to match the target system, while *add* represents values present in the current system that are not part of the target system and need to be removed.

For each task, the corresponding differences must be evaluated to determine the appropriate state transitions. The first task involves equalizing the dependencies, as they significantly impact the file system. Upon comparing the initial state σ_0 to the target state σ_t for the dependency property key, the *miss* and *add* values are identified as

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

follows: $miss = \{(apt, python)\}, add = \{\}$. Since the *add*, values are empty, only the python dependency needs to be installed using the apt package manager. Therefore, the first task t_1 will consist of a single state transition st_1 named *install_package* with the input parameters *apt* and *python*. Applying the task t_1 to the initial state σ_0 results in the new state σ_1, defined in Listing 4.

```
intermediate state σ1:
    files: []
    dependencies:
    - package_manager: apt
      package: python
    environment_variable: []
    running_processes: []
```

Listing 4: Properties of intermediate state σ_1

At this stage checking the difference between state σ_1 and σ_t for key *dependencies* will return empty *miss* and *add* values which means that those system properties are already the same and therefore this task was successful, and the next one can be analysed and created.

Having completed the task of equalizing dependencies between the target and the current state, the next step involves aligning the file systems of both states. By examining the *miss* and *add* values for the *file* property key, it is observed that *add* is empty and *miss* is $miss = \{(/app/main.py, < content >)\}$ indicating that only one file needs to be modified. Consequently, the next task t_2 will only include one state transition st_1 named *file_change* with the parameters */app/main.py* and the file's content. Applying task t_1 to state σ_1 yields a new system state σ_2 with the properties in Listing 5.

```
intermediate state σ2:
    files:
    - content: /app/main.py
      data: <content>
    dependencies:
    - package_manager: apt
      package: python
    environment_variable: []
    running_processes: []
```

Listing 5: Properties of intermediate state σ_2

The next task ensures that all missing files are added to the state. In this example, the file main.py in the /app directory needs to be included.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

As same as in the step before when checking for differences after task execution, in this case t_1, σ_2, σ_t for the property key $file$ will result in two empty $miss$ and add which concludes to two same properties.

Next, the focus shifts to equalizing the processes and environment variables. Just like in the previous steps, a task named t_3 is defined to tackle this objective. To determine the necessary actions, the differences between the required properties must be evaluated. For the state σ_2 and the key $environment_variables$, the add values are empty, while $miss$ holds a single value: $miss = \{(DEMO_PORT, 999)\}$. The $miss$ of the $environment_variable$ property will be called $miss_{env}$ further on.
Similarly, when examining the differences between the same states but for the property key $running_processes$, add remains empty, and $miss$ again holds a single value: $miss = (python\ main.py, /app/main.py, \{(DEMO_PORT, 999)\})$. From this point onward, the $miss$ of the $running_processes$ property will be denoted as $miss_{proc}$.

$miss_{proc}$ encompasses all the required information for restarting the same process. It includes the command used, the working directory in which the command was executed, and the environment variables set at the time of command execution. Armed with this information, the final task t_3 can be defined. Firstly, the missing environment variables $miss_{env}$ align with the environment variables in $miss_{proc}$, indicating that the environment variables only need to be set once right before starting the only process.

Now, let us move on to starting the processes. To achieve this, the working directory must be changed, followed by initiating the process within that directory using the correct command. Consequently, the task t_3 consists of three state transitions. The first transition, st_1, involves the $change_working_directory$ state transition with the input parameter $/app$. The subsequent state transition, st_2, is the $set_environment_variable$ state transition with the input parameters $DEMO_PORT$ and 999. Finally, the last state transition, st_3, is the $start_process$ state transition with the input parameters $python$ and $main.py$.

Applying task t_3 to the state σ_2 yields the new state σ_3, the result state will look like Listing 6:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
intermediate state σ3:
    files:
    - name: /app/main.py
      content: <content>
    dependencies:
    - package_manager: apt
      package: python
    environment_variable:
    - name: DEMO_PORT
      value: 999
    running_processes:
    - command: python main /app/main.py
      environment:
      - name: DEMO_PORT
        value: 999
```

Listing 6: Properties of intermediate state σ_3

Upon comparing all the property keys for states σ_3 and σ_t, it is evident that all have empty *miss* and *add* values. This indicates that σ_3 is identical to σ_t, thereby confirming that the target state has been successfully achieved.

The sequence in which tasks are created and executed plays a crucial role in minimizing the number of state transitions required to recreate the target state σ_t. For instance, prioritizing the equalization of file systems before installing missing dependencies can significantly reduce the number of additional *file_change* state transitions needed to modify the files that would otherwise be handled by the package manager during dependency installation.

The order of execution not only plays a critical role in minimizing the number of state transitions required, but also ensures the effectiveness of each individual state transition. This is demonstrated in the given example, where a python file is first copied and then executed. If the *start_process* state transition were to be executed prior to the *change_file* transition, it would fail due to the absence of the source file necessary for the process. Therefore, careful sequencing is essential to achieving the desired effects, as it guarantees that the requirements for every state change are satisfied.

In addition to the aforementioned state transitions, such as *start_process* and *change_file*, other transitions can be incorporated to optimize the process and reduce the overall number of required state transitions. For instance, introducing more abstracted state transition actions like *clone_from_git* or *download_from* provides alternative means of adding missing files to the system. These additions exemplify the potential for further optimization and enhancement of the system recreation process.

By carefully designing and arranging the sequence of state transitions, the efficiency of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

#	Task/State transition	Parameter
t1	resolve dependencies	
└─st1	└─install_package	apt, python
t2	recreate file system	
└─st1	└─file_change	/app/main.py, <content>
t3	start missing processes	
└─st1	└─change_working_directory	/app
└─st2	└─set_environment_variable	DEMO_PORT, 999
└─st3	└─start_process	python, main.py

Table 4.2: Defined Tasks & State Transitions

system state recreation can be refined while considering alternative transitions to meet specific requirements and further reduce the total number of necessary state transitions.

4.2 Implementation

Initially, a tool is required to simulate the source system, which will be used for generating the configuration. As discussed in the background section, there are three primary options available. Among these options, namely Ansible, Chef, and Docker, Docker was identified as the most fitting choice due to its similarity to a bare-metal server. Furthermore, Docker eliminates the need for redundant configuration management, since a single configuration suffices. Additionally, Docker images can seamlessly integrate with both Ansible and Chef in case there is a requirement for configuration management. This positions Docker as the ideal choice for mirroring the source system. Its setup process is straightforward and involves fewer elements.

The tool devised to analyse the source system and mirror it in a different environment is built using Python. Python's ability for quick prototyping, strong Docker compatibility, and adaptability in diverse deployment scenarios influenced this choice. Although C++ and Rust were potential contenders due to their seamless integration with the OS and Docker, they weren't selected. C++ was ruled out because of a familiarity gap. While Rust has its merits, it was superseded by Python's rapid prototyping, which bypasses the compilation step. Moreover, Python simplifies the task of compiling for various target platforms in multiple settings.

4.2.1 Workflow

Initially, an instance of the source image is launched, succeeded by an instance of the base image derived from the source. The source container is triggered only if it remains inactive. Exclusively read operations are executed on the source container, negating the necessity for mutual exclusion, transactions, or multiple containers. However, this

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

doesn't hold for the target container. Initially empty, the target container is gradually synchronized to reflect the source container. Hence, each iteration of Dockerfile generation spawns a fresh target container.

The generation tool accepts the source and target image names upon initiation. An optional flag detailing copy statement processing is also entertained. By default, each file copy is uniquely represented as a copy command in the resulting Dockerfile. If the universal copy command flag is activated, data replication employs the lone `COPY data .` command.

While Docker containers primarily rely on Linux OS containers, the tool is honed for this specific genre. In Linux, every data action translates to file operations. Ideally, copying every altered file from the source to the target container, followed by process execution, would mirror the source container's behaviour. Yet, this could inflate the Dockerfile with redundant `\COPY` commands. Filesystem modifications mainly stem from three sources:

1. **Direct File Addition:** This arises when files are directly pushed into the container via an explicit `\COPY` command in the source Dockerfile. Tracing the origins of such files proves challenging.
2. **Execution of RUN Commands:** The `\RUN` command facilitates command execution within the Docker container. Often, these commands are stealthy. Yet, when triggering other software, some operations might get recorded, like package manager activities. For a command resembling `RUN apt install git`, the Ubuntu package manager will record this act while installing the 'git' package.
3. **Processes Causing File Alterations:** Commands such as `\EXEC` or `\ENTRYPOINT` can initiate processes that produce or modify files. Log files typify such modifications.

Tracing each file's origin remains an intricate task. Yet, by acknowledging files possibly introduced via package managers, an approach crystallizes: querying these managers in both containers. By installing missing and purging surplus packages in the target container, file copying significantly reduces. Given the diversity of package managers, this technique must be replicated for each. Since evolving Dockerfiles primarily employ Ubuntu images intertwined with Python or Node.js projects, inquiries target 'apt', 'pip', and 'npm'.

Distinguishing between directly copied files and those spawned by processes is intricate. Therefore, files not associated with a package manager are presumed to be directly copied. In the rare event these files result from a process, this approach guarantees source-target container consistency. Yet, this assumption could present complications in subsequent phases.

After considering package manager operations, all source container files are evaluated for deviations compared to the base image, and these differences are matched against

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

the target container. Missing files in the target container but present in the source are relocated to the target via the \COPY command. Unaltered files in the target container, if modified in the source, are also migrated. Reducing copied files could entail overwriting unchanged target files with their altered source versions and noting the differences. However, crafting the bash command to reconcile the target file with its source counterpart is challenging. Adding to this, such a method might obfuscate the process, as interpreting an overwritten YAML configuration file is simpler than parsing a multi-line bash command, showcased in Listing 7.

```
RUN echo "datasource:" >> config.yaml && \
    echo "    url:127.0.0.1:6543" && \
    echo "    username:admin" >> config.yaml && \
    echo "address" >> config.yaml  && \
    echo "    street:Hauptstrasse" >> config.yaml
```

Listing 7: Illustrative Dockerfile modification command

Thus, modified source files that remain static in the target container are copied over. After filesystem alignment, the following steps encompass identifying active processes, their initiation sources, and related environment variables. To identify active processes, both container process tables are compared. Processes absent in the target container are launched subsequently. But prior, the process ID fetches the working directory and environment variables.

4.2.2 Implementation Details

While scrutinizing the source Docker container, only data accessible within the container is considered. This mirrors real-world scenarios where actual systems don't continuously log Docker metadata.

Spotting modified or new files in the source versus the target container is straightforward. If a file exists in the source but not in the target, it's an addition and should migrate to the target. When a file is in both containers, its source creation and modification timestamps are checked. If different, the source file replaces the target file. In sizable filesystems, like those with 10,000 files, this process can be protracted.

While real-world system analyses might lack Docker metadata, for prototyping and efficiency, we've integrated it. It's pivotal to realize Docker meticulously logs file alterations during image builds, readable from the container's metadata. Empirical evaluations affirm that while working without Docker metadata is plausible, leveraging it drastically enhances speed. For a thorough and prompt analysis, Docker metadata delivers a granular report on file changes. Nevertheless, a significant limitation is Docker's exclusive recording of build-time modifications; runtime alterations are unnoticed.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

4.3 Limitations

Generating a Dockerfile from an existing machine poses challenges, particularly concerning the resulting file's readability and accuracy. Not all steps delineated in a Dockerfile lead to discernible changes when recreated, leading to discrepancies between the original and the generated file. The main limitations of this method include:

- Inability to detect multi-stage builds
- Ambiguity in data origins
- Incapacity to identify volumes
- Indistinctness between ENTRYPOINT and CMD commands
- Some RUN commands leave no discernible traces
- Base image remains undetected
- Reliance on snapshot-based evaluation

4.3.1 Multi-stage Build

Analyzing a container crafted via a multi-stage build provides insight only into the final build stage. Typically, data from preceding stages are transferred to the current stage using a rudimentary copy command. As the data is replicated within the container, its original creation process remains obscured, making it apparent only that data was introduced to the container. Consequently, the generated Dockerfile will predominantly reflect the last build stage.

4.3.2 Data Origin

When examining the file system for new or modified files, we can extract and replicate them. Yet, the initial source of these files remains elusive. Thus, in the Dockerfile, while `COPY` commands will replicate the appropriate file, the true source from which the file was copied is indiscernible.

Moreover, it's uncertain from the resultant Dockerfile whether a file was copied individually or as part of a broader directory in the source Dockerfile. For instance, `COPY source_data /app/target_data` transfers all contents from the host's `source_data` directory to the container's `target_data` directory. But the new Dockerfile can't deduce if all items within `source_data` were copied collectively or if select files originated elsewhere—like a `config.json` file from another directory.

Consider the example in Listing 8 which illustrates an original Dockerfile. It replicates the local `demo` directory to the container's `/app/demo` and also copies a `local.yaml` config from the `configs` directory to the container's `demo` directory. However, the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

resultant Dockerfile shown in Listing 9 clearly diverges from the source. Notably, the generated version has a much larger command set, resulting from uncertainties about data origins.

For bigger projects, this issue exacerbates. For instance, a multi-stage Angular project might involve thousands of `COPY` commands solely for the `node_modules` directory. One remedy is a bulk copy command such as `COPY data /`, which simplifies the Dockerfile but sacrifices some clarity on file origins. Though improved in readability, the resultant Dockerfile lacks explicit detail on the individual files. Fortunately, examining the specified source directory or a generated log file can provide a clearer file breakdown.

Additionally, this ambiguity extends to environment variables. When sourced from `/proc/{process_id}/environ`, the origin of environment variables is uncertain. They might be custom additions during process initiation or from a copied file, making the generated Dockerfile possibly differ from the source.

```
FROM Ubuntu
COPY demo /app/demo
COPY configs/local.yaml /app/demo
```

Listing 8: Source Dockerfile Copy Example

```
FROM Ubuntu
COPY data/demo/main.py /app/demo/main.py
COPY data/demo/utils.py /app/demo/utils.py
COPY data/demo/assets/favicon.png /app/demo/assets/favicon.png
COPY data/demo/assets/favicon.png /app/demo/assets/favicon.png
COPY data/demo/assets/background.png /app/demo/assets/background.png
COPY data/demo/static/index.html /app/demo/static/index.html
COPY data/demo/static/user.html /app/demo/static/user.html
COPY data/local.yaml /app/demo/local.yaml
```

Listing 9: Created Dockerfile from Copy Example

4.3.3 Metadata

There are different commands that add metadata to docker images. As the name suggests that are commands that do not have any influence in the resulting image but adds information. Since metadata are not reflected inside a container, no information can be gained from inside and thus are not present in the resulting Dockerfile.

MAINTAINER was such a command, it set the Author field in the metadata of the resulting image, but has been deprecated and have been replaced with the generic key-value pair **LABEL** command.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.3.4 Variables

Dockerfile has the **ARG** command which allows arguments to be passed in during building and stores them in a variable. This variable is used in the Dockerfile instead, and where ever it is used late, the passed-on value is substituted. All the dynamic elements at building have the same problem that they are not in the Dockerfile present. Using Listing 10 as an example when building the image with `docker build --build-arg DIRNAME=/app` the dummy.txt file will be copied into /app. During analysis, only the result is visible, which is that the dummy.txt will have the path /app/dummy.txt. Therefore, the Dockerfile in Listing 11 will be created. The ARG command will not be present in the resulting Dockerfile and will be simply removed.

```
FROM Ubuntu
ARG DIRNAME
COPY dummy.txt $DIRNAME
```

Listing 10: Source Dockerfile Copy Example

```
FROM Ubuntu
COPY data/dummy.txt /app/dummy.txt
```

Listing 11: Created Dockerfile Arg Example

4.3.5 Volumes

Dockerfiles do not contain information of volumes. Volumes are mounted when running a container. That is where the problem lies. When analyzing a container, it is looked upon which files exist in the container that do not exist in the Base Image. Since data from mounted volumes are part of the file system, they are detected when checking for files. These files are copied out of the container and added as a copy command to the resulting Dockerfile. The Dockerfile will contain copy statements for all the files that were present on the mounted volume while scanning. This can be prevented by not mounting volumes during the scan, if possible, or excluding files with a certain path prefix from copying.

4.3.6 Expose

The docker file command **EXPOSE** describes which ports should be opened and reachable from the outside, but **EXPOSE** does not have any influence in the actual behavior and does not alter the anything inside the docker container. It is purely documentation to users for easier and better understanding of the created docker container. The problem with detecting open ports in containers is deciding whether or not EXPOSE should be added to the Dockerfile. It might be that **EXPOSE** was not part of the source Dockerfile, then adding it adds an inaccurate line to the Dockerfile and decreases the accuracy. Not adding **EXPOSE** is wrong as well, since the Dockerfile could have contained

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the **EXPOSE** command, and that would mean another line difference in the resulting Dockerfile.

4.3.7 Running Processes

Another big limitations is that executed **CMD** and **ENTRYPOINT** commands can be only detected if the container is analyzed while these commands are still in execution. To detect what might have been the **CMD** or **ENTRYPOINT** command, the process table is queried, to see which processes are currently running and how they have been started. If a process with a very short runtime it might happen that analyzing installed packages beforehand takes longer than the executed command processes, and when checking for processes it might not be running anymore which will cause that no **CMD** or **ENTRYPOINT** entry in the Dockerfile. There are methods that keep track of running processes and process history, but they require altering the source image by adding further dependencies.

CMD vs ENTRYPOINT

CMD and **ENTRYPOINT** have a very similar behavior. Both describe the command that gets executed when running the container, but with the difference that **ENTRYPOINT** defines the executable and **CMD** defines the default executable with parameters. **ENTRYPOINT** is used when additional arguments are provided when running the container for the predefined executable. When querying the process list, the command with its parameters is listed, it can not be distinguished where the parameters came from. It could be from a **CMD** command with parameters or an **ENTRYPOINT** command where the parameters were provided during starting the container. Since there is no identification of which command was used, the resulting Dockerfile will contain the **CMD** command with all parameters.

4.3.8 Command Executions during build time

The limitation with the greatest impact on the correct lines is the **RUN** command. The **RUN** command executes commands during build time inside the docker container. **RUN** has the limitation that it does not leave traces of the command inside the container. Since **RUN** commands executed during build time, they are not logged in the history file or other files. The only thing that is present is the result of the RUN command. There is the next problem, it's not clear which command contributed to which change in the container.

Fetching external data

wget, curl or git clone fetch data from external sources during building the image, which has the effect that many additional files are present in the container. Since RUN commands are not stored inside the container, there is no hint that certain files may not have been copied into the container via the **COPY** command or were fetched during

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

building. Therefore, it is assumed that the files were copied in via the `COPY` command. This leads to huge discrepancies between target and source Dockerfile.

Packages

Globally installed packages can be figured out quickly, the problem is, that to install packages the package manager need to know the remote address to it. When creating a new image, it might be that no new packages can be installed because the remote addresses are outdated and, therefore, not reachable anymore. In this case, the package manager needs to update the local cache first. This update can cause several new and changed files that can have different content compared to when the new image is built with the newest information the package manager needs.

Dealing with packages that are not installed globally is harder. One issue is that it cannot be said whether the package was copied into the container or if it was installed inside the container via a package manager. It strongly depends on which languages were used for the software that should be run.
Node.js projects are a great example for this case. It is easy to install packages since they are listed in the package.json file, but it is hard to know where the package installed via npm or maybe yarn or was not it installed at all, and it is part of a multi-stage build and just copied into the container.
Python does not have the issue where packages might be copied into the container, but has issues regarding installing them. Installing packages is mostly done in two different ways, either installing them all explicitly with a pip install command or by installing them reading from a requirements.txt file. Python does not have a lock file compared to Node.js which means that python package versions might not be the same when installing them by using the requirements.txt file in the source image and the build image from the generated Dockerfile. To prevent this, instead of installing python packages via the requirements.txt file, all packages are installed explicitly with their version.
Issues with package managers mostly occurred for scripting languages since for compiled languages like Java, rust, ... only the compiled software is copied into the container. Building is done in a previous stage of a multi-stage Dockerfile.

4.3.9 Base Image

Base images defined in Dockerfiles have the problem that in the container, there is no where mentioned what base image was used for creation. This means that when analyzing all data, all changes that were caused by the base image except OS image will be added to the Dockerfile. With an unknown base image, the resulting Dockerfile will be very differently from the source Dockerfile. This can be prevented to some extent via using the same base image from the source container for the target container.

Even knowing the base image can still cause a lot of discrepancies. This goes hand in hand with the above-mentioned limitation about package managers. When the base image was compiled significantly earlier, such that when installing new packages, these

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
...
COPY setup.sh setup.sh
RUN setup.sh
COPY . .
...
```

Listing 12: COPY Order Example

```
...
COPY data/app/fencing.py  /app/fencing.py
COPY data/app/location.py  /app/location.py
COPY data/app/main.py  /app/main.py
COPY data/app/setup.sh  /app/setup.sh
...
```

Listing 13: COPY Generated Order Example

cannot be found and the package manager has to update its cache, then this will lead to a lot of unnecessary and wrong COPY commands in the Dockerfile. This can be prevented by building the base image with the source image or in a closely timed manner.

4.3.10 Order of Execution

When analyzing a container's file system, it cannot be known in which order commands were written in the Dockerfile. The example in Listing 12 shows part of a Dockerfile where a setup script is first copied into the container and then later on different files. As mentioned above, RUN commands mostly can not be detected, but also the order of the COPY commands can not be traced back. It is only known that certain files need to be copied into the container, but not when. Therefore, all files are copied at the same time, which might look like Listing 13.

There is no precise order of commands, but the generated Dockerfile will the following structure.

1. Installation of missing system packages
2. Installation of further missing packages
3. Coping missing files
4. Set environment variables
5. Changing working directory
6. Process run command

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.3.11 Snapshot based evaluation

The Dockerfile generator analyzes the file system and remembers what files have changed. Any file that is changed beyond that point will not be registered and be ignored. Therefore, the generated Dockerfile might not result in a container that is exactly the same as the source container. Furthermore, the snapshot is taken after the source container had time to run for several seconds. If in this time, changes are done to the file system inside the Docker container, the state of the container's file system does not represent the of the file system of the image. This can cause differences in built images to be the generated Dockerfile.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

Evaluation

5.1 Methodology

For evaluation, the dataset used as ground truth are Dockerfiles from existing projects on GitHub as well as Dockerfiles created manually that represent the target state.

The dataset contains several Dockerfiles with several different key aspects.

1. using COPY and CMD commands
2. RUN apt install <Package, ...>
3. different base images
4. different programming languages and executables (python, JavaScript)
5. minimize RUN commands like wget and git clone
6. minimize multi-stage builds

5.1.1 Evaluation Subjects

For the evaluation, ten Dockerfiles were utilized to generate source images. The created images underwent analysis, which formed the basis for reconstructing new Dockerfiles. The newly created Dockerfiles were then assessed of their performance and functionality.

The selection of source Dockerfiles was diverse, containing six from various open-source projects, three that were custom-created for specific test scenarios, and one for a Postgres image. The open-source projects were chosen based on their suitability on possibility of occurences on real machines. Dockerfiles featuring multistage builds, the use of volumes, and dynamic data loading during the build process were excluded from the selection.

Name	Reason	Reference
Custom 1	tests file creation modification deletion	Github [27]
Custom 2	tests dynamic file changes occurred during run time	Github [27]
Custom 3	tests package managers	Github [27]
Traefik	common application proxy	Github [31]
Nginx	common application hosting web server and proxy	Github [22]
Ansible	testing python	Github [3]
React webpage	common technology for website	Github [5]
Django backend	common backend technology	Github [5]
Postgres	common database, testing data changed during build and when running	Github [7]
OpenVPN	common vpn sever and client	Github[20]

Table 5.1: Evaluation Subjects

The custom Dockerfiles, based on Debian images, incorporated a range of functionalities, including the installation of different software packages, file manipulation, and the execution of tasks to simulate edge case scenarios. These scenarios involved operations like running data generators and file deletion, emulating complex, real-world conditions.

5.1.2 Evaluation Procedure

1. Extract base image from source Dockerfile.
2. analyse the source Dockerfiles base image.
3. If base image container runs shorter than Dockerfile generation task.
   3.1. Create a Dockerfile for the new temporary base image.
   3.2. Set base image of new temporary base image to source Dockerfiles base image.
   3.3. Prevent container termination before analysing finished.
   3.4. Replace the base image in the source Dockerfile with the new base image.
4. Build source image fresh from scratch.
5. Run the Dockerfile generator.
6. Post-processing of generated Dockerfile.
7. Evaluate the generated Dockerfile

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Extracting the base image from the source Dockerfile is straightforward. Most of the time, it is identified in the first line, defined by the FROM command. For container analysis, the source container is consistently compared to an empty container that shares the same base image. However, maintaining a container with only the base image running throughout the duration of the analysis is essential but not always straightforward. Take the official Python image as an example. Executing `docker run python` in the terminal results in an empty line. Since there's nothing to run inside the container, it shuts down immediately. Consequently, it will be listed as exited with status 0 when inspecting all containers. Fortunately, the analysis will not be affected if the container exits early. To prevent the early exit of the container, it is necessary to initiate a process that keeps the container active. Various solutions exist for this challenge, but the simplest is creating a new base image for analysis, derived from the source image's base image. The choice of the base image influences the pre-installed utilities and packages that can be leveraged. One universally applicable method is to include the command `CMD ["tail","-f","/dev/null"]`. This command effectively prevents the container from exiting prematurely. However, a downside of this approach is that the CMD command may appear in the generated Dockerfile, necessitating manual removal.

Before every analysis, it is ideal to construct both the source and the new base image from scratch. This ensures that if package managers like apt or apk are required, both the source container and the temporary container used for analysis are in the same state, ready to install new packages. Otherwise, there is a risk of encountering errors during package installation due to outdated mirrors. In such situations, the package manager must update its cache, leading to modifications in numerous files. If cache updating occurs only in the temporary target container, these altered files are marked as differences from the source container, resulting in their manual transfer into the container.

Once the base image for analysis and the source image are prepared, the tool is then utilized to generate a Dockerfile.

The Dockerfile generated may include the command implemented to keep the base image container running, which, as previously mentioned, requires manual removal. The tool does not automatically remove this command, as doing so could lead to unintended deletions. It is impossible to determine solely from container analysis whether the command was added merely to prevent premature container termination. Furthermore, upon creation of the Dockerfile, the tool lacks the capability to identify the base image used. Consequently, a manual substitution of the base image is required afterward.

Then, the Dockerfiles can be compared. Comparing the number of lines between the original Dockerfile and the one generated through container analysis can reveal significant discrepancies. When assessing line-by-line similarities, where the nth line in the original Dockerfile is compared to the nth line in the generated Dockerfile, the difference is nearly 100%. Typically, only the first line, which specifies the base image, remains consistent. Manually setting of the base image is necessary for initiating the comparison with an empty container. Subsequent lines invariably differ. Numerous subtle variations can alter a line while preserving its original intent.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The COPY command often varies as the analysis tool assigns its preferred naming conventions to directories. For the RUN command, there can be a variation in the formatting of installed packages, the original Dockerfile might employ a multiline format, whereas the generated Dockerfile typically uses a single line. Furthermore, the order in which the packages are listed may also differ. This holds for any command. The order of arguments is not guaranteed to be the same.

Additionally, the order of commands in the Dockerfile might have been rearranged. A strict line-by-line comparison often fails to provide meaningful insights into Dockerfile content. Instead of merely assessing equality, analysing the impact and implications of each command offers greater informative value. Due to inherent limitations and necessary abstractions, certain commands might be absent in the resulting Dockerfile, such as RUN wget commands. These are not visible from within the Docker container, leading to the generation of COPY statements for each downloaded file.

In contrast, there are instances where the generated Dockerfile contains commands that were not found in the original source Dockerfile. For instance, ENV commands are encountered more frequently in the generated Dockerfile compared to the source Dockerfile.

Measurements

To evaluate the similarity between Dockerfiles, the analysis initially focuses on identifying if a line from the original Dockerfile or a variation of it exists in the generated Dockerfile in short *line diff*. When generic copy statements are used additionally, we evaluate the number of differences in files that are changed. This measurement is called *file diff*. When such a correspondence is established, that particular part is deemed equivalent. In cases where no analogous line in the generated Dockerfile matches the original, a more in-depth investigation is necessary. This process involves two critical aspects:

1. Determining whether the function of the original line has been fulfilled by a previously matching command in the generated Dockerfile.
2. Assessing whether a combination of commands in the generated Dockerfile cumulatively achieves the same purpose as the specific line in the original Dockerfile.

Should both aspects match all Docker instructions, then the Dockerfile is seen as similar.

The generated Dockerfile often merges or groups several commands from the original Dockerfile into a single command. This is particularly evident with COPY commands. In such cases, the generated Dockerfile typically replaces multiple COPY commands with one generalized COPY statement. An example is when the original Dockerfile includes multiple copy commands for various files at different stages. Conversely, the generated Dockerfile might incorporate all these actions into a single generalized COPY command, thus reducing the number of individual copy statements. Instead of copying

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

files at different stages, it copies all of them in the first stage. However, the opposite can also be true. Several commands in the generated Dockerfile might achieve the same result as a single, more precise command in the source Dockerfile. Consider again the COPY command: if the original Dockerfile contained one COPY command for a directory followed by a RUN command that deletes a file, the tool cannot replicate this and will instead copy files individually, resulting in more COPY commands.

5.1.3 Evaluation Procedure Example

Listings 14, 16, 15 and 17 show an example of such an evaluation procedure. Listing 14 is an example Dockerfile for a small flask python project. It uses an Ubuntu image as a base. The first command updates the package manager cache and installs python3. It then copies the requirements.txt into the image and installs the necessary dependencies. Subsequently, all remaining files are copied into the container, and the server is started.

Ubuntu does not work as a base image since, as a target container, there is nothing inside that keeps the container running, and therefore it will immediately exit, and no analysis can be done. To prevent this, a different base image must be chosen. In this case, the image base_image 15 is created and then used. The base_image itself has Ubuntu as its base image. To ensure that the container instance does not close immediately, lines 6 and 7 add the `tail -f /dev/null` command so that the container runs during evaluation. The source Dockerfile in Listing 14 updates its package managers to ensure that the cached mirrors are still valid. This command can cause a lot of file changes and thus results in several additional lines of COPY commands. Those copy commands have nothing to do with the essence of the container, which is the flask server. To make it easier to analyse the generated Dockerfile later, the update statement is also moved into the base_image, ensuring that the basis for the analysing container can easily install packages.

Following the creation of the new base image, the source Dockerfile must be adjusted. The base image was changed from Ubuntu to base_image 15, and the package manager update command was removed. By removing it, the base_image container instance ensures that it has the same cached package manager data as the source container. These modifications will result in the Dockerfile in Listing 16.

After running the analysis, the resulting Dockerfile will look something like Listing 17 (although the order of lines may vary). It is immediately clear that the generated Dockerfile is not a valid Dockerfile because no base image is declared. This is because just by analysing the base_image container instance, it is not clear what the base image actually is. In this case, it is referred to as base_image, which is similar to Ubuntu but not identical. That is why the base image has to be set manually. While Line 4 aligns with the Dockerfile used for analysis in Listing 16, it deviates from the original Dockerfile shown in Listing 14. Therefore, to maintain consistency with the original, the update command `apt update -y`, which was present in the original, must be reintroduced into the generated Dockerfile.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1  FROM Ubuntu
2
3  RUN apt update -y && apt install -y python3 python3-pip
4
5  WORKDIR /app
6
7  COPY requirements.txt requirements.txt
8  RUN pip3 install -r requirements.txt
9
10 COPY . .
11
12 CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

Listing 14: Source Dockerfile Evaluation Example

In the source Dockerfile, on line 5, the working directory is changed to the 'app' directory, facilitating the installation of packages. However, the generated Dockerfile postpones this working directory change until later in the script, since package installation does not occur directly from the requirements.txt file. It becomes necessary just before executing the Flask server. On lines 7 and 8 of the source Dockerfile, the requirements.txt file is copied, enabling the installation of all necessary dependencies from it. In contrast, the generated Dockerfile takes a different approach. It reads the Python packages already installed in the source container and directly installs them using the `pip install` command. As a result, the requirements.txt file is disregarded. While this approach enhances precision by reducing the number of code lines required, it simultaneously diminishes readability and complicates comprehensibility.

Line 10 from the source Dockerfile is almost the same as line 7 in the generated Dockerfile. The generated Dockerfile stores all files inside the data folder, therefore the data source is different, but their result will be the same. Therefore, the precession stays the same in the Dockerfile, but there might be hidden files copied that cannot be seen with this copy command. The running process is not analysed and is mostly ignored, but it might have an effect on the file system. For example, in this case, the flask server is logging into a file called log.txt. During the analysis phase of the file system, all files that are not the same in the source and target containers are extracted. Therefore, the log.txt file is also extracted. The `COPY data .` command in the generated Dockerfile will then copy the log.txt file. It guarantees that all the data from the source container are in the target container later on but log.txt would not be needed to copy since it is overwritten when the flask server starts.

5.1.4 Comparison tool

To facilitate comparative analysis and gain insights into the result of other tools with Docker, the Docker images generated by the Dockerfiles will be compared to a tool called

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
FROM Ubuntu

RUN apt update -y
RUN apt install -y bash

ENTRYPOINT ["tail"]
CMD ["-f","/dev/null"]
```

Listing 15: Base Image Dockerfile Evaluation Example

```
FROM base_image

RUN apt install -y python3 python3-pip

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .

CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

Listing 16: Source Dockerfile Evaluation Example

```
# insert base
# FROM <base_image>

RUN apt install -y python3 python3-pip
RUN pip install Flask==2.2.2 click==8.1.3 itsdangerous==2.1.2 ...

COPY data .

WORKDIR /app
CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

Listing 17: Generated Dockerfile Evaluation Example

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Whaler [25]. According to its developers and contributors, Whaler is described as follows: "Whaler is a Go program designed to reverse engineer Docker images into the Dockerfile that created them." It is important to emphasize that Whaler is specifically designed to operate on Docker images and does not function with running containers or actual virtual machines (VMs). During the reverse engineering process, Whaler examines the layers, file system differences, and other metadata that are part of an image, and it reverses these elements to reconstruct the original Dockerfile. Notably, Whaler does not take into account any changes in data that may have occurred during the operation of a container. Its focus is solely on the metadata associated with a Docker container created during the build process. Whaler is used in this context as a reference point to assess the capabilities of the tool developed in this thesis. It is important to note that while Whaler is designed for Docker, the tool in this thesis leverages Docker as a means of creating simple and efficient "machines", along with a method for preserving changes to facilitate the recreation of these machines. This approach is versatile and can also be applied to other tools such as Puppet, Chef, Ansible, Vagrant, and others for managing and creating machines.

Whaler vs Our Approach

The output of Whaler's Dockerfile generation depends strongly on the base image selected. If the base image is Debian, the resulting Dockerfile closely resembles the original Dockerfile. However, if a different base image is used, everything done by that base image is included in the resulting Dockerfile.

In the case of the example Dockerfile mentioned earlier 16, the output appears as shown in Listing 18. Note that there is no "FROM base_image" statement in the output because Docker metadata does not contain this information. Instead, it is resolved and substituted. Lines 21 and 22 originate from the "base_image," which itself is based on Ubuntu (lines 1 to 20). These lines represent approximately, 2570 lines of files that constitute the Ubuntu image.

Switching from Ubuntu-based images to Debian-based images results in significant differences, as evident in Listing 19. The initial section that involves copying all the files is omitted.

This underscores the crucial role of the base image. Depending on the chosen base image, the resulting Dockerfile can vary substantially. For a fair comparison, lines part of the base image in Whaler's reverse-engineered Dockerfiles are excluded from accuracy calculations.

Additionally, as mentioned earlier, Whaler does not consider files that change during container execution. For instance, the generic copy command in the Dockerfile generated by the thesis tool (line 7, 17) includes not only requirements.txt, app.py, and Dockerfile but also log.txt, which was generated while the server was running. For a comparison of the Dockerfile output that's tolerable but would result in an unacceptable docker file since the behavior would not be the same anymore.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
ARG LAUNCHPAD_BUILD_ARCH
LABEL org.opencontainers.image.ref.name=ubuntu
LABEL org.opencontainers.image.version=22.04
ADD file:c646150c866c8b5ecbc79....fa22502bdba3d38c53fc9a9 in /bin
        boot/
        dev/
        etc/
        etc/.pwd.lock
        etc/adduser.conf
        etc/alternatives/
        etc/alternatives/README
        etc/alternatives/awk
        etc/alternatives/nawk
        etc/alternatives/pager
        etc/alternatives/rmt
        etc/alternatives/which
        etc/apt/
        etc/apt/apt.conf.d/
        .... <2550 files creating ubuntu file system>
CMD ["/bin/bash"]
RUN apt update -y
RUN apt install -y bash
RUN apt install -y python3 python3-pip
WORKDIR /app
COPY file:bfdeeaa41d38c2c11....e34a27924811da in requirements.txt
        app/
        app/requirements.txt

RUN pip3 install -r requirements.txt
COPY dir:5cc3660f227c81ed884d58....ea74b259d854960be85ca61b1 in .
        app/
        app/app.py
        app/dockerfile

CMD ["python3" "-m" "flask" "run" "--host=0.0.0.0"]
```

Listing 18: Generated Dockerfile by Whaler for Evaluation Example

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1  CMD ["bash"]
2  RUN apt update -y
3  RUN apt install -y bash
4  RUN apt install -y python3 python3-pip
5  WORKDIR /app
6  COPY file:bfdeeaa41d38c2c11....e34a27924811da in requirements.txt
7          app/
8          app/requirements.txt
9
10 RUN pip3 install -r requirements.txt
11 COPY dir:5cc3660f227c81ed884d58....ea74b259d854960be85ca61b1 in .
12         app/
13         app/app.py
14         app/dockerfile
15
16 CMD ["python3" "-m" "flask" "run" "--host=0.0.0.0"]
```

Listing 19: Generated Dockerfile by Whaler for Evaluation Example

	Thesis			Whaler	
	line diff	files diff	similar	line diff	similar
custom 1	0	0	✓	0	✓
custom 2	1	1	✓	0	✓
custom 3	1	693	✓	7	✗
postgres	4	174	✓	1	✗
traefik	3	5	✓	2	✗
nginx	3	6	✓	3	✓
fastapi-react backend	2	12	✓	4	✗
fastapi-react frontend	4	3	✓	4	✓
openvpn	2	10	✓	0	✓
ansible	5	32	✓	1	✓

Table 5.2: Result comparison of our approach vs Whaler

5.1.5 Findings

Table 5.2 presents a the results obtained of the tools against those obtained using Whaler. The table is structured to provide insights into the two metrics defined above: line differences, and behavioral similarity between the original and the reconstructed Dockerfiles. For Dockerfiles generated by our approach, it shows the additional files that needed to be copied into the container because they could not be reconstructed based on

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

commands available in the metadata alone.

- **Line Difference (line diff):** Quantifies the number of lines that differ between the original Dockerfile and the reconstructed one.
- **File Difference (files diff):** Reflects the count of files that required relocation after the movement described in the Dockerfile.
- **Behavioral Similarity (similar):** Indicated by symbols (✓for similarity and ✗for dissimilarity), showing whether the behavior of the reconstructed Dockerfile aligns with the original source.

The evaluation reveals that 'Custom 1' and 'Custom 2' are remarkably consistent in both our approach and Whalers, demonstrating minimal differences in lines and files. These images show little change during runtime, which explains the similarity in outcomes from both tools. However, 'Custom 3' presents a stark contrast, as it was specifically designed to test file changes during runtime. This difference is highlighted by the substantial number of files that had to be loaded additionally into the container, which Whaler fails to detect, leading to a reconstructed image with altered behavior. Additionally, Whaler couldn't detect an extra process started by 'Custom 3'.

For applications like Traefik, Nginx, FastAPI-react frontend, and OpenVPN, the evaluation shows minimal line differences between the methods. The configurations of these applications are largely static, with the exception of logs and SSL renewal, thus showing little change during runtime.

In contrast, the 'FastAPI-react backend' and 'Postgres' differ considerably. In these cases, data generated or saved is not preserved by Whaler, potentially leading to significant data loss upon running the new image, hence not maintaining behavioral similarity.

Furthermore, the execution of Ansible led to the creation of some logs and an internal state, resulting in additional files. These files mainly improve Ansible's operational speed and are not crucial for replicating the same behavior.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

Conclusion

In this thesis, the focus was on formalizing the state, actions, and transformation functions necessary for evolving an empty system state into a desired target state with the goal of migrating legacy server snapshots into reproducible Dockerfiles used for cloud deployments. The process involved a detailed analysis and recording of all actions undertaken during the transformation. These recorded actions then served as the foundation for the creation of Dockerfiles.

After background research and evaluation of possible approaches the next phase involved developing a theoretical model that articulated the relationship between the system states and the actions required to transition from an initial, empty state to a specific target state. This model provids a structured approach to understand and replicate the processes in a system.

Due to practical constraints in sourcing VM snapshots for testing, Dockerfiles were used as an alternative, as they simultaneously function as ground truths for our evaluation. In the practical application phase, Docker images created from these Dockerfiles were analyzed. The objective was to reconstruct the original Dockerfiles, thereby testing the effectiveness of the theoretical model in real-world scenarios.

The tool developed for this purpose showed promising results in installing missing packages, tracing file changes, and starting missing processes. However, it also revealed areas for improvement, particularly in the detection of base images as well as detecting file downloads.

In conclusion, this thesis not only introduces a novel approach to Dockerfile generation from existing legacy systems, but also sets the stage for further advancements in this field. The insights gained offer a pathway to simplify the containerization process, making it more accessible and efficient. Future research can build on these findings, enhancing the tool's capabilities for more accurate and user-friendly Dockerfile generation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.1 Future Work

We envision future developments in this field in the following areas:

- The development of a tool to identify the optimal base image for a source container could significantly streamline the process of recreating containers.
- Implementing a mechanism to detect file downloads, particularly from sources like git and via wget/curl, with the aim of reducing unnecessary file copying.
- Separation into multiple Dockerfiles.

6.1.1 Base Image Detection

Currently, determining the base image from within a container is challenging due to the lack of relevant information. There is a need for a tool that can accurately infer the best-suited base image for a source container, ideally one that requires minimal changes for an accurate recreation of the original container. Such a tool would not only simplify the Dockerfile creation process, but also enhance its accuracy and specificity.

6.1.2 Tracing Data Downloads

A significant advancement would involve tracing file downloads, particularly those initiated via git, wget, or curl. Although finding git directories is relatively straightforward, challenges arise when repositories are inaccessible due to privacy settings or deletion. Additionally, detecting wget and curl downloads is complex due to the lack of clear indicators such as the .git folder in the git repositories. Potential solutions might include examining shell history or identifying downloaded but unremoved tar files, which could indicate extraction operations that were part of the Dockerfile.

6.1.3 Identify multiple possible containers

A potential area for future investigation involves analyzing more complex systems that encompass a variety of services such as databases, backends, and frontends. The objective would be to detect and isolate these components and generate separate Dockerfiles for each. Ideally, this process would also include the creation of a Docker Compose file that seamlessly integrates these components, ensuring they function cohesively as they did previously.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Figures

4.1 Flow graph depicting stage transitions from inputs *Virtual Machine (VM)* and a provided *Base Image* all the way to the resulting *Dockerfile* 25

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

4.1 System Model 22
4.2 Defined Tasks & State Transitions 31

5.1 Evaluation Subjects 42
5.2 Result comparison of our approach vs Whaler 50

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Listings

1 System Model Dockerfile example . . . 26
2 Properties of initial state σ_0 . . . 27
3 Properties of target state σ_t . . . 27
4 Properties of intermediate state σ_1 . . . 28
5 Properties of intermediate state σ_2 . . . 28
6 Properties of intermediate state σ_3 . . . 30
7 Illustrative Dockerfile modification command . . . 33
8 Source Dockerfile Copy Example . . . 35
9 Created Dockerfile from Copy Example . . . 35
10 Source Dockerfile Copy Example . . . 36
11 Created Dockerfile Arg Example . . . 36
12 COPY Order Example . . . 39
13 COPY Generated Order Example . . . 39
14 Source Dockerfile Evaluation Example . . . 46
15 Base Image Dockerfile Evaluation Example . . . 47
16 Source Dockerfile Evaluation Example . . . 47
17 Generated Dockerfile Evaluation Example . . . 47
18 Generated Dockerfile by Whaler for Evaluation Example . . . 49
19 Generated Dockerfile by Whaler for Evaluation Example . . . 50

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[1] Container and virtualization tools, lxc documentation.

[2] Dockerfile documentation.

[3] Ansible. Ansible/ansible: Ansible is a radically simple it automation platform that makes your applications and systems easier to deploy and maintain. automate everything from code deployment to network configuration to cloud management, in a language that approaches plain english, using ssh, with no agents to install on remote systems. https://docs.ansible.com.

[4] Awatef Balobaid and Debatosh Debnath. Cloud migration tools: Overview and comparison. In Alvin Yang, Siva Kantamneni, Ying Li, Awel Dico, Xiangang Chen, Rajesh Subramanyan, and Liang-Jie Zhang, editors, *Services – SERVICES 2018*, pages 93–106, Cham, 2018. Springer International Publishing.

[5] Buuntu. Buuntu/fastapi-react: Cookiecutter template for fastapi + react projects. using postgresql, sqlalchemy, and docker.

[6] John Daintith and Edmund Wright. *operating system*. Oxford University Press, 2008.

[7] Docker-Library. Docker-library/postgres: Docker official image packaging for postgres.

[8] Mahdi Fahmideh, Farhad Daneshgar, Fethi Rabhi, and Ghassan Beydoun. A generic cloud migration process model. *European Journal of Information Systems*, 28(3):233–255, 2019.

[9] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. volume 00, pages 171–172, 03 2015.

[10] Weili Fu, Roly Perera, Paul Anderson, and James Cheney. mupuppet: A declarative subset of the puppet configuration language. In *European Conference on Object-Oriented Programming*, 2016.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[11] Mahdi Fahmideh Gholami, Farhad Daneshgar, Graham Low, and Ghassan Beydoun. Cloud migration process—a survey, evaluation framework, and open challenges. *Journal of Systems and Software*, 120:31–69, 2016.

[12] Synergy Research Group. Covid-19 boosts cloud service spending by $1.5 billion in the third quarter: Synergy research group, Dec 2020.

[13] Synergy Research Group. Covid-19 helps to accelerate the shift to hosted and cloud collaboration solutions: Synergy research group, Sep 2020.

[14] Synergy Research Group. Huge cloud market still growing at 34% per year; Amazon, Microsoft & Google Now account for 65% of the total: Synergy research group, Apr 2022.

[15] O. Hanappi, W. Hummer, and S. Dustdar. Asserting reliable convergence for configuration management scripts. In E. Visser and Y. Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2016.

[16] Hassan Hazi, Arash Hassanzadeh, and Alireza Moeini. A comprehensive framework for cloud computing migration using meta-synthesis approach. *Journal of Systems and Software*, 128:87–105, 2017.

[17] Eric Horton and Chris Parnin. Dockerizeme: automatic inference of environment dependencies for python code snippets. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 328–338. IEEE / ACM, 2019.

[18] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Testing idempotence for infrastructure as code. In *14th International Middleware Conference (Middleware)*, pages 368–388, Beijing, China, December 2013.

[19] J. Hwang, Y. Huang, M. Vukovic, and N. Anerousis. Automation and orchestration framework for large-scale enterprise cloud migration. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM 2015)*, pages 1002–1007. IBM Journal of Research and Development, 2015.

[20] Kylemanna. Kylemanna/docker-openvpn: openvpn server in a docker container complete with an easyrsa pki ca.

[21] Microsoft. Server and cloud platform, 2013.

[22] Nginx-Proxy. Nginx-proxy/nginx-proxy: Automated nginx proxy for docker containers using docker-gen.

[23] Oracle. Migrate non-oracle databases and applications to oracle database 12c, 2013.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[24] Oracle. Oracle cloud infrastructure - application migration documentation, 2022.

[25] P3GLEG. P3gleg/whaler: Program to reverse docker images into dockerfiles.

[26] Claus Pahl, Huanhuan Xiong, and Ray Walshe. A comparison of on-premise to cloud migration approaches. In Kung-Kiu Lau, Winfried Lamersdorf, and Ernesto Pimentel, editors, *Service-Oriented and Cloud Computing*, pages 212–226, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[27] Schlizohr. Schlizohr/learning-software-infrastructure-configurations.

[28] Rian Shambaugh, Aaron Weiss, and Arjun Guha. Rehearsal: a configuration verification tool for puppet. *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[29] Mohammed Shuaib, Abdus Samad, Shadab Alam, and Shams Tabrez Siddiqui. Why adopting cloud is still a challenge?—a review on issues and challenges for cloud migration in organizations. In Yu-Chen Hu, Shailesh Tiwari, Krishn K. Mishra, and Munesh C. Trivedi, editors, *Ambient Communications and Computer Systems*, pages 387–399, Singapore, 2019. Springer Singapore.

[30] Kewei Sun and Ying Li. Effort estimation in cloud migration process. In *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, pages 84–91, 2013.

[31] Traefik. Traefik/traefik: The cloud native application proxy.

[32] Jeffrey Varia. Migrating your existing applications to the aws cloud: A phase-driven approach to cloud migration. AWS Cloud Computing Whitepapers, 2010.

[33] Miguel Xavier, Marcelo Neves, and Cesar De Rose. A performance comparison of container-based virtualization systems for mapreduce clusters. pages 299–306, 02 2014.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Bibliothek Your knowledge hub